The JBDS (Java Based Deep Space) Simulator: A new approach

Daniela Bruno

A dissertation submitted to the University of Dublin,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

2006

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: ____________________
Daniela Bruno
September 11, 2006

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Signed: ____________________

Bruno Daniela

September 11, 2006

Summary

Deep space network communication proposes new and challenging issues compared to normal wired and wireless networks.

The problems related to space links are mostly due to the link delay (because of the distances involved) and error rates that affect communication and make classical and widely deployed protocols (for example the Transmission Control Protocol) much less efficient in such an environment.

Network simulation can be extremely useful to evaluate protocol performance, but since deep space links are affected by long delays and high error rates, it is necessary, in order to obtain realistic evaluations, to try to re-create those problems.

Research in the field led to the implementation of simulators and emulators able to reproduce the environmental conditions that characterize communication in a deep space environment. These efforts and their deficiencies are described in this thesis, and, in addition, a new approach is proposed – the JDBS simulator.

The JBDS simulator is a framework able to simulate data communication in a deep space network environment. JDBS was implemented using the Java Programming language with low level features provided via the Java Native Interface (necessary in order to use raw sockets).

Communication between network nodes is implemented using Java sockets. The results obtained to date are affected by implementation issues that remain to be solved. Future work is proposed in order to overcome those problems that could lead to the development of a fully-featured Java deep space network simulator.

Acknowledgements

I would like to thank my project supervisor, Stephen Farrell, for the support he has provided me during the development of this thesis.

I thank my family for their love, sustain and encouragement and for all the sacrifices they made to give me the great opportunities I had in my life.

I am grateful to my friends, my flatmates and to the 2005/2006 NDS class who have made this year a memorable experience.

DANIELA BRUNO

University of Dublin, Trinity College

September 2006

Contents

1 INTRODUCTION ____ IX

1.1 Problems related to space communication ____ 3

1.2 Delay Tolerant Networks ____ 4

1.3 Earth Orbiting Satellites ____ 5
1.3.1 Low Earth Orbit (LEO) Satellites ____ 6
1.3.2 Medium Earth Orbit (MEO) Satellites ____ 7
1.3.3 Geosynchronous Earth Orbit (GEO) Satellites ____ 7

1.4 Protocols used in deep space network environment ____ 8
1.4.1 The Transmission Control Protocol ____ 9
1.4.1.1 Format of IP packet header ____ 10
1.4.1.2 Format of TCP segment ____ 12
1.4.1.3 Connection establishment and termination ____ 13
1.4.1.4 Flow and congestion control ____ 14
1.4.2 TCP deficiencies in deep space network environment ____ 16
1.4.3 The proposed solutions ____ 18
1.4.4 The TCP/IP protocol variations ____ 20
1.4.4.1 TCP Westwood ____ 20
1.4.4.2 TCP Peach ____ 21
1.4.4.3 TCP Peach + ____ 22
1.4.5 The Bundle Protocol ____ 22

1.5 STK (Satellite Tool Kit) ____ 25
1.5.1 The simulation objects ____ 28

1.6 The Scenarios ____ 30
1.6.1 The MoonProbe’s travel ____ 30

1.7 Related work ____ 33

2 THE JBDS SIMULATOR’S HIGH LEVEL DESIGN ____ 36

2.1 The JBDS simulator architecture ____ 36

2.2 The JBDS simulator’s user ____ 38
2.2.1 Edit simulation input ____ 38
2.2.2 Start simulation ____ 39
2.2.3 Terminate simulation ____ 39
2.2.4 Read output ____ 39

2.3 The JBDS simulator class and package diagram ____ 40

2.4 The JBDS simulator sequence diagram ____ 42

3 THE MANAGER ____ 43

3.1 The Initialization Manager ____ 44

3.2 The Request Data Manager ____ 45

3.3 The Node Update Manager ____ 49

3.4 The Time Manager ____ 49

3.5 The Report Writer Manager ____ 50

4 THE SIMULATION NODES ____ 53

4.1 A TrafficSourceNode ____ 55

4.2 Routing ____ 56
4.2.1 The XML input file ____ 56

5 THE PROXY ____ 60

5.1 The implementation ____ 62
5.1.1 JNI (Java Native Interface) and Raw Sockets ____ 62
5.1.2 The APIs used ____ 63
5.1.3 The Packet Filter ____ 64
5.1.3.1 How to distinguish between different simulation nodes ____ 66
5.1.4 The Packet Sender ____ 66
5.1.5 The Delay ____ 67
5.1.6 The BER (Bit Error Rate) ____ 69

6 TESTING AND EVALUATION ____ 71

6.1 The first Scenario ____ 71
6.1.1 The results obtained: the delay ____ 73
6.1.2 The results obtained: the queue size ____ 74
6.1.3 The results obtained: the throughput and fairness ____ 76

6.2 The second scenario ____ 78
6.2.1 The results obtained: the queue size ____ 79
6.2.2 The results obtained: the throughput and the fairness ____ 80

6.3 The third scenario ____ 81
6.3.1 The results obtained: the queue size ____ 82
6.3.2 The results obtained: the throughput and the fairness ____ 83

7 CONCLUSIONS AND FUTURE WORK ____ 84

BIBLIOGRAPHY ____ 86

List of Figures

Figure 1: The internet protocol suite (adapted from [22]) 9
Figure 2: TCP and IP headers' format (adapted from [25]) 12
Figure 3: TCP's sliding window (adapted from [23]) 14
Figure 4: STK 2D User Interface window 26
Figure 5: STK 3D User Interface window 26
Figure 6: STK/Connect interface to any external application (adapted from [12]) 28
Figure 7: The launch, the parking orbit and the Trans lunar injection 31
Figure 8: MoonProbe's trajectory adopted to enter the orbit around the Moon 32
Figure 9: The software's high level architecture. 36
Figure 10: The user's UML use cases 38
Figure 11: The JBDS simulator UML package diagram 40
Figure 12: The JBDS simulator UML class diagram 41
Figure 13: The JBDS simulator UML sequence diagram 42
Figure 14: The Manager's internal architecture 43
Figure 15: The Initialization Manager module 44
Figure 16: A sample line of the output file 51
Figure 17: Client-Server architecture 60
Figure 18: Proxy architecture 61
Figure 19: Mask used to apply the error rate to the packets 70
Figure 20: The first scenario 72
Figure 21: The first and second scenario topology 72
Figure 22: Increase in packet's delay as MoonProbe leaves Earth's attraction 74
Figure 23: Queue size at the MoonProbe, compared to the access times (first simulation) 75
Figure 24: Throughput first scenario 76
Figure 25: The second scenario 78
Figure 26: Queue size at the MoonProbe, compared to the access times (second simulation) 79
Figure 27: Throughput second scenario 80
Figure 28: The third scenario topology 81
Figure 29: Queue size at the MoonProbe, compared to the access times (third simulation) 82
Figure 30: Throughput third scenario 83

List of Tables

Table 1: Duration of the slow start phase for the different types of satellites [2]....... 18
Table 2: Example of an Hashtable used for routing purposes 59

1 Introduction

The enlargement of human horizons in recent decades has led to the growth in importance of deep space travel and, more generally, space communication. Global Positioning System technology and satellite television service are two examples of how this technology have become part of everyday life; but still much work has to be done in the field and deep space communication is still very much under study and development.

In particular, network communication in a space environment can be challenging for various reasons, for example limitations in bandwidth and resources (mostly due to difficulties to power supply), but it is also prone to be affected by long and variable delays, high error rates, and intermittent loss of connectivity.

Various new network protocols have been proposed by research scientists in order to hand such environment problems, as has the idea of using well known and trusted protocols also for deep space links. This latter idea is however affected by some protocol deficiencies in that environment. However, using TCP for example could lead to an effective decrease of the cost of dedicated hardware and would benefit from the years of usage and experience (especially from a security point of view). On the other hand, experiments and research have shown that TCP's performance, for example, could be very poor in such an environment [1] [2].

Thus, other protocols have been proposed. Some protocols, such as the Bundle protocol sit on the top of the transport layer (even if not necessarily on the top of TCP), whereas some other protocols sit at the data-link layer, for example the Licklider Transmission Protocol; finally, some others have been proposed as variations of the Transport Control Protocol and have added some features and new algorithms to make it more suitable for network links with high error rates and a non efficient use of the available bandwidth. The most well known protocols that belong to the last group are:

- TCP Westwood [4].
- TCP Peach [2],
- TCP Peach + [3],

Unfortunately these variations of TCP are just at the experimental stage and have no reliable implementations in Java.

All these protocols have been tested by research scientists using different means: network simulators or particular frameworks. In general, there is a need for a framework able to provide reliable information on how the protocols behave in the deep space environment.

A sample solution to the problem of simulating deep space links could be the development of the Space Based Internet (SBI) emulator described in [5] [6]. The approach proposed in Baliga's work will be explained in detail later on, but it was based on the use the quality of service (QoS) mechanisms provided by the Linux kernel, and since it had to apply some kernel modifications, it caused problems of compatibility with other kernel versions and in case of kernel upgrade.

A different approach to the SBI emulator would be to use network simulation. ns-2 is one of the most popular network simulators and it comes with an extension for wireless networks developed within the CMU Monarch project [7]. Henderson and Katz, proposed an enhancement of ns to make it suitable for deep space networks simulations [8].

Another important approach was proposed by S. Endres, M. Griffith and B. Malakooti in [9], and it was based on the use of real sockets. This approach and its drawbacks will be described later more in detail.

The common factor in [6], [5] and [9] is the use of a sophisticated commercial simulator called STK (Satellite Tool Kit) [12].

This thesis proposes a different approach to the problem of simulating deep space networks. Still the STK simulator will be employed to provide reliable data, but its output will be used within another simulation framework: the JBDS (Java Based Deep Space) simulator. This approach will involve the use of an IP layer proxy which will have the task of simulating the packets' delay and BER (bit error rate), in order to study the behaviour and throughput performance of the TCP protocol (and application layer protocols) in a deep space network environment.

The framework was developed using the Java and C Programming languages.

As already mentioned, the JBDS simulator was designed in order to test and evaluate performance of application layer protocols in a space network and in general that could be adapted to simulate a Delay Tolerant Network.

Three scenarios will be used to evaluate the framework. These scenarios have been created by the means of the STK simulator, and have been chosen in order to provide valuable and interesting results.
This thesis is composed of seven chapters. The first chapter is an introduction to the most important theoretical concepts: for example delay tolerant networks, satellite communication and protocols used in deep space environment; in addition this chapter provides a detailed description of the external simulator tool used (STK).
The second chapter describes the architecture of the framework, whereas the third the fourth and the fifth chapters, describe in detail the different JBDS simulator software modules. Chapter six shows and explains the scenarios employed for testing the framework and the results obtained.
Finally chapter seven provides some conclusion and suggestions about future work.

1.1 Problems related to space communication

Deep space data communication is affected in particular by long and variable delays, intermittent loss of connectivity, asymmetric data rates, higher signal-to-noise ratio and higher error rates, if compared to terrestrial links.
The long delay is due to the long distances involved. For instance, the distance between Earth and Mars fluctuates between 56 million and 400 million km, which means that the communication delay involved ranges from 4 to 20 minutes.
Variable delay is mostly due to planetary dynamics. For instance, the Earth's rotation could be the reason for a variation in the relative position of two endpoints in the network, and this could lead to an increase in distance and, as a consequence, to an increase of the communication delay between the two endpoints.
Because of the Earth's rotation, and, in general, of planetary dynamics, another problem could be encountered, the intermittent loss of connectivity. This problem can be also called "episodic connectivity" [13] and it prevents two endpoints from establishing and maintaining a continuous communication path. This is the primary reason why NASA, in order to provide continuous connections between two endpoints even when earth rotation obscures visibility, needs to use at least three stations (for example: Canberra in Australia; Madrid in Spain and Goldstone in

California, USA); these stations should be positioned at roughly 120 degree intervals around the Earth surface.
The asymmetry of data rates is another issue; it refers to the difference in rate of the inbound from the outbound traffic. For example, an asymmetry of 15:1 means that for 400000 bps received, 28800 bps are transmitted; in spacecraft mission this asymmetry can be on the order of 1000:1 or higher [13].
Another important issue is related to the fact that a deep space link is extremely bit error prone. In the normal internet, mostly the wired one, the bit error rate is very low. Packets are usually received uncorrupted at the destination. This is not assumed to be true in wireless networks, and mostly in deep space networks. For instance, absorption in a medium is one of the reasons that can cause loss of data and deep space links are particularly prone to atmospheric absorption which can be caused among the others by: rain, fog, clouds, snow, hail, water, molecular oxygen and ionospheric effects.
In general, many other factors can cooperate to absorb the signal, and one of the most important is signal attenuation. This value would highly increase with distance (the strength of a radio signal falls in proportion to the square of the distance travelled) [5].
In addition, it is fundamental to mention noise: one of the characteristics of a signal is the signal-to-noise ratio. Noise can affect a signal in space due to space elements like Sun, Moon, galactic noise, cosmic noise and atmospheric noise.
Finally, another relevant issue is bandwidth: the radio spectrum is unfortunately a limited natural resource; thus just a restricted amount of bandwidth is available to satellite systems, and it is typically controlled by licenses. This scarcity makes it difficult to trade bandwidth to solve other design problems.
Not only is bandwidth limited by nature, but the allocations for commercial communications are limited by international agreements so that this scarce resource can be used fairly by many different applications.

1.2 Delay Tolerant Networks

Deep space networks and sensor networks are two examples of how the environment can badly affect communication between endpoints. Because of the problems described in paragraph 1.1, but also for power management and visibility problems,

usually these networks deploy different protocols from the other networks and they rarely use IP. In order to achieve interoperability between these different networks, in 2002 Kevin Fall proposed a new approach to network architecture design described in [14] as: "…*a network architecture and application interface structured around optionally-reliable asynchronous message forwarding*…"
This architecture was defined as Delay Tolerant Network Architecture and was designed to operate as an overlay above the transport layers of the networks it interconnects, and to provide services such as storage and retransmission.
Delay tolerant networks include:

- Exotic media networks, which include Exotic near-Earth satellite communications, very long-distance radio links (e.g. deep space RF communications with light propagation delays in the seconds or minutes), communication using acoustic modulation in air or water, and some free-space optical communications.
- Terrestrial mobile networks.
- Military ad-hoc networks, which usually operate in hostile environments where mobile nodes, environmental factors, or intentional jamming may be cause for disconnection.
- Sensor and Sensor/Actuator networks, which are characterized by extremely limited end-node power, memory, and CPU capability. Moreover, these networks could comprise thousands or millions of nodes per network, and in this case scale is an issue.

The application layer protocol studied for this type of architecture is usually called the Bundle protocol and it will be described more in detail in paragraph 1.4.5.

1.3 Earth Orbiting Satellites

It is important to provide an overview of the Earth orbiting satellites, in order for the reader to understand the main issues regarding satellite communication and the evaluation scenarios chosen.

Satellites nowadays have different types of applications. Most of them are used for communication purposes, and depending on the type and the quality of service requested, satellites have different on-board instruments and different orbits. Based on the orbit's altitude, communication satellites can be divided in three major groups:

- Low Earth Orbit (LEO) satellites
- Medium Earth Orbit (MEO) satellites
- Geosynchronous Earth Orbit (GEO) satellites

These three types of satellites will be described more in detail in the following paragraphs.

1.3.1 Low Earth Orbit (LEO) Satellites

Low Earth Orbit (LEO) satellites' altitudes reach up to roughly 1000 Km, thus their orbit is relatively close to the Earth surface. These satellites, because of their low altitude, are only visible from within a radius of roughly 1000 kilometres from the sub-satellite point.
Communication between Earth and LEO satellites has low delay and low bit error rates; however, for total Earth coverage a large number of LEO satellites is required. Satellites in low earth orbit change their position quickly; thus, even for local applications, thus, a large number of satellites are needed to ensure uninterrupted connectivity (systems of more than one satellite are called constellations of satellites).
Even if a larger number of satellites is required for a LEO constellation, these satellites are less expensive to be placed in the right orbit compared to higher orbit satellites (such as GEO and MEO) and, because of their proximity to the ground, require lower signal strength (Signal strength's loss is proportional to the distance squared, which means that a huge signal strength is needed for high orbit satellites). In addition to this, there are important cost differences in the equipment needed onboard and on the ground to support the two types of missions.
Thus, the choice between a constellation of LEO satellites or of higher orbit satellites, should really depend on the type of application the satellites have and the type of service required.

For example, LEO satellites find one of the most common applications in the field of Earth Observation (because they have a high resolution for the Earth's surface).
Examples of commercial LEO projects are Iridium (780 km, 66 satellites) and Globalstar (1400 km , 48 satellites) [15]. Iridium is the largest commercial satellite constellation in the world; its main purpose is delivering essential communications services to and from remote areas where no other form of communication is available [18].
On the other hand, Globalstar [19] provides reliable voice and data service across USA and in more than 100 other countries.

1.3.2 Medium Earth Orbit (MEO) Satellites

Medium Earth Orbit (MEO) Satellites are found at altitudes ranging from a few hundred to a few thousand miles (1000 Km to around 10000 Km.).
The MEO orbit is a compromise between the LEO and GEO orbits. Compared to LEOs, they have more distant orbits, thus, a fewer number of MEO satellites is required to provide full earth coverage. Compared to GEO satellites, MEO operate with smaller and less expensive equipment and with less latency (signal delay).
However, even though MEO satellites are in view longer than LEO satellites, they may not always be positioned at an optimal elevation. Thus, MEO systems often offer coverage overlap from satellite to satellite, which requires more sophisticated tracking and switching schemes than GEO satellites.
MEO constellations have a number of satellites that range roughly from 10 to 17, and they are distributed over two or three orbital planes.
A typical example of a MEO global satellite system is ICO (10335 km, 10 satellites) [15], which integrates mobile satellite communications capability with terrestrial networks. ICO system can route calls from terrestrial networks through ground stations that will select a satellite through which the call will be connected [16][17].

1.3.3 Geosynchronous Earth Orbit (GEO) Satellites

Geosynchronous Earth Orbit (GEO) satellites orbit at an altitude of about 36000 Km and their orbit period equals to the rotation period of Earth.

GEO satellite systems offer maximum coverage of the Earth area with a minimum number of satellites. Three GEO satellites can, in fact, provide full Earth coverage. For these reasons, GEO satellites are mainly used for data broadcast purposes. However, the round trip time is about 250 ms and they tend to have high bit error rates.

Geostationary satellites are a particular type of geosynchronous satellites. They have zero eccentricity and zero inclination, i.e., they orbit the Earth over the equator.

A satellite in a geostationary orbit seems to remain in a fixed position to an earth-based observer, because it revolves around the Earth at a constant speed once per day over the equator.

The geostationary orbit is very useful for communications purposes because ground based antennae, which must be directed toward the satellite, can operate effectively without the need for expensive equipment to track the satellite's motion. Especially for applications that necessitate a large number of ground antennas, the money saved in ground equipment can more than justify the onboard complexity and extra cost of lifting a satellite into a geostationary orbit.

An example of a constellation of GEO satellites can be TDRSS (Tracking and Data Really Satellite System) [10][11]. TDRSS provides links between low earth orbiting spacecrafts and the ground. The TDRSS currently consists of six first generation Tracking and Data Relay Satellites (TDRS); just three of those satellites are actually used for operational support at any given time. The remaining vehicles provide backup in case of failure of an operational spacecraft and, in some specialized cases, various types of resources. The TDRSS supports many NASA missions; among them, the Hubble Telescope, the Space Shuttle and, in the near future, the International Space Station.

1.4 Protocols used in deep space network environment

Among the protocols that have been widely tested for application in a deep space environment, TCP is one of the most important, and it is the one that the JBDS simulator uses. Thus, it is necessary to provide an introduction to TCP's most relevant features.

1.4.1 The Transmission Control Protocol

The Transmission Control Protocol (TCP) standard was defined in RFC 793 [21] by the Internet Engineering Task Force (IETF) in 1981.

It is part of the "Internet suite" of protocols. Anther protocol that belongs to this suite is the Internet Protocol (IP), which sits below TCP and provides a number of services to its upper layer.

The hierarchy of protocols is shown in Figure 1.

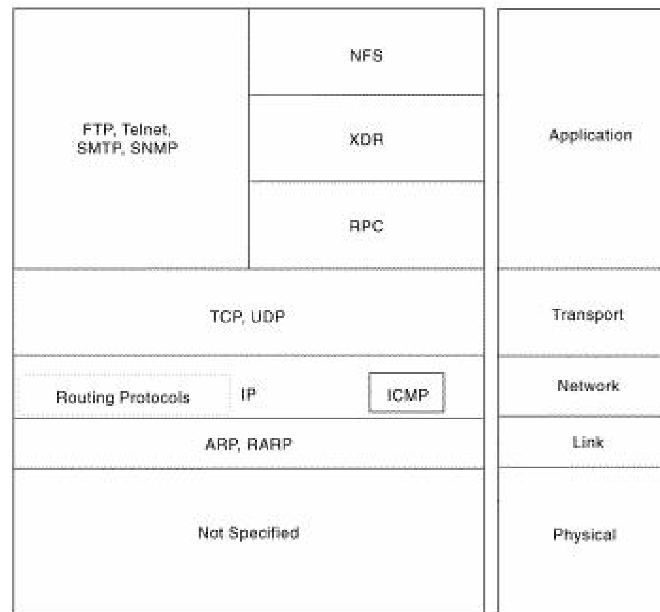

Figure 1: The internet protocol suite (adapted from [22])

TCP is the transport layer protocol, while IP is the network layer protocol. As shown in Figure 1, the transport layer protocol in the internet protocol suite, can be either TCP or UDP (User Datagram Protocol). The difference between the two protocols consists mostly in the type of service provided. TCP is a connection-oriented, reliable protocol and ensures that all segments sent by the source will be received at the destination. On the other hand, UDP does not establish a connection between source and destination (each datagram contains its own address information), and the correct and ordered delivery of data is not ensured. This means that the destination could receive a lower number of datagrams compared to the number of datagrams that was actually sent by the source and the messages could not be received in the right order.

