A LOGGING SERVICE AS A UNIVERSAL SUBSCRIBER

by

JAYSON SHARP

B.S., Kansas State University, 2012

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer And Information Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2014

Approved by:

Major Professor
Eugene Vasserman

Abstract

As medical systems expand to allow for the increase the number of devices, new ways to protect patient safety have be developed. The Integrated Clinical Environment, ICE, standard sets up a set of standards that define what an integrated hospital system is. Within the specification is a direct call for a forensic logger that can be used to review patient and system data. The MDCF is one implementation of the ICE standard, but it lacked a key component the ICE standard requires, a logger. Many loggers exist in industry, with varying rates of success and usefulness. A medically sound logger has to be able to completely retell exactly what happened during an event, including patient, device, and system information, so that the right medical professional can provide the best care. Several loggers have been built for MDCF, but few were practical due to the invasiveness of the service. A universal subscriber, a service that is able to connect to all publishing data streams, logging service was built for the MDCF which has the ability to record all information that passes over the MDCF messaging service. This implementation was then stress tested with varying numbers of devices and amounts of data. A reviewing tool was also built that allows for replay of device data that is similar to the original device UI. Future work will include looking into storing system information such as state changes within MDCF and system health. There is also a push to further integrate the forensic reviewer into the core MDCF UI.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisors, Dr. Dan Andresen and Dr. Eugene Vasserman. The amount of things they had to put up with from me was staggering and I'm just glad that they stuck it out. I'd also like to thank the Beocat administrators for helping me through all of my testing issues. Some of the computing for this project was performed on the Beocat Research Cluster at Kansas State University, which is funded in part by NSF grants CNS-1006860, EPS-1006860, and EPS-0919443.

Thank you to my friends for always being there for me. It's been a long ride.

Dedication

I dedicate this to my father.

Chapter 1

Introduction

The number of medical devices used in hospitals has increased over the past several years, as has the number of humans needing medical attention. Each medical device exists to provide care for a patient. The high number of devices allows for a clinician to gain better insight into a patient's condition. This data provides a way for safer care of a larger population without needing additional physicians. Each of the medical devices output large amounts of very useful data in a digital format. There is a need for a way for these devices to work together effectively, which the ICE standard provides. The ICE, or integrated clinical environment, standard outlines a structure where devices can communicate with each other to provide more intelligent care for patients.[1] This integrated system allows for things like smart alarms, which work by sharing vital information to only set off alarms when they are needed. For example, if there are five devices attached to a patient, and 4 of them report that the patient is fine and the fifth says the patient is having a problem, there is most likely something wrong with the device and not the patient. Something as simple as a loose connection to the patient can cause alarms that report that the patient is critical, causing alarm fatigue in the attending medical professionals.

The Medical Device Coordination Framework (MDCF) is an implementation of the ICE standard. MDCF has within it a messaging service that handles all data from medical devices as they communicate. It is obvious that there needs to be something recording this information for later analysis.[2] This analysis can be for the benefit of a patient using the devices, as well as the companies that created them. The companies benefit from adverse event forensics. It is a needed service that provides an extra layer of protection for all users of the medical system. The logging service needs to be able to handle the high stress loads of multiple users with multiple devices. It needs to be simple enough to be used by clinicians. The service also needs to stand apart from the medical system, both to protect itself and the medical system in cases of failure. This thesis presents my work creating such a logging service built to the specifications of the ICE framework.

The focus of the thesis is the process by which to build an effective logger. First, I lay out the overarching ICE standard that informs and guides the work. From there, I discuss the MDCF, which is an implementation of the ICE standard. The evolution of the project is outlined within as well. Specific storage programs are used, but are implemented in a way that they can be interchanged as needed. This style of logger is elegant in its simplicity. This idea can allow for the similar logger implementations in different medical systems without much change to how it works. This work will speed up the process of logging service implementation for other similar medical systems.

Chapter 2

Background and Requirements

My overall goal for this project was to build an optimal logger given our list of requirements. The requirements that the logging service must meet are driven by the overall structure of the MDCF project and the ICE framework. The intention was to learn the most about the overarching project structure and real world logger variations so that a logger could be built for MDCF.

2.1 ICE

The Integrated Clinical Environment (ICE) is defined in ASTM standard F2761.[1] The purpose of the ICE standard is to define ways that medical devices should interact in order to reduce the number of preventable errors during medical treatment. This is accomplished in a three part system: the ICE network controller ensures devices are operating properly and logs data generated by the devices, the ICE network supervisor is responsible for detecting and preventing medical errors, and the ICE network interfaces which allow medical devices to communicate with the system. The ICE standard calls specifically forensic data logger in section 4.2.4.

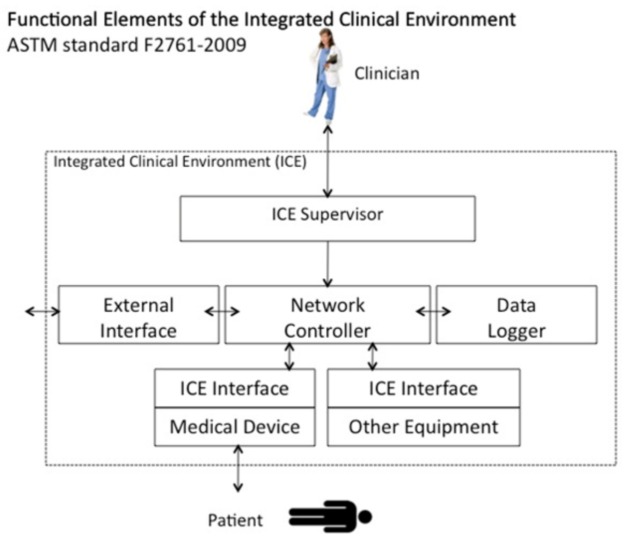


Figure 2.1: *Functional elements of the Integrated Clinical Enviroment*

2.2 MDCF

The MDCF, or Medical Device Coordination Framework, has the goal of providing a template for an integrated hospital system built using the ICE specification.[3] MDCF is built upon a publish-subscribe architecture. There is a centralized channel service that handles all message traffic and provides a connection hub for external devices and system applications. The channel service allows for the establishment of publishers and subscribers and acts as the information interchange. Publishers can send data over their designated channel to all subscribers that are signed up for that channel. Each device or application will most likely have several publishers and at least a couple subscribers a piece. The goal is to have system that can have real world medical devices connect to the centralized network, and from there share information with other connected devices to provide a safer environment for the patient. By facilitating the transmission of this information, the MDCF can use programs like smart alarms and device application to provide better care.

A logger is needed to allow for forensic data analysis. With a log you can determine how a patient reacted to a certain drug across several devices. The core MDCF contains the channel service and all items needed to manage it as well as an interface for launching applications that display incoming device data. A device or devices is shown that want to connect and start sharing information, as well as an application that would like to subscribe to the data being broadcast by the newly connected device. A very simple example scenario is shown in Figure 2.2.

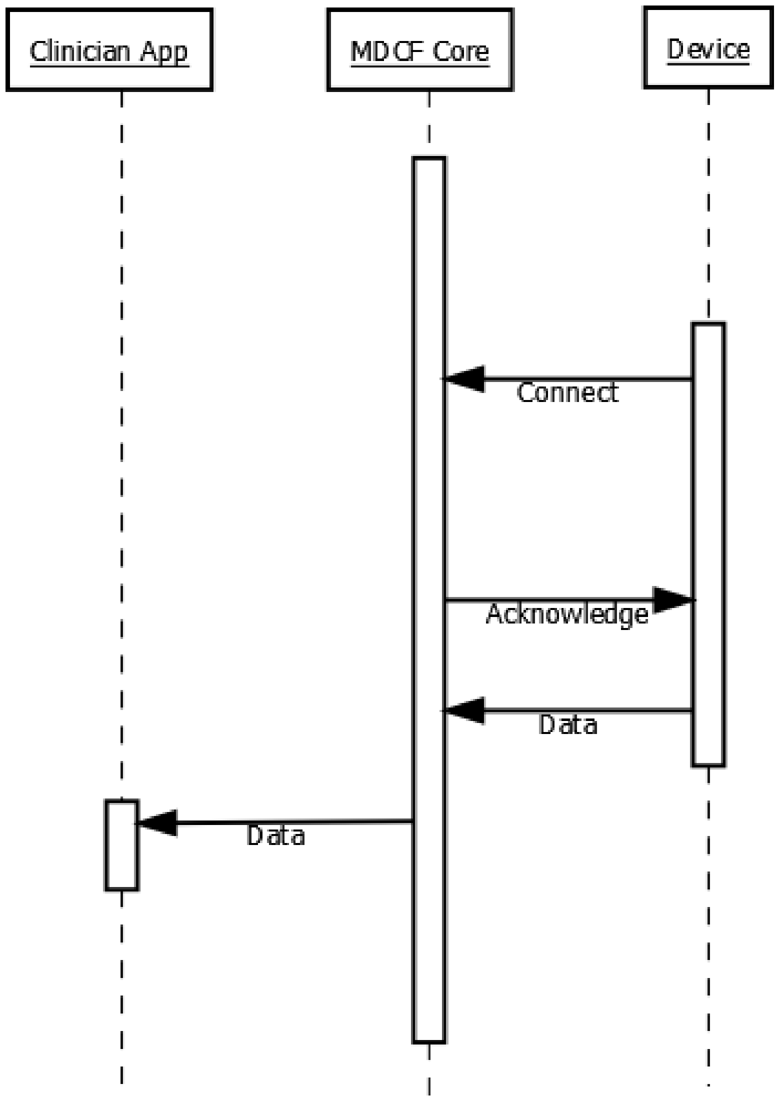


Figure 2.2: *Working single device scenario of the MDCF*

2.3 Requirements

To build the logger for MDCF we had to first define what it was we needed it to accomplish. This action seems basic but what needs to be stored extends beyond just the patient data that is visible on the monitors is a hospital room. Device state changes, system state changes all need to be saved. This is explained in more detail later, but in short we need to be able to completely replicate what happened in the hospital room down to the smallest detail. The interaction between devices could prove useful when determining the cause of an event. This data is valuable, and needs to be treated as such.

2.3.1 Separable but Integrated

Any logging service attached to a medical system must be self-reliant to prevent cascading failures.[2] This point can pushed one step further to say that any application cannot be a singular cause of ICE supervisor or network controller failure. A failure is defined by the services being in any state other than the one they are intended to be in. The ICE core, the combination of network controller and supervisor, cannot depend upon the logger, nor can the logger depend upon ICE. In cases where the logging service fails to start, ICE should be notified, but unaffected. This requirement was made based on basic safety ideas. This carries over into the ICE system. In Figure 2.3 we see how the logger connects to the ICE core service via subscribing to system channels (1). (2) shows the logger accepting device data as it is broadcast by devices through the network controller.

2.3.2 Scalable and High Performance

As the number and complexity of devices scale up due to the ever increasing population, so to must any logging service be able to scale and handle the extra load. If a logger cannot scale to meet the demands of a real world situation then it is not useful. This ideal goes hand in hand with the goal of high performance. As the resource requirements

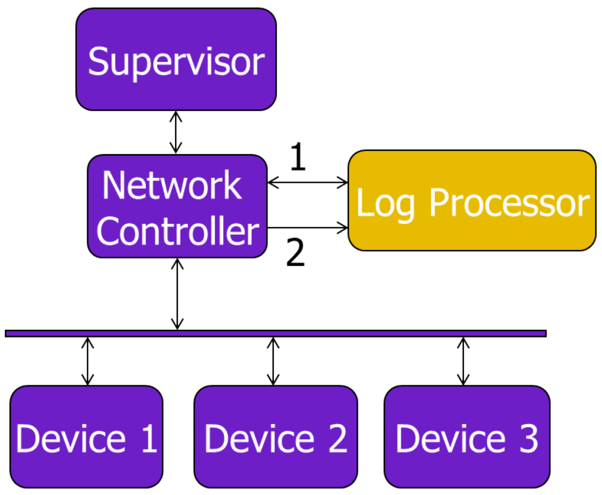


Figure 2.3: *High level view of how the logging service is attached to ICE*

for ICE core service increase, the resources needed for the logging service cannot exceed a growth limit. This limit is as of yet undefined, but we need to avoid exponential growth in resource requirements. For example, one ECG (electrocardiogram) device sends over 144,000 messages every second. It is important to note that this is just one device, and that a single hospital wing could have several dozen of these devices. A logging service needs to be able to handle a large amount of data without exceeding ridiculous resource requirements.

2.3.3 Usable

The usability requirement is twofold. First, the data stored needs to be accessed in a readable format that can easily analyzed by a clinician. There should be an easy way to read, and play back patient events as they happened in the hospital room. Forensics play a vital role in patient safety, and its imperative that data replay be as easy as possible for future clinicians to perform.[2] This means that basic text logs are too inefficient due to the length of time and expertise required to parse the data. The goal is to have the data available on demand for any clinician with the correct credentials, and to allow technicians to overview events to determine technical faults within the system.

