# Development and Evaluation of the Ethernet Interface(s) for the Monitoring and Control System of a New Beamforming Radio Telescope

Abirami Srinivasan

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

Steve W. Ellingson, Chair

Claudio da Silva

Yaling Yang

August 6, 2010

Blacksburg, Virginia

Keywords: Long Wavelength Array, Monitoring and Control System

# Development and Evaluation of the Ethernet Interface(s) for the Monitoring and Control System of a New Beamforming Radio Telescope

Abirami Srinivasan

## ABSTRACT

The Long Wavelength Array (LWA) is a large multi-purpose radio telescope, operating in frequencies between 10 and 88 MHz, designed for both long-wavelength astrophysics and ionospheric science. The LWA will eventually consist of 53 "stations", each consisting of 256 pairs of crossed-dipole antennas whose signals are formed into beams. The Monitoring and Control System (MCS), a subsystem of each LWA station, controls the station's subsystems and also monitors their status. This thesis addresses the interface-related features of MCS. The physical interface of the MCS with each subsystem is a Gigabit Ethernet connection and the interface protocol is User Datagram Protocol (UDP). An analysis of the throughput obtained through the interface using UDP is compared to that achieved using Transmission Control Protocol (TCP). It is seen that the throughput with UDP is 15% better than with TCP, and that UDP is a better choice for the given requirements. Implementation of a new ionospheric calibration scheme requires that the MCS be capable of repointing between astronomical sources on a 5 ms time scale. The rate at which beams can be repointed is analyzed. It is confirmed that MCS is at least 2 orders of magnitude faster than necessary, and is limited by the ethernet network throughput. Python software that facilitates the development and testing of MCS and other subsystems have been developed, and are described.

# Acknowledgments

I thank my advisor, Dr. Steve Ellingson for his patience and guidance without which, this thesis would not have materialized. I am grateful to Dr. Claudio da Silva and Dr. Yaling Yang for having agreed to be a part of my advisory committee and taking the time to review this thesis.

I am very thankful to my colleague, Christopher Wolfe, for his valuable suggestions and inputs with the research from time to time. I also thank my other colleagues - Kshitija Deshpande and Mahmud Harun for offering help and advice. I am extremely indebted to Dr. Shajedul Hasan, Nancy Goad, and Cindy Hopkins for all their kind words and their time whenever I wanted their advice.

I am very thankful for having had the most understanding roommates - Sushrutha Vigraham and Pavithra Sekhar, during the course of my stay here at Virginia Tech. I could not have asked for more unconditional support from them. Thanks to my brother here at Tech - Karthik Ganesan, for always being there to support and encourage me. I have to make a mention of those special times which gave me a much welcomed respite from all worries and concerns - Thank you Sujit, Tracy, Santosh, Sai, Rini, Bijoy and Nitya for the wonderful Friday evenings! Special thanks to Poornima, Puppy, Tree, Chandy, Sriram, and State for being there for me and giving me all the help and support that I could not have imagined was possible, that too remotely.

I am blessed to have a wonderful family; my parents and my brother, Sivaram have been amazing and I could not have reached this far without them. My father has always been my

inspiration and his unfathomable trust in my abilities has been a driving force at every stage in my academic life. I also thank my uncle, Dr. Durai, and my cousin, Shankar for all their prayers and concern during the most stressful times. Ravi, you were my strength during the last two years and I could not thank you enough for all the love, support and patience when it mattered the most.

I dedicate this thesis to my entire support-system and also to Abina, who I wish were with me to celebrate this moment.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The Long Wavelength Array (LWA) is a new multipurpose radio telescope operating in the frequency range 10–88 MHz.[1] A brief description of LWA is given in Section 1.1. The LWA will eventually consist of 53 electronically-steered phased-array "stations" distributed over a region of 400 km in diameter in the state of New Mexico. The LWA station architecture is described in Section 1.2. In this thesis, we address the Monitoring and Control System (MCS), a subsystem of the LWA. The station MCS is essentially the set of computers which control the station, and provides status information. It is briefly described in Section 1.3. The physical interface between MCS and other station subsystems is full-duplex Gigabit Ethernet. The objective of this thesis is to analyze certain features of the MCS design and evaluate its ethernet links. The contributions of this thesis are summarized in Section 1.4. The organization of the remainder of this thesis is described in Section 1.5.

## 1.1 The Long Wavelength Array

LWA is a large radio telescope that is being developed to probe into questions emerging in low frequency radio astronomy. A detailed description is provided by Ellingson *et al.* [1], here is only a brief overview to facilitate discussion of the MCS.

[1] http://lwa.unm.edu

An LWA station consists of 256 pairs of crossed-dipole antennas. The first LWA station, "LWA-1", is shown in Figure 1.1. A simplified block diagram of the signal flow in an LWA station is shown in Figure 1.2. Each antenna outputs a balanced signal from each of the two polarizations. The signal output from the antenna is amplified and filtered to the frequency range of interest. The signal is then digitized at 196 million samples per second (MSPS). The "Digital Processor" (DP) handles the task of beamforming. The method of beamforming consists of first applying integer-sample-period delays using a first-in first-out (FIFO) buffer (also called "coarse delay"), followed by a configurable finite impulse response (FIR) filter for each antenna to implement subsample delay (also called "fine delay"), followed by a matrix multiplication for polarization adjustments, and then summing the results across antennas to form the beam. This can be interpreted as a delay-and-sum beamformer, where the FIR filter is used to implement an approximately frequency-independent, continuously-variable delay. DP has the ability to form multiple beams, each of which can be pointed and tuned independently of the others. Additional information on the DP subsystem can be found in [2]. The outputs from the DP are routed directly to data recorders, where the output is consolidated. This is only a high-level description of the signal flow; a detailed discussion of the station architecture follows in the next subsection.

## 1.2 LWA Station Architecture

The formal architecture of an LWA station is specified in the LWA Station Architecture document [3] and is summarized by Figures 1.3 and 1.4. Within an LWA station, the subsystems Array (ARR), Analog Signal Processor (ASP), Digital Processor (DP), MCS, and Shelter (SHL) are referred to as "level-1" subsystems, for which there is one per station and which, taken together, comprise the entire station. Subsystems that are subordinate to the level-1 subsystems, such as the DP's Digital Receiver (DRX), ARR's Antenna (ANT), etc, are referred to as "level-2" subsystems. The purpose of MCS in this architecture is to control the station and provide

Figure 1.1: Aerial view of LWA-1.

status information. The MCS monitors and controls the level-1 subsystems through computers that are part of the subsystems. For example, both the ASP and DP have distinct monitoring and control systems, called ASP-MCS and DP-MCS, respectively.

The different subsystems in the LWA include:

- ARR: The Array subsystem includes antennas, ground screen, and "front end electronics" (FEE) collocated with antennas. A single dual-polarization antenna unit and associated FEE are tightly integrated and are therefore collectively identified as a "stand" (STD) subsystem.

- ASP: In the analog signal path, the signal output from the antenna is sent to the ASP, which is used to adjust the gains of each antenna signal, and provide adequate filtering

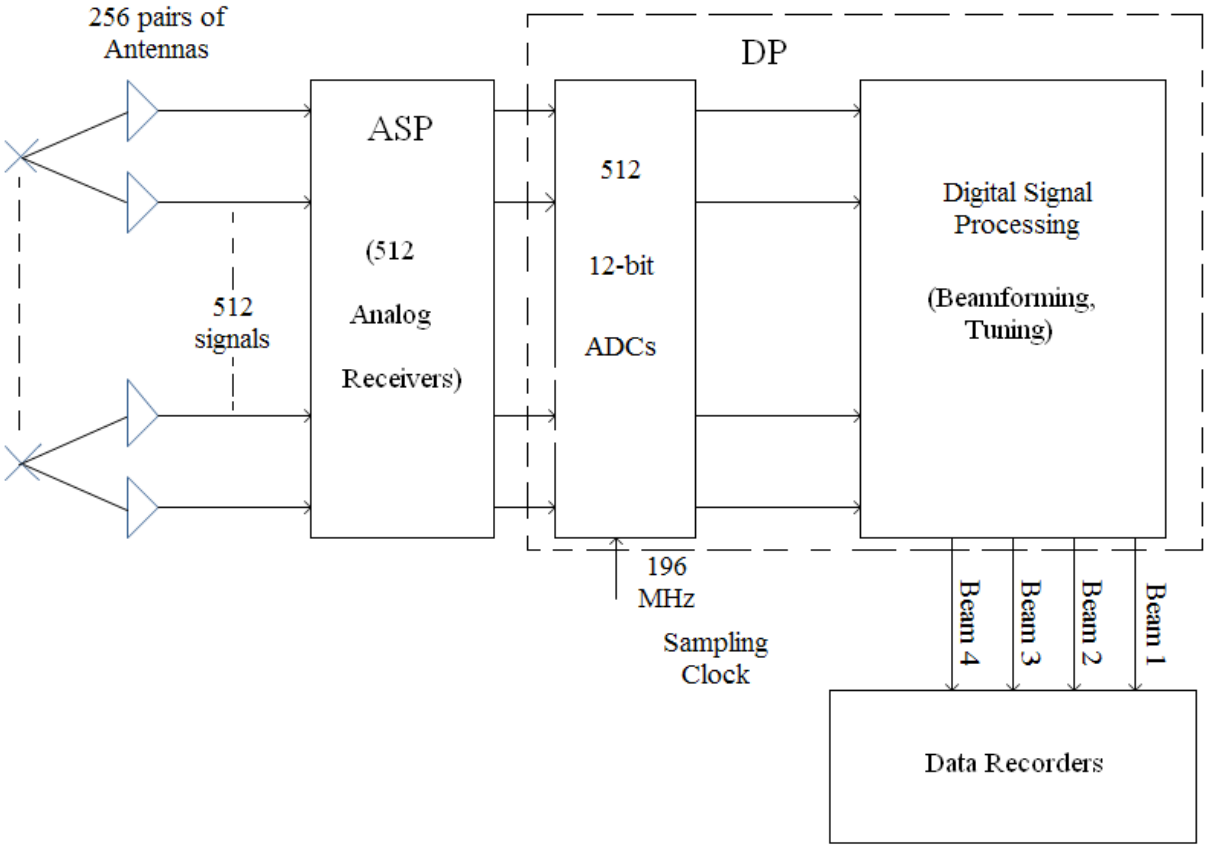


Figure 1.2: Simplified block diagram of an LWA station showing signal flow.

of the RF signal prior to digitization. The MCS controls the gains of each of the signals, the power settings for each polarization of each stand and the "Analog Receiver" (ARX) by communicating with the MCS-ASP.

- SHL: The SHL system includes the shelter itself, the "Shelter Entry Panel" (SEP), the Power Conditioning and Distribution (PCD), and the Environmental Control System (ECS). The SHL PCD and ECS are controlled by MCS. MCS "initializes" the SHL, sets the internal temperature of SHL and also controls its power ports.

- DP: In the DP, each signal is digitized by an "analog-digital" (A/D) converter and the digitized polarization pair is distributed to four "beamforming units" (BFU), the "narrowband transient buffer" (TBN), and the "wideband transient buffer" (TBW). Each BFU forms a beam in the desired pointing direction using coarse and fine delays as de-

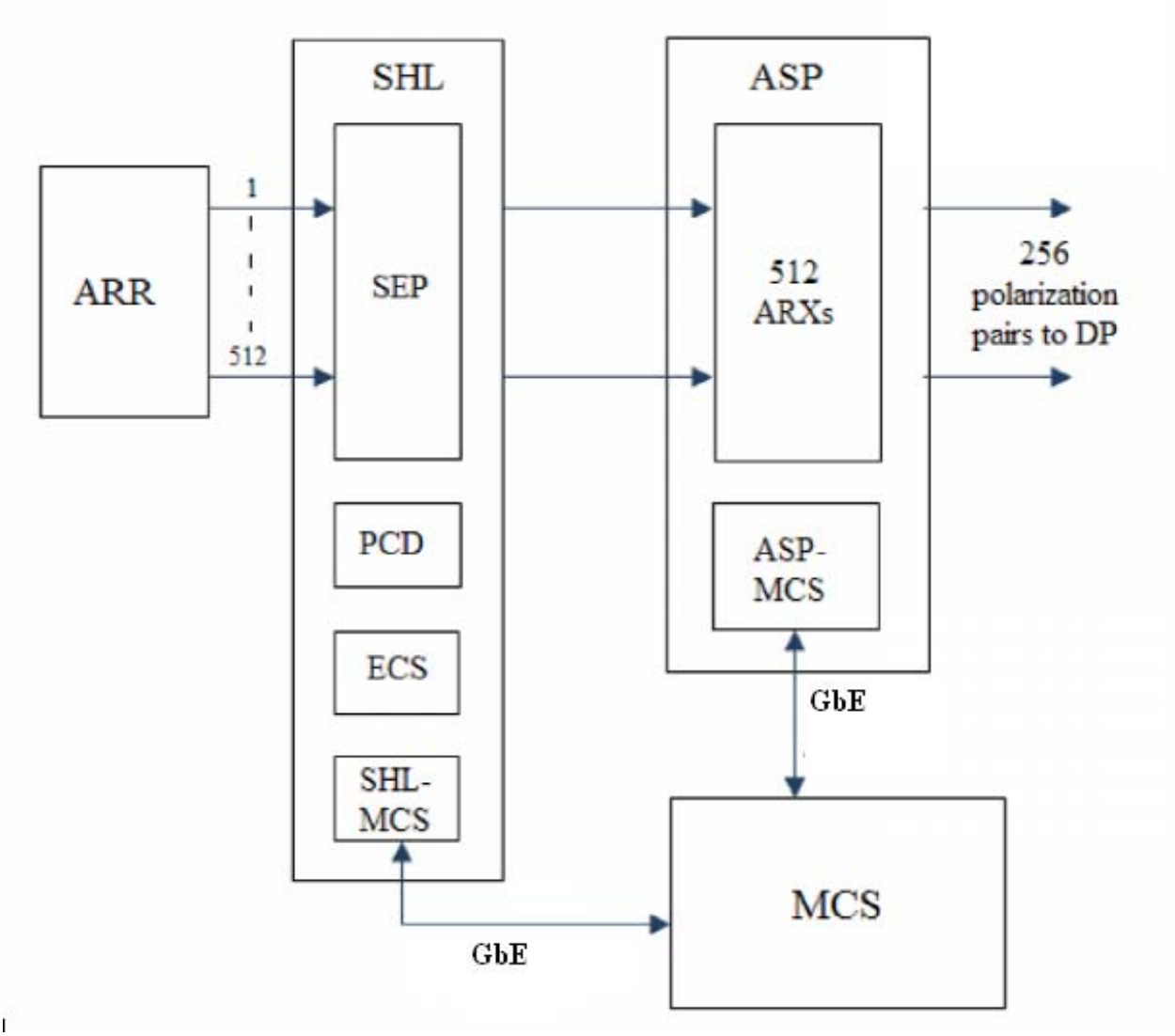


Figure 1.3: Block diagram showing LWA station architecture from antennas through analog receivers. "GbE" stands for "gigabit ethernet".

scribed earlier and outputs data for use by other subsystems using each "digital receiver" (DRX). Each DRX provides two "tunings" (in each tuning, a single spectral swath of certain width is selected from the digital passband, divided into smaller contiguous channels of smaller width, and downsampled accordingly) of the corresponding beam. Also, the transient buffers, TBN and TBW, accept all the outputs from the A/D converters

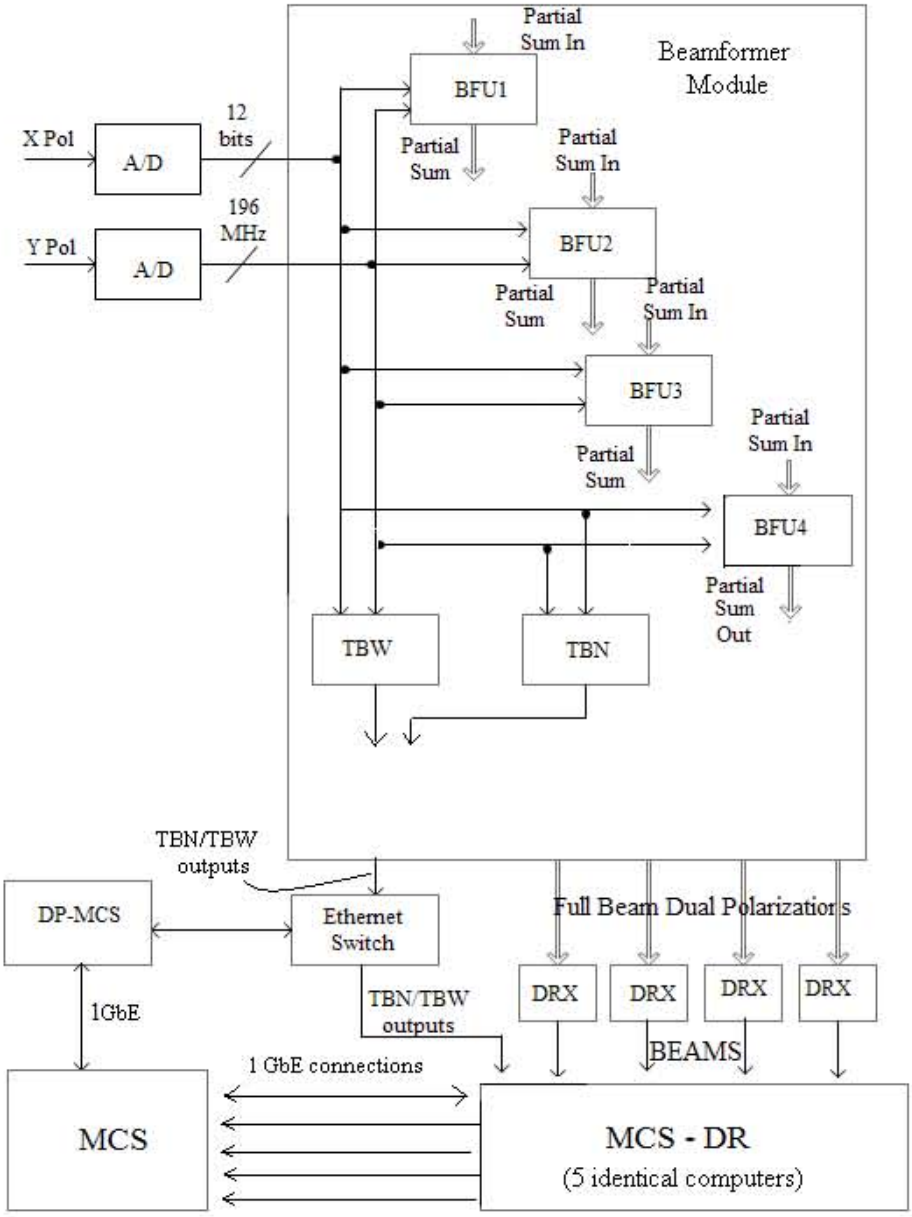


Figure 1.4: Block diagram showing "per-antenna processing" (two polarizations) in DP. The "partial sums" for combining each processed signal into beams are carried from a set of 10 "beam former modules" (a set of 4 BFUs, TBN and TBW) to the next 10, with each set of 10 such modules adding the signals from 10 antenna stands. "GbE" stands for "gigabit ethernet".

and coherently record the data stream in a form suitable for later recovery and analysis. The filter coefficients and other aspects of the configurations of TBN, TBW and DRX are controlled by the DP-MCS which communicates with the Station MCS.

- Data Recorders (MCS-DR): The outputs from DP, that is 4 beams from DRX, plus one output from either TBN or TBW are routed to the MCS-DR for data recording. MCS-DR consists of 5 identical computers. Each MCS-DR computer is connected to a separate, removable data recorder storage unit ("DRSU"). A DRSU is a hard drive array having total storage of 5 TB. All data acquired by an MCS-DR computer is streamed directly to its associated DRSU. Each MCS-DR computer is controlled by MCS as a separate level-1 subsystem. More information on the monitoring and control operations for the MCS-DR can be found in [4].

## 1.3 MCS

In this section, we briefly discuss the functions and architecture of MCS. The functions of MCS can be found in [5] and are summarized below:

- Monitoring: MCS monitors the state of the station and the progress of commanded activities.