A number of mechanisms are used to provide TCP's reliability. These mechanisms are:

1. Computation of checksum;
2. Deletion of duplicates;
3. Retransmission;
4. Time-out;
5. Sequencing.

The checksum is a computed value that depends on the content of a segment. This value is sent along with the segment when it is transmitted. The receiver computes a new checksum on the data it received and can compare this value with the one sent along with the segment, in order to understand if the data was received correctly.
Another feature is deletion of duplicates: TCP keeps track of the segments received in order to discard duplicate copies of data that has already been received.
Moreover, in order to guarantee the delivery of data, TCP retransmits segments that may be lost or damaged. When the destination receives correctly a segment it is supposed to send back an acknowledgement message to the sender. This message confirms successful reception of data. The lack of this message, leads to a retransmission.
The sender waits for a bounded length of time for an acknowledgement message. If the acknowledgement is not received within that time, then the source will retransmit the data segment.
Another feature that ensures reliable service is sequencing. Each TCP segment contains a sequence number that will give the receiver the chance to reconstruct the right order of the messages.

1.4.1.1 Format of IP packet header

Before analysing the format of the TCP packet header, it is important to briefly explain the IP packet header fields (shown in Figure 2), in particular IP version 4 (IPv4):

- The first field contains the current version of IP. For IPv4, this has a value of 4.
- The IP Header Length field is the second and it is usually set to 5
- The Type of Service field indicates particular Quality of Service needs from the network.
- Total length contains the sum of the IP header length and of the payload length.
- The Identification is used during reassembly of fragmented datagrams[1].
- The Flags field contains three flags that give instruction to routers about whether they are allowed to fragment a datagram
- The Fragmentation offset indicates where in the datagram this fragment belongs.
- Time to Live (TTL) is the maximum number of hops that the packet is allowed to live. It is useful to prevent routing loops. If this field contains 0, the packet should be destroyed.
- The Protocol field contains the value which marks the protocol used in the data field of the packet (In Figure 2 the value for the protocol field is 6 because the IP payload contains a TCP segment).
- Header Checksum contains the IP header checksum. This value can change at each hop (for example because of the TTL).
- Source Address contains the IP address of the sender.
- Destination Address contains the IP address of the receiver.
- The Options field contains different options, it has variable length and it is not often used.
- The Padding field is used to ensure that the IP header ends within a 32 bit boundary.

[1] Datagram fragmentation is the process in which an IP datagram is broken into smaller pieces to fit the requirements of a given physical network [53].

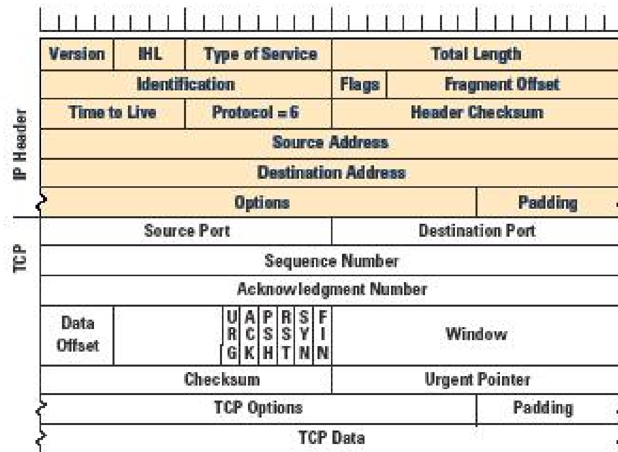

Figure 2: TCP and IP headers' format (adapted from [25])

1.4.1.2 Format of TCP segment

The format of a TCP segment is shown in Figure 2.

The source and port numbers identify the ports dedicated, at the source and at the destination, for the communication between sender and receiver. These two are two very important fields, because, as explained in chapter 5, in the JBDS simulator the source and destination ports number will be used to identify the endpoints in the simulation.

The sequence number field is used to reconstruct the order of segments, whereas the acknowledgement number field identifies the next data byte the sender expects from the receiver. Therefore, the number will be one greater than the most recently received data segment. This field is only used when the ACK control bit is turned on.

The Data Offset field specifies the length of the TCP header; whereas the bits that follow in the header are unused, reserved for future use.

The TCP segment header contains some useful control bits:

- Urgent Pointer (URG). This bit communicates to the receiver whether the urgent pointer field contains information that should be read or not.
- Acknowledgement (ACK). This bit is set when the segment is an acknowledgement message.

- Push Function (PSH). If this bit field is set, the receiver should deliver this segment to the receiving application as soon as possible.
- Reset the Connection (RST). This bit is set when the sender is aborting the connection.
- Synchronize (SYN). This bit is used during the initial stages of connection establishment between a sender and receiver and means that the sender is attempting to "synchronize" sequence numbers.
- No More Data from Sender (FIN). This bit is set when the sender wants to communicate to the receiver that it has reached the end of its byte stream for the current TCP connection.

The other fields that are always present in a TCP header are window, checksum and Urgent Pointer. The content of the checksum field has been already explained.
The window field tells the sender how much data the receiver is willing to accept, and the urgent pointer (read when the URG flag is set), communicates to the receiver where the last byte of urgent data in the segment ends.

1.4.1.3 Connection establishment and termination

The procedure to establish a connection between two endpoints is called 3-way handshake, because, during this phase, three messages are exchanged between the two hosts. The first message is sent from the host (A, for example) which wants to establish a connection with another host (B, for example); thus, it is a request for connection (the SYN bit is employed for this purpose). The second message (from B to A) is an acknowledgement to the first message and a connection acceptance.
The third message goes back from A to B and it is the acknowledgement to the second message, and states the establishment of the connection.
On the other hand, in order for a connection to be released, four segments are required to completely close a connection. This is due to the fact that TCP is a full-duplex protocol, which means that each end must shut down the connection independently from the other. The first two messages close the communication at one side and the other two close it at the other side.

1.4.1.4 Flow and congestion control

In TCP, flow control is a technique that ensures that the sender's sending rate matches the network and receiver transmission rate. Flow and congestion control are different mechanisms since the second mostly refers to overload of intermediate network devices, as IP routers.

Flow control is done by the means of the so-called sliding window. Figure 3 shows a sliding window of 4 segments.

TCP's window is the range of segment's sequence numbers that the receiver is prepared to accept in its buffer. Thus, the receiver, with every acknowledgement message, must specify the allowed number of octets that the sender may transmit before receiving further permission [21], (this is the purpose of the window field in the header). Therefore, the sender can send up to this amount of data before waiting for an update on the window size from the receiver. The sender has to buffer all its own sent data until it receives an acknowledgement for that data. When an acknowledgement message is received the sender can infer that the segment has been correctly received by the receiver and can move the sliding window (Figure 3 (b)).

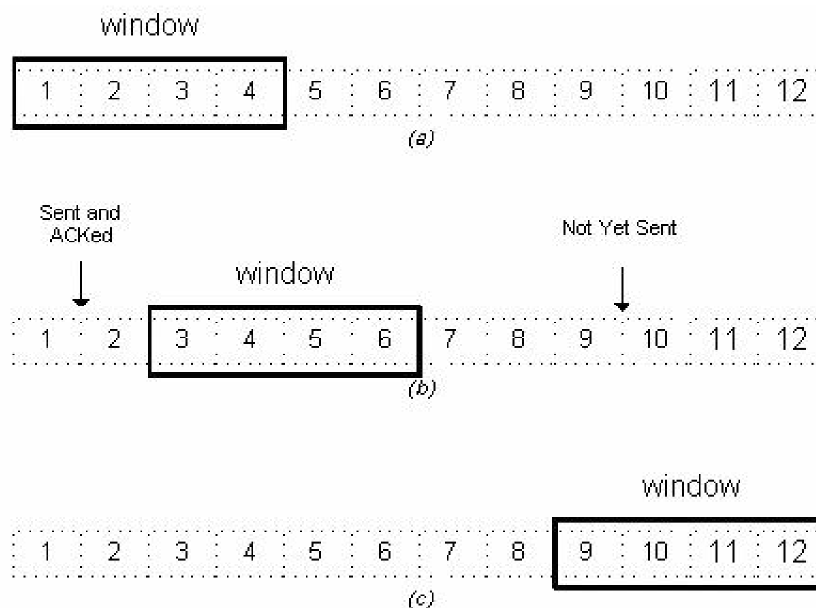

Figure 3: TCP's sliding window (adapted from [23])

If an acknowledgement for a segment is not received within the timeout, it is retransmitted.

TCP uses a congestion window (cwnd) on the sender side for congestion avoidance. The congestion window indicates the maximum amount of data that can be sent out on a connection without being acknowledged. TCP detects congestion when it fails to receive an acknowledgement for a packet within the estimated timeout or when it receives three duplicate acknowledgements. In such a situation, it decreases the congestion window to one maximum segment size and the congestion window threshold (ssthresh). At the time congestion is experienced ssthresh is set to half of the congestion window. However, if after a loss the sender receives acknowledgements for all the packets that it transmitted within the timeout period, then the congestion window increases quickly (it is doubled each time an acknowledgement is received) until the threshold is reached (slow start), and after that, it is increased by one segment every time a new acknowledgement is received (congestion avoidance). The slow start and congestion avoidance mechanisms were firstly introduced as an improvement of TCP's congestion performance, usually called TCP Tahoe [24].

There is a variation of the slow start algorithm known as fast recovery. In the fast recovery algorithm, when segments are not received (detected by three duplicate acknowledgements) the congestion window size is reduced to the slow-start threshold, rather than to one segment.

Before Tahoe, TCP used the mechanism called additive increase multiplicative decrease (AIMD), which incremented the congestion window by one for each acknowledgment received and halfed it for every timeout.

Another important algorithm introduced with TCP Tahoe was fast retransmit. This algorithm proposed the use of duplicate acknowledgements for congestion detection, in other words, if 3 duplicate acknowledgements arrive at the sender, TCP assumes that a segment has been lost and retransmits the missing segment without waiting for the timeout to expire.

Together with these enhancements, a lot of work has been done in trying to improve TCP. A great part of this work has been motivated by the shortcomings exhibited by TCP when used with large bandwidth-delay products (BDP) [26].

1.4.2 TCP deficiencies in deep space network environment

Both experimental and analytical studies [2] showed that TCP has performance deficiencies if applied in networks environments challenged by long propagation delays and relatively high link error rates such as deep space networks.
It has been showed with research experiments and simulations that, in deep space networks, TCP throughput decreases because of three main reasons:

- The long propagation delays cause longer duration of the Slow Start phase during which the sender may not use all the available bandwidth;
- TCP's throughput is limited by the formula (1)

$$throughput = window\ size\ /\ RTT \qquad (1)$$

  Thus, for high RTTs, TCP's throughput significantly decreases.

- When TCP detects a packet loss, the congestion window is halved or reduced to one segment (depending on the implementation). This tends to significantly affect TCP performance. In fact the protocol was initially designed to work in reliable network environments such as wired networks, i.e. with low link error rates, where segment losses were mostly due to network congestion. In wireless networks, instead, and especially in deep space networks, segment losses occur mostly because of link/bit errors or for temporary loss of connectivity. As a result, the sender decreases its transmission rate each time a loss is experienced, which causes an unnecessary decrease in the use of the available bandwidth and in the throughput in general.

In [27] the authors write about the problem of distinguishing between congestion induced drops and those caused by corruption, in the absence of such a distinction, packet loss is assumed to indicate congestion and the congestion window will be reduced with the subsequent, unnecessary, worsening of performance. On the other hand incorrectly interpreting loss as caused by corruption and not reducing the transmission rate accordingly, can lead to congestive collapse.

Many solutions have been proposed to overcome this problem, among them is worth mentioning Enhanced-TCP, or E-TCP [28], where an agent (transit agent) is able to determine the type of loss that occured with the use of an acknowledgment packet (called $ACK_{C\text{-}CLN}$). The proposed scheme allows the transport protocol to distinguish between congestion and corruption losses and then perform appropriate recovery actions. The drawback is that E-TCP requires the addition of some functionalities at the gateway node and the sender.

There are two standard methods that can be used by TCP receivers to generate acknowledgments. The first one states that an acknowledgement should be generated for each segment received at the destination, whereas the second states that in deep space environments "delayed" acknowledgements should be used [29]. With the latter approach, an acknowledgement message is generated for every second full-sized segment, or if a second full-size segment does not arrive within a given timeout. In other words, TCP delays transmission of acknowledgements for up to 200ms (in most implementations), the hope is to have data ready in that time frame. Then, the acknowledgement can be piggybacked with a data segment.

However, the congestion window size's growth occurs much more slowly when using delayed acknowledgements.

In the beginning of a new connection, the sender executes the slow start algorithm to probe the availability of bandwidth along the path. The average time required by the slow start to reach a bit rate *B* is shown in Equation (2).

$$t_{\mathrm{SlowStart}} = \mathrm{RTT} \bullet (1 + \log_2(B \bullet \mathrm{RTT}/l)) \quad (2)$$

RTT represents the round-trip time and l is the average packet length expressed in bits. Equation (2) is satisfied if the delayed option is not implemented.

Table 1 shows the duration of the slow start phase for the three different types of satellites described in paragraph 1.3, i.e., Low Earth Orbit (LEO), Medium Earth Orbit (MEO) and Geosynchronous Earth Orbit (GEO) satellites, and for different bandwidth (B) values.

Type of satellite	RTT	$t_{SlowStart}$ B=1Mb/sec	$t_{SlowStart}$ B=10Mb/sec	$t_{SlowStart}$ B=155Mb/sec
LEO	50 msec	0.18 sec	0.35 sec	0.55 sec
MEO	250 msec	1.49 sec	2.32 sec	3.31 sec
GEO	550 msec	3.91 sec	5.73 sec	7.91 sec

Table 1: Duration of the slow start phase for the different types of satellites [2]

If the delayed acknowledgement mechanism is implemented, then the time required by the slow start algorithm to reach the bit rate *B* becomes even higher than the one indicated in

Table 1. Many actual TCP applications, like HTTP, are based on the transfer of small files. Thus, it can happen that the entire transfer occurs within the slow start phase. In other words, it is possible that a TCP connection is not able to utilize all available resources in the network.

1.4.3 The proposed solutions

To cope with the performance problems of the slow start algorithm in long propagation delay networks such as satellite networks, there have been several proposed solutions in recent years.

Increasing the initial congestion window (Increasing Initial Window: IIW) is one of these solutions. This concept was first introduced by Allman, Floyd and Partridge in 1998 [30]. The idea is to initially set the congestion window size to a value larger than 1 but lower than 4, ($1 \leq cwnd \leq 4$). With this option, values reported in

Table 1 can be reduced by up to (3xRTT), however these values can still be very high.

Another proposal is connection spoofing. This solution consists of having a router located near the source, in between source and destination, that sends back acknowledgements for the TCP segments it receives, even if they are not yet received at the destination, in order to give the source the illusion of a short delay path.

TCP spoofing improves throughput performance but still it is not the perfect solution, because of three main reasons:

1. The router is responsible for the correct delivery of the all the TCP segments it receives and acknowledges to the source. Thus, it is obliged to carry on a huge amount of work.
2. The path should be static and the router should be always available. A failure can bring to packet losses.
3. If IP encryption is used, the scheme cannot be applied.

Another proposed solution, similar to TCP spoofing, is cascading TCP, also called split TCP [31]; where one TCP connection is divided into multiple connections. This solution is based on the introduction of proxy agents that split TCP into localized segments. When a link failure occurs in one segment, other segments are still available. Since, link failures are more probable in longer connections than in shorter ones, the introduction of proxy agents especially benefits longer connections. In spite of the advantages introduced with split TCP, unfortunately this solution has the same problems as TCP spoofing.

Finally one of the most important algorithms: fast start [32]. Proposed by Padmanabhan and Katz in 1998 especially for speed purposes in web data transfer, the fast start algorithm is an alternative to the slow start algorithm. The basic idea of the fast start alogorithm is to reuse the values of the transmission rate used in the recent past. However, the transmission rate used in the past might be too high for the current actual network condition, which may lead to congestion in the network. Thus, the TCP segments transmitted during this fast start period are carried by low-priority IP packets in order not to decrease the throughput of the actual data segments. Experiments showed the effectiveness of the fast start algorithm compared to slow start. However, fast start has the two main problems:

- The transmitted low-priority segments carry information that must be delivered to the receiver, thus, they are important as the other segments, and, if they get lost, then they must be retransmitted. Since, they have low priority, they can be lost very easily; thus, the sender needs to enhance its recovery algorithms.
- Fast start can only be applied if a recent value of the congestion window size and for the same path, is available at the sender. This implies that within a

short time the same sender has to transfer several files to the same receiver, which may often not be the case.

Some other approaches are studied for links with high product bandwidth-delay, and the results of these studies became variants of TCP. These protocols will be described in the following paragraph.

1.4.4 The TCP/IP protocol variations

Some alternatives to TCP have been proposed to overcome its deficiencies especially in deep space environment. TCP Westwood, TCP Peach and TCP Peach + are considered particularly relevant.

1.4.4.1 TCP Westwood

TCP Westwood was proposed by Gerla, Sanadidi and Ren Wang in 2001 [4]. It consists in a sender-side modification of the classical TCP congestion window algorithm; it can be applied in both wired and wireless links, but the improvement is most significant in wireless links with connectivity loss problems.
TCP Westwood estimates the end to end bandwidth in order to discriminate the cause of packet loss (congestion or packet corruption).
The previous paragraph described a technique called TCP spoofing. The importance of TCP Westwood lies in the fact that this technique is not needed for performance enhancement; in fact, it fully complies with the end-to-end TCP design principle. The idea is to continuously monitor the rate at which the acknowledgements are received at the sender. The estimate is then used to compute the congestion window size and the slow start threshold after a packet loss, which can be detected by three duplicate acknowledgments or after a timeout.
The idea behind TCP Westwood is very simple, while TCP decreases the congestion window after three duplicate acknowledgments, TCP Westwood selects a slow start threshold and a congestion window which are consistent with the effective bandwidth used at the time when congestion is experienced. This mechanism is called faster recovery.

1.4.4.2 TCP Peach

TCP-Peach proposes two new algorithms, called sudden start and rapid recovery. The new algorithms are based on the concept of using low-priority segments, called dummy segments to probe the availability of network resources. These segments don't carry any additional information (they are generated as a copy of the last transmitted data segment), and they do not affect the delivery of actual data traffic.
Dummy segments are sent just after the segment containing data.
If a router on the connection path is congested, it starts discarding dummy segments first (because of their low priority), thus there is no throughput decrease of actual data segments, i.e., the traditional segments.
A very important thing to underline is that TCP Peach doesn't imply any modification of the TCP header; in fact the sender just has to set one or more of the six unused bits in the TCP header to distinguish dummy segments from actual data segments. However, this requires a modification of the receiver implementation.
The receiver recognizes dummy segments and acknowledges them to the sender. The acknowledgements for dummy segments are also marked using one or more of the six unused bits of the TCP header and are carried by low-priority IP segments. The sender interprets the acknowledgements for dummy segments as the evidence that there are unused resources in the network and accordingly, can increase its transmission rate.
Another very important algorithm TCP Peach introduced is the sudden start algorithm, which is supposed to substitute slow start.
In order to explain how this algorithm works, the concept of rwnd should be introduced. Rwnd is the maximum value for the congestion window; thus, in the sudden start algorithm, during the connection establishment phase, the sender sets the congestion window to 1 and after the first data segment, it transmits rwnd - 1 dummy segments every τ, where τ is computed as follows:

$$\tau = RTT/rwnd \qquad (3)$$

As a result, after one RTT, the congestion window size increases very quickly. It is also important to say that the RTT can de determined in advance during the connection setup phase.

The rapid recovery algorithm is supposed to substitute fast recovery; its aim is to solve the throughput degradation problem due to link errors. When a segment loss is detected through a number of duplicate acknowledgments, the fast retransmit algorithm is used. Just after the fast retransmit algorithm has completed, the rapid recovery algorithm takes place, which will terminate when the acknowledgement for the lost data segment is received. This means that the rapid recovery lasts for a RTT.

1.4.4.3 TCP Peach +

Experimental results assure the important result introduced by TCP Peach with the sudden start and the rapid recovery algorithms. However, in 2002, Akyildiz, Zhang, and Fang introduced a variation in the TCP Peach protocol that was called TCP Peach+ [3].

This protocol consists in an enhancement of TCP Peach goodput[2] performance, for application in satellite networks.

In TCP Peach+, two new algorithms are proposed: jump start and quick recovery (variations of sudden start and rapid recovery), which are based on the usage of low priority segments called NIL segments, that are mostly used to check resources availability and error recovery.

Simulation experiments show that in particular situations TCP Peach+ performs better than TCP-Peach, in terms of goodput, and achieves a fairer share of network resources.

Further enhancement of these protocols are TCP Peach ++ and TCP Peachtree, these protocols will not be described in detail.

1.4.5 The Bundle Protocol

[2] Goodput is the number of bits per second forwarded to the correct destination minus the number of bits lost or retransmitted.

