Satellite-based Web server

Enock Vongani Maluleke

Thesis presented in the fulfilment of the requirements for the degree of Master of Engineering Science at the University of Stellenbosch.

Supervisor: Mr R.M. Barry

October 2002

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirely or in part submitted it at any University for a degree.

Signature

Date

Abstract

There is a large variety of telemetry receiving software currently available for the reception of telemetry information from different satellites. Most of the software used in receiving telemetry data is satellite specific. Hence, a user-friendly way is needed to make telemetry data easily accessible. A satellite-based web server is aimed at providing telemetry information to any standard web browser as a way of bringing space technology awareness to the people. Two different satellite-based web server methods are examined in this thesis. Based on the evaluation, the on-board File server with proxy server was proposed for satellite-based web server development. This requires that the File server be ported to the on-board computer of the satellite. The web proxy server is placed on the ground segment with the necessary communication requirements to communicate with the on-board File server. In the absence of satellite, the satellite-based web server was successfully implemented on two computers, laying a good foundation for implementation on the on-board computer of the satellite (OBC).

Opsomming

Daar is 'n groot verskeidenheid telemetrie ontvangs sagteware huidiglik beskikbaar vir die ontvangs van telemetrie informasie vanaf verskillende satelliete. Die meeste van die sagteware wat gebruik word om telemetrie data te ontvang is satelliet spesifiek. Gevolglik,'n gebruikers vriendelike metode is nodig om telemetrie data maklik beskikbaar te maak. 'n Satelliet-gebaseerde web-bediener word beoog om telemetrie informasie te verskaf aan enige standaard web-blaaier as 'n metode om mense bewus te maak van ruimte tegnologie. Twee verskillende satelliet gebaseerde web-bediener metodes sal ondersoek word in hierdie tesis. Gebaseer op 'n evaluering, word die aanboord leêr-bediener met instaanbediener voorgestel vir satelliet-gebaseerde web-bediener ontwikkeling. Hiervoor is dit nodig dat die leêr-bediener na die aanboord rekenaar van die satelliet gepoort word. Die web instaanbediener word op die grond segment geplaas met die nodige kommunikasie benodighede, om te kommunikeer met die aanboord leêr-bediener. In die afwesigheid van die satelliet was die satelliet-gebaseerde web-bediener met sukses geïmplementeer op twee rekenaars, met die gevolg dat 'n goeie fondasie gelê is vir die implementering op die aanboord rekenaar van die satelliet (OBC).

Acknowledgments

I would like to extend my gratitude to my supervisor, Mr. R Barry, for guiding me throughout my research. To my friends, thank you for the support. To my family, Mom Caroline, my sisters and my brothers, thank you for your patience and for your inspirations. To my star (Thembakazi), thank you for your pantience, encourangement, and your support. Above all I would like to thank God for renewing my courage when I was discouraged and for giving me wisdom without finding fault.

List of Figures

List of Tables

List of Abbreviations and Acronyms

HTTP	Hypertext Transfer Protocol
HTML	Hypertext Makeup Language
I/O	Input/Output
MIME	Multipurpose Internet Mail Extensions
TCP/IP	Transport Control Protocol/ Internet Protocol
ARQ	Automatic Repeat Request
OBC	On Board Computer
PC	Personal Computer
SCP	Serial Communication Protocol
PPP	Point-to-Point Protocol
JVM	Java Virtual Machine
DT	Download Time
API	Application Programming Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WWW	World Wide Web

Table of contents

Chapter1

1 Introduction

Many people today want to communicate while moving from one point to another. This need brought a challenge to terrestrial communication links such as fiber, which for some time seemed to be a long-term solution because of being capable of providing high bandwidth services. With the introduction of satellite technology, this challenge can be overcome at a lower cost.

Unlike terrestrial fiber, which provides high-bandwidth services, but to a small number of users, satellite communication provides services to many users. Today's satellite applications include remote sensing, broadcasting, weather focus and many more. In addition to these, one of the main objectives of microsatellites, in particular, is to educate people about space communication (Chetty, 1999).

Microsatellites communication system consists of a space segment and a ground segment. The space segment contains the satellite and all terrestrial facilities for the control and monitoring of the satellite. These facilities include telecommand and telemetry subsystems.

A telemetry subsystem gathers readings from a multitude of sensors placed inside and outside the satellite (Steenkamp, 1999). These sensors include current and voltage and these are stored in the on-board computer (OBC) of the satellite. The operator uses telemetry information for daily housekeeping of the satellite. Radio amateurs use telemetry data particularly from microsatellites for experimental purposes.

There are different formats in which telemetry data is transmitted by different satellites. Hence, there is a large variety of telemetry receiving software currently used for the reception, decoding and displaying of telemetry data from different satellites. Although, there are some software packages that can be used to receive data from more than one satellite, there is a need to find a standard user-friendly way in which telemetry data can be distributed, particularly to radio amateurs, and also be made available to people as a way of bringing space technology awareness to the community.

The aim of this thesis is to design and develop a satellite-based web server. The satellite-based web server will make telemetry information accessible through an existing standard web browser. This way people interested in space communication will be freed from having to write their own software to receive, decode and display telemetry information. Also, most people are well oriented with the web browser, hence they will find it simple to utilize.

This document is structured as follows:

- Chapter 2 presents a complete overview of the web server in general, emphasizing its communication with the client browser. It also provides different ways, which a web server uses to process requests in parallel. Finally, it presents two flow control techniques, which are mostly used for communication protocol.
- Chapter 3 describes and evaluates the possible designs of the satellite-based web server. This chapter finally proposes the design that will be implemented for satellite-based web server.
- Chapter 4 focuses on the implementation of the proposed design. It provides the operation and real implementation of each software component.
- Chapter 5 discusses the results obtained during the functional test of the final software and concludes by presenting the future enhancements of the system.

Chapter 2

2 Literature Study

Yeager and McGrath (1996) define a web server as a computer with a connection to the internet, with the system software to run the computer and to connect to other systems on the internet. The network software and the operating system of the computer hardware form the computing platform of the web server. The web server software holds the whole system together, and provides the connection between the web server documents and the computing platform as shown in figure 2.1.

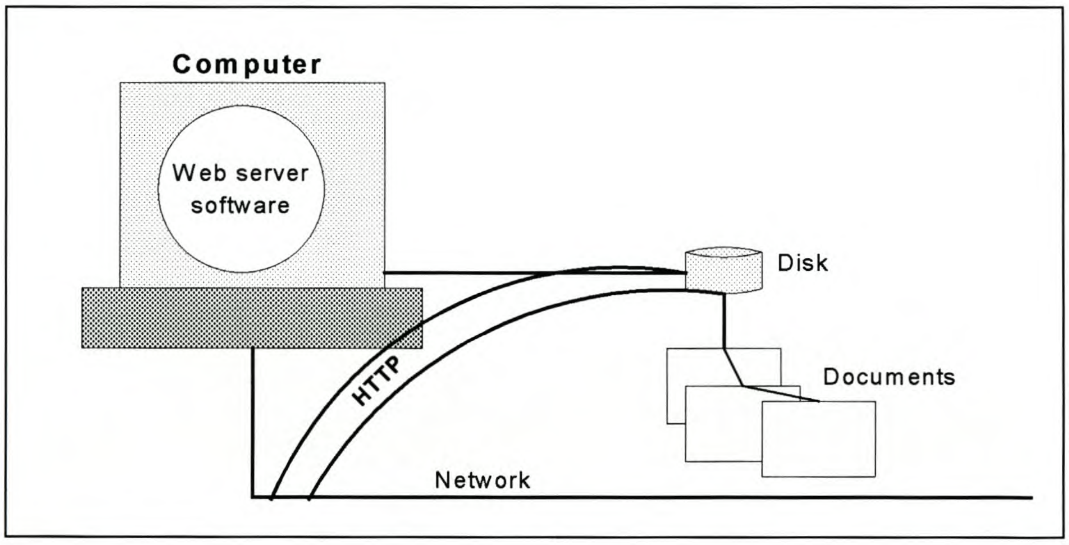

Figure 2.1 Web server system (Yeager and McGrath, 1996)

The web server software receives requests from the client browser over the network, decodes each one of the requests to determine which resource is needed, find the resource if it is available, and sends the resource back to the client over a network. The interaction between the client and server side is discussed in section 2.1

2.1 Web communication model

Figure 2.2 shows the interaction between the client and web server. Typically, the client is a web browser. The client initiates a request to the web server for server side resource such as HTML file or image.

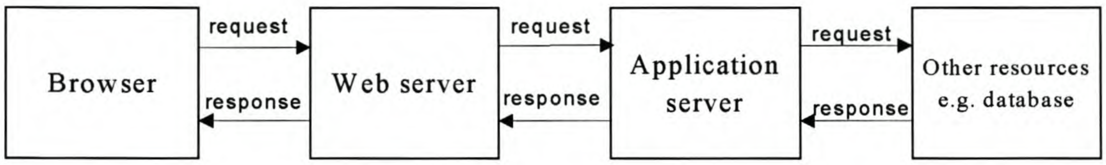

Figure 2.2 Web Application Communication Model

A web server first receives the request from the client. The web server serves the requested resources to the client. A web server processes requests from clients for static content. It cannot directly process a request for dynamic content. An example of dynamic content is data from a database system.

In such cases, the web server relies on the application server to process the request. If the request is to be processed by the application server, the request will be forwarded to the application server. Therefore, a web server can be thought of as an application that constantly looks for client requests and processes them as they arrive, either directly or in conjunction with the application server.

The application server does the processing and sends the results to the web server. The web server forwards these results to the client in the form of HTML as part of the response. The application server interacts with other necessary resources such as database in order to process requests.

In some implementations, the web server and application servers are not separated. Both the functionalities are put together in one server application, which serves both as web server and the application server.

In this thesis, the focus is on the web server communication flow. The satellite-based web server to be designed is of limited capability. The test phase will only demonstrate the ability of the web server to serve HTML files having images and text file.

2.2 Hypertext Transfer Protocol (HTTP)

In this thesis, the features of HTTP protocol are discussed from the point of view of implementing a simple web server application whose capability should be to serve the client with HTML files. The HTTP protocol is an application-level protocol for distributed, collaborative, hypermedia information systems (T.Berners-Lee, 1997).

