

Duiker: A data acquisition software suite for
remotely deployed Q330 digitizers

Sam Irvine
sirvine@cs.ucla.edu

November 26, 2006

Contents

1	Introduction	4
1.1	Description	4
1.2	Motivation	4
1.3	System Components	4
2	Architecture	5
2.1	Data Acquisition	5
2.2	Data Transport	6
2.3	Data Bundle Acknowledgement and Removal	6
2.4	Bundle Conversion	8
3	Implementation	9
3.1	Data Acquisition	9
3.1.1	Q330 Network Protocol Overview	9
3.1.2	Data Acquisition Program Flow of Control	10
3.1.3	Authentication	10
3.1.4	Registration and Packet Acknowledgement	11
3.1.5	Status Information Queries and Reporting	11
3.1.6	File/Module Descriptions	13
3.2	Data Bundle Transport	15
3.2.1	Overview	15
3.3	Data Bundle Acknowledgement	16
3.3.1	File Acknowledgement	16
3.4	Data Bundle Conversion	17
3.4.1	Bundle Contents	17
3.4.2	Data Decoding Algorithm	18
4	Configuration and Usage	20
4.1	Duiker Acquisition Binary	20
4.1.1	Installation	20
4.1.2	Configuration	20
4.1.3	Usage	23
4.2	File Mover	24
4.2.1	Installation	24
4.2.2	Configuration	24

4.2.3	Usage	25
4.3	Data Sink File Remover	25
4.3.1	Installation	25
4.3.2	Configuration	26
4.3.3	Usage	26
4.4	Bundle Decoder	26
4.4.1	Installation	26
4.4.2	Configuration	27
4.4.3	Usage	27
4.5	GUI Decoder Manager	27
4.5.1	GUI mode operation	27
5	Extending Duiker	29
5.1	Configuration and Calibration by the Acquisition Binary	29
5.2	Real time access to data	29
5.3	Multiple frequency outputs per channel	29
5.4	200Hz sampling rates	30

1 Introduction

1.1 Description

Duiker is a set of data acquisition tools designed to capture data from a Quanterra Q330TM Seismic Digitizer, push captured data through a multi-hop network towards a sink node and process captured data for analysis and archiving at the sink node. The tools are designed to run on embedded Stargate/ARM Linux installations and i386 Linux installations with a very small memory and CPU footprint (640K memory and < 1 % CPU on a 400MHz Xscale processor).

1.2 Motivation

Duiker came into existence because of cost, efficiency and usability issues.

1.3 System Components

Duiker is divided into 4 primary components. The first component is a data bundle acquisition client that runs on a remote microserver attached to a Q330. Its purpose is to connect to a Q330, siphon off data and store the collected data to a secondary storage mechanism. The next component is a data transport server that pushes data "bundles" through a statically configured network towards a data sink node. As data is pushed through the network a backup copy resides at the data source until it is actively acknowledged. A mobile agent that deploys from the sink node provides this acknowledgement and cleans up extra copies of a bundle that may persist in the network. Finally, data is converted to miniseed and archived at a data processing node at the final hop in the network.

The data acquisition and transport components reside on microservers deployed remotely with a Q330 to collect data. The data bundle conversion component runs on a the network sink node point for all data in the network. Finally, the data bundle acknowledgement agents run on both the Stargate nodes and the data sink.

2 Architecture

The Duiker architecture is broken up into 4 subsystems. They include data bundle acquisition, data bundle transport, data bundle acknowledgement and data bundle conversion.

2.1 Data Acquisition

The Duiker data acquisition program is designed to run on a microserver class embedded node. It connects over Ethernet to a Q330 A-D converter and siphons data off using a proprietary UDP/IP protocol. Data that has been removed from the Q330 is stored locally in a "bundle." Each bundle contains a collection of state information, configuration information and data. The exact contents and format are specified in the implementation?? section. Bundles are packaged into files based on the time span of the data they represent. For example, they can be configured to store N minutes of data. Bundles are stored locally as files on the microserver's secondary storage device, typically a Compact Flash card or Microdrive.

Since the Q330 is a network enabled device an obvious question is raised with respect to the need for Duiker. After all, if the Q330 has can communicate over Ethernet why not simply have it transmit data to a sink node? The answer to this question is simple: In order to siphon data off of the Q330 an end-to-end connection must be made with a client program. However, for remote nodes operating over a 20 hop ad-hoc wireless connection the likelihood of a good end-to-end link is very low. This type of configuration is exactly what Duiker was designed for.

Furthermore, even with an end-to-end ad-hoc wireless link established, latency can grow very large which in turn reduces the throughput of the connection. The Q330 has a very limited data buffer. At most, the Q330 can buffer data for a few hours under peaceful conditions in which the digitizer does not output large wave forms. If the end to end throughput is to low, then the data buffer will overflow and data will be lost. The Duiker acquisition program does away with the need for an end-to-end connection by collecting data at the source and aggregating it into bundles for transport.

Duiker also periodically collects meta data relating to the health of the Q330 and the local node. Node temperature, GPS coordinates, disk usage, GPS time and battery voltage are all collected and reported back to the sink node using the bundle transport program.

2.2 Data Transport

Data transport is not handled with an end-to-end mechanism as is typically found in Internet applications. Duiker is designed to run on microserver class machines in remote locations. These microservers may have limited bandwidth and frequent periods of network isolation. To account for this the transport mechanism uses a hop-by-hop technique. Data bundles are pushed from one node to the next as complete files. Once an entire bundle has been pushed from one node to the next it can then be forwarded to node in the same fashion.