Second, the service must be designed in such a way that developers can update it and integrate with devices without much work. The best possible logger design would require no change to device code. The second portion of this requirement rests mainly upon the shoulders of the logger developers to write clean coherent code that can be updated without issues. The design should also allow for easy integration with the core system to be logged, as well as not requiring extra work from device companies so that their devices can have their data logged.

2.4 Ideal Logger

An early conception of the ideal logger was made assuming no real world limitations such as money, bandwidth, processing, politics, and storage space. In an ideal service, all patient, system, service, and app information would be stored in a log. This is a lot of data, and although storage is cheap, keeping this amount of data for long periods could prove troublesome even with a lack of physical space for hard drives. This data provides insight into all patient related events. Using the stored data a clinician should be able to replay an entire situation down to the smallest detail to find the cause of an event. Forensic analysis is vitally important to understanding what caused an event. By understanding intimately how events happened in the past clinicians can work to prevent those same events from happening again. This idea can go one step further to protect device companies of liability in the event of adverse events. All events, not just patient health data, are stored in the log, so that even device or system state changes can be included in scenario replays. With this much data someone could follow an event from its beginning in faulty device (A) giving a poor reading which allows device (B) to give more medication than the patient can safely have. Which device is at fault? The one that gave the medicine, or the one that told it to dispense it? The ideal logger would be able to recreate the chain of events with high fidelity.

2.5 Real World Implementations

The flight data recorder, or black box, has become one of the most prominent logging devices in this century. These small devices record either voice or flight telemetry information and can also record the communications between pilots and traffic controllers. The flight telemetry information is stored for a minimum 25 hours, and is then over written in a loop. The cockpit voice recorder stores data for a minimum 2 hours,[4] again in a continuous loop. The importance of these devices is two fold. We can see what happened in this situation to this plane, and we can also better understand how to prevent problems like this in the future. This logger allows us to recreate the situation that caused a plane problems in the same fashion that a logger for the MDCF must be able to show why an event occurred.

Operating system logs provide insight to a multitude of system issues that can be both hardware and software related. These logs are kept for a variety of reasons, and Microsoft diagnostic tools list specifically: debugging and error handling, performance profiling, error reporting, and event notification to name a few.[5][6][7] The logs are a tool that can be used to diagnose problems with a system or to just monitor how the system is running. There are limitations to these logs though. Components that malfunction may no longer be able to report information to the system logs so diagnosis of issues is hindered. The other metric where this realistic logger falls short of our goals is how closely coupled this logger is with the system it logs. It is literally a program that is run by the operating system. The new logger we created needed to be more separate from the system than an operating system log, while still providing the same level of detail as a flight data recorder. Using the ideals we set up and with the real world limitations we had, we built a suitable logging service for MDCF. It functions in a similar fashion to the real world loggers we discussed, but blended some of the features together.

Chapter 3

Implementation

The MDCF logging service underwent many iterative changes. My work initially built upon the existing MDCF logger, which is illustrated in Figure 3.1. Each device had logging code attached and saved directly to individual log files. This method did not meet several of the requirements we set forth for the project and was scrapped, which allowed us to rethink how the logger functioned. I knew we needed to catch all device data as it is being broadcast, so I delved into the MDCF network controller. Using the debugging tools within Eclipse, the Java programming environment used for MDCF, I was able to pinpoint the piece of code that creates the new channels for each device as they connect to MDCF. After this distinction was made, we were able to catch all data that devices broadcast. We then were able to focus on what to do with that data. Again, we utilized previous work until it became apparent that it functioned much too slowly for our purposes. This is where our message processing became important, because it allowed for us to store data within a database for fast retrieval and efficient support for rich queries. All together, these pieces form what we call, the universal subscriber.

The universal subscriber can take all the messages that are sent over the MDCF system and output them to a storage space. This changed the logging service from just a piece of code that would be specific to MDCF, into an idea that could be adapted for other projects

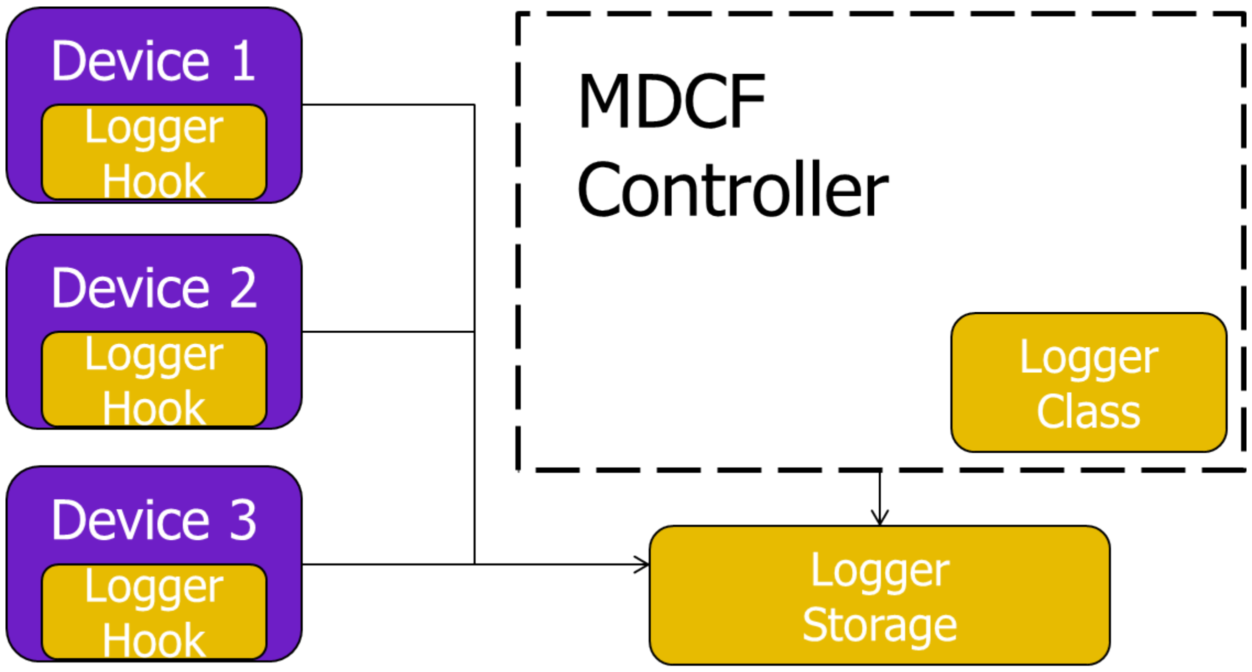


Figure 3.1: *Original MDCF logging structure*

that use a similar publisher and subscriber approach. There are three parts to the universal subscriber: the hook, log processor, and storage mechanism. A majority of the work is centered in the log processor, which is also the runnable service. Figure 3.2 shows the universal subscriber setup. The way it is built not only satisfies many of the requirements set forth for the project, it also allows for a distributed system where the log processor, storage mechanism, and MDCF could all run on different machines.

3.1 The Hook

The hook code is how the logging service receives information from the host system. The goal is to put as few lines of code into the host system to allow for the logging service to receive relevant channel information. For this project, an extra data channel was added to the MDCF list of system-created channels to facilitate the transmission of new channel connection information. This channel information includes all the data needed for estab-

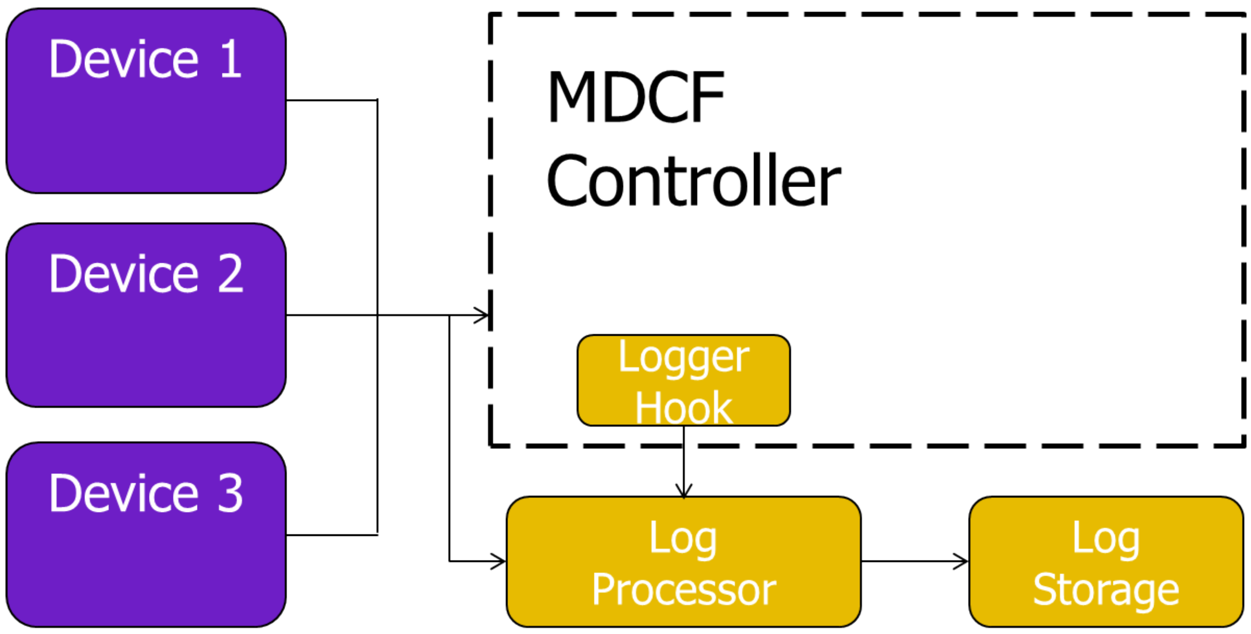


Figure 3.2: *Universal subscriber MDCF logging structure*

lishing a new subscriber for a publisher. The logger channel, LoggerChan, is started along with the other MDCF internal channels when the MDCF starts. Inside the channel service of the MDCF there is a function that is called whenever a new device connect with the MDCF. All new device connections go through this function. At this point I added a simple check statement that could pass the new device channel information across LoggerChan to the logging service, without changing the original operation of the function. Section 3.4 discusses situations where the logger is not running properly.

```
if (!servers[0].equals(channelName))
{
        LoggerSender.sendMessage(``Channel Created:"+ channelName);
}
```

Figure 3.3: *Message that the server sends to the logger whenever a new channel is created*

This meets the requirement for separable but integrated. It allows for the transmission of information to the logger without much added code to the host system. Any failure on the part of the logger cannot affect the MDCF. Using this process, as long as the logging service is running when a new device connects, all of the new channel information is sent to the log processor.

3.2 Log Processor

The log processor is the heart of the logging service. When this process is started, it receives new publisher data over LoggerChan from the hook code in the MDCF channel service. The new channel data is then used to establish a subscriber. This process is repeated for every device. The subscriber joins the list of others that have already been connected. Each device that connects to the MDCF can have multiple internal channels, and each of these will be connected to the logger in succession after the initial device connection. The processor is a runnable program that stands apart from the MDCF.

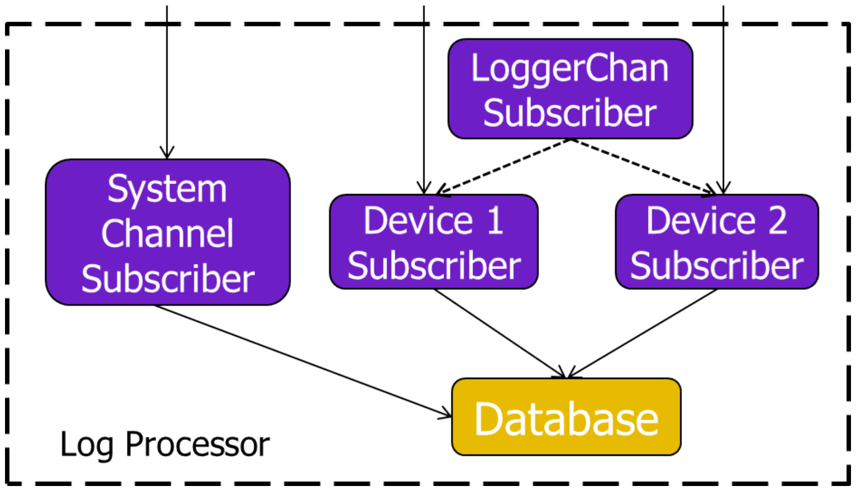


Figure 3.4: *Log processor structure*

It is important to note at this point that because of the universal subscriber method there will be, in the worst case, an increase in message traffic by one third of previous traffic. Every channel will have one extra subscriber; the log processor itself, increasing the number of subscribers also by at most one third. This is important information when thinking about the scalability of this service. On most of these channels, patient data will be filtered through a message processor. A little bit of message manipulation is done to make retrieval of data easier at a later date.

The current system uses the following in its message structure: a channel name, two timestamps, device UUID (universally unique identifier), and data payload. The channel name and device UUID are combined in the data payload and then parsed into separate parts for storage. This parsing also allows for data retrieval based on device, which can have multiple channels, instead of the unique device/channel name combination. This process happens for every incoming message on every channel. After the messages are parsed into a data structure, the data point, it is output to a storage mechanism.