The HTTP is based on a request/response paradigm. The client establishes connection with a server and sends a request to the server. The server responds to the client request. The connection is established by the client prior to each request and closed by server after sending the response. On the internet, HTTP communication generally takes place over Transmission Control Protocol/ Internet Protocol (TCP/IP) connections. The default port is TCP 80, but other ports can also be used.

2.2.1 HTTP Methods

The two most widely used methods for HTTP are GET and POST. The method of interest in the case of the web server to be implemented is the GET method. The GET method is used to retrieve whatever information identified by the request-URI. This information could be a simple HTML file, a text file or an image. If the request-URI refers to a dataproducing process, it is the produced data that shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

2.2.2 Status Codes

As part of the response, the web server sends a status code back to the client. The status code is a number that indicates the result of the request. Table 2.1 below shows the list of some possible status codes, which are defined by HTTP protocol.

Code	**Status**
200	OK
204	No Content
301	Moved Permanently
302	Moved Temporarily
304	Not Modified
401	Unauthorized
403	Forbidden
404	Not Found
500	Server Error
501	Not Implemented
502	Bad Gateway

Table 2.1 Status Codes

2.2.3 Web Request

A Full-Request will have, in general, the Request-Line followed by the Request-Header and entity-Header. The end of an entity-Header is marked by a CRLF character. Followed by the entity-Header is the entity body. A Request-Line message from a client to a server includes the method to be applied to the resource, the identifier of the resource, and the protocol version in use. An example of the request-Line looks like the following:

GET/Thembakazi.html/HTTP/1.0

The method GET signifies a simple-request, Thembakazi.html is the resource being requested and HTTP/1.0 is protocol version.

The request-header also allows the client to pass additional information about the request or about the client itself. The message tells the web server of the type of browser sending the request, and Multiple Internet Mail Extensions (MIME) types the browser can be able to accept. The list of MIME types can be replaced by a wildcard, *Accept:* */*, which says the browser will accept any type of information (Yeager and McGrath, 1996:32).

2.2.4 Web Response

An HTTP response message will be send back to the client after the server received and interpreted the request message. A Full-Response will have, in general, a Status-Line followed by the response-header and entity-header. The end of an entity-header is marked by the CRLF character. Followed by the entity-header is the entity body. It is the entity body that contains the requested resource.

The first line of the response is the Status-Line, which looks like the following:

HTTP-Version Status-Code Reason-Phrase CRLF
e.g., HTTP/1.0 200 OK

The Status-Line is followed by the additional information about the response referred to as metainformation, which tells the browser what it must know to interpret and display the information.

If the web server receives many requests simultaneously and only responds to one at a time, many of the requests will be delayed. For instance if the request for an image is made, which can take several seconds to be fulfilled, it will cause a delay to the other requests that can arrive during the time the server is busy serving the first request. Section 2.2.5 discusses different methods that a web server can use to process requests in parallel.

2.2.5 Serving more than one Request

Most servers handle as many requests as possible at the same time. Yeager and McGrath (1996) define three methods that a Web server can use to serve many requests at the same time. The methods are Cloning, Multithreading, and Helper server programs.

2.2.5.1 Cloning

Cloning method can be referred to as one-process-per-request. The web server program clones itself into a number of identical copies corresponding to the number of incoming requests, with each copy having its own network connection. The parent or the original copy goes back and listen to incoming requests, while the child process is busy serving the first request.

However, creating a new process for each incoming request can be relatively costly. This is because each copy of the process is scheduled with other processes, and also competes with other programs and with the operating system itself for a limited processor time.

Suppose 20 requests arrive simultaneously, the server will have to create 20 copies of itself. In case the server is busy creating processes that it cannot respond rapidly to the new requests, the users will definitely experience some delays.

The overhead caused by the creation of a new process would be necessary if a separate process was really needed for each request. However, web requests are very short and simple and do not need a process. Since cloning a new copy of the web server for each request carries such a large overhead, the next alternative method seeks to reduce this overhead.

2.2.5.2 Multiple Threads

With this method the web server is capable of keeping track of the several connections, switching between them as needed. The first request is processed, and the first part of data is read from the disk. The server records what it was doing i.e. handling the request, the port to send it to, and what has been done so far, and begins handling the second request.

The advantage of a multithreading web server is that it can be effective on a single-processor. While the program waits, the processor switches to another program or to another thread of the same program. This overlapping also happens with cloning, but because switching between threads is much faster than switching between processes, much overlapping is possible with threads than with processes.

Multiple threading reduces the overhead for each request because instead of duplicating the whole server program, it only duplicates the memory and the process control information relevant to the request. A multithreaded server eliminates much of the overhead incurred by one-process-per-request server, so it can serve more requests and serve each request faster.

Multithreading can be a very effective way of serving simple web requests, such as retrieving a web document. Additionally, Java provides built-in support multithreaded programming. Hence, the satellite-based web server will be developed in Java and will implement threads to process requests in parallel. The other advantages of Java will be discussed in section 2.3.

However, if the future enhancement of the satellite-based web server will require that the satellite-based web server service requests that execute scripts, threads cannot be used. This is because requests that execute scripts can not be able to run as a thread but require a separate process for the application server. If the server provides a lot of scripts, multithreaded server will not necessarily improve performance because the scripts do not benefit from multithreading. The next section discusses the method that can be used to handle requests in parallel and also ensure that all simple and complex requests are satisfied with minimal overhead.

2.2.5.3 Helper Server Programs

The third alternative method of serving more than one request at a time is to start few processes and have one process known as a dispatcher, dedicated to receiving requests from the network and passes them out to other processes. In this model each request is handled by a separate copy of the server, so the limitations of a multithreaded server are avoided.

One thing that makes the helper program a better method when compared with cloning is that the processes are created only once and re-used, so the overhead of process creation, which happens in cloning, is also eliminated. The dispatcher process accepts the connection from the client, passes it to the helpers and thereafter closes its copy of network connection.

The helper processes do not listen for request on the network, instead, they listen for requests from the dispatcher process. The helper processes handle only one request at a time. Therefore, they are relatively simple and they can execute scripts without any special arrangement.

The helper does not terminate after serving a request; instead it waits for another request to arrive from the dispatcher process. The number of helpers must be managed. This is because when helpers are waiting for requests, they use up memory and operating system resources. Therefore, it is advisable not to create more helpers than necessary, because the system can be slowed down.

If all helpers are busy serving requests and more requests arrive, the dispatcher program must either reject the overflow requests or queue them up until the helper is free. This is because starting an extra helper process while the others are busy can slow down the system. Therefore, queuing the requests for few seconds is better than rejecting them.

2.3 The Advantages of Java

The reason behind choosing Java for this project is because network support is built into the language and rich support for much of the details required for network communication is taken care of by the underlying virtual machine (Berg and Fritzinger, 1999). Additionally, unlike other languages, such as C in which the programmer is responsible for socket connection management, Java supports many of the tedious details of managing the socket connection through various classes of its standard Java.net package. This capability leaves the Java programmer with easier work of just opening and closing the sockets (Frederick F. Chew, 1998).

Furthermore, Java source code is compiled into bytecode, not native machine code. Each environment that supports stand-alone Java applications would have a program called virtual machine that is capable of interpreting and executing the bytecode file. From the point of view of the Java application, every operating environment appears the same. Therefore, this is portability at its finest.

2.4 Proxy Server

The implementation of the web proxy server is also one of the possibilities for the satellite-based web server. The detailed motivation for this possibility will be discussed in section 3.2 in the following chapter. This section only gives a brief description of what a proxy server is and also what it does. A proxy server is an intermediary program, which acts as both a server and a client for the purpose of making requests on behalf of other clients (T.Berners-Lee, 1997). Figure 2.3 shows the block diagram of a proxy server communication model.

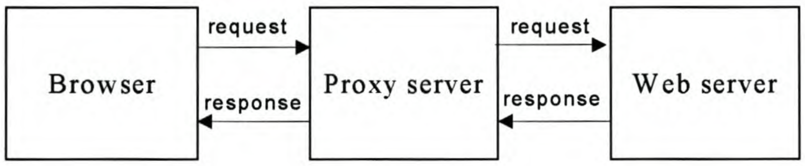

Figure 2.3 Proxy server communication model

Requests are serviced internally or by passing them on, with possible translation, to other servers. A proxy server must implement both the client and server requirements in order to meet the above-mentioned functionality.

2.5 Data Link Control Protocol

Since there is no satellite upon which the web server is to be implemented, this concept can be demonstrated on two personal computers. In order for the two personal computers to be able to exchange data, a protocol will be needed. This section explores some of techniques that can be used to implement data communication protocols. The discussion includes flow control, error detection and error control.

2.5.1 Flow Control

Flow control is the technique for speed matching of the transmitter and receiver. It ensures that the transmitter does not overflow the receiver with data. The receiver allocates a data buffer of some maximum length for transfer. When data is received, the receiver has to spend some time validating the received data before passing it over to the high-level software. In the absence of flow control, the receiver's buffer may be filled up and overflow while the receiver is busy processing the old data. The next two sections discuss the operation of some commonly used flow control protocols.

2.5.2 Stop-and-Wait

In stop-and-wait flow control, the receiver indicates its readiness to receive each packet frame by sending an acknowledgment to the sender. The sender only sends the next packet frame after receiving an acknowledgment confirming the receipt of the previous packet frame. The receiver has the potential to stop the flow of data by not acknowledging the received packet frame. Stop-and-wait flow control works fine when a large block of data is broken down into small blocks and sends data in many packet frames. When a large block of data is broken into small blocks, errors are detected sooner and smaller amount of data needs to be retransmitted. The major drawback of the stop-and-wait flow control is that only one packet frame can be in transit at a time, which leads to inefficiency if propagation time is greater than the transmission time.

2.5.3 Sliding-Window

The problem with stop-and-wait flow control is that only one packet frame can be in transit. In situations wherein propagation time[*] is greater than the transmission time[+] serious inefficiencies result. One way to improve the efficiency is to allow multiple packet frames to be in transit. This can be achieved by connecting the sender and receiver via the full-duplex link. The receiver allocates buffer for W packet frames. The sender is allowed to send W packet frames without waiting for an acknowledgment.

