

Design of a Distributed Satellite Ground System

Richard Malan Barry



THESIS PRESENTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF MASTER OF
ENGINEERING AT THE UNIVERSITY OF STELLENBOSCH

Supervisor: Prof. P.J. Bakkes
December 2001

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 entirety or in part submitted it at any university for a degree.

Abstract

This thesis describes the development of a distributed ground support system to be used in the small satellite program at the University of Stellenbosch.

A literature study was done to determine the current trends in ground support design. These trends are presented together with an analysis of the SUNSAT groundstation software. New technologies in the field of distributed systems are discussed and used in the design of a distributed ground support system. The design centres around a services-based architecture where services are distributed on the network. The design is evaluated according to attributes exhibited by effective distributed systems. Certain services were implemented to demonstrate the feasibility of the design. The implementations are discussed and suggestions are made for future improvements and fields of possible further study.

Opsomming

Hierdie tesis beskryf die ontwikkeling van 'n verspreide-grondstasie-sisteem vir gebruik in die Universiteit van Stellenbosch se satelliet-program.

Eerstens is 'n literatuurstudie gedoen om die huidige verwickelinge in die veld van satellietondersteuning-ontwikkeling vas te stel. Hierdie verwickelinge word bespreek en gebruik om die SUNSAT-grondstasie sagteware te evalueer. Nuwe tegnologieë in die veld van verspreide stelsels word ondersoek en bespreek. Die ontwerp van die grondstasie is gebaseer op 'n verspreide dienste argitektuur. Die ontwerp word beskryf en geëvalueer aan die hand van kenmerke tipies van 'n effektiewe verspreide stelsel. Om die werkbaarheid van die ontwerp te toon, is sekere van die dienste geïmplementeer, en die funksionering daarvan word bespreek. Voorstelle word ook gemaak oor moontlike toekomstige navorsing wat gedoen kan word.

Contents

Acknowledgements	vi
List of Abbreviations and Acronyms	vii
List of Figures	ix
List of Tables	x
1 Introduction	1
1.1 Problem Statement	1
1.2 Previous Work	2
1.3 Thesis Overview	3
2 Developments in Ground Support Systems Design	4
2.1 Trends in Ground Support Design	4
2.1.1 Computing Hardware Technology	4
2.1.2 Distributed Object Computing Technology	6
2.1.3 Internet	6
2.1.4 Automation Technology	7
2.1.5 Software Development Technology	8
2.1.6 Conclusions	8
2.2 Existing Ground Systems	9
2.2.1 Jsat/Jswitch	9
2.2.2 SCOS II	10
2.2.3 TEAMSAT	12
2.3 SUNSAT Groundstation Evaluation	13
2.3.1 Physical Topology	13
2.3.2 Software Architecture	15

2.3.2.1	Groundstation Command and Control Client	16
2.3.2.2	Telemetry Decoding Program	17
2.3.2.3	SUNSAT Database	17
2.3.2.4	Telemetry Extraction and Visualisation Client	17
2.3.3	Conclusions	17
2.3.3.1	Command and Control Client	18
2.3.3.2	Telemetry Decoding Program	18
2.3.3.3	SUNSAT Database	18
2.3.3.4	Telemetry Extraction and Visualisation Client	19
2.4	Conclusions	19
3	Design Choices	20
3.1	Unified Information Model	20
3.1.1	Future Development Language	21
3.1.2	Data Description Language	21
3.1.3	Ground System Architecture	21
3.2	Extensible Markup Language (XML)	22
3.2.1	XML-RPC and SOAP	24
3.2.1.1	XML-RPC vs Other Middleware Architectures	25
3.2.2	JDOM	26
3.3	Services-based Architecture	26
3.3.1	Network Service	27
3.3.2	Data Storage Service	27
3.3.3	Data Access Service	28
3.3.4	Orbit Data Service	28
3.3.5	Time Service	29
3.3.6	Security Service	30
3.3.7	Telemetry Processing Service	30
3.3.8	Groundstation Control Service	30
3.3.9	Mission Control Service	31
3.3.10	Mission Planning Service	31
3.4	Conclusions	31
4	High Level UML Design	33
4.1	Services-based Architecture Design	33

4.1.1	Network Service	33
4.1.2	Telemetry Processing Service	35
4.1.3	Data Storage Service	36
4.1.4	Data Access Service	37
4.1.5	Groundstation Control Service	38
4.1.6	Orbit Data Service	39
4.1.7	Mission Control Service	41
4.1.8	Mission Planning Service	42
4.1.9	Security Service	44
4.1.10	Time Service	45
4.2	Proposed Clients	46
4.2.1	Hardware Control and Status Display	46
4.2.2	SatFTP Client	46
4.2.3	Telecommand Client	47
4.2.4	Orbit Prediction Client	47
4.2.5	Telemetry Display Client	47
4.2.6	Network Control Client	47
4.3	Design Evaluation	48
4.3.1	Security	49
4.3.2	Synchronisation	49
4.3.3	Compatibility	49
4.3.4	Availability	49
4.3.5	Performance	50
4.3.6	Complexity	50
5	Implementations	51
5.1	Network Service	51
5.1.1	Use Case Implementations	52
5.1.2	Error Handling	55
5.1.3	Network Service Configuration Clients	55
5.1.3.1	Local Client	56
5.1.3.2	Remote Client	58
5.2	Orbit Data Service	60
5.2.1	Use Case Implementations	60
5.2.1.1	PREDICT Source Code Modifications	61

5.2.1.2	XML-RPC Handler	62
5.2.2	Tracking Client	63
5.2.2.1	ImagePanel Component	65
5.2.2.2	XML-RPC Client	65
5.3	Conclusions	66
6	Conclusions and Future Work	67
6.1	Thesis Results	67
6.2	Future Work	68
6.3	Conclusion	69
A	XML-RPC Specification	70
A.1	Overview	70
A.2	Request example	70
A.3	Header requirements	71
A.4	Payload format	72
A.5	Scalar <value>s	72
A.6	<struct>s	73
A.7	<array>s	73
A.8	Response example	74
A.9	Response format	75
A.10	Fault example	75
A.11	Strategies/Goals	76
A.12	FAQ	77
A.13	Additions	79
B	PREDICT's Socket Commands	80
B.1	System Configuration	80
B.2	Program Operations	81
B.3	Client Program Interface	81
B.4	PREDICT Socket Command Summary	81
B.4.1	Command: GET_WITH	82
B.4.2	Command: GET_SAT	83
B.4.3	Command: GET_DOPPLER	85
B.4.4	Command: GET_SUN	85

B.4.5	Command: GET_MOON	86
B.4.6	Command: GET_LIST	86
B.4.7	Command: RELOAD_TLE	87
B.4.8	Command: GET_VERSION	88
B.4.9	Command: GET_QTH	88
B.4.10	Command: GET_TLE	89
B.4.11	Command: GET_TIME	89
B.4.12	Command: GET_TIME\$	90

Bibliography	91
---------------------	-----------

Acknowledgements

I would like to thank the following people for their help and support during this project:

My study leader, Prof. Pieter Bakkes, for his help and support during the project and for helping me to focus on the job at hand.

Ben van der Merwe for his thesis on which this project greatly built, and for his suggestions and help throughout the project.

My parents and friends for their support and for trying to understand what exactly I was talking about when I tried to tell them what I was doing.

Lisa for her support, interest and encouragement during the whole time I worked on this project.

Lindie Cloete and Nikki Steenkamp for their hard work in proof-reading this thesis.

List of Abbreviations and Acronyms

4GLs	Fourth generation languages
AOS	Acquisition of Signal
API	Application Program Interface
AX.25	Amateur X.25
C&C	Command and Control
CCSDS	Consultative Committee for Space Data Systems
CORBA	Common Object Request Broker Architecture
COTS	Commercial-off-the-shelf
DBMS	Database Management System
DCOM	Distributed Component Object Model
DDL	Data Description Languages
DOM	Document Object Model
DTD	Document Type Definition
ESA	European Space Agency
GUI	Graphical User Interface
HTTP	Hypertext Transport Protocol
IDL	Interface Definition Language
IP	Internet Protocol
JSSE	Java Secure Socket Extension
LAN	Local Area Network
LOS	Loss of Signal
NASA	National Aeronautics and Space Administration
NTP	Network Time Protocol
OSCAR	Orbital Satellite Carrying Amateur Radio
PACSAT	Packet Satellite
PBBS	Packet Bulletin Board System
PC	Personal Computer
RMI	Remote Method Invocation
SAX	Simple API for XML
SatFTP	SUNSAT File Transfer Protocol
SCC	Serial Communication Controller
SGML	Standard Generalised Markup Language
SOAP	Simple Object Access Protocol

SQL	Structured Query Language
SSL	Secure Socket Layer
SSTP	SUNSAT Simple Time Protocol
SUNSAT	Stellenbosch University Satellite
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UIM	Unified Information Model
UML	Unified Modeling Language
UTC	Co-ordinated Universal Time
WOD	Whole Orbit Data
XML	Extensible Markup Language
XML-RPC	Extensible Markup Language Remote Procedure Call

List of Figures

2.1	Groundstation physical topology [1]	14
2.2	Ground system data flow [1]	15
3.1	Services-based architecture	27
4.1	Network Service	34
4.2	Telemetry Processing Service	36
4.3	Data Storage Service	37
4.4	Data Access Service	38
4.5	Groundstation Control Service	38
4.6	Orbit Data Service	40
4.7	Mission Control Service	41
4.8	Mission Planning Service	43
4.9	Security Service	44
4.10	Time Service	45
5.1	Network Service Implementation	52
5.2	Network Service Class Diagram	53
5.3	XML-RPC Server Startup	55
5.4	Local Configuration Client	56
5.5	Local Configuration Client Class Diagram	57
5.6	Remote Configuration Client	58
5.7	Remote Configuration Client Class Diagram	59
5.8	Orbit Data Service Implementation	60
5.9	PREDICT Server Mode	61
5.10	Orbit Data Service Handler Class Diagram	62
5.11	Tracking Client	63
5.12	Tracking Client Class Diagram	64

List of Tables

2.1	Technologies changing the design of satellite ground support	5
A.1	XML-RPC Supported Data Types	72

Chapter 1

Introduction

1.1 Problem Statement

To operate and manage a satellite mission successfully, an effective ground support system needs to be in place. The ground support system firstly has to retrieve, process, store and present both operational and telemetry data from the satellite it supports. Secondly, it has to provide reliable control over the satellite and other hardware systems. The problem of control and data handling becomes even more complex when multiple groundstations are used, requiring communication and co-ordination between the different parts that make up the system. Operating the system effectively requires an interface from where the entire system can be controlled and managed.

The ground support system used to support SUNSAT at the University of Stellenbosch provided effective management of the mission, but areas of improvement were identified during the phase of routine operation. Due to time and staff constraints, the software of the ground support system was implemented by re-using as much as possible of the flight software. This led to an implementation that was very effective at supporting SUNSAT, but with limited possibility for expansion. The groundstation software required a high level of human interaction, and introducing automation would have involved big changes and software re-development. Expanding the system to multiple groundstations

also would have required new software to be developed, as the current groundstation software could not be distributed.

The main needs identified during the operation of the SUNSAT mission were:

- A software architecture that allows for future expansion and use across different missions.
- Automation of the ground support system to provide autonomous operation.

The software architecture of the ground support system lays the foundation for future development and operation. This thesis covers the design and implementation of a software architecture that meets the needs identified above. The proposed design aims first of all to provide the same functionality as the system used for SUNSAT, but with the main aim of being simple to expand and evolve. The software implementations of modules in the system aim to prove the feasibility of the design and principles behind the design.

1.2 Previous Work

The work done in this thesis was mainly a continuation of the work done in another thesis by Ben van der Merwe [1]. That thesis covered the design and implementation of the current SUNSAT ground support system. At the end of that thesis several recommendations are made on how to improve the system, and areas of further study are identified. This thesis covers the study of some of the identified areas, namely:

- The choice of a suitable ground system distributed protocol.
- The investigation of various technologies based around the common data description language XML.
- The specification of a design for a next generation ground support system.

1.3 Thesis Overview

The thesis follows the path of the work done in the design of the new ground support system architecture. Previous work is analysed and a design is proposed to improve on the existing system. A review is then given of the implementations that were made to test the design principles.

Chapter 2 looks at previous work done in the field and identifies trends in the design and implementation of such systems. A brief review of the SUNSAT ground support system is then given, and the software is then analysed according to the trends previously identified. Recommendations are made on relevant changes to bring the system in line with current trends.

Chapter 3 first looks at the key technologies and tools used in the design and implementation of the system. The second part of the chapter proposes a services-based architecture for the system and gives a description of the proposed services.

Chapter 4 is a detailed design description of the proposed services-based architecture. The functionality of each service is discussed by the use of a UML use case diagram. The design is evaluated according to a set of principles that identify effective distributed systems.

Chapter 5 gives a review of the implementation of the services and clients. The functionality of the implementations are discussed and important implementation decisions are motivated.

Chapter 6 looks at the results obtained in the thesis and makes recommendations on future work to be done on the system and possible areas of further research.

Chapter 2

Developments in Ground Support Systems Design

2.1 Trends in Ground Support Design

The relatively low cost required to manufacture and operate small satellites have brought satellite technology to customers who could not afford it previously. Small satellites are simpler and therefore faster and cheaper to build, and also require less complex ground support systems. Up to 60% of the total cost of a satellite mission is spent on the operations phase[2]. Therefore it makes sense to create a ground support system that can easily be re-used on future small satellite missions. Emerging technologies have changed the design of satellite ground support systems [3]. The five technologies listed by [3] and how they are changing the rules in satellite ground support are discussed.

2.1.1 Computing Hardware Technology

Computer technology improves so quickly that the technology used in the first stages of a mission design can be obsolete when the satellite is finally launched. The power of today's desktop systems also makes it possible to run computations on a desktop

<i>Technology</i>	<i>Type of Change</i>
Computing Hardware	Computing power continues to get cheaper and more portable
Distributed Object Computing	Software can be distributed as needed
Internet	The internet is a cheap, established network that can be used in a ground station network
Automation	Systems are designed for autonomy and less human intervention
Software development	Development power is shifting towards re-use and integration of COTS products

Table 2.1: Technologies changing the design of satellite ground support

previously reserved only for mainframe and workstation computers.

Old rule: Ground system computing hardware is big and expensive.

In the past, the performance required for satellite ground support could only be delivered by expensive computer hardware and dedicated communication networks. Therefore it was only in reach of large organisations with big budgets to spend on ground support systems.

New rule: Ground system computing hardware is portable and cheap.

Ground support still requires high performance levels from the hardware used, but the changes in technology now make the necessary performance available in desktop PC's and cheap commercial networks like the internet. This has also made the groundstation equipment more portable, and changed the distribution of functionality from a centralised control on a mainframe to distributing the functionality over different systems geographically independent of each other.

2.1.2 Distributed Object Computing Technology

Distributed Object Computing makes it possible for applications to execute wherever sufficient resources are available and makes the development and execution of the software platform independent through a three-tier architecture. It also promotes re-use of software components through object-oriented languages like Java and C++.

Old rule: Ground system elements must be centralised in facilities.

In the past distributed systems were expensive to build and manage. Major ground system elements were centralised in one facility. This placed specific needs on the design of the software in terms of the platform and language used.

New rule: Ground system elements can be centralised or decentralised as desired.

Ground systems can be built from objects distributed across different platforms. Distributed objects allow re-use and execution where sufficient resources are available. Middleware architectures simplify the design and distribution of the objects, making the distributed system seem like a virtual computer to the objects. Designing a distributed system is different from designing a centralised system. Interfaces between the objects and responsibilities of the objects need to be defined thoroughly in the early stages of development.

2.1.3 Internet

Satellite control systems require a reliable communications network to support data collection and control from multiple groundstations. In the past, this necessitated the use of an expensive, dedicated network. The popularity of the internet has made it a globally accessible and cheap alternative to the solutions of the past.

Old rule: Dedicated, closed communications networks are required for ground systems.

Sharing data over a wide geographical region was expensive and required a specialised communication infrastructure.

New rule: Commercial, open communication networks can support ground systems.

The internet makes it possible not only to share data, but also to increase functionality across wide areas. Facilities no longer need to be contained in one geographical region. The use of a public network brings additional complexity in designing security and access, but the advantages in using the network far exceeds the disadvantage of the extra cost to development.

2.1.4 Automation Technology

Trends in ground system design are towards automating routine tasks and putting the human control back in the domain of making decisions about the mission. The definition of routine might change from mission to mission, therefore the design should always allow for human interaction if necessary.

Old rule: Ground systems should make human interaction easy.

In the past, ground system control interfaces were designed to make it easy for the controller to interact with the system, for instance using graphical interfaces.

New rule: Ground systems should make human interaction unnecessary.

The system should still be designed to provide the necessary output to the end user (humans), but with software replacing the routine tasks previously carried out by operators. The techniques of expert systems, artificial intelligence and case-based reasoning can be used to enable the system to make informed decisions. This makes it possible for users not trained in the detailed working of the groundstation to make requests to the system and have it delivered in a format that makes sense to the user.

2.1.5 Software Development Technology

Software development has seen two big trends in the recent past. The first is the use of a disciplined process to effectively develop software that is more maintainable. The second is the use of rapid application development (RAD) tools. This enables the software creator to develop a new piece of software by adding together and integrating existing software components. Specialised commercial-off-the-shelf (COTS) components can be bought and integrated into a new system without the need to develop that part of the system.

Old rule: Software developers build and maintain custom ground support systems.