The advantage of sending bundles from node to node is that an end-to-end connection between the source and the sink does not need to exist in order for data transmission to achieve forward progress. If one link is down because of a power outage or other transient failure data can be forwarded up to that node. Once the link is re-established data can proceed down the line towards the sink node.

This was an expected scenario in the original deployment conditions Duiker was designed for. The most likely error condition was expected to be power outage due to the obscuring of a nodes solar cells during intense cloud cover. If a given node lost power due to a storm it would come online and begin forwarding data again.

The data transport mechanism currently only supports static topology configurations. Each node knows only its next hop in the chain towards the sink. A dynamic configuration utility would be of great value, however it was beyond the scope of what the data transport software was originally intended for.

2.3 Data Bundle Acknowledgement and Removal

When a data source pushes a data bundle forward through the data transport network it retains an original copy of the data. This only occurs at the node that produced the data and not at subsequent hops in the network. The data

transport functionality described thus far does not provide a means to delete old bundles from their originating source. This is a necessary requirement. Original data copies will build up and eventually overflow the secondary storage capacity of the source nodes. For long running remote deployments it may be logistically infeasible to physically service nodes before they fill up. To account for this problem a mobile agent is used to remove old files from remote nodes.

At the sink node each time a data bundle is received it undergoes an integrity check by computing an MD5 of the data and comparing it with the stored MD5 in the bundle. If the MD5s match, the sink node stores the bundle name to a file. Periodically, the sink node packages this list of received bundles with a mobile agent. The agent is then copied over to the first hop in the data transport topology and is instructed to execute. When an agent finishes removing files and copying itself to neighbor nodes it then terminates itself. In the case where a neighbor node is not currently available, the agent still terminates. The file removal system operates under best effort principles and does not attempt to achieve consistency. It should be noted that currently a protocol is being added to allow the network to achieve soft state consistency. Files that have not been deleted after N days are retransmitted through the network. If these files reach the sink node they will be re-acknowledged and deleted by a new agent sometime later. This implementation is currently not yet completed.

The mobile agent carries the static topology it intends to traverse with it. At any given node it can compute its next hops from a mapping stored internally. Once the agent has been pushed forward to a next-hop, it compares any bundles stored on the local node to its list of collected bundle names. If the local node is the origination point of a bundle that has been received by the sink the agent deletes the bundle from the local node. After deleting all files that have been properly received at the sink the agent moves on to its next set of hops and deletes itself from the current node.

A mobile agent based approach was chosen because of its flexibility. The original deployment of the Duiker system did not include a mechanism for removing old files in the network. By using a mobile agent, it was not necessary to upgrade nodes running in the field with new software.

2.4 Bundle Conversion

The process of data conversion occurs at a data sink node. Bundles are deposited into a directory that are periodically checked by the conversion software. For each bundle in the directory specified for conversion, the conversion software checks its validity using the stored MD5 sum, unbundles the data and converts it to a Miniseed file. This file is then moved to an archive directory. If an error occurs during the conversion process then the file is archived to an error directory.

3 Implementation

3.1 Data Acquisition

The data acquisition program is designed as a simple state machine. At system startup configuration information is gathered from a configuration file. After configuration information has been processed the acquisition program authenticates itself with the Q330 and registers as a receiver for data on one of four data channels. The Q330 then proceeds to send data packets to the acquisition program.

3.1.1 Q330 Network Protocol Overview

The Q330 network protocol sandwiches transport and application layer functionality into one layer. Each packet contains packet ordering information, data fragments and fragment re-assembly information.

Transport layer information allows for out of order packet delivery with selective retransmission of missing packets. Each packet contains a sequence number and an ack number. Packets are acknowledged with an ACK packet that contains a bitmap representing packets received and a sequence number representing the minimum packet number being acknowledged by the bitmap. The bitmap is 128 bits in length: the maximum window size the Q330 uses to transmit packets in parallel.

Application layer functionality is integrated into the transport layer. The basic unit of data in the protocol is called a "blockette". Blockettes are both fixed and variable length and conform to one of five structure definitions. Blockettes can be fragmented across multiple packets, which actively occurs when the MTU of MAC layer is less than the size of the data packet.

Flow control is achieved by a simple rate limiting scheme. This is because Duiker was designed to run on a LAN connection to the Q330. The rate of data collection is not typically high enough to warrant TCP like flow control over a lan situation (most of the time it is less than 10kbps). Furthermore, the data collection rate regulates the transmission rate. If the collection rate is above the maximum transmission rate of the network then the queue will be unstable

and overflow the Q330's data buffer, rendering the system useless. The only scenario where flow control is necessary is connecting remotely to a Q330 that has buffered a significant amount of data. Since the data will be transmitted in an extremely bursty manner, it may be possible to lose a significant number of packets in these bursts. When the packet loss rate becomes greater than 10%, the acquisition program will throttle back the Q330's transmission rate to 50Kbps by setting a rate limiting parameter in the acknowledgement packets.