The packet frames are labeled with a sequence number in order to keep track of which packet frame has been acknowledged. The receiver acknowledges the frames by sending an acknowledgment, which includes the sequence number of the next frame the receiver is expecting to receive. The sender sends the next frames starting with the frame whose sequence number is specified in the acknowledgment.

Sliding-window flow control is potentially much more efficient than stop-and -wait flow control. The reason being that, with sliding-window flow control, the transmission link is treated as a pipeline that may be filled with packet frames in transit.

2.5.4 Error Control

The types of errors that occur during the flow of data are damaged frames, and lost frames. Most error control techniques are based on error detection scheme and retransmission scheme. The error control schemes that involve detection and retransmission of lost corrupted packet frames are referred to as automatic repeat request (ARQ) error control. The most common ARQ retransmission schemes are Stop-and-Wait

[*] Propagation time is the time it takes for a bit to travel from the sender to receiver.

[+] Transmission time is the time it takes a station to transmit a frame.

ARQ, Go-Back-N ARQ, and Selective Repeat ARQ and are discussed in the next three sections.

2.5.4.1 Stop-and-Wait ARQ

Stop-and-wait ARQ is based on the stop-and-wait flow control technique. The sender sends a packet frame and wait for an acknowledgment before sending the next frame. The receiver sends an acknowledgment if a packet frame is correctly received. Two sorts of errors can occur during transmission of data. The first error could be that a packet frame that arrives at the receiver is damaged. The receiver detects an error and discards the damaged packet frame. The sender takes care of this kind of situation by setting the timer. After a packet frame is sent, the sender waits for an acknowledgment and if no acknowledgment is received until the timer expires, the sender retransmits the packet frame. This method requires that the sender maintain the copy of the transmitted packet frame until an acknowledgment is received.

The second possible error that may occur is a damaged acknowledgment. In such cases, the sender times out to allow retransmission of an unacknowledged packet frame. However, retransmission of an unacknowledged packet frame duplicates the packet frame accepted by the receiver. In order to avoid this problem, frames are alternatively labeled with 0 and 1. The positive acknowledgments are of the form ACK0 and ACK1. ACK0 acknowledges the receipt of packet frame1 to indicate to the sender that the receiver is ready to receive packet frame numbered 0.

The main advantage of stop-and-wait ARQ is its simplicity. However, only one packet frame can be in transit at a time, and if the propagation time is greater than the transmission time, the transmission line is under utilized.

2.5.4.2 Go-Back-N ARQ

Go-Back-N ARQ uses the sliding window flow control protocol. If no errors occur the operations are identical to sliding window. The sender sends multiple frames allowed by the window size. In case the receiver detects an error in a particular frame, it sends a negative acknowledgment for that frame. The receiver will then discard that frame in error and future incoming frames until the frame in error is correctly retransmitted. Therefore, the sender, when it receives a negative acknowledgment, must retransmit the frame in error plus all succeeding frames that were sent before with the frame that was found to be in error. If an acknowledgment gets lost, the sender times out to allow retransmission of the packet frames.

2.5.4.3 Selective-Back ARQ

Selective-Back ARQ is similar to Go-Back-N ARQ. However, the sender only retransmits packet frames for which the negative acknowledgment is received. The advantage of Selective-Back ARQ is that retransmission's are fewer. The disadvantages of this form of error control are the complexity of the sender and receiver, each packet frame should be acknowledged because there is no accumulative acknowledgment, and the receiver may receive the frames out of sequence.

2.6 Summary

Handling multiple requests by means of cloning might create a system deadlock if a dozen of requests can arrive simultaneously. This is because the arrival of each request implies the creation of a new process. Multithreaded server unlike multiple-process server uses one process to handle more requests. It dramatically reduces the overhead for each request because only the memory and process control information relevant to the request are duplicated. However threads are only best for simple request. A request to execute a script will not be able to run on a thread, but will need a separate process. Using helper program to handle multiple requests seems to be the best way, because processes are created once and re-used. In situations whereby more requests arrive while all helper processes are busy serving some other request, the dispatcher process places the requests in a queue until a helper program is free.

Stop-and-wait and sliding-window flow control techniques were also discussed in this chapter. Stop-and-wait flow control technique was then chosen for the implementation of serial communication protocol. The next chapter describes and evaluates design options of the satellite-based web server.

Chapter3

3 System Design Options

The satellite-based web server is intended to provide a simple way of accessing telemetry information of the satellite using a standard client browser such as Internet Explorer as a way of bringing space technology awareness to the community. The client browser located on the ground station should be able to retrieve telemetry information from the satellite-based web server. This chapter describes two possible designs namely, on-board Web server, and the on-board File server with the Web Proxy server on the ground - that can be used to achieve the objective of this project. It thereafter proposes, based on evaluation, the design that will be implemented.

3.1 On-board Web server

Figure 3.1 depicts the block diagram of the on-board web server design. In this design the developed web server system will be ported to the on-board computer (OBC) of the satellite. This way the client browser located on the ground segment with necessary communication requirements will have a direct connection with the on-board web server. Upon receiving the request for the telemetry resource, the web server will then be able to retrieve the requested resource from the disk storage of the satellite, which is in the text format, and return it to client browser. In order to specify more details of how the telemetry text document should be displayed on the browser, the Hypertext Makeup Language (HTML) conventional text marks will be used. The HTML conventional marks will also make it possible for any web browser to display the telemetry information.

The implementation of the satellite-based web server system that will allow all standard web browser to connect and retrieve telemetry information requires that TCP/IP network software be also implemented on the OBC of the satellite. This is because all web servers must use TCP/IP network software, which governs network connections and follows the rules of HTTP (Yeager and McGrath, 1996). The HTTP is the protocol used by the World Wide Web to define how the web servers and the web browsers should communicate over a TCP/IP connection.

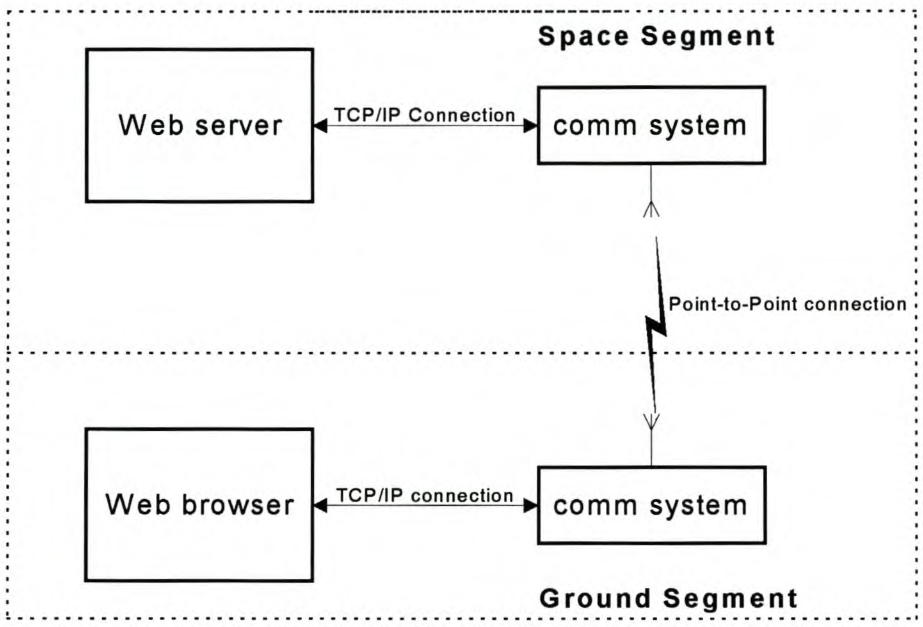

Figure 3.1 On-board web server block diagram

The advantages of this design are that, firstly, the client browsers do not need any modification in order to connect to the on-board web server since this kind of implementation is the standard way of implementing the web server system. Secondly, the TCP/IP protocol stack design is available hence its implementation can be done much easier than it could if the design was part of the implementation. Lastly, the TCP/IP protocol is standard; hence its future enhancement will not be much challenging to the programmers, as it would be with a non-standard protocol.

However, this design will only be best for geostationary satellites. Geostationary satellites are always within the range of access, such that users will be able to access web resource at all times. However, for Low Earth Orbit satellites, users will have limited access time since they are not always within the range of access. Therefore, users will only have to connect and retrieve the telemetry resource when the satellite is within the range of access.

Additionally, this design will be costly for the users. This is because each user will need the communication system for communication with the on-board web server. Furthermore, the TCP/IP network software was developed to be a protocol upon which the bulk of new protocol development is to be done. Hence, implementing the TCP/IP protocol on the satellite to only satisfy web server requirements will be a waste of resources unless it would be used for all communications in the satellite.

In order to bypass the implementation of the TCP/IP network software, the on-board web server can be replaced with the File server system, with the addition of the web proxy server on the ground segment. The detailed description of how the on-board File server will function to achieve the objective of the project is discussed in section 3.2.

3.2 On-board File server with Web proxy Server

The implementation of the TCP/IP network software, as a necessity for the on-board web server on the OBC of the satellite, can be bypassed by having the File server system ported to the satellite with the addition of the web proxy server on the ground segment. Figure 3.2 shows the block diagram of this design. The File server will be the software, which upon receiving the name of the requested resource, locates the resource from the OBC storage disk and breaks it into a packet format required for a data communication protocol that will be used.

The web proxy server on the ground segment will be responsible for forwarding the request from the client browser to the on-board File server and also for relaying the response from the on-board File server to the client browser. Additionally, since the telemetry information will be received from the on-board File server in the text form, the web proxy server will use the HTML conventional text marks to format a response before relaying it to the client browser. This way it is simple to provide the client browser with the telemetry information without having to port the web server to the satellite.