In the past, ground system software was developed to match each mission's needs exactly. An increase in productivity could only be achieved with each company keeping its own software library and re-using and maintaining its own code.

New rule: End user programmers integrate ground systems using commercially available components and tools.

Traditional development tools like compilers, linkers and debuggers are giving way to visual programming tools and fourth generation languages (4GLs). This enables end-user programmers with less training in programming, but more in the system being developed, to build software from available components. It is estimated that in 2005 the United States will have 55 million end-user programmers and only about 2.75 million traditional software developers [4]. In the design of a ground system element the developer could be the operator or a specialist on a part of the mission.

2.1.6 Conclusions

Technical advances are placing more power in the hands of the end user. They allow faster and cheaper development of ground systems. They increase re-usability and introduce a level of software autonomy that eliminates repetitive operations.

The design of these systems emphasises engineering discipline, because more higher level design decisions have to be made out of the wider range of options. The satellite and ground system must be seen as a single system providing the best solution for the mission. The new development tools do not eliminate the need for good design practice, but rather enhance the design process.

Designing for automation brings fundamental changes in the design and operations of a ground system. Design strategies aimed at developing easy-to-use graphical interfaces need to be changed to developing autonomous software capable of decision making. Automation is not only a way of cutting costs, but also a way of freeing human abilities from repetitive tasks and applying them towards more challenging problems.

2.2 Existing Ground Systems

2.2.1 Jsat/Jswitch

Goddard Space Flight Center has developed a Java-based user interface to a spacecraft command and control system called Jswitch [5]. As part of the Jswitch system Jsat was developed, a Java-based information-on-demand science data trend analysis tool. The system reduces the cost of operations control by providing access to spacecraft telemetry and control via a web interface. Jsat eliminates the need for the analysis of specialised scientific payload data before presentation to the user. It removes the middle layer by giving the user direct access to the data and distributing the processing of the data. The system uses COTS products where possible to reduce development cost and time. The following design lessons can be learned from the Jswitch project:

- **Design for re-usability.** Using Java as development platform makes the software platform independent. The object-oriented approach also makes integration of existing software easier.
- **Separation of interface from back-end.** By separating the user interface from the rest of the system, user interfaces can be standardised across different missions.

The information serving system can therefore change without affecting the design of the user interface.

- **Use of existing and COTS software.** Re-use of existing software makes the design process cheaper and faster.
- **Use of the internet for distribution of the ground system.** The use of the internet to deliver information to the end-user makes it possible to display the information to the user in a browser. Existing software can easily be integrated into the ground system software.

The Jswitch system demonstrated successfully that it is possible to receive telemetry, send commands, process events and analyse statistical data over the web.

2.2.2 SCOS II

SCOS II is a spacecraft control system developed by the European Space Agency (ESA)[6]. The system was developed to address needs identified in previous systems:

- Easy mission customisation and extension.
- Vendor independence.
- Improved performance.
- Improved functionality.

These needs were addressed using the following techniques:

- Object-oriented technology to enable re-use.
- Using widely adopted standards (like TCP/IP, Unix) to improve vendor independence.
- C++ as implementation language for efficiency.

- Client-server software architecture.

The SCOS II project found that in implementing a distributed system the choice and implementation of middleware is a very important aspect influencing performance, but that the physical distribution of the system is equally important. Certain critical services, like the commanding kernel, need to be centralised to ensure good enough performance. In deciding which components to centralise, it is important to decide which components are going to gain little from a distributed architecture and need the performance guarantees of a centralised system. One of the advantages of designing for distribution is that it is always easier to centralise a distributed system, usually through simple reconfiguration, than to distribute a system initially designed for centralisation.

One of the biggest design issues faced in the SCOS II project was the possibility of high bandwidth requirements for data distribution in a distributed system. Take as an example a simple telemetry display client residing on a remote workstation. If every client has to retrieve the data from the mission database, the resource requirements on the network and at the telemetry server will increase as the number of clients connected to the network increase. However, if the telemetry server broadcasts the data and the clients cache the data locally, the number of clients would not influence the load on the resources as much. It was therefore decided to make use of a system where data is broadcasted on the network and central servers archive all data. Clients cache the data for real-time display or future display, and can also make requests to the servers. The servers answer to these requests in bursts of data. The size and times between bursts are configurable at the server to prevent multiple requests from influencing the server's response negatively.

Even though certain services in the system were centralised, the servers were still implemented only as providing services to the network. As an example, the command kernel was implemented as a centralised server, but access to the server was still via command sources like the manual command client and the scheduling client. These clients are located on client workstations on the network, and therefore the whole command system can still be seen as distributed.

The following design lessons can be learned from the SCOS II project:

- Design for distribution as much as possible, but centralise where necessary.
- Design the system with data storage as the core of the system and make the retrieval possible by distributed means.
- Broadcast data to clients and cache locally to improve performance.
- The choice of middleware and distribution of the parts of the system are equally important aspects influencing performance.

2.2.3 TEAMSAT

The TEAMSAT project was a ground support system designed to support both the TEAM and YES satellites. Both satellites were built by young engineers and students, and a low cost support system was needed. The development of the system is discussed in [7].

The main goal of the system was to design the pre-launch testing software and systems so it could be re-used in the mission control after launch.

The system used COTS products as much as possible for telemetry processing and telecommand. The fact that ESA packet telemetry and telecommand standards were used, made a wide range of COTS products also using these standards available.

After the development of the testing system, it was realised that re-using it as the mission control system was an over-simplification of the mission control system's needs. A mission control system based on the ESA SCOS II ground support system was developed. The flight control required support for multiple groundstations in the network, spacecraft control, flight dynamics support (in particular orbit prediction) and mission planning support. The developed mission control system provided the following capabilities to the staff:

- Spacecraft database setup.
- Telemetry processing and archiving.

- Raw data distribution, archiving and retrieval.
- Time correlation.
- Telemetry display
- Commanding
- Mission event logging

The TEAMSAT ground support system was developed and deployed in a very short time (around 8 months in total [7]). It is a good example of the use of existing systems and software, where possible, to save cost and time.

2.3 SUNSAT Groundstation Evaluation

A detailed description of the SUNSAT groundstation software is given in [1]. The description will be reviewed briefly and recommendations for improvements for future systems will be made.

2.3.1 Physical Topology

The groundstation consists of the following components (figure 2.1):

Tracking station. The tracking station interfaces with the antennae controller hardware to track the satellite as it passes overhead. Orbit prediction software is used to calculate the pass times.

Multiplexer, radios and modems. Amateur radios are used to communicate with the satellite. A multiplexer is used to route the analogue data between different combinations of modems and radios.

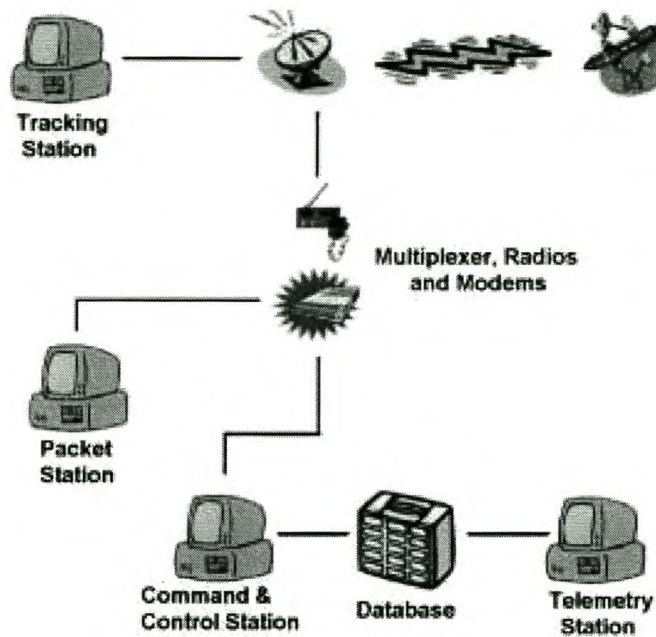


Figure 2.1: Groundstation physical topology [1]

Command and control station. This station commands the satellite and centralises the control of the other components of the groundstation. The control of the station requires a high degree of human intervention. The software was developed mostly in Modula-2 and runs on the MS-DOS platform.

Database. Here the data acquired from the communications with the satellite is stored. A wide range of data, mostly telemetry data, but also status and error information as well as scientific data, is stored in the relational database.

Telemetry station. This station allows for the extraction and processing of the data from the repository as well as for exporting the data to third-party software.

Packet station. Here an interface to the ground segment of the PACSAT protocol suite is provided. Facilities provided by PACSAT satellites, such as the Packet Bulletin Board System (PBBS) and file uploading, can be accessed from here.

2.3.2 Software Architecture

This section will describe the architecture of the ground system software and make suggestions about future improvements on the system.

The software consists of the following software modules:

- Groundstation Command and Control Client.
- SatFTP Client.
- Telemetry Decoding Program.
- SUNSAT Database.
- Telemetry Extraction and Visualisation Client.

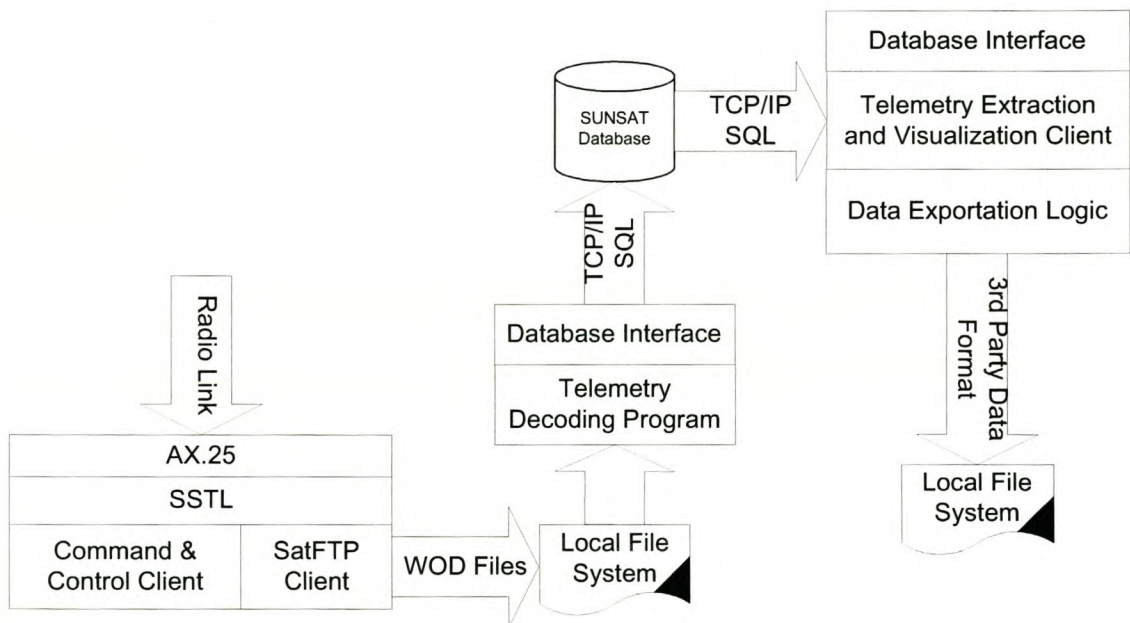


Figure 2.2: Ground system data flow [1]

The data follows the following path through the software from the satellite to the end-user (see figure 2.2):

The data arrives via the AX.25 radio link at the command and control station where it is handled by the layered protocol stack and stored on the local file system in the form of Whole Orbit Data (WOD) files by the SatFTP Client. The WOD files are then fed to the telemetry decoding program and stored in the SUNSAT database via TCP/IP network connection to the SQL database. The data can then be extracted using the telemetry extraction and visualisation client and stored in a form readable to third-party analysis tools.

2.3.2.1 Groundstation Command and Control Client

The command and control (C&C) client is a user interface to a number of the groundstation's command and configuration facilities:

- **AX.25 Status Display and Command Console.** The status and activity of the link is displayed for easier analysis.
- **Radio Command Console and Status.** The configuration of the various radios can be displayed and changed from here.
- **SatFTP Client Console.** It provides an interface for communication with the satellite-based SatFTP server.
- **Multiplexer Configuration.** This allows for the configuration of the routing of signals between individual radios and modems.
- **Log Server Client.** This allows the display of log messages from the satellite.
- **SSTP Console.** Commanding of the SUNSAT Simple Time Protocol (SSTP) is done from here.
- **Client Configuration.** The Serial Communication Controller (SCC) cards can be configured from here.
- **Telecommand and Command and Control Console.** Telecommand and predefined command and control instructions can be sent to the satellite via the AX.25 link.

The C&C client was implemented in Modula-2. A limited text based interface was developed and the set of functions implemented as one piece of software. This made automating or distributing the system with the current software very difficult.

2.3.2.2 Telemetry Decoding Program

The telemetry decoding program extracts frames from the WOD file that was used as input and then extracts individual telemetry elements from the frames. It can then connect to the database through a network interface and store the elements in the appropriate tables.

2.3.2.3 SUNSAT Database

At the center of the ground system is the database. The processed telemetry data received from the satellite is stored in the relational database. Access to the database is controlled by the Database Management System (DBMS). The DBMS provides a standard interface via the Structured Query Language (SQL) over the network. Queries and submissions are therefore independent of the DBMS and underlying database, as well as the location of the client accessing the data over the network.

2.3.2.4 Telemetry Extraction and Visualisation Client

The client provides a graphical user interface (GUI) to extract telemetry data from the database, calibrate it using meta-data and display it graphically. The data can then be exported to a form usable by third-party software like Matlab.

2.3.3 Conclusions

As seen in section 2.1.6 the trends in ground support system design aim towards a higher degree of automation, a distributed architecture and future re-use. The implementation

of the SUNSAT ground system can be evaluated according to how suitable its components are towards these trends. Recommendations on changes to the architecture of the system will then be made.

2.3.3.1 Command and Control Client

The fact that the client was developed in Modula-2 severely limits the possibility of automation as well as the distribution of the implemented facilities as separate services, mainly because of a lack of network support and multitasking in the operating system. A solution would be to re-develop the needed functions as individual services distributed on the network. This would make re-use and automation easier to implement. The use of a more modern programming language, like Java, would make this re-development less time consuming.

2.3.3.2 Telemetry Decoding Program

The telemetry decoding connects to the database through a network connection and can be included easily in an automated system. It could be changed to accept input from a client over the network and therefore be deployed as a service on the network.

2.3.3.3 SUNSAT Database

The DBMS allows connections to the database to be made over the network via SQL commands. The DBMS also provides security for access to the database. The database is therefore already available as a service on the network and can be incorporated in a distributed and automated system.

2.3.3.4 Telemetry Extraction and Visualisation Client

The client connects to the database over the network. This is an excellent example of the client type front-end required for a distributed system. Clients make use of the services available on the network (e.g. the database) to fulfil a need of the user (e.g. telemetry display).

2.4 Conclusions

From the reviews of current ground support systems in sections 2.1 and 2.2, certain trends can be identified. These trends in design and implementation have as goal to produce ground support systems that are more efficient and reliable at supporting their missions, while at the same time providing a more productive and automated interface for the operators. Important trends identified in this chapter are:

- Design for re-use and use of COTS products to save costs and time.
- Distributed system architecture for reliability, expansibility and implementation flexibility.
- Automation for increased productivity and functionality.
- Separation of the user interface and back-end to give more flexibility in implementation and future changes.

These principles will form the basis of the design of a new ground support system in the following chapters. In the next chapter the technologies needed to design and implement a system based on these principles will be reviewed. A distributed architecture will be proposed for the system and designed and implemented in the following chapters.

Chapter 3

Design Choices

3.1 Unified Information Model

The idea of a groundstation design centered around a unified information model is developed in [1]. By unified information model for a satellite is meant the interoperability of all software and data, in other words, portable software and portable data. It has as basic goals the efficient handling of data generated by mission results, as well as the safe and efficient control of the mission. A systems analysis was done in [1], based on the requirements learnt from experience in the SUNSAT mission. An analysis of other similar missions was also done. The recommendations made will be reviewed briefly and used as the basis for further study in a distributed ground system. The following areas were studied:

Future Development Language

Data Description Language

Ground System Architecture

3.1.1 Future Development Language

Java is proposed as the language of choice for future ground system development. It has broad support for many of the new technologies proposed later and enables rapid prototyping in the study of future technologies to be used in ground system design. Java also provides portability to software due to its platform independence. Therefore, it is a good choice for developing distributed applications.

3.1.2 Data Description Language

The Extensible Markup Language (XML) is proposed as a data description language to be used in the development of the future ground system. The broad support for XML and the advantages of spin-off technologies like XML-RPC, a remote procedure call protocol based on XML, are cited as some of the advantages of using XML.

3.1.3 Ground System Architecture

The development of a distributed architecture is proposed in [1]. Using a distributed architecture has the following benefits:

- The creation of a standard link between network systems, enabling the expansion of services and the addition of future systems.
- Increased flexibility and redundancy which leads to greater reliability and scalability.
- The physical distribution of services to where resources are best utilised.
- Automation can be introduced in increasing levels by adding services to facilitate automation.

It is clear that the development of a distributed ground support system will lay the groundwork for a system that can be expanded and changed as needed in future. Using

XML as the data description language will enable components that were developed independantly to work together in the distributed system. Java as language enables rapid development of components. From the study of existing ground systems in section 2.2 it is clear that big savings in development resources can be made by re-using existing software and using COTS products where possible.