3.3 Storage

The goal of the storage mechanism is to provide a space to securely save large amounts of data. Originally the storage mechanism was integrated into the logging service, but I found that, as requirements changed for the MDCF, it became more important to have the ability to easily change the type of storage without major changes to the service. Once this was done, it became trivial to upgrade from a logger that output data to a text file, to a simple database and from there to more advanced databases. The current implementation of the logging service uses the SQLite database structure to store the data received from the MDCF.[8] A more in-depth description of the original text storage mechanism is located in Section 3.6.

The original text logger was just a direct export of the incoming messages after they are converted to strings. While it was a very fast option for storing data, because it requires no messaging processing, data retrieval was unacceptably slow. Figure 3.5 shows the line of SQL code that outputs the data point object, that was created from each incoming message to the logger, into the database. By using SQLite or other databases and not the original text logger we can retrieve data from a log much faster. This helps with meeting

```
String temp = ``insert into dataPoints (" +
                ``channel, " +
                ``timeStamp, " +
                ``isDeviceChannel, " +
                ``deviceUUID, " +
                ``data) " +
                ``values(...)";
systemStatement.executeUpdate(temp);
```

Figure 3.5: *SQLite code used for storing a data point*

the requirements for usability as well as scalability. Not only is the logger easy to use, it is easy to retrieve the data after the fact.

3.4 Scenarios

In this section I walk through three different scenarios for how the logging service can interact with the MDCF and devices. Figure 3.6 shows the correct ordering in which to startup services. First, the MDCF core service starts and establishes its channel system. Next the logging service is started, and it connects to MDCF system including LoggerChan. After that point, any number of devices can connect to the MDCF core service, which will then notify the logging service. All data will then pass fom the device to the core MDCF service, and then to both the logger and the clinician application, if it is turned on.

The second and third scenarios are similar to each other, in that they both show failed sequences of events. Note that they fail in different ways. In Figure 3.7, the MDCF started first, but the devices started second. The MDCF sends the notification over LoggerChan whether or not logger is there to receive the information, which in this case it is not because the logger is not turned on. After this point the logger is started and any future devices connected to the MDCF will be logged as intended, but the first one will not be logged. The fix for this problem would require more information to pass back and forth between the logging service and the MDCF to request for current list of channels created, and reporting

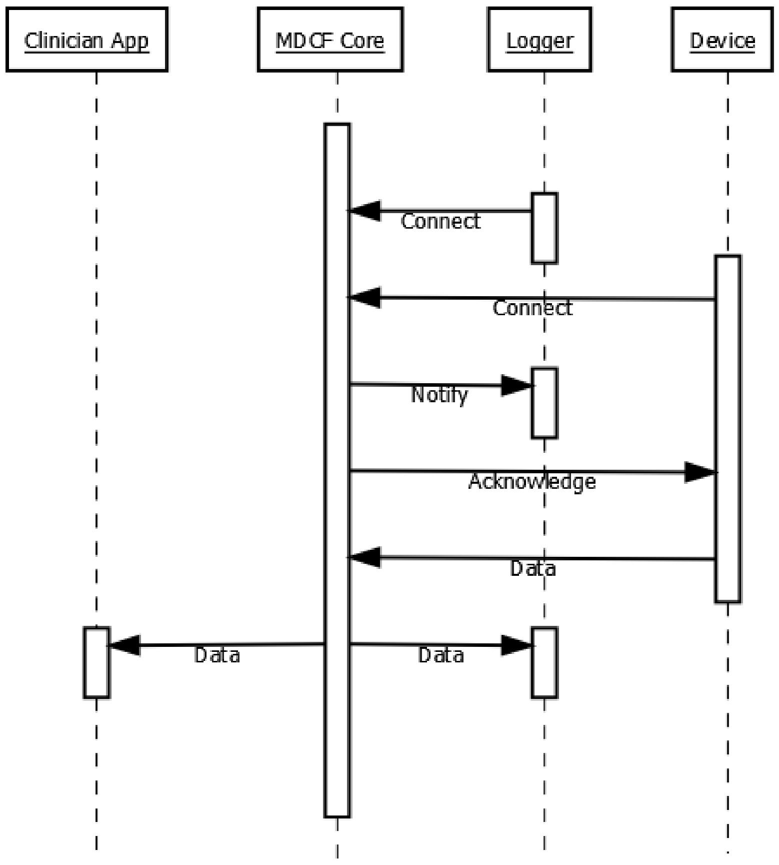


Figure 3.6: *Working single device scenario*

the current list of channels being logged.

The final scenario is the most devastating problem with the easiest solution. In this scenario, the logging service is started prior to the MDCF core. When the logging service attempts connect to the MDCF channels it fails. After this point the logger is cut off from all channels and will sit idle. The fix for this is to have the logger retry to connect with the MDCF until a connection is made. If no connection can be made after a certain point, the logger should notify the proper technician that something is amiss. The most important part of this scenario is that is shows that the logger needs to be able to identify if it fails to connect to the MDCF to prevent the loss of log data.

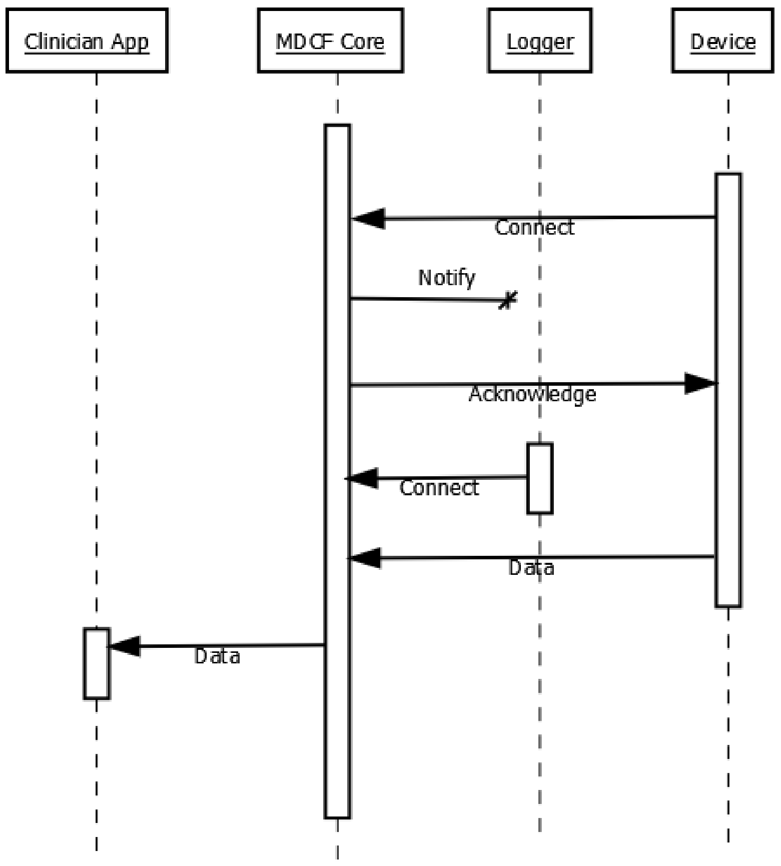


Figure 3.7: *Device starts before the logger*

3.5 Replay

The log viewer started as a way to better read the text logs that the logging service was creating: a proof of concept tool to show that the logger was functioning as intended. From there, the viewer program was expanded into a program that could not only show a easier to read text version of the stored data, but could also show the data in graphs similar to the ones on the devices that were logged. The replay program works by submitting SQL queries to a database. A user can then pick different channels for replay. In Figure 3.9, the lower box shows the text of the log, and the middle area is left to display the graphs.

This is a very rudimentary viewer program, and a more integrated viewer program is the subject of future work. The end goal of the log viewer is to make it possible to perform

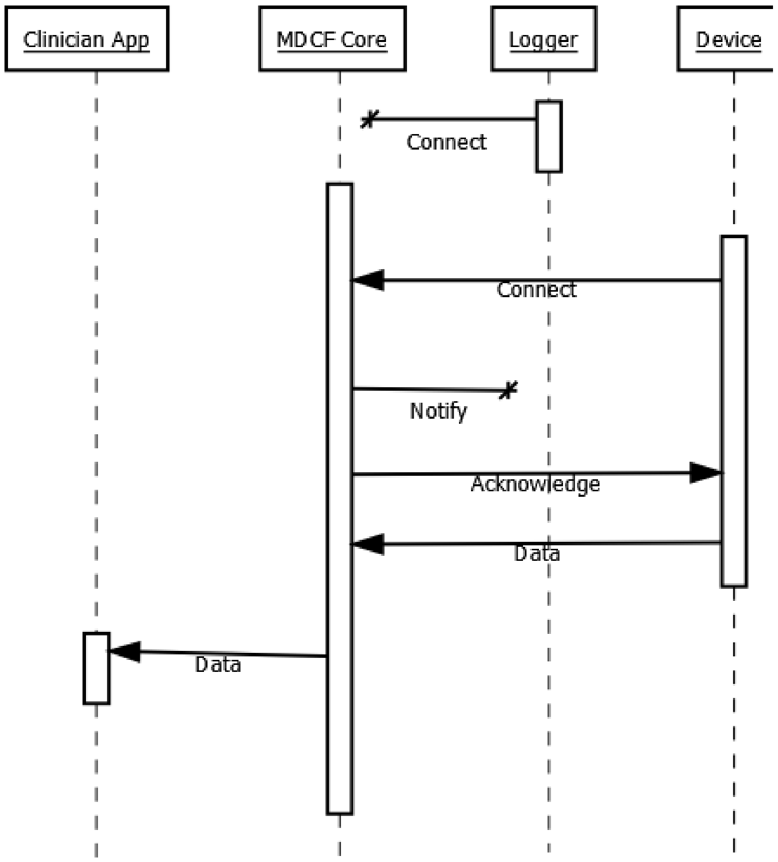
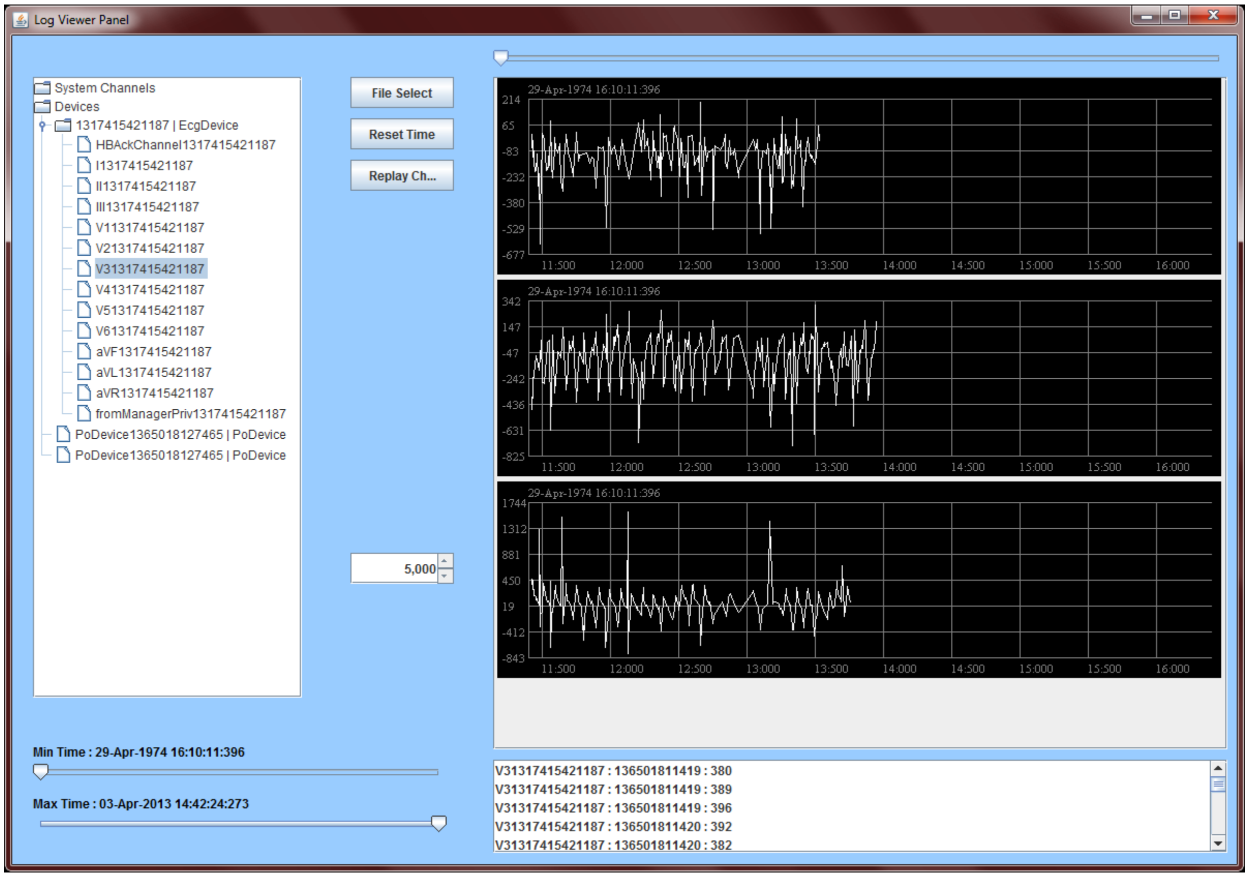


Figure 3.8: *Logger starts before the MDCF*

forensic analysis on what happen in the system. This can be critically important for the health of the patient to prevent future issues, and liability for device companies that are integrated into the MDCF system.