- Logging: MCS maintains a record of activity which includes the commands received, the change in the status of observations and state of subsystems such as error conditions reported, actions taken, etc.

- Control: Users interact with the LWA subsystems through the MCS. MCS converts commands issued by users to commands sent to the subsystems. Control function might also occur independent of user commands.

### 1.3.1 MCS Requirements Imposed by Ionospheric Calibration Requirements

Cohen and Paravastu (2008) describe a scheme for using LWA to provide data on the dynamically-varying ionosphere in support of imaging [6]. This scheme drives requirements for beam repointing time. As described in Section 1.2, DP performs beamforming, and MCS controls the configuration of DP required in beamforming. The calibration scheme requires that LWA-1 be capable of switching a beam between widely separated astronomical sources within 5 ms [7]. That is, the time required to repoint a beam, including the time required for any transients (any other overheads that might be associated in the process of repointing), should be less than or equal to 5 ms. This impacts the design of MCS because MCS controls the parameters involved in beamforming. Thus, it is necessary to perform an analysis of the rate of beam repointing that MCS can support to ensure that it conforms with the ionospheric calibration requirement. This is one of the objectives of this thesis and is carried out in Section 3.3.

### 1.3.2 MCS Architecture

The station MCS is essentially the set of computers which control the station, and provides status information. Various subsystems (e.g., ASP, DP, etc.) also have MCSs, which are computers subordinate to the station MCS. The subsystem MCSs are implemented to facilitate modularity in the station design and to facilitate independent development of subsystems. The architecture of the station MCS is specified in [8] and summarized in Figure 1.5.

The "Scheduler" is a computer that issues commands and receives status updates from other LWA subsystems. The communication of these commands and updates takes place through an ethernet switch known as the "Command Hub". The "Executive" is a computer which interprets observation requests and generates data which form the content for the command messages issued by Scheduler. It also receives the contents of response messages via Scheduler. The "Task Processor" is the primary interface with users, managing command line and GUI

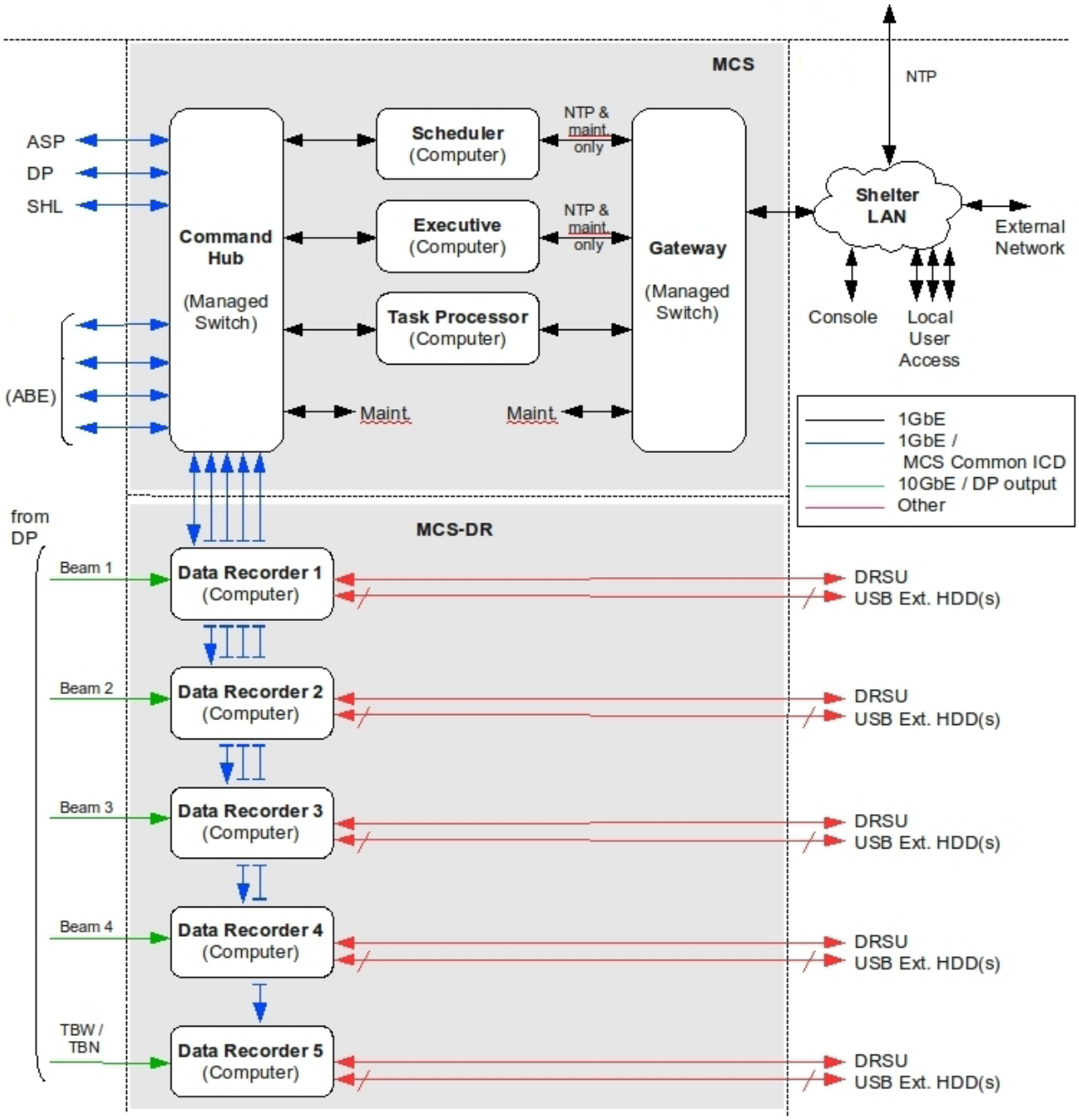


Figure 1.5: MCS architecture and interfaces to subsystems. "Maint" refers to access ports to allow connection of computers for development, integration and troubleshooting activities. From [8].

actions. It is also responsible for the scheduling and interpretation of internal diagnostics, and manages MCS-DR. The "Gateway" is a managed switch which, under the control of the Exec-

utive, regulates the flow of network traffic between the Shelter LAN and the various computers of MCS and MCS-DR.

The Command Hub is a managed switch which, under the control of the Scheduler, regulates the flow of network traffic between the Scheduler and subsystems, utilizing the "MCS Common Interface Control Document (ICD)" [9]. This ICD provides a framework for the interface between MCS and level-1 subsystems which are directly connected to it, including electromechanical interfaces and protocol information. The physical interface with MCS is 1000BASE-T (full-duplex gigabit ethernet) and uses User Datagram Protocol (UDP), with direct passing of messages using sockets [10]. Justifying the choice of UDP as the protocol interface is a topic for analysis and is one of the objectives of this thesis. A summary of the MCS Common ICD appears in Section 3.1.

The capability exists within the station to connect multiple subsystems to the DP beam and TBW/TBN outputs. These "alternative backends" (ABE) are also connected to the MCS to be monitored and controlled by it.

### 1.3.3 MCS Software Architecture

Software has been developed to provide "user-side" interfaces to MCS [11]. A "Tier 0" interface does not need any MCS support; Tier 0 software communicates directly with the subsystem. This is intended primarily for subsystem development and integration. We discuss development of Tier 0 software in Section 4.2. A "Tier 1" interface to subsystems uses MCS Scheduler to communicate with the subsystem [12]. Tiers 2 and higher involve MCS Executive and applications running on MCS Task Processor.

## 1.4 Contributions

The specific contributions of this thesis include the following:

- A study of the performance of TCP as an alternative to UDP for the MCS Common ICD.

Section 3.2 presents the results.

- An analysis of the maximum repointing rate of beams, demonstrating that it meets the requirement of 5 ms as required by the new calibration scheme described in Section 1.3.1. The beam repointing rate is dependent on MCS, as MCS controls DP. The results are presented in Section 3.3.

- Tier 0 Software to facilitate the development and integration of level-1 subsystems to MCS. Software such as subsystem emulation, interface for MCS to communicate with other subsystems, etc were written in Python. Chapter 4 documents the developed software.

## 1.5 Organization of this Thesis

The rest of this thesis is organized as follows: Chapter 2 ("Review of the Ethernet Standard and Networking Protocols") presents a review of the reference network model known as the Open System Interconnection (OSI) model. Chapter 2 also discusses the Ethernet standard, the User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). This forms a background for the analysis of the MCS Common ICD interface that follows in Chapter 3 ("Analysis of Some Interface-Related Features of the LWA Station MCS"). Chapter 3 includes a discussion on the protocol used by the MCS Common ICD and an analysis of the rate at which commands can be sent. Some of the software written to facilitate the testing of the MCS project is documented in Chapter 4 ("Software for Development and Integration of MCS Interfaces"). We summarize our findings in Chapter 5 ("Conclusions").

# Chapter 2

# Review of the Ethernet Standard and Networking Protocols

In this chapter, a brief review of the Ethernet standard and the networking protocols, in particular the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), is presented. This is useful for the discussion that follows in the next chapter, where the performance of the protocols are compared on the basis of their rates of message delivery across the network (also known as throughput). To compare throughput, the packet structures for the protocols must be considered. This chapter provides the necessary information about the packet structures of the different protocols and their overheads.

Data to be transferred across a "connectionless" network (packet-switched network) is divided into pieces called *packets*. Between the source and destination, each of these packets travels through a set of communication links and *packet switches*. Each packet may take a different route. These networks can be described in terms of the OSI Reference Model (discussed in Section 2.1) as a stack of layers or levels, each one built upon the one below it. The purpose of each layer is to offer certain services to the higher layers, shielding those layers from the details of how the offered services are actually implemented. At the transmitting end, each layer passes data and control information to the layer immediately below it, until the lowest layer is

reached. Communication between like layers at source and destination is by using protocols, which provide the rules for communication.

## 2.1 The OSI Reference Model

The International Organization for Standardization (ISO) developed the Open Systems Interconnection (OSI) Reference Model [13]. Figure 2.1 shows the OSI stack for two systems connected across a network. A network "host" is a computer that is connected to a data network. The information that has to be sent from one host to another is called the "message" or packet. At the sending host, the higher-layer message is passed to the lower layer, where additional information (the corresponding layer's header information) is appended. Figure 2.2 shows the different protocols and software (relevant to LWA) operating on the different OSI layers. The OSI reference model has seven layers and they are briefly described below:

- Physical Layer: This layer deals with the physical aspects of the media being used to transmit the data, such as mechanical components and connectors, electrical aspects, modulation and encoding of data bits on carrier signals. Examples of the physical specifications for "Local Area Network (LAN)" systems are Ethernet and IEEE 802.3 [14].

- Data Link Layer: The Data Link Layer provides the functional and procedural means to transfer data between two hosts and to detect and possibly correct errors that may occur in the Physical Layer. It accomplishes this by having the sender break up the input message into units called *data frames* which include Layer 2 header information. The link layer also performs logical link control (LLC) and media access control (MAC). LLC involves the establishment and control of logical links between devices on a network while MAC involves procedures used to control access to the physical layer. Examples of Layer 2 devices are switches, bridges, "network interface cards (NICs)" etc.

- The Network Layer: The Network Layer performs network routing, and might also perform fragmentation and reassembly of Layer 3 packets called *datagrams*, and report

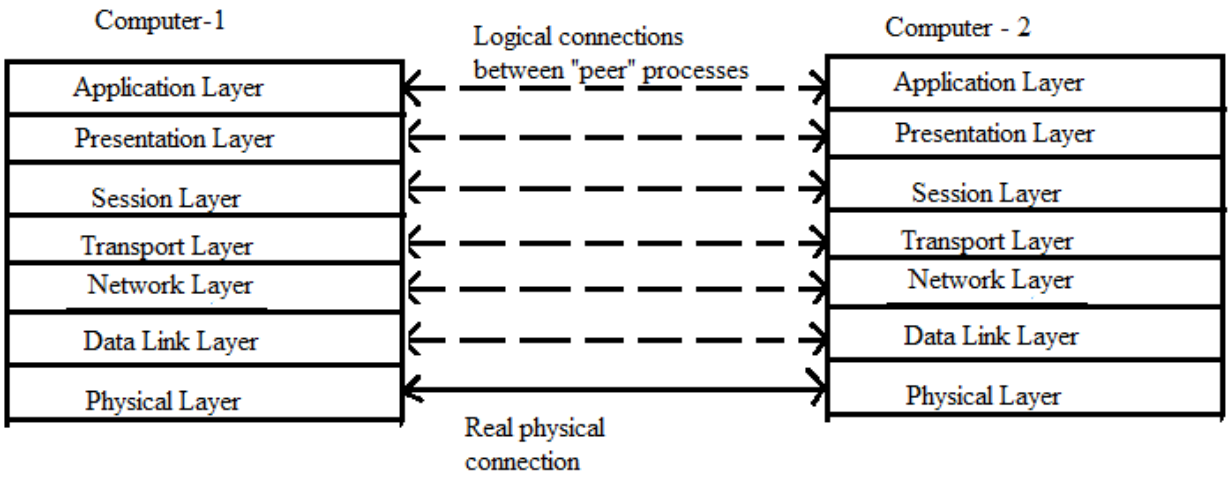


Figure 2.1: OSI Reference Model.

delivery errors. This layer is also responsible for "logical addressing", which is discussed in Section 2.3. "Routers" and Layer 3 switches operate at this layer. An example of a protocol defined at this layer is Internet Protocol (IP) [15].

- The Transport Layer: The Transport Layer converts the packets received from higher layers into "segments" with the addition of a Layer 4 header. This layer is responsible for end-to-end (also called source-to-destination) delivery. The Transport Layer controls the reliability of a given link through flow control, segmentation/desegmentation, and error control. The Transport Layer keeps track of the segments and retransmits those that fail. Protocols that operate on this layer include TCP [16], UDP [17], etc.

- Layers 5-7: The Session Layer, Presentation Layer and Application layer are concerned with establishment of "dialogues" (connections), encryption, compression and conversion of the data transmitted. MCS software (described earlier in Section 1.3.3) runs on the Application Layer through the establishment of "sockets" (described in Section 2.4.3) which run on the Session Layer. In this case, Layer 7 communicates with Layer 5 and Layer 6 services are not required.

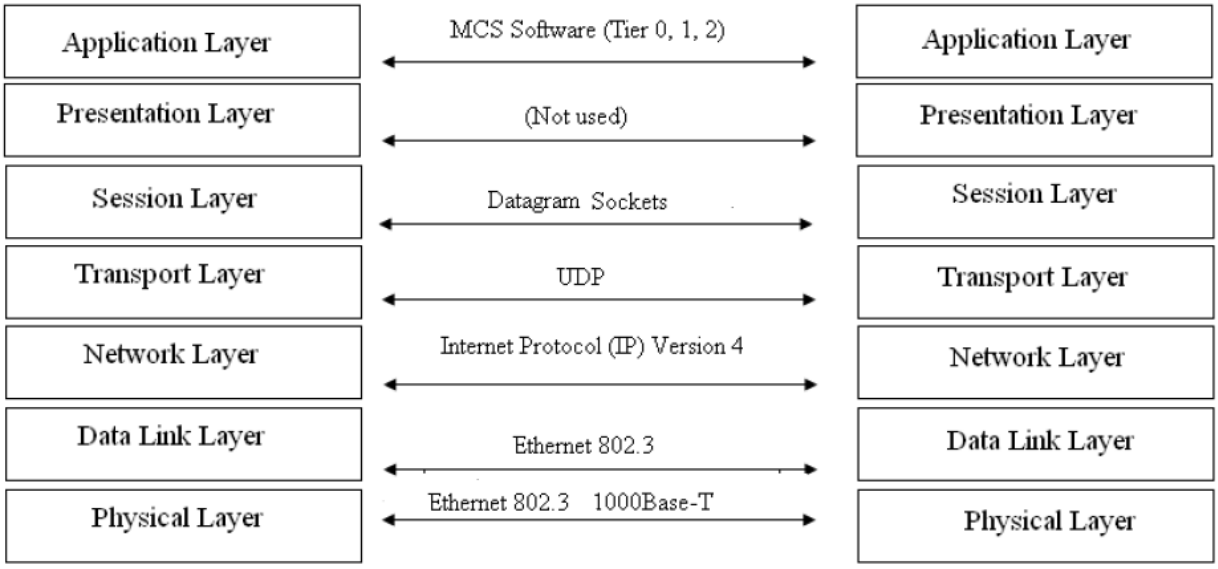


Figure 2.2: Protocols used in the LWA-1 MCS.

## 2.2 Ethernet

Ethernet is a popular packet-switched technology developed in the early 1970s. The term "Ethernet" since 1985 refers to the family of protocols defined by the Institute for Electrical and Electronic Engineers (IEEE) 802.3 standard [14]. One class of Ethernet physical interfaces is known as *twisted pair Ethernet*. A set of four pairs is used to connect each computer to a Ethernet hub or switch. A version known as Fast Ethernet operates at 100 Mbps (million bits per second), and Gigabit Ethernet (GigE) operates at 1000 Mbps. Known formally as 100Base-T and 1000Base-T, these versions use the "Cat-5" (category 5 twisted pair) wiring.

Table 2.1 shows the Ethernet frame structure. An octet (or byte) is a set of 8 bits. Ethernet frames are of variable length, with no frame smaller than 64 octets or larger than 1518 octets. In addition to identifying the source and destination, each frame transmitted contains a preamble field, start-of-frame type field, Cyclic Redundancy Check (CRC) field, and a 12 octet interframe gap. The preamble and the start-of-frame fields consist of 64 bits of alternating *0*s and *1*s to help the receiver synchronize. The 32-bit CRC helps the receiver detect transmission errors; the

| Preamble | Start of frame | Destination Address | Source Address | Frame Type | Frame Data | CRC | Interframe |
|---|---|---|---|---|---|---|---|
| 7 octets | 1 octet | 6 octets | 6 octets | 2 octets | 46 - 1500 octets | 4octets | 12 octets |

Table 2.1: The format of an IEEE 802.3 (Ethernet) frame

sender computes the CRC as a function of the data in the frame, and the receiver recomputes the CRC to verify that the frame has been received intact. The frame type field contains a 16-bit integer that identifies the type of data being carried in the frame.

## 2.3 Internet Protocol (IP)

The IP is a network layer protocol designed for use in interconnected systems of packet-switched computer communication networks. IP facilitates transmission of blocks of data, called *datagrams*, from source to destination, where source and destination are identified by fixed-length addresses.

### 2.3.1 IP Addressing Scheme

An IP address uniquely identifies a host on an IP-based network. An IP address is a 32 bit binary number. This value is interpreted as 4 decimal values, each representing 1 byte in the range 0 to 255, separated by decimal points; this is known as "dotted-quad notation". An example of an IP address is 172.16.81.100. IP-based networks are partitioned into logical segments called "subnets". Devices on a subnet share a contiguous range of IP addresses. The IP address is divisible into a network portion and a host portion with the help of a subnet "mask" [18].

### 2.3.2 The IP Datagram Format

This section describes the structure of an IP datagram. An IP datagram consists of a header part and a text part. The header format is shown in Table 2.2. The header has a 20-byte fixed

| Bits | 0 - 3 | 4 - 7 | 8 - 15 | 16 - 18 | 19 - 31 |
|---|---|---|---|---|---|
| 0 | Vers | Hlen | Service Type | Total Length | |
| 32 | Identification | | | Flags | Fragment Offset |
| 64 | Time to live | | Protocol | Header Checksum | |
| 96 | Source IP Address | | | | |
| 128 | Destination IP Address | | | | |
| 160 | IP Options and Padding (if any) | | | | |
| 192 | DATA | | | | |

Table 2.2: The format of an IP datagram

part (first 12 fields) and an optional variable-length part. The "Vers" (Version) field indicates the version of the protocol for the datagram. Currently the versions being used are versions 4 and 6, referred to as "IPv4" and "IPv6" respectively. A field in the header, "HLen", is provided to indicate how long the header is, as the header length is not constant. The most common header, which contains no options and no padding, measures 20 bytes and has Hlen = 5 (indicating 5 32-bit words). The 8-bit "Service Type" field specifies how the datagram should be handled. The "Total length" field gives the length of the IP datagram measured in octets, including the length of the header and data. The maximum possible size of an IP datagram is $2^{16}$ or 65535 octets, as the "Total length" field is 16 bits long. Three fields in the datagram header, "Identification", "Flags", and "Fragment Offset", control fragmentation and reassembly of datagrams. The "Time to live" field limits a datagram's lifetime. The "Protocol" field specifies the format of the "DATA" field; i.e., the encapsulated protocol. This value indicates the specific transport-layer protocol to which the data portion of the datagram should be passed at the destination. For example, a value of 6 indicates that the data portion is passed to TCP. A list of protocol numbers is defined in [19]. The "Header checksum" ensures integrity of header values. Fields "Source IP" and "Destination IP" addresses contain the 32-bit IP addresses of the datagram's original sender and ultimate intended recipient.