3.2 Extensible Markup Language (XML)

XML [8] is a meta-language used to define other languages, like its predecessor Standard Generalised Markup Language (SGML). Unlike HTML or most other markup languages, XML does not specify the tag set or the grammar for a language. The tag set is the set of tags that have a meaning in the specific language, while the grammar defines the correct use of the language's tags. XML allows the definition of the content of the data in a variety of ways, as long as the definition conforms to the general required structure.

A XML document must be well-formed to be parsed correctly. A well-formed document is one that has a matching closing tag to every opening tag, and does not have any tags nested out of order.

XML documents can also be valid, but it is not required. A valid document is one that conforms to its document type definition (DTD). The DTD defines the grammar and tag set for a specific XML formatting. Another way for a XML document to be valid is to conform to a schema, a new way of defining format replacing DTD's. A document conforming to a schema is said to be schema-valid.

A simple example of an XML document is:

```
<?xml version="1.0" ?>
<!-- Example XML Document -->
<Roottag>
    <Childtag1> Contents of Childtag1 </Childtag1>
    <Childtag2> Contents of Childtag2 </Childtag2>
</Roottag>
```

As can be seen, the document structure is very simple and almost any kind of data can be represented in this way by simply defining the tag set to suit the data and nesting tags. All tags are closed (matched by a tag with the same name preceded with a `"/`) and nested in order. Tags of the `<? ?>` format are special command tags, and tags of the `<!-- -->` format are comments.

Some of the advantages of XML are that it is legible for humans and reasonably simple, that it is easy to use and to create and that it has support for a wide area of applications. Using XML to describe data gives the data a high degree of portability, separating the data from the applications using the data. Separating the data itself from the presentation of the data makes it possible to present the same data in different ways on different platforms. It therefore brings the type of portability to data that Java brings to software.

Other technologies and tools that developed out of XML and that are relevant to this study are:

- XML-RPC and SOAP
- JDOM

3.2.1 XML-RPC and SOAP

XML-RPC is a protocol based on XML to make remote procedure calls. Because the remote procedure calls are formatted in XML, the protocol is language-, platform- and application independent.

The Simple Lightweight Object Protocol (SOAP) can be seen as the successor of XML-RPC. It also uses XML and HTTP (the underlying protocol of most internet transport) to invoke methods on servers, services components and objects. Although the SOAP specification is not bound to HTTP as the only transport protocol, most current implementations use HTTP. Using HTTP makes using the existing internet infrastructure to transport the remote method invocations possible. The binary transport protocols of CORBA and DCOM have a big disadvantage over SOAP in this area, because they need specific ports to be opened on firewalls and therefore can cause security risks [9]. This causes a problem in the eventual deployment of the distributed application.

Although SOAP is a superset of XML-RPC, they are not compatible [10]. SOAP has support for a much wider range of data types, where XML-RPC has support only for the basic data types. While XML-RPC is already widely deployed, SOAP is still a very new specification and not as widely used. It is expected that the deployment of SOAP services will increase dramatically in the future, as many big names in the software industry have announced their support for it. The fact that Microsoft's new .NET framework of distributed web services uses SOAP as a rpc protocol [11] means that SOAP could be widely deployed in the future and will then make a good choice as a rpc protocol. At the moment XML-RPC is the better choice for development because of the availability of implementations in many languages. Another factor currently counting against SOAP becoming widely deployed is the larger size of current implementations (in contrast to XML-RPC with small and efficient implementations), as well as the fact that not all the implementations are 100% compatible. XML-RPC was therefore used for development in this project, because of its availability. As more stable SOAP implementations become available, the XML-RPC layer in the protocol stack can be replaced by a SOAP layer with total transparency to the layers above and below.

3.2.1.1 XML-RPC vs Other Middleware Architectures

In comparing XML-RPC with middleware technologies like CORBA and DCOM, it has to be noted that these technologies provide a wider set of services than only a rpc protocol. In fact, they both use their own specific protocols for communication between components. They also provide services to manage components on the network bus, like a central repository where the location of interfaces are stored.

The disadvantage of a middleware architectures like CORBA and DCOM, is that it is not trivial to use and deploy. Interfaces for components have to be defined in CORBA's Interface Definition Language (IDL), thus necessitating the programmer to learn another language. The full functionality and complexity of these middleware architectures are not needed in the SUNSAT project and future similar projects. It would therefore be an overly complex solution to the problem to use CORBA or something similar. XML-RPC presents a simpler solution without a big learning curve like CORBA. Only the needed services can be implemented, and once implemented re-used. Services can be developed and added at any stage to expand the functionality of the distributed system.

Another advantage XML-RPC has over more complex architectures, is that most implementations of XML-RPC use HTTP as transport protocol. This gives it a big advantage when deploying the distributed system beyond the boundaries of firewalls, because the ports already opened on the firewall for HTTP traffic are used for the transport [12]. Encryption is simple with Secure Socket Layer (SSL) encryption available for HTTP. A wealth of other tools for both the client and server side of the HTTP protocol are available and can be used with XML-RPC.

As mentioned previously, XML-RPC is language- and platform independent. This gives it a great advantage over other simpler protocols like Java's Remote Method Invocation (RMI). RMI can only be used for communication between Java objects.

3.2.2 JDOM

JDOM is a Java API developed from a specification written by Brett McLaughlin and Jason Hunter [13, 14]. It provides the user with an easy-to-understand tree structure of a parsed XML document. Parsed XML data is presented in a JDOM Document. It can be manipulated and again stored in XML form. JDOM aims to provide a JAVA-based alternative to the two leading JAVA XML parsers, DOM and SAX, but also integrates well with them. Both DOM and SAX are more complex and more difficult to understand than JDOM, but provide extra functionality in specific situations. JDOM aims to satisfy the 80/20 rule of usability, namely to solve 80% of the problems with 20% or less of the effort.

JDOM allows an XML document to be seen as a whole and makes any member of the document available at any time. Methods are provided for handling the construction and modification of XML constructs, and Java classes are used for input and output (`InputStreams`, `OutputStreams`, `Files`, etc.). It supports validation through DTD's and elements are created by direct object instances.

Using JDOM in the development allows the programmer to use XML without worrying about the details like syntax and parsing.

3.3 Services-based Architecture

At the base of the services-based architecture is the viewpoint of the ground support system and satellite or satellites used to attain a set of goals. The ground support system can be seen as consisting of a group of services designed to provide the functionality needed to manage the satellite mission, an idea developed in [15] and [1]. A certain set of these services are mission-specific and are ground-based as well as flight-based. There is also a set of standard services used by every mission. These services form the core of the ground support system and are shared and re-used in different missions. They fall into mainly two groups, namely a group providing application type functionality (like orbit prediction) and another for data acquisition, processing and storage.

Services needed to provide the functionality needed for future satellite projects are presented in figure 3.1 and will be discussed in the following part of this chapter.

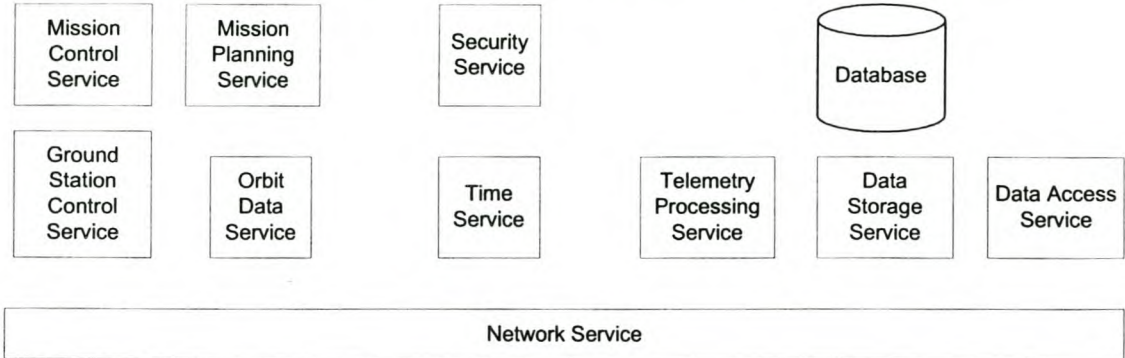


Figure 3.1: Services-based architecture

3.3.1 Network Service

The network service provides a network bus for components to function as a distributed system with total transparency. In other words, the components are developed without the need to know where they will be distributed, or that they will be distributed at all. This allows developers with the needed scientific expertise to develop components without the need to learn complex network programming. The basis of communication in the network service is XML-RPC. Functionality can be added to the basic network service. This includes encryption via SSL, service monitoring and a directory to locate active services.

3.3.2 Data Storage Service

The center of the data processing layer of the ground support system is the database. The data storage service provides the functionality to store data in the database. In an automated system this will typically be done by the telemetry processing service. Security is provided by the DBMS and prevents unauthorised users from entering data into the

database. The data storage service is therefore an XML-RPC interface for entering data into the database.

3.3.3 Data Access Service

The data access service provides an XML-RPC interface for retrieving data from the database. Data presentation clients receive the data in XML format and can display the data as needed without the database server having to format the data according to the platform or application used. For example, presenting the data on the web becomes trivial with a Java applet acting as an XML-RPC client displaying the data.

3.3.4 Orbit Data Service

The orbit data service provides orbit information for a number of different satellites. This data is used in the planning and operation of the mission. An example of where the data provided by this service would be used, is when tracking a satellite during a pass over a groundstation. The following data is provided by most orbit propagating software, and could be made available by the service:

Satellite data:

- Sub-satellite longitude.
- Sub-satellite latitude.
- Azimuth.
- Elevation.
- Time of acquisition of signal if out of range.
- Time of loss of signal if in range.
- Footprint radius.

- Slant range distance between satellite and groundstation.
- Altitude.
- Orbital velocity.
- Orbit number.
- Visibility.
- Doppler shift information.
- Satellite Keplerian element data.

Other data:

- List of satellites.
- Sun azimuth and elevation.
- Moon azimuth and elevation.
- Default groundstation location.
- Current time.

The returned satellite data can be requested for the default groundstation location and current time, or a query can be made for a specific location and time.

3.3.5 Time Service

The time service is used to ensure that the distributed network is time-synchronised. The Network Time Protocol (NTP) is a protocol specifically used for time synchronisation and many clients and servers are freely available. NTP provides accuracy in the order of milliseconds on a LAN. From experience in the SUNSAT program this is more than accurate enough.

3.3.6 Security Service

Security on the distributed system is provided on an individual service basis. Access to services like the data storage service has to be restricted to users authorised to use them to protect the system and data from malicious damage. Currently security is implemented in each service that requires it, but it can be changed to a central security service where users are first authenticated and then given access to authorised services. Users are given a key by the security service and then use this key to access the other services. The services then authenticate the user according to the key from the security service.

3.3.7 Telemetry Processing Service

This service provides processing of raw telemetry data received from a satellite. It will typically process the raw data and store it in the database using the data storage service. The telemetry storage and forward mechanism for each mission probably will not be the same, so the telemetry processing service will be mission-specific. A new processing component will have to be developed for every mission.

3.3.8 Groundstation Control Service

The groundstation control service provides distributed control over a specific groundstation. This includes control over all the hardware like radios and antennas and the tracking involved. Deploying multiple groundstations means duplicating and adapting this service to provide a standard interface for controlling each groundstation. Standardisation of the interface to these services makes distributed control of multiple groundstations possible and allows a higher level of automation to be included, by making these services available for use by higher level services. As an example, an operator uses the mission control service to command a specific satellite. The mission control service then automatically schedules the use of the correct groundstation control services to communicate with the satellite when it next comes into range.

3.3.9 Mission Control Service

The mission control service provides the functionality to manage all components and services used for the control of the specific mission. It provides a front-end to send commands to the satellite and receive data from the satellite. It uses the groundstation control service to communicate with the satellite. As automation is introduced it could schedule the use of the correct groundstation automatically by using the orbit data service to see which groundstation will next be in contact with the satellite.

3.3.10 Mission Planning Service

This service provides a transparent direct interface to the satellite and experiments. Automation of the ground system can be seen as a second layer of services on top of the services providing basic functionality. These basic services can be used together to control a satellite mission manually, like the manual control of the current SUNSAT groundstation reviewed in section 2.2. The automation services layer then automates the use of the basic services by providing a higher level of abstraction for the user to work with. This enables non-technical users (like a scientist doing an experiment) to use the system without the need for detailed training on the use of the system. The mission planning service provides an interface to these automated services so that high level mission planning can be done. Automation algorithms then break down the tasks ordered by the user into tasks performed by the basic services, and schedules these tasks to be performed when necessary.

3.4 Conclusions

The use of Java and XML, together with the tools that emerged from these technologies, will allow the efficient development of a distributed ground support system. The architecture of the system should be services-based, and including the services described should lead to an efficient system with the needed functionality.

The tools and technologies presented in this chapter will be used in the design and implementation of a distributed groundstation in the following chapters. In the next chapter a Unified Modeling Language (UML) design of the system be presented. Use cases for each service will be discussed and recommendations will be made on how to implement the services.

Chapter 4

High Level UML Design

4.1 Services-based Architecture Design

The services-based architecture proposed in section 3.3 is developed further in this chapter. The functional design of each of the proposed services is described using UML use case diagrams. Suggestions will also be made on implementing each service. In section 3.1.1 Java is proposed as development language. The system design will therefore be discussed in the context of Java as development language.

4.1.1 Network Service

The network service provides a network interface for access to software components. The components are registered as services and are available for use through a standard XML-RPC interface.

The network service provides the following functionality (see figure 4.1):

Add service. It allows the registration of classes to the XML-RPC server. The methods of the class or group of classes then provide the functionality required for a specific service.

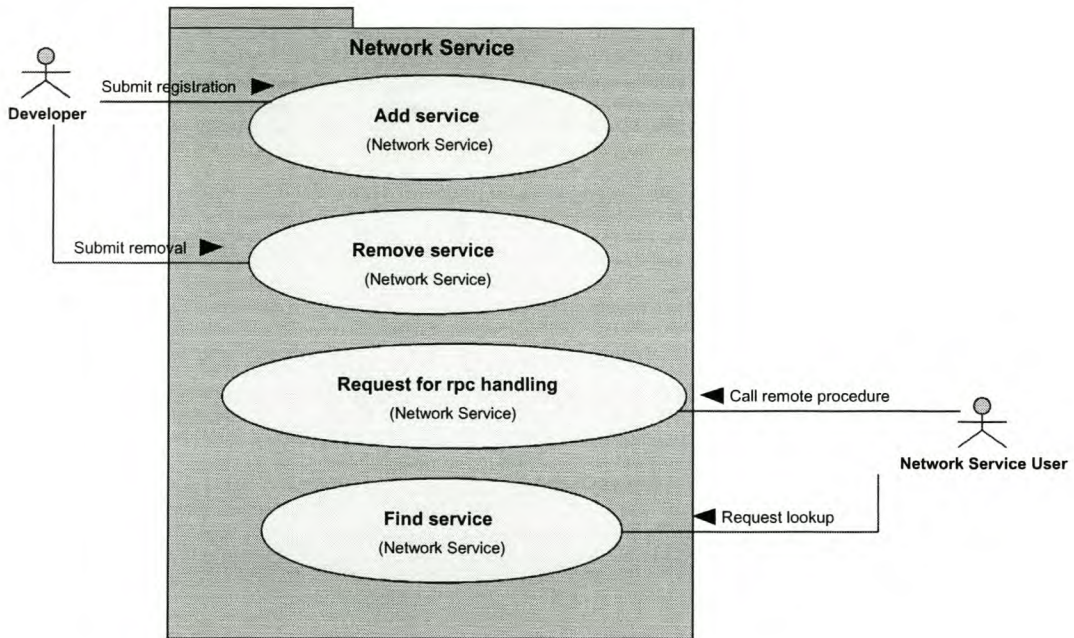


Figure 4.1: Network Service

The class name is registered to a XML-RPC handler name. The handler name is then used by a client to call any public methods in the registered class remotely. The registration data is stored locally for each server and updated as services are added. In section 3.1.2 the use of XML as data description language is proposed. It therefore follows logically that the configuration information for the server and the registered handlers should also be stored in XML form. This will lend the portability of XML data to the configuration data and provide the opportunity of distributed management of the XML-RPC servers. An example of where this would be an advantage, is where a directory service is created for the services. The XML configuration data could then be processed by the directory service, independent of the network service implementation.

Remove service. It allows for the removal of a class from the XML-RPC server. The configuration data is also updated to exclude the class the next time the server starts up. The adding and removal of classes are done through an XML-RPC call to the XML-RPC server. The server registers itself as a XML-RPC handler at startup to provide access to its add and remove methods.

Request for rpc handling. When a client initiates a remote procedure call, the XML-RPC server parses the XML request to find the corresponding method in the class registered to handle the procedure call as well as the parameters sent to it. The method is then invoked with the supplied parameters and the returned value is converted to XML and returned to the client.

Find service. The XML-RPC server returns a list of handlers it has registered on request. This, together with a list of known XML-RPC servers, can be used to locate any service on the network. The functionality can be extended by a directory service that stores information about available services centrally. The directory service can then be queried by the client to find the location of a service and will allow services to be relocatable dynamically. In the context of the SUNSAT project and similar future projects this is not seen as an important function. The services provided on the network will be static in location and well known, therefore dynamic service discovery is not implemented in this basic design but can be added later if necessary.

It is recommended that the Java XML-RPC API, developed by Hannes Wallnofer [16], be used in the implementation of the network service. The API provides a web server implementation used to receive the incoming rpc calls. Handlers can be registered with the server to handle the incoming requests. The server handles the XML parsing and parameter extraction and saves time with the development of parsing code. It is also recommended that the Java Secure Socket Extension (JSSE), a Java Secure Socket Layer (SSL) implementation, be used to implement secure communication between the components.