3.1.2 Data Acquisition Program Flow of Control

The data acquisition program executes the following steps during normal operation

1. The acquisition program authenticates itself with the Q330.
2. The acquisition program registers itself with the Q330 to receive data and listens for data packets over the network. Each packet received is stored in a bundle file.
3. Periodically, the acquisition program queries the Q330 for GPS, temperature and time-stamp status information.
4. Every N minutes, the acquisition program packages up the data received so far into a bundle. A new bundle is then created and subsequently received data is added to this bundle.

3.1.3 Authentication

Authentication is handled using a shared secret key algorithm. The algorithm proceeds as follows:

1. The client requests authentication from a Q330 by sending a request packet. The request packet contains the serial number of the Q330 the client wishes to communicate with. The packet is sent to one of 5 ports on the Q330, depending on which service the acquisition program is authenticating with. For instance, if the acquisition program wants to communicate with a configuration port it sends the packet to the base port specified in the Q330 configuration. If the acquisition program wants to communicate with data ports 1-4 it sends the packet to $\text{base_port} + 2 * (\text{data_port_number} - 1)$.

2. The Q330 responds with a challenge value (aka nonce) and a registration number.
3. An MD5 hash is created as an ASCII string of the following values (in order).
 - (a) The 64 bit challenge value received, formatted as a 16 character ASCII string (using uppercase digits).
 - (b) The 32 bit IP address of the Q330, formatted as an 8 character ASCII string (using uppercase digits).
 - (c) The 16 bit port number of the Q330, formatted as a 4 character ASCII string (using uppercase digits).
 - (d) The 16 bit registration number provided by the Q330 from the request, formatted as an 4 character uppercase ASCII hex number.
 - (e) The 64 bit shared key value, formatted as a 16 character ASCII string (using uppercase digits).
 - (f) The 16 character uppercase, ASCII representation of the serial number of the Q330.
 - (g) The 64 bit random nonce reply, formatted as a 16 character ASCII uppercase hex string.

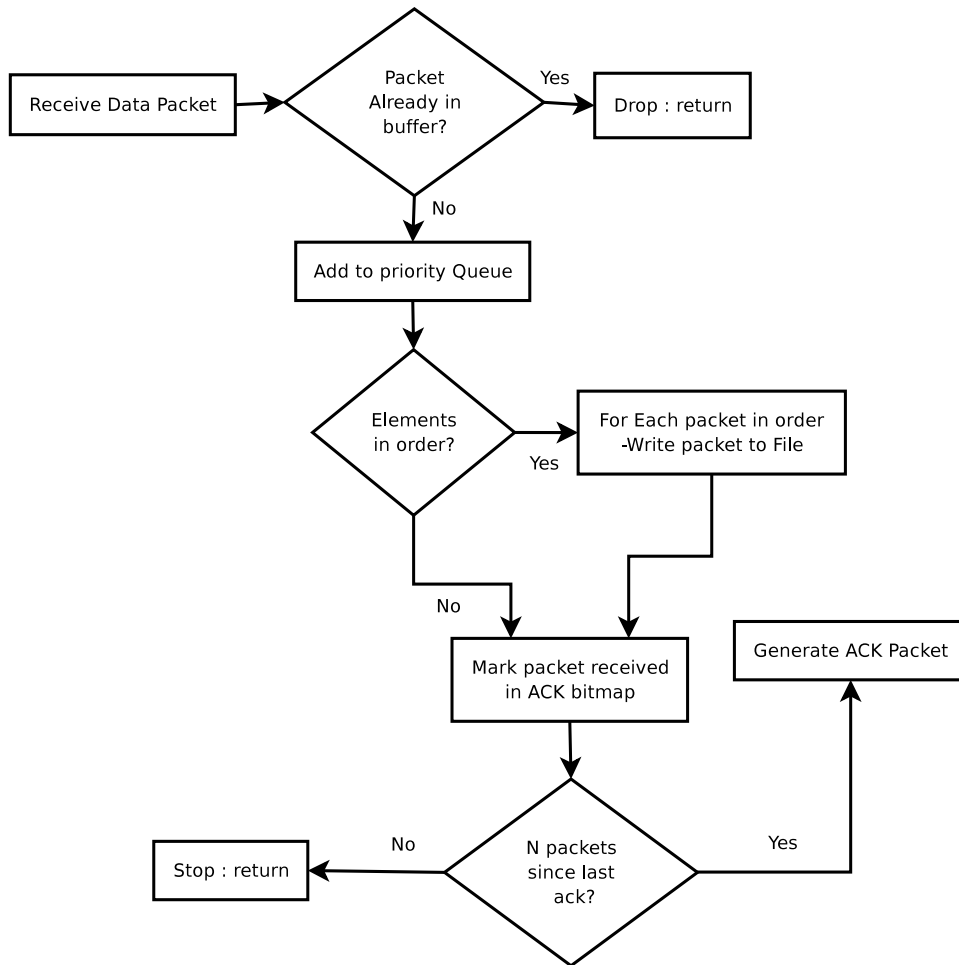
3.1.4 Registration and Packet Acknowledgement

Once the data acquisition program has authenticated itself with the Q330 it sends a Data open packet to the Q330. This data open packet instructs the Q330 to begin transmitting data for the channel specified during registration. Once this channel has been open, Duiker monitors it to ensure data is being delivered. If no data has been delivered for 2 minutes, the data acquisition program attempts to reopen the channel.

Packets are acknowledged using a cumulative, selective ack mechanism. Diagram 3.1.4 depicts the algorithm used to generate ack packets and write data packets to the bundle output.