### 2.3.3 Maximum Transmission Unit (MTU)

The Maximum Transmission Unit (MTU) is the largest frame size that can be transmitted over a network. MTU may be fixed by standards (such as Ethernet) or decided at the time a connection between two hosts is made. Nearly all IP over Ethernet implementations allow a MTU of 1500 bytes [20].

While a sender will know the MTU of its interface, it will not initially know the smallest MTU on the series of links to other hosts on the network. To get around this issue, IP allows fragmentation, which refers to dividing the datagram into pieces, each small enough to pass over the single link that is being fragmented for, using the MTU parameter configured for that interface. This fragmentation process takes place at the network layer and these fragments are marked so that the IP layer of the destination host knows how to reassemble the fragments into the original datagram.

### 2.3.4 Switches

Traditional switches operate on Ethernet frames and thus are layer 2 devices. These switches forward frames based on MAC addresses. When a frame reaches a switch, the switch examines the layer 2 destination address of the frame and attempts to forward the frame on the interface that leads to the destination. Thus, switches can interconnect different physical layer technologies, including 10BaseT, 100BaseT, and Gigabit Ethernet. Switches which operate on layer 3 are called layer 3 switches. A layer 3 switch can perform some or all of the functions normally performed by a router (device that forwards traffic from one network to another).

## 2.4 Transport Layer Protocols

Network layer protocols such as IP normally employ “best effort” communications where delivery is not guaranteed. Transport layer protocols such as TCP and UDP provide multiplexing of data from different sources and possibly also reliability. A port is a transport-layer software

entity that is identified by a port number, a 16 bit integer value, allowing for port numbers between 0 and 65,535.

### 2.4.1 User Datagram Protocol (UDP)

UDP is a layer 4 protocol [17]. UDP requires that IP be used as the associated Layer 3 protocol. With UDP, a host can send messages to other hosts on an IP network without requiring prior communication to set up defined transmission channels or data paths. UDP does not guarantee reliability, ordering, or data integrity; that is, Layer 4 segments may arrive out of order, appear duplicated, or go missing without notice. UDP assumes that error checking and correction is either not necessary or is performed in the higher layers, avoiding the overhead of such processing.

The UDP segment structure is shown in Table 2.3. It consists of a header part and a "DATA" part. The UDP header consists of only 4 fields:

- Source port: This field identifies the sending port when meaningful and should be assumed to be the port to reply to if needed.
- Destination port: This field identifies the destination port number and is required.
- Length: A 16-bit field that specifies the length in bytes of the entire segment; i.e., the header and data. The minimum length is 8 bytes since that is the length of the header. The field size sets a limit of 65,535 bytes for a UDP segment.
- Checksum: The 16-bit checksum field is used for error-checking of the header and data. The algorithm for computing the checksum is different for transport over IPv4 and IPv6. If the checksum is omitted in IPv4, the field uses the value all-zeros. This field is not optional for IPv6.

The UDP message is encapsulated into the DATA field of the IP datagram shown in Table 2.2 when it is passed from the transport layer to the network layer.

| Bits | 0 - 15 | 16 - 31 |
|---|---|---|
| 0 | UDP Source Port | UDP Destination Port |
| 32 | Message Length | Checksum |
| 64 | DATA | |

Table 2.3: UDP segment structure.

### 2.4.2 Transmission Control Protocol (TCP)

TCP is a transport layer protocol and runs over IP [16]. Along with all features mentioned for UDP, TCP is intended to provide a reliable process-to-process communication service. TCP is said to be connection-oriented because before one process can begin to send data to another, the two processes must first "handshake" with each other; i.e., they must send preliminary segments to each other to establish the parameters of the ensuing data transfer. The maximum amount of data that can be placed in a TCP segment is limited by the "maximum segment size" (MSS) which is typically equal to the MTU. TCP uses data sequence numbering to identify segments, and explicit acknowledgments (ACKs) to notify the sender and receiver of reliable transfer. This form of reliable protocol design is termed "end-to-end" control.

The structure of the TCP segment is shown in the Table 2.4. The TCP segment consists of header fields and a data field. The data field contains application data. When TCP sends a large amount of data, it typically breaks the data into pieces of size MSS or less. As with UDP, the header includes source and destination port numbers, which are used for multiplexing/de-multiplexing data from/to upper-layer applications. The header also includes a checksum field. Apart from these, the TCP segment header also contains the following fields:

- Sequence Number and Acknowledgment Number: These 32-bit fields are used by the TCP sender and receiver in implementing a reliable data transfer service.

- Hlen: The 4-bit header length field specifies the length of the TCP header in 32-bit words. The header length can be of variable length due to the Options field. Typically, the options field is empty, so the length of the typical TCP header is 20 bytes.

| Bits | 0 - 3 | 4 - 9 | 10 - 15 | 16 - 31 |
|---|---|---|---|---|
| 0 | Source Port | | | Destination Port |
| 32 | Sequence Number | | | |
| 64 | Acknowledgment Number | | | |
| 96 | Hlen | Reserved | Code Bits | Window |
| 128 | Checksum | | | Urgent Pointer |
| 160 | Options (if any) | | | |
| 192 | DATA | | | |

Table 2.4: The structure of a TCP segment

- Window: The 16-bit window field is used for flow control. It is used to indicate the number of bytes that a receiver is willing to accept.
- Code Bits and Urgent Pointer: "Code Bits" is a 6-bit flag field and the Urgent Pointer is a 16-bit field.
- Options: The variable-length "options" field is used when a sender and receiver negotiate the MSS. Additional information can be found in [21].

### 2.4.3 Sockets

A process associates with a particular port to send and receive data; that is, it will "listen" for incoming packets whose destination port number and IP destination address match that port, and/or send outgoing packets whose source port number is set to that port. Processes create associations with transport protocol ports by means of sockets. A socket is a software entity used as the transport layer end-point. It is created by the operating system for the process and bound to a socket address which consists of a combination of a port number and an IP address. Sockets may be set to send or receive data in one direction at a time (half duplex) or simultaneously in both directions (full duplex). Different types of network sockets include datagram sockets (which use UDP), and stream sockets (which use TCP).

## 2.5 Client-Server Model

A "server" is a host that runs one or more programs which share its resources with "clients". A client does not necessarily share any of its resources, but requests a server's content or service function. Servers create sockets on start up that are in a listening state. These sockets wait for initiation from client programs. A TCP server may serve several clients concurrently, by creating a "child process" for each client and establishing a TCP connection between each child process and the client. Unique dedicated sockets are created for each connection. These are in "established" state, when a socket-to-socket virtual connection, also known as a TCP session (thus, existing on the Session Layer), is established with the remote socket, providing a bidirectional byte stream. A UDP socket cannot be in the established state. A UDP server does not create new child processes for every concurrently served client, but the same process handles incoming data packets from all remote clients sequentially through the same socket.

## 2.6 Throughput

Throughput is defined as the average rate of successful host data delivery over a communication channel. It is a metric used to measure the performance of a protocol stack over a given network. Host data is the information that the user sends. This is distinct from "total data" that includes header information for different OSI layers during the communication. For our discussion in this thesis, all communication occurs over IPv4 and Ethernet. Hence, we consider an Ethernet frame as the total data unit.

The maximum theoretical throughput, $T_{max}$ is given by

$$T_{max} = aT_i \tag{2.1}$$

where $T_i$ is the data rate for the Ethernet physical interface and

$$a = \frac{L_{fr} - L_{hdr}}{L_{fr}} \tag{2.2}$$

where $a$ accounts for the protocol overhead, $L_{hdr}$ is the length of the frame headers which includes the Ethernet, IP and the UDP headers; and $L_{fr}$ is the total length of the Ethernet frame, given by

$$L_{fr} = MTU + L_{Eth} \tag{2.3}$$

where $L_{Eth}$ is the length of the Ethernet header. These definitions and relationships apply to UDP as well as to TCP over Ethernet. TCP "segments" the data into sizes of MSS, which is typically set by the MTU; whereas for UDP, there is no explicit segmentation but IP fragments the data into fragments which equal the MTU. Thus, for both UDP and TCP over Ethernet, $MTU = 1500$ bytes.

From Sections 2.2, 2.3, 2.4.1, and 2.4.2, we calculate lengths of the IP header ($L_{IP}$), UDP header ($L_{UDP}$), TCP header ($L_{TCP}$) and $L_{Eth}$ to be 20 bytes, 8 bytes, 20 bytes, and 38 bytes respectively. Thus, $L_{hdr} = 66$ bytes for UDP over Ethernet, and $L_{hdr} = 78$ bytes for TCP over Ethernet, and we have $a = 9.57 \times 10^{-1}$ and $T_{max} = 0.957T_i$ for UDP; and $a = 9.49 \times 10^{-1}$ and $T_{max} = 0.949T_i$ for TCP. Thus, on the basis of frame structure, the throughput of TCP appears to be only about $\sim$1% less than that of UDP. However, this does not account for the additional communication required by TCP to achieve reliable delivery. The actual throughput of TCP will be studied by experiment in Section 3.2.3.

# Chapter 3

# Analysis of Some Interface-Related Features of the LWA Station MCS

As explained in Section 1.3, the physical interface between MCS and other LWA-1 level-1 subsystems is a 1000Base-T ethernet connection. In this chapter, we analyze certain features of the interface. Section 3.1 ("MCS Common ICD") provides a review of the defined interface between MCS and connected subsystems. An analysis comparing the performance of TCP and UDP over this interface is carried out in Section 3.2 ("Protocol Throughput Analysis – UDP vs. TCP"). An analysis of the rate of beam repointing that can be achieved is carried out in Section 3.3 ("Beam Repointing Rate").

## 3.1 MCS Common ICD

As described in Section 1.3.2, the MCS Common Interface Control Document (ICD) defines a common interface between MCS and connected subsystems [9]. The physical interface is a single 1000Base-T connection. The protocol interface is UDP, with direct passing of messages using sockets. The term "message" is defined to mean a single command or response, contained entirely within the data field of one or more UDP packets. Connected subsystems never initiate communications, and only respond to an MCS message. Messages from MCS are commands.

Commands can request action, status information, or both.

### 3.1.1 Management Information Base (MIB)

The MCS Common ICD specifies a "management information base" or MIB as a means for organizing subsystem status information in a manner that is jointly understood by communicating subsystems. The MIB consists of a set of entries, with each entry consisting of an "index" and a "label". Each MIB index/label possibly also has an associated data value. MIB labels consist only of text characters. MIB data values need not be text characters, and can be raw binary. Each subsystem communicating with MCS using the MCS Common ICD specifies a MIB as part of its subsystem-specific ICD. This MIB consists of required MIB entries, plus additional MIB entries which are subsystem-specific. The required MIB entries are summarized in Table 3.1. MIB indices "2" or higher are subsystem-specific, and are defined in the associated subsystem-specific ICDs.

### 3.1.2 Message Structure

The maximum size of a MCS Common ICD message is 8192 bytes. A message is subdivided into fields as defined below. Unless indicated otherwise, data are right-justified in their fields, and padded with the character " " (space). All fields except possibly DATA are in ASCII format.

1. DESTINATION: This field is 3 bytes in length. This is the intended recipient of the message. Examples are "ASP" (analog signal processor), "DP " (digital processor), "MCS", etc.

2. SENDER: This field is 3 bytes in length. This is the subsystem that sends the message.

3. TYPE: The length of this field is 3 bytes and it indicates the type of message. A detailed discussion of message types follows in Section 3.1.3.

| Index | Label | Length (in bytes) | Description | Valid Values |
|---|---|---|---|---|
| 1.1 | SUMMARY | 7 | Summary state of the subsystem | "NORMAL","WARNING", "ERROR", "BOOTING", and "SHUTDWN" |
| 1.2 | INFO | 256 (maximum) | When MIB entry 1.1 is WARNING or ERROR, this entry contains a list of MIB labels indicating the problem condition. | Relevant MIB labels. |
| 1.3 | LASTLOG | 256 (maximum) | Last internal log message. Text string with format specified in subsystem-specific ICD. | Relevant text string. |
| 1.4 | SUBSYSTEM | 3 | 3 character string indicating the subsystem. | "DP ", "ASP", "MCS", etc. |
| 1.5 | SERIALNO | 5 (maximum) | A string identifying the specific subsystem hardware "serial number". | Assigned by subsystem manufacturer in coordination with LWA Systems Engineer. |
| 1.6 | VERSION | 256 (maximum) | Version-number of locally installed software. | Associated version number |

Table 3.1: MIB entries required by the MCS Common ICD.

4. REFERENCE: MCS assigns reference numbers to messages. Responses to MCS command messages use the same reference number appearing in the command message. This field is of length 9 bytes.

5. DATALEN: This field is 4 bytes in length. This is the number of bytes in the DATA field. The DATA field is described below.

6. MJD: This field is 6 bytes in length. It is the integer part of the modified Julian day (MJD).

7. MPM: This field is 9 bytes long and is milliseconds past Universal Time (UT) midnight. The purpose of the MJD and MPM fields is primarily to confirm to the recipient that the sender has a consistent understanding of what time it is, and also to provide a convenient mechanism for keeping or searching logs.

8. The MPM field is always followed by a space.

9. DATA: This field is of variable length and is of variable format; i.e., it can be either ASCII or binary format. The contents of DATA depend on the message type.

### 3.1.3 Message Types

Message types that are common to all subsystems (i.e., required by the MCS Common ICD) are listed below. ICDs between MCS and specific subsystems may specify additional message types.

- PNG: "Ping". The purposes of this message are to confirm that a commanded system is functioning, and to disseminate or confirm time information. The subsystem responds with a PNG response message.

- RPT: "Report". The purpose of this message is to facilitate reporting of subsystem MIB data values.

- SHT: "Shutdown". The purpose of this message is to direct the subsystem to shut down.

The controlled subsystem is expected to respond to every message from the MCS with a "response message". The response message is required to be transmitted within 3 seconds of receipt of the command message from MCS. The DATA field of a response message has the following structure:

- R-RESPONSE: This is either the character "A" or "R" to indicate that the command was accepted or rejected.

- R-SUMMARY: This is the data value associated with MIB index 1.1 (see Table 3.1).

- R-COMMENT: If R-RESPONSE is "R", then this field is used to send error codes or log messages, as specified by the subsystem ICD.

### 3.1.4 Example of a Command and Response

The following example demonstrates a simple MCS Common ICD message and the resulting subsystem response message. MCS sends the message:

```
DP'MCSPNG'''''1391'''0'54828'12345678'
```

Here, "'" represents a single space and is used for clarity. DESTINATION is the DP subsystem, SENDER is MCS, and PNG is the command issued. "1391" is the REFERENCE. The length of the DATA field is "0". "54828" and "12345678" indicate the integer part of the modified Julian day (MJD) and milliseconds past UT midnight (MPM), respectively. There is no DATA field (as this is a PNG message).

The subsystem responds as follows:

```
MCSDP'PNG'''''1391'''8'54828'12345698'A'NORMAL
```

The response follows the same structure as the MCS message except for the "`A'NORMAL`" which indicates that the command was accepted by DP and the SUMMARY entry of the MIB has the value "NORMAL".

## 3.2 Protocol Throughput Analysis – UDP vs. TCP

The protocol interface between MCS and other level-1 subsystems is UDP. In this section, we analyze the performance of UDP over this interface and compare it to the performance of TCP. The metric we use for comparison is "throughput" over the interface with each protocol. This has been defined in Section 2.6. Section 3.2.1 describes the experimental set-up with which the performance evaluation was done. The set-up emulates the communication interface between the actual MCS and the subsystem. The sections following this description discuss the throughput obtained over the interface for UDP and TCP and provides a conclusion about the apt choice of protocol for the MCS interface.

### 3.2.1 Experiment Description

Figure 3.1 shows the set-up used for the experiment. Two computers, identified as "PC1" and "PC2" in Figure 3.1, are connected via a 5-port gigabit ethernet switch, Linksys Model SD2005, using Cat-5 ethernet cables. Here, PC1 is the client and PC2 is the server. PC1 (representing MCS/Scheduler) has an Intel® D945GCLF2 motherboard with integrated dual-core Intel® Atom™ processor with 533 MHz system bus, 2 GB system memory, and onboard network adapter that supports 1000 Mb/s connectivity. PC2 (representing the connected subsystem such as DP, ASP, etc) is a Shuttle XPC system that has a similar configuration as PC1, except that the onboard network adapter is only 100Base-T compliant. Hence, the set-up is limited by the data rate capability of PC2. Both PCs run the Ubuntu Linux 8.10 operating system.

The software we have developed in Python with socket programming (which will be discussed in Section 4.2) is used to estimate the throughput of the interface. The source code is included in Appendix A.1. The software consists of two Python scripts; one script that is run on the client and the other that is run on the server.

The client script builds a random ASCII message which is 8192 bytes long. This message is then sent to the server 1000 times. Each time a message is received by the server, the server records the time of receipt. The time taken for $n$ messages to arrive at the server is

$$\delta t = t_{n+1} - t_1 \tag{3.1}$$

where $t_{n+1}$ and $t_1$ are the times at which $(n+1)^{th}$ and $1^{st}$ messages are completely (the entire 8192 bytes) received by the server. The throughput $T$ in b/s is calculated as follows:

$$T(n) = \frac{8Bn}{\delta t} \tag{3.2}$$

where $B$ is the length of the message, i.e., 8192 bytes.

Additionally, the messages received by the server are sent back to the client to verify that they are the same as the messages that were sent. It was seen that there were no errors; that

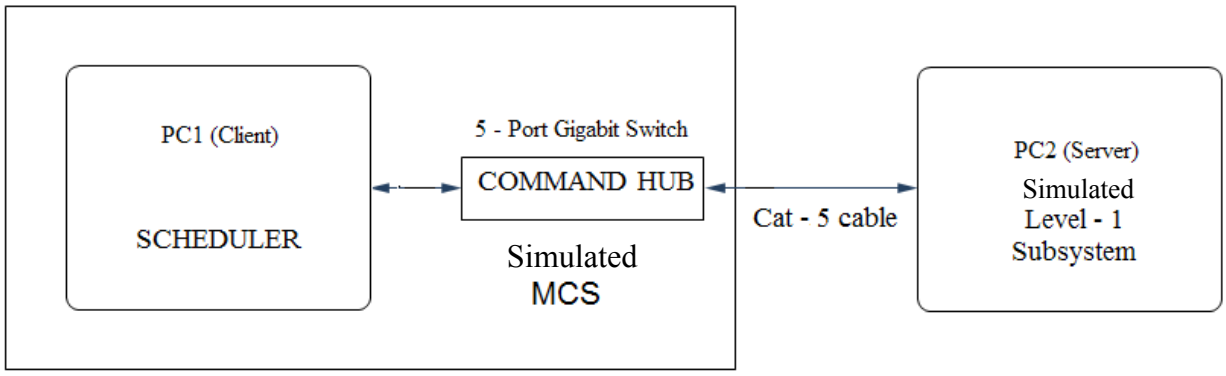


Figure 3.1: Block diagram for experimental throughput analysis.

is, no error in message content as well as no packet losses. All the messages that were sent from PC1 were received correctly and completely by PC2.