4.1.2 Telemetry Processing Service

The telemetry processing service provides the ability to process raw telemetry data received from a satellite and to store it in a standard format in the data archive. The data is sent to the telemetry processing service by a data source (which could be another service in an automated system), processed and stored in the archive using the data storage service.

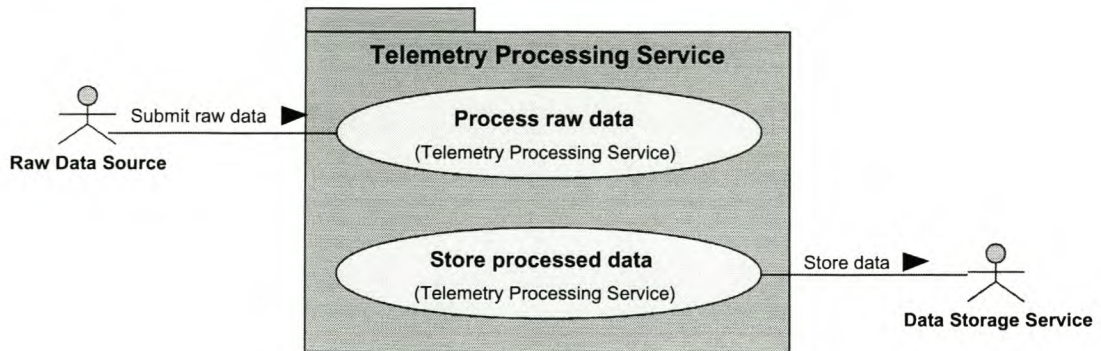


Figure 4.2: Telemetry Processing Service

The use case diagram of the telemetry processing service consists of two use cases (see figure 4.2):

Process raw data. Telemetry data is stored in different formats by different satellite missions, therefore the raw data has to be processed to extract the telemetry data for storage. In the case of multiple missions, the processing service will have to know the source of the data to be able to process it correctly. With each new mission a new processing service will have to be implemented. In the case of the SUNSAT mission, the existing telemetry processing software can be used by simply providing a Java interface to the existing C program and then registering the class as a service.

Store processed data. The extracted telemetry data is stored in the data archive through a remote procedure call to the data storage service.

4.1.3 Data Storage Service

The data storage service is used to store data in the data archive using XML-RPC. The service interfaces with the DBMS using SQL. Facilities like security and data consistency provided by the DBMS are therefore available to the data storage service as well.

The use case diagram for the data storage service can be seen in figure 4.3:

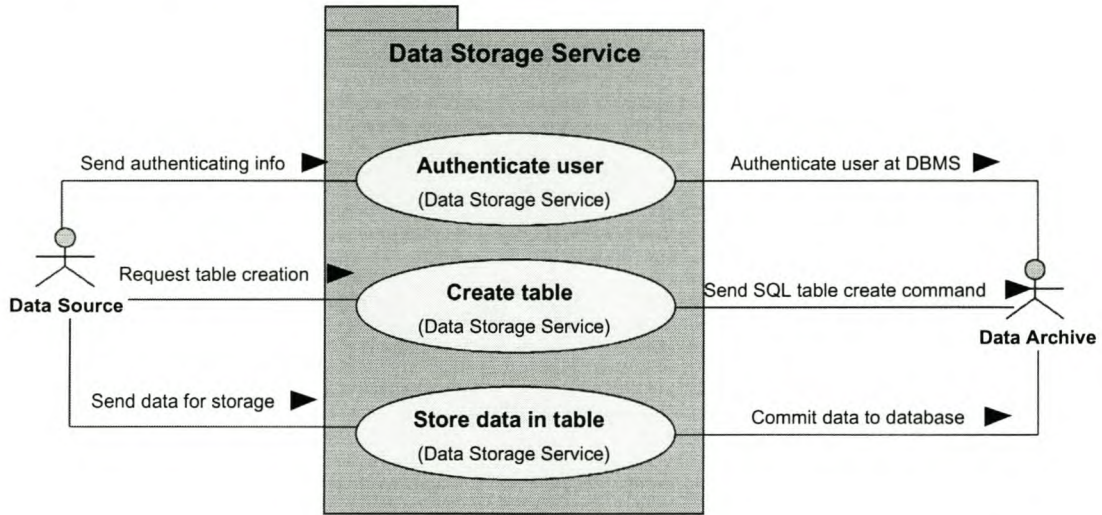


Figure 4.3: Data Storage Service

Authenticate user. Authentication is necessary to prevent unauthorised users from storing data in the data archive. A username and password must be provided with any data submitted for storage. This is then used to log in to the DBMS.

Create table. An authorised user can create tables in the database. When an XML-RPC is received, a SQL command is sent to the DBMS to create the table.

Store data in table. An authorised user can store data in the database using an XML-RPC call. The DBMS is then used to store the data in the database.

4.1.4 Data Access Service

The data access service provides an interface to extract data from the data archive. Display clients will be primary users of this service. Figure 4.4 shows the use case for this service:

Retrieve formatted data. An XML-RPC request for data is translated to a SQL command. The data is then retrieved from the database by using the DBMS and returned

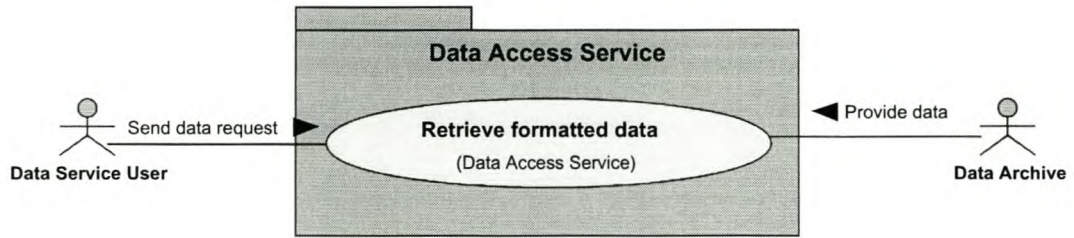


Figure 4.4: Data Access Service

to the client. The fact that the network service is based on a web server makes the creation of a web gateway to make the data available on the web unnecessary. The portability of the XML data makes display on a wide range of clients possible.

4.1.5 Groundstation Control Service

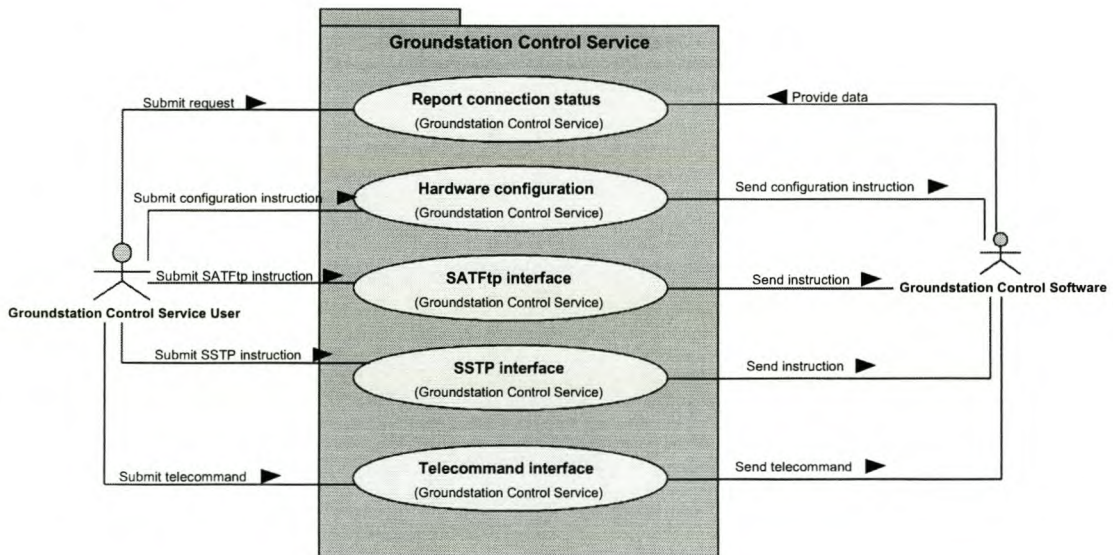


Figure 4.5: Groundstation Control Service

The groundstation control service provides an XML-RPC interface to control a specific groundstation. The XML-RPC handler acts as a Java wrapper to the software controlling the groundstation to enable remote control of the groundstation. The use case (see figure

4.5) diagram is based on the functionality of the current SUNSAT groundstation. Future groundstations will necessitate the creation of a different interface.

Report connection status. The client can request the status of the connection to the satellite.

Hardware configuration. Remote configuration of all hardware in the groundstation, like the radios, antennae and modems, is possible. The status of the hardware can also be queried.

SatFTP interface. An interface is provided to make communication with the SatFTP server on the satellite possible through XML-RPC. Any downloaded data is returned to the client and will probably be passed on to the telemetry processing service by the client in an automated system.

SSTP interface. SSTP commanding is made available remotely by the XML-RPC interface.

Telecommand interface. Telecommands can be sent to the satellite when in contact. An XML-RPC interface makes remote commanding of the satellite possible.

4.1.6 Orbit Data Service

Many of the services and clients in the ground support system need satellite orbit data to know where a satellite is at a specific point in time. The mission control service, for instance, has to know when the satellite is in range of a certain groundstation to calculate which groundstation to use when commanding a satellite. The orbit data service provides orbit data when queried. Rather than writing an orbit propagator to generate the data, any orbit propagating software can be used and wrapped in a Java wrapper. PREDICT is a freeware software product that runs under the Linux operating system. It is recommended that PREDICT be used as orbit propagator, as it already has network functionality and the source code is available for free. The existing network interface provides real-time data for 24 satellites for a specific groundstation location. It can be extended easily to provide

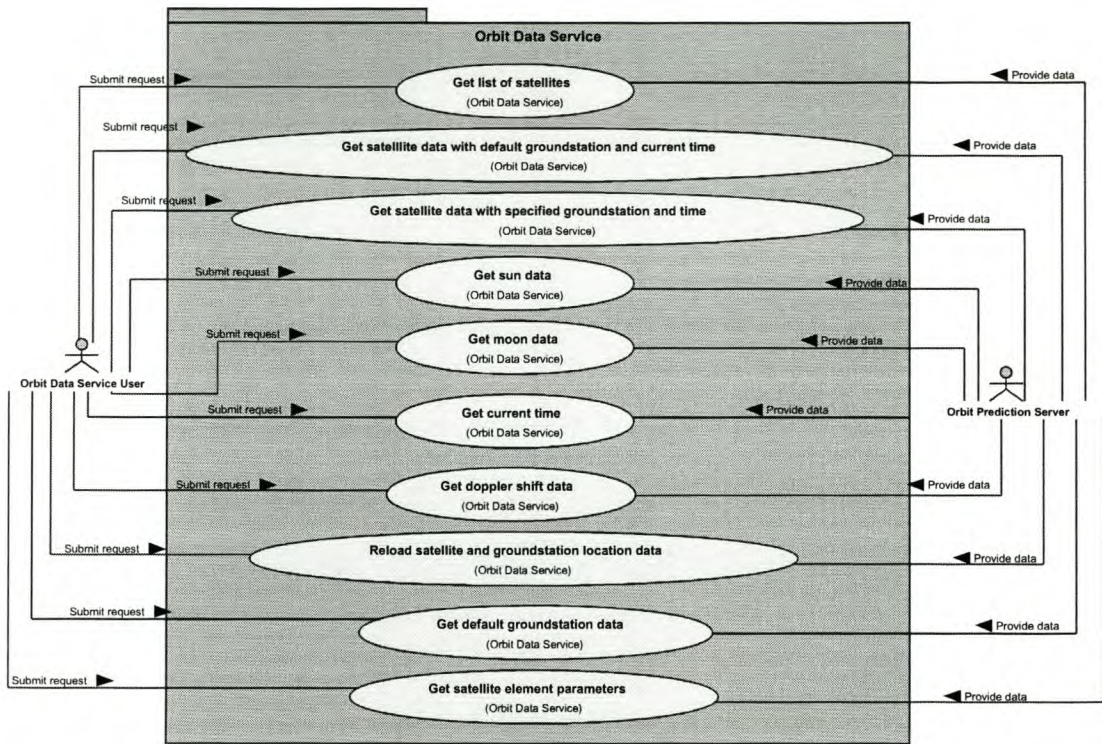


Figure 4.6: Orbit Data Service

data for a specific location and time. The orbit data service can then query the software through the network interface and provide the data via XML-RPC. The use case diagram for the service can be seen in figure 4.6. These use cases describe functionality provided by PREDICT available through XML-RPC by the orbit data service.

Get list of satellites. PREDICT can provide satellite data for 24 different satellites. A list of available satellites is available to a client on request.

Get satellite data with default groundstation and current time. The real-time satellite orbit data is provided for a specific default groundstation location.

Get satellite data with specified groundstation and time. When the client specifies a satellite name, time and groundstation location, the satellite data for that location and time is returned.

Get sun data, Get moon data. The real-time azimuth and elevation data for the sun and moon are returned on request.

Get current time. The current time at the server, in unix format (seconds since midnight UTC on January 1, 1970), is returned on request.

Get doppler shift data. The doppler shift data for a satellite (normalised to a downlink frequency of 100MHz) is returned on request.

Reload satellite and groundstation location data. The PREDICT server can be forced to reload the configuration data stored in its local configuration file. This is used when configuration data for a satellite is updated or the default groundstation location is changed. All other data returned from the time of the forced reload is then in relation to the new configuration.

Get default groundstation data. The default groundstation location data, as stored locally for the PREDICT server, is returned on request.

Get satellite parameters. The satellite orbit parameters used to calculate the orbit data are returned on request.

4.1.7 Mission Control Service

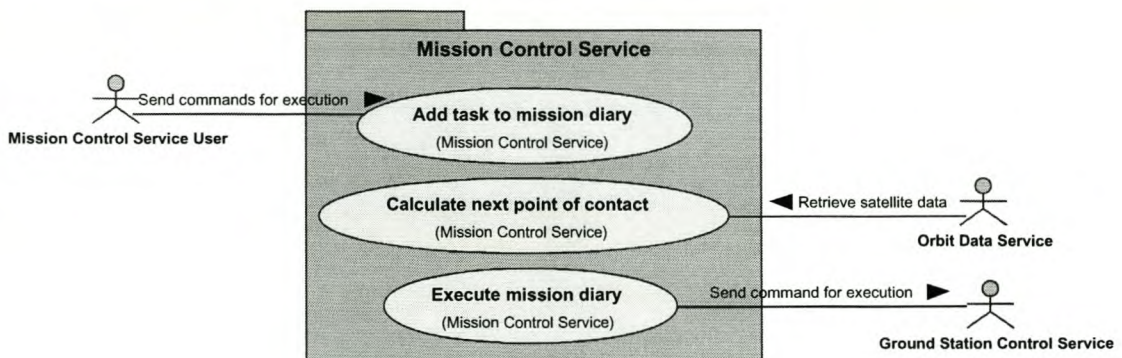


Figure 4.7: Mission Control Service

In a ground system with multiple groundstations the management of communication with the satellite becomes more complex than with a single groundstation. With only one groundstation, communication with the satellite can be done through interfacing with the groundstation control service itself. With multiple groundstations, it is necessary to provide a higher layer of abstraction and present the multiple groundstations as one virtual groundstation to the user. This layer of abstraction has to decide which groundstation to use for communication and send the commands to the selected groundstation for execution. This layer of abstraction is provided by the mission control system. By using a simple diary mechanism for the virtual groundstation, the mission control service can receive commands for execution and execute them at the next point of contact. Figure 4.7 shows the use cases for this service:

Add task to mission diary. The ground system diary contains a sequence of commands to send to the next groundstation that is in contact with the satellite. The commands are submitted to the diary remotely via XML-RPC.

Calculate next point of contact. For the mission control service to decide which groundstation to use for communication with the satellite, it uses the orbit data service to obtain information about the satellite's next point of contact.

Execute mission diary. When the next time of contact arrives, the mission control service executes the commands stored in the mission diary.

4.1.8 Mission Planning Service

To be able to provide a completely transparent link between the user on the ground and an experiment on the satellite it is necessary to provide a layer of abstraction on top of the mission control service. The mission planning service provides a high level interface with instructions that do not necessarily map directly to commands sent to the ground control service. As an example, a user can give an instruction effecting an experiment on the satellite. The mission planning service then translates and maps the instruction to the instruction set available through the mission control service. These instructions are then sent to the mission control service and the data returned is displayed to the user in

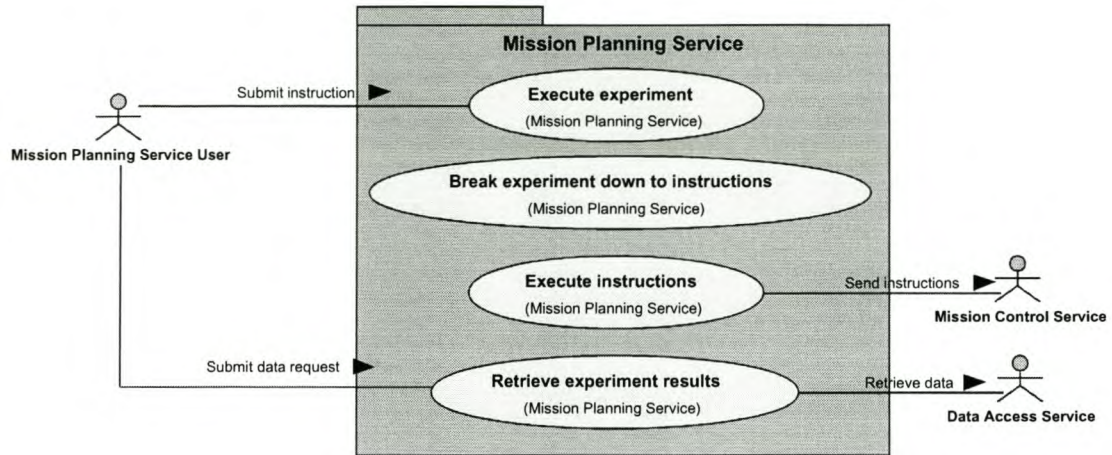


Figure 4.8: Mission Planning Service

a meaningful way. The ground support system therefore becomes accessible to users not fully trained in the working of the underlying system. It is therefore possible to provide direct control to owners of a third-party experiment over their experiment. Smaller ground support teams can therefore be used. This leads to a big saving in operational costs. The mission planning service can check the requested commands against the current state of the satellite, and therefore provide a sanity check for the protection of the satellite. The use case diagram for the mission planning service is shown in figure 4.8.