The Bundle protocol proposed by K. Scott and S. Burleigh, finds its application in challenged network environment, as the other protocols just described.
The Bundle protocol is an application layer protocol. Thus, it sits on the top of the transport layer (it could also sit on the top of TCP).
The key features of the Bundle Protocol include [33] [34]:

1. Custody-based retransmission;
2. Ability to cope with intermittent connectivity;
3. Ability to take advantage of scheduled, predicted, and opportunistic connectivity (in addition to continuous connectivity);
4. Late binding of overlay network endpoint identifiers to constituent internet addresses.

The Bundle Protocol doesn't necessarily sit on the top of TCP/IP, but the interface between it and the transport layer must be implemented according to the deployed transport protocol. This interface is called "convergence layer adapter".
The data unit exchanged between nodes implementing the Bundle Protocol (Bundle Nodes) is called bundle.
Each node implementing the Bundle Protocol (bundle node) has three conceptual components: a "bundle protocol agent", a set of "convergence layer adapters", and an "application agent".
The bundle protocol agent (BPA) of a node is the node component that offers services to the Bundle Protocol and executes its procedures. The way this is accomplished is an implementation matter. For example, BPA functionality might be implemented individually in each of the nodes that belong to the network, or it could also be coded as a single library that is used in common by any number of bundle nodes on a single computer; finally it might be also implemented in hardware.
A convergence layer adapter (CLA) is an entity that sends and receives bundles on behalf of the BPA, utilizing the services of the underlying transport protocol.
The application agent (AA) of a node is a component of the bundle node. The application agent utilizes the services provided by the Bundle Protocol to effect communication. It is composed by two elements, an administrative element and an application-specific element.

The application-specific element requests transmission and accepts delivery of data units, in addition it processes application specific data units.
On the other hand, the administrative element requests transmission of administrative records (status reports and custody signals), and it accepts delivery of and processes any custody signals that the node receives. The way in which the AA performs its functions is wholly an implementation matter.
Finally there is the Bundle endpoint. A bundle endpoint (or simply “endpoint”) is a set of bundle nodes that are identified as a group by a “bundle endpoint ID”, which is a text string.
Each bundle endpoint ID has the following structure:

<scheme name> **:** <scheme-specific part, or "SSP">

The scheme identified by the <scheme name> in an endpoint ID is a set of syntactic and semantic rules that fully explain how to parse and interpret the scheme specific part. The set of acceptable schemes is unlimited.
When the bundle protocol agent of a node states that a bundle must be “forwarded” to an endpoint, the bundle is sent to all of the nodes that are currently in the “minimum reception group” of that endpoint.
The “minimum reception group” of an endpoint is composed by:

- All the nodes that belong to an endpoint.
- A subset of n nodes belonging to an endpoint.
- A single node if the endpoint contains just one node.

A node must be registered to a group in order to belong to that group. Any number of registrations may be concurrently associated with a given endpoint, and any number of registrations may be concurrently associated with a given node.
When a node receives a bundle, its behaviour depends on whether or not the receiving node is registered in the bundle's destination endpoint; if it is, then the bundle is normally “delivered” to the node's application agent.
In order to understand what happens when the receiving node is not registered in the bundle’s destination endpoint, the concept of bundle Custody must be introduced. A

bundle node “accepts custody” upon forwarding a bundle when it retains a copy of the bundle until the custody of that bundle is “released”.
A “custodial node” of a bundle is a node that has accepted custody of a bundle and has not yet released that custody. When the necessity of forwarding the bundle is determined, then the bundle is forwarded.

1.5 STK (Satellite Tool Kit)

The STK (Satellite Tool Kit) is a piece of software intended to support realistic mission scenarios, and simulate them in real time with animation. STK is a very useful tool for space, land, air and sea missions design. The user can provide different kinds of input information and has the chance, by the means of two graphical user interfaces (one in 2D and the other in 3D, shown in Figure 4 and Figure 5), to control the objects interaction during the animation.
STK gives the user the chance to decide how many and which types of objects to insert inside the simulation scenario, together with the position and the role they have in the simulation.
The software comes with an up-to-date and complete database of:

- Well-known ground facilities.
- Well-known sites of interest.
- The most important stars, planets cities and satellites.
- Other important sites.

It is possible, for example, to use the internal database in order to determine the position of a particular facility on Earth, such as Canberra or Goldstone STDN (spaceflight tracking and data network) stations, or launch complexes such as Cape Canaveral in Florida, but it is also possible to determine the location of site of interest, for example the Apollo landing sites on the Moon.
Moreover it is possible to insert in the simulation user-defined objects loaded from external files.
Thanks to the output files produced from the simulation it is possible to know with extreme precision the distance between objects at a given time (for example between

a satellite and a ground antenna); together with the access and lighting times, or the error rates that can affect the communication between them. The accuracy of this rate is based on complex computations involving among the others, the study of the typical weather conditions of all the Earth areas.

The STK software suite comes with two beautiful graphical user interfaces (2D and 3D) that allow the user keeping track of the object in real time during the animation.

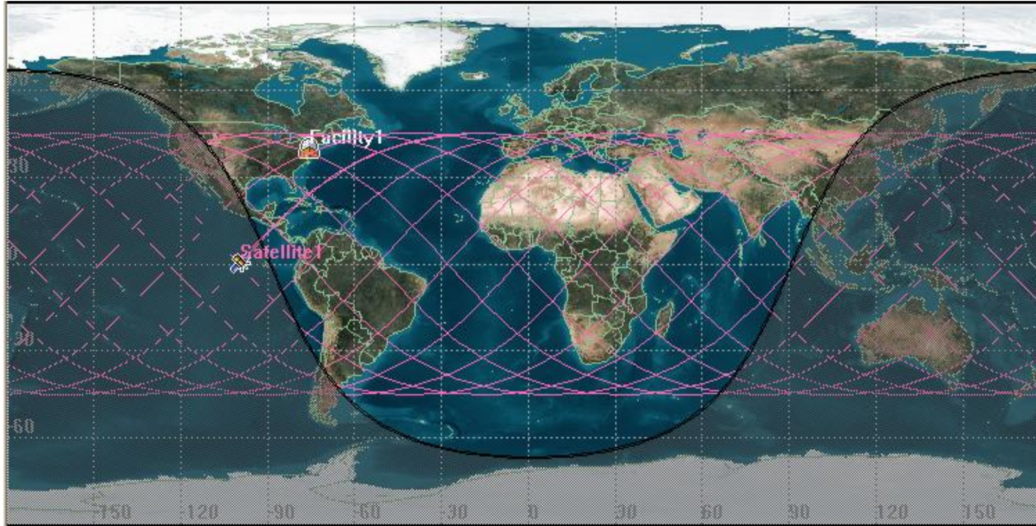

Figure 4: STK 2D User Interface window

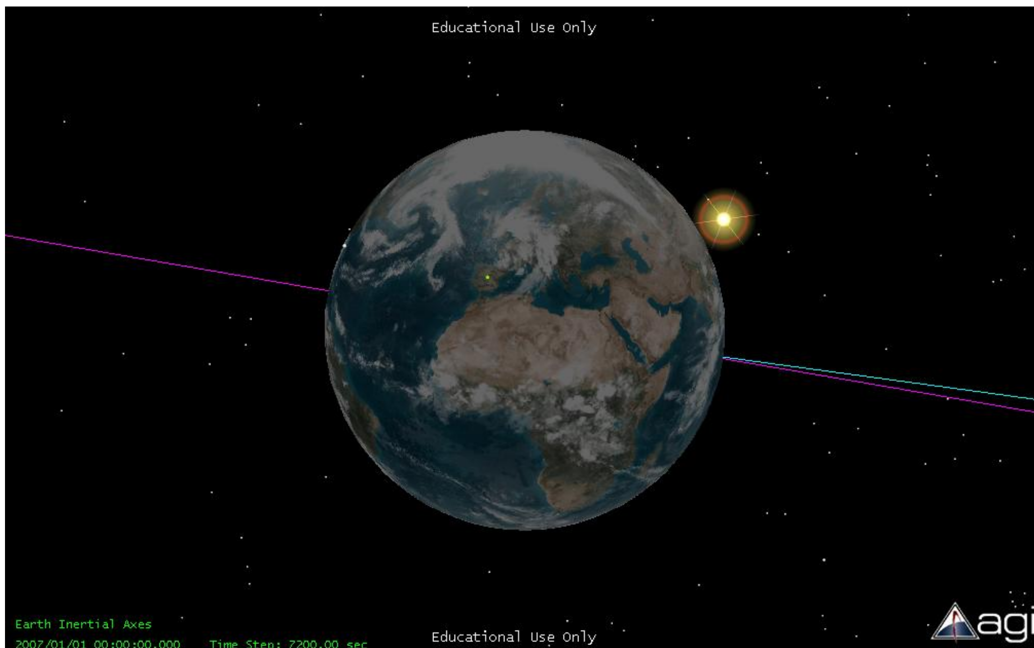

Figure 5: STK 3D User Interface window

The STK simulator allows the user animating the scenario, mastering the animation speed and other parameters. Output data can be produced dynamically during the animation (this feature lets the user keep trace of the objects in real time), or statically independently from the animation.
The output reports and graphs can be accessed using the user interface provided by the software (commands are available in order to generate text reports and graphs); but it is also possible to access them using an external application written in C, Visual Basic or Java.
The STK website [12] provides free downloadable implementation of the scripts in C, Visual Basic and Java that allow any external application to interface STK. These interfaces communicate with STK via a TCP/IP socket interface.
All STK features and capabilities can be divided in modules. Some of them are for free and some others are not. Thanks to the licence obtained for educational purposes, it was possible to use the STK's commercial modules necessary for the development of this thesis.
In particular three of them will be described:

- STK/Astrogator
- STK/Connect
- STK/Comm and Radar

STK/Astrogator is a powerful tool that lets the user program complex space mission scenarios. Every part of the mission is considered as a different Mission control sequence (MCS). By adding, removing, rearranging and editing MCS it is possible to plan and create high complexity scenarios.
Obviously the plan and the creation of a space mission with STK/Astrogator require specific knowledge about physics and astronomy. For this reason the STK simulator provides a number of tutorials with explanations on how to create the desired scenario and hints on how to set the different parameters.
The STK/Connect module provides an easy way to externally interface STK and work in a client-server environment. It also provides a library to easily build applications that communicate with STK. This library contains functions, constants and other

messaging capabilities that can connect third-party applications to STK, and let them receive information about the simulation.
In order to retrieve particular information from STK, a string command should be sent to the simulator. The help manual contains the whole list of all the possible commands that can be used.
Unfortunately the STK/Connect doesn't provide some useful commands. It is not possible for example to poll the simulator for dynamic reports to generate real time data during an animation.
Figure 6 explains how the applications (STK and the external applications) communicate with one another.

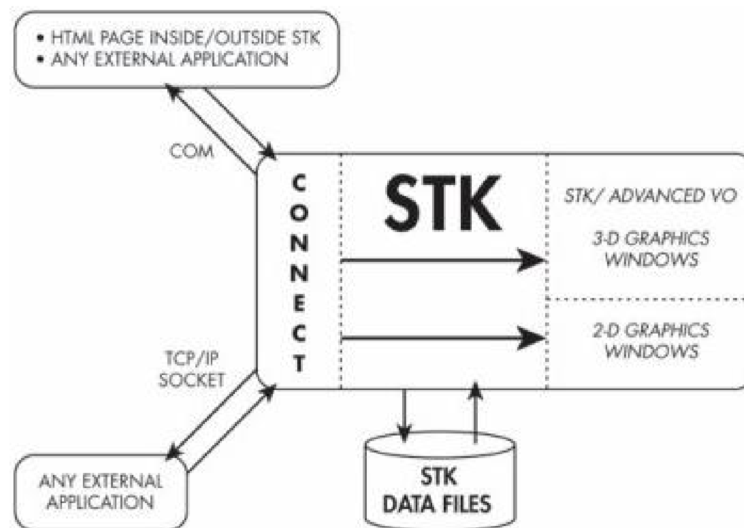

Figure 6: STK/Connect interface to any external application (adapted from [12])

The STK/Comm and Radar module provides some very useful objects. Some of them are sub-objects (which means that they are supposed to be internal to other objects such as Facilities of Satellites), among them particular importance have Receivers and Transmitters, needed for networking purposes.

1.5.1 The simulation objects

The type of STK scenario objects used for this thesis is a sub-set of all the available objects. They include:

- Satellite objects

- Facility objects;
- Receiver objects (internal to a Facility or a Satellite object);
- Transmitter objects (internal to a Facility or a Satellite object);
- Planet objects.

A Facility is a ground object that can be positioned on a number of planetary surfaces: on Earth, on the Moon, or on any other planet object contained in the STK database (which contains all the planets and the most important natural satellites of the solar system).

A Satellite object can represent a satellite or a spacecraft (thanks to the STK/Astrogator module). For each Satellite object is possible to set the type of orbit and the type of propagator. Many different choices are available: for the types of orbit (Geostationary, Polar, Molniya, Sun synchronous, Circular and so on), and for the types of propagators (Two Body, J2 and J4 perturbation and Astrogator, of course).

Transmitter and Receiver objects are useful to simulate data concerning communication. For example, it is possible to insert a transmitter as a sub-object of a satellite object and a receiver as a sub-object of a Facility object, compute access between them, and be able to obtain from the simulator report data such as:

- BER, which is the total Bit Error Rate as carried through the whole link. It includes uplink noise contributions and any gains or losses.
- C/No, which is the total Carrier-to-Noise density at the receiver input as carried through the whole link. As the BER, also the C/No carries uplink noise contributions and any pre-receive gains or losses.
- EIRP, which is the Equivalent Isotropic Radiated Power: a measure of the strength of the signal radiated by an antenna. EIRP takes into account the losses in transmission and the gain of the antenna[3] [35].

Finally the Planet objects. This type of object lets the user decide which of the available planets object to insert in the simulation and its source of ephemeris data.

[3] Antenna gain: "the measure of the effective performance of an antenna compared to a theoretical antenna called an isotropic antenna" [50].

1.6 The Scenarios

In order to evaluate the framework developed in this thesis, three scenarios were created and saved with the STK user interface, and with the help of the STK's tutorials [42] and online forums
[43]. Chapter 6 provides a complete description of the differences between these scenarios, while this paragraph introduces some theoretic concepts useful in order to understand the network architecture employed in the scenarios.
The simulation scenarios were chosen in order to be as realistic as possible, hence the information contained in the paper [44], which describes the network architectures employed in deep space networks, have been widely used.
In all of the three scenarios, the objects involved in the simulation are:

1. A planet object with the Moon as central body.
2. A satellite object called MoonProbe
3. Three Facilities objects on the Moon, called Lunar1, Lunar2 and Lunar3
4. Four Facilities objects on Earth: three of them are actively used within the simulation. The fourth one is just the launch site for MoonProbe. Thanks to STK's database, it was possible to introduce in the scenario some well known Facilities. The three antennas are the well-known Goldstone, Madrid and Canberra, while the launch site is located in Cape Canaveral. These well known Facilities were chosen with the idea of conferring more realism to the scenario.

A short description of the space mission is necessary, in order to understand how communication can be affected by the differences between the three scenarios.
The satellite object's travel was created using the STK module called Astrogator, this module allows the user to "build" the mission dividing it in different parts. All these parts will be described in the following paragraph.

1.6.1 The MoonProbe's travel

The space mission is settled in the first days of the year 2007. The MoonProbe is launched from Cape Canaveral on the 1st January at 4:13 am. At 4:23 am the probe

starts orbiting around Earth. This orbit has a very low altitude (it is a LEO), and it is called “parking orbit”. A parking orbit is an intermediate, waiting orbit adopted by a spacecraft between two phases of a mission. The two phases, in this case are the launch and the Trans lunar injection.

The purposes of adopting a parking orbit are mostly three [45]:

- Perform an injection burn into a higher orbit;
- Break or escape orbit, leaving influence of the body around which it is orbiting (which is the case of our scenario);
- Perform a decent burn, to enter (or re-enter) the atmosphere (if it exists) of the body around which it is orbiting and then land.

At 5:09 am the probe leaves the parking orbit, using a propulsion maneuver called Trans lunar injection. This maneuver is used to set the spacecraft on a trajectory that will intersect the Moon, and it is usually performed by a rocket engine.

Figure 7 shows the different stages in the probe’s travel that had been described before.

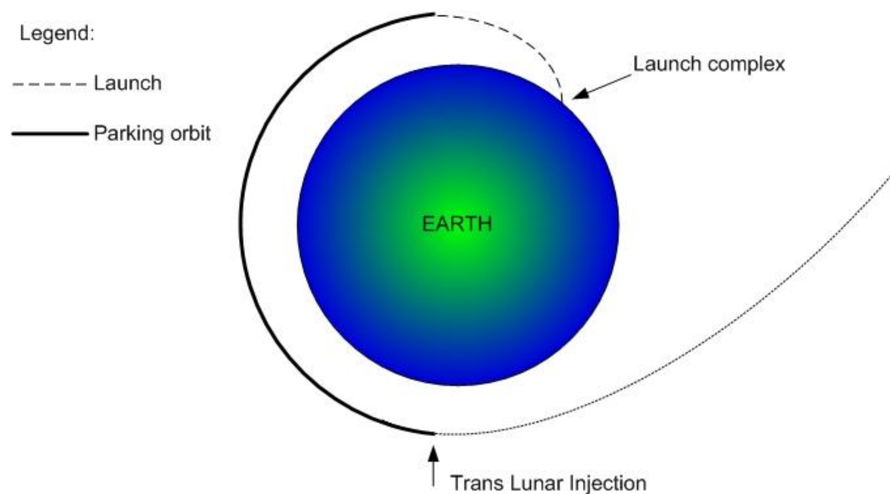

Figure 7: The launch, the parking orbit and the Trans lunar injection

After almost 3 days, the spacecraft leaves the Earth gravity attraction and enters the Moon's gravity. This attraction will lead the spacecraft to the Periselene point[4] in order to enter the orbit around the Moon.

STK's tools provide a valid help in planning the probe's orbit. For instance, it is necessary to input the simulator with the desired eccentricity and inclination for the probe's orbit around the Moon, and STK will automatically change parameters like the launch date, trajectory or speed.

In the sample simulation scenario MoonProbe enters its orbit around the Moon on the 4th of January, with an eccentricity[5] of 0.3 and an inclination[6] of roughly 90°. After the MoonProbe starts orbiting around the Moon it starts being visible to the three Lunar facilities, from this point on the landers will start communicating with the probe[7].

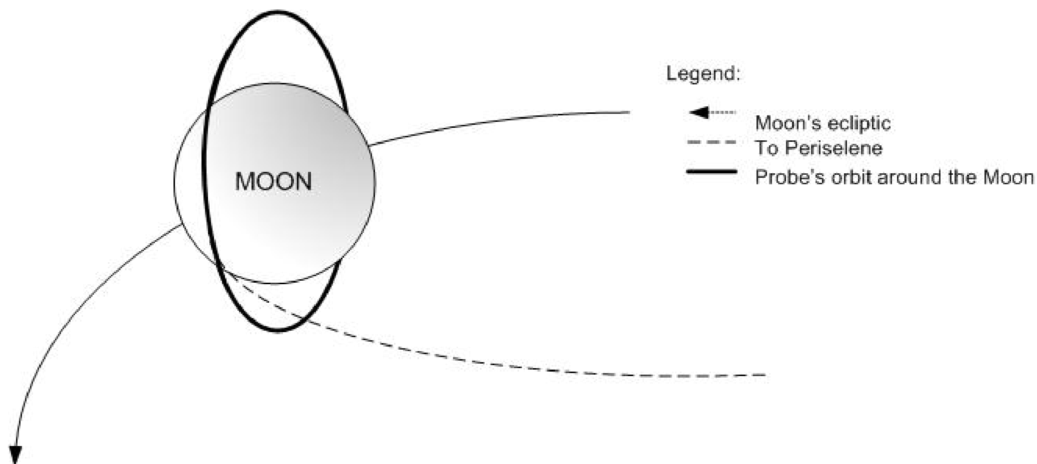

Figure 8: MoonProbe's trajectory adopted to enter the orbit around the Moon

The three ground antennas positions were chosen with the purpose of providing continuous visibility to the probe.

---

[4] "*The Periselene point in an elliptical orbit is the point of closest approach to the Moon, as opposed to Aposelene, which is the farthest point in the orbit*" [46].

[5] "*In astrodynamics, under standard assumptions any orbit must be of conic section shape. The eccentricity of this conic section, the orbit's eccentricity, defines its absolute shape. Eccentricity may be interpreted as a measure of how much this shape deviates from a circle. Eccentricity is equal to 0 for circular orbits, whereas it is comprised between 0 and 1 for elliptical orbits*" [47]

[6] "*Inclination is the angular distance of the orbital plane from the plane of reference (usually the primary's equator or the ecliptic of the central body), normally stated in degree*"[35].

[7] A probe is a device fitted with a number of general purpose or mission specific sensors that is designed to study conditions on, or near a planet.

The communication scenario uses just some of the objects in the STK scenario: the three ground Facilities, the Moon Probe and the three Lunars; each of them represents a different node.
Moreover, in order to re-create the communication scenario in the most realistic way, another node was introduced; a Traffic source. This node simulates a generic endpoint on Earth that sends commands and receives data through the ground antennas. This node could simulate a computer in a space centre (for example in the Kennedy Space Centre).