3.6 Previous Work

The universal subscriber model was not the original design for the logging service. To better understand the current design, it is important to know where we started. Prior to my work on the logging service, there was a functional logger. This used Apache's Log4j, a logger class file, and a logger properties file.[9][10] An instance of the logger had to be added to every device, piece of system code, and application that was to be logged. This was



```
V31317415421187 : 136501811419 : 380
V31317415421187 : 136501811419 : 389
V31317415421187 : 136501811419 : 396
V31317415421187 : 136501811420 : 392
V31317415421187 : 136501811420 : 382
```

Figure 3.9: *Log viewer snapshot that shows the replay of three ECG channels*

cumbersome. New devices and code would all need to be tagged with logging information that the developers may not be trained in. This was a big violation of the separable but integrated project requirement. When we changed to the universal subscriber model, we kept Log4j at first, but then found better options for the storage portion of the service as our needs changed. Log4j did provide a way to store information with different levels of importance, but without furthering the MDCF project as a whole and defining which messages are more important this feature was not useful at the time. Log4j was being used as a glorified text file writer. To save processing power I dropped the use of Log4j and output the log messages to text files manually in Java code. Currently we use SQLite as the storage mechanism. As messages are received, they are processed into their individual parts and stored in the database.

Chapter 4

Evaluation

To test the logging service implementation, we used the Beocat high performance computing cluster.[11] We used two cluster nodes: one ran the MDCF and devices, the other ran the logging service. Each machine had two 8-Core Intel Xeon E5-2690 processors and 64GB of RAM.

One of the major items we wanted to test was the difference in the number of devices versus the number of messages, and how that affected the MDCF and logging service. There were four pairs of tests, where each pair shared the same number of data messages sent. The first part of a test had one device outputting a variable number of messages per second. The second part had a number of devices outputting 1 message per second. I wanted to track the different ways the logger and MDCF reacted to these situations. A control test was also run where the logger was never turned on. The effect of the programs was measured by Nmon, Nigel's performance Monitor for Linux.[12] The goal is to show that there is not a major scalability issue with the logger when the number of devices increase, and to also show that the MDCF is not affected too much by the extra message throughput.

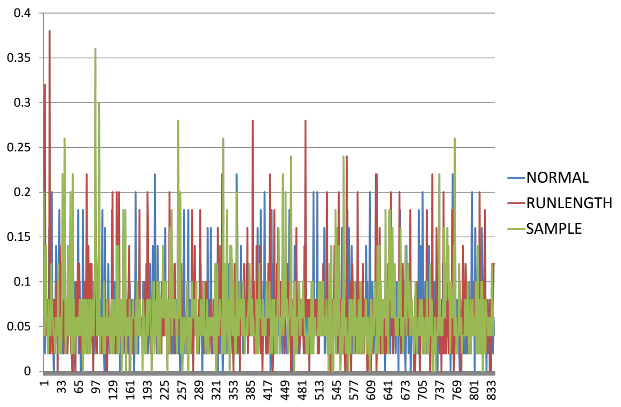
There are 4 pairs of tests where each pair of tests each sends the same amount of patient data messages from the running devices. Even though the amount of patient data sent for each pair of tests is the same, the total number of messages sent in each test is different. The

Test	Device count	Message per second per device	Patient messages per second
0	1	10	10
1	10	1	10
2	1	100	100
3	100	1	100
4	1	500	500
5	500	1	500
6	1	1000	1000
7	1000	1	1000

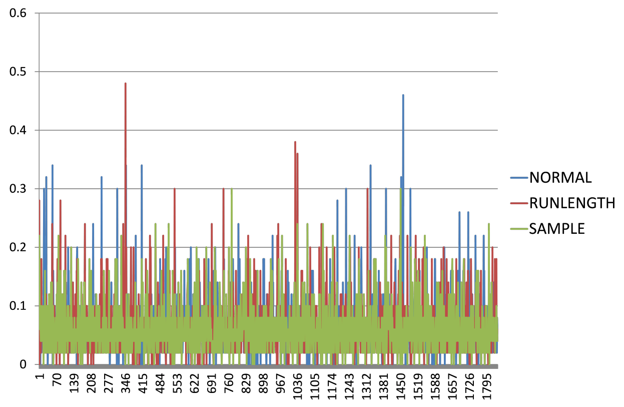
Table 4.1: *Testing variables*

difference in total messages sent is because of setup and heartbeat channels for the devices. One device will have a group of setup channels that help with the device connecting to the MDCF. The heartbeat channel is just another channel to log from the logging service's point of view, but on the MDCF side the heartbeat channel is used to determine if a device is still active. For example, in test 0, we have 1 device producing 10 messages per second. That device also has a group of setup channels and a heartbeat channel. Test 1 has 10 devices outputting 1 data message a second. This is the same amount data messages, but there is an increase in total channels due to the setup and heartbeat channels each device creates. So, each pair may share the same number of data messages, but the higher numbered test in each pair has a increased number of setup channels.

Three different logger types were tested. The first is what I called the the "normal" logger. It stores every message that is passed to it. The other two logger types were built to give some insight to future logger ideas. The first variant is a logger that samples data from the channels it is connected to. Every tenth message on each channel is stored in the log, the rest are dropped. To do this, I added a counting system that keeps track of the current count for each channel connected to the logger. The other type of logger uses run length encoding. Run length encoding is the process of determining if a given set of data points are within an acceptable range. If the data points do not exceed or drop below a certain threshold you can store that the data points are valid instead of storing the 100



(a) Test 0 logger CPU usage



(b) Test 7 logger CPU usage

Figure 4.1: *Logger CPU usage test results*

individual data points. I save temporarily 10 consecutive messages from a channel and then performing a mathematical calculation, then store the new number. For the test I performed an average, in the real world the math will be different for each patient and device. The goal was to try and have the structure needed to perform those acts present during the testing.

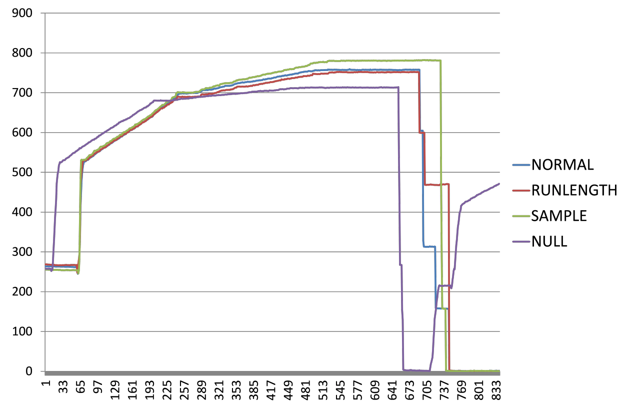
4.1 Results

I measured CPU, memory usage, disk writing, and net packet travel. For the MDCF measurements we are less concerned with disk usage, since the MDCF core service never writes to disk, and any measurements seen there are from writing to standard error and out. In most demanding test, with 1000 devices sending out 1 message per second, the logger in all 3 variations never exceeded 50% total usage of a single core of a 16 core machine. In the smallest test the CPU usage measurements for the logger never exceed 40% usage of a single core. This rough 10% increase CPU usage happens after an increase in message traffic of 10000%.

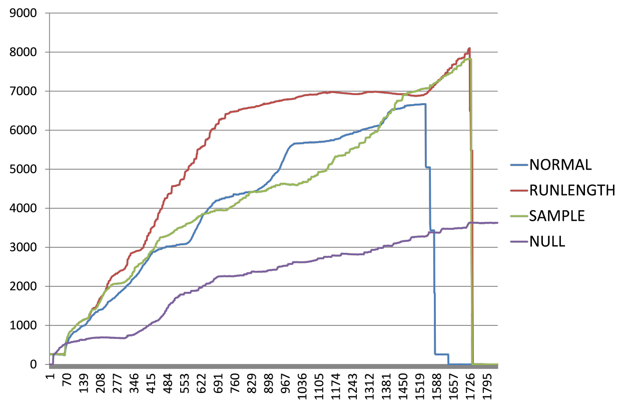
Memory usage did not have any significant increase for the logger across the board between all the tests and logger variations. There an approximate 8% gap in memory needed between the normal logger and the run length logger in total memory needed, but this measurement difference could very well have been caused by outside forces. Not to mention that the 8% gap, is less then 300 MB in size when there is an available 64 GB. The interpretation of the little to no change in memory usage of the logger can be attributed to an aspect of the JVM. Prior to the start of the logging service, a predetermined amount of memory is allocated by the program and no more is needed from that point. If more memory was needed we should see a spike at that point in the memory graph. Across all 8 tests, the amount of memory for the logging service never varied from its initial measurement.

On the MDCF core service side the data is just as easy to follow. During the tests with multiple devices, the CPU usage spikes above the 100% usage on all 16 cores, but quickly goes down. This spike is from the device startup sequence. After the initial spike there is an average use of 10 cores on the worst test case run, with 1000 devices outputting 1 message a second.

The important metric on the MDCF side is the control measurement, or test without the logger running. During the heaviest test, the control, NULL as labeled on the graphs, measurement was consistent with the other logger logger types, it had the same initial spike and leveling off point. This shows that the increase in message output of the MDCF to the logger did not heavily impact its CPU use. Memory usage steadily increases over time as devices start up and have a steep drop off after the ten minute mark when all devices are killed off. Across the tests this remains the same. The drop off point varies slightly due to the use of lock files to control the testing sequence. The memory graphs were normalized by setting the starting value of each test series to lowest point of each test. After performing that action we can see that the graphs appear very similar. The rest of the graphs can be seen in Appendix A and B. The tests show that logger does not have a significant impact upon the functioning of the MDCF service, and that the logger can expand to more devices without bloating exponentially in size.



(a) Test 0 server memory usage



(b) Test 7 server memory Usage

Figure 4.2: *Server memory usage test results*

4.2 Adaptation to the MDPnP system

To further show that the logging method is applicable to systems other than the MDCF we transplanted our method to another medical middleware system: Medical Device Plug and Play interoperability program developed by CIMIT.[13] Their system shares a majority of the same goals as the MDCF, but the way MDPnP communicates is quite different.MDPnP uses DDS, Data Distribution Service, as its message protocol.[14] Their DDS implementation has a channel for each data type, so all numeric data, for example, travels on the same channel. Using that concept I was able to implement a logger based on the universal subscriber method.

It is important to understand how the DDS implementation used in the MDPnP project differs from the networking service used by the MDCF project. In MDCF, each device uses one or more distinct channels to send data to the system. The channels each represent a different logical stream of data from the device. The MDCF channel service is responsible for managing and maintaining the list of channels the system is using. Before a device can use a channel, the server must create it, and each device or application that wishes to communicate using that channel must explicitly connect to the channel through the MDCF server. Therefore, the logger must be notified of each new channel as it is created, as there is no way to query the server to receive a list of active channels.

The DDS implementation used by MDPnP operates in a much different way. One can think of DDS as a stand-alone networking server completely outside of the MDPnP server itself. Therefore, any system that wishes to connect to the server only needs to know the correct DDS settings. The MDPnP system delineates channels based on the type of data being shared, and not the individual device data streams. Therefore, more than one device may send data across a single channel. In addition, each channel does not need to be created by the system; instead, devices and applications must simply know the name of a channel they wish to use to communicate, and DDS will handle making the connection work. DDS provides a method to query the system for a list of channels in use, and actually includes a network sniffing program, DDS Spy, that will reveal all data being sent across the DDS system.

The MDPnP logger, like the MDCF logger, has all three method parts, the hook, the processor, and storage mechanism. Though, because of the way that DDS functions, this version of the logging service can be completely standalone from MDPnP. The important part is how DDS handles data. Once a channel has been created, anything can connect to that channel and listen for the data that it needs. It is a more predictable system because there are fewer channels that are not unique on startup like devices on the MDCF.

For the purposes of this paper I used DDS Spy, a tool that allows for tracking of DDS messages over a network to determine the names of channels and the types of data that traversed them. The hook consists of a series of functions to connect to the list of data type channels were found using DDS Spy. From there, the messages are processed in a slightly more readable format and then stored in a text file. This version of the logger is very rudimentary, but it does prove the point that the universal subscriber method can work in multiple situations. A more thorough logger could be built on top of DDS Spy if given access to the source. All that would be needed is message processing and a storage space attached to the program that can already track every DDS channel and message packet.

While the MDPnP logger only stored a few channels, it has the potential to be more thorough than the MDCF logger without any extra effort. The MDCF logging service requires additional messages to be placed with the MDCF to provide system state information, while on MDPnP, all of that data is already sent over one of the default DDS channels. This experiment provided a concrete example that can show how this logging method works in different systems.