Execute experiment. High level instructions are submitted to the service via XML-RPC by the service user.

Translate instructions. The high level instructions are broken down and mapped to commands executable by the mission control system.

Execute commands. Once the high level instruction is translated, it is sent to the mission control service for execution.

Retrieve experiment results. The mission planning service also provides the user with the functionality to view experiment results. The data is retrieved using the data access service.

4.1.9 Security Service

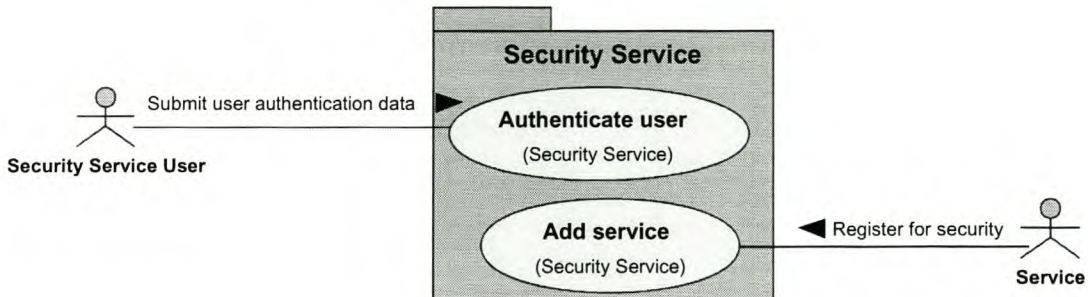


Figure 4.9: Security Service

The security service provides user authorisation to the ground support system. Some of the services need to be protected from unauthorised access. This can be done at each service or the authentication can be centralised. If done on a per service basis, each service requiring authentication must receive a username and password as additional parameters for authentication. If done centrally, the user must first log in at the security service before access is granted to certain services. It is recommended that the authentication be done at each service requiring it initially, because it is the simplest to implement. If it is later found to be inadequate, the security service can be extended to provide a central authentication mechanism. One way of implementing central authentication is to give the user a secret key generated by the central security server. This key must then be sent with all requests to services the user is authorised to use. Services requiring security then register at the security service to fall under the service's protection. The use case diagram for the security service is seen in figure 4.9.

Additional security can also be implemented to protect the whole system from unauthorised access. A firewall can be installed to separate the groundstation from the outside network and to only grant access to authorised users. By using SSL to encrypt the communication, secure communication between components can be established, as proposed for the network service.

Authenticate user. A user is authenticated when the username and password provided by the user matches a username and password combination in the security database. This

can be done at each service or centrally.

Add service. When a central authentication service is implemented, services register at the security service to be able to use the service for user authentication purposes.

4.1.10 Time Service

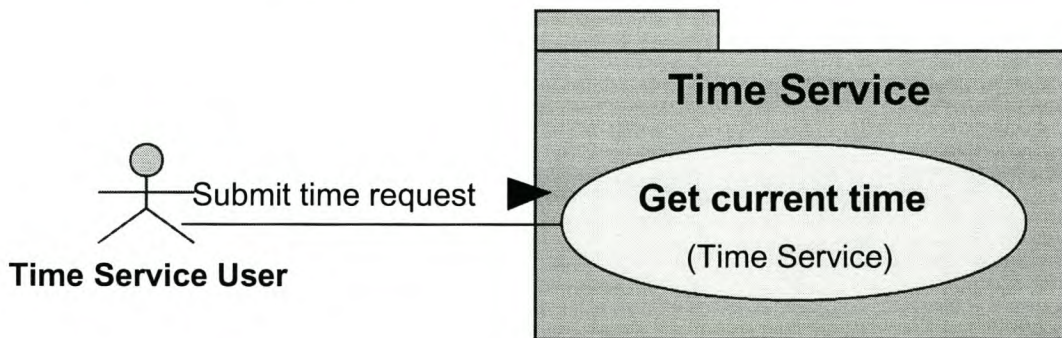


Figure 4.10: Time Service

It is critical that the time across the whole ground support system be synchronised. The time service provides a standard time for all clients to synchronise to. Every PC on the system must run a time service client to ensure that it stays in synchronisation with the standard time. It is proposed that the time service be implemented simply by accessing any of the many NTP servers on the internet. Many clients are available for use on the internet and can be installed on each station in the ground support system. The use case diagram for the time service can be seen in figure 4.10.

Get current time. The NTP client synchronises the time on the station where it resides with the time on the NTP server at regular intervals.

4.2 Proposed Clients

The services-based architecture described above provides the functionality needed for the ground support system with client software providing the user interface to the system. One of the biggest advantages of the services-based architecture is the flexibility it provides by separating the functionality from the user interface. Clients can be developed and changed as needed without changing the basic architecture of the system. As an example of the relative easiness of developing these clients, a set of clients will be proposed to provide the same functionality as the current SUNSAT groundstation.

4.2.1 Hardware Control and Status Display

This client provides the interface to configure the hardware of the groundstations and to display the current status of the hardware and any links with satellites. It uses the groundstation control service for each groundstation to provide an interface to control all the groundstations at one point remotely.

4.2.2 SatFTP Client

This client provides an interface to communicate with the SatFTP server on the satellite through the groundstation control service. Files can be downloaded from or uploaded to the server. This facility is mainly used to download telemetry and experiment data files and upload diary and software update files. It uses the groundstation control service for remote access to the SatFTP functionality and the orbit data service to determine which groundstation to use for communication with the satellite. Downloaded telemetry data files can be sent to the telemetry processing service for processing or stored for future processing.

4.2.3 Telecommand Client

The groundstation control service provides the functionality to send telecommands and predefined control commands to the satellite. The telecommand client gives the operator the ability to interface with this functionality and to send commands remotely to the satellite while in contact.

4.2.4 Orbit Prediction Client

This client uses the orbit data service to provide a graphical representation of the location of a satellite. Data, such as the time of the next point of contact, visibility and range, provided by the orbit data service, can also be displayed to the user. This client can be used in the planning of communication with the satellite and other mission operations planning.

4.2.5 Telemetry Display Client

The client provides an interface to extract telemetry data from the data archive using the data access service. Telemetry data is displayed graphically for analysis by operators.

4.2.6 Network Control Client

The network control client provides control over the network service. It is used to register and remove handlers at the XML-RPC servers and monitor the status of the servers.

4.3 Design Evaluation

In the design of a distributed system one of the most important aspects is the interfacing between the components. The communication between the components influences the performance of the overall system as well as its usability. It is therefore important that the design is tested against the principles of an efficient distributed system. The design can be evaluated according to the following criteria [17]:

- **Security.** All applications have to control access to data to prevent unauthorised access. With distributed applications, controlling access is more difficult because of the dispersion of data and software.
- **Synchronisation.** The distributed system has to protect the integrity of data accessible over the network. Multiple updates of data have to be synchronised to ensure that a request for data always results in the most recent data available.
- **Compatibility.** The components of the distributed system have to be able to communicate across different operating platforms and languages to give maximum flexibility.
- **Availability.** To ensure reliable operation of the system, the separate components have to be available when needed. Failure of a single critical component results in the failure of the whole system.
- **Performance.** The design of the distributed system has to optimise the tasks the system was designed for in the first place. Performance is also a very important factor in delivering data reliably and on time.
- **Complexity.** Distributed systems are complex to design and maintain. Thorough planning of the system and the interfaces between components is essential to keep the system as simple as possible. The level of complexity of the system also influences the design time and programming complexity and should therefore be kept as simple as possible.

4.3.1 Security

The proposed design handles security through the security service. Access to each component can be controlled separately and existing security provided by the DBMS is used to protect the data in the data archive. SSL encryption ensures secure communication between components. Authentication information can therefore be sent as part of the plain text XML without the danger of interception.

4.3.2 Synchronisation

By providing a single point of storage access to the data archive, through the data access service, the synchronisation of data in the data archive is ensured. The DBMS ensures the integrity of the data in case of multiple simultaneous updates to the archive.

4.3.3 Compatibility

Compatibility between components in the system is one of the main advantages of using XML-RPC for communication. No bounds are placed on the language or platform of the implementation. Although most implementations of XML-RPC use HTTP as transport protocol, it could be implemented using any underlying transport protocol. Therefore there is a possibility of using XML-RPC over AX.25 to include the satellite as part of the distributed system in future.

4.3.4 Availability

By duplicating critical components in the system, redundancy can be used to ensure availability of these components at all times. This is another big advantage distributed systems have over centralised systems. The modularity of the proposed design makes implementing redundancy simple. As an example, in the proposed design the orbit data

service could be provided by several orbit propagators in different locations. The client then simply queries a list of servers that provide the orbit service, and switches to another in case one server becomes unavailable.

4.3.5 Performance

One of the main reasons for developing the distributed ground support system is to enable more efficient operation. The design allows automation to be introduced, and the distribution of the components allows for better utilisation of resources. The proposed design therefore allows for more efficient support of satellite missions in terms of resources. This outweighs the disadvantage of the slower performance of a distributed system over a centralised system. Processes that require centralised performance, can be implemented and distributed as a single service to eliminate the possible performance penalty of distributed communication between processes.

4.3.6 Complexity

The proposed design aims to minimise the complexity of implementing the system through the modularity of the system. The components of the system can be developed independently of each other and distributed on the network without any complex networking code through pre-defined interfaces. While the development of a distributed system is bound to be more complex than a centralised system, this design aims to provide a solution that is simpler than the more complex solutions based on other architectures. To quote Jon Udell [18]:

“Does distributed computing have to be any harder than this? I don’t think so.”

Chapter 5

Implementations

To test the viability of the system design as discussed in chapter 4, some of the services and clients were implemented. The most important service of the services-based architecture is the network service. It is at the center of the whole distributed system and had to be functional before any of the other services could be implemented. It was therefore the first service to be implemented. A configuration client for the network service was also implemented. To demonstrate the development of another service, the orbit data service was implemented. The goal was to show that the service can be developed and distributed without writing any extra code to make distribution possible. A client that uses the data obtainable from the orbit data service to display the location of a satellite graphically on a world map was also developed.

5.1 Network Service

A block diagram of the network service implementation can be seen in figure 5.1. The XML-RPC server handles incoming RPC requests and passes the parameters on to the handlers registered at the server. The configuration information is stored in a configuration file in XML format. The local client is used to edit the configuration file directly, while the remote client uses remote procedure calls to the server to change the

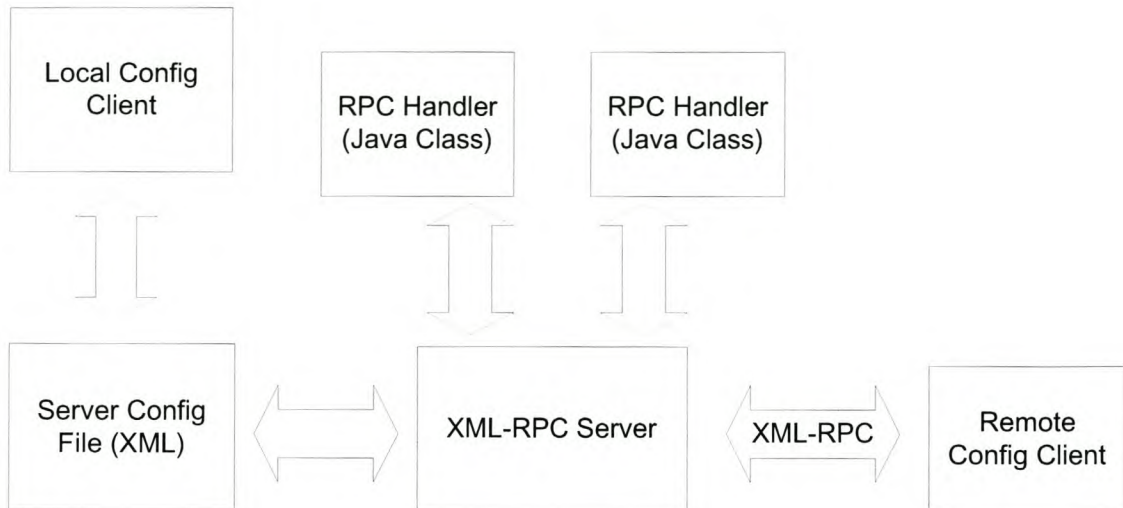


Figure 5.1: Network Service Implementation

configuration of the server.

The class diagram for the network service can be seen in figure 5.2. It shows the class structure of the service and will be explained in more detail in this section.

5.1.1 Use Case Implementations

The most basic functionality provided by the network service is the ability to receive incoming XML-RPC requests, invoke the correct method and return the result. This is represented by the **Request for rpc handling** use case. Therefore, a server that can receive and parse XML-RPC requests submitted via a HTTP POST operation is required. The XML-RPC server was implemented using the XMLRPC-Java library developed by Hannes Wallnofer [16]. The library contains an `XmlRpcServer` class used to parse the XML-RPC request, execute the correct method and return the result. The name of the class is somewhat misleading, because the class actually does not provide a listening server. It rather works on any `InputStream` passed to it, therefore a server class is needed to handle the socket connections. The `XmlRpcServer` returns the data as a byte array and can then be returned by the server via the `OutputStream`. Classes can be registered at the

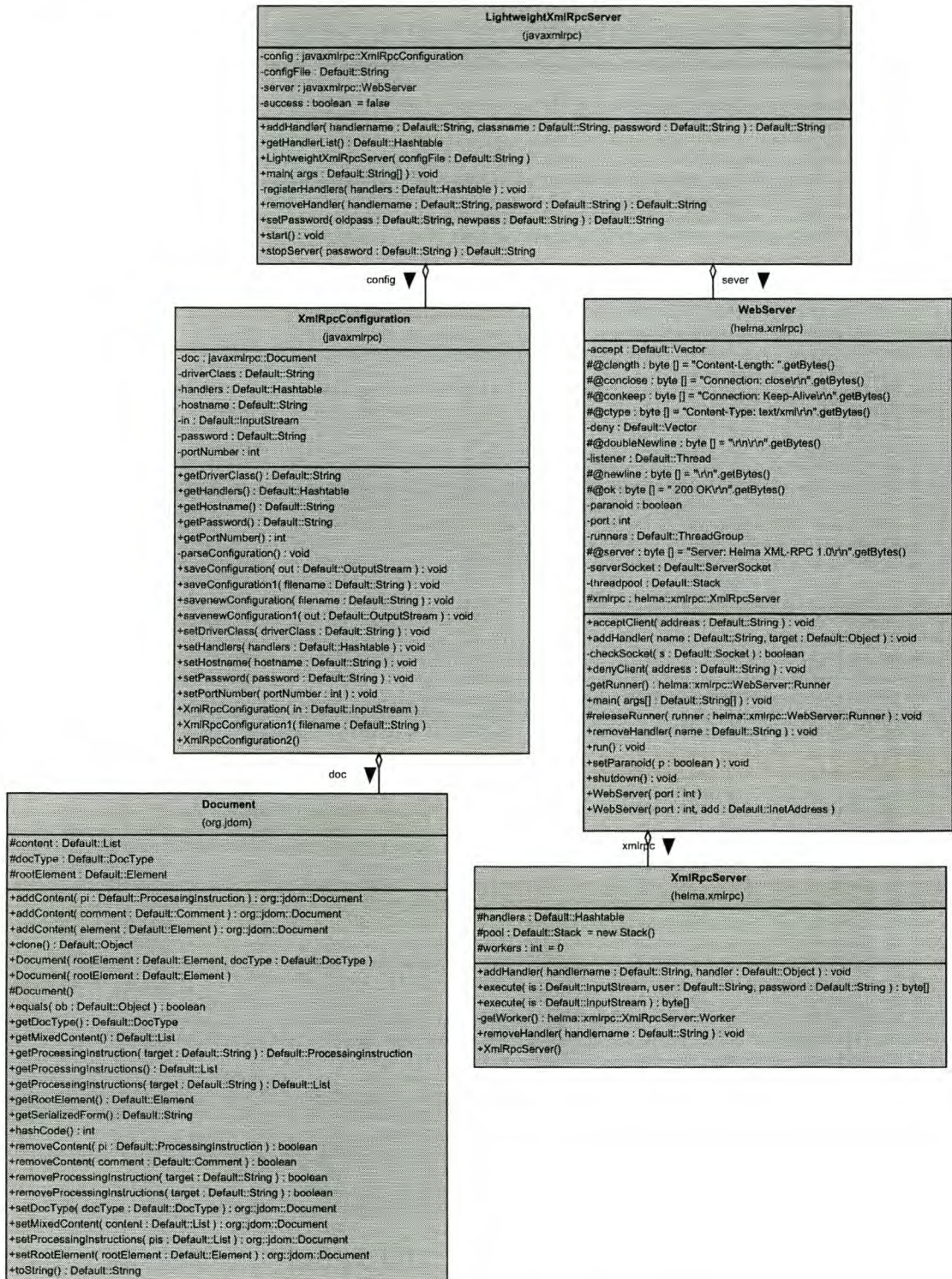


Figure 5.2: Network Service Class Diagram

`XmlRpcServer` to handle the incoming XML-RPC requests. The library also includes a minimal web server implementation in the `WebServer` class. It listens on a port and passes incoming HTTP data to the `XmlRpcServer`. The main class developed to implement the network service is the `LightweightXmlRpcServer` class. It is based on a XML-RPC server proposed in [14]. The `LightweightXmlRpcServer` class uses an instance of the `WebServer` class to listen for HTTP requests and return the result.