1.7 Related work

Three are the main contributions in the field, given by three papers:

1. “Design of a space based Internet emulation system” by S. Baliga [6];
2. “Space Based Internet Network Emulation for Deep Space mission applications” by Endres, Griffith and Malakooti [9];
3. “Network Simulation for LEO Satellite Networks” [8].

The first paper is dated 2002, and it is Baliga’s work. The approach described in this paper, benefits from the work done by S. Rallapalli in [5]. They are both technical reports published by the University of Kansas and sponsored by the Glenn Research Center (NASA).
The approach proposed by Baliga and Rallapalli is to *“…emulate the space communication environment characteristics in hardware: This involves implementing innovative algorithms for emulating the latency and bit error rate characteristics of space communication links…”* [6].
Their emulator was called SBI (Space Based Internet) emulator. In the SBI emulation system, a physical node emulates a satellite in space, and an Ethernet device on the node which is capable of transmission emulates communication on a satellite.
The SBI emulator is able to perform also routing tasks, and is composed by the following three main parts (or modules):

- The emulation software module, which is supposed to control and monitor the emulation scenario and emulate portions of the scenarios, actually deployed in hardware.
- The node software module, which comprises the modules responsible for the routing, switching, instrument scheduling and data sourcing tasks.
- The emulation network, which deals with the data transfer mechanism.

Operations related to emulation interfaces are performed by the Virtual Ethernet device layer within the Linux kernel and the features of satellite communication links, like bandwidth constraint, propagation delay and bit error rate are emulated in the SBI emulation system. In order to model deep space links, the SBI emulator uses the QoS queuing disciplines; thus the code had to make kernel modifications. These modifications are kernel specific, thus the operating system and the kernel can't not be upgraded without modification of the SBI emulator. This feature confers low portability to the emulator.

This is the most important reason for the work done by Endres, Griffith and Malakooti, in [9]. Their approach is socket based, but the details are not specified in the paper; for example the language used to implement the emulator is unknown. All the implementation details are hidden, apart from the steps performed to emulate the delay and error rate:

1. Apply an error rate to the packet byte string.
2. Apply a delay.
3. Send the packet using a socket.

As already mentioned, the real implementation is hidden, but the delay and error rate are applied at the node before the packet is actually sent. For this purpose *"every packet sent over the emulation network is passed through a filter before it is sent over the emulation network"*, and this filter is responsible for applying the delay and error rate to the packet.

The authors underline that their emulator framework can be suitable for simulations using different protocols, however an upper layer implementation of those protocols should be provided.

Both these first two approaches use STK as a tool to obtain realistic simulation data.
The last approach described relies on a network simulator, maybe the most popular and used: ns-2. The strength of ns-2 is that it is an open source piece of software and because of this reason it has been improved and modified over time.
ns-2 comes with an extension, developed in 1998 within the CMU Monarch project [7], that allows the user simulating data communication in wireless networks.
It is possible to simulate the delay between nodes and their movement over time.
Unfortunately the CMU Monarch Project extension didn't consider that two nodes could move in a 3D space environment. For this reason Henderson and Katz introduced an enhancement of ns-2 for 3 Dimension scenarios.
This approach leads to the chance of simulating very different network scenarios, for example, with ground stations on Earth, Moon, Mars and so on, and satellites orbiting around the ground stations, along with rovers and aircrafts moving on planet's surfaces.

2 The JBDS simulator's high level design

The new approach proposed in this thesis, relies on the use of Java TCP/IP sockets and IP layer sockets (raw sockets), in order to simulate data communication between nodes in a deep space environment. In the JBDS simulator the nodes have direct connections with one another and the delay and error rate are simulated using an IP layer proxy that is completely transparent to the nodes. This framework's application field lays in the analysis of deep space protocols performance, and since it uses TCP, it could be useful to test and compare the performance of application layer protocols (for example the Bundle protocol). This approach offers two important enhancements compared to the former approaches:

1. Kernel modifications are not deployed; thus the framework doesn't need any code or kernel update to make it portable. The JBDS simulator runs in Linux to benefit of Unix raw-socket's compatibility.
2. No protocols' implementations provided by any simulator are used, but the real protocol stack instead, and actual data packets are sent through the network. This feature is likely to make the result data obtained more realistic.

2.1 The JBDS simulator architecture

The software high level architecture is shown in Figure 9:

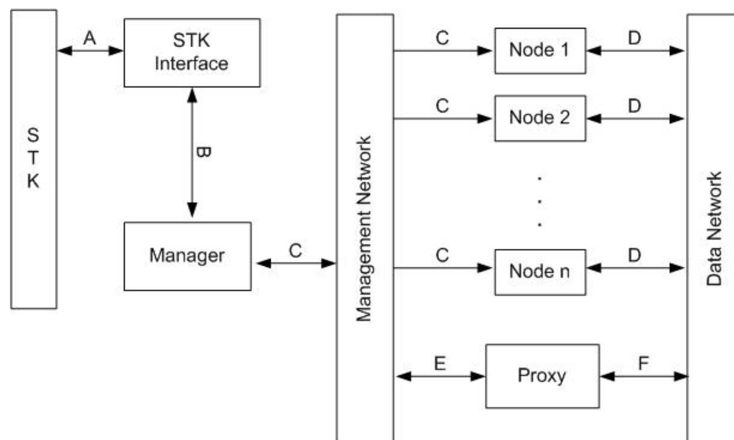

Figure 9: The software's high level architecture.

The STK module (displayed on the left in Figure 9) has been described in paragraph 1.5 and the way it connects to any external application is in Figure 6. The A interface in Figure 9, represents the TCP/IP socket interface that makes any external application communicate with the simulator. In order to properly poll STK for information, commands must be sent in a particular format. Hence, an interface is needed to provide this service (STK Interface module in Figure 9). The Java class StkCon is freely downloadable from the STK website
[12], and provides methods such as connect ( ), disconnect ( ) and sendConCommand ( ), in particular, the last one in the list takes a String object as a parameter, which contains the command to forward to STK. The string sent to STK via the sendConcommand ( ) and their syntax are described in chapter 3.
The Manager module in Figure 9 has the core functionalities in the JBDS simulator; its internal components communicate one another, with the nodes and with the Proxy via the Management network. Message and command passing over this network is not really done using sockets, but through method calls instead, using the O-O way. It is important to distinguish between this network and the data network, which uses sockets for communication and it is employed for message passing between the nodes of the simulation. The idea of the separation between these two networks was taken from [5]. Each node represents a different object in the STK scenario; nodes have different functions: they send packets to the other nodes, perform trivial routing using tables and are responsible for the simulation output; the information they use to send packets to the other nodes (such access times, delay and BER, bit error rate, associated with a link) is constantly uploaded by the manager. Each node has direct socket connections with the other nodes it is allowed to communicate with.
The Proxy module (shown on the bottom in Figure 9), is responsible for filtering the packets at the IP level, delaying them, and applying an error to the packets. The functionality of the Proxy in the simulation will be described more in detail in chapter5.

2.2 The JBDS simulator's user

The JBDS simulator's user has the chance to interact with the application providing and editing the XML input file; moreover he/she can start and terminate the simulation (the simulator polls STK for the start and stop date, however the user can decide to interrupt the simulation beforehand). Finally, the user can read the output of the simulation, after its termination.

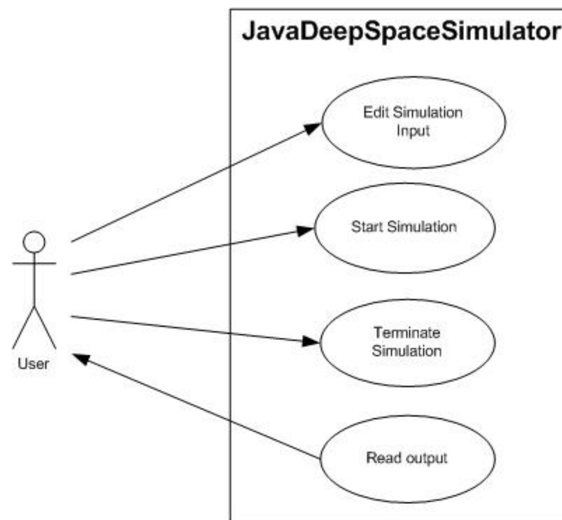

Figure 10: The user's UML use cases

The use cases shown in Figure 10 are commented in the following paragraphs, with a list, for each of them, of the normal and alternative flows of events.
For each of the alternative events, an explanation of how the system, or the user behaves in that situation is provided.

2.2.1 Edit simulation input

Normal flow of events:

1. The user starts a text editor application in order to create the XML input file;
2. The user starts editing the XML file;
3. The user finishes editing the file;
4. The user saves the XML file and closes the editor.

2.2.2 Start simulation

Normal flow of events:

1. The user opens the Linux shell;
2. The user types a command to compile the JBDS simulator;
3. The user runs the JBDS simulator.

Alternative flow of events:

1. The XML file is not well formed or not correct.
    a. The application displays an error message and forces termination.
2. The connection with STK fails.
    a. The application displays an error message and forces termination.

2.2.3 Terminate simulation

Normal flow of events:

1. The user opens the Linux shell;
2. The user terminates the application.

2.2.4 Read output

Normal flow of events:

1. The user browses for the folder where the output file is stored
2. The user opens the output file with a text editor

Alternative flow of events:

1. The simulation didn't produce any output data

a. The simulation was not run correctly, and the user is obliged to run it again in order to obtain the result file.

2.3 The JBDS simulator class and package diagram

The JBDS simulator package diagram is shown in Figure 11, whereas the class diagram is shown in Figure 12: The JBDS simulator UML class diagram.

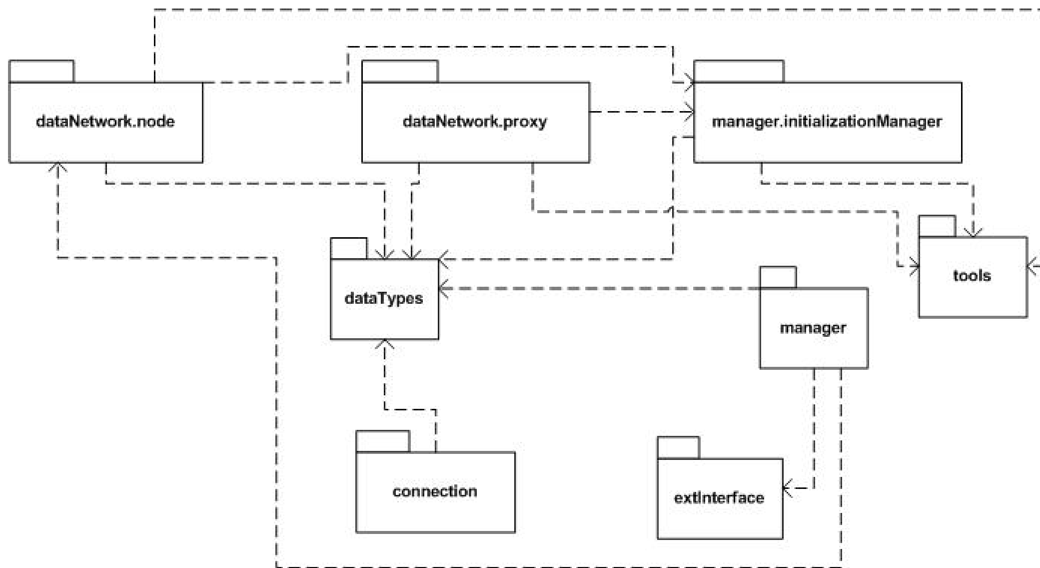

Figure 11: The JBDS simulator UML package diagram

The package diagram shows the dependencies between packages, whereas the classes' diagram shows the classes contained inside the packages and their interactions. The classes and some details about their implementations are given in chapters 3, 4 and 5 more in detail.

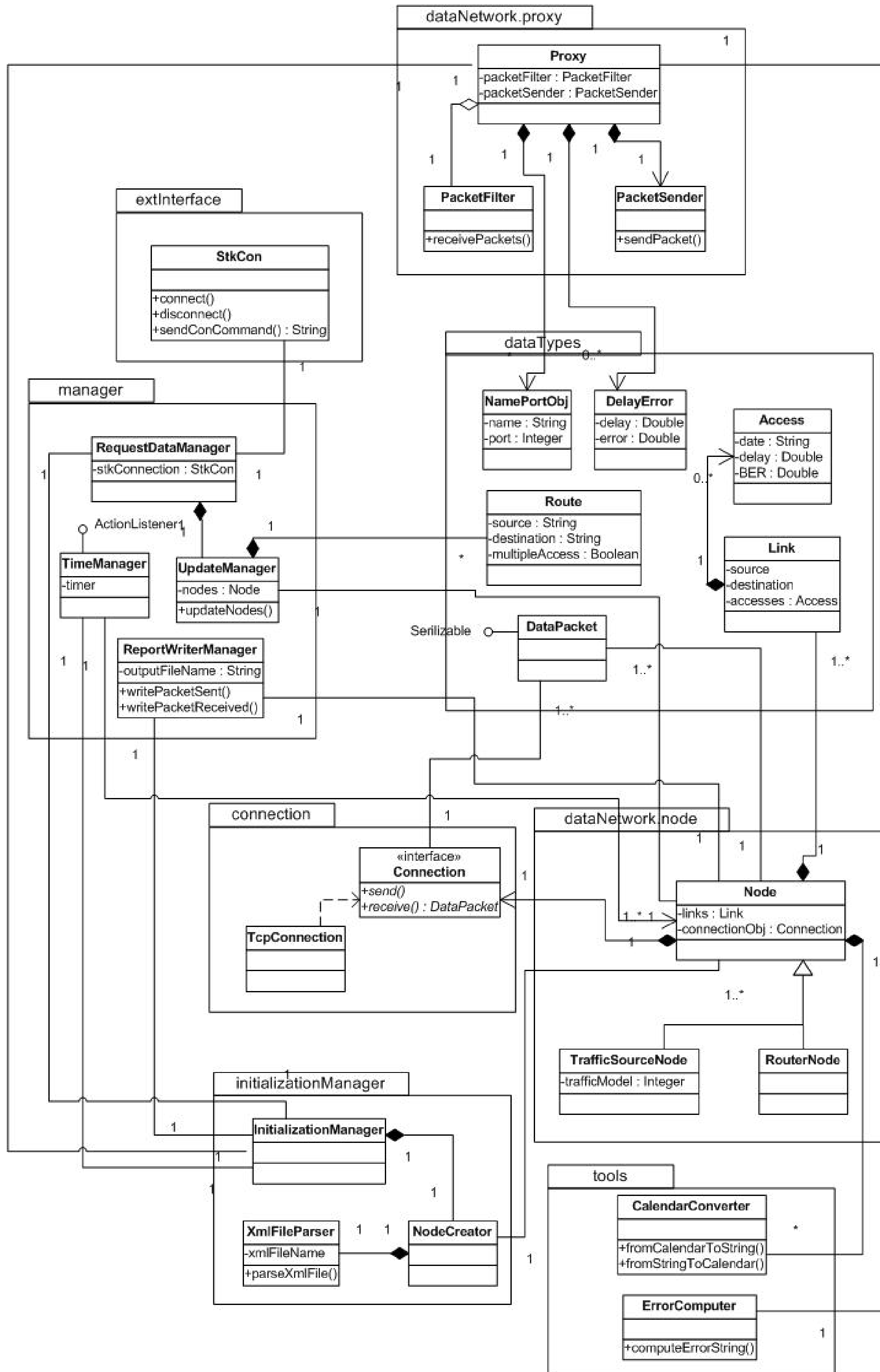

Figure 12: The JBDS simulator UML class diagram

2.4 The JBDS simulator sequence diagram

Figure 13: The JBDS simulator UML sequence diagram shows a detailed UML sequence diagram, which explains the steps performed during the initialization of the simulation and the interaction between the different components.

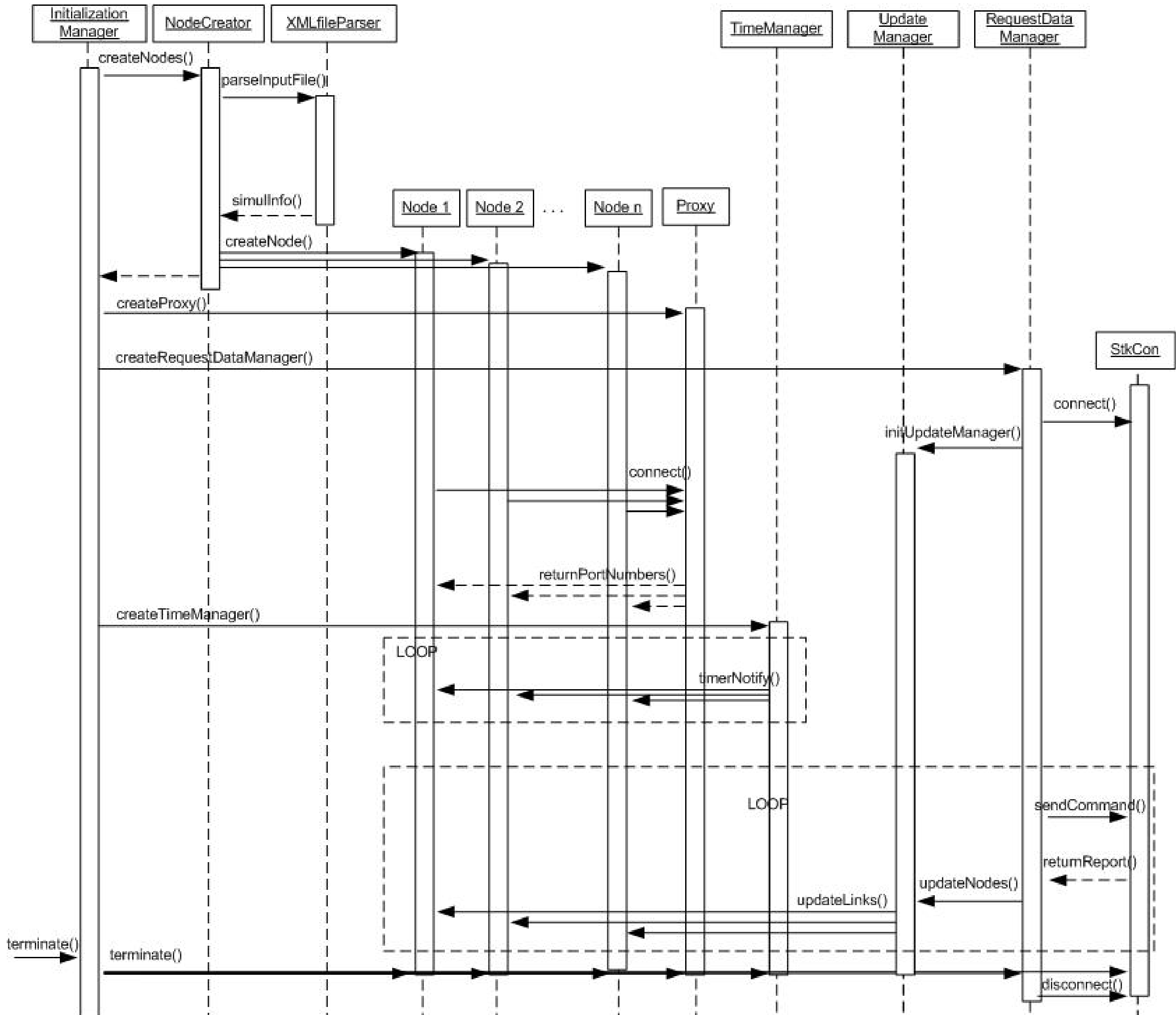

Figure 13: The JBDS simulator UML sequence diagram

All the details about the components' message passing will be described in the following chapters.

3 The Manager

The manager performs the core functionalities of the software. It is responsible for:

- The initialization of the simulation.
- The input of data from STK.
- The updating of nodes' information
- The control of simulation time.
- The output of the simulation.

The manager is the module that is supposed to master all these tasks, but it has been modularized, thus each of the modules is actually responsible for just one of these tasks.

The manager's sub-modules and their interfaces are shown in Figure 14. All the interfaces between sub-modules are displayed using arrows; each of those arrows represents communication and data exchange that takes place in the management network. The sub-modules that make up the manager module are the following:

- The Initialization Manager;
- The Request Data Manager;
- The Node Update Manager;
- The Time Manager;
- The Report Writer Manager.

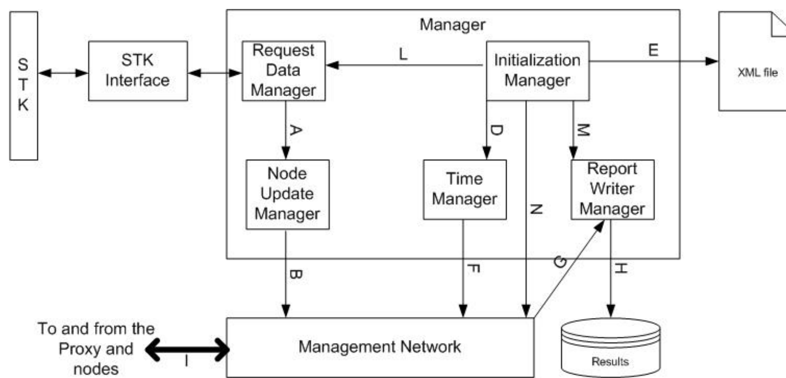

Figure 14: The Manager's internal architecture

3.1 The Initialization Manager

The Initialization manager is the module that deals with the initialization of the simulation.

Figure 14 shows how this module interfaces the other manager's modules (the D, L, M and N interfaces) and the external XML file (the E interface).