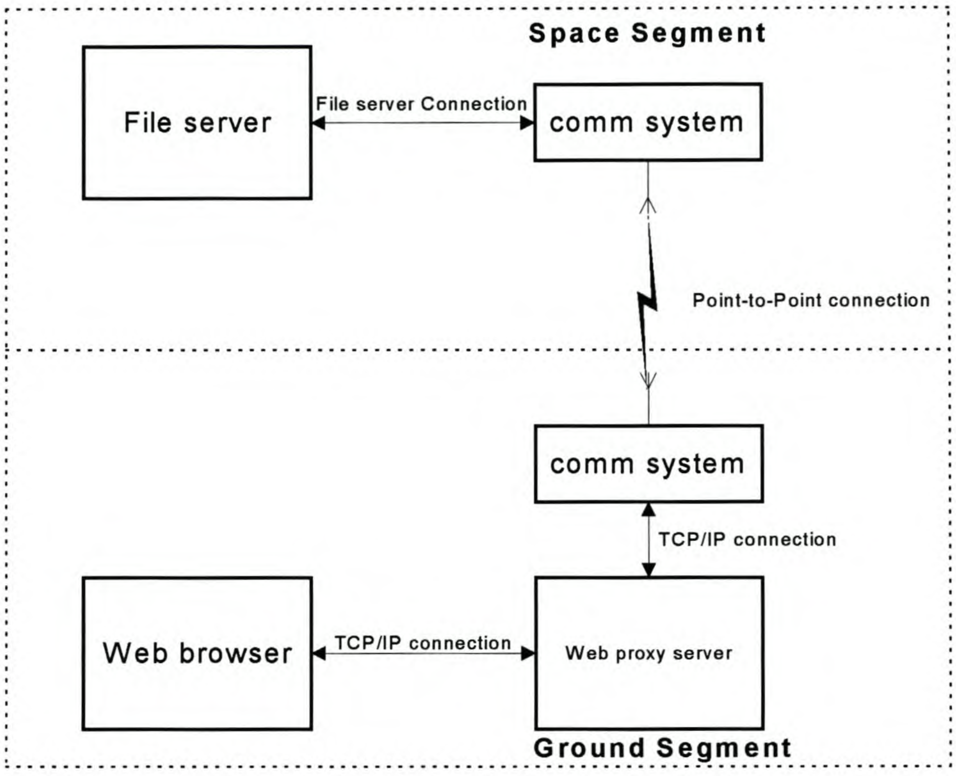

Figure 3.2 On-board File server with web proxy server

This design can be further developed to improve the access time of the Low Earth Orbit satellites. This can be accomplished by the use of multiple ground stations to serve as relays between the on-board File server and the client browsers. Each ground station will consists of the web proxy and the communication requirements necessary for communication with the on-board File server. Among these ground stations, there will be one ground station called main station to which the client browsers will connect. The rest are referred to as sub-stations and will be directly connected to the main station and also to the on-board File server. Like the sub-stations, the main station also connects to the on-board File server. Figure 3.3 shows how this might work. When the satellite gets out of the access range of the main station, the next sub-station captures it. The sub-station within the access range will communicate with the main station so that any request coming from the client browser at that particular time should not be forwarded to the on-board File server from the main station, but through the sub-station within the range of access. In this way, all requests will be served.

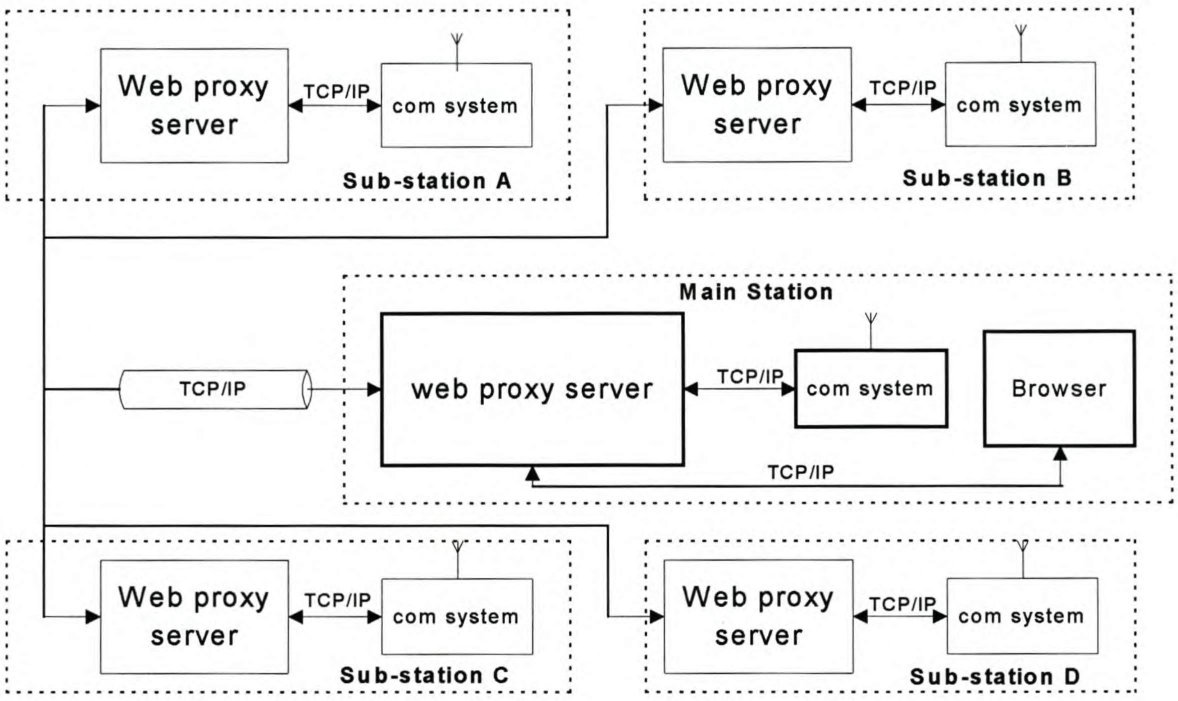

Figure 3.3 Multiple Ground Stations

Having the File server ported to the OBC instead of the web server, it is not necessary to have the TCP/IP network software implemented on the satellite as already mentioned. The web proxy server will be on the ground segment running on the standard computer using the existing TCP/IP network software to connect with the client browser. Additionally, having the web proxy server on the ground segment to serve as an intermediate between the client browser and on-board File server overcomes the communication requirements cost for users because they will all communicate with on-board File server through the web proxy server. This implies that the web proxy server will be the only system that will use the communication system required for communication with the satellite. Therefore, based on the fact that the File server can be implemented without TCP/IP, this design was then proposed for the implementation of the satellite-based web server.

In the absence of the satellite upon which the proposed design is to be implemented, it was then decided to use two personal computers to implement and also to demonstrate the operation of the system. One computer will run the File server system to replace the OBC of the satellite. The other computer will run the web proxy server with the client browser software. The use of two personal computers to implement the satellite-based web server requires that a data communication protocol be developed to provide a communication link between the web proxy server computer and File server computer. Therefore, simple error recovery half-duplex serial communication protocol based on stop-and-wait technique will then be developed and used to transfer the file from the File server to the web proxy server.

The reasons for choosing stop-and-wait technique are that, firstly, stop-and-wait flow control protocol is simple to implement. Secondly, the objective of this thesis is to develop a satellite-based web server, not a protocol, hence the development of the serial communication protocol is only for the demonstration of the final software.

3.3 Summary

The on-board web server will only be best for geostationary satellites. This is because the geostationary satellites are always within the range of access, which implies that users will be able to access it at all times. However, the on-board web server will require that the TCP/IP protocol be implemented for the reasons mentioned in section 3.1. Contrary to the on-board web server, the on-board File server with the web proxy server is cost effective since only one communication system will be needed in order for all users to retrieve telemetry information. Furthermore, the on-board File server with the proxy server design can be done without TCP/IP implementation. Therefore, based on the fact that the implementation of the File server on the OBC can be done without TCP/IP network software, this design will be implemented. In the absence of the satellite upon which the File server should be implemented, a personal computer will be used to replace the satellite. The next chapter focuses on the implementation of the all software components of the proposed design.

Chapter 4

4 Implementation

This chapter provides the implementation of the proposed design of the satellite-based web server. The satellite-based web server is composed of three software components. The components are Web proxy server, File server and the serial communication protocol (SCP). Section 4.1 describes the operation of the assembled system. The rest of this chapter discusses the detailed operation and the implementation of each software component.

4.1 Satellite-based Web server System

As already mentioned the satellite-based web server consists of the web proxy server, File server and the SCP software components. The SCP component is the protocol that provides communication link between the web proxy server and File server. Figure 4.1 shows the data flow diagram of the satellite-based web server software. The web proxy server starts by creating the server socket and binds it to port 80, the default port for web service. Once the server socket is created, the proxy server continues to listen for incoming connection. When the connection is established, the proxy server creates a thread whose task is to handle the input and output responsibilities for that particular connection. Once a thread is created, the proxy server returns to wait for a new connection.

The created proxy server thread continues to read the request from the client socket and forwards this request to the File server through the SCP port. The File server reads the request from the SCP port, produce the response and writes it to the SCP port.