3.1.5 Status Information Queries and Reporting

The data acquisition program ignores status information fed through the data streams. Instead, it polls the Q330 periodically for the information required.



The following information is captured by diker from the Q330.

1. GPS Status and coordinates (if available)
2. System temperature status
3. Temperature of the Q330
4. Input voltage level to the Q330

To query for these parameters the data acquisition program sends a `C1_RQSTAT` packet to the Q330 with the bitmap set for global, gps and temperature information.

3.1.6 File/Module Descriptions

- *acquire_main.c* This file contains the main function for the data acquisition program. It calls the initial configuration routines, sets up the call-back handlers for packet/timer events and starts the Duiker acquisition process.
- *blockettes.h* This module defines blockette types. Each type is generically named, since any given type/format can be interpreted in many ways. It also defines a `blockette_reader_t` structure and operations that can be performed on the `blockette_reader`. The `blockette_reader` provides sequential access to blockettes from a raw packet file.
- *bundle_processor.h* This module contains functions for processing bundles, converting them from their raw format to minised.
- *bundle_processor_main.c* This module contains the main function for the `bundle_processor` binary. It encapsulates runtime parameters and sets up a bundle to call the `bundle_processor.h` functions.
- *bundle_producer.h* This module contains functions for packaging Q330 packets into a bundle.
- *bundle_reader.h* This module contains functions for reading a bundle. The `bundle_processor` modules depend on this functionality to extract information from each bundle.

- *challenge.h* This module contains functions for processing an authentication challenge from a Q330.
- *configuration.h* This module contains functions and structures for the main configuration information for Duiker. Each configuration parameter for the `/opt/duiker/bin/duiker.conf` is defined in this module.
- *confuse.h* This module has been stripped from another open source project called libconfuse. It is a configuration library and is used to read in the configuration from a file. See `confuse.h` for more information on libconfuse.
- *crc.h* This module contains functions that implement the Q330 CRC computation on little endian architectures.
- *link_structures.h* This module contains structures that link packet reception call-back functions to the file descriptor they call select on.
- *md5.h* This module implements the MD5 hash function. It was written by someone else. See `md5.h` for more information on its implementation and copyright.
- *packet_buffer.h* This module implements a packet buffer on the Q330 to ensure that data is output in order. The packet buffer stores packets received by the Q330 in an ordered queue. As long as packets are received in order they are flushed immediately. However, if a packet is missed, packets after that packet are buffered until the missed packet is retransmitted by the Q330. All in order packets are then flushed to the packet bundle.
- *packet_handler.h* This module implements all of the packet reception call-back functions.
- *packets.h* This module defines all of the different packet types used by duiker to communicate with the Q330.
- *start_event.h* This module defines a start event that kicks off Duiker's execution.
- *steim_decoder.h* This module defines functions for decoding Steim2A and Steim2 data and converting it into 1 second time-spans.

- *system_state.h* This module defines the structure of the system state as well as common system functions.
- *timeout_handler.h* This module defines the actions that occur when a timeout occurs on any timer function.

3.2 Data Bundle Transport

3.2.1 Overview

The data bundle transport program is named `file_mover`. It is located in the `/opt/duiker/bin` directory of a stargate that duiker has been deployed on. It operates as a daemon that polls a message and data directory. When bundles are available to be forwarded, the daemon forwards these bundles to its configured next-hop using SCP (secure copy).

The `file_mover.conf` configuration file specifies two directories that are monitored for data to transport. The first is the *message_queue*. The *message_queue* parameter determines the location where system messages are forwarded to. The next is the *data_queue*. The *data_queue* determines the location where data bundles are forwarded to. *message_queue* files are given priority over *data_queue* files. Messages are given priority over data because they report the system status of the deployed node. In addition, most messages are only reported when problem events occur and thus it is important that these messages pass through with priority. Finally, messages are orders of magnitude smaller than bundles and thus take significantly less time to transmit forward.

Pseudo code for priority polling process is given below

The following function returns the name of the first file in the directory specified.

```
char * firstFile(char * dirName);
```

The following structure stores the next hop, message queue and bundle queue parameters for the currently executing file mover.

```
struct Configuration {
    char nextHopAddress[16]; // ex: 169.232.144.211\0
    char messageQueue[64]; // ex: /home/watchdog_logs
    char bundleQueue[64]; // ex: /opt/duiker/file_mover
}
```

The following function takes a given file and forwards it to the next hop. After forwarding the file, it deletes it from the current node. To forward the file, it forks a child process and calls `execl` with `'/usr/bin/scp'` as the process designated to execute.

```
void forwardFile(char * fileName, char * ipAddress);
```

The following function combines all of the previous functions to monitor the local message and bundle queues and forward any data they receive. If no files exist to forward the process sleeps for 1 second.

```
monitor(Configuration c) {
    while ( true ) {
        select( c.queueFileDescriptors );
        if (filesExist(c.messageQueue)) {
            forwardFile(firstFile(c.messageQueue),c.nextHopAddress);
            continue;
        }
        if (filesExist(c.bundleQueue)) {
            forwardFile(firstFile(c.bundleQueue),c.nextHopAddress);
            continue;
        }
    }
}
```

3.3 Data Bundle Acknowledgement

3.3.1 File Acknowledgement

Each bundle received at a sink node must be acknowledged. In order to properly acknowledge a bundle it first must undergo an MD5 check. The MD5 of the `< timestamp >.tar.gz` is stored internally as `< timestamp >.md5`. The MD5 represents the original MD5 value of the `< timestamp >.packets` file that is also stored inside of the bundle.