The Initialization Manager is responsible for all the tasks that have to be performed when the simulation starts. Most of the times it doesn't perform these tasks itself but it loads other modules that will be responsible for them. Figure 14 shows the E interface that lets the Initialization Manager's module load the configuration information from an XML file edited by the user. After loading the XML file a sub-component of the Initialization manager parses it and the output of this process is sent through the Management Network to the second sub-component: the Node Creator, as shown in Figure 15.

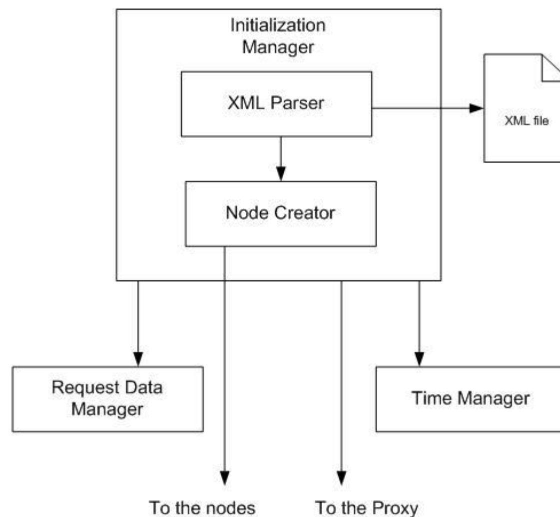

Figure 15: The Initialization Manager module

The Node Creator is responsible for the creation and initialization of the nodes, thus it directly communicates with them. After the Node Creator completes its task the Initialization Manager loads and starts, the Request Data Manager, the Time Manager and the Proxy modules. The Time Manager is the last component to be initialized. When the Time Manager is created the simulation begins.

The initialization phase is shown in the sequence diagram in Figure 13. At first the InitializationManager initialises the NodeCreator, which parses the XML input file and then, with the information gathered from the input file it creates the nodes. After creating the nodes the Initialization Manager initializes the Proxy, the RequestDataManager and the TimeManager (with the createProxy(), createRequestDataManager () and createTimeManager() methods).
Since it initializes all the simulation's modules, InitializationManager is the only class in the project that contains the main method.
All the manager's classes are contained in a package called manager shown in Figure 12, but the InitializationManager class, together with its support classes (NodeCreator and XmlFileParser) are included in a sub-package called initializationManager.

3.2 The Request Data Manager

The task of the Request Data Manager is polling STK for information. It initializes the connection with the simulator (STK), and periodically asks for "fresh" information.
The interface between STK and the Manager is bidirectional (interface B in Figure 9): the Manager asks for information providing a command string, and the STK interface returns the information required.
As already mentioned, the methods belonging to the StkCon class that are used in this module are connect ( ), disconnect ( ) and sendConCommand ( ). This last method is needed in order to be able to send the appropriate commands to STK and obtain the needed information.
The flow of execution of these commands is shown on the right in Figure 13: the RequestDataManager establishes a connection with STK using StkCon and then, inside a loop sends commands and receives data. Some commands are used just during the initialization phase; some others are used in the whole simulation.
A very relevant command is the initialization of the animation.
The idea for this thesis, was to use STK, not just as a source of reliable data, but also as a way to graphically check the progress of the simulation, since, as already metioned, STK provides two graphical interfaces (the 2D and the 3D).

For this reason, one of the first commands sent to STK, is:

```
stkConnectObject.sendConCommand("Animate Scenario/"+ scenarioName + " Start End");
stkConnectObject.sendConCommand("Animate Scenario/"+ scenarioName + " TimeStep 10");
```

The first line of code asks STK to start the animation for the scenario identified by the scenarioName String, whereas the second sets the animation TimeStep, i.e. the animation speed.

Unfortunately the hardware requirements for STK are particularly high, thus running the animation and the polling STK for information at the same time tends to slow STK so much that, with the machine used for the development of this thesis, it becomes hard to see in real time the progress of the animation.

Before animating the STK simulation, it is necessary to load the scenario and to set the main features of the output data.

It is possible to create the scenario, all the objects involved and their interactions sending command to STK from an external application using STK/Connect, instead of using the STK user interface. However, this choice was rejected because of the complexity of some of the commands and for the complexity of the scenario to create; as a matter of fact, the command list doesn't provide some of the necessary commands.

Hence, the scenario was created using the STK user interface, saved and loaded from the simulator. Before loading the scenario is therefore necessary to check whether it is not already loaded, this is particularly important because if the scenario is already loaded and an external application tries to load it a second time STK could display an error message.

As already mentioned, at this stage (after loading the scenario), it is necessary to ask STK to provide the output in the desired format. An example of this could be the date format. By default, STK manages dates using the Gregorian UTC format (for example: "1 Jul 2005 12:00:00.000"), for ease of comparison between dates it was necessary to change it into the following format: YYYY/MM/DD Time ("2005/07/01 12:00:00.000"). This is a STK/Connect parameter; thus, it has to be set from the external application, and within the Connect session.

Two of the other important commands used, are the ones that return the range (distance) between objects (fro example from a Facility to a Satellite) and link budget

information (for example the BER). These commands are shown below, the first one polls for the report concerning Azimuth[8], Elevation and Range (AER), but just the range value is extracted from the report and actually used for the simulation.
The range value (which is the distance between objects) is particularly relevant for the JBDS simulator, because it will be used to compute the delay associated to a data link.

```
String range = stkConnectObject.sendConCommand("GetReport */"
                                + route.getSourceType()
                                + "/"
                                + route.getSourceName()
                                + "/Transmitter/Transmitter"
                                + route.getSourceName()
                                + " \"AER\" /"
                                + route.getDestinationType()
                                + "/"
                                + route.getDestinationName()
                                + "/Receiver/Receiver"
                                + route.getDestinationName() + " \""
                                + start + "\" \"" + stop + "\" 60");
```

This command specifies the source and destination objects, within the scenario and asks STK to compute the AER report for them; the start and stop Strings indicate the start and stop times, and 60 the time-step between measurements.
The time step is quite relevant because the command is actually asking STK to report AER values for every simulation minute.
The start string at the beginning of the simulation corresponds to the simulation start time provided by STK at initialization time, whereas the stop string is computed from the start string with the addition of an interval that can be set according to the desired time granularity. By default, the interval was set to one simulation day, which means that the report generated and sent by STK will contain the AER values computed at each simulation minute (the 60 in the command), for the total duration of one simulation day. After the first request the content of the start string will become the one of the stop string and the stop string value will incremented by one day, for the next request of data.

[8] Azimuth: "*The horizontal angular distance between the vertical plane containing a point in the sky and true south.*"[54].

The code to poll for a link budget report is very similar to the previous one:

```
String budget = stkConnectObject.sendConCommand("GetReport */"
                                + route.getSourceType()
                                + "/"
                                + route.getSourceName()
                                + "/Transmitter/Transmitter"
                                + route.getSourceName()
                                + " \"Link Budget\" "
                                + route.getDestinationType()
                                + "/"
                                + route.getDestinationName()
                                + "/Receiver/Receiver"
                                + route.getDestinationName() + " \""
                                + start + "\" \"" + stop + "\" 60");
```

The difference between the two commands is, of course, the name of the report (AER in the first command and Link Budget in the second).
The reply that comes from STK to such requests is given as a String; the following is a sample (part of an AER report):

```
"Time (YYYY/MM/DD)","Azimuth (deg)","Elevation (deg)","Range (km)"
2007/01/01 05:12:26.235,322.596,0.000,2264.630913
2007/01/01 05:13:26.000,338.033,2.399,2274.570585
2007/01/01 05:14:26.000,352.419,3.904,2437.709547
2007/01/01 05:15:26.000,4.390,4.487,2722.843548
2007/01/01 05:16:26.000,13.772,4.412,3092.048249
2007/01/01 05:17:26.000,20.988,3.952,3514.586225
2007/01/01 05:18:26.000,26.564,3.289,3969.343258
2007/01/01 05:19:26.000,30.938,2.533,4442.657255
2007/01/01 05:20:26.000,34.432,1.743,4925.784968
...
```

For the JBDS simulator purposes two very important data contained in the previous report will be extracted. The first datum is the range (that, as already mentioned, will be used to compute the delay), and the second datum is the relative time.
It is possible to obtain the output of these commands in a text file (using the command Report instead of GetReport and specifying the location where to store the file and the type of file to generate). However, the choice of one approach over the other really depends on the purposes of the external application. Thus, after performing a

comparison between the speeds of one approach and the other the return String was preferred to the output file.
As shown in Figure 14, the Request Data Manager module produces some data output that will be used by another module: the Node Update Manager.
The RequestDataManager class was implemented as an extension of the Thread class. Thus it contains a run method which is supposed to periodically send requests to STK and pass the information obtained to the Node Update Manager module (represented by the UpdateManager class). To avoid useless computations, the RequestDataManager thread sleeps for some time before forwarding a new request to STK.

3.3 The Node Update Manager

The Node Update Manager inputs the reports generated by the Request Data Manager and updates the Node information in order to make them use this information during the simulation.
The execution flow is shown in Figure 13: the RequestDataManager initializes the UpdateManager module during the initialization, and then after receiving data from STK it asks the UpdateManager to update the nodes. Figure 13 shows a loop, which means that these operations are done continuously till the end of the simulation.
The datum passed from the RequestDataManager to the UpdateManager is a string, thus the UpdateManager relies on a tool that tokenizes the input string received from the RequestDataManager and extracts the useful information from the STK report.
The Node Update Manager creates data structures that will be used by the nodes to read the information related to a particular Link at a certain time. The type of object used to store these data is an Access object which contains one field for the date, another field for the delay, and another one for the error rate. Thus the output of this module is an update in the collection of Access objects for each of the Links.

3.4 The Time Manager

All the nodes in the simulation should be synchronized, otherwise if the information used by one of the nodes to communicate with the others is not up-to-date, the output

of the simulation could show a strange behaviour (for example a node could send data when the destination is not available). The Time Manager is a component that overcomes to this problem. It is loaded at the beginning of the simulation by the Initialization Manager, and then it deals with signalling the nodes all together and at regular intervals (as shown in Figure 13). It works with a wait and notifyAll ( ) system, this way all the nodes are notified at the same time.

By default, the JBDS simulator notifies the nodes every second; however it is possible to change this value depending on the user's desired simulation speed.

3.5 The Report Writer Manager

The Report Writer Manager module deals with the output of the simulation. The simulation produces an output text file, where every node is identified by its name. When a node sends or receives a packet, it records that event in the text file. The file generated is called "Report.txt", and a sample of the content looks like the following:

```
0 0 s 2007/1/1 5:20:0.000 1153142834462 s:0  d:3
0 0 r 2007/1/1 5:21:0.000 1153142835875 s:0  d:3
0 0 s 2007/1/1 5:22:0.000 1153142836696 s:3  d:4
0 0 r 2007/1/1 5:22:0.000 1153142836796 s:3  d:4
0 1 s 2007/1/1 5:22:0.000 1153142851247 s:1  d:3
0 1 r 2007/1/1 5:23:0.000 1153142852508 s:1  d:3
0 1 s 2007/1/1 5:23:0.000 1153142862192 s:3  d:5
0 1 r 2007/1/1 5:23:0.000 1153142862343 s:3  d:5
0 2 s 2007/1/1 6:0:0.000 1153142876984 s:2  d:5
1 2 s 2007/1/1 6:0:0.000 1153142876984 s:3  d:5
2 2 s 2007/1/1 6:0:0.000 1153142876984 s:3  d:5
0 2 r 2007/1/1 6:0:0.000 1153142877104 s:3  d:5
1 2 r 2007/1/1 6:0:0.000 1153142877124 s:3  d:5
2 2 r 2007/1/1 6:0:0.000 1153142877174 s:3  d:5
1 1 s 2007/1/1 6:1:0.000 1153142878416 s:1  d:3
1 1 r 2007/1/1 6:1:0.000 1153142879627 s:1  d:3
1 1 s 2007/1/1 6:2:0.000 1153142881560 s:3  d:5
1 1 r 2007/1/1 6:2:0.000 1153142881660 s:3 d:5
3 2 s 2007/1/1 6:27:0.000 1153142904774 s:2  d:5
3 2 r 2007/1/1 6:27:0.000 1153142904884 s: 2  d:5
```

In the scenario that produced the previous output file there are six nodes called 0, 1, 2, 3, 4 and 5. There are two possible roles for the nodes in the simulation, they can be either traffic source or routers (these nodes can't generate traffic, but they can receive packets and forward them). In this particular scenario, the traffic sources are identified by the numbers 0, 1 and 2, whereas the three router nodes are identified by 3, 4 and 5. A single line in the output file should be read as follows:

```
0 0 s 2007/1/1 5:20:0.000 1153142834462 s:0  d:3
```

Figure 16: A sample line of the output file

The first "0" is the packet's sequence number whereas the second "0" indicates the name of the traffic source.

If, as in the example, there is more than one traffic source, then there will be more than one packet with the same sequence number. Recording the source it is the only way to distinguish between them.

The "s" stands for "sent", which means that in that moment a packet created by the source, is being sent or leaves the (sending) router's queue and is forwarded to the destination or to the next hop.

On the other hand "r" stands for received, which could mean both that, the packet has been received by the destination, or that it has been queued up at the (receiving) router.

What follows is the date; in the particular example it is "2007/1/1 5:20:0.000". This date is the current simulation time whereas the following value: "1153142904884" is the system clock's value at the time the event is reported, this value was added in the output file in order to have a more precise estimation of the moment when the packet is actually sent or received.

In the previous example, according to the simulation time, it seems that packets are sent very rarely (roughly one packet per minute). This can be modified very easily, by changing the time granularity of the STK reports and/or changing the TimeManager time step. In general, if the simulation lasts for a long time it is probably better to use larger time steps; the drawback to such approach is that the error rate and the distance (therefore the delay), could have a sudden increase/decrease in value

The last two sub-strings in Figure 16 indicate the packet's source and temporary destination (not the destination but the next hop). For example in Figure 16 the packet has been sent by 0 and received by 3, while in this case:

0 0 **r** 2007/1/1 5:21:0.000 1153142835875 s:0 d:3

Packet 0 is received by 3 and was sent by 0 (the node indicated after the "s:" string is not the traffic source, but it is the last node in the path that forwarded the message, however, it can correspond to the traffic source, as in the example). Thus, in this last example 3 is the temporary destination for the packet and will have the task to forward it to the next hop. This is the reason why, keeping reading through the output file, the following line can be found:

0 0 r 2007/1/1 5:22:0.000 1153142836696 s:3 d:4

In this case node 3, who has received packet "0 0" at time 1153142835875, is forwarding it to the destination, which is node 4. It is not actually possible to infer from this line that the destination for the packet is actually node 4. The only way to understand it is to browse for other lines in the output file that report information related to packet "0 0".

As shown in Figure 14, the Report Writer Manager module is initialized by the Initialization Manager at initialization time using the M interface. It is at this time that the Report.txt file is created together with the output streams used to write into the file.

4 The Simulation Nodes

Every entity in the STK scenario is represented by an object of type Node. As shown in Figure 9, all the nodes in the simulation are initialized and uploaded by the Manager, but they never directly query the Manager for information. Interface C in Figure 9 shows that the link is unidirectional. Thus, the nodes don't really directly use the Management Network but exclusively the Data Network.

Since every object in the STK scenario has a name, all the nodes in the JBDS simulator topology must have a name, which corresponds to the one in the STK scenario. As explained in paragraph 3.5, the Node's names are also used as identifiers in the output file.

A Node can be a Facility, a Satellite or a Traffic Source. Facilities and Satellites are objects inside the STK scenario, whereas a Traffic Source is an entity that represents an endpoint that periodically produces output data. This entity has been designed, in order to confer more realism to the simulations. For instance, assuming to have, in the scenario, a ground antenna (for example Goldstone, Canberra or Madrid) that communicates with a satellite, then it would be more realistic not to assume that the antenna is the real source of traffic or destination for the information coming from or going to the satellite; in other words, a ground antenna shouldn't be considered an endpoint. These kinds of Facilities are supposed to receive and send traffic data, but, of course, they can't create traffic. Hence, there should be a traffic source: for example it could be a computer that from a space centre communicates to the antenna using the wired network.

For this reason, in the JBDS simulator the link between Facilities and Traffic Source is assumed to have a constant delay of 100 ms. However, it is worth underlining that there could be some kind of Facilities able to produce data traffic. It is the case of, for example, the Lunar objects that belong to the scenarios used for the evaluation of this framework (chapter 6).

In general there is a distinction between two types of nodes:

1. The first type of node behaves as a traffic source and it could also be a sink for data (ultimate destination for packets).
2. The second type of node behaves as a router, and/or as a sink for data.

The first type uses a traffic model to generate packets to be sent to certain destinations. Some of these nodes could also act like sinks for data, which means that they could be the destination for some of the packets.

A traffic model is a model that describes the frequencies and time periods for generation of packets by a Traffic Source. The traffic model could be random or have particular time constraints. For example, a Traffic Source could produce traffic just in particular periods of the day, or of the year, or at random intervals.

The second type of node is like a router; it is not allowed to produce traffic data and can only receive and forward packets. These nodes have an internal buffer where every incoming packet is queued, in order to be forwarded to the next hop in the path. This kind of nodes could also represent a sink for data. In that case an incoming packet is not queued, but it is discarded (after reporting the receipt to the output file).

This difference is reflected in the code with the use of two different classes, children of a super-class which implements their common behaviour. The super class is Node and the two sub-classes are TrafficSourceNode and RouterNode. Figure 12, on the right part, shows these three classes and their relationships (contained in the dataNetwork.node package).

As the Node class is supposed to implement the general behaviour, it contains all the methods that let the manager update the information relative to the links. This collection of data is stored inside a Java Vector, containing objects of type Link.

The objects of type Link represent, as the name suggest, a single connection that the Node has with another Node. The different fields in a Link object are:

- The link's source. This field corresponds to the Node's name; it is useful especially for the UpdateManager class.
- The link's destination. This field represents the link's destination name.
- The Access Vector, which contains the Access objects created by the UpdateManager class using the information retrieved from STK.
- The multipleAccess field is a Boolean that will be described in more detail later on.

4.1 A TrafficSourceNode

A TrafficSourceNode represents a Node that can generate traffic using a traffic model. In this document traffic source will be used to refer to a TrafficSourceNode. Every traffic source can have many available destinations, and each of the packets it generates can have just one possible destination. In the simplest scenario possible a TrafficSourceNode would iterate through all its possible destinations, generate one packet for each destination and forward it to the next hop in the path that leads to the final endpoint.

The most important difference between a TrafficSourceNode and a RouterNode is the lack of a buffer (queue) in the TrafficSourceNode. As a matter of fact, this kind of node doesn't need to buffer any of the packets it receives, because it behaves as traffic sink; thus, after receiving a packet and report the event to the ReportWriterManager, the packet is discarded.

Every node has a direct socket connection with the other nodes it communicates with. The implementation of the class TrafficSourceNode relies on threads for sending and receiving packets, this last feature, implies to have a different receiving threads for each of the possible links. In order to avoid the excessive slowdown of the application, every node has just a sending thread.

In a TrafficSourceNode, the sending thread iterates through the possible destinations and sends a packet to all of them, taking care of three very important aspects.

- The traffic model
- The visibility (or access)
- The multiple access order

The traffic model can be a constraint if, for example, a traffic source is supposed to generate packets just in particular moments or periods. This constraint applies just for this type of node, while the other two affect the sending process also in router nodes.

The visibility or access is determined on the basis of the information retrieved from STK.

The multiple access is a Boolean property of a link which states that, when a link is not available (when there is no visibility) but it has the multiple access property set to true, then either one of the other available links departing from the same node, that

have multiple access set to true, can be chosen to reach the same destination. It is like an alternative path to the same destination. This property can be extremely useful in a scenario where a traffic source, which can be a single computer in a space centre, has to communicate with a satellite, the chosen ground transmitter the one that is in view of the satellite in that particular moment.
The JBDS simulator implements only the situation where a node has just one group of links with multiple accesses. It could have been interesting to implement also the case where a node can have more than one set of links with multiple accesses, maybe distinguishing them with a short integer instead of using a Boolean.

4.2 Routing

Since it can happen that a packet destination is farther than one hop from the traffic source, then the packet should be routed to the right direction. A node is not able to compute the best path, thus the XML input file should provide for every node the available destinations.