### 3.2.2 Throughput with UDP

Figure 3.2 shows the throughput measured in the above experiment in one run of 1000 messages, as the number of messages $n$ increased from zero to 160 over the course of the experiment. We observe many large variations in the plot for a very small number of messages ($\sim$50). This is expected as Equation 3.2 will approximate to the true value only for large $n$. For small values of $n$, timing inaccuracies in the calculation of $\delta t$ dominate and hence, the throughput calculation is not accurate. These timing inaccuracies get averaged out for values of $n$ greater than 75 or so and the measured throughput approximates to the true value. For 1000 messages, the average throughput across the network was found to be 95.95 Mb/s. In Section 2.6, the throughput was calculated with the maximum header overhead and $T_{max}$ is found to be 95.7 Mb/s for $T_i$ = 100 Mb/s. A possible reason why the measured throughput is greater is as follows: As seen earlier in Section 2.6, UDP does not segment as TCP does and fragmentation is done at the network layer by IP. The UDP header has a fixed length of 8 bytes. So, the first frame can carry up to 1472 bytes of data, which gives us the throughput to be 95.7 Mb/s. But, subsequent frames can carry a maximum of 1480 bytes of data (as these frames do not carry

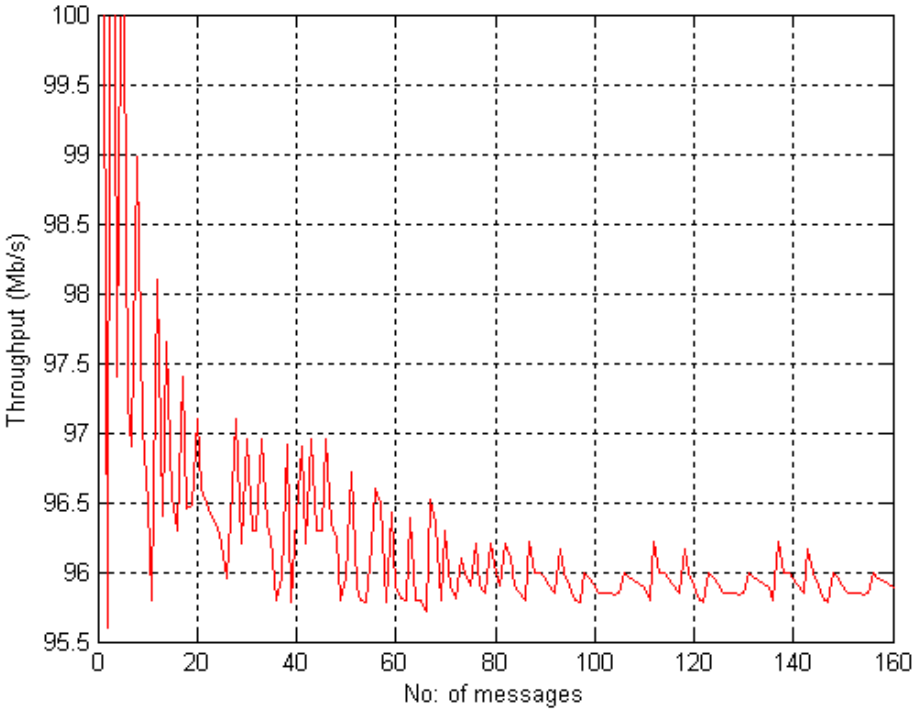


Figure 3.2: Throughput with UDP vs. number of messages.

the UDP header) which gives a maximum theoretical throughput for the subsequent frames as 96.2 Mb/s. Thus, we find that theory and experiment are in close agreement.

As discussed in Section 2.4.1, UDP is not a reliable protocol; in the sense that, it provides no built-in mechanism to ensure that the packets arrive in the proper order, or even arrive at all, and cannot detect errors. For the experiment conducted however, it was seen that there were no packet losses across the network. Thus, for our purpose, UDP is sufficiently reliable.

The experimental conditions do not take network congestion into account as this is not expected to be a problem for the actual MCS scenario. In LWA-1, each subsystem has a dedicated gigabit ethernet interface with the MCS Command Hub (as described in Section 1.5), and the Command Hub is managed in order to guarantee sufficient contention-free bandwidth to connected devices.

### 3.2.3 Throughput with TCP

Here, we repeat the experiment using TCP in place of UDP. Figure 3.3 shows the throughput vs. number of messages, in the same manner as Figure 3.2. The throughput ranges between a maximum value of 81.65 Mb/s and a minimum value of 80.3 Mb/s and does not change much from 81.4 Mb/s for large $n$. Figure 3.4 is another example that shows the throughput vs. number of messages. The experiment was repeated many times and similar behavior was observed in all trials. The maximum theoretical value of the TCP throughput (from Section 2.6) is 94.92 Mb/s for $T_i = 100$ Mb/s. The experimentally obtained throughput is significantly less than the theoretically obtained throughput and the throughput obtained with UDP.

To determine why the observed throughput is less than the expected value of throughput, TCP packets received at PC2 were monitored at the physical layer using the network analyzer software, Wireshark.[1] It was observed that each message of length 8192 bytes was fragmented into 6 fragments of size equal to MSS (explained earlier in Section 2.4.2) or less. The MSS was seen to be 1460 bytes in our experiment as is expected since the MTU is 1500 bytes and 40 bytes account for the header overhead. Thus, fragmentation, acknowledgments (ACKs) for the fragments, and reassembly of the fragments at the receiving end also contribute to the overhead. The time required to send a TCP segment and receive ACK for the segment sent is called the "roundtrip time" (RTT) and this affects the throughput. Higher the RTT, lesser is the throughput. This was not accounted for in the theoretical maximum throughput calculation of Section 2.6, which considered only protocol overhead due to headers. This experiment shows that TCP acknowledgment mechanism reduces the maximum throughput by an additional 14%.

We now attempt to explain the distinctive variations and discontinuities in throughput vs. time seen in Figures 3.3 and 3.4. Huang and Subhlok (2005) described an approach to predict throughput for TCP, based on TCP "flow pattern" [22]. To understand this better, we first describe the TCP flow. TCP has no advance knowledge of network characteristics and uses congestion control mechanisms to adjust its behavior to avoid overwhelming the network. TCP

[1] http://www.wireshark.org

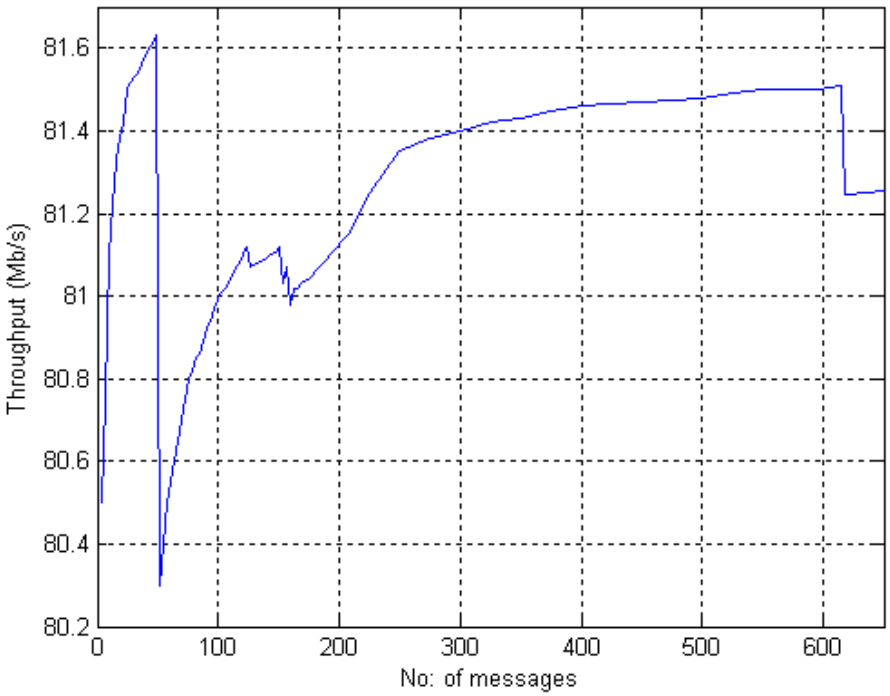


Figure 3.3: Throughput with TCP vs. number of messages.

limits its sending rate with a congestion window (*cwnd*), so TCP can send up to *cwnd* bytes at a time. A TCP flow normally starts with a slow start, followed by a steady-state governed by TCP congestion control and flow control. In the initial "slow start" phase, TCP tries to determine the maximum available bandwidth by increasing *cwnd* exponentially, until a predefined timeout or a loss event occurs. A TCP flow enters "steady state" phase after the initial slow start phase. Thus, the initial ramp up seen in Figures 3.3 and 3.4 may be the "slow start". In the steady state phase, the throughput of TCP is deterministic and theoretically predictable [23]. Thus, according to the flow patterns classifications described by Huang and Subhlok, the throughput pattern in our experiment (Figures 3.3 and 3.4) falls under the "rate control limited (RC)" pattern. In this, the receiving rate of TCP segments are limited by receiver's buffer size (flow control) and/or sender's management of outgoing TCP streams. This pattern is characterized by large "flat regions" in the throughput over time after the "slow start". This is consistent with what is seen in Figures 3.3 and 3.4.

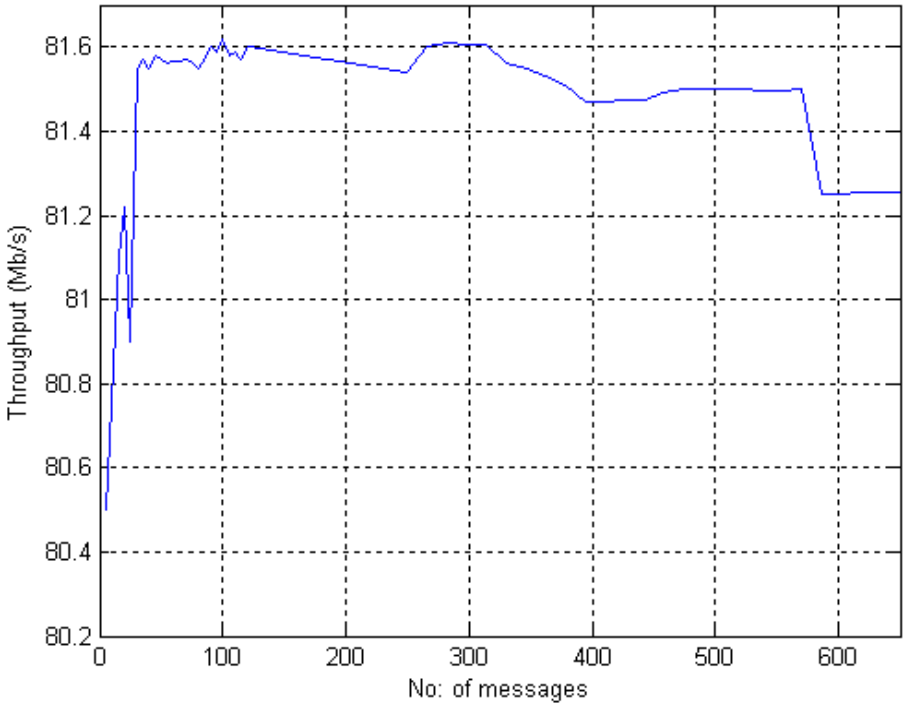


Figure 3.4: Throughput with TCP (Mb/s) vs. number of messages.

### 3.2.4 Comparison

The measured average throughput with UDP is ∼15% greater than the throughput with TCP. Even though TCP offers reliability, UDP is better for our purpose, because 1) Network reliability is already guaranteed because subsystems have a dedicated ethernet interfaces with the MCS Command Hub, and the Command Hub is managed to ensure sufficient bandwidth to the connected devices. 2) UDP is faster, as is demonstrated by the experiment. The observed UDP throughput is very close to the maximum throughput that is possible.

## 3.3 Beam Repointing Rate

The discrete-time nature of array digital signal processing in LWA means a beam cannot track a source continuously as it moves through the sky, but rather only in discrete steps. Ellingson (2008) mentions that repointing the LWA station beam as quickly as possible will result in a subtle “rumbling” of both the time-domain output and the antenna pattern in a way that may

be hard to quantify and that is likely to compromise instrumental stability [24]. Thus, tracking with LWA is achieved by holding the station beam utterly constant for as long as possible, and then "jumping" to a new pointing only when necessary. This is called beam "repointing". As mentioned in Section 1.2, it is desired to use one of the LWA station beams to observe sources in a rapidly repeating cycle, in order to provide data on the dynamically-varying ionosphere [6]. Ellingson and Cohen (2008) find that this scheme drives requirements for the "maximum repointing time" for station beamforming [7]. "Maximum repointing time" is defined as the maximum time required to repoint the beam, and during which the output of the station electronics i.e., the DP, is assumed to be invalid. The most demanding speed requirement for MCS is the ability to repoint a "calibration beam" within 5 ms, repeating every 60 ms [7].

In this section, we estimate the theoretical maximum rate at which beams can be switched as a function of the maximum theoretical throughput for the ethernet interface, $R$ (which is equal to the $T_{max}$ in Section 2.6) for a UDP-based MCS Common ICD connection with no contention. More information on the beam repointing is discussed in Section 3.3.1. The analysis is carried out in Section 3.3.2.

### 3.3.1 Background

As described in Section 1.2, the DP subsystem performs the task of beamforming. MCS controls DP according to the ICD for DP [25]. The associated command message types are "BAM" and "DRX". Repointing a beam may require adjustment of gain, delay, dispersion, DRX center frequency, and/or DRX bandwidth. Gain, delay, and dispersion are controlled by the "BAM" command. DRX center frequency and bandwidth are controlled by the "DRX" command. Repointing a beam requires at most one BAM and one DRX command.

### 3.3.2 Analysis

The maximum rate of beam repointing depends on the time required to transmit a BAM and a DRX command. From Section 2.2, 2.4.1, and 2.3, we see that the length of each UDP

segment over IP depends on the length of the UDP/IP headers, Ethernet header (because of encapsulation), and the length of the payload. The payload length in turn depends on the length of the MCS message. Time required to transmit a single Ethernet frame (as defined in Section 2.6) of length $L$ bytes at the maximum possible throughput $R$ bytes/s is

$$T_{tr} = L/R. \tag{3.3}$$

The total length of the frame for the command, $L$, is

$$L = L_{UDP} + L_{Eth} + L_{IP} + L_M \tag{3.4}$$

where $L_{UDP}$, $L_{Eth}$, $L_{IP}$ follow the definitions in Section 2.6 and $L_M$ denotes the length of the message from MCS to DP (which follows the structure described in Section 3.1.2), respectively. Let the time taken to transmit a frame for the BAM command be given by

$$T_1 = \frac{L_1}{R}, \tag{3.5}$$

where $L_1 = L_{UDP} + L_{Eth} + L_{IP} + L_{M1}$, and $L_{M1}$ denotes the length of message issued by MCS for the BAM command.

Similarly, the time taken to transmit the DRX command is given by

$$T_2 = \frac{L_2}{R}, \tag{3.6}$$

where $L_2 = L_{UDP} + L_{Eth} + L_{IP} + L_{M2}$, and $L_{M2}$ denotes the length of message issued by MCS for the DRX command.

The total time required to repoint a beam, assuming that the time required to assemble, interpret and implement the command is negligible in comparison, is

$$T = T_1 + T_2. \tag{3.7}$$

Thus:

$$T = \frac{2}{R}(L_{UDP} + L_{Eth} + L_{IP}) + \frac{1}{R}(L_{M1} + L_{M2}). \tag{3.8}$$

Substituting for the values of the lengths of the Ethernet and UDP/IP headers; i.e., $L_{UDP}$ = 8 bytes, $L_{Eth}$ = 38 bytes, and $L_{IP}$ = 20 bytes, the beam repointing time is:

$$T = \frac{132 \text{ bytes}}{R} + \frac{1}{R}\left(L_{M1} + L_{M2}\right).$$

$L_{M1}$ includes the DATA field for the BAM command along with the fixed lengths (as described in Section 3.1.2). Thus, according to [25], $L_{M1}$ = 3145 bytes and the length of the entire message for the DRX command is $L_{M2}$ = 51 bytes. Thus, the total beam repointing time, as a function of the throughput $R$ is

$$T = \frac{132 \text{ bytes}}{R} + \frac{3196 \text{ bytes}}{R}. \tag{3.9}$$

The beam repointing rate, $S_r$, is the reciprocal of the time required to repoint a beam,

$$S_r = \frac{R}{3328 \text{ bytes}}. \tag{3.10}$$

Here, $R$ is the maximum possible theoretical throughput, which is less than the throughput over the ethernet interface, $R_i$. The maximum possible theoretical throughput $R$ is given by

$$R = aR_i \tag{3.11}$$

where $a$ is the factor that accounts for the header overhead which results in a reduced throughput, given by Equation 2.2. Thus, $S_r$ is given by:

$$S_r = \frac{aR_i}{3328 \text{ bytes}}. \tag{3.12}$$

where for $R_i$ = 1000 Mb/s and $a = 9.57 \times 10^{-1}$, $S_r$ is 35.9 kHz and $T$ is $2.77 \times 10^{-2}$ ms. This

$T$ meets the system requirement that the maximum repointing time should be less than 5 ms.

This analysis does not take network congestion into account. As described in the previous section, congestion is not expected. However, it is interesting to consider how congestion might affect the repointing rate. Network congestion would further reduce the throughput and increase the repointing time. The throughput could go as low as

$$R_{min} = \frac{3328 \times 8 \text{ bits}}{5 \times 10^{-3} \text{ s}} \tag{3.13}$$

i.e., $R_{min} = 5.32$ Mb/s, which is $\sim$188 times less than the expected throughput, would still meet the 5 ms requirement.

# Chapter 4

# Software for Development and Integration of MCS Interfaces

Software to facilitate the development of actual subsystems and MCS itself were developed in Python. This chapter discusses in detail about this software. Section 4.1 discusses the reasons why we choose Python for development of this software. A "tier zero" interface to communicate directly with the subsystems was also written and is discussed in Section 4.2. Software was developed to check the compliance of a subsystem's responses with the MCS ICD and is explained in Section 4.3. Section 4.4 gives an overview of the "Shelter" (SHL) subsystem's functions and discusses the SHL emulator software.

## 4.1 Python as the Programming Language

The Python Standard Library is an excellent attribute of Python. The standard library allows the programmer to perform complex tasks without having to write additional software or go elsewhere for suitable scripts. A module is a file containing a set of Python functions. Python functions provide date/time functionality, file and directory manipulation, etc. We use many of these built-in modules in the development of the software described in this chapter. For example, we use the `socket` module as the interface to the operating system's socket implementation.

A detailed description of socket programming in Python can be found in [26]. The `socket` module consists of the `socket()` function, which is used for socket initiation/creation. We will be primarily concerned with this function only, i.e., `socket.socket()`. The `socket` module supports various protocols for communication, including TCP and UDP. Here, we discuss only UDP socket programming. We introduce an example in Section 4.2 to demonstrate socket programming in Python.

## 4.2 "Tier-Zero" Interface

As explained in Section 1.3, a tier-zero interface is one that is used to communicate directly with subsystems using the MCS Common ICD protocol [9]. The Python script `mcs_tzi1.py` (included in Appendix A.2 and also found in [27]) was written with the intent to facilitate subsystem development. This script acts as a client-side interface to send commands to the subsystem (the server) under test. The following steps are followed to create a socket and send a message:

1. Import the `socket` module: `import socket`
2. Initialize parameters necessary before creating a socket; i.e., the IP address of the host (the server), the port number, and the buffer size. The buffer is the maximum size of the data stream or the payload that can be transmitted at a time through the socket. In our software, we assign the buffer value to be 8192 bytes as that is the maximum length for a MCS message.
3. Create a datagram socket: `t = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)`. `SOCK_DGRAM` indicates that we are using UDP for communication.
4. Connect to the server: `t.connect((DEST_IP, PORTT))` where DEST_IP and PORTT are the initialized values for the server IP address and port number.
5. Build the message and send it to the server: `t.send(message)`

The message is built according to the MCS message structure described in Section 3.1. Similarly, a "receive" socket is also created for receiving messages from the server.

1. Create a datagram socket: `r = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)`.
2. "Bind" the socket to the server's port ("PORT") to enable it to accept connections: `r.bind(('', PORT))`.