Chapter 5

Conclusions and Future Work

The point of this project was to build a logger that can grow and evolve with the times. As the project progressed it became less about a logger to work in this one particular situation, MDCF, and more about how to build an effective logger for medical systems in a general sense. The idea of the universal subscriber, or a method where a single service subscribes to all publishers, is not new, but in this instance it can be really effective for the storage of MDCF service information. The project has several more points where it can press forward from my initial work, but the foundation of the universal subscriber will most likely remain the same. The major focus will most likely be integrating a log viewer into the MDCF UI. The logging service will most likely survive the test of time because of the three main requirements: separable but integrated, scalability and performance and usability.

For the future there are a few points where the current implementation needs to improve. First, the logger needs to be able to connect to the MDCF core after a connection failure. If after a predetermined amount of tries the logger fails to connect to the MDCF, a notification needs to be sent about the issue. Additionally there needs to be a way for the logger to query about channels it may have missed while not connected to the MDCF core service. With these two items, the logger would be able to stand much more on its own, and it would not require such a strict startup policy. The one drawback is that this requires additions to the MDCF core service to answer the channel information requests.

Another step towards the future would be to use a industry used database choice rather than the functionally simple SQLite. A new log viewer is already being built with the intention of replaying entire scenarios as they were recorded in the log data. These advancements would push the logger closer to the ideal one described. The last is log size reduction. Not all information needs to be stored in the log. Out of the 144000 messages an ECG sends every second, we do not need every one of them. The future goal is to implement a way to change the amount data stored on a patient by patient basis. If a patient is in critical condition, all possible information should be stored, but for an otherwise healthy patient, not all information is needed. This would require more detailed patient information passed to the logger to allow it to know what messages can be dropped, or for the channel service to just not send messages to the logger that do not need to be logged. By reducing the amount of information that goes into the log we an save on storage costs.

Bibliography

[1] ASTM International. ASTM standard F2761, 2013, medical devices and medical systems - essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ICE) - part 1: General requirements and conceptual model, 2013.

[2] David Arney, Sandy Weininger, Susan F. Whitehead, and Julian M. Goldman. Supporting medical device adverse event analysis in an interoperable clinical environment: Design of a data logging and playback system. In Olivier Bodenreider, Maryann E. Martone, and Alan Ruttenberg, editors, *ICBO*, volume 833 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[3] Andrew King, Sam Procter, Dan Andresen, John Hatcliff, Steve Warren, William Spees, Raoul Jetley, Paul Jones, and Sandy Weininger. A publish-subscribe architecture and component-based programming model for medical device interoperability. *SIGBED Rev.*, 6(2):7:1–7:10, July 2009. ISSN 1551-3688. doi: 10.1145/1859823.1859830.

[4] Flight data recorders and cockpit voice recorders. (20 C.F.R 91.609). http://www.ecfr.gov/cgi-bin/text-idx?&rgn=div5&node=14:2.0.1.3.10#14:2.0.1.3.10.7.7.5, May 2014. [Online; accessed 2-June-2014].

[5] MSDN. Diagnostics. 2014. URL http://msdn.microsoft.com/en-us/library/windows/desktop/ee663269(v=vs.85).aspx. [Online; accessed 2-June-2014].

[6] Apple Inc. Logging errors and warnings. https://developer.apple.com/library/mac/documentation/macosx/conceptual/bpsystemstartup/chapters/LoggingErrorsAndWarnings.html, October 2013. [Online; accessed 2-June-2014].

[7] syslog.conf(5) - Linux man page. http://linux.die.net/man/5/syslog.conf. [Online; accessed 2-June-2014].

[8] SQLite. About SQLite. http://www.sqlite.org/about.html. [Online; accessed 2-June-2014].

[9] The Apache Software Foundation. Foundation project. http://www.apache.org/foundation/, May 2012. [Online; accessed 2-June-2014].

[10] The Apache Software Foundation. Apache Log4j 1.2. http://logging.apache.org/log4j/1.2/, 2012. [Online; accessed 2-June-2014].

[11] Computing and Information Sciences. Beocat. 2014. URL http://beocat.cis.ksu.edu/beocat. [Online; accessed 2-June-2014].

[12] nmon. nmon for linux. 2012. URL http://nmon.sourceforge.net/pmwiki.php?n=Main.HomePage. [Online; accessed 2-June-2014].

[13] MD PnP Program. About program. http://www.mdpnp.org/about.html. [Online; accessed 2-June-2014].

[14] Real-Time Innovations. RTI connext DDS professional. http://www.rti.com/products/dds/index.html. [Online; accessed 2-June-2014].