The **Add service** and **Remove service** use cases are implemented in the server by registering a Java class to a given handler name using the `WebServer` class's `addHandler` and `removeHandler` methods. The configuration information for each server is stored locally in an XML file. The `XmlRpcConfiguration` class is a utility class created to provide methods for writing and reading the configuration information to and from the XML file. The server configuration can be changed by either editing the configuration file, or by adding or removing a handler remotely via XML-RPC. The remote configuration is made possible by the fact that the `LightweightXmlRpcServer` registers itself as a handler when it starts up, thereby making the `addHandler` and `removeHandler` methods available remotely. The `addHandler` method registers the given class to a handler name at the `WebServer` and adds the new handler's information to the configuration file using an instance of the `XmlRpcConfiguration` class. The `removeHandler` method de-registers the class and updates the configuration file in the same way.

The `LightweightXmlRpcServer` class can be queried through the `getHandlerList` method to provide a list of handlers registered at the server. Client software can query each XML-RPC server to find the location of a service, or a central directory can hold the information for each server. A directory server will hide the service location from the clients and make dynamic service discovery possible.

Figure 5.3 shows the output of the server at startup. The configuration information is obtained from the configuration file and the specified handlers are then loaded.

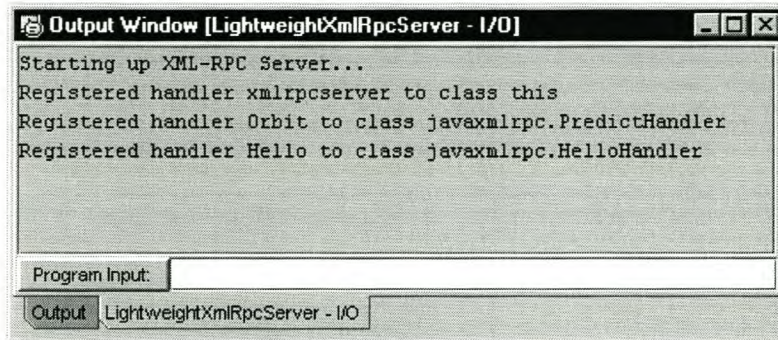


Figure 5.3: XML-RPC Server Startup

5.1.2 Error Handling

The XML-RPC specification [19] defines a payload for a fault message in XML. Error messages are therefore returned to the client through the protocol. **Exceptions** in a Java handler are caught by the `XmlRpcServer` class and returned as formatted fault messages through XML-RPC. On the client side, the message is parsed and if it is a fault message an **Exception** is thrown. The impact of all this is that the handler can handle errors in the usual Java way, by throwing and catching **Exceptions**. This is in line with the goal of developing the handler without the need for non-standard code to enable distribution of the code on the network.

5.1.3 Network Service Configuration Clients

Clients were developed to configure the XML-RPC servers of the network service. A remote configuration client was developed to add and remove handlers from the XML-RPC server, and a local client was developed to create or update the configuration file.

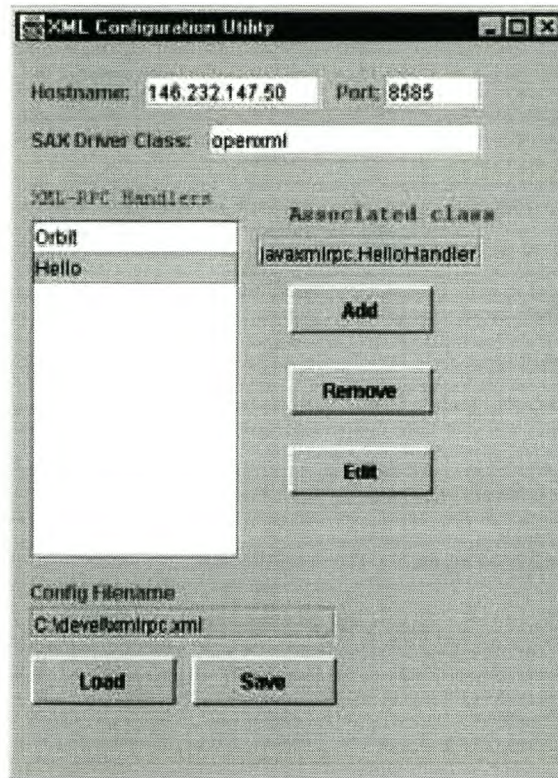


Figure 5.4: Local Configuration Client

5.1.3.1 Local Client

All the data in the configuration file can be changed and saved. The client edits the configuration file directly and does not use the configuration methods exposed in the server. That makes it very useful in the creation of a new configuration file for a server when no previous files exist, because the server need not be running when the configuration file is edited. The client uses an instance of the `XmlRpcConfiguration` class to retrieve the data in the configuration file. The class diagram for the client can be seen in figure 5.5.

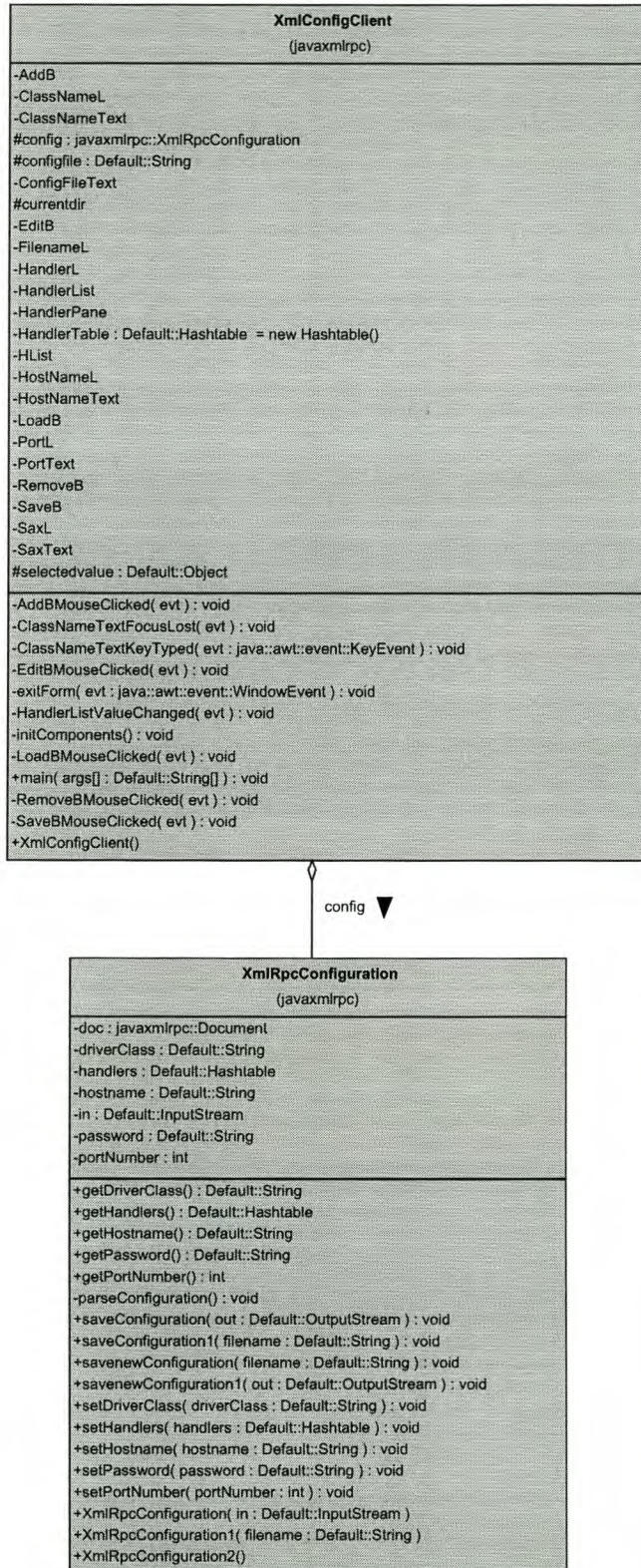


Figure 5.5: Local Configuration Client Class Diagram

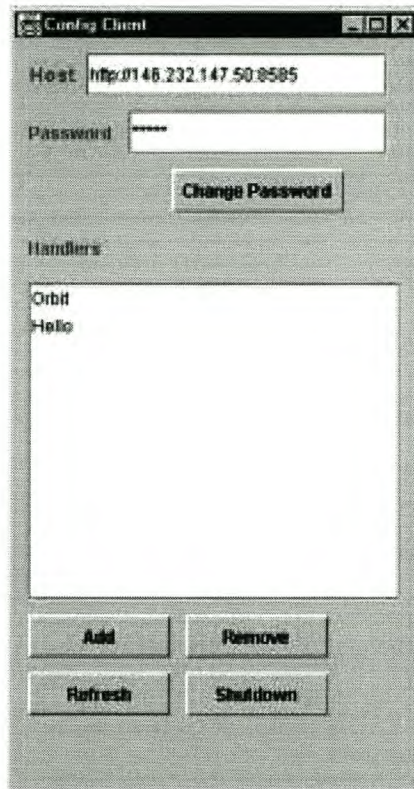


Figure 5.6: Remote Configuration Client

5.1.3.2 Remote Client

The remote client is used to configure the XML-RPC servers remotely. The class diagram for the client can be seen in figure 5.7. Handlers can be added and removed and the remote configuration password can be changed. The client displays a list of handlers registered at a specified server by a remote procedure call to the `getHandlerList` method of the network service. The `XmlRpcClient` class is used to implement the XML-RPC communication with the server. Handlers can then be added and removed using the `addHandler` and `removeHandler` methods of the server. These methods are protected by a remote configuration password stored in the server's configuration file. The password is sent as a parameter and checked by the method. The client can also shut down the server using the `stopServer` method. This method is also password-protected to prevent the server from being shut down without authorisation.

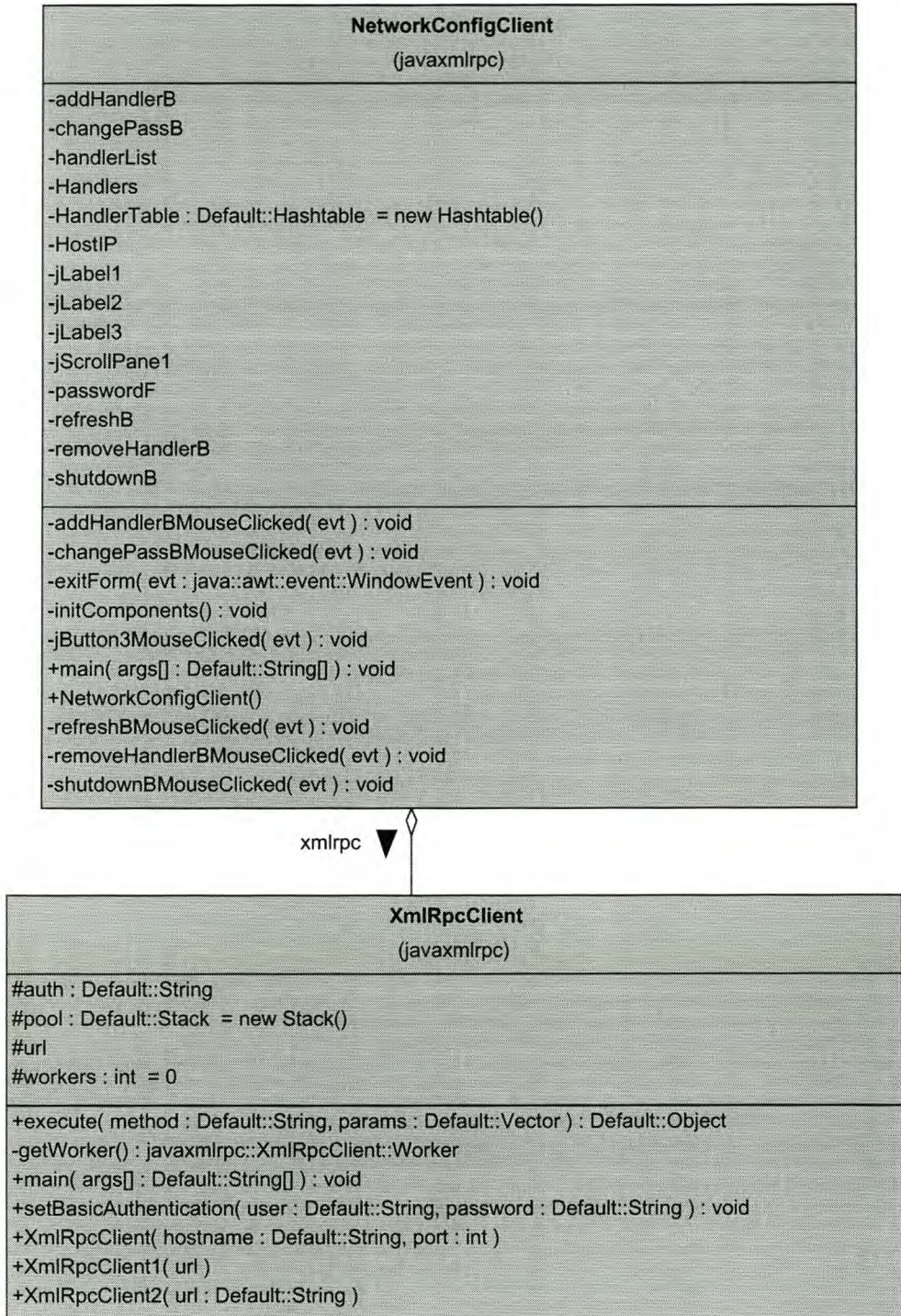


Figure 5.7: Remote Configuration Client Class Diagram

5.2 Orbit Data Service



Figure 5.8: Orbit Data Service Implementation

The orbit data service was implemented to demonstrate the relative ease of integration of COTS products or existing software into the ground support system. It also demonstrates the ease of implementing network services without the need for complex networking code. Figure 5.8 represents the implementation of the orbit data service. The handler uses the orbit propagator to retrieve data requested by the tracking client via XML-RPC.

5.2.1 Use Case Implementations

To implement the functionality seen in the use case diagram of the orbit data service (see figure 4.6), an orbit propagator was needed. The propagator could have been implemented using a standard orbit propagating algorithm, but with many propagators already in existence it was decided to use an existing software package. The PREDICT orbit propagator [20] was chosen to be used in the orbit data service, because it provides the needed functionality and also because it is available under the GNU General Public License. The C source code was available and could therefore be changed as needed. To make it distributable by the network service a Java interface was developed. PREDICT already had a UDP network interface to supply data to a client, but the functionality of the interface was limited to providing real-time data and only for the default groundstation location. The source code was modified to enable the client to specify the time and groundstation location for the requested data.

```

Terminal
File Sessions Settings Help

PREDICT Real-Time Multi-Tracking Mode
Current Date/Time: Mon 25Jun01 08:12:04

Satellite Az El Lat Long Range | Satellite Az El Lat Long Range
-----|-----
OSCAR-10 183 -45 -20 159 45992 D OSCAR-29 342 -29 34 2 7741 D
OSCAR-11 117 -34 -31 252 8138 D OSCAR-36 62 -79 40 186 13167 D
OSCAR-14 191 -13 -72 8 5004 D OSCAR-40 180 -62 0 162 53320 D
PACSAT 13 -38 49 322 8995 D TECHSAT 14 -52 73 294 11085 D
WEBERSAT 344 -84 43 158 13481 D TMSAT 25 -22 20 320 6453 D
LUSAT 24 -11 5 327 4632 D RS-12/13 135 -7 -52 299 4484 D
OSCAR-20 191 -33 -69 129 8443 N RS-15 171 -32 -65 184 9727 D
OSCAR-22 11 -58 80 261 11698 D SUNSAT 79 -26 -6 281 6994 D
OSCAR-23 234 -44 -25 99 10606 N MIR 242 -63 0 117 11631 N
OSCAR-25 5 -49 70 326 10597 D UARS 254 -49 -6 91 10390 N
ITAMSAT 7 -52 75 316 11014 D HUBBLE 195 -58 -28 147 11472 N
OSCAR-27 173 -60 -24 168 11973 N ISS 94 -68 21 208 12203 D

Upcoming Passes
-----|-----
Sun Moon
-----|-----
39.48 Az LUSAT on Mon 25Jun01 08:15:35 UTC 78.54 Az
+22.90 El TMSAT on Mon 25Jun01 08:20:17 UTC -14.04 El
PACSAT on Mon 25Jun01 08:27:34 UTC

```

Figure 5.9: PREDICT Server Mode

5.2.1.1 PREDICT Source Code Modifications

When PREDICT is run in server mode (see figure 5.9), it provides real-time orbit data on request through the UDP interface. For a complete discussion on PREDICT's server mode, refer to appendix B. To generate the data, the software runs in multiple tracking mode and continuously updates global variables with the data. The UDP server runs as a different thread than the computational thread, and when a UDP request arrives, it simply returns the data currently stored in the global variables to the client. To enable the client to request data for a specified groundstation location and time, the source code was modified to compute the orbital data for a specific satellite at the specified location and time. The server thread was modified to set a global flag variable when the `GET_WITH` command arrives. It then waits for the computational thread to compute the values and store them in another set of global variables. As soon as the computational thread is done, it sets another global flag to indicate to the server thread the data is available. The

server thread then returns the data to the client. The computation of the data by the computational thread is done using the existing procedures in the source code. The real-time data stored in the global variables is stored in backup variables, the location and time are changed and the computational procedures are then called as normal. The computed data is then transferred to other global variables and the real-time data restored. The flag signalling the end of computation is then set to enable the server thread to return the data.