The script is a generic one and is not subsystem-specific. Thus, the user is required to enter all the necessary information required to set up a communication with the subsystem as command-line arguments. The syntax for executing the script is as follows:

```
$ python mcs_tzi1.py <DEST_NAME> <DEST_IP> <TX_PORT> <RX_PORT> <CMD> <REF> <DATA>
```

or,

```
$ python mcs_tzi1.py <DEST_NAME> <DEST_IP> <TX_PORT> <RX_PORT> <CMD> <REF> -f <DATA_FILE>
```

where,

- DEST_NAME is the subsystem identifier (3-character string).
- DEST_IP is the IP address of the destination.
- TX_PORT is the transmit port set up for socket communication.
- RX_PORT is the receive port set up for socket communication.
- CMD is the command required to be issued to the subsystem.
- REF is the reference number MCS assigns (described in Section 3.1).
- DATA is the "DATA" field for the command, used when the command accepts exclusively ASCII-formatted arguments.
- DATA_FILE is used when the command requires raw binary input (as is the case for many DP commands). In this case, DATA_FILE specifies the file that contains all the data that goes in the "DATA" field.

A sample run of this script is shown below:

```
$ python mcs_tzi1.py SHL 127.0.0.1 1738 1739 RPT 'VERSION'
sent> SHLMCSRPT 1391 7 55046 49080819 VERSION|
recd> MCSSHLRPT 1391 21 55046 49080823 A NORMAL mch_shl_1.py|
```

where "`mch_tzi11.py`" is the name of the script, "`sent>`" and "`recd>`" indicate the messages sent and received by the script respectively, and "|" is for readability and indicates the termination of the messages. The shaded text represents the information entered by the user. The various fields displayed in the messages are same as that in a MCS message and the response from the subsystem.

## 4.3 MCS ICD Compliance Check Software

The compliance check software checks the compliance of the subsystem responses against the MCS Common ICD. The script `mcs_mincompchk_1.py` for the compliance check is included in Appendix A.3 and can also be found in [28]. This code emulates the MCS side, sending a sequence of command messages, and evaluating the responses. This script checks whether the minimal set of commands (i.e., PNG, RPT, and SHT) and minimum required MIB entries (i.e., the MCS-RESERVED branch explained in Section 3.1) are supported, and verifies that the responses are compliant with the MCS Common ICD. The results of the tests are written to an output file (`MCS_mincomp_Report.txt`). The tests are described below.

The compliance check is run for the subsystem under test. The subsystem's responses to the minimal set of commands; i.e., the commands in the MCS-RESERVED section of the ICD (described previously in Section 3.1) are checked with the message structure defined in the MCS Common ICD. PNG, RPT and SHT commands are issued by the script, and the resulting responses checked. Also, the response times (the time taken to receive a response from the server once the client has sent its message to the server) are also checked to ensure they are less than the maximum time within which the subsystem is required to respond; i.e., 3 s according to MCS Common ICD. The results of the tests, along with the messages sent and received at the

client side, are written to a file. An example of use is given here. From the client side:

```
$ python mcs_mincompchk_1.py SHL 1738 1739
$ cat MCS_mincomp_Report.txt
STARTING TO TEST COMPLIANCE with MCS Common ICD...
   COMMANDS required by MCS Common ICD
1.  PNG Command
sent> SHLMCSPNG 1391 0 55046 43490669 |
recd> MCSSHLPNG 1391 8 55046 43490673 A NORMAL|
[PASS]
RPT Command
2.  With Label:  SUMMARY
sent> SHLMCSRPT 1391 7 55046 43490674 SUMMARY|
recd> MCSSHLRPT 1391 15 55046 43490677 A NORMAL NORMAL|
[PASS]
3.  With Label:  INFO
sent> SHLMCSRPT 1391 4 55046 43490678 INFO|
recd> MCSSHLRPT 1391 21 55046 43490681 A NORMAL This is INFO|
[PASS]
4.  With Label:  LASTLOG
sent> SHLMCSRPT 1391 7 55046 43490682 LASTLOG|
recd> MCSSHLRPT 1391 24 55046 43490686 A NORMAL This is LASTLOG|
[PASS]
5.  With Label:  SUBSYSTEM
sent> SHLMCSRPT 1391 9 55046 43490686 SUBSYSTEM|
recd> MCSSHLRPT 1391 12 55046 43490690 A NORMAL SHL|
[PASS]
6.  With Label:  SERIALNO
sent> SHLMCSRPT 1391 8 55046 43490691 SERIALNO|
recd> MCSSHLRPT 1391 10 55046 43490695 A NORMAL 2|
[PASS]
7.  With Label:  VERSION
sent> SHLMCSRPT 1391 7 55046 43490695 VERSION|
recd> MCSSHLRPT 1391 21 55046 43490699 A NORMAL mch_shl_1.py|
[PASS]
8.  With Invalid/Bogus Label RANDOM
```

```
sent> SHLMCSRPT 1391 6 55046 43490699 RANDOM|
recd> MCSSHLRPT 1391 26 55046 43490703 R NORMAL Invalid MIB label|
The requested label not found.  Expected behavior.  Looks good
[PASS]
9.  Passing an invalid command - RND
sent> SHLMCSRND 1391 0 55046 43490704 |
recd> MCSSHLRND 1391 31 55046 43490707 R NORMAL Command not recognized|
[PASS]
RESPONSE TIMES
10.  Running PNG 100 times
Minimum response time = 1.08504295349 ms
Maximum response time = 6.59799575806 ms
Number of times PNG was accepted = 100
Number of times PNG was rejected = 0
[PASS]
11.  Running RPT with valid arguments 100 times
Minimum response time = 1.1260509491 ms
Maximum response time = 1.33800506592 ms
Number of times RPT was accepted = 100
Number of times RPT was rejected = 0
[PASS]
12.  Running RPT with invalid arguments 100 times
Minimum response time = 1.13916397095 ms
Maximum response time = 1.23596191406 ms
Number of times RPT was accepted = 0
Number of times RPT was rejected = 100
[PASS]
SHT Command
13.  Testing with argument:  RESTART
sent> SHLMCSSHT 1391 7 55046 43491169 RESTART|
recd> MCSSHLSHT 1391 8 55046 43491172 ASHUTDWN|
[PASS]
14.  Testing with argument:  RANDOM
sent> SHLMCSSHT 1391 6 55046 43491172 RANDOM|
recd> MCSSHLSHT 1391 32 55046 43491174 RSHUTDWN Invalid extra arguments|
Rejected due to incorrect argument.  Expected.
```

```
[PASS]
15.  Testing with argument:  SCRAM
sent> SHLMCSSHT 1391 5 55046 43491174 SCRAM|
recd> MCSSHLSHT 1391 8 55046 43491176 ASHUTDWN|
[PASS]
16.  Testing with argument:
sent> SHLMCSSHT 1391 0 55046 43491176 |
recd> MCSSHLSHT 1391 8 55046 43491179 ASHUTDWN|
[PASS]
TESTING COMPLETED...
PASS
```

In this case, the subsystem under test has passed. Had one or more of the tests failed, then the script would have terminated with the message "`FAIL`" and also given explanatory comments on the tests that failed.

## 4.4 MCS Common ICD Emulation Software for SHL

The MCS Common ICD Emulation software for the shelter subsystem (SHL) consists of a Python script. An additional Python script, which acts as the client (MCS) is also provided to demonstrate the emulator. The scripts are included in Appendix A.4 and can also be found in [29]. This software was developed to support the development of MCS by providing a simulation of SHL as it appears via the MCS Common ICD, and to support SHL development by demonstrating the expected behavior of the subsystem and providing "starter code" for the development of SHL-MCS and MCS's of other subsystems. The software consists of two Python scripts, one of which (`mch_shl_1.py`) emulates the shelter (SHL) subsystem and is run as the server, while the other (`mch_cl_1.py`) is used to test and demonstrate the emulator and is run as the client that emulates the MCS. The SHL emulator is described in Section 4.4.1 and the client for the SHL is explained in Section 4.4.2. An example demonstrating the execution of the above scripts is provided in Section 4.4.3.

| Command | Purpose | DATA format [Length in Bytes (B)] | Example |
|---|---|---|---|
| INI | Informs SHL of set-point, differential and number of racks installed | "Setpoint&Differential&Racks installed" [5B&3B&6B] | "00072&1.0&111000" (Set-point = 72°F Differential = 1.0°F Racks 1, 2, 3 installed) |
| TMP | Changes shelter temperature set-point | "Temperature" [5B] | "00072" (Set-point = 72°F) |
| DIF | Changes shelter temperature differential | "Differential" [3B] | "1.5" (Differential = 1.5°F) |
| PWR | Controls power ports | "RackPortControl" [1B2B3B] | "204OFF" (Rack 2 Port 4 turned OFF) |

Table 4.1: Summary of SHL commands.

### 4.4.1 SHL emulator

The SHL emulator (`mch_shl_1.py`) implements all the commands defined in the SHL Common ICD [30]. The commands defined in the SHL Common ICD are summarized in Table 4.1. The commands in the MCS-RESERVED section (PNG, SHT, RPT) of the MIB are also implemented. The SHL emulator updates a text file, `MIB_shl_1.txt` which stores data specified in the SHL MIB. A description of the SHL MIB can be found in [30].

The subsystem script emulates the SHL behavior completely but does not do anything "intelligent"; in particular, it does not take actions independently of commands issued. For example, the current temperature (recorded in the MIB) is set equal to the value set by the TMP/INI command, and upon receiving a SHT command, the emulation changes the MIB SUMMARY field indicating a 'shutdown'.

### 4.4.2 Client for the SHL emulator

The script `mch_cl_1.py` is used in lieu of MCS to communicate with the SHL emulator. The client script also checks for errors in the values entered by the user with the range of values accepted by SHL. The client also maintains a file, `MIB_cl_1.txt`, which is updated as valid responses are received. This is a simple text file that can be used to observe the state of `MIB_shl_1.txt` that is maintained by the SHL emulator.

### 4.4.3 Example of use

Server-side:

```
$ python mch_shl_1.py
Loading MIB...
I am SHL...
Running...
```

Client-side:

```
$ python mch_cl_1.py INI 90 2.5 111110
sent> SHLMCSINI 1391 16 55026 74173718 00090&2.5&111110|
rcvd> MCSSHLINI 1391 8 55026 74173730 A NORMAL|
Error Code (0 is good):  0
```

where "|" is for readability and indicates end of a message. The shaded text represents the information entered by the user.

# Chapter 5

# Conclusions

We discussed the functions and architecture of LWA MCS in Chapter 1. We analyzed some of the interface-related features of MCS in Chapter 3. The MCS Common ICD defines the MCS's interface with level-1 subsystems as a 1000Base-T ethernet interface which uses UDP for communication. We compared the performance of TCP and UDP in an experiment that emulated the actual MCS design and found UDP to be ∼15% faster than TCP and reliable for our requirements. We also discussed the rate at which MCS can direct DP to repoint beams in Section 3.3. We confirmed that MCS is at least 2 orders of magnitude faster than necessary, and that the rate of beam repointing is limited primarily by the network throughput. Python software supporting the development of LWA subsystems, including MCS, were discussed in Chapter 4.

Improvements could be made to our experiment and analysis. We did not consider congestion in the network while estimating the throughput over the interface since all subsystems have dedicated interfaces with the MCS Command Hub. But if load on the MCS (with more number of subsystems connected) increases, it would be then useful to take congestion in the network into account also and again compare performance of UDP and TCP. With congestion in the network, TCP (with its congestion control) could be expected to outperform UDP. Also, in the analysis of repointing beams, we assumed the time to assemble and process a message

is negligible. These values could be estimated and also accounted for in the time required for repointing. Even then, we are compelled to believe that MCS will still outperform the minimum repointing time required of it by a large margin.

# Appendix A

# Python Software

## A.1 Scripts for Estimation of Throughput

This section is referred to from Section 3.2.1. We include the Python scripts for the estimation of throughput with TCP and UDP.

### A.1.1 Throughput with UDP

Here, we include the Python scripts for the estimation of throughput with UDP. It consists of two scripts; one that is run on the server (`thput_udp_c.py`) and the other that is run on the client (`thput_udp_c.py`).

Script to be run on PC2 ("server"):

```
# thput_udp_s.py — A. Srinivasan, VT — 2009 Nov
# Used to determine integrity and throughput with UDP between
# two PCs connected through the MCS Command Hub
# This is the "server" side (the other end is the "client")
# Set HOST.  May also neet to change PORT.
# This code runs forever — use CTRL—C or whatever to crash out when done.

import socket
```

```
import time
from pylab import * #To plot graph
HOST = '192.168.30.2' # The IP address of the client
PORT = 1738           # The port address

B = 8192 # [bytes] Buffer size —— this must be >= B
# as set on the client side.
# Set up the receive socket for UDP
r = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
r.bind(('',PORT))  # Accept connections from anywhere
r.setblocking(1)   # Blocking on this sock
# Set up the transmit socket for UDP
t = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
t.connect((HOST,PORT))
print 'Running... Nothing will be displayed'
N = 0 # So far, no message received
n_arr = [] # array to store instantaneous
number of messages received
throughput = [] #array to store values of throughput for increasing n
while 1:
    data = r.recv(B)  # wait for something to appear
    #t.send(data) # Data sent back to client to check for message losses or errors
    if (N==0):
      t0 = time.time() # Set time of receipt of first message
    N = N + 1 # Increment N
    #print len(data)
    t1 = time.time() # Time of receipt for successive messages
    th = 8*B*(N-1)/((t1-t0)*1e+6) # Throughput calculation
    n_arr.append(N)# Store instantaneous N in n_arr
    throughput.append(th) # Store respective throughput values
    if (N==1000):
       plot(n_arr,throughput)
       axis((0,250,80,110))
       xlabel('No: of messages')
       ylabel('Throughput (Mb/s)')
       title('Throughput of the interface with UDP')
       show()

# never get here, but what the heck.
s.close()
t.close()
```

Script to be run on PC1 (“client”)

```
# thput_udp_c.py — A. Srinivasan
# Used to determine integrity and throughput with UDP
between two PCs connected through the MCS Command Hub
# This is the "client" side (the other end is the "server")
# Set N and HOST.  May also need to change PORT.
# This is a "one shot" code —— it does the test and then quits

import socket
import time   # for measurement of transfer rate
import random # for generation of pseudorandom data

N = 1000 # number of times to repeat transfer (make large for accurate estimate)
HOST = '192.168.30.1' # The IP address of the server
PORT = 1738          # The port address

B = 8192 # [bytes] Message size;
amount of data to transfer per call
# build a message consisting of random digits (ASCII)
payload = ''
for n in range(B):
    payload = payload + str(random.choice(range(10)))
# print len(payload), payload

# Set up the receive socket for UDP
r = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
r.bind(('', PORT))  # Accept connections from anywhere
r.setblocking(1)    # Blocking on this socket

# Set up transmit socket for UDP
t = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
t.connect((HOST, PORT))
# Main loop
t0 = time.time()       # remember start time
for n in range(N):     # do this N times
    t.send(payload)        # send the message
    #data = r.recv(B)      # see what the server
comes back with
    # if data==payload: # to check for message errors
    #    print str(n)+' success'
```

```
    # else:
    #    print str(n)+' fail'
    t1 = time.time()       # remember end time

r.close() # close receive socket
t.close() # close transmit socket
```

### A.1.2 Throughput with TCP

Here, we include the Python scripts for the estimation of throughput with TCP. It consists of two scripts; one that is run on the server (`thput_tcp_s.py`) and the other that is run on the client (`thput_tcp_c.py`).

Script to be run on PC2 ("server"):

```
# thput_tcp_s.py — A. Srinivasan, Nov 2009
# Used to determine integrity and throughput over the interface
# between two PCs connected through the MCS Command Hub
# This is the "server" side (the other end is the "client")
# Set HOST.  May also neet to change PORT.
# This code runs forever —— use CTRL—C or whatever to crash out when done.

#import matplotlib.pyplot as plt
import socket
import time
from pylab import * # For plotting graph
HOST = '192.168.30.2' # The IP address of the client
PORTR = 1738          # The port address

B = 8192 # [bytes] Buffer size
# this must be >= B as set on the client side.
r = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# TCP Stream Socket
r.bind(('',PORTR))  # Accept connections from anywhere
r.listen(1)
r.setblocking(1)
time_arr = [] #Initializing an array for time—interval,
(delta)(t)
print 'Running... Nothing will be displayed'
N = 0 # So far, no packets
```

```
n_arr = []
throughput = [] #Initializing array for instantaneous throughput
while 1:
   client_sock, address = r.accept()
   if (N==0):
     t0 = time.time() #Setting initial time
   client_sock.settimeout(3)
   #data = ''
   #while(len(data)<B):
   l = 0;
   data = ''
   time_arr.append(t0)
   while(l<B):
      data = data + client_sock.recv(B)
# Wait until the entire message is received — needs explicit delimiting of messages.
      l = len(data)
      #print 'Server: Received length '+ str(l)+'\n'
   t1 = time.time() # Time of arrival of successive packets
   time_arr.append(t1)# Storing the time—delay for inter—pakcet arrival
   client_sock.close()# Terminate connection
   N = N + 1 #Increment N
   n_arr.append(N)# Stroing N in n_arr
   th = 8*N*B/((t1—t0)*1e+6)#calculating throughput
   throughput.append(th)
   if (N==1000):
      #print throughput
      plot(n_arr, throughput)
      show()

# never get here, even then,
r.close()
```

Script to be run on PC1 (“client”):

```
# thput_tcp_c.py — A. Srinivasan, Nov 2009
# Used to determine integrity and throughput with TCP between two PCs connected through the MCS Command Hub
# This is the "client" side (the other end is the "server")
# Set N and HOST.  May also need to change PORT.
# This is a "one shot" code —— it does the test and then quits

import socket
```

```
import time   # for measurement of transfer rate
import random # for generation of pseudorandom data

N = 1000 # number of times to repeat transfer (make large for accurate estimate)
HOST = '192.168.30.1' # The IP address of the server
PORT = 1738          # The port address
B = 8192 # [bytes] Message size;
amount of data to transfer per call
# build a message consisting of random digits (ASCII)
payload = ''
for n in range(B):
    payload = payload + str(random.choice(range(10)))
# Main loop
t0 = time.time()       # remember start time
for n in range(N):     # do this N times
    l = 0; #Setting length of received message = 0
    r = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    r.connect((HOST, PORT)) #Connecting to the server
    r.setblocking(1)
    r.settimeout(3) #Setting a timeout of 3s
    r.send(payload) # send the message
    data =''
    r.close() #Close connection
    #while (l<B): #This segment used only to check for packet losses
     # data = data + r.recv(B)      # see what the server comes back with
    # print str(len(data)) #Checking for message errors/losses
    # print '\n'
    #if data==payload:
     #  print str(n)+' success'
    #else:
     #  print str(n)+' fail'
      #if not data:
      #   break
      #print '\n'+'Received Length '+str(len(data))

#print data+'/'
t1 = time.time()       # remember end time
```