Appendix A

Test Results: Memory Graphs

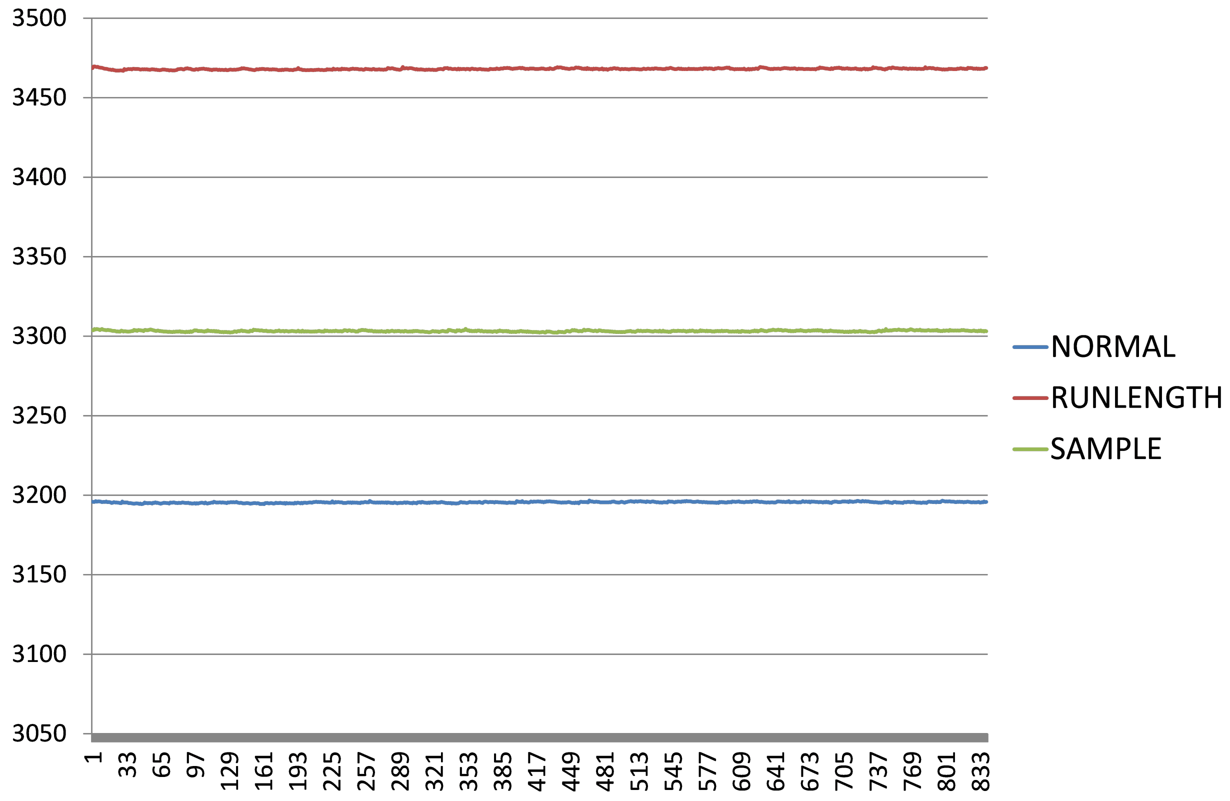


Figure A.1: *Logger: Test 0 : Memory Usage*

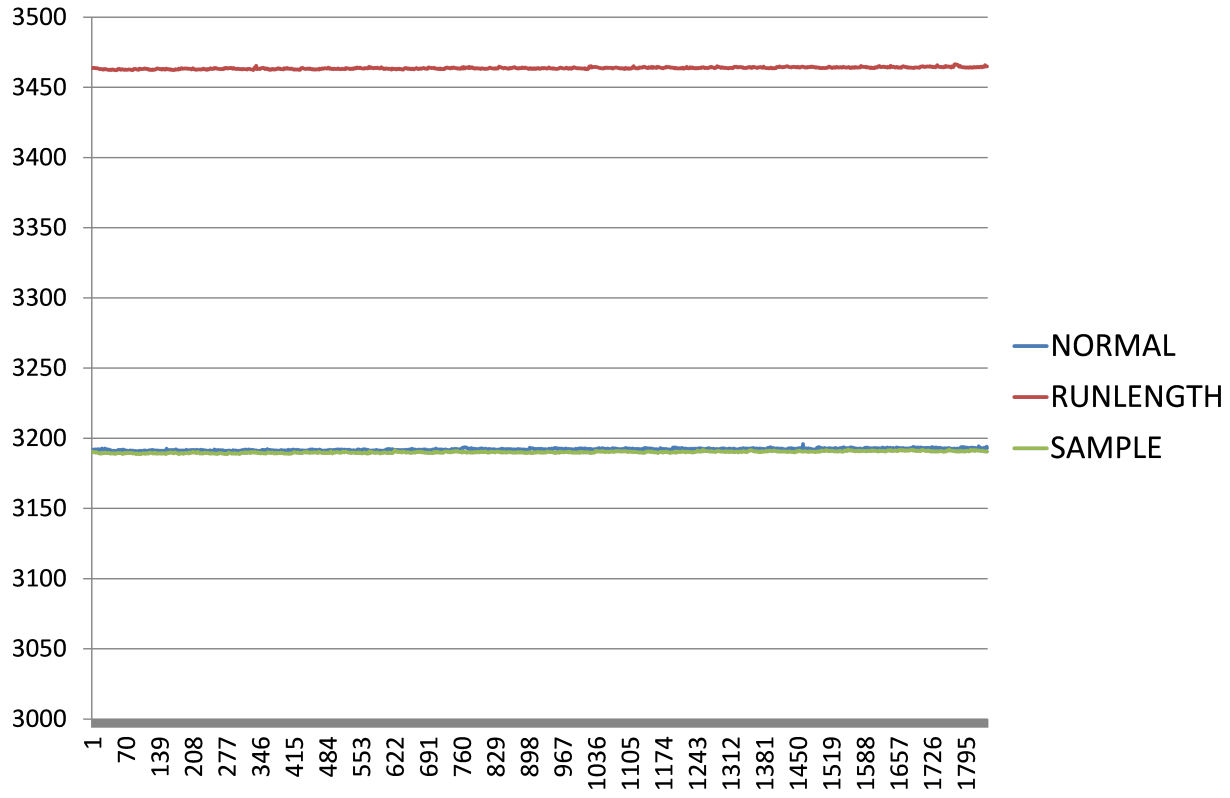


Figure A.2: *Logger: Test 7 : Memory Usage*

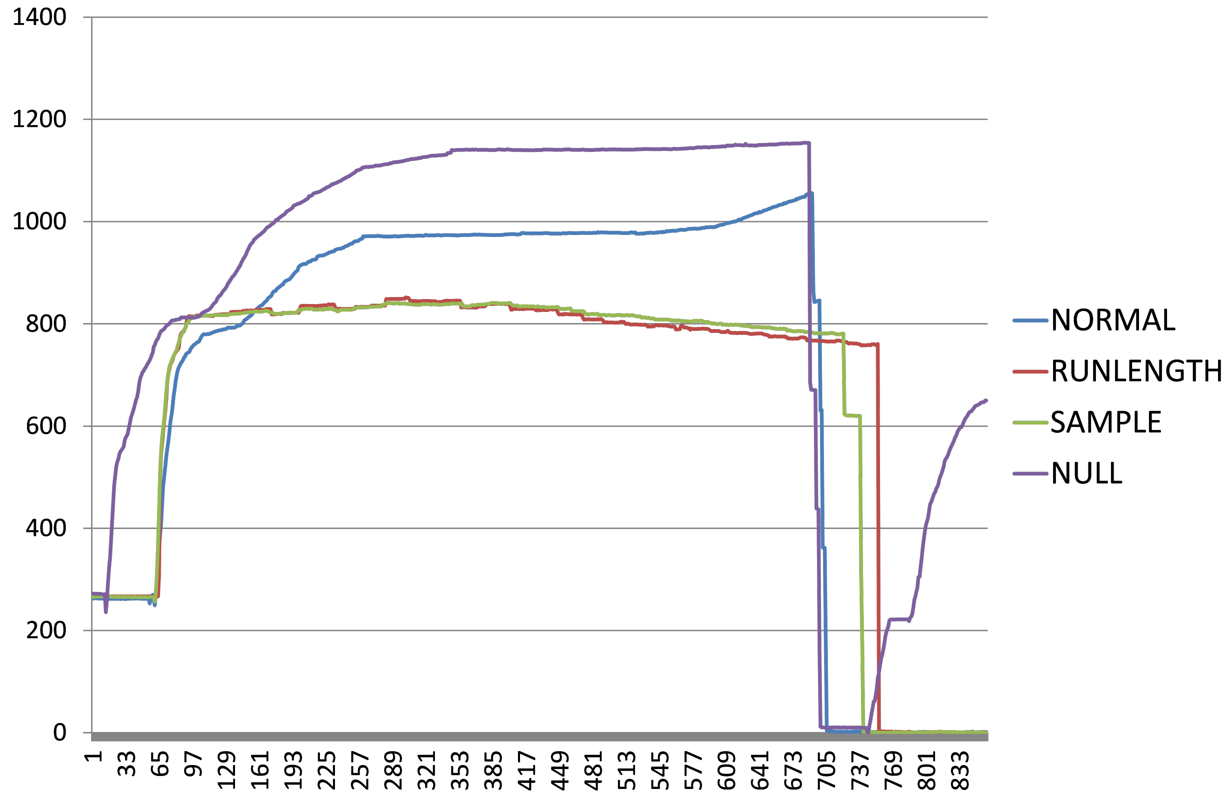


Figure A.3: *Server: Test 1 : Memory Usage*

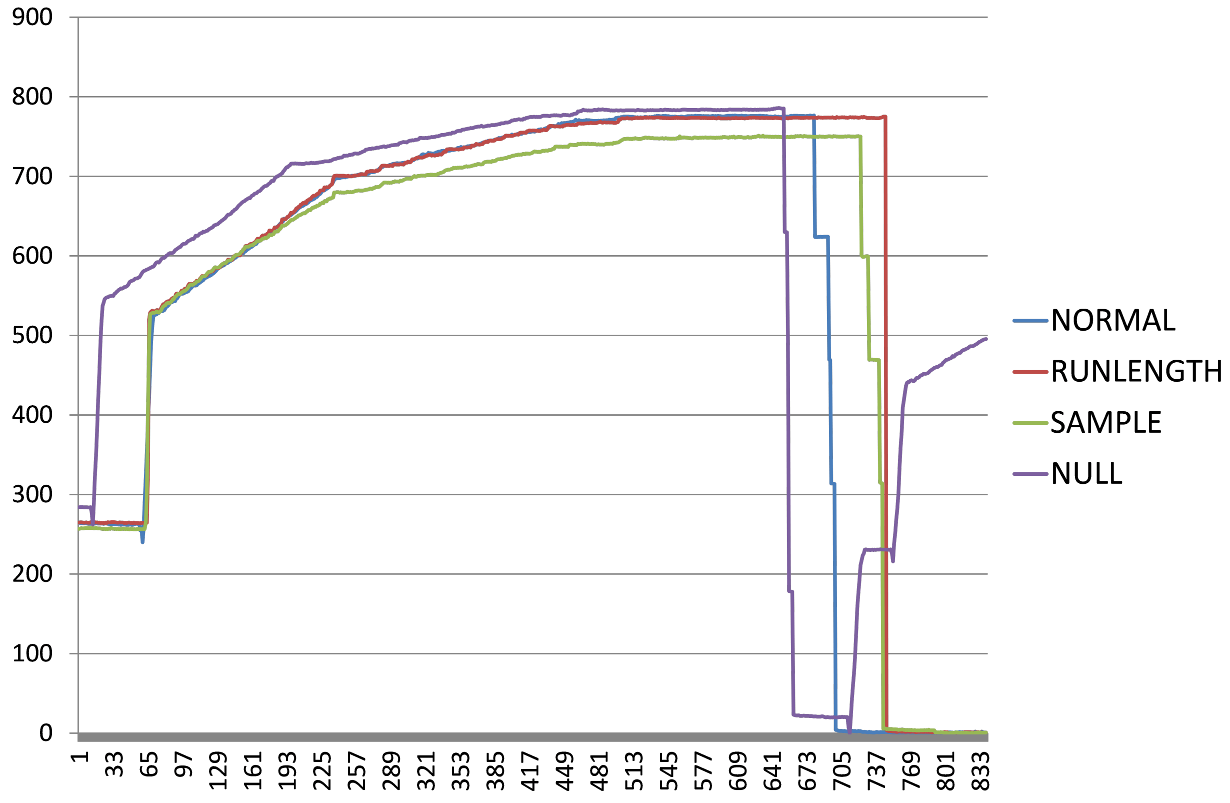


Figure A.4: *Server: Test 2 : Memory Usage*

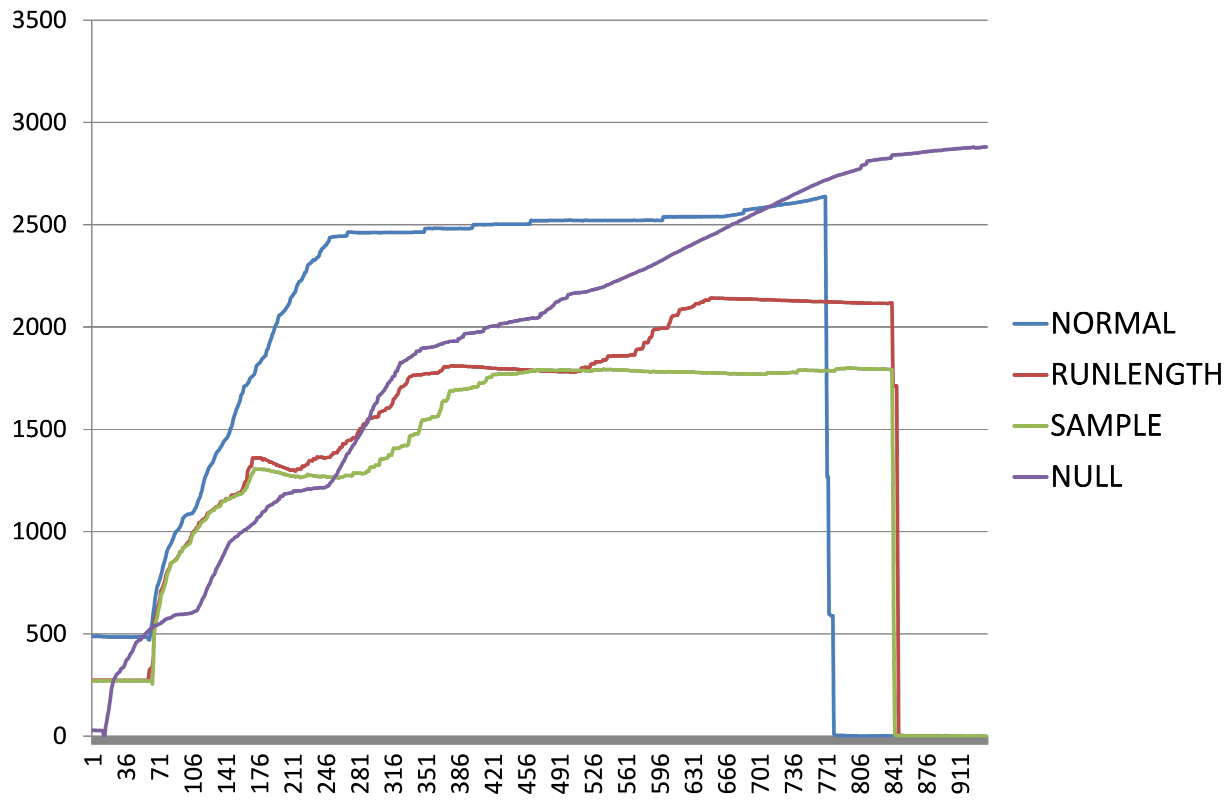


Figure A.5: *Server: Test 3 : Memory Usage*