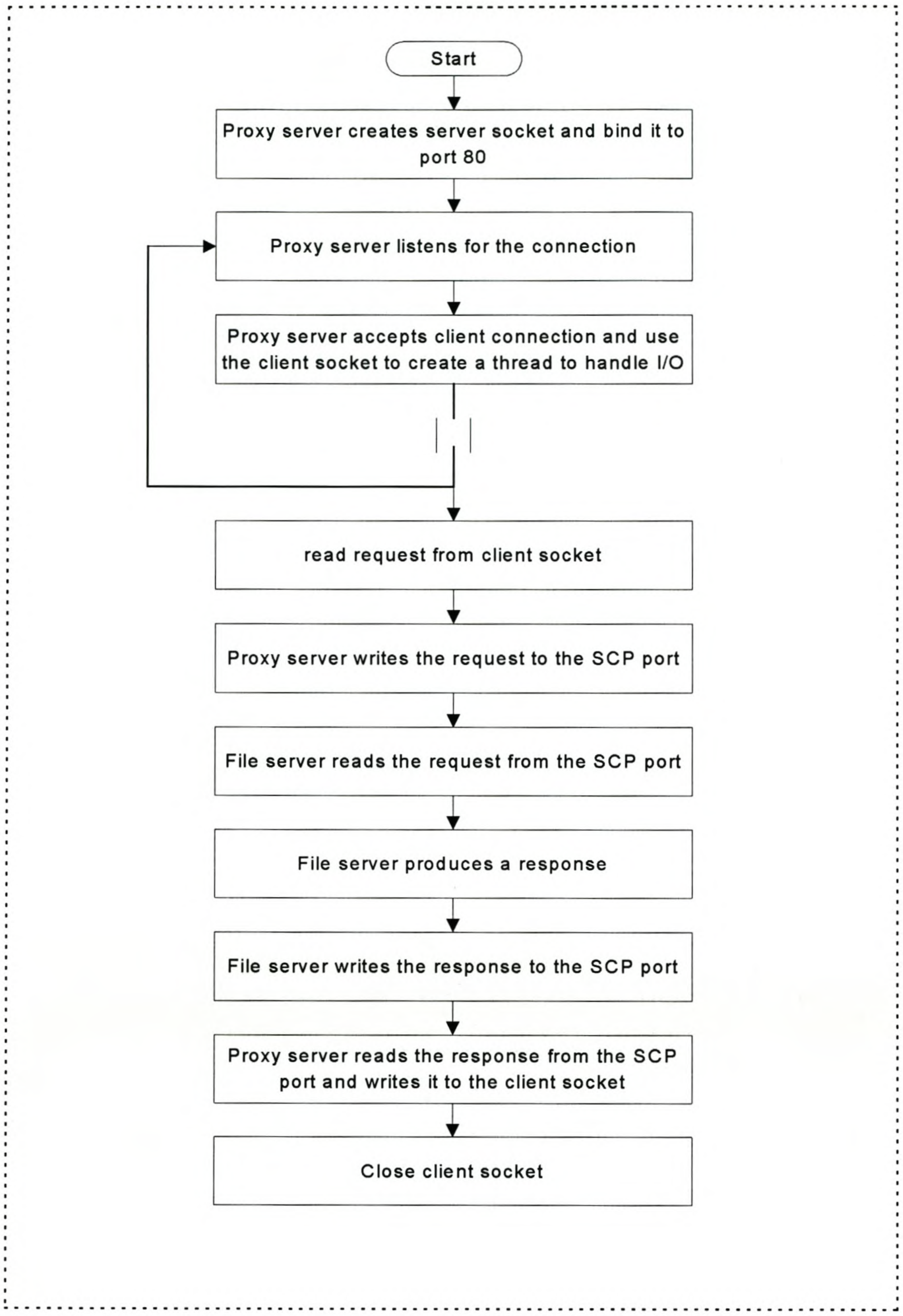
Start
Proxy server creates server socket and bind it to port 80
Proxy server listens for the connection
Proxy server accepts client connection and use the client socket to create a thread to handle I/O
read request from client socket
Proxy server writes the request to the SCP port
File server reads the request from the SCP port
File server produces a response
File server writes the response to the SCP port
Proxy server reads the response from the SCP port and writes it to the client socket
Close client socket

Figure 4.1 Satellite-based web server data flow diagram

The proxy server thread then reads the response from the SCP port and writes it to the client socket. When the proxy server thread finishes writing the response to the client socket, it then closes the client socket and simply goes away.

4.2 Web proxy server

The web proxy server serves as an intermediate between the client browser and the File server, forwarding the request from the browser to the File server and returning the response from the File server to the client browser. It consists of four classes. The classes are Web_proxy_server, which is the main class, ConnectionAcceptor, which is dedicated to listening for the incoming connections, RequestHandler class, which is the thread class responsible for input and output and the Receiver class whose job is to write the name of the requested resource to the SCP port and to read the response from the SCP port. The class diagram of the web proxy server is shown in figure 4.2.

When the Web_proxy_server class is started, it issues a method call to the ConnectionAcceptor class which creates a server socket on port 80, the default port for web service, and call the accept () method to wait for a new connection. When a request for connection arrives, the accept () method returns a socket which is then used to create a new thread (RequestHandler class) whose task is to serve that particular client. For every request, the ConnectionAcceptor class creates an instance of RequestHandler thread and starts it. This allows the Web proxy server system to process the requests in parallel.

The RequestHandler class uses the socket produced by the connection to create an InputStream and OutputStream. Inside the run () method, the RequestHandler class invokes the processingRequest () method whose task is to check the name of the requested web resource. The web proxy server can only process the GET command from the web browser, hence the name of the requested web resource could be of the form GET/index.html HTTP/1.1. The processingRequest () method invokes the receive ()

method of the Receiver class to forward the name of the requested web resource to the File server and to receive the response from the File server. The detailed description of the Receiver class will be given in section 4.4

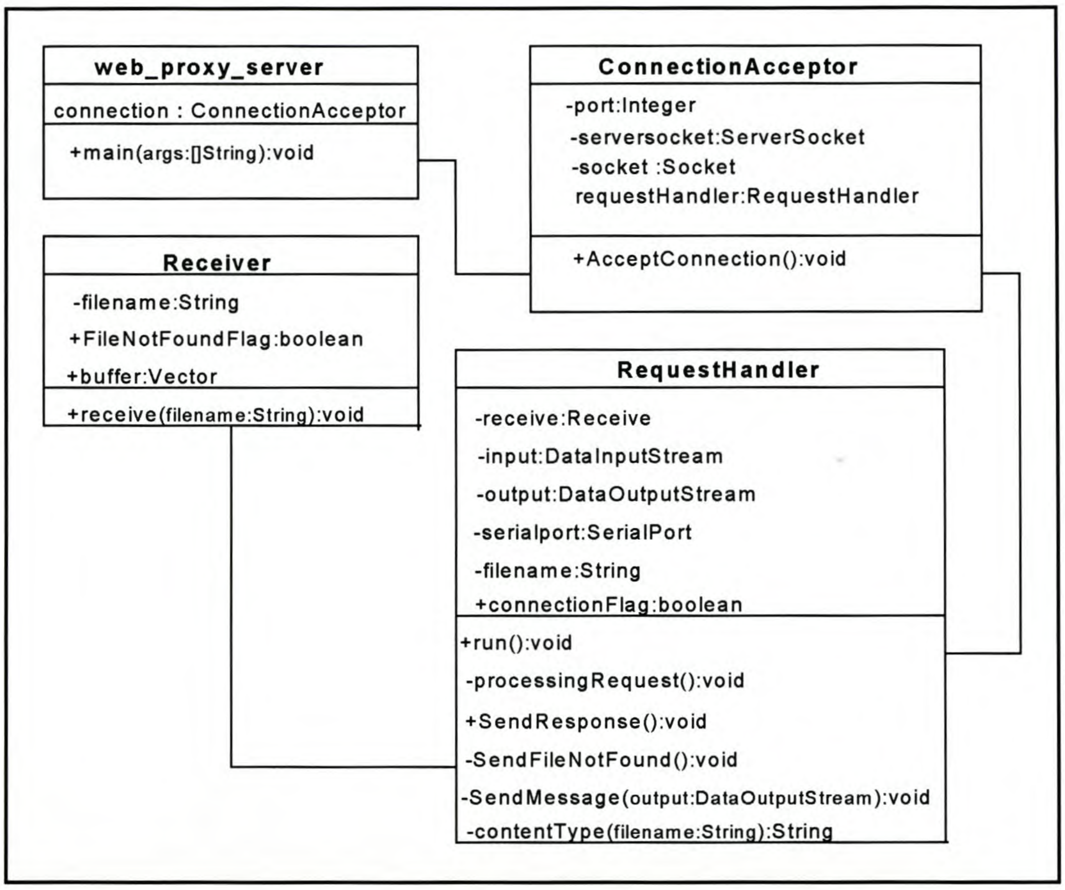

Figure 4.2 Web proxy server class diagram

When the response is completely downloaded, the processingRequest () method invokes the SendResponse () method, which sends the appropriate header to the web browser. The actual resource transmission begins after the headers are sent. If the requested web resource is not found, the processindRequest () method invokes the SendFileNotFound () method, which invokes the SendMessage () method to format the error message 404 and send it to the client browser. The error message 404 informs the web browser that the requested web resource is not found.

As already mentioned the web proxy server processes the requests in parallel. However, the developed SCP software component is limited to handling only one request at a time. Once the requested web resource is in transit, any attempt by the web proxy server of forwarding the request that arrives before the first resource is completely downloaded, causes the File server to receive a request in the wrong state, which is the state wherein the File server is expecting an acknowledgment instead of the name of the web resource. The reception of the request in the wrong state causes the File server to disregard the request and consider it as an error. Therefore, in order to ensure that all requests forwarded to the File server are considered as requests and not ignored, the web proxy server should only forward a request when the first requested resource is completely downloaded and the File server has returned to the initial state to wait for the next request.

In order to achieve this, the RequestHandler class declared a global flag and uses it to control access of the shared resource. When the web proxy server receives a request, it sets the flag to block the request that arrives before the first requested resource is completely downloaded. The blocking continues until the resource in transit is completely downloaded. When the web proxy server receives the end of transmission byte indicating that the resource is completely downloaded, it then reset the flag to release the blocked request. Therefore, once the requested resource is in transit, there is no other request that can interfere until the resource or the requested resource is completely downloaded. The next section describes the operation and the implementation of the File server system.

4.3 File server

The File server system consists of three classes, the File_server class, Transmitter class, and the SettingDefault class as shown in figure 4.3. When the File server is started, it creates and starts the transmitter thread class. Inside the run () method, the transmitter thread keeps on polling, waiting for the name of the resource to be forwarded by the web proxy server. When the request arrives, the transmitter passes it as a parameter to the resourceLocator() method of the File server class.