## A.2 "Tier-Zero" Interface

This section is referred to from Section 4.2. Here, we include the Python script for the Tier 0 interface (`mcs_tzi1.py`) .

```
# mcs_tzi1.py
# A. Srinivasan, Aug 7, 2009
# "Tier Zero" interface code for direct communication with subsystems
# Usage:
# $ python mcs_tzi1.py <DEST_NAME> <DEST_IP> <TX_PORT>
# <RX_PORT> <CMD> <REF> <DATA>
#or $ python mcs_tzi1.py <DEST_NAME> <DEST_IP> <TX_PORT>
# <RX_PORT> <CMD> <REF> -f <DATA_FILE>
#   <DEST_NAME>; e.g. SHL. (source name is automatically MCS)
#   <DEST_IP>; e.g., 127.0.0.1 for local loopback
#   <TX_PORT>; e.g., 1738
#   <RX_PORT>; e.g., 1739
#   <CMD>; e.g., PNG.  No error checking is done.
#   <REF>; e.g., 1361
#   <DATA> is whatever should go in the DATA field.
#    Enter as a string with single quotes.
#   <DATA_FILE> is the file that contains all
#    the data that needs to go in the DATA field, in any format
# Examples:
# $ python mcs_tzi1.py SHL 127.0.0.1 1738 1739 RPT 1361 'VERSION'
# $ python mcs_tzi1.py SHL 127.0.0.1 1738 1739 RPT 1361 -f data_field_contents.dat

import socket
import time
import datetime
import math
import string
import struct   # for packing of binary to/from strings
import sys

from optparse import OptionParser
parser = OptionParser()
parser.add_option("-f")
(options,args) = parser.parse_args()

if (len(sys.argv)<7):
```

```
    print 'More arguments expected. No action taken'
    exit()
DEST_NAME = sys.argv[1]
DEST_IP = sys.argv[2]
PORTT = int(sys.argv[3])# TX_PORT
PORTR = int(sys.argv[4]) # RX_PORT
cmd = sys.argv[5]
ref = sys.argv[6] # reference
if (len(sys.argv)>7):
    if (sys.argv[7] == '-f'):
        data_file = sys.argv[8]
        file_pointer = open(data_file,'rb')
        data = file_pointer.read()
        data_print = '' # data from a file not displayed

    else:
        data = sys.argv[7]
        data_print = data
else:
    data = ''
    data_print = data

B = 8192 # Buffer Size
#==============================
# Message Processing Subroutine
#==============================
def command(iref,cmd,data,data_print):
    # (in)  iref:    [int] message REFERENCE number to use
    # (in)  cmd:     [str] three-letter command; e.g., "PNG"
    # (in)  data:    [str] DATA field of message
    # (out) err:     [int] Error code: sum of the following:
    #                   0: no errors
    #                   1: (not currently used)
    #                   2: wrong "DESTINATION"
    #                   4: R-RESPONSE = "R" (command reject indicator) received
    # (out) summary: [str] R-SUMMARY (i.e., sender's MIB 1.1)
    # (out) rdata:   [str] Contents of DATA field with R-* fields stripped away
    # -------------------------
    # Set up sockets
    # -------------------------
    # Set up the receive socket for UDP
```

```
r = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
r.bind(('', PORTR))  # Accept connections from anywhere
r.setblocking(1)    # Blocking on this socket
r.settimeout(3)     # Set timeout to 3 s

# Set up transmit socket for UDP
t = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
t.connect((DEST_IP, PORTT))
# ----------------------
# Message Preparation
# ----------------------
# determine current time
dt = datetime.datetime.utcnow()
year        = dt.year
month       = dt.month
day         = dt.day
hour        = dt.hour
minute      = dt.minute
second      = dt.second
millisecond = dt.microsecond / 1000

# construct the MJD field
# adapted from http://paste.lisp.org/display/73536
# can check result using http://www.csgnetwork.com/julianmodifdateconv.html
a = (14 - month) // 12
y = year + 4800 - a
m = month + (12 * a) - 3
p = day + (((153 * m) + 2) // 5) + (365 * y)
q = (y // 4) - (y // 100) + (y // 400) - 32045
mjdi = int(math.floor( (p+q) - 2400000.5))
mjd = string.rjust(str(mjdi),6)
#print '#'+mjd+'#'
# construct the MPM field
mpmi = int(math.floor( (hour*3600 + minute*60 + second)*1000 + millisecond ) )
mpm = string.rjust(str(mpmi),9)
#print '#'+mpm+'#'

# Build the message
# Note we are just using a single,
#non-updating REFERENCE number in this case
message =           DEST_NAME
```

```
    message = message + 'MCS'
    message = message + cmd
    message = message + string.rjust(str(iref),9)
    message = message + string.rjust(str(len(data)),4)
    message = message + str(mjd)
    message = message + str(mpm) + '␣' # note trailing space
    message_print = message
    message = message + data
    for j in range(len(data_print)):
        if (ord(data_print[j])<32 and  ord(data_print[j])>126): #ascii values of non-printable characters
            data_print[j] = '␣' #removing non-printable characters
    if (len(data_print)>45): #truncating data displayed to 45 bytes
       data_print = data_print[:45]
    message_print = message_print + data_print
    # ------------------------
    # Sending and Receiving
    # ------------------------
    # Send it and wait for response
    t.send(message)             # send the message
    print message_print # say what was sent
    #response = ''
    response = r.recv(B)        # this blocks until the server responds
    response_data = response[38:]
    for j in range(len(response_data)):
        if (ord(response_data[j])<32 and  ord(response_data[j])>126): #ascii values of non-printable characters
            response_data[j] = '␣' #removing non-printable characters
    if (len(response_data)>45):
       response_data = response_data[:45]
    response_print = response[:38]+response_data
    print response_print
    r.close() # close receive socket
    t.close() # close transmit socket
# attempting to connect to the subsystem
command(ref,cmd,data,data_print)
```

## A.3 MCS ICD Compliance Check Software

This section is referred to from Section 4.3. We include the Python script for the MCS ICD Compliance Check Software (`mcs_mincompchk_1.py`).

```
# mcs_mincompchk_1.py —— A. Srinivasan — VT, 2009 Jul 31
# Script to ensure compliance of subsystem responses with MCS Common ICD
# usage:
#   $python mcs_mincompchk_1.py <subsystem_ID> <dest_ip> <transmit port> <receive port>
# Compliance checked with commands required by MCS Common ICD implemented on a subsytem
# This is the "client" side (MCS) and checks compliance of the "server"(susbystem) with the MCS ICD
# DEST_IP set to loopback by default.
# If the destination IP is not entered
# as a command line argument, the default value is used.
# The results of the compliance tests are written to a file 'MCS_mincomp_Report.txt'
# Minimal commands that are checked:
# —— PNG, RPT, SHT
# —— RPT checked with minimal subsystem (server) MIB labels
# —— Response times for PNG/RPT commands are also checked.

import socket
import time
import datetime
import math
import string
import struct   # for packing of binary to/from strings
import sys

DEST_IP = '127.0.0.1' # The default IP address of the server
if len(sys.argv)<4: # no command specified —— crash out
    print 'Proper␣usage␣is␣"mcs_mincompchk_1.py
␣<subsystem_ID>␣<dest_ip>␣<transmit_port>␣<receive_port>"
Only␣dest_ip␣is␣optional'
    print 'No␣action␣taken.'
    exit()
else:               # it's here —— use it
    DEST_NAME = string.strip(sys.argv[1])
# The 3—character MCS Common ICD designator for the server
    if (len(sys.argv)>4):
        PORTT = int(string.strip(sys.argv[3]))
# The transmit port address
        PORTR = int(string.strip(sys.argv[4]))
# The receive port address
        DEST_IP = string.strip(sys.argv[2])
    else:
        PORTT = int(string.strip(sys.argv[2]))
```

```
# The transmit port address
        PORTR = int(string.strip(sys.argv[3]))
# The receive port address

# Below are things that shouldn't be changed
B = 8192
# [bytes] Max message size
RPT_FILE = 'MCS_mincomp_Report.txt' # Name of file containing the report of the tests
#==============================
# Message Processing Subroutine
#==============================
def command(iref,cmd,data,resp_printer):
    # (in)  iref:    [int] message REFERENCE number to use
    # (in)  cmd:     [str] three-letter command; e.g., "PNG"
    # (in)  data:    [str] DATA field of message
    # (out) err:     [int] Error code: sum of the following:
    #                   0: no errors
    #                   1: (not currently used)
    #                   2: wrong "DESTINATION"
    #                   4: R-RESPONSE = "R" (command reject indicator) received
    # (out) summary: [str] R-SUMMARY (i.e., sender's MIB 1.1)
    # (out) rdata:   [str] Contents of DATA field with R-* fields stripped away
    # ------------------------
    # Set up sockets
    # ------------------------
    # Set up the receive socket for UDP
    r = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    r.bind(('', PORTR))  # Accept connections from anywhere
    r.setblocking(1)     # Blocking on this socket
    r.settimeout(3)      # Set timeout to 3 s

    # Set up transmit socket for UDP
    t = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    t.connect((DEST_IP, PORTT))
    # ----------------------
    # Message Preparation
    # ----------------------
    # determine current time
    dt = datetime.datetime.utcnow()
    year        = dt.year
    month       = dt.month
```

```
day         = dt.day
hour        = dt.hour
minute      = dt.minute
second      = dt.second
millisecond = dt.microsecond / 1000

# construct the MJD field
# adapted from http://paste.lisp.org/display/73536
# can check result using http://www.csgnetwork.com/julianmodifdateconv.html
a = (14 - month) // 12
y = year + 4800 - a
m = month + (12 * a) - 3
p = day + (((153 * m) + 2) // 5) + (365 * y)
q = (y // 4) - (y // 100) + (y // 400) - 32045
mjdi = int(math.floor( (p+q) - 2400000.5))
mjd = string.rjust(str(mjdi),6)
#print '#'+mjd+'#'
# construct the MPM field
mpmi = int(math.floor( (hour*3600 + minute*60 + second)*1000 + millisecond ) )
mpm = string.rjust(str(mpmi),9)
#print '#'+mpm+'#'

# Build the message
# Note we are just using a single, non-updating REFERENCE number in this case
message =           DEST_NAME
message = message + 'MCS'
message = message + cmd
message = message + string.rjust(str(iref),9)
message = message + string.rjust(str(len(data)),4)
message = message + str(mjd)
message = message + str(mpm) + '␣'  # note trailing space
message = message + data
#print '#'+payload+'#'
# ------------------------
# Sending and Receiving
# ------------------------
# Send it and wait for response
t0 = time.time()
t.send(message)            # send the message
# print 'sent> '+message+'|' # say what was sent
#response = ''
```

```
    response = r.recv(B)       # this blocks until the server responds
    t1 = time.time()
    r.close() # close receive socket
    t.close() # close transmit socket
    if (resp_printer):
       fp.writelines('sent> '+message+'|'+'\n')
       fp.writelines('recd> '+response+'|'+'\n')
    message = response        # the message
    # print 'recd> '+message+'|'
    t_diff = t1-t0 #time interval between sending the command and receiving the response from the subsystem
    # return
    return message,mjd,t_diff
#-----------------------------
#Message splitting subroutine
#-----------------------------
def split_string(smp_string):
    str_dest = string.strip(smp_string[:3])
    str_snd = string.strip(smp_string[3:6])
    str_cmd = string.strip(resp[6:9])
    str_ref = string.strip(resp[14:18])
    str_dt = string.strip(smp_string[23:28])
    str_acc = smp_string[38:39]
    str_summ = string.strip(smp_string[39:46])
    str_data = string.strip(smp_string[47:])
    return str_dest,str_snd,str_cmd,str_ref,str_dt,str_acc,str_summ,
str_data


# -------------------------------
# Begin tests
# -------------------------------
iref = 1391
fp = open(RPT_FILE,'w')
fp.writelines('STARTING TO TEST COMPLIANCE with MCS Common ICD... \n')
fp.writelines('\n')
fp.writelines('COMMANDS required by MCS Common ICD \n')
# Testing PNG
fp.writelines('\n')
fp.writelines('1. PNG Command \n')
final_Result = True # Assume unless proved otherwise
cmd = 'PNG'
resp, sent_date,t_int = command(iref,cmd,'',True)
```

```
#print '|||'+sent_date+'|||'
resp_dest,resp_snd,resp_cmd,resp_ref,resp_dt,resp_acc,resp_summ,resp_data = split_string(resp)

if (resp_dest=='MCS' and resp_snd==DEST_NAME and
resp_cmd==cmd and resp_dt==string.strip(sent_date)
and resp_ref==str(iref)):
   if (resp_acc == 'A'):
       if (resp_summ=='NORMAL' or resp_summ=='BOOTING'):
           fp.writelines('[PASS] \n')
       elif (resp_summ=='WARNING' or resp_summ=='ERROR'or resp_summ =='SHUTDWN'):
           fp.writelines('[PASS] \n')
       else:
           fp.writelines('**SUMMARY Field in response is incorrect\n')
           fp.writelines('[FAIL] \n')
           final_Result = False

   elif (resp_acc == 'R'):
       fp.writelines('[FAIL] \n')
       final_Result = False
   else:
       fp.writelines('**Command can be either accepted (A)
 or rejected (R).  '+'('+resp_acc+') '+'not understood\n')
       fp.writelines('[FAIL]\n')
       final_Result = False
else:
   fp.writelines('**Command returned with incorrect destination/sender/command name/date/reference \n')
   fp.writelines('[FAIL] \n')
   final_Result = False

labels = ['SUMMARY','INFO','LASTLOG','SUBSYSTEM','SERIALNO',
'VERSION','RANDOM']
# Testing with one invalid label — RANDOM
# Testing RPT
cmd = 'RPT'
fp.writelines('\n')
fp.writelines('RPT Command \n')
for i in range(len(labels)):
    fp.writelines('\n')
    if labels[i]=='RANDOM':
       fp.writelines(str(i+2)+'. With Invalid/Bogus Label '+labels[i]+'\n')
    else:
```

```
       fp.writelines(str(i+2)+'. With Label: '+labels[i]+'\n')
    resp, sent_date,t_int = command(iref,cmd,labels[i],True)  resp_dest,resp_snd,resp_cmd,resp_ref,resp_dt,
resp_acc,resp_summ,resp_data = split_string(resp)
    valid = True # Assume this unless it becomes false
    if (resp_acc=='A'):
        if (resp_dest=='MCS' and resp_snd==DEST_NAME
 and resp_cmd==cmd and resp_dt==string.strip(sent_date)
 and resp_ref==str(iref)):

            if labels[i]=='INFO' or labels[i]=='LASTLOG'
 or labels[i]=='VERSION':
                if (len(resp_data)>256):
                    valid = False
                    fp.writelines('** Length of the data
 field exceeds 256 bytes \n')

            if labels[i]=='SUBSYSTEM':
                if(resp_data!=DEST_NAME):
                  valid = False
                  fp.writelines('** Subsytem Entry incorrect.
 It should be '+DEST_NAME+'\n')

            if labels[i]=='SERIALNO':
                if(len(resp_data)>5):
                   valid = False
                   fp.writelines('** Length of the data
field exceeds 5 bytes \n')
            if (resp_summ!='NORMAL' and resp_summ!='BOOTING'
 and resp_summ!='WARNING'):
                if (resp_summ!='ERROR'
and resp_summ !='SHUTDWN'):
                   valid = False
                   fp.writelines('** R—SUMMARY field
invalid \n')
            if valid == False:
                fp.writelines('[FAIL]\n')
                final_Result = False

            else:
                if (labels[i]!='RANDOM'):
                   fp.writelines('[PASS]\n')
```

```
                else:
                    fp.writelines('**RPT passes through
for an invalid MIB Label. Unexpected behavior\n')
                    fp.writelines('[FAIL]\n')
                    final_Result = False
        else:
            fp.writelines('**Command returned with incorrect destination/sender/command_name/date/reference \n')
            fp.writelines('[FAIL]\n')
            final_Result = False
    elif (resp_acc == 'R'):
          if (resp_data!='Command not recognized'):
              if (labels[i]!='RANDOM' and resp_data == 'Invalid MIB label') :
                  fp.writelines('** The requested label
not found.Server MIB File does not contain miminal entries\n')
                  fp.writelines('[FAIL]\n')
                  final_Result = False
              elif (resp_data == 'Invalid MIB label'):
                  fp.writelines('The requested label not
found. Expected behavior. Looks good\n')
                  fp.writelines('[PASS]\n')
              else:
                  fp.writelines('** RPT command rejected.
Unexpected behavior\n')
                  fp.writelines('[FAIL]\n')
                  final_Result = False
          else:
              fp.writelines('** RPT command rejected.
Unexpected behavior\n')
              fp.writelines('[FAIL]\n')
              final_Result = False
    else:
         fp.writelines('**Command can be either accepted (A)
 or rejected (R).  '+'('+resp_acc+') '+'not understood\n')
         fp.writelines('[FAIL]\n')
         final_Result = False

# Testing with bogus/invalid command
cmd = 'RND'
fp.writelines('\n')
fp.writelines('9. Passing an invalid command - RND \n')
resp, sent_date,t_int = command(iref,cmd,'',True)
```

```
resp_dest,resp_snd,resp_cmd,resp_ref,resp_dt,
resp_acc,resp_summ,resp_data = split_string(resp)
if (resp_acc == 'A'):
    fp.writelines('** Invalid command accepted.
Faulty behavior\n')
    fp.writelines('[FAIL]\n')
    final_Result = False
elif (resp_acc == 'R'):
    if (resp_data == 'Command not recognized'):
       fp.writelines('[PASS] \n')
    else:
       fp.writelines('[FAIL] \n')
       final_Result = False
else:
    fp.writelines('**Command can be either accepted (A)
or rejected (R).  '+'('+resp_acc+') '+'not understood\n')
    fp.writelines('[FAIL] \n')
    final_Result = False

# Checking minimum and maximum times of reponse for PNG command
fp.writelines('\n')
fp.writelines('RESPONSE_TIMES \n')

fp.writelines('10. Running PNG 100 times\n')
ind = 0
resp_time = range(1,101)
a_cnt = 0 # To maintain acceptance (A) count
r_cnt = 0 # To maintain rejection (R) count
i_cnt = 0 # To maintain invalid response count
while (ind<100):
       resp, sent_date,t_int = command(iref,'PNG','',False)
       resp_dest,resp_snd,resp_cmd,resp_ref,resp_dt,
resp_acc,resp_summ,resp_data = split_string(resp)
       resp_time[ind] = t_int
       if resp_acc == 'A':
          a_cnt = a_cnt + 1
       elif resp_acc == 'R':
          r_cnt = r_cnt + 1
       else:
          i_cnt = i_cnt + 1
       ind = ind + 1
```

```
min_time = (min(resp_time))/pow(10,-3)
max_time = (max(resp_time))/pow(10,-3)
# print '#'+str(ind)+'#'
fp.writelines('Minimum response time = '+
str(min_time)+' ms'+'\n')
fp.writelines('Maximum response time = '+
str(max_time)+' ms'+'\n')
fp.writelines('Number of times PNG was accepted =
'+ str(a_cnt)+'\n')
fp.writelines('Number of times PNG was rejected =
'+ str(r_cnt)+'\n')
if (max_time<3000 and a_cnt == 100):
   fp.writelines('[PASS] \n')
else:
   fp.writelines('[FAIL] \n')
   final_Result = False
fp.writelines('\n')
fp.writelines('11. Running RPT with valid
arguments 100 times\n')
ind = 0
resp_time = range(1,101)
a_cnt = 0 # To maintain acceptance (A) count
r_cnt = 0 # To maintain rejection (R) count
i_cnt = 0 # To maintain invalid response count
argmnts = ['SUMMARY','VERSION','INFO','LASTLOG'] #
i=0
while (ind<100):
       if i>3:
          i=0
       resp, sent_date,t_int = command(iref,'RPT',
argmnts[i],False)
       i = i + 1
       resp_dest,resp_snd,resp_cmd,resp_ref,resp_dt,
resp_acc,resp_summ,resp_data = split_string(resp)
       resp_time[ind] = t_int
       if resp_acc == 'A':
          a_cnt = a_cnt + 1
       elif resp_acc == 'R':
          r_cnt = r_cnt + 1
       else:
          i_cnt = i_cnt + 1
```

```
       ind = ind + 1
min_time = min(resp_time)/pow(10,-3)
max_time = max(resp_time)/pow(10,-3)
# print '#'+str(ind)+'#'
fp.writelines('Minimum response time =
'+ str(min_time)+' ms'+'\n')
fp.writelines('Maximum response time =
'+ str(max_time)+' ms'+'\n')
fp.writelines('Number of times RPT was accepted =
'+ str(a_cnt)+'\n')
fp.writelines('Number of times RPT was rejected =
'+ str(r_cnt)+'\n')
if (max_time<3000 and a_cnt==100):
   fp.writelines('[PASS] \n')
else:
   fp.writelines('[FAIL] \n')
   final_Result = False
fp.writelines('\n')
# print 'Results of the Compliance tests written to "MCS_mincomp_Report.txt"\n'
fp.writelines('12. Running RPT with invalid
arguments 100 times\n')
ind = 0
resp_time = range(1,101)
a_cnt = 0 # To maintain acceptance (A) count
r_cnt = 0 # To maintain rejection (R) count
i_cnt = 0 # To maintain invalid response count
while (ind<100):
       resp, sent_date,t_int = command(iref,'RPT','RANDOM',False)
       resp_dest,resp_snd,resp_cmd,resp_ref,resp_dt,
resp_acc,resp_summ,resp_data = split_string(resp)
       resp_time[ind] = t_int
       if resp_acc == 'A':
          a_cnt = a_cnt + 1
       elif resp_acc == 'R':
          r_cnt = r_cnt + 1
       else:
          i_cnt = i_cnt + 1
       ind = ind + 1
min_time = min(resp_time)/pow(10,-3)
max_time = max(resp_time)/pow(10,-3)
# print '#'+str(ind)+'#'
```

```
fp.writelines('Minimum response time =
'+ str(min_time)+' ms'+'\n')
fp.writelines('Maximum response time =
'+ str(max_time)+' ms'+'\n')
fp.writelines('Number of times RPT was accepted =
'+ str(a_cnt)+'\n')
fp.writelines('Number of times RPT was rejected =
'+ str(r_cnt)+'\n')
if (max_time<3000 and r_cnt == 100):
   fp.writelines('[PASS] \n')
else:
   fp.writelines('[FAIL] \n')
   final_Result = False

# Testing SHT
cmd = 'SHT'
fp.writelines('\n')
fp.writelines('SHT Command \n')
# Testing SHT with no argument/SCRAM/RESTART
argmnts = ['RESTART','RANDOM','SCRAM','']
#Including a bogus argument
for i in range(len(argmnts)):
    fp.writelines('\n')
    fp.writelines(str(i+13)+'. Testing with argument:
'+argmnts[i]+'\n')
    resp, sent_date,t_int = command(iref,cmd,argmnts[i],True)
    resp_dest,resp_snd,resp_cmd,resp_ref,resp_dt,
resp_acc,resp_summ,resp_data = split_string(resp)
    valid = True # Assume this unless proved otherwise

    if (resp_acc == 'A'):
       if (resp_dest!='MCS'):
          valid = False
          fp.writelines('** Destination in the
response not correct\n')
       if (resp_ref!=str(iref)):
          valid = False
          fp.writelines('** Reference in the
response not correct\n')
       if (resp_snd!=DEST_NAME):
          valid = False
```

```
          fp.writelines('** Sender field in the
response not correct\n')
       if (resp_cmd!=cmd):
          valid = False
          fp.writelines('** Command field in the
response not correct\n')
       if (resp_dt!=string.strip(sent_date)):
          valid = False
          fp.writelines('** Response not received on the
same day it was sent\n')
       if (resp_summ!='SHUTDWN'):
          valid = False
          fp.writelines('** Summary on the MIB file incorrect\n')
       if valid:
          fp.writelines('[PASS]\n')
       else:
          fp.writelines('[FAIL]\n')
          final_Result = False
    elif (resp_acc == 'R'):
          if (resp_data!='Command not recognized'):
              if (argmnts[i]!='RANDOM'):
                  fp.writelines('** Valid arguments with
SHT rejected. Faulty behavior\n')
                  fp.writelines('[FAIL]\n')
                  final_Result = False
              elif (resp_data == 'Invalid extra arguments'):
                  fp.writelines('Rejected due to incorrect argument. Expected.\n')
                  fp.writelines('[PASS]\n')
              else:
                  fp.writelines('** SHT command rejected. Unexpected behavior\n')
                  fp.writelines('[FAIL]\n')
                  final_Result = False
          else:
              fp.writelines('** SHT command rejected.
Unexpected behavior\n')
              fp.writelines('[FAIL]\n')
              final_Result = False
    else:
       fp.writelines('**Command can be either accepted (A) or rejected
                    (R).'+'('+resp_acc+') '+'not understood\n')
       fp.writelines('[FAIL]\n')
```

```
        final_Result = False
fp.writelines('\n')
fp.writelines('TESTING_COMPLETED...\n')
if (final_Result):
    fp.writelines('PASS\n')
else:
    fp.writelines('FAIL\n')

fp.writelines('\n')
fp.close()
```