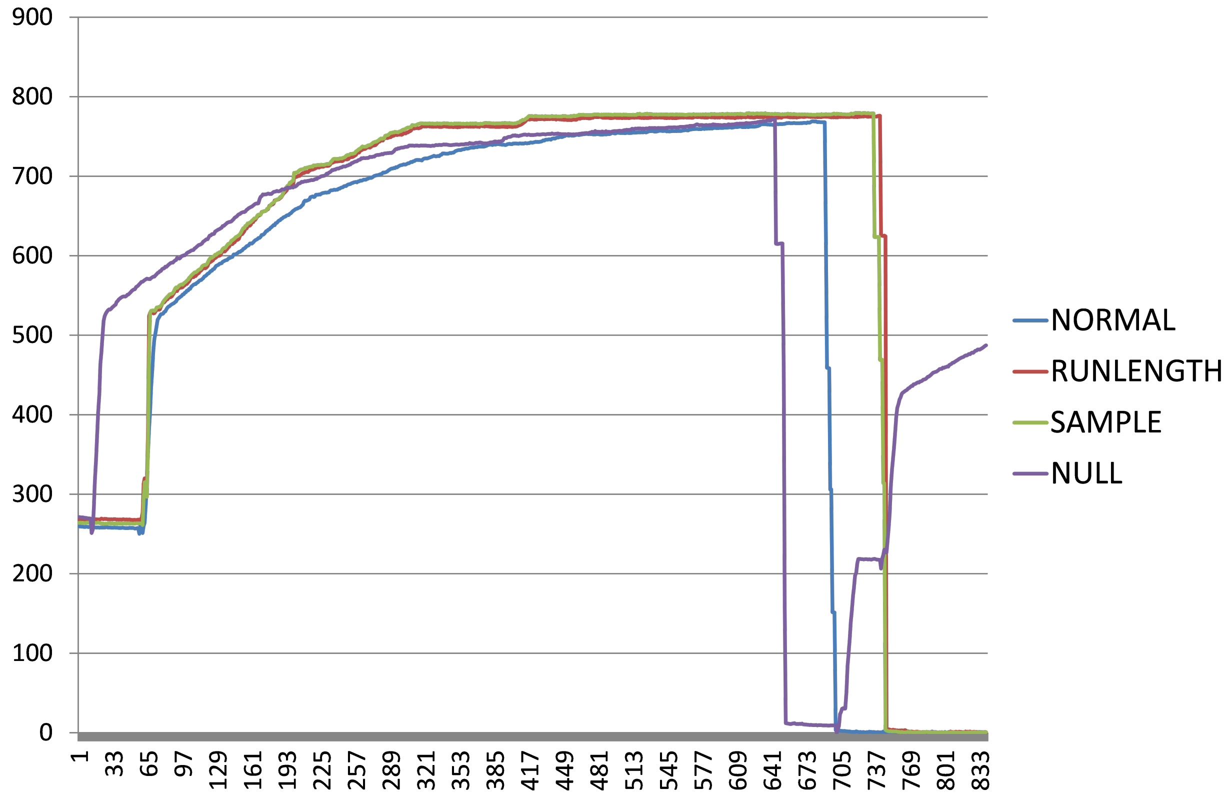


Figure A.6: *Server: Test 4 : Memory Usage*

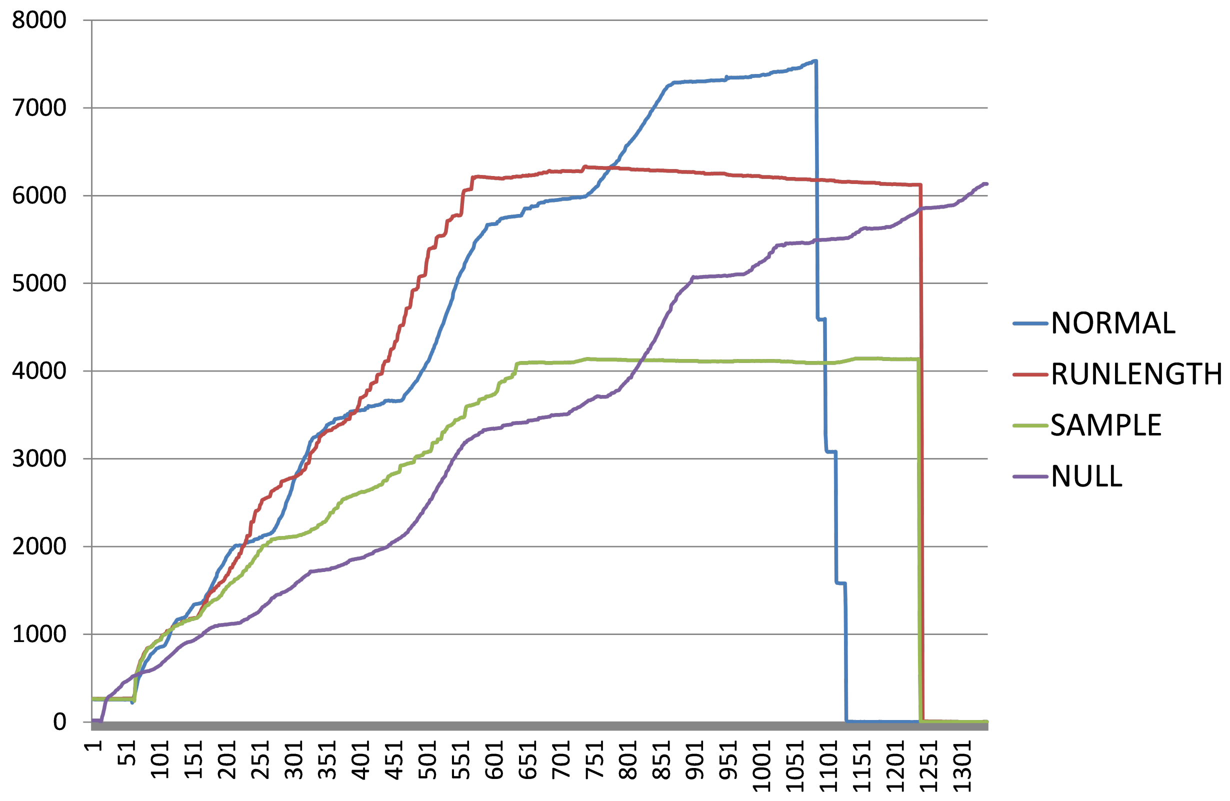


Figure A.7: *Server: Test 5 : Memory Usage*

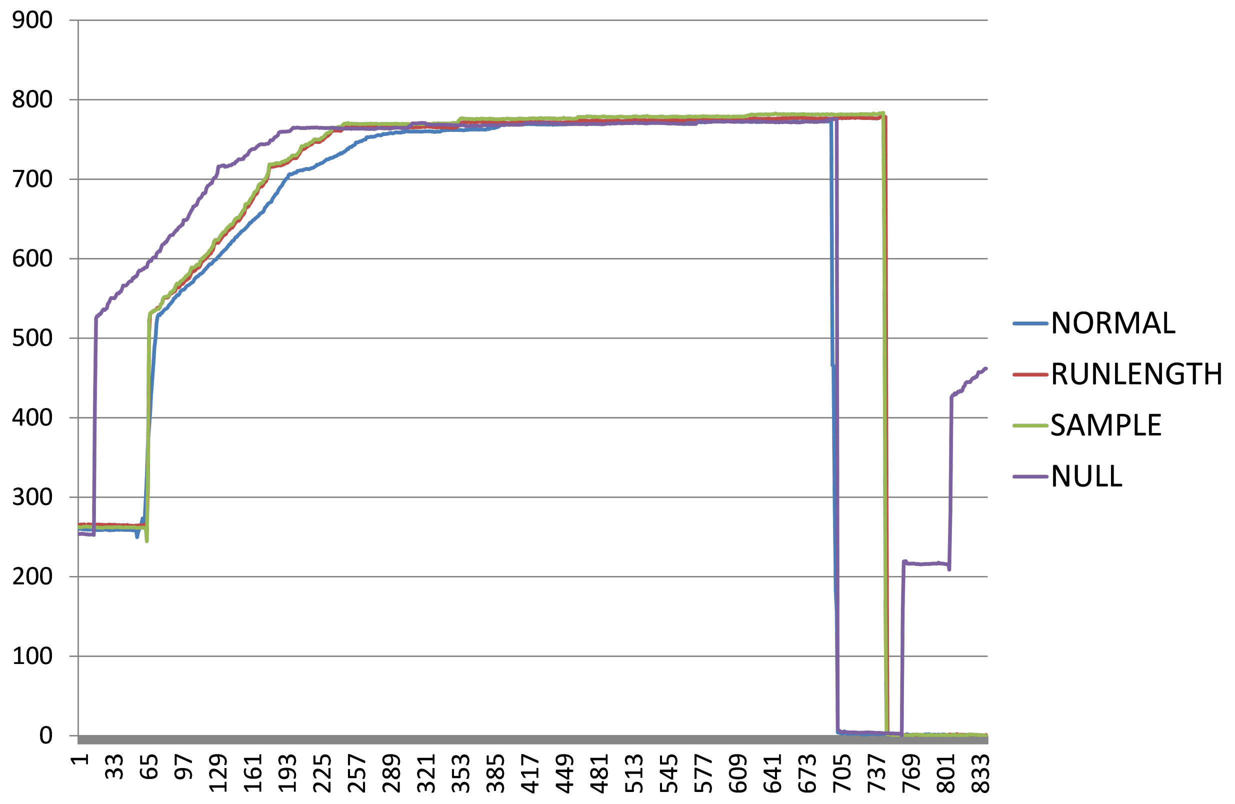


Figure A.8: *Server: Test 6 : Memory Usage*

Appendix B

Test Results: CPU Graphs

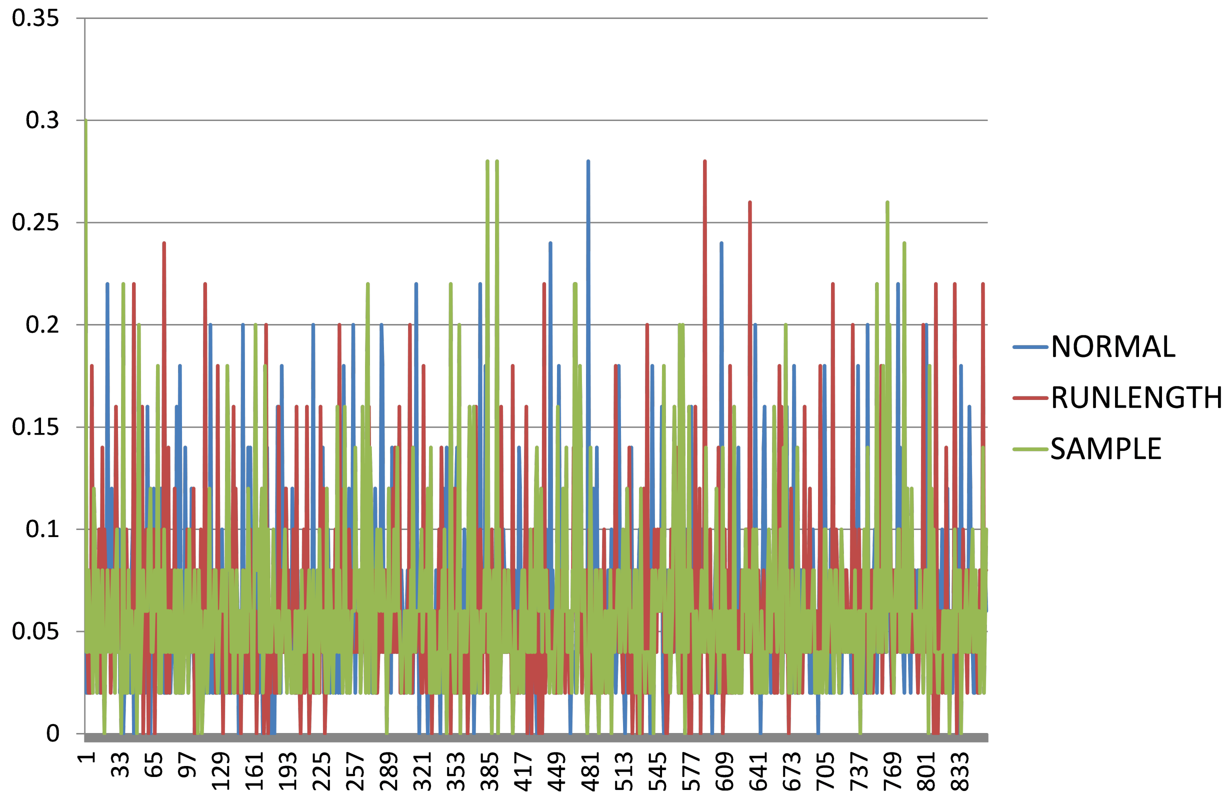


Figure B.1: *Logger: Test 1 : CPU Usage*

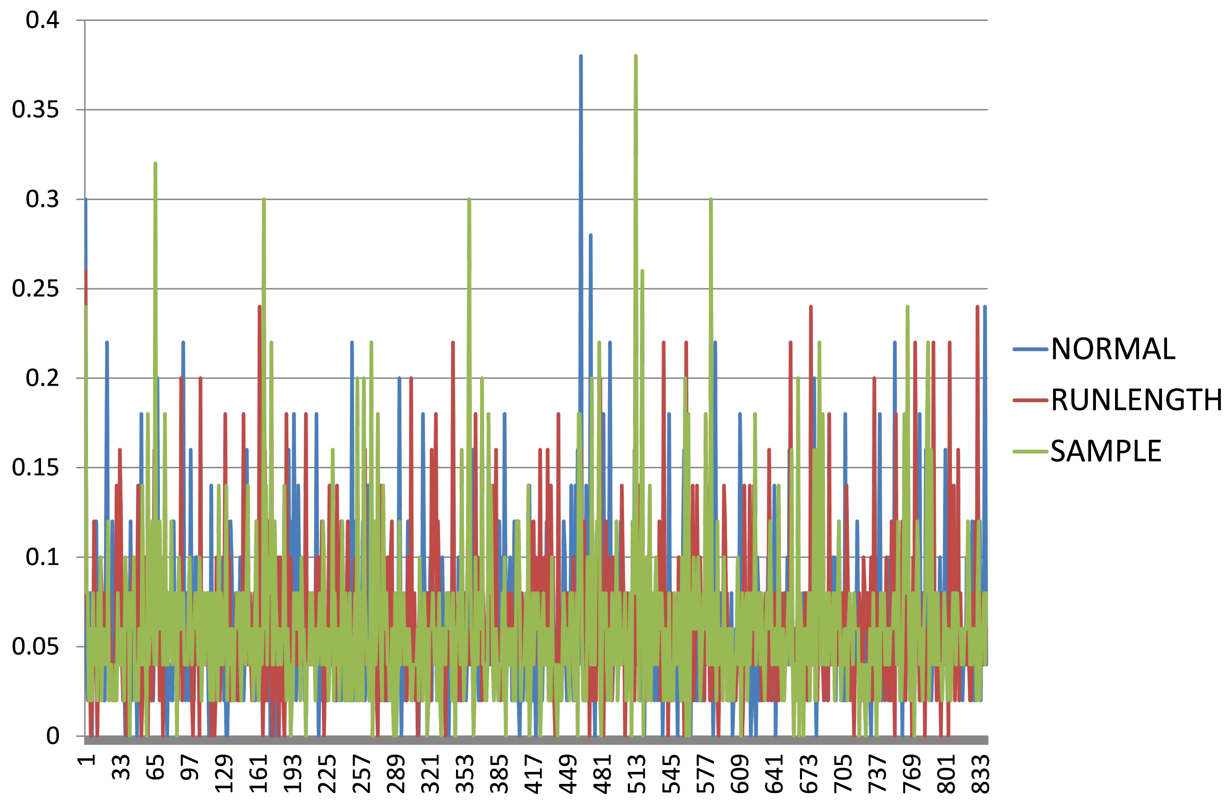