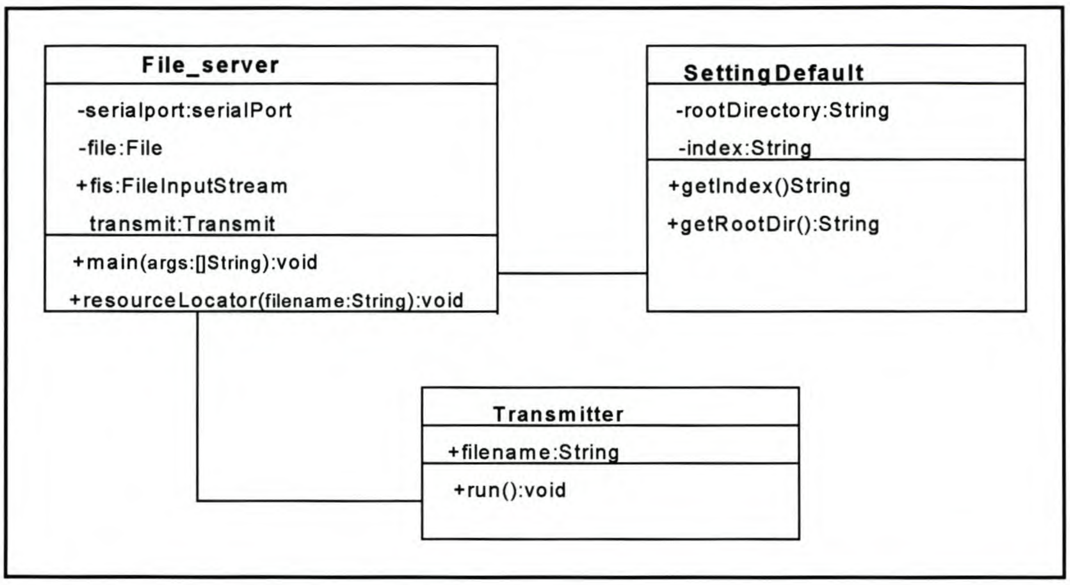

Figure 4.3 File server class diagram

Inside the resourceLocator() method, the File_server class creates a new file object and invokes the getRootDir() method from the SettingDefault class to return the root directory of the requested resource. The File_server continues using the created file object to determine whether the requested web resource is a directory or plain file. If the requested resource is a directory, the File_server class invokes the getIndex () method from the class SettingDefault, which returns the index file. If the resource of interest is a plain file, the File_server class continues, using the created file object, to check if the requested file exists. Checking the existence of the file is important because it is possible to create a file object even if the file referred to by the created object does not exist.

In case the file does not exist, the File_server, through the transmitter thread class, sends an error message byte to the web proxy server. Upon receiving the error message byte, the web proxy server constructs the error message and sends it to the client browser. Otherwise, the File_server continues to check if it has permission to read the file. If the reading permission is granted, the File_server opens the FileInputStream, reads it into packets and send them to the web proxy server through the transmitter thread. When the requested file resource is completely sent the File server closes the file and returns to the initial state to wait for the next request.

4.4 Serial Communication Protocol (SCP)

The SCP is an error recovery protocol, which provides a communication link between the web proxy server computer and File server computer via serial, asynchronous communications. It is based on the stop-and-wait flow control technique as mentioned in section 3.2. The transmitter transmits a single packet and then waits for an acknowledgment before sending the next packet. No other packets can be sent until the receiver's reply arrives at the transmitter. The SCP consists of two layers, the byte-level layer and packet-level layer as shown in figure 4.4

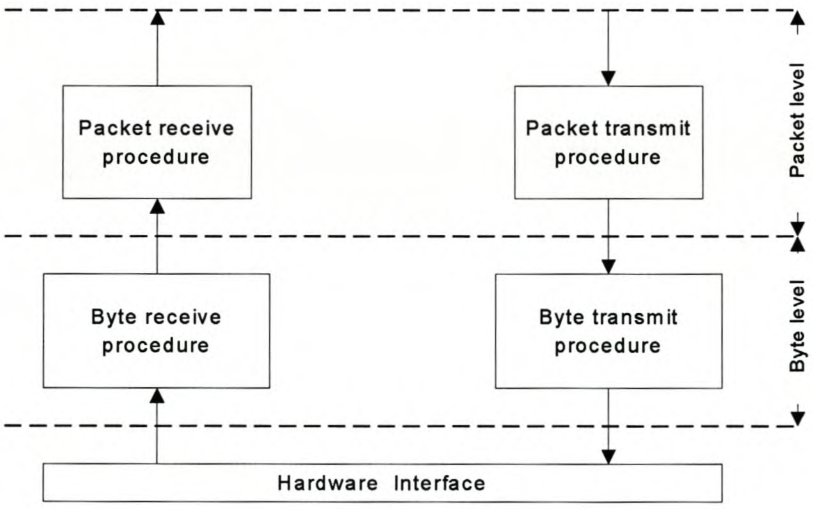

Figure 4.4 SCP Layers

The byte-level layer uses status-polling mechanism to access serial port for either reading or writing data. The status-polling method requires that the byte-level layer obtains the status of the interface prior to transferring data, in order to determine if it is allowed to read from or write to the interface. The status is typically obtained by reading a status register from the interface. Polling mechanism is simple to write and debug, and the response time is easy to determine. However, polling mechanism wastes processor time, if an interface produces short bursts of data, separated by long period of inactivity.

With event-driven mechanism the processor waits for an event to occur in the form of new data in the input buffer. As soon as new data arrives in the buffer, it sends an event to the processor. This way processor time is preserved. The polling mechanism was chosen because of its simplicity. In addition, the protocol was only developed for the demonstration of the final software of the satellite-based web server whose objective was to demonstrate the accessibility of the telemetry information through the web browser software.

The use of polling mechanism caused some bytes to be overwritten. This happened when the transmitter was faster than the receiver was. Therefore, in order to ensure that all bytes transmitted reach their destination, the byte-level layer acknowledges each byte sent across. The transmitting procedure of the byte-level layer application sends a byte and waits for an acknowledgment before sending the next byte. The receiving procedure receives a byte and acknowledges, indicating to the transmitter that it is ready for the next byte.

The packet-level layer is responsible for the packet creation and packet validation processes. It consists of two classes, the transmitter class and receiver class. The transmitter class takes the data, creates packets and passes them to the byte-level layer, which does the physical transmission of data over the hardware interface. When the packet arrives at the destination, the receiver through the byte-level layer reads the data into the buffer, validates it and thereafter passes it to high level. The next two sections describe the detailed operation of the transmitter class and the receiver class.

4.4.1 Receiver

The receiver drives the protocol. When the SCP protocol is started, the transmitter waits to receive the name of the requested file from the receiver. When the request packet arrives, the transmitter returns an acknowledgment, which informs the receiver to get the ready to receive the first packet of the requested resource. Figure 4.5 depicts the state transition of the receiver class.

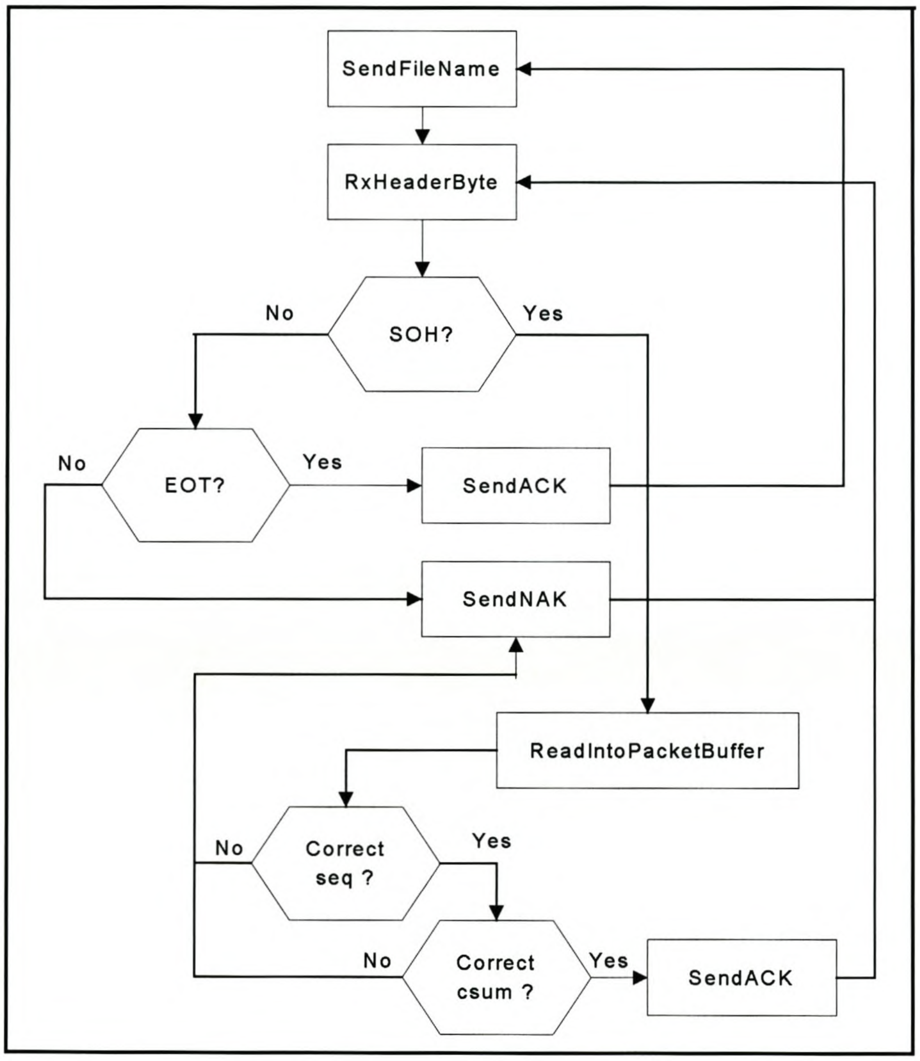

Figure 4.5 Receiver state transition

The operation of the receiver is achieved in five states:

- **SendFileName**. In this state, the receiver takes the name of the requested file, creates a packet and sends it over to the transmitter. When the request packet is sent, the current state is changed to RxHeaderByte state.
- **RxHeaderByte**. This is the state wherein the receiver reads the first byte from the transmitter. This byte should either be the SCP packet header or the end of transmission byte. If the first received byte is neither header byte nor end of transmission byte, the current state is changed to SendNAK state. When the receiver receives the header byte, the current state advances to ReadIntoBuffer state. However, if the receiver receives the end of the transmission byte, the current state is changed to SendACK state.
- **ReadeIntoBuffer**. In this state the receiver allocates the buffer for the incoming SCP packet and reads the data into the allocated buffer. When the packet is completely received, the receiver validates the packet to determine if it is correctly received. It does this by comparing its own computed checksum with the checksum sent by the transmitter. If the checksums match, the receiver knows that the packet is error-free, and then changes the current state to SendACK state. However, if the checksums do not match, the receiver knows that the received packet is incorrect. It then discards the bad packet and changes the current state to SendNAK state.
- **SendNAK**. The receiver enters the SendNAK state whenever data is received with an error. In this state, the receiver sends the byte that informs the transmitter to retransmit data that was received in error. When the retransmission byte is sent, the current state is reset to the RxHeaderByte state.
- **SendACK**. In this state, the receiver returns an acknowledgment to the transmitter to inform it that the transmitted packet was received without an error. With an acknowledgment, the receiver also informs the transmitter to send the next data packet. When the receiver finishes sending an acknowledgment byte, the current state is reset back to RxHeaderByte state to wait for the next data packet. The last byte the receiver must receive is the end of transmission byte. When the receiver receives the end of transmission byte, it knows there is no more data coming. It then

acknowledges the end of transmission byte and reset the current state to the SendFileName state to wait for the next request.