## A.4 MCS Common ICD Emulation Software for SHL

This appendix is referred to from Section 4.4. We include two python scripts here: 1) for the SHL Emulator (`mch_shl_1.py`) and 2) for the Client for the SHL Emulator (`mch_cl_1.py`).

### A.4.1 SHL Emulator

```
# mch_shl_1.py — 2009 Jul 14
# A. Srinivasan
# usage:
#    $ python mch_shl_1.py
# Crude "stub"/"alpha" implementation of the MCS Common
# ICD for controlled subsystems
# —— Minimum MIB: MCS—RESERVED section SHL—POWER
# and SHL—ECS
# —— PNG supported
# —— RPT supported, but is limited to 1 index at a time (no branches)
# —— SHT supported, but doesn't actually do anything
# apart from setting SUMMARY to SHUTDWN  But, it will return correctly.
# —— INI supported, overwrites the MIB with the new values
# —— TMP supported, overwrites the MIB with the new values
# —— DIF supported, overwrites the MIB with the new values
# —— PWR supported, overwrites the MIB with the new values
# This is the "server" side; the other end is MCS ("client"), which runs mch_cl_1.py
# This code uses loopback IP for DEST_IP. To run this on a
# network, change it to the client's IP.
# This code runs forever —— use CTRL—C or whatever
```

```
# to crash out when done.
# About the MIB:
# -- The MIB is implemented as a text file: "MIB_shl_1.txt"
# -- The file is provided by the subsystem software,
# and is read when this program starts
# -- The format is: one row per MIB entry; each row
# is index (space) label (space) value
# -- Upon receipt of INI,TMP,DIF or PWR command
#this file is overwritten to update MIB entries
# -- Upon receipt of a PNG or RPT command, the
# file is re-read to get values for the response
# Some notes:
# -- Intended to be compliant with SHL Common ICD v C, except as noted above
import socket
import time
import datetime
import math
import string
import struct   # for packing of binary to/from strings
DEST_IP = '127.0.0.1' # The IP address of the client (MCS)
PORTT = 1739             # The transmit port address
PORTR = 1738             # The receive port address
# Below are things that shouldn't be changed
B = 8192                    # [bytes] Max message size
MIB_FILE = 'MIB_shl_1.txt'  # Name of the MIB File
# ------------------------------
# Load the MIB
# ------------------------------
print 'Loading MIB...'
# Read the file containing the MIB into a string
fp = open(MIB_FILE,'r')
file = fp.read()
fp.close()
#print file
# Parse the string into MIB entries
mi = [] # this becomes a list of MIB labels
me = [] # this becomes a list of MIB entries (data)
mind = [] # this becomes a list of MIB indices
line = file.splitlines()   # split file into lines
for i in range(len(line)):
    column = string.split(line[i],' ',2)
```

```
# split line into columns
    mind.append(column[0])
    mi.append(column[1])
    me.append(string.strip(column[2]))
print 'I␣am␣'+me[3]+'.'
# ——————————————————————
# Set up comms
# ——————————————————————
# Set up the receive socket for UDP
r = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
r.bind(('',PORTR))  # Accept connections from anywhere
r.setblocking(1)   # Blocking on this sock
# Set up the transmit socket for UDP
t = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
t.connect((DEST_IP,PORTT))
print 'Running...'
initialised = False
#To keep a check on whether the SHL is being initialised
while 1:
    payload = r.recv(B)  # wait for something to appear
    # Say what was received
    print 'rcvd>␣'+payload+'|'
    # ——————————————————————
    # Analyzing received command
    # ——————————————————————
    # The default is that a response is not necessary:
    bRespond = False
    # Now the possibilities are:
    # (1) The message is not for us; then
no response should be made
    # (2) The message is for us but is not a PNG, RPT, or SHT message.  In this case, send "reject" response
    # (3) The message is for us and is a PNG, RPT, or SHT message.  In this case, respond appropriately
    destination = payload[:3]
    sender      = payload[3:6]
    command     = payload[6:9]
    reference   = int(payload[9:18])
    datalen     = int(payload[18:22])
    mjd         = int(payload[22:28])
    mpm         = int(payload[28:37])
    data        = payload[38:38+datalen]
    #print 'DESTINATION: |'+destination+'|'
```

```
    #print 'SENDER:      |'+sender+'|'
    #print 'TYPE:        |'+command+'|'
    #print 'REFERENCE: ', reference
    #print 'DATALEN:   ', datalen
    #print 'MJD:       ', mjd
    #print 'MPM:       ', mpm
    #print 'DATA: |'+data+'|'
    if (destination==me[3]) or (destination=='ALL'):
# comparing to MIB entry 1.4, "SUBSYSTEM"
        bRespond = True
        # ——— Reread the MIB ———
        # Read the file containing the MIB into a string
        fp = open(MIB_FILE,'r')
        file = fp.read()
        fp.close()
        # Parse the string into MIB entries
        mi = [] # this becomes a list of MIB labels
        me = [] # this becomes a list of MIB entries (data)
        mind =[] # this becomes a list of MIB indices
        line = file.splitlines()   # split file into lines
        for i in range(len(line)):
            column = string.split(line[i],' ',2)
# split line into columns
            mi.append(column[1])
            me.append(string.strip(column[2]))
            mind.append(column[0])
            response = 'R'+string.rjust(str(me[0]),7)+
' Command not recognized'
# use this until we find otherwise
        if command=='PNG':
            if initialised:
               response = 'A'+string.rjust(str(me[0]),7)
            else:
               response = 'R'+string.rjust(str(me[0]),7)+' Initialise the SHL first'
        if command=='RPT':
            if initialised:
               response = 'R'+string.rjust(str(me[0]),7)+' Invalid MIB label'
# use this until we find otherwise
               mib_label = string.strip(data)

               for i in range(len(mi)):
```

```
                    if mi[i]==mib_label:
                        response = 'A'+string.rjust(str(me[0]),7)+' '+me[i]
            else:
               response = 'R'+string.rjust(str(me[0]),7)+' Initialise the SHL first'
        if command=='SHT':
            if initialised:
               me[0] = 'SHUTDWN'
               response = 'A'+string.rjust(str(me[0]),7)
# use this until we find otherwis
               arg = string.strip(data)
               fp = open(MIB_FILE,'w')
               for i in range(len(line)):
                        fp.writelines(mind[i]+' '+
mi[i]+' '+me[i]+'\n')
               fp.close()
             # verify arguments
               while len(arg)>0:
                  args = string.split(arg,' ',1)
                  args[0] = string.strip(args[0])
                  if (not(args[0]=='SCRAM') and not(args[0]=='RESTART')):
                      response = 'R'+string.rjust
(str(me[0]),7)+' Invalid extra arguments'
                  if len(args)>1:
                      arg = args[1]
                  else:
                      arg = ''
            else:
               response = 'R'+string.rjust(str(me[0]),7)+' Initialise the SHL first'
        if  command=='INI':
             # response = 'A'+string.rjust(str(me[0]),7)
# use this unless we find otherwise
             arg = string.strip(data)
             # print '|'+arg+'|'
             if len(arg)>0:
                args = string.split(arg,'&',3)
                # print '|'+str(len(args))+'|'
                if (len(args)<3):
                    response = 'R' + string.rjust(str(me[0]),7)+' Invalid number of arguments'

                elif (len(args[0])!=5 or len(args[1])!=3
or len(args[2])!=6):
```

```
                    response = 'R' +  string.rjust(str(me[0]),7)+ '␣Invalid␣argument␣length'
            else:
                set_point = args[0]
                diff_point = args[1]
                racks_install = args[2]
                min_tmp = 60 # Min valid temperature
                max_tmp = 110 # Max valid temperature
                check = min_tmp
                while (check<=max_tmp):
                   if (float(set_point) == check):
                       break
                   else:
                       check = check+0.5 #Valid increment
                if (check>max_tmp):
                   response = 'R' + string.rjust
(str(me[0]),7) + '␣Set—Point␣should␣be␣between
60␣F␣and␣110␣F␣in␣increments␣of␣0.5␣F'
                elif (min_tmp>float(set_point)):
                   response = 'R' +
string.rjust(str(me[0]),7)+
'␣Set—Point␣should␣not␣be␣lesser␣than␣60F'
                else:
                   inc = 0.5
                   while (inc<=5):
# Checks for validity of differential point
                      if (float(diff_point)==inc):
                          break
                      else:
                          inc=inc+0.5

                   if (inc>5):
                      response = 'R' + string.rjust(str(me[0]),7)+'␣Differential␣Point␣should␣be
in␣increments␣of␣0.5␣between␣0.5␣and␣5'

                   elif (float(diff_point)<0.5):
                      response = 'R' + string.rjust
(str(me[0]),7)+'␣Differential␣should␣not␣be␣lesser␣than␣0.5'

                   else:
                      initialised = True
# INI command passes
```

```
                            fp = open(MIB_FILE,'w')
                            ri = list(racks_install)
                            init = False
#flag to check if MIB has already been
#appended with new values
                            for i in range(len(mi)):
                                if (mi[i] == 'SUMMARY'):
                                    me[i] = 'NORMAL'
                                    fp.writelines(mind[i]+
' '+mi[i]+' '+me[i]+'\n')
                                elif (mi[i] == 'INFO' or
mi[i] == 'LASTLOG' or mi[i] == 'SUBSYSTEM'):
                                    fp.writelines(line[i]+'\n')
                                elif (mi[i] == 'SERIALNO'
 or mi[i] == 'VERSION'):
                                    fp.writelines(line[i]+'\n')
                            if not init:
                                 init = True # MIB being updated
                                 for j in range(len(ri)):
                                       app_index = '2.'+str(j+1)+'.1'
                                       app_label = 'PORTS-AVAILABLE-R'+str(j+1)
                                       if (ri[j] == '1'):
                                           no_ports = '50'
# Set number of ports available
# to 50 when initialised
                                           fp.writelines(app_index+' '+app_label+' '+no_ports+'\n')
                                           k = 1
                                           default_val = 'OFF'
# setting all ports under the 'set' rack to OFF
                                           while (k<=50):
# initialising all 50 ports for the 'set' rack
                                               new_index = '2.'+str(j+1)+'.2'+'.'+str(k)
                                               new_label =
'PWR-R'+str(j+1)+'-'+str(k)
                                               fp.writelines(new_index+' '+new_label+' '+default_val+'\n')
                                               k = k+1
                                           current = 0
# setting a default value to 0
                                           fp.writelines('2.'+str(j+1)+'.3 '+'CURRENT-R'+str(j+1)+' '+str(current)
                                       else:
                                           no_ports = '0'
```

```
# Number of ports made zero when rack is not set
                                        fp.writelines(app_index+' '+app_label+' '+no_ports+'\n')
                            if init:
                                sp_index = '3.1'
# MIB index for Set-Point
                                dp_index = '3.2'
# MIB index for Differential
                                temp_index = '3.3'
# MIB index for Temperature
                                fp.writelines(sp_index
+' '+'SET-POINT'+' '+str(set_point)+'\n')
                                fp.writelines(dp_index
+' '+'DIFFERENTIAL'+' '+str(diff_point)+'\n')
                                fp.writelines(temp_index+' '+'TEMPERATURE'+' '+str(set_point))
# Setting temperature to set_point
                                response = 'A'+
string.rjust(str(me[0]),7)
                            fp.close()


        if command == 'TMP':
           if initialised:
              arg = string.strip(data)
              if (len(arg)!=5):
                  response = 'R' + string.rjust(str(me[0]),7)
 + ' Invalid argument length'
              else:
                  min_tmp = 60 # Min valid temperature
                  max_tmp = 110 # Max valid temperature
                  check = min_tmp
                  int_arg = float(arg)
                  while (check<=max_tmp):
                      if (float(arg) == check):
                         break
                      else:
                         check = check+0.5 # Valid increments
                  if (check>max_tmp):
                     response = 'R' + string.rjust(str(me[0]),7) + ' Set-Point should be between 60 F and 110 F i
                  elif (min_tmp>float(arg)):
                     response = 'R' + string.rjust(str(me[0]),7)+' Set-Point should not be lesser than 60F'
                  else:
                     fp = open(MIB_FILE,'w')
```

```
                    response = 'A' + string.rjust(str(me[0]),7)
                    for i in range(len(mi)):
                        if (mi[i]=='SET_POINT'):
                           fp.writelines(mind[i]+' '+'SET_POINT'+' '+str(arg)+'\n')
                        elif (mi[i]=='TEMPERATURE'):
                           fp.writelines(mind[i]+' '+'TEMPERATURE'+' '+str(arg)+'\n')
                        else:
                           fp.writelines(line[i]+'\n')
                    fp.close()
            else:
               response = 'R'+  string.rjust(str(me[0]),7)
+' Initialise the SHL first'

        if command == 'DIF':
            if initialised:
               arg = string.strip(data)
               if (len(arg)!=3):
                   response = 'R' + string.rjust(str(me[0]),7)
 + ' Invalid argument length'
               else:
                   min_diff = 0.5
# Min Valid differential set_point
                   max_diff = 5
# Max valid differential set_point
                   check = min_diff
                   while (check<=max_diff):
                       if (float(arg) == check):
                          break
                       else:
                          check = check+0.5
                   if (check>max_diff):
                      response = 'R' + string.rjust(str(me[0]),7) + ' Differential Set_Point should be between
0.5 F and 5 F in increments of 0.5 F'
                   elif (min_diff>float(arg)):
                      response = 'R' +
string.rjust(str(me[0]),7)+
' Differential Set_Point should not be lesser than 0.5 F'

                   else:
                      fp = open(MIB_FILE,'w')
                      response = 'A' + string.rjust(str(me[0]),7)
```

```
                    for i in range(len(mi)):
                        if (mi[i]=='DIFFERENTIAL'):
                           fp.writelines(mind[i]+' '+'DIFFERENTIAL'+' '+str(arg)+'\n')
                        else:
                           fp.writelines(line[i]+'\n')
                    fp.close()
            else:
               response = 'R'+  string.rjust(str(me[0]),7)
+' Initialise the SHL first'


        if command == 'PWR':
           if initialised:
              response = 'A' +string.rjust(str(me[0]),7)
#assume this unless there is some error
              arg = data
              if (len(arg)>6):
                 response = 'R' + string.rjust(str(me[0]),7)
+ ' Invalid arguments | Larger than 6 bytes'
              else:
                 rack = arg[:1]
                 port = arg[1:3]
                 control = arg[3:]
                 if (int(rack)>6 or int(rack)==0):
                    response = 'R' + string.rjust(str(me[0]),7) + ' Invalid rack number'
                 else:
                    rack_index = 'PORTS—AVAILABLE—R'+rack
                    for i in range(len(mi)):
                        if mi[i]==rack_index:
                           if (int(port)>int(me[i]) or int(port)==0):
                              response = 'R' + string.rjust(str(me[0]),7) + ' Invalid port number'
                           else:
                              if (control!='ON ' and control!='OFF'):
                                 response = 'R' + string.rjust(str(me[0]),7) + ' Invalid control argument'
                              else:
                                 port_index =
'PWR—R'+rack+'—'+str(int(port))
                                 for j in range(i,len(mi)):
                                     if mi[j]==port_index:
                                        me[j] = control
                    ports_on = 0
                    for i in range(len(mi)):
```

```
                    if ((mi[i])[:6]==('PWR-R'+rack)):
                        if (me[i]=='ON' or me[i]=='ON '):
                            ports_on = ports_on+1
                            #print '$$$'+str(ports_on)
                    if (mi[i]==('CURRENT-R'+rack)):
                        me[i] = str(100*pow(10,-3)*ports_on)     # Setting a value of 100mA per port enabled
                fp = open(MIB_FILE,'w')
                for i in range(len(line)):
                    fp.writelines(mind[i]+' '+mi[i]+' '+me[i]+'\n')
                fp.close()
        else:
            response = 'R' +string.rjust(str(me[0]),7) + ' Initialise the SHL first'
# ----------------------
# Message Preparation
# ----------------------
payload = '(nothing)' # default payload
if bRespond:
    # determine current time
    dt = datetime.datetime.utcnow()
    year        = dt.year
    month       = dt.month
    day         = dt.day
    hour        = dt.hour
    minute      = dt.minute
    second      = dt.second
    millisecond = dt.microsecond / 1000
    # compute MJD
    # adapted from http://paste.lisp.org/display/73536
    # can check result using http://www.csgnetwork.com/julianmodifdateconv.html
    a = (14 - month) // 12
    y = year + 4800 - a
    m = month + (12 * a) - 3
    p = day + (((153 * m) + 2) // 5) + (365 * y)
    q = (y // 4) - (y // 100) + (y // 400) - 32045
    mjdi = int(math.floor( (p+q) - 2400000.5))
    mjd = string.rjust(str(mjdi),6)
    #print '#'+mjd+'#'
    # compute MPM
    mpmi = int(math.floor( (hour*3600 + minute*60 + second)*1000 + millisecond ))
    mpm = string.rjust(str(mpmi),9)
    #print '#'+mpm+'#'
```

```
        # Build the payload
        # Note we are just using a single, non-updating REFERENCE number in this case
        payload = 'MCS'+me[3]+command+string.rjust(str(reference),9)
        payload = payload + string.rjust(str(len(response)),4)+str(mjd)+str(mpm)+'␣'
        payload = payload + response
        t.send(payload)      # send it
    print 'sent>␣'+payload+'|' # say what was sent (exclude checksum)
# never get here, but what the heck.
s.close()
t.close()
```