5.2.1.2 XML-RPC Handler

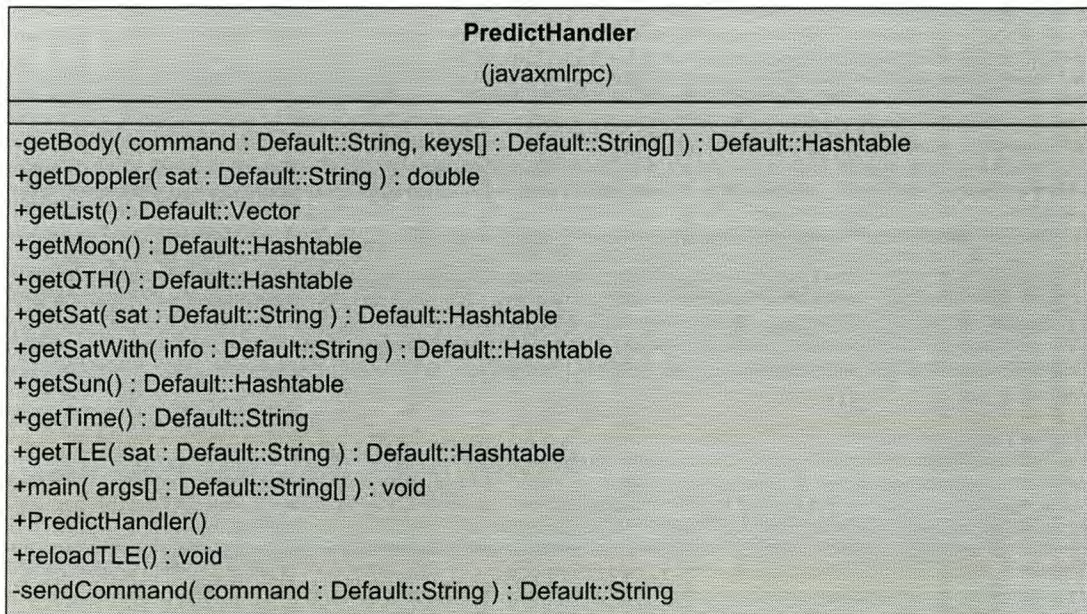


Figure 5.10: Orbit Data Service Handler Class Diagram

To deploy the PREDICT orbit propagator as an XML-RPC service, a handler acting as an interface between the normal UDP interface, used in PREDICT, and the XML-RPC network service was developed (see figure 5.10). The handler basically provides a method for each PREDICT command that is made available through XML-RPC. Each one of these methods sends a UDP packet with the command and parameters to the PREDICT server and then parses the returned data string to extract the requested values. The parsed data is then stored as data types available and returned via the XML-RPC interface.

5.2.2 Tracking Client



Figure 5.11: Tracking Client

The tracking client was implemented to demonstrate the relative ease of developing clients for the ground support system using the available services. The tracking client uses data obtained from the orbit data service to display the location of a satellite on a world map. Other tracking clients usually have to implement the algorithms necessary to compute the location of the satellite, making it a complex piece of software to develop. This client, however, is nothing more than a front-end that displays data retrieved from the orbit data service. Java is ideal as development language for graphic front-end clients, because of the graphics support in the base libraries. The separation of the front-end from the computational service also has the advantage that the computational algorithms can be changed as needed without effecting the client, as long as the data is still supplied in the same way.

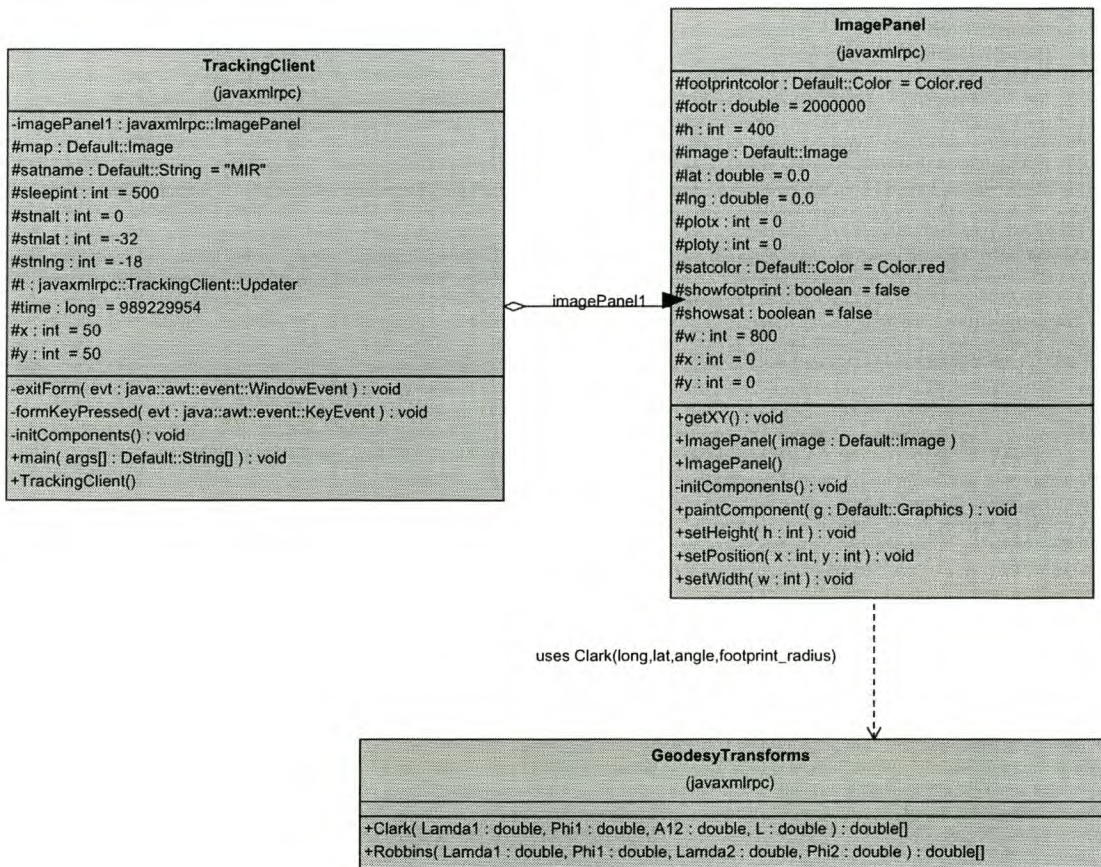


Figure 5.12: Tracking Client Class Diagram

5.2.2.1 ImagePanel Component

The class diagram for the client can be seen in figure 5.12. The `ImagePanel` component was created to provide a standard component for plotting latitude and longitude points on a world map. It extends the `JPanel` Swing component. The `lat` and `lng` fields of the component are used to specify the location of the point to plot. A footprint radius can also be specified and if enabled, plotted. This is done with the `footr` and `showfootprint` fields. Plotting a point and footprint is therefore a simple matter of providing the latitude and longitude of the point and the footprint radius, setting the `showsat` and `showfootprint` fields to `true` and calling the `repaint` method of the component. To calculate the latitude and longitude of the points on the footprint circle, the component uses the `Clark` method from the `GeodesyTransforms` library written by E.M. Hugo [21]. The `Clark` method computes the latitude and longitude of a location, given another location and the direction and distance from the given location. By calculating points for the footprint radius and various angles between 0 and 360 degrees, the footprint can be plotted.

5.2.2.2 XML-RPC Client

The tracking client application uses an instance of the `ImagePanel` component to plot the data received from the orbit data service on a world map supplied to the component. A separate thread is created to poll the orbit data service continuously for the information of the satellite. An instance of the `XmlRpcClient` class is used to communicate with the XML-RPC server. The returned location data is then used to set the location of the point to plot on the `ImagePanel` component, and the `repaint` method is called to update the component. The data is retrieved using a standard XML-RPC call to the `getSatWith` procedure of the orbit data service.

5.3 Conclusions

The network service implementation provides a simple mechanism for distributing components in the system, without the need for complex network programming. The client allows for remote configuration of each network service server.

The orbit data service implementation demonstrates the easy of COTS product integration in the system. The client demonstrates the use of a service to provide information to the user.

Chapter 6

Conclusions and Future Work

6.1 Thesis Results

The outcome of the thesis can be evaluated according to the problem statement in chapter 1. The following important results were obtained:

Services-based Architecture Design Specification. The proposed design for a services-based architecture forms the foundation of a system that is easy to expand and that allows the incorporation of COTS products and existing code. Services can be developed and added to the system without the need for complex network programming.

Introduction of Automation. The system design allows for automation to be introduced by adding services and clients that allow the user to interact with the system on a more abstract level. Automation services make use of the other services to automate routine tasks that previously required human interaction.

6.2 Future Work

The design proposed and partially implemented in this thesis has the potential to form the core of a new ground support system at the University of Stellenbosch. Although this design aims to duplicate the functionality of the SUNSAT ground system, it was designed to expand and evolve as new needs are identified and services added. The design also opens up opportunities for research in different areas:

Automation Technologies. The automation of routine tasks by the ground support system is only the beginning of automation in the system. By deploying more complex decision making systems in the ground system as well as on the satellite it supports, autonomous operation can bring big savings to micro-satellite missions operations.

The use of multiple groundstations and the support of multiple satellites also make automation more complex. Higher level services will require a certain degree of intelligence to make decisions on what resources to use for executing a certain task.

Artificial intelligence technologies like remote software agents and rule-based decision making systems are some of the areas where further study would be helpful.

Telemetry Standardisation. The re-use of components can be increased greatly by adopting standards for data processing. Adopting standards like the Consultative Committee for Space Data Systems' (CCSDS) Recommendation for Packet Telemetry [22] will make the re-use of components and software easier and will standardise data handling across missions in the ground support system.

Inter Service Communication. Communication mechanisms like data broadcasting and subscriptions and callbacks between services have to be investigated to enable data to flow optimally in the system. As new distributed architecture protocols like SOAP mature, the upgrading of the network service has to be investigated.

6.3 Conclusion

The work done in this thesis provides a platform for further work in developing a fully functional distributed ground support system. The services-based architecture of the system provides the functionality and flexibility required for the efficient management and data handling of micro-satellite missions.

The implementation and evolution of the system will lead to a ground support system that can be re-used across different missions and adapted in future for functionality not even envisioned at this time.

Appendix A

XML-RPC Specification

This is the XML-RPC specification by Userland [19].

A.1 Overview

XML-RPC is a Remote Procedure Calling protocol that works over the Internet. An XML-RPC message is an HTTP-POST request. The body of the request is in XML. A procedure executes on the server and the value it returns is also formatted in XML.

Procedure parameters can be scalars, numbers, strings, dates, etc.; and can also be complex record and list structures.

A.2 Request example

Here's an example of an XML-RPC request:

```
POST /RPC2 HTTP/1.0
```

```
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181
```

```
<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
      <value><i4>41</i4></value>
    </param>
  </params>
</methodCall>
```

A.3 Header requirements

The format of the URI in the first line of the header is not specified. For example, it could be empty, a single slash, if the server is only handling XML-RPC calls. However, if the server is handling a mix of incoming HTTP requests, we allow the URI to help route the request to the code that handles XML-RPC requests. (In the example, the URI is /RPC2, telling the server to route the request to the "RPC2" responder.)

A User-Agent and Host must be specified.

The Content-Type is text/xml.

The Content-Length must be specified and must be correct.

Tag	Type	Example
< i4 > or <int>	four-byte signed integer	-12
< boolean >	0 (false) or 1 (true)	1
< string >	ASCII string	hello world
< double >	double-precision signed floating point number	-12.214
< dateTime.iso8601 >	date/time	19980717T14:08:55
<base64>	base64-encoded binary	eW91IGNhbid0IHJlYWQgdGhpcyE=

Table A.1: XML-RPC Supported Data Types

A.4 Payload format

The payload is in XML, a single <methodCall> structure.

The <methodCall> must contain a <methodName> sub-item, a string, containing the name of the method to be called. The string may only contain identifier characters, upper and lower-case A-Z, the numeric characters, 0-9, underscore, dot, colon and slash. It's entirely up to the server to decide how to interpret the characters in a methodName.

For example, the methodName could be the name of a file containing a script that executes on an incoming request. It could be the name of a cell in a database table. Or it could be a path to a file contained within a hierarchy of folders and files.

If the procedure call has parameters, the <methodCall> must contain a <params> sub-item. The <params> sub-item can contain any number of <param>s, each of which has a <value>.

A.5 Scalar <value>s

<value>s can be scalars, type is indicated by nesting the value inside one of the tags listed in table A.1:

If no type is indicated, the type is string.

A.6 <struct>s

A value can also be of type <struct>.

A <struct> contains <member>s and each <member> contains a <name> and a <value>.

Here's an example of a two-element <struct>:

```
<struct>
  <member>
    <name>lowerBound</name>
    <value><i4>18</i4></value>
  </member>
  <member>
    <name>upperBound</name>
    <value><i4>139</i4></value>
  </member>
</struct>
```

<struct>s can be recursive, any <value> may contain a <struct> or any other type, including an <array>, described below.

A.7 <array>s

A value can also be of type <array>.

An `<array>` contains a single `<data>` element, which can contain any number of `<value>`s.

Here's an example of a four-element array:

```
<array>
  <data>
    <value><i4>12</i4></value>
    <value><string>Egypt</string></value>
    <value><boolean>0</boolean></value>
    <value><i4>-31</i4></value>
  </data>
</array>
```

`<array>` elements do not have names.

You can mix types as the example above illustrates.

`<arrays>`s can be recursive, any value may contain an `<array>` or any other type, including a `<struct>`, described above.

A.8 Response example

Here's an example of a response to an XML-RPC request:

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT
```



```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><string>South Dakota</string></value>
    </param>
  </params>
</methodResponse>
```

A.9 Response format

Unless there's a lower-level error, always return 200 OK.

The Content-Type is text/xml. Content-Length must be present and correct.

The body of the response is a single XML structure, a `<methodResponse>`, which can contain a single `<params>` which contains a single `<param>` which contains a single `<value>`.

The `<methodResponse>` could also contain a `<fault>` which contains a `<value>` which is a `<struct>` containing two elements, one named `<faultCode>`, an `<int>` and one named `<faultString>`, a `<string>`.

A `<methodResponse>` can not contain both a `<fault>` and a `<params>`.

A.10 Fault example

```
HTTP/1.1 200 OK
Connection: close
```

```
Content-Length: 426
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:02 GMT
Server: UserLand Frontier/5.1.2-WinNT
```