4.4.2 Transmitter

The transmitter takes the data, divides it into 397 byte pieces and places it in the SCP packet. The SCP packet takes 397 bytes and three flow control bytes. Therefore the size of the SCP packet is 400 bytes (see section 4.4.4). The flow control bytes are, start of header (SOH), sequence number (SEQ), and checksum byte (CSUM). Figure 4.6 depicts the state transition of the transmitter.

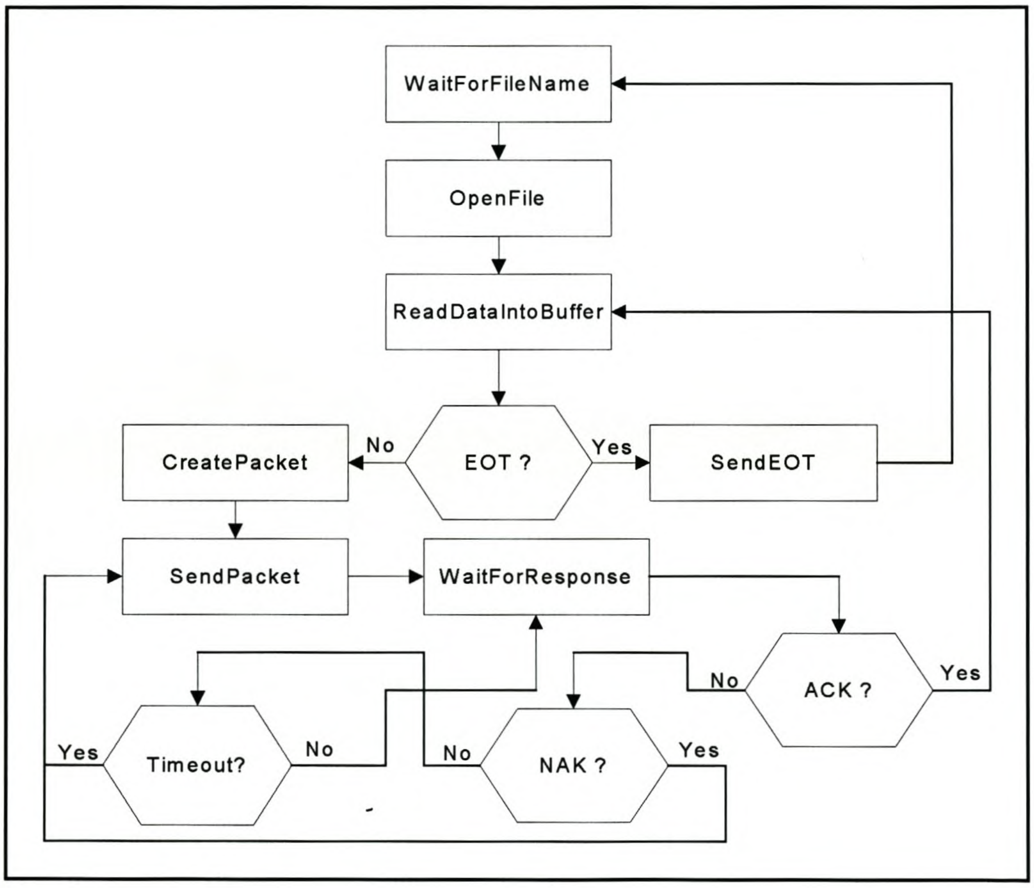

Figure 4.6 Transmitter state transition

The operation of the transmitter is achieved in six states:

- **WaitForFileName**. In this state the transmitter waits for the resource request packet. When the request packet arrives, the transmitter compares its own checksum computation with the checksum sent by the transmitter. If the checksums do not match, the transmitter replies the receiver with the negative acknowledgment to inform it to retransmit the request packet and the current state is reset back to WaitForFileName state. However, if the checksums match, the transmitter replies the receiver with the positive acknowledgment, and thereafter advances the current state to OpenFile state.
- **OpenFile**. In this state the transmitter opens the requested file resource, reads data and stores it into the allocated buffer. When the allocated buffer is full, the current state is changed to CreatePacket state.
- **CreatePacket**. In this state the transmitter takes data that was stored in the buffer and creates the SCP packet. When the creation of the SCP packet is complete, the current state is advanced to SendPacket state.
- **SendPacket**. In this state, the created SCP packet is sent over to the receiver. After the SCP packet is completely sent, the current state is changed to WaitForResponse state. Once the transmitter has transmitted the last SCP packet and has received an acknowledgment, the current state is advanced to SendEOT state.
- **WaitForResponse**. In this state the transmitter waits for the response from the receiver about the transmitted SCP packet. There are two possible responses that the transmitter should expect. The responses are negative acknowledgment and the positive acknowledgment. When the transmitter receives the positive acknowledgment, it knows the SCP packet was received without errors. The transmitter changes the current state to CreatePacket state to create a next SCP packet. If the response is the negative acknowledgment, the transmitter knows that something went wrong about the transmitted SCP packet. The transmitter changes the current state to SendPacket state to retransmit the packet. If no response is received about the transmitted packet, the transmitter uses the timer to avoid the system from hanging up. The detailed description about the operation of the timer is given in section 4.4.3.

- **SendEOT**. In this state the transmitter sends an end of transmission byte (EOT) and then waits for the final acknowledgment from the receiver before going to the initial state. When the transmitter receives the final acknowledgment, the current state is reset back to WaitForFileName state to wait for the next request.

4.4.3 SCP Error Recovery

Error recovery is the method that the SCP uses to detect and correct errors that occurs during the transmission of the data packets. Two sorts of errors could occur during the transmission. The first error could be the packet that arrives at the destination damaged. The receiver detects the damaged packet using the checksum technique and simply discards the damaged packet. To recover the damaged packet, the receiver sends a negative acknowledgment to the transmitter, which resend the packet. The second error is a lost or damaged acknowledgment. To account for the lost or the damaged acknowledgment, the transmitter is equipped with the timer.

When the transmitter sends a packet, the timer is started. The timer is set to go off after an interval long enough for the packet to reach the destination, be processed, and have an acknowledgment sent back to the transmitter. When the transmitter receives the acknowledgment, it cancels the timer. However, if no acknowledgment is received until the timer expires, the same packet is sent again. If the packet is correctly received but its acknowledgment got damaged, such that is not recognizable by the transmitter, the transmitter times out and resend the same packet. This duplicate packet arrives and is accepted by the receiver. The receiver has therefore two copies of the same packet. To avoid this problem, the transmitter assigns a sequence number to each SCP packet. The sequence number is checked to confirm that the packet is not a duplicate of the one already received.

4.4.4 SCP Packet Size

As already mentioned, the size of the SCP packet is 400 bytes. The packet size was decided upon after performing a packet size experiment using different packet sizes. In this experiment, one file was downloaded over and over again until the desirable average download time for each chosen packet size was obtained. Each packet includes three flow control bytes, which are used to detect and to discard, damaged and duplicated data packets.

The elapsed times to download and process the requested file were measured by instrumenting the SCP source code. The code used to measure the elapsed time was inserted into the SCP source code and SCP source code was recompiled. The time was measured using getTime() call of Date.util Java package. The getTime() method call returns the value of the system clock, which should be accurate to at least one millisecond.

Packet size	**File download time**	**Flow control overhead**	**File size**
20 bytes	97.6 seconds	15%	28.2 KB
50 bytes	75.3 seconds	6%	28.2 KB
100 bytes	70.8 seconds	3%	28.2 KB
200 bytes	69.2 seconds	1.5%	28.2 KB
400 bytes	68.2 seconds	0.75%	28.2 KB
800 bytes	70 seconds	0.375%	28.2 KB
1000 bytes	71.5 seconds	0.3 %	28.2 KB
1500 bytes	73.25 seconds	0.2 %	28.2 KB
2000 bytes	74.5 seconds	0.15%	28.2 KB

Table 4.1 packet size experiment results

The times were read when the request was sent and after the requested file resource was completely downloaded. The time read when the request was sent was stored in startTime variable and the time read after the requested file was completely downloaded was stored in finishTime variable. The difference between the two times is the elapsed time of the request, or the file download time. The experimental trials were repeated with the instrumented SCP source code until the desirable times were obtained. Table 4.1 shows the median elapsed times for each chosen packet size.

The results show that the small packet size requires many overhead bytes, which increases the download time of the requested resource. When the packet size is increased, the overhead bytes and the file download time were reduced. This was the case from packet size of 20 bytes to 400 bytes. However, beyond 400 bytes, increasing the packet size did not decrease the download time any longer, instead the download time was observed to be gradually increasing. This was because the procedure used to read the time of the system clock has the accuracy of one millisecond, which means any time less than one millisecond was read as zero time. Therefore, when the packet size was increased beyond 400 bytes, the packet processing time, which was zero before, became a considerable value. The packet processing time was added to the download time. The addition of the packet processing time gradually increased the download time.

4.5 Summary

The web proxy server acts as an intermediate server between the File server and the client browser. It forwards all the requests from the client browser to the File server and also returns a response from the File server to the client browser. The File server receives the request from the web proxy server via the SCP, locates the requested file resource and packetizes it according to the rules of the SCP and returns it to the web proxy server which eventually relay it to the client browser. The SCP provides the communication link between the web proxy server and the File server. It is based on stop-and-wait flow control technique. The transmitter transmits a single packet and then waits for an acknowledgment before sending the next packet. The next chapter discusses the results obtained during the functional test of the final software and concludes by presenting the future enhancements of the system.