### A.4.2 Client for the SHL Emulator

```
# mch_cl_1.py — A. Srinivasan, 2009 Jul 14
# Usage:
#    $ python mch_cl_1.py <CMD> <ARGS>
#    <CMD> = three-letter command TYPE (e.g., PNG)
#    <ARGS> = space-delimited list of arguments; depends on command
# Crude "stub"/"alpha" implementation of the MCS Common ICD from the client (MCS) side
# —— PNG
# —— RPT (but only one MIB entry at a time; not branches)
# —— SHT
# —— INI
# —— TMP
# —— DIF
# —— PWR
# The other end is the controlled subsystem (SHL or "server"); runs mch_shl_1.py
# This code uses the loopback IP for DEST_IP. To run it on a network, change it to the server's IP address.
# This is a "one shot" code —— it does it once and then quits.
# When run, it reads MIB_cl_1.txt, which represents the MIB in the following format:
#    index (space) label (space) value <CR>
# The command provides information about what is happening as it goes.
# The command overwrites MIB_mincl_1.txt with an updated MIB, if appropriate.
# Some notes:
# —— Note receive operation blocks, so execution will hang if server doesn't respond
# —— Intended to be compliant with MCS Common ICD v.1.0, except as noted above
# —— REFERENCE is currently fixed always to be 1391
# —— This is not production code.
import socket
import time
import datetime
```

```
import math
import string
import struct   # for packing of binary to/from strings
import sys

DEST_IP = '127.0.0.1' # The IP address of the server
DEST_NAME = 'SHL'       # The 3-character MCS Common ICD designator for the server
PORTT = 1738             # The port address
PORTR = 1739             # The port address
# Below are things that shouldn't be changed
B = 8192                    # [bytes] Max message size
MIB_FILE = 'MIB_cl_1.txt' # Name of file containing MIB
#==============================
# Message Processing Subroutine
#==============================
def command(iref,cmd,data):
    # (in)  iref:    [int] message REFERENCE number to use
    # (in)  cmd:     [str] three-letter command; e.g., "PNG"
    # (in)  data:    [str] DATA field of message
    # (out) err:     [int] Error code: sum of the following:
    #                   0: no errors
    #                   1: (not currently used)
    #                   2: wrong "DESTINATION"
    #                   4: R-RESPONSE = "R" (command reject indicator) received
    # (out) summary: [str] R-SUMMARY (i.e., sender's MIB 1.1)
    # (out) rdata:   [str] Contents of DATA field with R-* fields stripped away
    # ------------------------
    # Set up sockets
    # ------------------------
    # Set up the receive socket for UDP
    r = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    r.bind(('', PORTR))  # Accept connections from anywhere
    r.setblocking(1)    # Blocking on this socket
    r.settimeout(3)     # Set timeout to 3 s
    # Set up transmit socket for UDP
    t = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    t.connect((DEST_IP, PORTT))
    # ----------------------
    # Message Preparation
    # ----------------------
    # determine current time
```

```
dt = datetime.datetime.utcnow()
year        = dt.year
month       = dt.month
day         = dt.day
hour        = dt.hour
minute      = dt.minute
second      = dt.second
millisecond = dt.microsecond / 1000
# construct the MJD field
# adapted from http://paste.lisp.org/display/73536
# can check result using http://www.csgnetwork.com/julianmodifdateconv.html
a = (14 - month) // 12
y = year + 4800 - a
m = month + (12 * a) - 3
p = day + (((153 * m) + 2) // 5) + (365 * y)
q = (y // 4) - (y // 100) + (y // 400) - 32045
mjdi = int(math.floor( (p+q) - 2400000.5))
mjd = string.rjust(str(mjdi),6)
#print '#'+mjd+'#'

# construct the MPM field
mpmi = int(math.floor( (hour*3600 + minute*60 + second)*1000 + millisecond ) )
mpm = string.rjust(str(mpmi),9)
#print '#'+mpm+'#'
# Build the message
# Note we are just using a single, non-updating REFERENCE number in this case
message =           DEST_NAME
message = message + 'MCS'
message = message + cmd
message = message + string.rjust(str(iref),9)
message = message + string.rjust(str(len(data)),4)
message = message + str(mjd)
message = message + str(mpm) + '␣'  # note trailing space
message = message + data
#print '#'+payload+'#'
# -------------------------
# Sending and Receiving
# -------------------------
# Send it and wait for response
t.send(message)            # send the message
print 'sent>␣'+message+'|' # say what was sent
```

```
    #response = ''
    response = r.recv(B)       # this blocks until the server responds
    r.close() # close receive socket
    t.close() # close transmit socket
    # ------------------------
    # Analyzing response
    # ------------------------
    # parse message into payload and sent checksum
    message = response        # the message
    # say what was received
    print 'rcvd> '+message+'|'
    bDest     = False     # assume this until noted otherwise
    bResponse = False     # assume this until noted otherwise
    summary   = 'UNKNOWN' # use this until we have a proper value
    rdata     = ''        # use this until we have a proper value
    # check for correct recipient (DESTINATION)
    destination = message[:3]
    #print 'DESTINATION=|'+destination+'|'
    if destination=='MCS':
        bDest = True
        # check for command rejection (R-RESPONSE)
        rresponse = message[38:39]
        #print 'R-RESPONSE=|'+rresponse+'|'
        if rresponse=='A':
            bResponse = True
            # get R-SUMMARY and any remaining data
            summary = string.strip(message[39:46])
            rdata = string.strip(message[47:])
    # generate error code
    err = 0
    if not(bDest):
       err = err+2
    if not(bResponse):
       err = err+4
    # return
    return err, summary, rdata
#==================
# Main Program
#==================
# Check for required command line argument <CMD>
if len(sys.argv)<2: # no command specified -- crash out
```

```
    print 'Proper usage is "mch_minc_1.py <CMD> <ARGS>.  <CMD> missing.'
    print 'No action taken.'
    exit()
else:               # it's here -- use it
    cmd = string.strip(sys.argv[1])
    #print '|'+mib_label+'|'
iref = 1391  # hardwired for now
eErr = 0 # 0 = OK.  >0 represents an error code.
         # 1 = checksum error in received message
#----------------
# Load the MIB
#----------------
# Read the MIB
fp = open(MIB_FILE,'r')
file = fp.read()
fp.close()
#print file
# Initialize MIB variables
mib_index = []
mib_label = []
mib_value = []
line = file.splitlines() # split file into lines
nmib = len(line)         # number of MIB entries
for i in range(nmib):
    column = string.split(line[i],' ',2) # split into columns
    mib_index.append(string.strip(column[0]))
    mib_label.append(column[1])
    mib_value.append(string.strip(column[2])


#--------------------
# Execute command
#--------------------
if cmd=='PNG':
    err, summary, rdata = command(iref,cmd,'')
    print 'Error Code (0 is good): '+str(err)
    mib_value[0] = summary # this works because summary=='UNKNOWN' if there    is an error
elif cmd=='RPT':
    # Check for required command line argument <CMD>
    if len(sys.argv)<3: # no arguments specified -- crash out
        print 'Proper usage is "mch_minc_1.py RPT <MIB_LABEL>.  <MIB_LABEL> missing.'
        print 'No action taken.'
```

```
        exit()
    else:               # it's here -- use it
        arg = string.strip(sys.argv[2])
        # print '|'+arg+'|'
    # Is this really a MIB label?
    imib = -1
    for i in range(nmib):
        if arg==mib_label[i]:
            imib = i
    if imib==-1:
        print 'MIB_LABEL_|'+arg+'|_not_found.'
        print 'No_action_taken.'
        exit()
    err, summary, rdata = command(iref,cmd,arg)
    print 'Error_Code_(0_is_good):_'+str(err)
    mib_value[0] = summary # this works because summary=='UNKNOWN' if there is an error
    if err==0:
        mib_value[imib] = rdata
elif cmd=='SHT':
    # Check for additional command line arguments
    arg=''
    for i in range(2,len(sys.argv)):
       if sys.argv[i]=='SCRAM':
           arg = arg+'SCRAM_'
       elif sys.argv[i]=='RESTART':
           arg = arg+'RESTART_'
       else:
           print 'I_do_not_understand_|'+sys.argv[i]+'|'
           print 'No_action_taken.'
           exit()
    arg = string.strip(arg)
    # print '|'+arg+'|'
    err, summary, rdata = command(iref,cmd,arg)
    print 'Error_Code_(0_is_good):_'+str(err)
    mib_value[0] = summary # this works because summary=='UNKNOWN' if there is an error
elif cmd=='INI':
    # Check for additional command line arguments
    arg=''
    if (len(sys.argv)<5):
        print 'Proper_usage_is_"mod_client.py_INI
<Set-Point>_<Diff_Set_Point>_<Racks_Installed>"
```

```
|␣Enter␣appropriate␣number␣of␣arguments'
        print 'No␣action␣taken'
        exit()
    else:
        arg1 = string.strip(sys.argv[2])
        if (len(arg1)>5):
             print 'The␣Set-Point␣should␣not␣be␣larger␣than␣5␣bytes'
             print 'No␣action␣taken'
             exit()
        if (len(arg1)<5):
            arg1 = string.rjust(arg1,5,'0')
        arg2 = string.strip(sys.argv[3])
        if (len(arg2)>3):
             print 'The␣Differential␣Set-Point␣should
not␣be␣larger␣than␣3␣bytes|
Should␣be␣between␣0.5␣F␣and␣5␣F'
             print 'No␣action␣taken'
             exit()
        if (len(arg2)<3):
             arg2 = string.rjust(arg2,3,'0')
        arg3 = string.strip(sys.argv[4])

        if (len(arg3)!=6):
             print '␣"No:␣of␣racks␣installed"␣argument
format␣incorrect.␣Enter␣6␣bytes'
             print 'No␣action␣taken'
             exit()
        arg = arg + arg1 +'&'+ arg2 + '&'+arg3
    err, summary, rdata = command(iref,cmd,arg)
    print 'Error␣Code␣(0␣is␣good):␣'+str(err)
    mib_value[0] = summary
elif cmd == 'TMP':
     arg = ''
     if (len(sys.argv)<3):
          print 'Proper␣usage␣is␣"mod_client.py
TMP␣<Set-Point>"␣|
Enter␣appropriate␣number␣of␣arguments'
          print 'No␣action␣taken'
          exit()
     else:
          arg1 = string.strip(sys.argv[2])
```

```
        if (len(arg1)>5):
            print 'The Set-Point should not be larger than 5 bytes'
            print 'No action taken'
            exit()
        if (len(arg1)<5):
            arg1 = string.rjust(arg1,5,'0')
        arg = arg + arg1
        # print '|'+arg+'|'
        err, summary, rdata = command(iref,cmd,arg)
        print 'Error Code (0 is good): '+str(err)
        mib_value[0] = summary
elif cmd == 'DIF':
    arg = ''
    if (len(sys.argv)<3):
        print 'Proper usage is "mod_client.py
DIF <Differential Set-Point>" |
Enter appropriate number of arguments'
        print 'No action taken'
        exit()
    else:
        arg1 = string.strip(sys.argv[2])
        if (len(arg1)>3):
            print 'The Differential Set-Point
should not be larger than 3 bytes|
Should be between 0.5 F and 5 F'
            print 'No action taken'
            exit()
        if (len(arg1)<3):
            arg1 = string.rjust(arg1,3,'0')
        arg = arg + arg1
        # print '|'+arg+'|'
        err, summary, rdata = command(iref,cmd,arg)
        print 'Error Code (0 is good): '+str(err)
        mib_value[0] = summary
elif cmd == 'PWR':
    arg =''
    if (len(sys.argv)<5):
        print 'Proper usage is "mod_client.py
PWR <Rack Number> <Port Number> <Control(On/OFF)>" |
Enter appropriate    number of arguments'
        print 'No action taken'
```

```
        exit()
    rack = string.strip(sys.argv[2])
    port = string.strip(sys.argv[3])
    control = string.strip(sys.argv[4])
    if (len(rack) == 1):
        if (int(rack)>=1 and int(rack)<=6):
            arg = arg + rack
        else:
            print 'Invalid rack entry |
Enter values between 1 and 6'
            print 'No action taken'
            exit()
    else:
         print 'The rack entry should not be larger
than a byte'
         print 'No action taken'
         exit()
    if (len(port)>2):
        print 'The port entry should not be larger
than 2 bytes | Should be between 1 and 50'
        print 'No action taken'
        exit()
    else:
        check = float(port)
        if (check == int(port)):
           if (int(port)>=1 and int(port)<=50):
              if (len(port)==1):
                  port = string.rjust(port,2,'0')
              arg = arg + port
           else:
              print 'Invalid port entry |
Should be between 1 and 50'
              print 'No action taken'
              exit()
        else:
           print 'Enter only valid integer values for the ports'
           print 'No action taken'
           exit()

    if (len(control)>3):
        print 'Invalid Control argument'
```

```
            print 'No action taken'
            exit()
        if (control == 'ON' or control == 'OFF'):
            if (control == 'ON'):
                control = 'ON '
            arg = arg + control
            # print '#'+arg+'#'
        else:
             print 'Invalid control argument'
             print 'No action taken'
             exit()
        # print '|'+arg+'|'

        err, summary, rdata = command(iref,cmd,arg)
        print 'Error Code (0 is good): '+str(err)
        mib_value[0] = summary
else:
    print 'I do not recognize the command <'+cmd+'>.'
    print 'No action taken.'
    exit()
#----------------
# Update MIB
#----------------
# Overwrite the MIB file
if eErr==0:
    fp= open(MIB_FILE,'w')
    for i in range(nmib):
        fp.write(mib_index[i]+' '+mib_label[i]+' '+mib_value[i]+'\n')
    fp.close()
```

# Bibliography


[1] S. Ellingson *et al.*, "The Long Wavelength Array," *Proc. IEEE*, Vol. 97, No. 8, pp. 1421-1430, Aug 2009.

[2] L. D'Addario and R. Navarro, "Preliminary Design for the Digital Processing Subsystem of a Long Wavelength Array Station," Long Wavelength Array Memo 154, Feb 11, 2009, available online: http://www.phys.unm.edu/~lwa/memos.

[3] J. Craig, "Long Wavelength Array Station Architecture Ver. 2," Long Wavelength Array Memo Series, No. 161, Feb 24, 2009, available online: http://www.ece.vt.edu/swe/lwa/.

[4] C. Wolfe, S. Ellingson and C. Patterson, Interface Control Document for Monitor and Control System Data Recorder (MCS-DR), Long Wavelength Array Engineering Memo, MCS0025, Ver. 1.0, Mar 22, 2010, available online: http://www.ece.vt.edu/swe/lwavt/.

[5] S. Ellingson, "MCS Subsystem Description", Long Wavelength Array Engineering Memo MCS0004, Ver. 2, Feb 23, 2009, available online: http://www.ece.vt.edu/swe/lwavt/.

[6] A. Cohen and N. Paravastu, "Probing the Ionosphere with the LWA by Rapid Cycling of Celestial Radio Emitters", Long Wavelength Array Memo Series, No. 128, Mar 12, 2008, available online: http://www.phys.unm.edu/~lwa/memos.

[7] S. Ellingson and A. Cohen, "Beam Dwell and Repointing Time Requirements Derived From Memo 128 Considerations", Long Wavelength Array Memo Series, No. 146, Dec 7, 2008, available online: http://www.ece.vt.edu/swe/lwa/.

[8] S. Ellingson, "MCS Architecture", Long Wavelength Array Engineering Memo MCS0007, Ver. 4, Nov 7, 2009, available online: http://www.ece.vt.edu/swe/lwavt/.

[9] S. Ellingson, "MCS Common ICD", Long Wavelength Array Engineering Memo MCS0005, Ver. 1.0, Apr 04, 2009, available online: http://www.ece.vt.edu/swe/lwavt/.

[10] J. Winett, "RFC147 - Definition of a socket", May 07, 1971.

[11] S. Ellingson *et al.*, "Monitoring and Control System (MCS) Critical Design Review", Long Wavelength Array Engineering Memo, MCS0024, Nov 20, 2009, available online: http://www.ece.vt.edu/swe/lwavt/.

[12] S. Ellingson, "MCS/Scheduler Software Version 0.5 (pre-alpha)", Long Wavelength Array Engineering Memo MCS0028, June 1, 2010, available online: http://www.ece.vt.edu/swe/lwavt/.

[13] H. Zimmermann, "OSI Reference Model - The ISO Model for Architecture for Open Systems Interconnection", *IEEE Transactions on Communications*, Vol. Com-28, No. 4, Apr 1980.

[14] IEEE Std 802.3 1985, "IEEE standards for local area networks: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications".

[15] J. Postel, "RFC791 - Internet Protocol", Sep 1981.

[16] J. Postel, "RFC793 - Transmission Control Protocol", 1981.

[17] J. Postel, "RFC768 - User Datagram Protocol", 1980.

[18] J. Mogul, "RFC950 - Internet Standard Subnetting Procedure", Aug 1985.

[19] J. Postel, "RFC790 - Assigned Numbers", Sep 1981.

[20] C. Hornig, "RFC894 - A Standard for the Transmission of IP Datagrams over Ethernet Networks", Apr 1984.

[21] V. Jacobson, R. Braden and D. Borman, "RFC1323 - TCP Extensions for High Performance", May 1992.

[22] Tsung-i Huang and J. Subhlok, "Fast Pattern-based Throughput Prediction for TCP Bulk Transfers", Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 1, 2005.

[23] Qi He *et al.*, "Exploiting the Predictability of TCPs Steady-State Behavior to speed up Network Simulation", *In Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 101-108, Oct 2002.

[24] S. Ellingson, "Beam Dwell and Repointing", Long Wavelength Array Memo Series, No. 144, Nov 25, 2008, available online: http://www.ece.vt.edu/swe/lwa/.

[25] M. Soriano, "Interface Control Document for Digital Processing Subsystem Ver G", Long Wavelength Array Internal Report, Apr 7, 2009.

[26] N. Gift and J. Jones, "Python for Unix and Linux System Administration", O'Reilly, 2008.

[27] A. Srinivasan and S. Ellingson, "Python Code for Direct Communication with Subsystems", Long Wavelength Array Engineering Memo MCS0015, Aug 7, 2009, available online: http://www.ece.vt.edu/swe/lwavt/.

[28] A. Srinivasan and S. Ellingson, "MCS ICD Compliance Check Software", Long Wavelength Array Engineering Memo MCS0013, Jul 31, 2009, available online: http://www.ece.vt.edu/swe/lwavt/.

[29] A. Srinivasan and S. Ellingson, "MCS Common ICD Emulation Software for SHL", Long Wavelength Array Engineering Memo MCS0012, Jul 19, 2009, available online: http://www.ece.vt.edu/swe/lwavt/.

[30] J. Craig, "Interface Control Document for Shelter (SHL) Ver C", Long Wavelength Array Engineering Memo, Jul 8, 2009.