4.2.1 The XML input file

An XML file could be defined in its structure and list of legal elements by two types of documents
[51]:

- A DTD (Document Type Definition)
- An XML Schema

DTDs are easier to create compared to XML schemas, however XML schemas are more powerful for description of complex XML files. For the purpose of this thesis the descriptive syntax of DTDs was considered appropriate for the definition of the XML input file.
The DTD that describes the XML input file is the following:

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

```
<!ELEMENT dtn (router+)>
<!ATTLIST dtn name CDATA #REQUIRED  path  CDATA #REQUIRED >
<!ELEMENT router (address, routes+)>
<!ATTLIST router name  CDATA #REQUIRED  type  CDATA #REQUIRED >
<!ELEMENT address (#PCDATA)>
<!ELEMENT routes (route)>
<!ELEMENT route (destination)>
<!ATTLIST route network  CDATA #REQUIRED  gateway  CDATA  #REQUIRED
                gatewayName CDATA #REQUIRED  gatewayType  CDATA #REQUIRED >
<!ELEMENT destination (#PCDATA)>
```

The following is a sample of a very simple XML input file for the application:

```
<jbdss name="Scenario1" path="D:Program Files/AGI/STK6.0/WorkFiles/">
        <router name="Facility1" type="Facility">
                <address>facility1.dsn.earth.simulation</address>
                <routes>
                        <route network="satellite.simulation" gateway="satellite.simulation"
                        gatewayName="Satellite1" gatewayType="Satellite">
                                <destination>Satellite1</destination>
                        </route>
                        <route network="earth.simulation"  gateway="earth.simulation"
                        gatewayName="TrafficSource" gatewayType="TrafficSource">
                                <destination>TrafficSource</destination>
                        </route>
                </routes>
        </router>
        <router name="TrafficSource" type="TrafficSource">
        <address>earth.simulation</address>
        <routes>
                <route network = "satellite.simulation"  gateway = "facility1.dsn.earth.simulation"
                 gatewayName="Facility1" gatewayType="Facility">
                        <destination>Satellite1</destination>
                </route>
        </routes>
        </router>
        <router name="Satellite1" type="Satellite">
                <address>satellite.simulation</address>
                <route network="earth.simulation"  gateway = "facility1.dsn.earth.simulation"
                 gatewayName="Facility1" gatewayType="Facility">
                        <destination>TrafficSource</destination>
                </route>
        </router>
</jbdss>
```

In this very simple example, the Scenario is called “Scenario1” and the name is an attribute of the jbdss (JBDS simulator) tag, whereas the path where the scenario is stored is another attribute of jbdss tag, called path.
Each of the objects in the simulation is represented by a router tag. In the example proposed there are three objects: TrafficSource, Satellite1 and Facility1. TrafficSource, as mentioned before, doesn’t belong to the STK scenario but it has meaning just inside the JBDS simulator.
The address that identifies the object in the simulation is studied to be compatible with the bundle protocol’s notion of regional networks (the thesis, at the beginning was supposed to perform a comparison between bundle and TCP). Thus, each of the sub-strings in the address identifies a sub-network of the general network which is “simulation”. However the concept of regional network is disappearing with the new DTN implementations.
The information provided in the XML input file is necessary to determine the routes that lead to a destination. In the previous XML file for example, TrafficSource communicates with Satellite1 through Facility1.
The route and destination tags have a very important role. During the initialization of the nodes, the XML file is parsed and all the routes are stored in a data structure. Each of these routes represents a node that is one hop distant from the router node. Whereas the internal tag <destination> indicates an endpoint which could be more than a hop distant from the source node, and that can be reached via the gateway indicated in the route tag with the gateway attribute. For instance, in the following lines:

```
<router name="TrafficSource" type="TrafficSource">
        ...
        <route ... gatewayName="Facility1" gatewayType="Facility">
                <destination>Satellite1</destination>
        </route>
        ...
</router>
```

TrafficSource has a direct link with Facility1, and uses Facility1 to reach Satellite1, which is the destination.

Each node keeps a Hashtable object which contains all the possible destinations that a node can reach. Every destination is represented by its name, which is inserted as a key in the Hashtable, together with the id of the link that leads to that destination. In the previous example the Hashtable at the TrafficSource node would look something like Table 2.

Key	Value
Facility1	0
Satellite1	0

Table 2: Example of an Hashtable used for routing purposes

In the example, both the destinations have the same id because the path that leads to the destination, which is Satellite1 goes through the gateway called Facility1.

5 The Proxy

The proxy was the component of the software that required most time and effort.
In the initial stage of the project, the real aim of the dissertation was to explore the differences in performance between the Bundle protocol and TCP, using the STK simulator environment. Unfortunately STK doesn't simulate network communication, thus the JBDS simulator was implemented in order to supply to this deficiency.
The first architecture designed for the data network, i.e. for the communication between nodes, looked like the one shown in Figure 17.

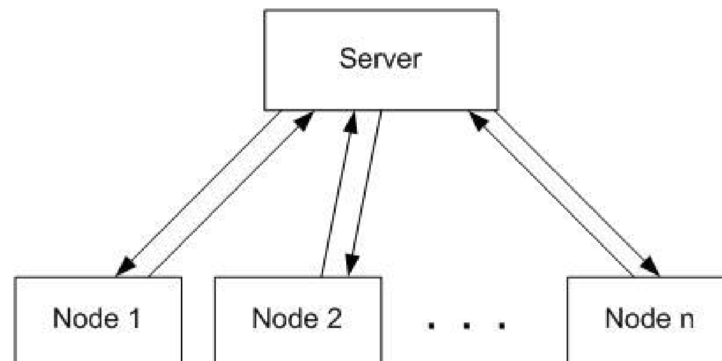

Figure 17: Client-Server architecture

It was a client-server architecture, the clients represented the nodes in the simulation and the server had the task of receiving all the packets sent by the nodes, wait for some time (delay time provided by STK), and finally send the packet to the correct destination. All the nodes and the server application were designed to run on the same machine. This architecture was not accurate for two main reasons:

1. There was no direct connection between the source and the destination;
2. The delay was just simulated and not effective, which could lead to having no real interesting results concerning the poor performances TCP exhibits in deep space environment.

A new solution had to be found.
Unfortunately the choice of using Java, didn't help; as a matter of fact, it is a high level language that hides all the real socket and stream implementation, thus the first

idea of overcoming the problem changing the implementation of a Java Socket or implementing a FilterInputStream (in order to apply the delay to the packets just before their reception) or a FilterOutputStream (or before their delivery) had to be rejected.

FilterOutputStream and FilterInputStream are still too high level classes, and TCP would not be affected by the delay anyway.

Moreover, in this case the software wouldn't work properly because of another very important reason.

Assuming that a node sends a packet at time t, then it will be delayed by the filter for the delay time (t_1) and then sent to the destination (it would arrive at time $t + t_1$). However, assuming that the same node tries to send a new packet at time $t + 1$, this last packet will be delayed longer than the actual delay time because it will have to wait till the first packet is sent to the destination. The same thing would happen for the FilterInputStream. An alternative to that would be using a pool of threads, in order to delay the packets at the same time and not one after the other. Unfortunately this solution is very expensive when there are a big number of packets involved. Moreover the software already makes use of a number of threads, which would make this approach inconvenient.

The solution that was finally chosen was to implement of a Proxy able to capture the packets at the IP layer, delay them, apply an error rate as a byte string on them (according to the information received from STK) and then send the packet to the intended destination. Still the design involved the use of a single machine.

The Proxy module was designed to be completely transparent to the nodes, as the one in Figure 18.

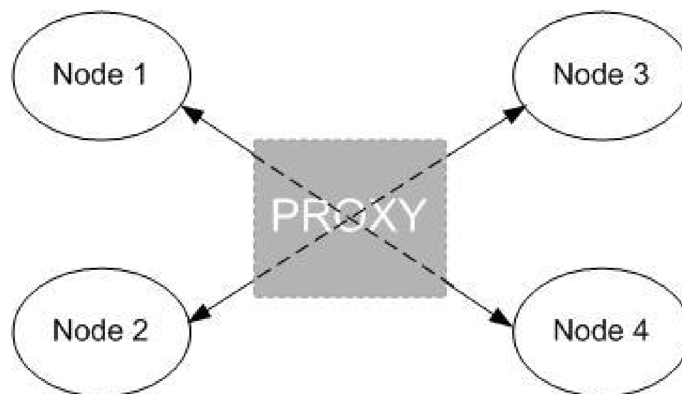

Figure 18: Proxy architecture

This design was thought to be the most appropriate in order to evaluate TCP's performances.
During the initialization phase the Proxy acts like a server for the nodes. All the nodes connect to the proxy and communicate it the names of the nodes they want to connect to, the proxy computes these data and sends back to the nodes the information needed in order to establish a connection with the right nodes. This phase is shown in Figure 13, the nodes connect to the Proxy using connect() and they receive the port numbers of the nodes they want to connect to.

5.1 The implementation

Sending and receiving network packets belonging to protocols not processed by the operating system is not possible in Java, but it becomes feasible if Java methods "wrap" native methods.

5.1.1 JNI (Java Native Interface) and Raw Sockets

The Java Native Interface (JNI) is a programming framework that allows Java code to call and be called by native applications and libraries written in other languages, such as C, C++ and assembly. This approach is necessary in order to run applications specific for a type of hardware or operating system platform.
For the purpose of this thesis JNI was necessary in order to be able to use raw sockets. Raw sockets are a special type of socket used to access the packet headers on incoming and outgoing packets. This is the real difference between a raw socket and a normal Java socket; the latter just receives the payload of the packet.
In C, in order to use sockets two libraries have to be included:

```
#include        <sys/types.h>
#include        <sys/socket.h>
```

A socket is initialized in the following way:

```
int socket(int domain, int type, int protocol)
```

The code needed to initialize a raw socket looks like the following:

```
socket (PF_INET, SOCK_RAW, IPPROTO_TCP);
```

PF_INET stands for protocol family, opposed to AF_INET that stands for address family. SOCK_RAW indicates that this is a socket of raw type and finally IPPROTO_TCP means that the protocol used is TCP. Other options are available for the last parameter: for example IPPROTO_IP, IPPROTO_UDP or IPPROTO_ICMP.
Due to the fact that raw sockets allow users to craft packet headers themselves, their power can be abused to perform IP address spoofing for Denial-of-service attacks.
When Windows XP was first released in 2001 with raw socket support implemented into the Winsock interface, the media attacked Microsoft saying that raw sockets are only used by hackers. Three years later, Microsoft silently deprive of strength and efficiency Winsock's raw socket support in a non-removable hotfix[9] and offered no further support or workarounds for applications that used them.
A very small number of Windows applications, in general, make use of raw sockets. The Windows XP implementation of TCP/IP still supports receiving traffic on raw sockets. However, the ability to send traffic over raw sockets has been restricted in two ways [36]:

- TCP data cannot be sent over raw sockets.
- UDP datagrams with invalid source addresses cannot be sent over raw sockets. The IP source address for any outgoing UDP datagram must exist on a network interface or the datagram is dropped.

5.1.2 The APIs used

It was possible to find available online two extremely useful open source APIs:

- RockSaw [37];
- Virtual Services TCP/IP, also called Vserv [38].

---

[9] Hotfix is a Microsoft term for a bug fix, which is accomplished by replacing one or more existing files in the operating system or application with revised versions [39].

The first one allows to create a raw socket using JNI methods. It actually contains just one Java class called RawSocket, other two classes are used as an example of how to interface the API (in particular there is an example of a Ping application). This API comes with a .dll library for Windows and a .so shared object for Linux.
The second API, Vserv, was important in order to read the different fields in the IP or TCP headers. This API turned out to be very useful to recognize the packets received by the Proxy.
The API contains classes like TCPPacket, UDPPacket and IPPacket. Each one of these classes have appropriate methods that enable the user to read the different fields in the header, in particular: getDestinationPort( ), getSourcePort( ) were useful to distinguish between the packets created by the JBDS simulator and the packets that are not.
The Proxy is made up of three components (and three classes contained in the dataNetwor.proxy package, as shown in Figure 12): the Proxy, the Packet Filter and the Packet Sender.

5.1.3 The Packet Filter

The first idea was to work using Windows, mostly because the STK's version provided to Trinity College is supposed to run in Windows. However, after the first experiments with JNI and raw sockets, and above all, after reading about the modifications to Winsock library[10], it was necessary to start using Linux.
At the beginning, mostly because of inexperience with JNI, many problems were experienced, especially related to .dll libraries. The JVM (Java Virtual Machine) seemed not able to load/link the .dll file throwing an UnsatisfiedLinkError.
However, even after configuring the environment and providing the requested libraries, the code still had problems. There was no UnsatisfiedLinkError anymore, but the problem was noticed to appear when trying to use the bind ( ) method in C on a raw socket.

[10] Short for Windows Socket, Winsock is an Application Programming Interface (API) for developing Windows programs that can communicate with other machines via the TCP/IP protocol. [40]

Hence, after reading about the new security policies for Windows XP, the problems was thought to be due to the restriction in power for the Winsock library in Windows XP.

In Linux, raw sockets work properly, and this led to a quicker improvement in the implementation. It became possible to code the first Packet Filter and test simple applications.

The first Packet Filter implemented was very simple. It could listen in on TCP traffic and filter the packets sent by a client-server Java application. The source's and destination's ports were used to recognize the origin of the different packets. If the filter was not applied on the incoming packets, a big number of packets external to the application would be received.

The implementation of the filter is the one that follows:

```
do {
      socket.read(InetAddress.getByName("127.0.0.1"), recvData);
}
while(recvPacket.getProtocol() != RawSocket.getProtocolByName("tcp")
      || recvPacket.getSourcePort()!=1234
      || recvPacket.getDestinationPort()!=1235
      );
```

The server's port number was 1234, whereas the client's port number was 1235. RecvData is the parameter provided to the read method, it is a byte array that represents the payload of the IP packet received. On the other hand, recvPacket is an object of type TCPPacket, that lets the programmer read inside the TCP header fields. In the example the fields used to recognize the packet are:

- Protocol (which is a field in the IP header), which was supposed to be TCP;
- Source port;
- Destination port.

Hence, the socket kept reading till it didn't receive a packet that satisfied the given conditions.

The initial implementation of that filter was the basis on which the Proxy was coded.

5.1.3.1 How to distinguish between different simulation nodes

The way that was thought to be the most realistic in order to distinguish between different nodes in the simulation and between packets created by the JBDS simulator and external ones, was to use different IP addresses. As mentioned before, the application was designed to run on the same machine; since the whole 127.x.x.x addresses suite refers to “localhost”, an idea could be using a different IP address for each of the nodes in the simulation. That way sender and receiver could be identified reading the 32 bits source address and 32 bits destination address in the IP packet header.

Unfortunately, this was not possible. After testing the application, was noticed that an address like 127.0.0.2 was automatically changed to 127.0.0.1. This problem is apparently due to the fact that, in order to subnet the 127.x.x.x suite, modifications of the kernel routing tables have to be done, and this update needs particular versions of the kernel and the right tools. Because it involved modifications to the kernel, this approach was rejected.

A simpler solution was chosen: distinguish between different nodes using port numbers, as done in the first Packet Filter implementation.

5.1.4 The Packet Sender

The Proxy is not supposed to just filter the incoming packets, but it has to delay them, apply the error rate to the byte array (both according to the information provided by STK) and finally send the packets to their intended destinations.

The Packet Sender’s task is sending the packets to their destinations whenever after applying the relative delay. Since what a safe raw socket does is just listening to the incoming traffic and not capturing the packets; what the Packet Filter can do is just listening to the traffic but not interfering with it; in other words it can’t prevent the destination from receiving the packet, as well. This problem is due to the fact that the nodes processes and the Proxy run on the same machine.

It is necessary to make the Proxy receive the packets instead of the destination, otherwise the destination would receive two equal packets (the original packet sent by the source and the delayed packet) and the second one would be discarded as a

duplicate. In this case, of course, the Proxy should be forced to receive the packet before the destination.

It appears to be possible to delete the packets from the TCP/IP stack (even if not without kernel modifications), but the real problem seems to be ensuring that a packet is received at the proxy first. Using process priorities could be a solution to this problem, but, since the Packet Filter performs some computations after it receives a packet, it is actually very likely to be the second and not the first one to receive the packet.

Thus, another solution should be found to overcome the problem. The idea became to notify the receiver, at the moment it is supposed to receive the delayed message. In order to apply this solution, the Packet Sender has the task of creating a TCPPacket, without payload that would be sent using a raw socket. These packets will have the source port equal to the sender source port and the destination port equal to the one of the receiver. This was done in order to be able to recognize source and destination of the original un-delayed packet. The Packet Filter is able to capture these packets sent by the Packet Sender and distinguish them from the JBDS simulator packets, this recognition is quite trivial; it is based on sequence numbers. It was noticed that all packets created by the JBDS simulator application had large sequence numbers, so it was necessary to use small sequence numbers (from 1 to 900). This solution isn't perfect, of course, but gave immediate and good results.

At the moment the Packet Filter receives one packet generated by the Packet Sender, it understands that a messages should be received at the destination, thus, it notifies the proxy about that, which in turn notifies the receiver, allowing it to read from the socket. This approach, unfortunately is not perfect: the packets used to establish the connection (the SYN packets) are not delayed, moreover the notification of the receiver instead of the receipt of the delayed packet can lead not to experience any change in TCP's behaviour has in deep space links (because the packets are buffered at the receiver before the invocation of the read method).

5.1.5 The Delay

In order to implement the delay at the Proxy every incoming packet is stored in a buffer for the time it is supposed to be delayed and then, sent to the destination. This

buffer contains, the packet together with the delay and error rate associated to it. The delay field in the buffer doesn't match with the link's actual delay: the value stored is in fact the delay associated to the link added to the current system clock value, as returned by the method:

```
System.getCurrentTimeMillis();
```

This value indicates the moment the packet should be received by the destination. This new delay value and the error rate associated to the packet is then stored in the buffer; a very simple sorting algorithm was implemented in order to keep the next packet to send at the first position in the buffer.

A very simple thread, checks every 10 ms if the current system time is equal or higher than the delay value of the first packet in the buffer. If so, the Packet sender is called in order to notify the receiving thread at the destination node.

At first the delay relative to the link was meant to be read from the payload of the packet, thus the Proxy didn't have to keep any information about links and didn't have to query the nodes. Unfortunately it was not possible to apply this solution, because it was not possible to read an object contained in the payload of a raw packet. The object the nodes send one another is called DataPacket (contained in the dataTypes package, Figure 12); it is a Serializable object and it is made up by various fields; some of them contain String objects.

In order to send a Serializable object using a Java socket, it is necessary to use a particular type of stream called ObjectOutputStream, whereas an ObjectInputStream is deployed in order to receive it. These streams, in order to give to the destination the chance to reconstruct the object, add a header to the actual payload. This header is object-specific, thus it is the same for every instance of a class.

The first attempt led to the discovery that it was not possible to obtain the payload object at the proxy just reading the data field of the packet using an ObjectInputStream, and a ByteArrayInputStream. The code threw the following exception:

```
java.io.StreamCorruptedException: invalid stream header
```

Hence, an experiment was done using a string object and comparing:

- The size of the string object;
- The size of the same string object plus the header assigned by the ObjectOutputStream;
- The length payload of a raw packet containing the same string.

The length of the payload of the raw packet was expected to be equal to the length of the object plus the header, but it was actually shorter.
In order to understand this strange behaviour, it was thought that reconstructing the header and comparing it to the byte array contained in the payload could help in finding the problem. It was actually possible to reconstruct the header for the object, but unfortunately no match between them was found, hence this idea was discarded and another approach had to be chosen.
Storing delay and error rate information at the Proxy (and at the Nodes) was a very expensive choice, because it involved a too large modification of the existing code and design.
However another possibility was represented by querying the nodes for the delay and error information relative to a packet.

5.1.6 The BER (Bit Error Rate)

Due to implementation problems, it was not possible to apply the BER to sent packets, because basically, as explained before, it was not possible to let the destination receive only the packets captured and processed by the proxy.
A definition of BER that can be found in [41] is the following:
"In a digital transmission, BER is the percentage of bits with errors divided by the total number of bits that have been transmitted, received or processed over a given time period. The rate is typically expressed as 10 to the negative power. For example, four erroneous bits out of 100,000 bits transmitted would be expressed as 4×10^{-5}*, or the expression* 3×10^{-6} *would indicate that three bits were in error out of 1,000,000 transmitted".*

The idea on how to apply the error rate on a string of bytes was to use a mask made of bits randomly generated using the Java Random class. The Random.java class gives the chance to specify a range within the numbers that will be generated by the random generator. This range could be set in order to match the BER. Thus, the probability for the generator to generate a precise number would be equal to the probability that a bit could be inverted, according to the BER.
The string generated by this method could be look something like the one in Figure 19.

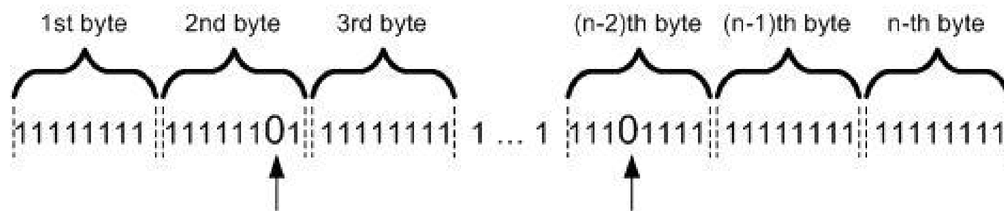

Figure 19: Mask used to apply the error rate to the packets

For example, in Figure 19, the bits that won't be received correctly at the destination will be 2. The string shown in Figure 19 is be used as a mask to obtain the string of bytes that will actually be sent. The error string is calculated using the formula (2)(4).

$$outpkt = m \bullet inpkt \tag{4}$$

In (4), *outpkt* indicates the byte string that is generated after the error is applied, whereas *inpkt* represents the byte string corresponding to the un-corrupted message. The mask is indicated by *m* and the symbol • stands for a bitwise and.

6 Testing and evaluation

Three scenarios were chosen in order to test the application. They all represent space missions to the Moon and they are quite similar, but the differences in the access periods between objects change significantly the results expected from the simulation. The common features in the scenarios are explained in chapter 1.

The first and the second scenarios have many common features; the difference between them is the location of the Lunars on the Moon surface, changing their position was in fact useful for the evaluation of data like the queue size and the throughput.

In order to run the three simulations, two machines are employed. The first machine is used to run STK and the second to run the JBDS simulator. These two computers, in order to communicate, use the TCP/IP socket interface provided by STK/Connect. The first machine employs Windows XP Professional (2002), the CPU is an Intel Pentium 4 (2.80GHz), and the RAM is 512 MB. The second machine uses Ubuntu 5.10, the version of the kernel is 2.6.12, the CPU is an Intel Pentium M 1,300MHz and the RAM is 512 MB. The first two simulations last roughly 6 simulation days, whereas the third one lasts 8 simulation days.