After a bundle passes the MD5 check it is safe to generate a deletion command. This is done by storing the names of bundles to be deleted and periodically deploying a mobile agent to actively seek these files out and remove

them. The mobile agent is implemented as a set of bash scripts. Each agent is a collection of files that move across the network as a tar archive.

When an agent is active on a node its first goal is to remove any files resident on that node that match its list of files to delete. After deleting these files, the agent attempts to copy itself to neighbor nodes listed in a topology map that travels with the agent. After copying itself successfully to a neighbor, the agent extracts itself remotely from the archive and creates a soft link from `/etc/cron.minute` to the new copy of the agent on the remote machine. The agent then exits, having completed its useful work.

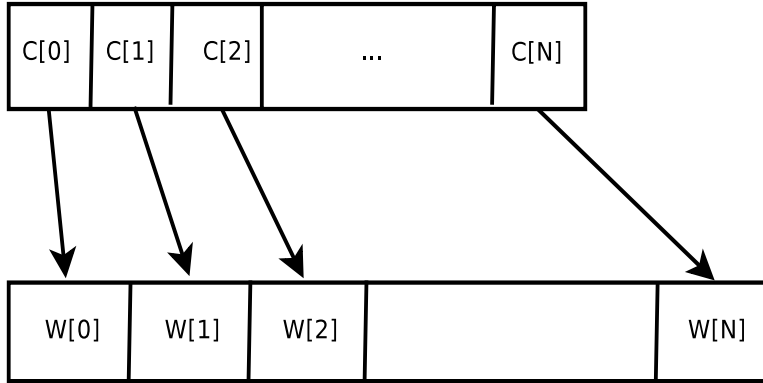
If an agent encounters an error its default behavior is to terminate without forwarding itself. This is to prevent a mis-configured agent from moving forward in the network.

3.4 Data Bundle Conversion

3.4.1 Bundle Contents

A bundle consists of a tar archive that contains the following files

1. `< timestamp > .gps` This file stores a lots of GPS reading values stored for the given bundle. One GPS values is recorded every minute to this file when Duiker is running.
2. `< timestamp > .state` This file stores state information from the Q330. Most of the internal state information is not necessary for packet decoding. However, a frequency map stored in the state file assists in computing the frequency of each output channel from the data file.
3. `< timestamp > .config` This file stores information about the configuration state of Duiker at the time when this bundle was collected.
4. `< timestamp > .md5` This file stores the MD5 checksum value of the `.packets` file.
5. `< timestamp > .packets` This file stores the actual data from the Q330's data channel. The encoding protocol is described in the Q330 reference manual (ref-2)



6. $\langle timestamp \rangle .version$ This file stores the CVS tag of the version of Duiker that created this file. This is critical to ensure the proper bundle decoder is used.