Chapter 5

5 Results and Conclusion

The satellite-based web server software discussed in this thesis has operated successfully when tested on two personal computers. This chapter will discuss to what degree the satellite-based web server has met the objective of the project. The problems encountered during the implementation and their solutions are also presented. The chapter concludes by presenting the future enhancements of the satellite-based web server.

5.1 Satellite-based web server Operation

The satellite-based web server developed in this thesis has proven, during the test performed on two personal computers, to meet the objective of the project, which was to provide telemetry information to the client browser as a way of bringing space technology awareness to the community. The HTML files were successfully downloaded without any complication. As already indicated the satellite-based web server software consists of three software components, the web proxy server, SCP and the File server. The next three sections present operational results of each software component starting with the web proxy server component.

5.1.1 Web proxy server Software Component

As already described in section 3.2 the web proxy server serves as an intermediate between the client browser and the File server system. The web proxy server software operated successfully in forwarding the request and also in relaying the response to the client browser. The capability of the web proxy server to handle or process requests in parallel made it possible for the images embedded on the HTML files to be automatically downloaded. However, having the SCP protocol limited to handling one request at a time as described in section 4.2 has seemed to be a problem to the web proxy server while conducting the first operational test. Whenever a new connection arrives the web proxy server, as its normal way of operation will try to forward the request to the File server even if the first requested resource is not yet completely downloaded. This caused a conflict on the SCP as the shared resource.

In order to avoid a conflict on the shared resource, the web proxy server declared a global flag, and uses it to control the SCP access. Once the SCP is busy with the request, any request that arrives before the previous request is completely downloaded will be blocked until the SCP resource is free. Only then will the web proxy server be able to forward that particular request. This way the web proxy server has successfully processed requests in parallel without causing any problem to the SCP capabilities.

5.1.2 File server Software Component

The primary function of the File server was to serve the requests coming from the web proxy server by locating the requested resource from the disk storage and to return the resource to the web proxy server. The File server has also operated successfully during the functional test and has met all its requirements as described in section 3.2.

5.1.3 SCP Software Component

The SCP protocol was based on stop-and-wait flow control technique as described section 4.4. This protocol has operated successfully during its functional test. Disconnecting the network cable during transmission proved the error recovery capability of this protocol. When the network cable was disconnected, data transmission stopped.

However, having equipped the transmitter with the timer has enabled the transmitter to keep on trying to re-establish the connection by retransmitting the byte that was not acknowledged as the result of the cable disconnection. When the cable was put back, the data transmission continued successfully until the file was completely transmitted.

Reliable operation is one of the basic requirements of any protocol. The protocol designers implement different mechanisms, which enforces reliability to their protocols. The SCP protocol enforces reliability in two ways. As already indicated the SCP protocol consists of two layers, the byte-level layer and the packet-level layer.

The SCP protocol ensures reliability by acknowledging each byte sent across so as to make sure all bytes reach their destination. The acknowledgment of each byte as the way of forcing them through is done on the byte-level layer. The byte-level layer is also equipped with the timer to take care of the lost acknowledgments as described in section 4.4. However, the byte-level layer is only interested in getting every byte through, but does not check the correctness of the received bytes.

In order to ensure the correctness of the received bytes, the packet-level layer validates each packet before sending an acknowledgment as described in section 4.4.1. This way a better performance was achieved and having the byte-level layer acknowledging each byte eliminated most of wrong packet errors, which result if the checksum computed by the receiving part of the protocol does not match the checksum computed by the transmitting part of the protocol.

Section 4.4.4 presents the results obtained when conducting an experiment for SCP packet size choice. As the results show the packet size of 400 bytes gives a lesser download time of the requested resource. The communication link between the two computers used for operational test was connection-oriented. The connection-oriented communication link unlike connectionless-oriented communication link encounters less interference during data transmission. With this advantage data can then be broken into large blocks in order to reduce the flow control overhead bytes which in turn reduces resource download time.

However, with the SCP protocol, it was not possible to increase the packet size beyond 400 bytes because of the limitation caused by the procedure used to measure the download time and packet processing time (see section 4.4.4). Although the SCP packet size could not be increased beyond 400 bytes, the SCP has satisfied its requirements by successfully transferring a file from the File server to the web proxy server as proposed by the chosen design. In conclusion, the web server software has operated successfully and has met the objective of the project. As the concept demonstrate it is possible to have a satellite-based web server.

5.2 Recommendations

The recommendations for the future enhancement of the satellite-based web server are as follows:

- The implementation of the File server on the satellite in Java will need Java Virtual Machine support to interpret Java bytecode into native machine language. It will therefore be advisable to rewrite the File server in C, which can be compiled to executable code and does not need Java Virtual Machine support. Additionally, C is the most widely supported language in the embedded world, with a large number of high qualities, optimizing C compilers available for most platforms.
- The limitation of the SCP capabilities resulted in having requests that arrive while the SCP is still busy processing the first request to be delayed. In order to provide a better service to the users, the SCP protocol can further be developed into full duplex and replace stop-and-wait flow control technique with sliding-window flow control technique. This way an inefficient line utilization which normal happens when using stop-and-wait flow control technique can be overcome.
- Connectionless-oriented communication links mostly suffer from interference, which causes lot of data retransmissions. It will therefore be advisable in the case of the real satellite to break data into small blocks so that errors can be detected sooner.

Furthermore, retransmitting a small block of data will take less time, as compare to time it will take to retransmit the larger blocks of data.

5.3 Summary

To summarize, the satellite-based web server developed in this thesis has proven during the functional test performed on two personal computers to meet the objective of the project. The HTML files were successfully downloaded. The capability of the web proxy server to process requests in parallel was demonstrated by downloading the images that were embedded on the HTML files.

6 Bibliography

1. Yegear J. & McGrath R. 19996.*Web server Technology*. San Francisco, Carlfonia. Morgan Kaufmann publisher.
2. Chetty C. 1999. *Evaluation of the effectiveness, performance and integrity of sunsat's telemetry system*. M.ENG Thesis. University of Stellenbosch.
3. Berg, D.J. & Fritzinger J.S. 1999. *Advanced techniques for Java Developers*. Revised edition. New York. John Wiley.
4. Richard W.S. 1996.*TCP/IP ILLustrated Volume3*. Addison-Wesley Publishing Company.
5. Bruce E. 1998. *Thinking in Java*. Upper Saddle River, New Jersey. Prentice Hall PTR
6. Naughton P & Schildt H. 2001. Java 2.*The Complete Reference*. 4th Edition. New York. McGraw-Hill.
7. Tanenbaum A.S. 1996.*Computer Networks*. 3rd Edition. Vrije Universiteit Amsterdam, the Netherlands. Prentice-Hall.
8. Coetzee C. 1999.*On Optimizing high-Latency asymmetric satellite TCP/IP*. M.ENG Thesis. University of Stellenbosch.
9. Woeginger G.J. October 2000. *Monge strikes again: Optimal placement of Web proxies in the internet*. Operations Research letters vol.27, no.3, p. 93-96.
10. Charzinski J. October 2000. *HTTP/TCP connection and flow characteristics*. Performance Evaluation vol.42, no.2-3, p.149-62.
11. Heller P. & Roberts S. 1996.*Java 2 Developers' Handbook*. San Francisco. SYBEx.
12. Yu P.S. 2000.*Latency-sensitivite hashing for collaborative Web caching*. Elsevier Science B.V.
13. Feldmann A. June 2000. *BLT: Bi-layer Tracing of HTTP and TCP/IP*. Computer Networks, Vol 33, no1-6, pp.321-335.
14. Fielding R, Irvine U.C, Gettys J, Mogul J, Frystyk H & Berners-Lee T. 1997. RFC2068: *Hypertext Transfer Protocol --HTT/1.1*.

15. Naughton P & Schildt H. 1999. *Java 2.The Complete Reference*. 3rd Edition. New York. McGraw-Hill.

16. Barry Holmes.1998. *Programming with Java*. Boston. Jones & Barlett Publishers.

17. Steven Holzner. 19998. *Java 1.2 in Record Time*. San Francisco. SYBEx.

18. Ed Taylor, 1998. *TCP/IP Complete*. New York. McGrath-Hill.

19. Mark A. & Miller, P.E.1999. *Troubleshooting TCP/IP*. 3 rd Edition. Foster City, CA. M∈tT Books.

20. Judith Bishop & Nigel Bishop. 2000. *Java Gently for Engineers and Scientists*. Harlow. Addison-Wesley.

21. William Stallings. 2000. *Data & Computer Communications*.6th Edition. Upper Saddle River, New Jersey. Prentice Hall.

7 APPENDIX A

Table A.1 gives the brief description of Java programs developed for the satellite-based web server project.

Web proxy server classes	
Class name	**Description**
Web_proxy_server.java	This is the main class that invokes the main method to start executing the program. Inside main() method, the web_proxy_server invokes the ConnectionAcceptor thread.
ConnectionAcceptor.java	This class listens for the incoming connection, accepts the connection and creates a RequestHandler thread to handle that particular connection.
RequestHandler.java	Handles the input and output responsibilities for the requests
File server classes	
File_server.java	Invokes the main method to start running the File server
SettingDefault.java	This class entails the method that returns the index document of the web page.
Serial Communication Protocol classes	
Transmitter.java	This class is responsible for receiving the name of the requested resource and for breaking the data into data packet required for the creation of the packet and thereafter sends them over to the receiver class.
Receiver.java	This class issues a call to the method that sends the name of the requested resource to the File server and thereafter receives data packets coming from the File server and passes them to the web_proxy_server class.
MakePacket.java	Creates the SCP packet and passes it to the Transmitter class

	for transmission.
SerialPort.java	Used to initialize the serial port for either reading or writing of data.
Timer.java	Times out to allow either the receiver or transmitter to repeat its action
IO.java	Used to provide an interface through which the transmitter and receiver access input and output devices.

8 APPENDIX B

Source code available on request