The speed of the JBDS simulator is affected by the large number of threads employed and by hardware issues, however it is important to say that STK is polled for information every 10 seconds, and the time granularity of the reports is 60 simulation seconds (a different value for the range and error rate is provided for every simulation minute), whereas the Time Manager's timestep is 1000 ms.

With these inputs the JBDS simulator was noticed to need roughly 2-3 seconds in order to compute a simulation minute (1:30 seconds).

6.1 The first Scenario

The first scenario is shown in Figure 20, whereas the topology is shown in Figure 21.

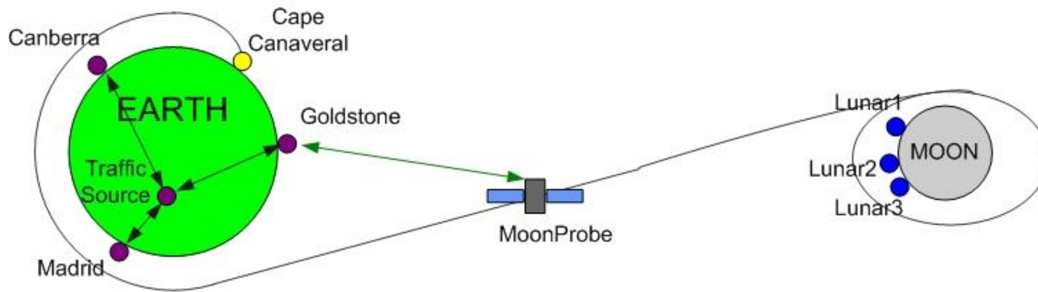

Figure 20: The first scenario

In the first scenario the communication links between objects are the following:

- TrafficSource ← → Canberra
- TrafficSource ← → Madrid
- TrafficSource ← → Goldstone
- MoonProbe ← → Canberra
- MoonProbe ← → Madrid
- MoonProbe ← → Goldstone
- MoonProbe ← → Lunar1
- MoonProbe ← → Lunar2
- MoonProbe ← → Lunar3

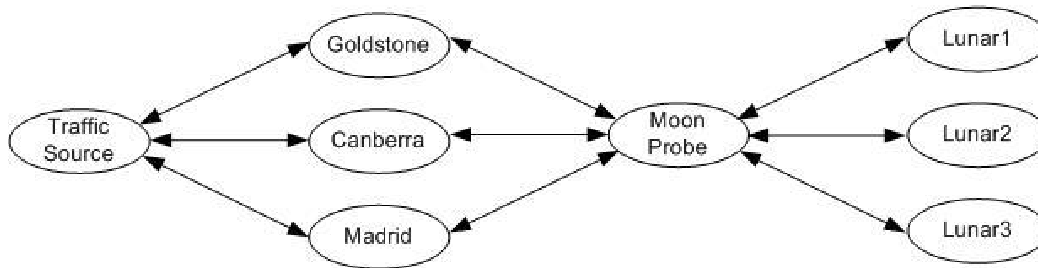

Figure 21: The first and second scenario topology

During the first simulation days, when the MoonProbe is approaching the Moon, there is no communication between MoonProbe and Lunar1, 2, and 3. Moreover the communication between Earth and MoonProbe is unidirectional. The Probe receives the commands form Earth but doesn't send any packet. It basically behaves as a sink for data. When the MoonProbe starts orbiting around the Moon, it also starts behaving as a router, it receives packets form Earth and sends them to the appropriate

destination on the Moon, but it also starts receiving packets form the Lunars and forwards them to Earth.

STK uses a coordinate system to locate places on the Moon. It is a based on longitude and latitude; the locations of the three Lunars are identified by the coordinates:

- Lunar 1
    - latitude 19.8
    - longitude 31
- Lunar 2
    - latitude -9.11
    - longitude 15.65
- Lunar 3
    - latitude 80
    - longitude -2

All of them are located on the visible side of the Moon.

6.1.1 The results obtained: the delay

In the first 4 simulation days, it is not interesting to analyse the queue size or the throughput. This is because the packets basically are sent from Traffic Source through the three ground Facilities to the MoonProbe. However, in order to check if the delaying Proxy was working correctly a Parser was coded in order to analyse the delay associated to each packet. Subtracting the time (system clock) when the packet is sent to the time it was received, it is possible to obtain the time each packet was delayed. The graph in Figure 22 shows how the delay increases as the MoonProbe leaves the Earth's attraction. Due to the high number of packets created the graph in Figure 22 covers just the first simulation days (from the launch date until the 2nd of January at 6:57 am). The distance between the visible antenna and the probe in that moment is 218385.707563 km, which corresponds to a delay of about 730 ms. Thus, the average of the measurements contained in the graph is quite accurate. It must be considered anyway that some time is spent in computations at the Proxy. In Figure 22 are also visible a few peaks, where the delay reaches almost 1400. These erroneous

values are probably due to thread scheduling (since there are many threads running at the same time), or to latencies due to computations.

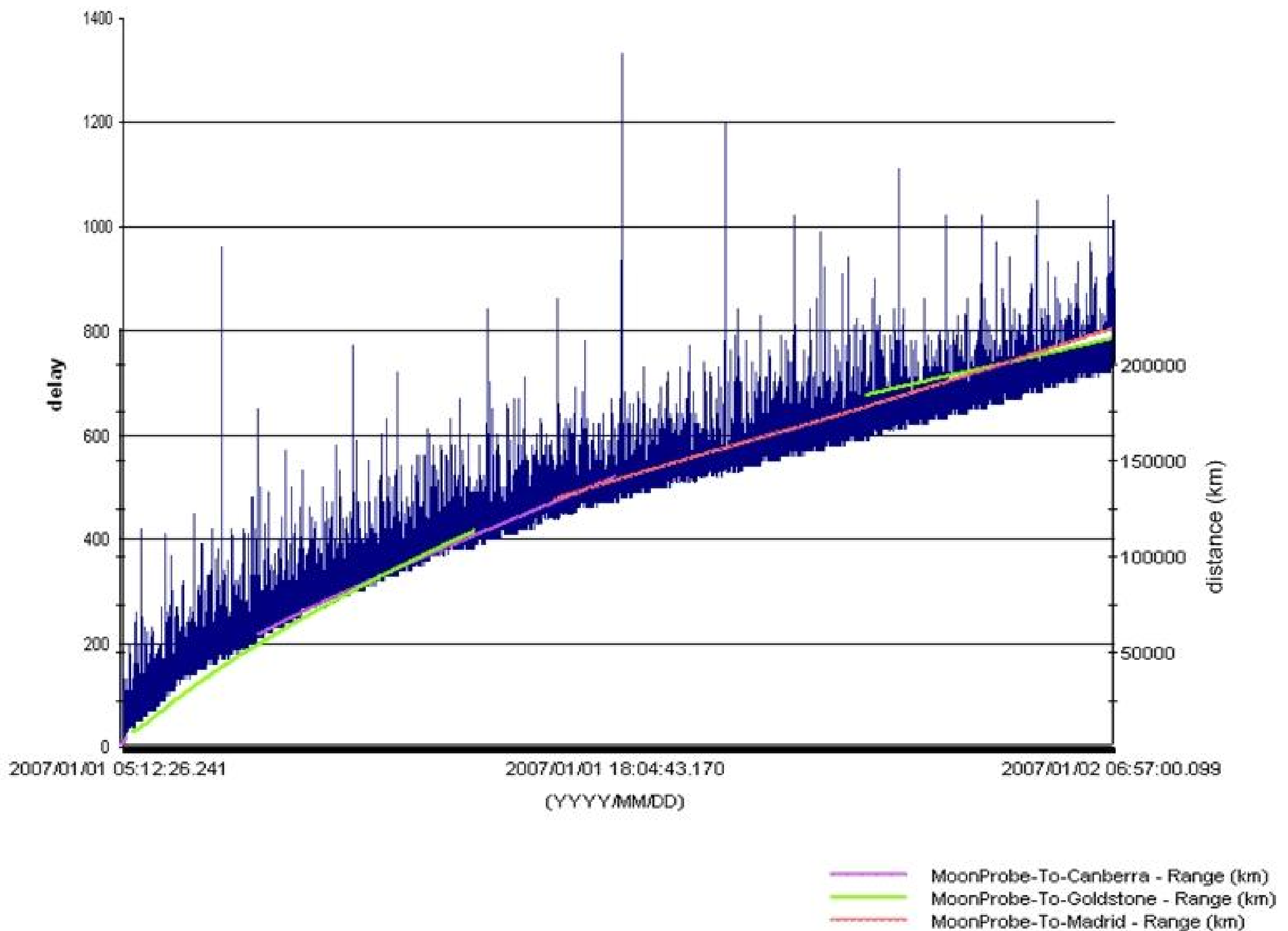

Figure 22: Increase in packet's delay as MoonProbe leaves Earth's attraction

The first simulation days (when no packets are either sent or receiver by the Moon Facilities) were also noticed to be simulated more slowly, this can be due to the fact that the threads that are supposed to send data departing from the Lunars loop continuously looking for visibility of the destination, and they never find it.

6.1.2 The results obtained: the queue size

The first simulation days (i.e. during the spacecraft's days of journey to the Moon), don't give many interesting metrics to measure, whereas when the MoonProbe starts

orbiting around the Moon, the number of relevant data to evaluate increases. In order to be able to measure the queue size at the main router, which is MoonProbe, a few lines of code were added to the method that is supposed to send the packets. When a packet is sent, the queue size is stored in a .txt file called Queue.txt.

The results are shown in Figure 23.

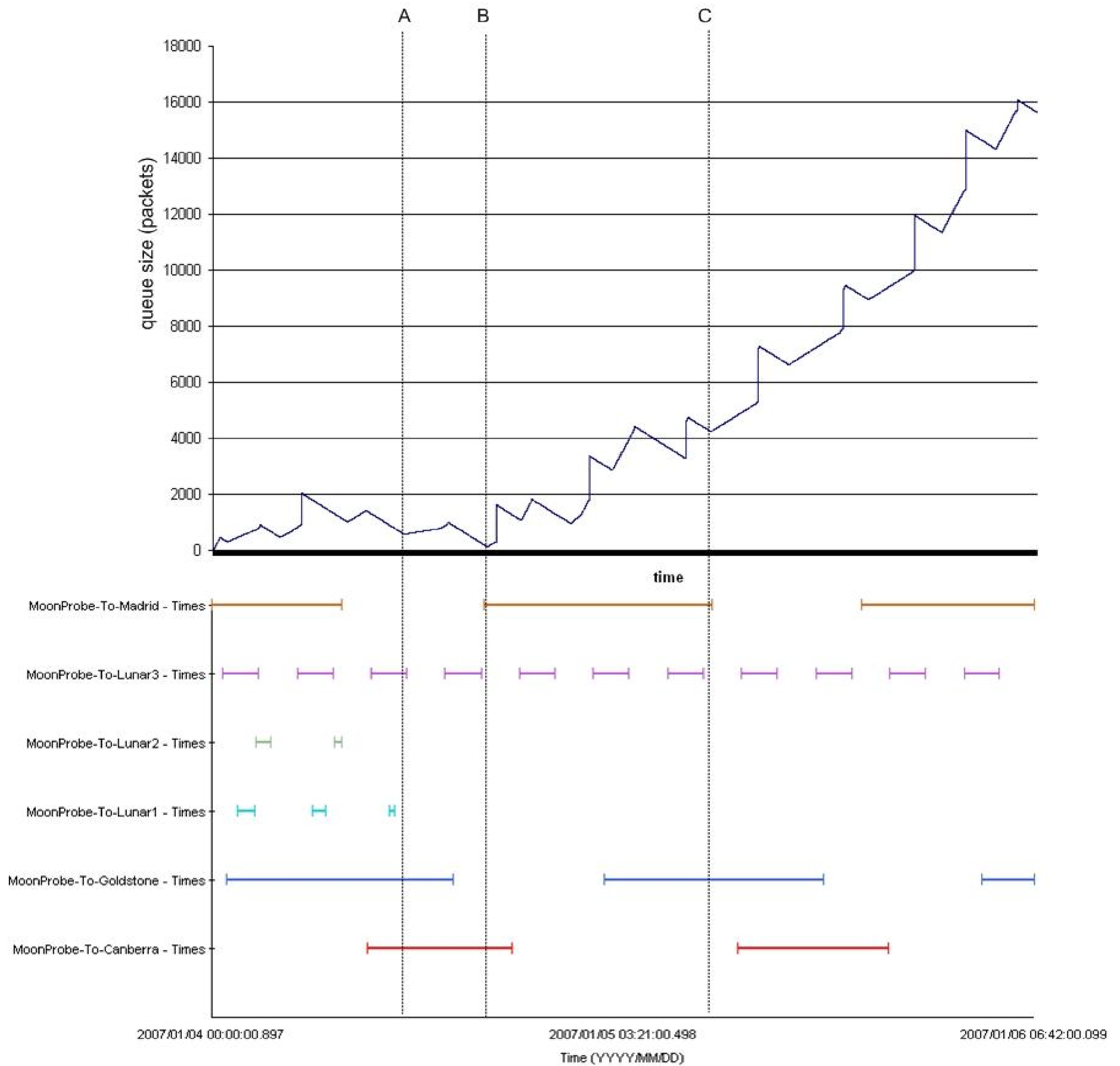

Figure 23: Queue size at the MoonProbe, compared to the access times (first simulation)

Figure 23 shows the queue size at the MoonProbe (top graph) compared to the Facilities access times to the MoonProbe (bottom graph). Since Lunar1 and 2 have access to the probe just in the first hours, and the Traffic Source keeps sending packets to the MoonProbe, directed to Lunar 1 and 2, these packets are never delivered and they are stored inside the queue. This is the reason why the queue size increases. However, it is important to notice how the access to the Lunar 3 affects the size of the queue. Figure 23 shows the line marked with C, which shows a decrease and then an increase in the queue size. It is noticeable that this access provokes small, but regular decreases in the queue size.

6.1.3 The results obtained: the throughput and fairness

The throughput value has been computed for each of the receiving nodes (TrafficSource, Lunar1, Lunar2 and Lunar3) as the amount of bytes received per second. The result of this computation is showed in Figure 24.

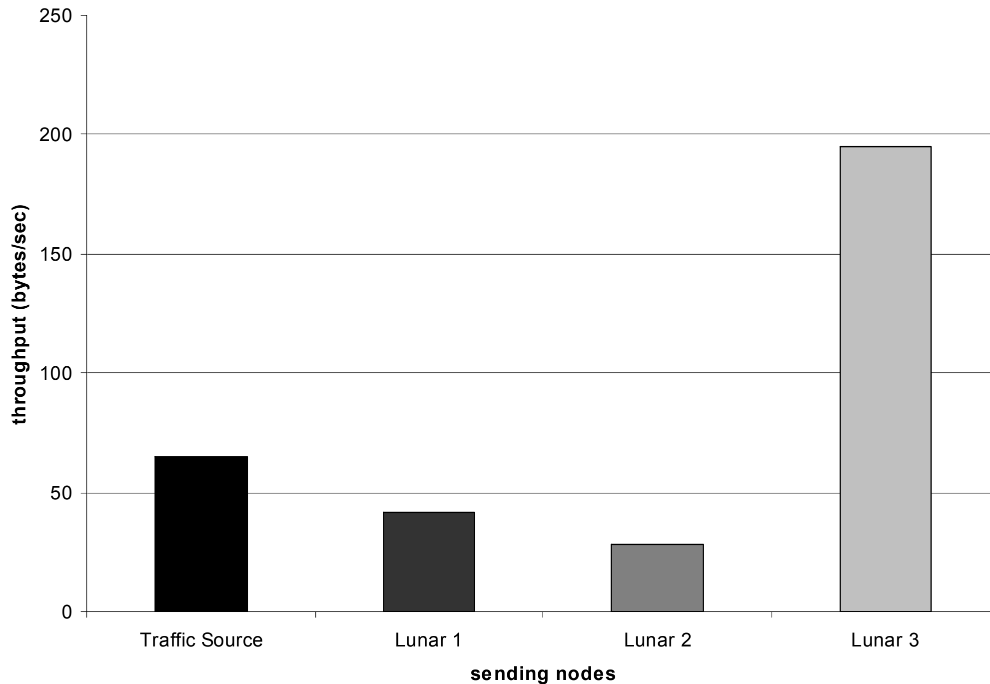

Figure 24: Throughput first scenario

The results shown in Figure 24 have to be analysed bearing in mind the access times between objects, shown in Figure 23. Since Lunar1 and Lunar2, have a very short access period with the MoonProbe (which is supposed to deliver them the packets sent by TrafficSource), they receive a low number of packets. Thus the throughput is very low (less than 50 bytes/sec). On the other hand Lunar3 has regular access periods with MoonProbe, thus the delivery of the packets coming form TrafficSource and directed to Lunar3, can be regularly completed.
For this reason Lunar3 is the facility with the highest throughput (almost 200 bytes/sec).
TrafficSource's throughput is quite low (less than 70 bytes/sec); it seems to be affected by TrafficSource's distance from the MoonProbe (formula (1) explained how in TCP the throughput decreases as the RTT increases).
The reason for TrafficSource's low throughput was also thought to be due to the management of the queue at the MoonProbe. However, the router (MoonProbe) keeps four different queues, one for each destination and regularly sends a packet extracted from each of the queues; hence, it should have a fair management of the queue and ensure the delivery to TrafficSource at all time (because it is always visible). Therefore, the reason for such low throughput should be linked to the distance that TrafficSource has (compared to the Lunar facilities) from the MoonProbe and not to the queue management.
Because of the huge differences between the throughputs, the fairness[11] is very low; this metric was measured twice: one for all the sending nodes and one just for the Moon facilities, the results obtained are the following:

- Global fairness 0.50516282
- Lunar's fairness 0.57437855

[11] The fairness was computed using Jain's formula:

$$f(x_0, x_1, x_2, \cdots, x_n) = \frac{\left(\sum_{i=0}^{n} x_i\right)^2}{n \sum_{i=0}^{n} x_i^2}$$

Where x_i represents the total throughput for flow i and n represents the total number of nodes.

6.2 *The second scenario*

The second scenario is shown in Figure 25.

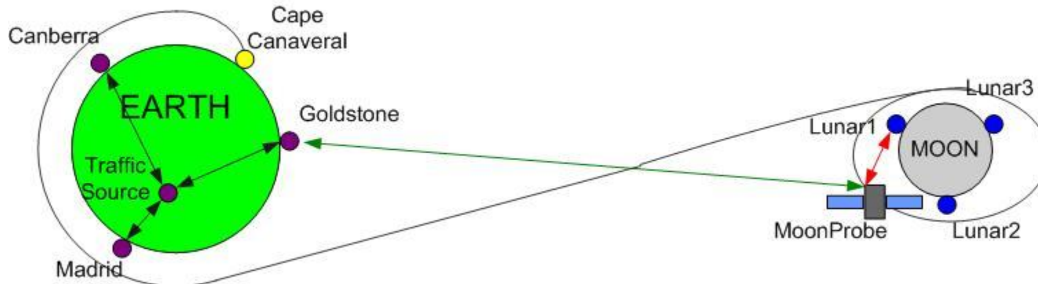

Figure 25: The second scenario

The difference between the first and the second scenario lies in the position chosen for the Lunars on the Moon's surface. In the second scenario the coordinates are the following:

- Lunar 1
    - latitude 10.8
    - longitude 136
- Lunar 2
    - latitude -86.5
    - longitude -6.1
- Lunar 3
    - latitude 80
    - longitude -2

The location of the Lunar3 Facility doesn't change, whereas the locations of Lunar1, and 2, are totally different from the ones used in the previous scenario. How the locations affect the access periods and subsequently the queue size at the probe, is explained in the following paragraph.

6.2.1 The results obtained: the queue size

The graph in Figure 26 shows the queue size of the MoonProbe, related to the access times in the second scenario.