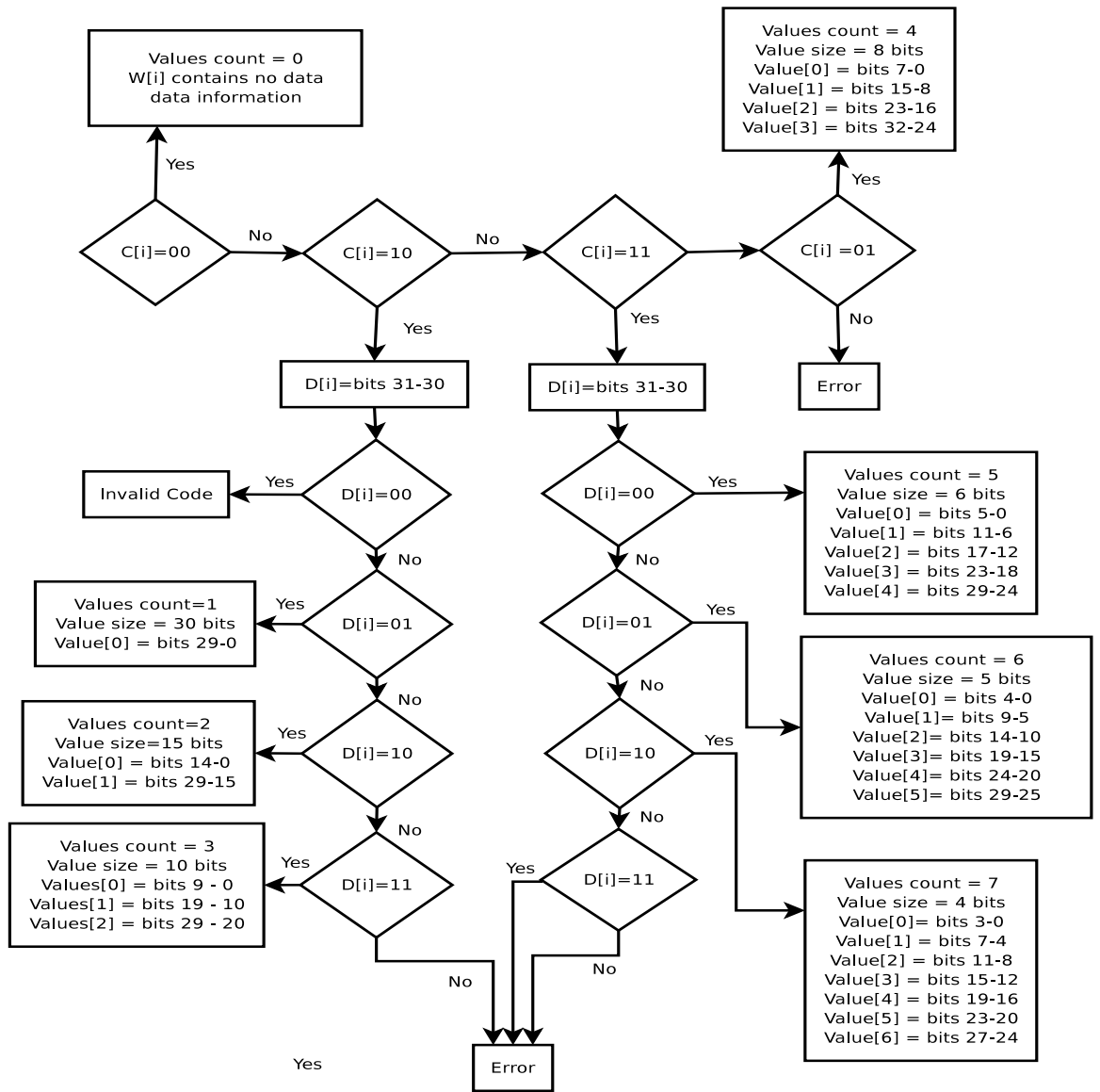
3.4.2 Data Decoding Algorithm

Data is stored in the $\langle timestamp \rangle .packets$ file and is encoded in a Steim2A format. Steim 2A encodes data using an initial value X , an array C of n compression codes and an array W of n compression values. Each compression code C_i corresponds to the compressed value W_i . The compression code C_i is 2 bits in length and is used to determine the compressed values stored in W_i .

To obtain the array of compression codes C , it is first necessary to compute n . n is not explicitly specified in the protocol, and must be obtained by using the following computation $(packet_length - offset_to_data)/4$

Once n is obtained, C corresponds to the linear ordering of bytes starting at the offset to the compression map and terminating at $2 \times n + offset$ bits from the start of the compression map (figure 3.4.2).

Figure 3.4.2 describes the algorithm that is used to decode data. The resultant output values each represent a 24 bit difference sample. Passed at the start of each Steim 2A block is an initial value. Each difference is added to the previous sum of differences to give the actual data point at that instance.



4 Configuration and Usage

This section describes the installation, configuration and usage of the various components of the Duiker seismic acquisition system.

4.1 Duiker Acquisition Binary

4.1.1 Installation

The Duiker acquisition installation package is currently only available for stargates using the CENS-seismic stargate filesystem. Do not attempt to install the Duiker binary on any other system unless one has been provided to you specifically for your architecture. Because stargates typically have a very small flash-drive, it is necessary to use a larger compact flash card to store Duiker data. This drive must be mounted to the /opt directory of the root file system. To format and mount a CF card using the Seismic stargate build, do the following:

- Insert the flash card into the CF card slot
- Execute the following commands:

```
shell>mkreiserfs /dev/hda
shell>mount /dev/hda -t reiserfs /opt
```

- Modify your /etc/fstab to reflect this mount point

To install Duiker on a Stargate obtain a copy of the duiker ipk release you wish to install; for example, duiker-07-21-2005_arm.ipk. Move the ipk file to the stargate you want to install duiker on and type

```
shell> ipk install duiker-07-21-2005_arm.ipk
```

Duiker should now be installed on the stargate.

4.1.2 Configuration

Configuration information is located in the duiker.conf file under /opt/duiker/bin. This configuration information is used to connect to a pre-configured Q330. Duiker itself does not configure any part of the Q330. To configure the Q330, it is necessary to use a program such as Willard (provided by Quanterra). The configuration file is a set of parameter value pairs of the form

<parameter>=<value>

The following are parameters used to configure the Duiker client

- *rdd_directory* is the directory RDD will read from. RDD is a program authored by Andrew Parker (aparker@gmail.com) for reliably moving bundles from a source to a sink.
- *server_ip* is the IP address of the Q330 this instance of Duiker intends to communicate with. Any valid Internet address of the form xxx.yyy.zzz.www is legal.
- *server_base_port_number* The base port number the Q330 uses to communicate with. The Q330 default value is 5330. Legal values include integers in the range of 1 to 65535 inclusive.
- *server_logical_data_port_number* The logical data port number this instance of Duiker will use to communicate with the Q330 over. The Q330 has 4 logical data ports, numbering 1,2,3 and 4.
- *q330_timeout* The timeout period after which a command retransmission will occur. The timeout parameter is specified in milliseconds.
- *serial_number* The serial number of the Q330 this instance of duiker will communicate with. The serial number is a 16 digit hex string specified in lower case hex digits (e.g. 0-9a-f).
- *authcode* The authentication code of the Q330 this instance of duiker will communicate with. The authentication code is a 16 digit hex string specified in lower case hex digits (e.g. 0-9a-f).
- *version* The Q330 protocol version this instance of Duiker will use to communicate with the Q330. Currently only protocol version 2 is supported.
- *transmission_retries* The number of command packet retransmissions to attempt after a timeout has occurred before giving up.
- *transmission_timeout* The timeout period after which a packet is retransmitted.
- *client_ip* The IP address of the local machine that this instance of duiker will use to communicate with the Q330 over.