```
<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>faultCode</name>
          <value><int>4</int></value>
        </member>
        <member>
          <name>faultString</name>
          <value><string>Too many parameters.</string></value>
        </member>
      </struct>
    </value>
  </fault>
</methodResponse>
```

A.11 Strategies/Goals

Firewalls. The goal of this protocol is to lay a compatible foundation across different environments, no new power is provided beyond the capabilities of the CGI interface. Firewall software can watch for POSTs whose Content-Type is text/xml.

Discoverability. We wanted a clean, extensible format that's very simple. It should be possible for an HTML coder to be able to look at a file containing an XML-RPC procedure call, understand what it's doing, and be able to modify it and have it work on the first

or second try.

Easy to implement. We also wanted it to be an easy to implement protocol that could quickly be adapted to run in other environments or on other operating systems.

A.12 FAQ

The following questions came up on the UserLand discussion group¹ as XML-RPC was being implemented in Python.

- The Response Format section says "The body of the response is a single XML structure, a `<methodResponse>`, which *can* contain a single `<params>`..." This is confusing. Can we leave out the `<params>`?

No you cannot leave it out if the procedure executed successfully. There are only two options, either a response contains a `<params>` structure or it contains a `<fault>` structure. That's why we used the word "can" in that sentence.

- Is "boolean" a distinct data type, or can boolean values be interchanged with integers (e.g. zero=false, non-zero=true)?

Yes, boolean is a distinct data type. Some languages/environments allow for an easy coercion from zero to false and one to true, but if you mean true, send a boolean type with the value true, so your intent can't possibly be misunderstood.

- What is the legal syntax (and range) for integers? How to deal with leading zeros? Is a leading plus sign allowed? How to deal with whitespace?

An integer is a 32-bit signed number. You can include a plus or minus at the beginning of a string of numeric characters. Leading zeros are collapsed. Whitespace is not permitted. Just numeric characters preceded by a plus or minus.

¹See URL <http://discuss.userland.com/>

- What is the legal syntax (and range) for floating point values (doubles)? How is the exponent represented? How to deal with whitespace? Can infinity and "not a number" be represented?

There is no representation for infinity or negative infinity or "not a number". At this time, only decimal point notation is allowed, a plus or a minus, followed by any number of numeric characters, followed by a period and any number of numeric characters. Whitespace is not allowed. The range of allowable values is implementation-dependent, is not specified.

- What characters are allowed in strings? Non-printable characters? Null characters? Can a "string" be used to hold an arbitrary chunk of binary data?

Any characters are allowed in a string except `<` and `&`, which are encoded as `<` and `&`. A string can be used to encode binary data.

- Does the "struct" element keep the order of keys. Or in other words, is the struct "foo=1, bar=2" equivalent to "bar=2, foo=1" or not?

The struct element does not preserve the order of the keys. The two structs are equivalent.

- Can the `<fault>` struct contain other members than `<faultCode>` and `<faultString>`? Is there a global list of faultCodes? (so they can be mapped to distinct exceptions for languages like Python and Java)?

A `<fault>` struct **may not** contain members other than those specified. This is true for all other structures. We believe the specification is flexible enough so that all reasonable data-transfer needs can be accommodated within the specified structures. If you believe strongly that this is not true, please post a message on the discussion group.

There is no global list of fault codes. It is up to the server implementer, or higher-level standards to specify fault codes.

- What timezone should be assumed for the `dateTime.iso8601` type? UTC? localtime? Don't assume a timezone. It should be specified by the server in its documentation what assumptions it makes about timezones.

A.13 Additions

- <base64> type.

Copyright and disclaimer

©Copyright 1998-99 UserLand Software. All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and these paragraphs are included on all such copies and derivative works.

This document may not be modified in any way, such as by removing the copyright notice or references to UserLand or other organizations. Further, while these copyright restrictions apply to the written XML-RPC specification, no claim of ownership is made by UserLand to the protocol it describes. Any party may, for commercial or non-commercial purposes, implement this protocol without royalty or license fee to UserLand. The limited permissions granted herein are perpetual and will not be revoked by UserLand or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and USERLAND DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Appendix B

PREDICT's Socket Commands

This information in this appendix is mainly taken from the documentation distributed with the PREDICT software, but has been modified to reflect changes made to the original source code [20].

The network sockets feature of PREDICT allows the program to operate as a server providing tracking data to client applications using the UDP protocol. It is even possible to have the PREDICT server and client applications running on separate machines provided the clients are connected to the server through a network.

B.1 System Configuration

For the socket-based server features of PREDICT to function, the following line needs to be added to the end your `/etc/services` file:

```
predict 1210/udp
```

The port number (1210) can be changed if desired. There is no need to recompile the program if it is changed.

B.2 Program Operations

Start PREDICT with the `-s` switch (`predict -s`) to start the program as a socket-based server. The program will start and automatically go into the multi-satellite tracking mode. Clients may poll PREDICT for tracking data when the program is running in either the multi-satellite or single-satellite tracking mode. When in multi-tracking mode, tracking data for any of the 24 satellites in the program's database may be accessed by client programs. When in single-tracking mode, only live tracking data for the single satellite being tracked may be accessed. Either tracking mode may be ended at any time. When this is done, the socket code will return the last calculated satellite tracking data until the program is again put into a real-time tracking mode. This allows the user to return to the main menu, and use other features of the program without sending potentially harmful data to client programs.

B.3 Client Program Interface

In operation, a character array is filled with the command and arguments to be sent to PREDICT. A socket connection is then opened, the request is sent, a response is received, and the socket connection is closed. The command and arguments are in ASCII text format.

B.4 PREDICT Socket Command Summary

The following are the socket commands interpreted by PREDICT when the program is running in either the single satellite or multi-satellite tracking mode:

B.4.1 Command: GET_WITH

Argument: satellite name or object number, ground station lat, ground station long, ground station elevation, time

Purpose: To poll PREDICT for tracking data for specified location and time.

Return value: Newline ('\n') delimited string of tracking data.

Example: GET_SAT SUNSAT -32.0 18.2 200 975795600

Data returned:

```
SUNSAT
52.66
+57.28
31.51
+5.93
975795873
6063.75
2721.52
795.31
26757.69
9339
D
```

Description: The values are identified by the order in which they are returned. Referring to the example above,

Name: SUNSAT

Long: 52.66 (degrees West)

Lat: 57.28 (degrees North)

Az: 31.51 (degrees)

El: +5.93 (degrees)

Next AOS/LOS: 975795873 (seconds since 01-Jan-1970) = Sat Dec 2 22:24:33 2000

Footprint: 6063.75 (kilometers)

Range: 2721.52 (kilometers)

Altitude: 795.31 (kilometers)

Velocity: 26757.69 (kilometers/hour)

Orbit Number: 9339 (revolutions)

Visibility: D (Currently in Daylight)

If the satellite is in either a geostationary orbit or an orbit that does not permit AOS to occur at the ground station, a zero (0) is returned for the next AOS/LOS time. Otherwise, the next AOS time is provided for satellites not currently in range of the ground station. If the satellite is in range, then the LOS time is provided.

The name provided as an argument to GET_WITH must match the full length name contained in PREDICT's orbital database, and may contain spaces. The command string passed to PREDICT must end with an end of line ('\n') character. The satellite's object number may be used in lieu of the satellite name.

The visibility codes returned are the same as those displayed in PREDICT's multi-satellite tracking mode. An 'N' indicates the satellite is not in sunlight, nor is it optically visible at the ground station. A 'D' indicates that the satellite is in sunlight, but not optically visible at the ground station. A 'V' indicates the satellite is in sunlight, while the ground station is in darkness, meaning the satellite may be optically visible at the ground station.

B.4.2 Command: GET_SAT

Argument: satellite name or object number

Purpose: To poll PREDICT for live tracking data.

Return value: Newline ('\n') delimited string of tracking data.

Example: GET_SAT SUNSAT

Data returned:

SUNSAT

52.66

+57.28

31.51

+5.93

APPENDIX B. PREDICT'S SOCKET COMMANDS

84

975795873
6063.75
2721.52
795.31
26757.69
9339
D

Description: The values are identified by the order in which they are returned. Referring to the example above,

Name: SUNSAT
Long: 52.66 (degrees West)
Lat: 57.28 (degrees North)
Az: 31.51 (degrees)
El: +5.93 (degrees)
Next AOS/LOS: 975795873 (seconds since 01-Jan-1970) = Sat Dec 2 22:24:33 2000
Footprint: 6063.75 (kilometers)
Range: 2721.52 (kilometers)
Altitude: 795.31 (kilometers)
Velocity: 26757.69 (kilometers/hour)
Orbit Number: 9339 (revolutions)
Visibility: D (Currently in Daylight)

If the satellite is in either a geostationary orbit or an orbit that does not permit AOS to occur at the ground station, a zero (0) is returned for the next AOS/LOS time. Otherwise, the next AOS time is provided for satellites not currently in range of the ground station. If the satellite is in range, then the LOS time is provided.

The name provided as an argument to GET_SAT must match the full length name contained in PREDICT's orbital database, and may contain spaces. The command string passed to PREDICT must end with an end of line ('\n') character. The satellite's object number may be used in lieu of the satellite name.

The visibility codes returned are the same as those displayed in PREDICT's multi-satellite tracking mode. An 'N' indicates the satellite is not in sunlight, nor is it optically visible at the ground station. A 'D' indicates that the satellite is in sunlight, but not optically visible at the ground station. A 'V' indicates the satellite is in sunlight, while the ground station is in darkness, meaning the satellite may be optically visible at the ground station.

B.4.3 Command: GET_DOPPLER

Argument: satellite name or object number

Purpose: To poll PREDICT for normalised Doppler shift information.

Return value: satname '\n' Doppler shift information.

Example: GET_DOPPLER OSCAR-27

Data returned:

961.742249

Description: The Doppler shift returned by PREDICT is a normalised to a 100 MHz downlink from the satellite, and must be scaled by the client to the operating frequency of interest. For example, to determine the amount of Doppler shift experienced on a 435 MHz downlink, simply multiply the value returned by 4.35. To calculate the Doppler shift on a 146 MHz uplink, multiply the amount by -1.46. NOTE!! The GET_DOPPLER command no longer echoes back the name of the satellite for which Doppler shift information was requested.

B.4.4 Command: GET_SUN

Argument: none

Purpose: To poll PREDICT for the Sun's current position.

Return value: The Sun's current azimuth and elevation headings.

Example: GET_SUN

Data returned:

299.58

-54.09

Description: Azimuth is returned first, followed by elevation.

B.4.5 Command: GET_MOON

Argument: none

Purpose: To poll PREDICT for the Moon's current position.

Return value: The Moon's current azimuth and elevation headings.

Example: GET_MOON

Data returned:

148.72

+55.83

Description: Azimuth is returned first, followed by elevation.

B.4.6 Command: GET_LIST

Argument: none

Purpose: To poll PREDICT for the satellite names in the current database.

Return value: String containing all satellite names in PREDICT's database.

Example: GET_LIST

Data returned:

OSCAR-10

OSCAR-11

OSCAR-14

PACSAT

WEBERSAT

LUSAT
OSCAR-20
OSCAR-22
OSCAR-23
OSCAR-25
ITAMSAT
OSCAR-27
OSCAR-29
OSCAR-36
OSCAR-40
TECHSAT
TMSAT
RS-12/13
RS-15
SUNSAT
MIR
UARS
HUBBLE
ISS

Description: Names are returned as a string that must be parsed by the client to pull out individual names. NOTE!!! Versions of PREDICT prior to 2.1.3 returned ONLY ONE name at a time, and had to be invoked 24 times to download the entire list. This has since changed! Since satellite names returned by PREDICT are no longer abbreviated (as they were in earlier versions), a 625 byte buffer is now required to store the results of this command.

B.4.7 Command: RELOAD_TLE

Argument: none

Purpose: To force a re-read of PREDICT's orbital database file.

Return value: NULL

Example: RELOAD_TLE

Data returned:

Description: Forces PREDICT to re-read the orbital database file. Useful after the database has been updated by something other than the running version of the program (i.e. PREDICT -u filename), and eliminates the need to re-start PREDICT under these conditions to force a re-read of the database.

B.4.8 Command: GET_VERSION

Argument: none

Purpose: To determine what version of PREDICT is running as a server.

Return value: String containing the version number.

Example: GET_VERSION

Data returned: 2.1.3\n

Description: Allows clients to determine what version of PREDICT they're talking to.

B.4.9 Command: GET_QTH

Argument: none

Purpose: To determine the ground station location (QTH) information.

Return value: String containing the info stored in the user's predict.qth file.

Example: GET_QTH

Data returned:

W1AW

41.716905

72.727083

25

Description: The ground station callsign, latitude, longitude, and altitude above sea level

are returned. Useful for plotting the user's location on a map.

B.4.10 Command: GET_TLE

Argument: satellite name or catalog number

Purpose: To read the Keplerian elements for a particular satellite.

Return value: String containing Keplerian orbital data.

Example: GET_TLE OSCAR-25

Data returned:

```
OSCAR-25
22828
00 306.69662848
98.3880
359.3791
0.0009012
197.6165
162.4707
14.28826398
6.76e-06
33830
```

Description: The satellite name, object number, reference epoch, inclination, right ascension at ascending node, eccentricity, argument of perigee, mean anomaly, mean motion, decay rate, and orbit number for the satellite in question are returned.

B.4.11 Command: GET_TIME

Argument: none

Purpose: To read the system date/time from the PREDICT server.

Return value: Number of seconds since midnight UTC on January 1, 1970.

Example: GET_TIME

Data returned:

977533528

Description: Unix Time is returned by the server. This command allows clients to display clock/calendar information or sync their system clocks with that of the server.

B.4.12 Command: GET_TIME\$

Argument: none

Purpose: To read the system date/time from the PREDICT server.

Return value: UTC Date/Time as an ASCII string.

Example: GET_TIME\$

Data returned:

Sat Dec 23 01:05:28 2000

Description: Returns an ASCII representation of the current date/time in UTC. Useful for displaying the current date/time in client applications if the local system clock cannot be synced using the GET_TIME command.

Bibliography

- [1] B. van der Merwe, "Micro-satellite data handling: a unified information model," masters thesis, University of Stellenbosch, Department of Electric and Electronic Engineering, March 2001.
- [2] Ely, Neal, and T. P. O'Brien, "Space logistics and reliability," in *Space Mission Analysis and Design* (J. R. Wertz and W. J. Larson, eds.), (London), pp. 633–656, Kluwer Academic Publishers, 1991.
- [3] D. Boland, W. Steger, D. Weidow, and L. Yakstis, "How emerging technologies are changing the rules of spacecraft ground support," in *Proceedings of the 4th International Symposium on Space Mission Operations and Ground Data Systems*, (Munich, Germany), September 16-20, 1996.
- [4] Boehm, Barry, B. Clarke, E. Horowitz, C. Westland, R. Madachy, and R. Selby, "Cocomo 2.0 software estimation model," tech. rep., USC Center for Software Engineering, <http://sunset.usc.edu/research/COCOMOII/index.html>, 1995.
- [5] A. H. Maury, A. Critchfield, J. Langston, and C. K. Adams, "Jswitch/jsat: Real-time and offline world wide web interface," in *Proceedings of the 12th Annual AIAA/USU Conference on Small Satellites*, (Utah State University, Logan, Utah, U.S.A.), 1998.
- [6] A. Baldi, E. Perdrix, and S. T. Smith, "The role of centralization in a distributed architecture: The scos ii experience," in *Proceedings of the 4th International Symposium on Space Mission Operations and Ground Data Systems*, (Munich, Germany), September 16-20, 1996.
- [7] M. Jones, B. Melton, and M. Bandecchi, "Teamsat's low-cost egse and mission control systems," *ESA Bullitin*, August 1998.

- [8] T. Bray, J. Paoli, C. Sperberg-McQueen, and E. Maler, "Extensible markup language (xml) 1.0," recommendation, W3C, October 2000.
- [9] J. R. Borck, "Future of networked apps." http://www.byte.com/features/1999/06/0607XML_RF July 2001.
- [10] D. Winer, "Soap frequently asked questions." <http://soap.weblogs.com/faq>, April 2000.
- [11] M. Kirtland, "The programmable web: Web services provides building blocks for the microsoft .net framework." <http://msdn.microsoft.com/msdnmag/issues/0900/WebPlatform/WebPlatform.asp>, January 2001.
- [12] J. Udell, "The scoop." www.webbuildermag.com, May 28 1999.
- [13] B. McLaughlin and J. Hunter, "Jdom." <http://www.jdom.org/>, 2001.
- [14] B. McLaughlin, *JAVA and XML*. 101 Morris Street, Sebastopol, CA 95472: O'Reilly and Associates, Inc., 1st ed., June 2000.
- [15] W. Tai and D. Sweetnam, "A mission operations architecture for the 21st century," in *Proceedings of the 4th International Symposium on Space Mission Operations and Ground Data Systems*, (Munich, Germany), September 16-20, 1996.
- [16] H. Wallnofer, "Xmlrpc-java." <http://helma.at/hannes/xmlrpc>, 2001.
- [17] G. Champine, *Distributed Computer Systems - Impact on management, design and analysis*. 52 Vanderbilt Avenue, New York, N.Y., 10017, U.S.A.: North-Holland Publishing Company, 1980.
- [18] J. Udell, "Byte.com feature - exploring xml-rpc." http://www.byte.com/features/1999/06/0607XML_RPC5.html, June 1999.
- [19] D. Winer, "Xml-rpc specification." <http://www.xmlrpc.org/spec>, October 1999.
- [20] J. A. Magliacane, "Predict - a satellite tracking/orbital prediction program." <http://www.qsl.net/kd2bd/predict.html>, 2000.
- [21] E. M. Hugo, "class geodesytransforms." ESL, University of Stellenbosch, 2001.

- [22] CCSDS, "Recommendation for packet telemetry," recommendation, Consultative Committee for Space Data Systems Secretariat, 2000.
- [23] P. Viallefont, "Distributed middleware for future ground segment architectures," in *Proceedings of the 4th International Symposium on Space Mission Operations and Ground Data Systems*, (Munich, Germany), September 16-20, 1996.
- [24] C. A. Kitts, "A global spacecraft control network for spacecraft autonomy research," in *Proceedings of the 4th International Symposium on Space Mission Operations and Ground Data Systems*, (Munich, Germany), September 16-20, 1996.
- [25] P. Holding, "Mission control systems for simple satellites," in *Proceedings of the 4th International Symposium on Space Mission Operations and Ground Data Systems*, (Munich, Germany), September 16-20, 1996.
- [26] H.-J. Fischer and E. Rabenau, "Potentials in increasing efficiency of small satellite ground operations - an analysis of existing methods and concepts," in *Proceedings of the 9th Annual AIAA/USU Conference on Small Satellites*, (Utah State University, Logan, Utah, U.S.A.), September 18-21, 1995.
- [27] C. Larman, *Applying UML and Patterns*. Upper Saddle River, New Jersey 07458: Prentice Hall PTR, 1997.
- [28] C. Kitts, R. Twiggs, F. Pranajaya, J. Townsend, and B. Palmintier, "Experiments in distributed microsatellite space systems," in *Proceedings of the AIAA Space Technology Conference and Exhibit*, (Albuquerque, NM, USA), September 1999.
- [29] K. Rajan, M. Shirley, W. Taylor, and B. Kaneksky, "Ground tools for autonomy in the 21st century," in *Proceedings of the IEEE AeroSpace Conference*, (Big Sky, MT, USA), March 2000.