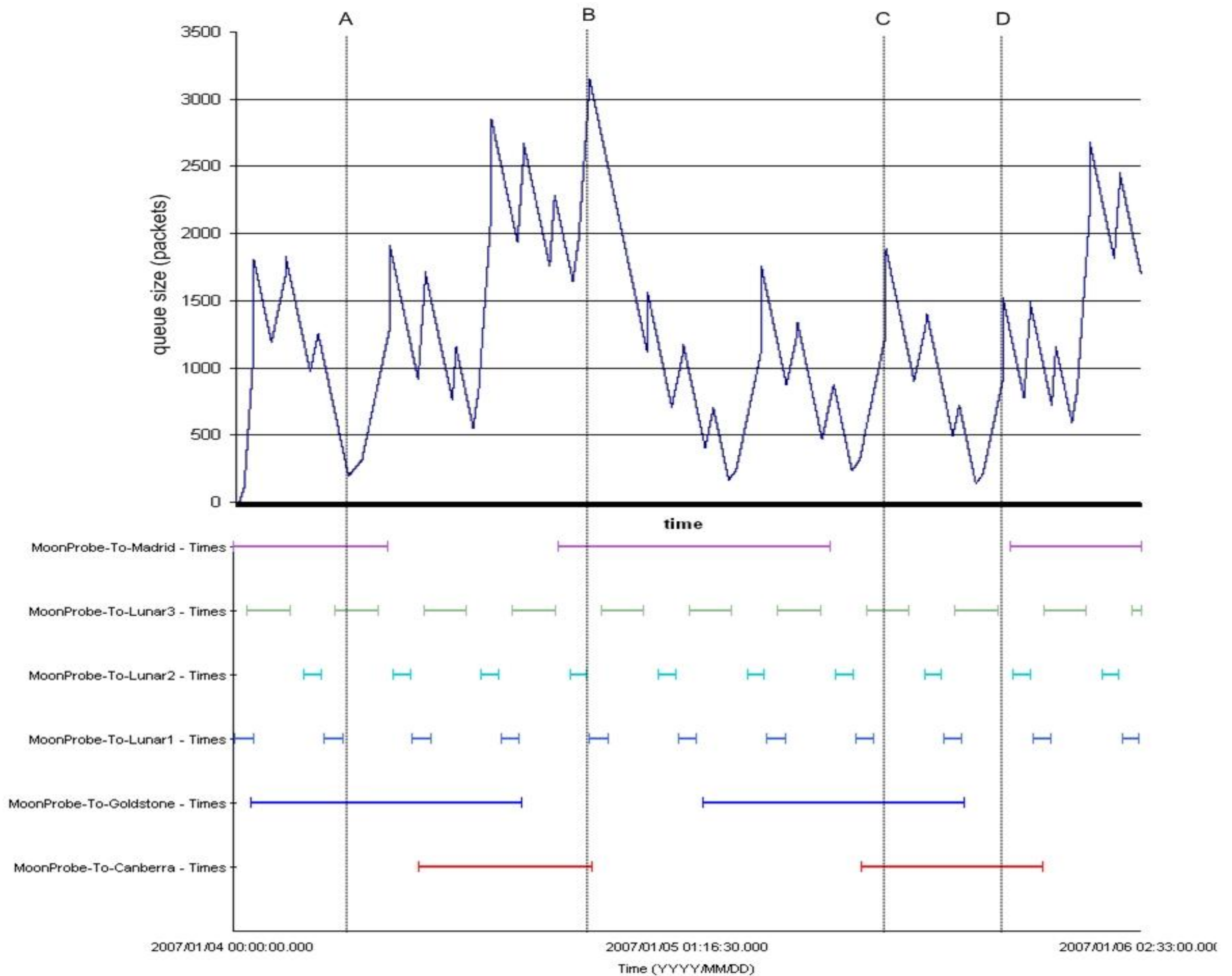

Figure 26: Queue size at the MoonProbe, compared to the access times (second simulation)

The difference in the location of the Lunars changes radically the access graph, and shows that Lunar 1 and 2 have regular access intervals; this makes feasible for MoonProbe to deliver the messages directed to those facilities (which was impossible in the first scenario). For this reason the queue size doesn't have a steady increase (as happened in the previous scenario), but it is more noticeable a fluctuation in size. This

fluctuation is quite regular, anyway, which was the behaviour expected since the access periods are regular as well. The peaks in Figure 26 (B, C and D) can be linked to the lack of access to the Lunar facilities.

6.2.2 The results obtained: the throughput and the fairness

Figure 27 shows the throughput values for each of the sending nodes in the second scenario. The throughput relative to the TrafficSource facility is still quite low, which was the behaviour expected since there is no real difference in the network topology with the first scenario. On the other hand all the three Lunar facilities show high throughputs, which was the behaviour expected, since the access times with the MoonProbe are periodic for all of them.

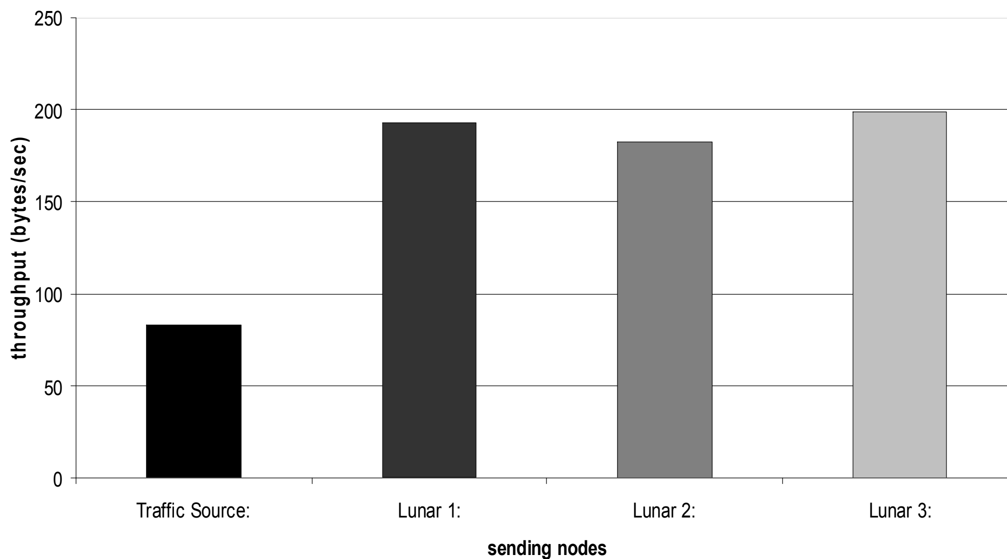

Figure 27: Throughput second scenario

The fairness is higher than in the first scenario:

- Global fairness 0.92328674
- Fairness lunar 0.99878365

6.3 The third scenario

In the third scenario the Facilities on Earth and Moon are all located on the visible side, equally to the first scenario (Figure 20). Their coordinates are:

- Lunar 1
    - latitude 6
    - longitude 7
- Lunar 2
    - latitude -7.8
    - longitude 11
- Lunar 3
    - latitude 80
    - longitude -2

The most relevant difference between the two scenarios is in the topology, shown in Figure 28.

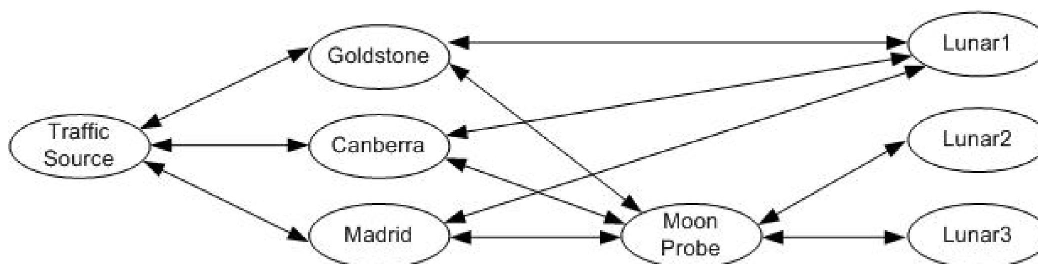

Figure 28: The third scenario topology

One of the Moon facilities: Lunar1 has direct connection to Earth. In this case the MoonProbe acts as a router just for Lunar 2 and 3.

The simulation has been run for 8 simulation days, two days more than the previous ones and the results obtained are analysed in the following paragraphs.

6.3.1 The results obtained: the queue size

The queue size at the MoonProbe is shown in Figure 29. The graph shows an unpredictable behaviour for the queue size, this value seem to fluctuate, but not regularly, and not following precise patterns. This simulation was run at first for 6 days, but since the queue size showed this strange behaviour, it was run again for longer. But it is still difficult to explain the results obtained. There are a few peaks (A, B, C, D, E and F), but they seem to be independent from the access that the probe experiences with the Lunar facilities.

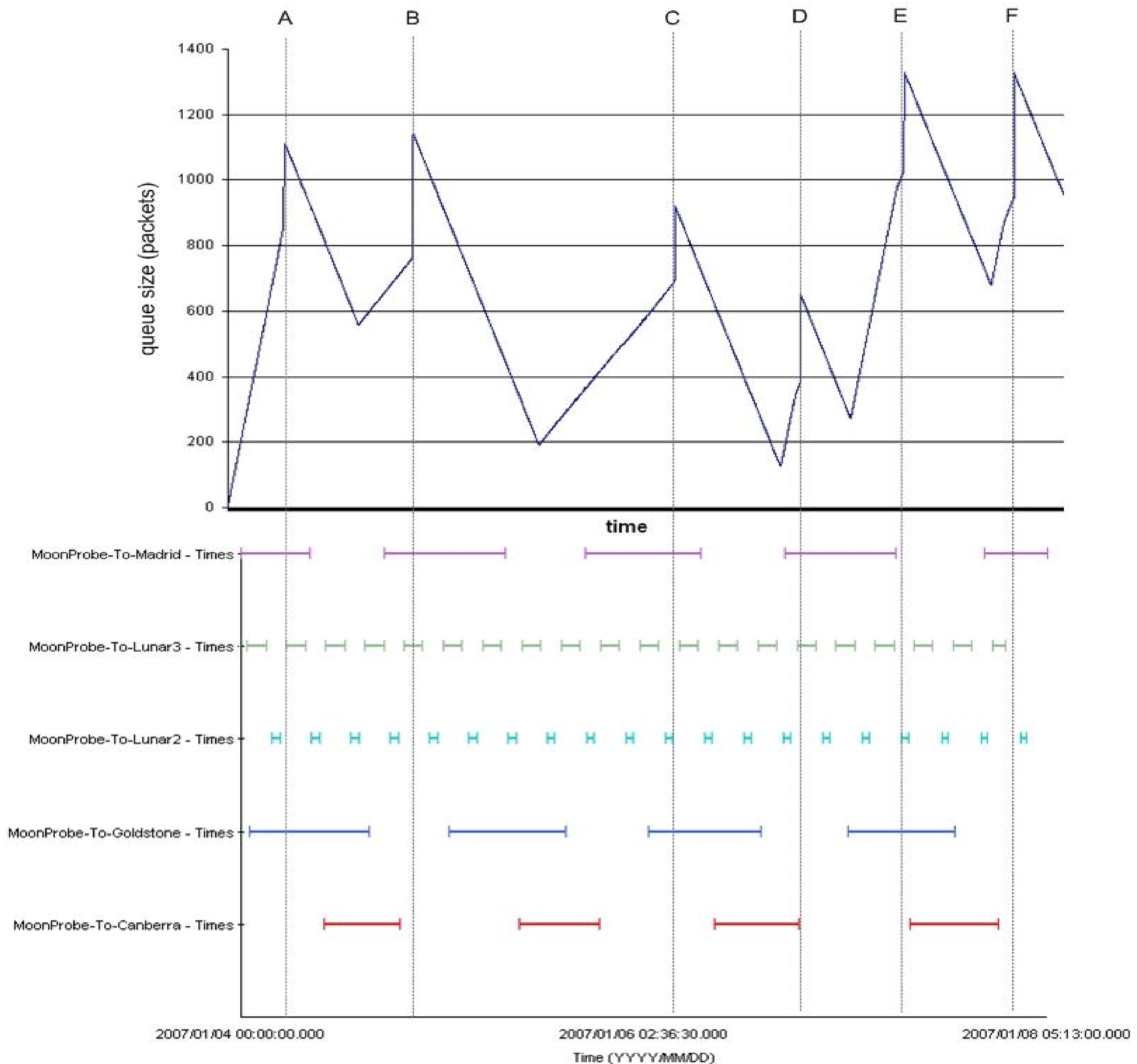

Figure 29: Queue size at the MoonProbe, compared to the access times (third simulation)

6.3.2 The results obtained: the throughput and the fairness

The throughput values obtained are shown in Figure 30. In this scenario, due to the direct connection between Lunar 1 and TrafficSource, the TrafficSource's throughput is much higher than in the previous scenarios. Moreover, Lunar 1 has a higher throughput if compared to Lunar 2 and 3. This can be explained considering that TrafficSource is always visible for Lunar1 and viceversa, whereas Lunar 2 and 3 have to wait for the Probe to be accessible in order to send and receive packets. This was the behaviour expected.

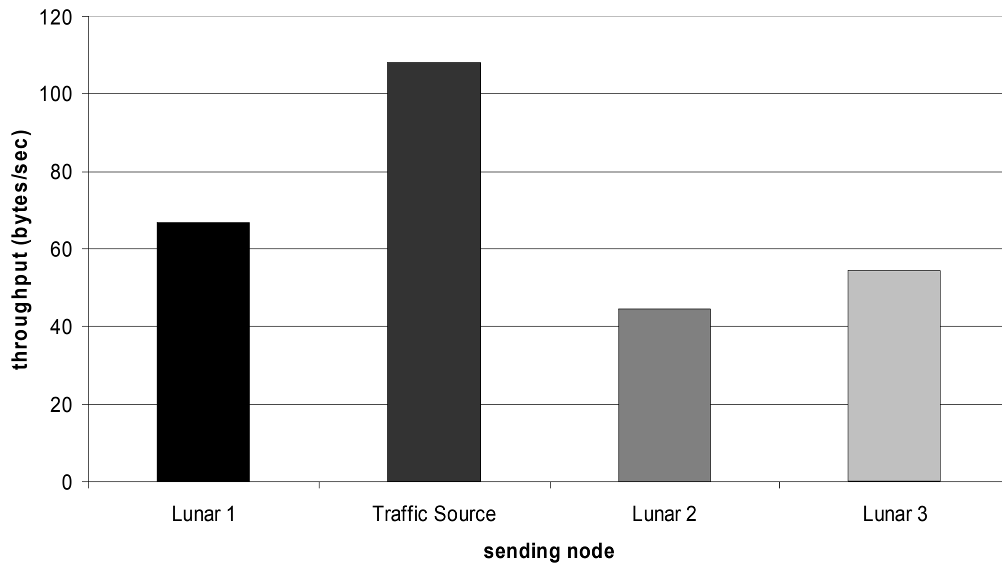

Figure 30: Throughput third scenario

The results obtained for the fairness are the following:

- Global fairness 0.8896817
- Lunar's fairness 0.9735295

7 Conclusions and future work

The framework developed in this thesis represents a new approach in the field of deep space network simulation.

The lack of kernel modifications together with the use of Java, which is a widespread language, makes the JBDS simulator portable from one platform to the other. Moreover the JBDS simulator was designed to implement direct connections between nodes, so that the link delay and error rate could be transparently applied to the packets.

Unfortunately, mostly because of implementation problems, but also of time constraints that prevented from implementing the simulator differently (solving some of the problems related to the design), the proxy is not able to transparently apply the delay and the error rate to the packets.

However the scenarios used to evaluate the simulator and the results obtained show that the behaviour of TCP seems to be someway affected by the delay applied to the packets.

As future work, it could be extremely interesting to implement the simulator in a way that each of the components (nodes, proxy and Manager) runs on a separate machine (maybe one for each of the nodes, or just using a different machine for the proxy or the Manager). This approach would reduce the problems encountered running the whole application in localhost. However this solution could be quite expensive and could lead to scalability problems (for example related to the number of nodes in the scenario).

Another improvement proposed for the code is the implementation of the error rate at the Proxy, which could lead to a more accurate simulation of the behaviour of a deep space link.

Small workarounds have been used in order for example to distinguish between the packets sent by the Packet Sender and the packets created by the simulator nodes (the sequence numbers were employed for that purpose). Some of the used solutions are not perfect and in the future, those implementation problems should be solved.

Another important issue is the output file; it should be able to keep track of the sent and the received packets as well as the acknowledgements for those packets,

moreover the simulator should delay the acknowledgements as well as the data packets; feature that is not currently implemented.

Implementation problems, mostly related to the fact that the socket implementation in Java is too high level, could be maybe solved with future Java releases. As a matter of fact every new Java version contains a larger number of low level features; thus, maybe in the future there will be no need to use the native interface.

After the solution of these implementation issues could be interesting to employ the simulator for the task it was supposed to accomplish from the beginning, and use it for the comparison of protocols' performance.

All these enhancements are left for future development and their correct implementation could let the JBDS simulator be a powerful tool for research in the deep space network field.

Bibliography

[1] M. Allman, C. Hayes, H. Kruse, S. Ostermann. "TCP performance over satellite links". From the 5th Conference of Telecommunication Systems – 1997.

[2] I. F. Akyildiz, G. Morabito, and S. Palazzo. "TCP-Peach: A New Congestion Control Scheme for Satellite IP Networks". IEEE/ACM Transactions on networking – 2001.

[3] I. F. Akyildiz, X. Zhang, and J. Fang, "TCP-Peach+: Enhancement of TCP-Peach for satellite IP networks" IEEE Communications Letters - 2002.

[4] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and Ren Wang.

"TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links". ACM SIGMOBILE – 2001.

[5] S. Rallapalli, "Emulation of a Space Based Internet Communication Link: Design and Implementation". MS Thesis, University of Kansas, Lawrence, 2002.

[6] S. Baliga. "Design of a Space Based Internet Emulation System". MS Thesis, University of Kansas – 2002.

[7] The CMU Monarch Project. The CMU Monarch Project's Wireless and Mobility Extensions to NS – 1998. Available from http://www.monarch.cs.cmu.edu/.

[8] T. Henderson, R. Katz. "Network Simulation for LEO Satellite Networks". Proceedings of 18th International Communication Satellite Systems Conference – 2000.

[9] S. Endres, M. Griffith, B. Malakooti. "Space Based Internet Network Emulation for Deep Space Mission Applications". International Communications Satellite Systems Conference & Exhibit – 2004.

[10] NASA, Jet Propulsion Laboratory website: available at http://www.jpl.nasa.gov

[11] NASA, Jet Propulsion Laboratory, Mission and Spacecraft Library website: available at http://msl.jpl.nasa.gov

[12] Satellite Tool Kit website: available at http://www.agi.com

[13] R. C. Durst, P. D. Feighery, K. L. Scott. "Why not use the Standard Internet Suite for the Interplanetary Internet?". Interplanetary Internet Study Seminar, California Institute of Technology –1999.

[14] K. Fall. "A Delay-Tolerant Network Architecture for Challenged Internets". SIGCOMM – 2003.

[15] http://www.mnm-team.org/pub/Publikationen/kell98d/HTML-version/node2.html

[16] http://66.249.93.104/search?q=cache:8S5Dm19ZREIJ:www.comlinks.com/satcom/ico.htm+ico+satellite+MEO&hl=en&gl=ie&ct=clnk&cd=2&client=firefox-a

[17] ICO Global Communications website: available at http://www.ico.com

[18] IRIDUM Satellite system website: available at http://www.iridium.com

[19] GlobalStar Satellite system website: available at http://www.globalstarusa.com/en

[20] F. Warthman. "Delay Tolerant Networks – A Tutorial". DTN Research Group Internet Draft – 2003.

[21] "Transmission Control Protocol DARPA Internet Program Protocol Specification" University of Southern California – 1981.

[22] http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ip.htm

[23] http://condor.depaul.edu/~jkristof/technotes/tcp.html

[24] V. Jacobson. "Congestion Avoidance and Control". Proceedings of SIGCOMM – 1988.

[25] http://ispcolumn.isoc.org/2004-07/tcp1.html

[26] T.V. Lakshman, and U. Madhow, "The Performance of TCP/IP for Networks with High Bandwidth Delay Products and Random Loss". IEEE/ACM Transactions on Networking – 1997.

[27] M. Allman, D. Glover, L. Sanchez, "Enhancing TCP Over Satellite Channels using Standard Mechanisms", RFC 2488 – 1999.

[28] D. Chandra, RJ Harris, N. Shenoy, "Congestion and Corruption Loss Detection with Enhanced TCP", ATNAC 2003, Melbourne, Australia - 2003

[29] R. Braden, "Requirements for Internet Hosts --Communication Layers", STD 3, RFC 1122 - 1989.

[30] M. Allman, S. Floyd, and C. Partridge. "Increasing TCP's initial window". Internet RFC 2414 - 1998

[31] A. Bakre and B. R. Badrinath. "I-TCP: Indirect TCP for mobile hosts" in Proc. 15th Int. Conf. Distributed Computing Systems (ICDCS) - 1995.

[32] V. N. Padmanabhan and R. Katz. "TCP Fast Start: A technique for speeding up web transfer". In Proc. IEEE Globecom - 1998.

[33] V. Cerf et. al., "Delay Tolerant Network Architecture", draft-irtf-dtnrg-arch-05.txt - 2006

[34] K. Scott, S. Burleigh, "Bundle Protocol Specification", draft-irtf-dtnrg-bundle-spec-04.txt - 2006

[35] Wikipedia website: available at http://en.wikipedia.org/wiki

[36] http://blogs.msdn.com/michael_howard/archive/2004/08/12/213611.aspx

[37] http://www.savarese.org/software/rocksaw

[38] http://www.savarese.org/software/vserv-tcpip/index.html

[39] http://www.labcompliance.com/glossary/g-i-glossary.htm

[40] http://www.angelfire.com/anime3/internet/communications.htm

[41] http://www.angelfire.com/anime3/internet/data.htm

[42] http://www.agi.com/resources/download/tutorials/

[43] http://www.agi.com/resources/faqSystem/faq_welcome.cfm

[44] K. Basin. "Developing Architectures and Technologies for an Evolvable NASA Space Communication Infrastructure". NASA/TM—2004-213108. July 2004

[45] http://www.homoexcelsior.com/omega.db/datum/aerospace_engineering/parking_orbit/7180

[46] http://www.bpccs.com/lcas/Dictionary/perisele.htm

[47] http://scienceworld.wolfram.com/physics/Eccentricity.html

[48] http://www.ontariosciencecentre.ca/school/clc/visits/glossary.asp

[49] http://www.ex-astris-scientia.org/treknology2.htm

[50] http://www.wirelesstelcorp.com/glossary_of_terms.htm

[51] http://www.w3schools.com/default.asp

[52] http://www.visolve.com/squid/whitepapers/qos.php

[53] http://www.tjiss.net/glossary_i.html

[54] http://lightingdesignlab.com/library/glossary.htm