- *client_configuration_port* The port number of the local machine that this instance of duiker will use to communicate with the Q330's configuration port over.
- *client_data_port* The port number of the local machine this instance of duiker will use to communicate with the Q330's data port.
- *client_data_control_port* The port number of the local machine this instance of duiker will use to communicate with the Q330's control port.
- *packets_per_ack* The number of packets to receive successfully before transmitting an ACK packet back to the Q330.
- *channel_one_orientation_code* The channel orientation code for channel one. Typically channel one is reserved for the Z axis.
- *channel_two_orientation_code* The channel orientation code for channel two. Typically channel two is reserved for the North-South axis.
- *channel_three_orientation_code* The channel orientation code for channel three. Typically channel three is reserved for the East-West axis.
- *channel_four_orientation_code* The channel orientation code for channel four. Typically channel four is reserved for the Z axis.
- *channel_five_orientation_code* The channel orientation code for channel five. Typically channel five is reserved for the North-South axis.
- *channel_six_orientation_code* The channel orientation code for channel six. Typically channel six is reserved for the East-West axis.
- *use_file_mover* Specifies whether or not to place output files to the file mover directory.
- *use_rdd* Specifies whether or not to place output files to the file mover directory.
- *output_directory* The directory completed packet bundles are placed in when
- *output_rotation_period* The period in minutes after which a new packet bundle will be generated by duiker.

- *sensor_a_seismometer_code* The seismometer code for sensor input A as defined by the IRIS SEED specification.
- *sensor_b_seismometer_code* The seismometer code as defined by the IRIS SEED specification. If this parameter is left blank the value will default to the value specified for sensor input A.
- *sensor_a_corner_period* The corner period of the seismometer attached to sensor input A. The corner period is specified as an integer. If the actual corner period is a fraction apply the following rules to specify the corner period.
 1. If the corner period is greater than or equal to 10, specify an integer greater than 10.
 2. If the corner period is greater than 0.5 but less than 10, specify an integer greater than 0.5 and less than 10.
 3. If the corner period is less than 0.5, specify 0.
- *sensor_b_corner_period* The corner period of the seismometer attached to sensor input B. Use the rules specified above for choosing a corner period. If this parameter is left blank, the value will default to the value specified for sensor A.
- *station_id* The five character station identifier. This identifier should be unique relative to all other stations. The parameter can be variable in length from one to five characters and must be values from a-zA-Z0-9.
- *network_id* The two character seismological network this station belongs to. For the UCLA/Caltech Mexico deployment this value is TO. This parameter must be exactly two characters long and must contain values from a-zA-Z0-9
- *location_id* The two character location identifier used for mseed headers. This parameter must be exactly two characters in length and must contain values from a-zA-Z0-9.

4.1.3 Usage

The duiker ipk file installs a cron job that will automatically monitor the duiker process and start it if it is not already running. If duiker is not currently running

it can be run by executing the following commands

```
shell>cd /opt/duiker/bin
shell>./duiker
```

The process will place itself in a daemon state and continue to run after the shell that launched it exits.

4.2 File Mover

4.2.1 Installation

The file mover is designed to run on a stargate with the CENS-seismic filesystem. Do not attempt to install it on any other system configuration unless a binary compatible with your architecture has been provided to you. The file mover also requires that the /opt directory be mounted to the external compact flash card.

To install the file mover on a stargate, obtain the ipk deployment version you wish to install; for example, file-mover_07-25-2005_arm.ipk. Copy the .ipk file over to the stargate and execute the command

```
ipkg install file-mover_07-25-2005_arm.ipk
```

4.2.2 Configuration

The file_mover program structure is organized as follows

- /opt/duiker/bin/file_mover contains the executable file mover.
- /opt/duiker/bin/file_mover.conf contains the configuration information for the file mover.
- /opt/duiker/bin/file_mover.stdout contains run time errors that reflect configuration or usage.
- /opt/duiker/bin/file_mover.stderr contains run time errors that reflect system errors.

The following parameters from file_mover.conf are used to configure the file mover program

- *next_hop* The IP address of the next node to push a data bundle to.

- *previous_hop* The IP address of the previous node data was pushed from. This parameter is currently not used by the file mover, but it should be included if known.
- *queue_directory* The directory that data bundles will be forwarded to on the remote machine. This is also the same directory that will be monitored locally for new data to forward.
- *message_directory* The directory that meta-data bundles will be forwarded to on the remote machine. This is also the same directory that will be monitored locally for new meta-data to forward. Files in this directory always have priority over data files. This is because meta-data is considered more important, as it contains system health information. Also, the meta data is orders of magnitude smaller and as such should take significantly less bandwidth to relay.

4.2.3 Usage

If the `file_mover` .ipk file has been installed, then a cron job will be present that will automatically start the `file_mover` program. If it is not currently running, then it can be manually started with the command

```
cd /opt/duiker/bin
./file_mover
```

The program will place itself in a daemon state and will continue to execute after the shell that spawned it exits.

4.3 Data Sink File Remover

4.3.1 Installation

The file remover is designed to run on any architecture. It is composed of shell scripts and thus its only requirement is a bash shell. To install the File Remover, unzip the package to any location in the system. Next, create a softlink from `cron.hourly` to the `ack-files` script located in the deployment directory that was unpackaged in the previous step. Finally, make the `ack-files` script executable.

4.3.2 Configuration

The following files must be configured correctly in order to run the File Remover software.

- *ARCH*
This file indicates the architecture this instance of the File Remover server is running on. Legal values include "X86" and "stargate".
- *FIRST-HOP*
This file indicates the first hop into the topology of stargates that are collecting data. Legal values include any IP address that represents the first hop into the network of micro-servers.
- *NETWORK-PREFIX*
This file indicates the network prefix that the stargates are configured to use as their subnet. Legal values take the form XXX.YYY.ZZZ. Where XXX,YYY and ZZZ are octet values from 0-255.
- *NEXT-HOP*
This file is a list of mappings that define the topology of stargates in the network. Each line of the file takes the format of *Current-stargate-id Next-hop-stargate-id*. For each *current-stargate-id* value that equals the current node, the *next-hop-stargate-id* node will be forwarded a copy of the mobile agent.

4.3.3 Usage

Once configuration information has been setup, the File Remover will run automatically once per hour. It can be run manually by simply executing the `ack-files` script in the deployment directory.

4.4 Bundle Decoder

4.4.1 Installation

The bundle decoder is typically installed at `/opt/duiker/packet_processor/bundle_decoder`. This is where most of the decoding scripts look for the binary, including the GUI front end and the automated conversion scripts. It can be executed from any directory however as a stand alone process.

4.4.2 Configuration

The bundle decoder program does not require configuration. It reads a tar.gz'ed bundle produced by duiker and produces a number of files that represent data and meta-data.

4.4.3 Usage

The bundle decoder program can be executed by running:

```
./bundle_decoder input - file - nameinput - file - directoryoutput - file - directory
```

Where *input - file - name* is the name of the file to read without the leading directory location, *input - file - directory* is the directory that the input file will be read from and *output - file - directory* is the directory where the output files will be routed to.

After the command is executed the miniseed files produced from the input file will be located in the *output - file - directory*.

4.5 GUI Decoder Manager

The GUI processor uses a Java front end. In order to utilize it, you must have a Java Runtime Environment (or SDK) version 1.4 or higher.

In order for the GUI processor to work correctly the bundle decoding software must be installed and deployed to the `/opt/duiker/packet_processor` directory of the local machine.

The GUI processor has two modes of operation, one is graphical mode the other is `-silent` mode. Graphical mode displays the GUI and allows the user to manage conversion settings.

4.5.1 GUI mode operation

Conversion settings include

- The input directory to look for bundled files in.
- The output directory to place processed files in.
- The directory to backup files to.

- An option to delete the original raw files after processing has been completed.
- An option to backup files to the directory listed.

5 Extending Duiker

This section provides information on how to extend Duiker to enable it to do common tasks that might be useful but are not provided currently .

5.1 Configuration and Calibration by the Acquisition Binary

Various configuration commands can be sent to either the data or configuration ports of the Q330. The best approach is to route all configuration commands to the configuration port, as the data ports do not accept some configuration parameters.

In order to send configuration commands to the Q330, a configuration client must authenticate itself with the configuration port. See the authentication subsection of the implementation section to learn more about this process.

Many configuration parameters require a reboot in order to take affect. For example, changing the recorded sampling rates requires this. This means that after a configuration parameter has been changed it will be necessary to issue a save and reboot command and re-register the configuration client.

Configuration commands take the form of single packets sent to the Q330 and do not require any protocol state management (in my experience you can safely ignore the sequence numbers in command packets).

5.2 Real time access to data

The Q330 protocol implements a packet called QuickView request. The response of this packet contains uncompressed data for the channels requested. The data corresponds to data collected during the past 1-4 seconds and can be used to view waveforms in near real time. Quick view cannot be used for data collection because there is no guarantee of delivery with respect to the data.

5.3 Multiple frequency outputs per channel

The naming conventions for output files in Duiker did not allow for multiple frequencies on the same channel. A relatively simple way to fix this is to append the frequency of the output channel to the file name before the miniseed file suffix is added.

In `packet_processor.c` in the `initialize_output_file_info` function, append the string form of the frequency to the `file_name` parameter before the extension is appended.

5.4 200Hz sampling rates

Currently Duiker does not provide decoding for 200Hz sampling. This is primarily because the use case was not included in the initial design of Duiker and the protocol for obtaining 200Hz samplings was not well understood.

The Q330 does not support the classical layered network model that is typically used to separate network functionality. Instead, the network, transport and data layers are all integrated into one layer. Each packet contains a sequence number, some meta-data about the data contained in the packet and a sequence of "blockettes". These blockettes contain the actual data that has been compressed into a variant of Steim2 compression. Unfortunately, blockettes at times are split across packets. When 200Hz samples are used, the resulting uncompressed data can be 800 bytes long, where as the MTU of any packet is 576 bytes. In order to implement 200Hz samples, it will be necessary to dig through

Although it is not specified by the protocol, the following invariant has been observed to hold true: Given two packets with sequence numbers k and $k + 1$, if packets P_k and P_{k+1} share a split blockette then the blockette will be the final blockette of P_k and the first blockette of packet P_{k+1}

From this, the following algorithm will construct a blockette that has been fragmented across packets.

When a blockette is located that is a split type, add that blockette to a queue. When the final split blockette segment is located, assemble a `one_second_sample` structure and pass it the same was as other `one_second_sample` structures are passed to the channel outputs.