Figure B.2: *Logger: Test 2 : CPU Usage*

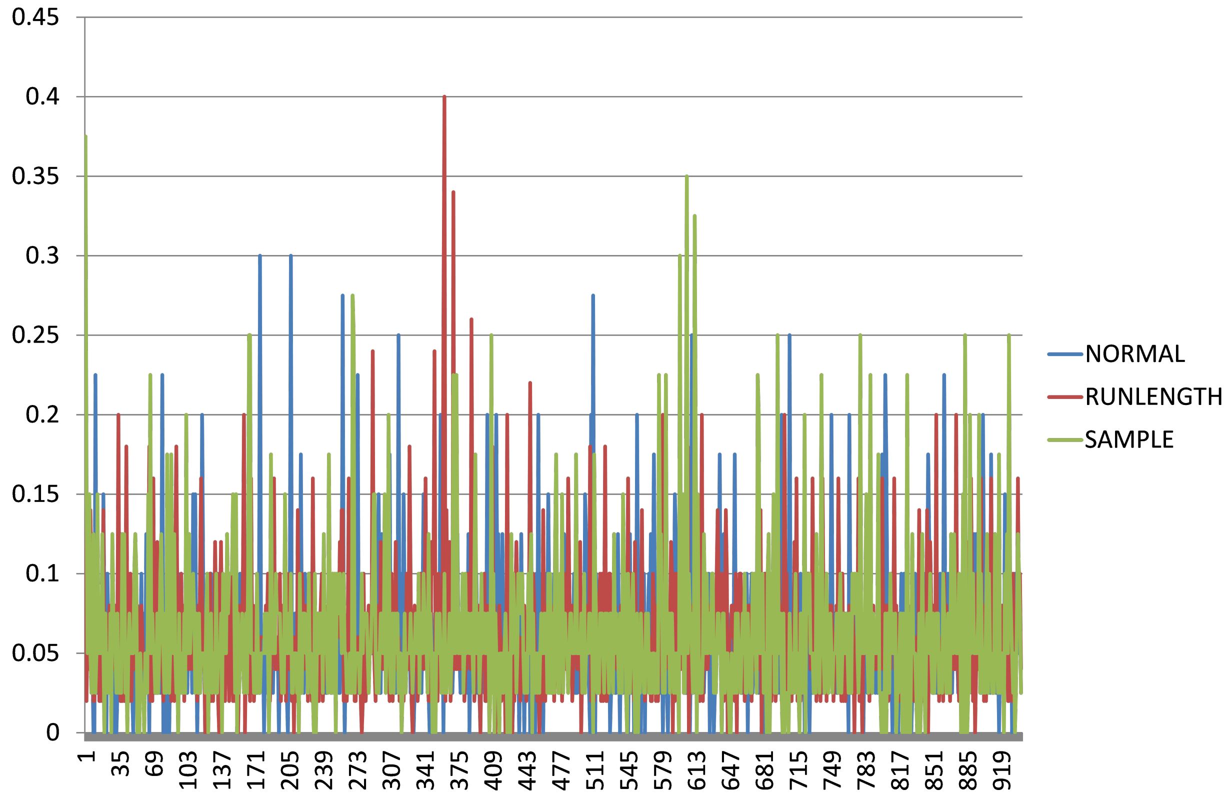


Figure B.3: *Logger: Test 3 : CPU Usage*

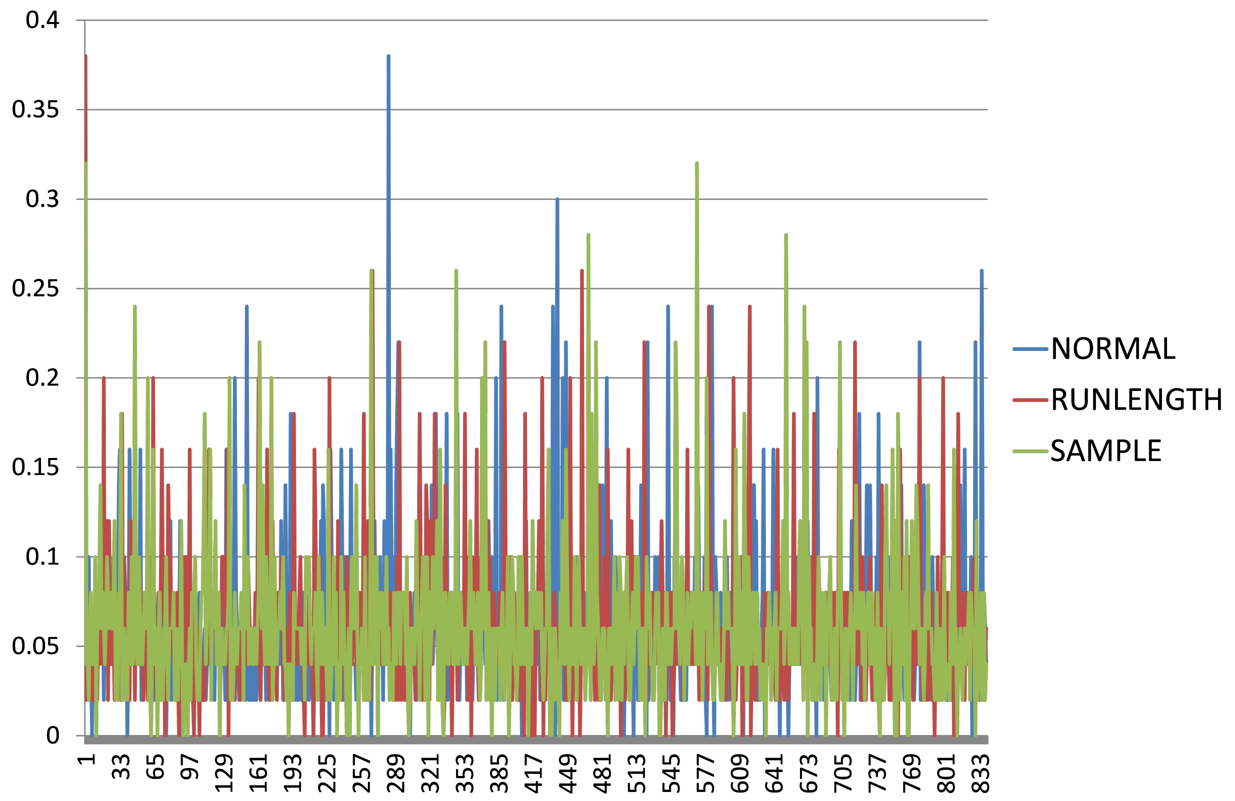


Figure B.4: *Logger: Test 4 : CPU Usage*

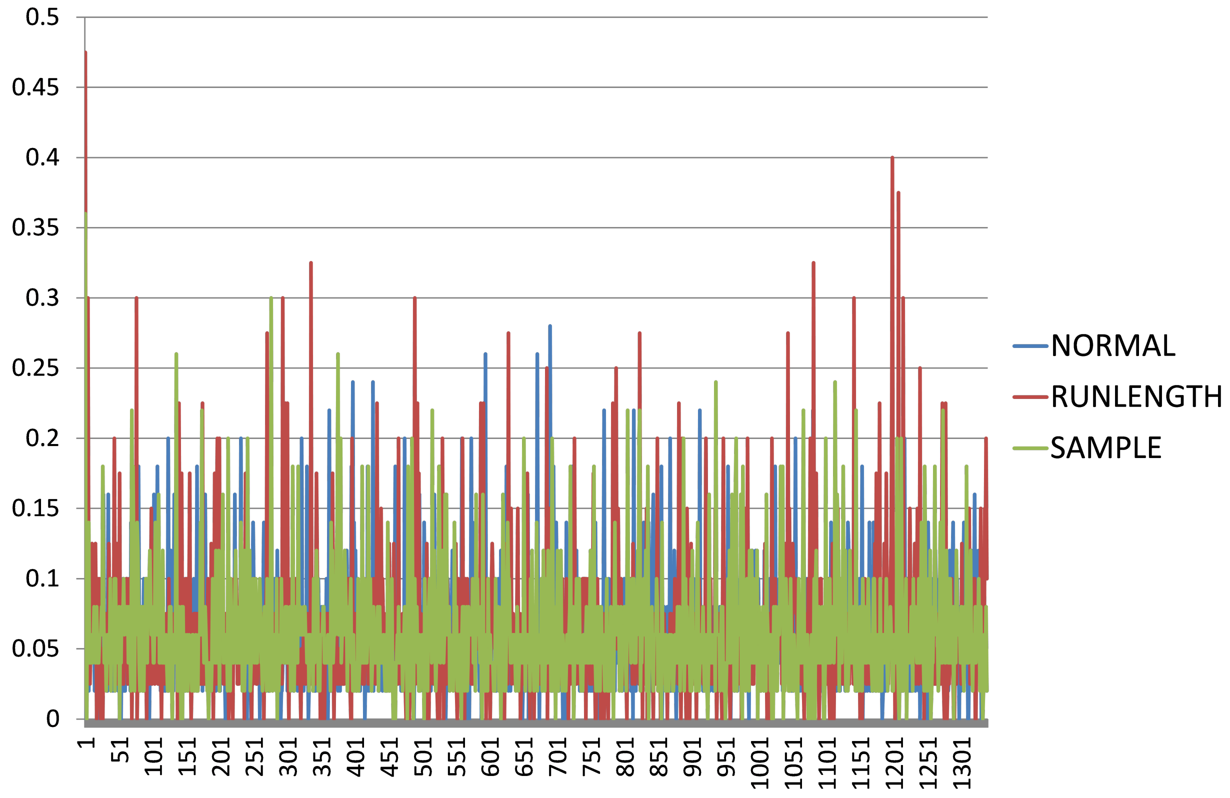


Figure B.5: *Logger: Test 5 : CPU Usage*

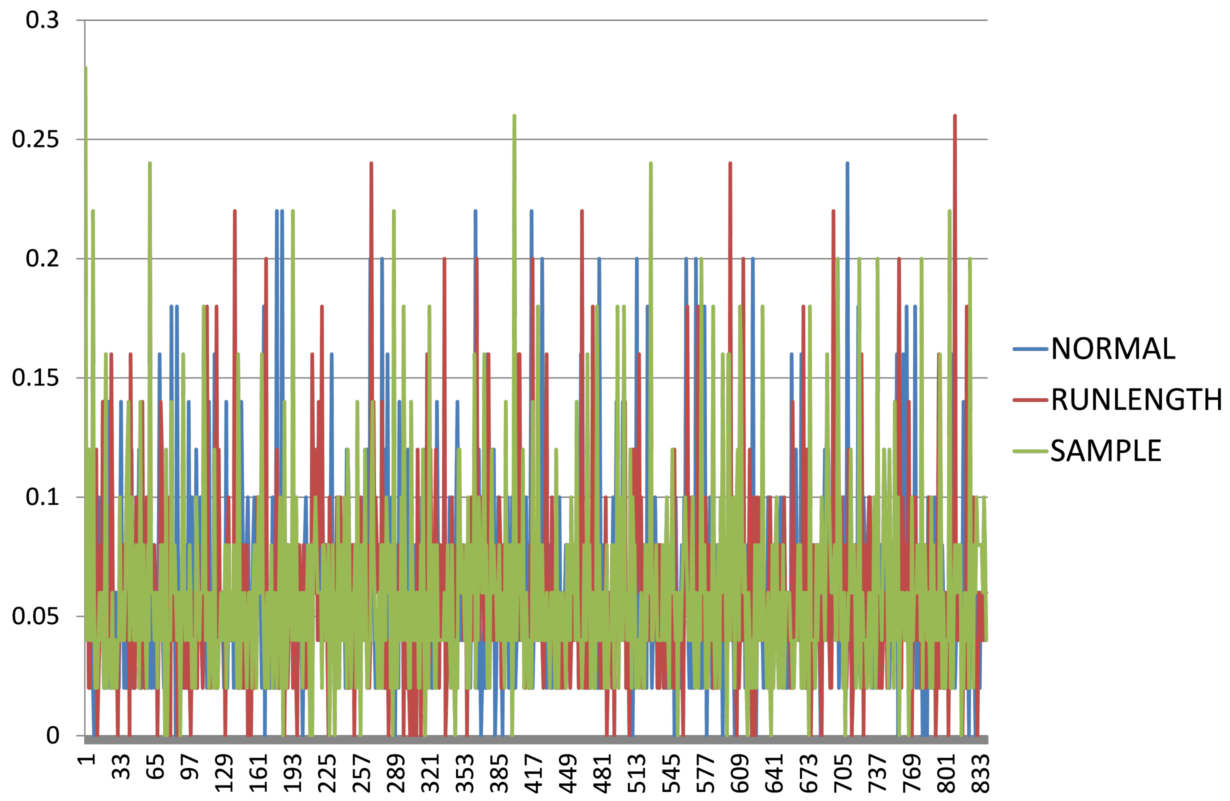


Figure B.6: *Logger: Test 6 : CPU Usage*

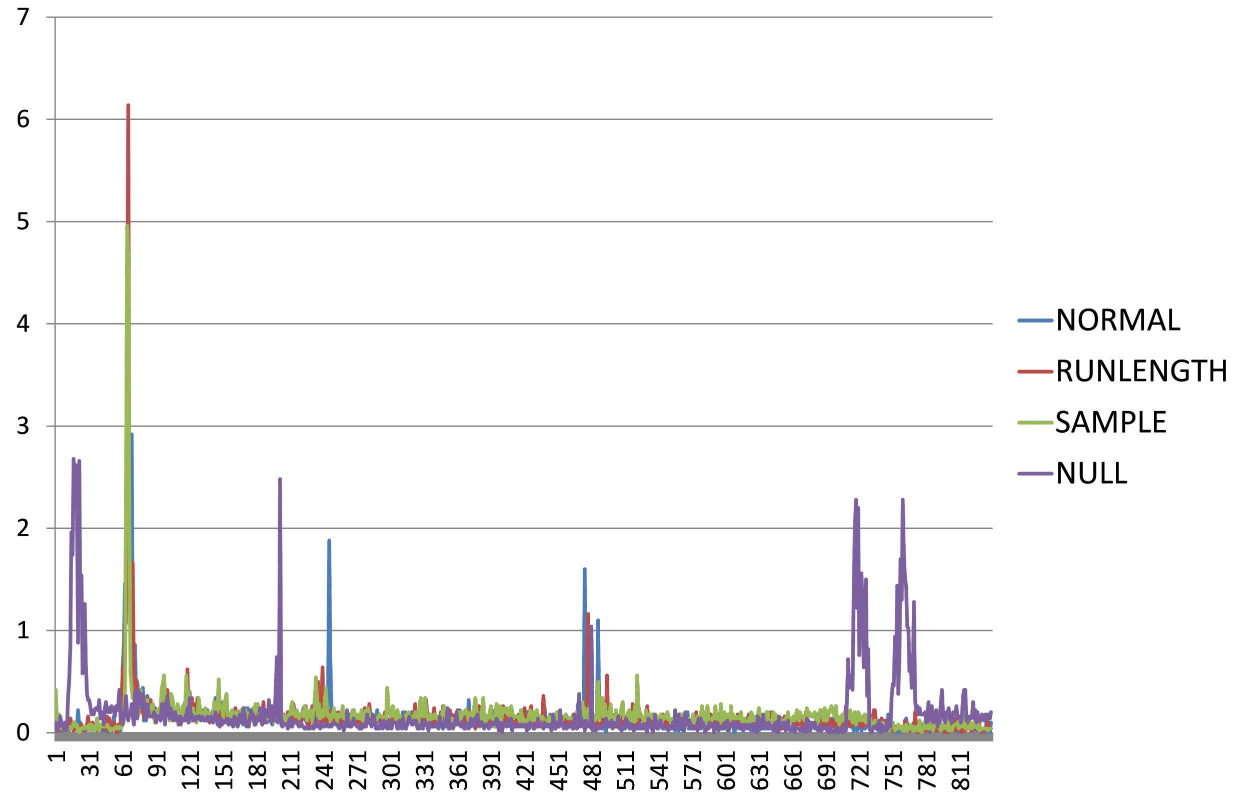


Figure B.7: *Server: Test 0 : CPU Usage*

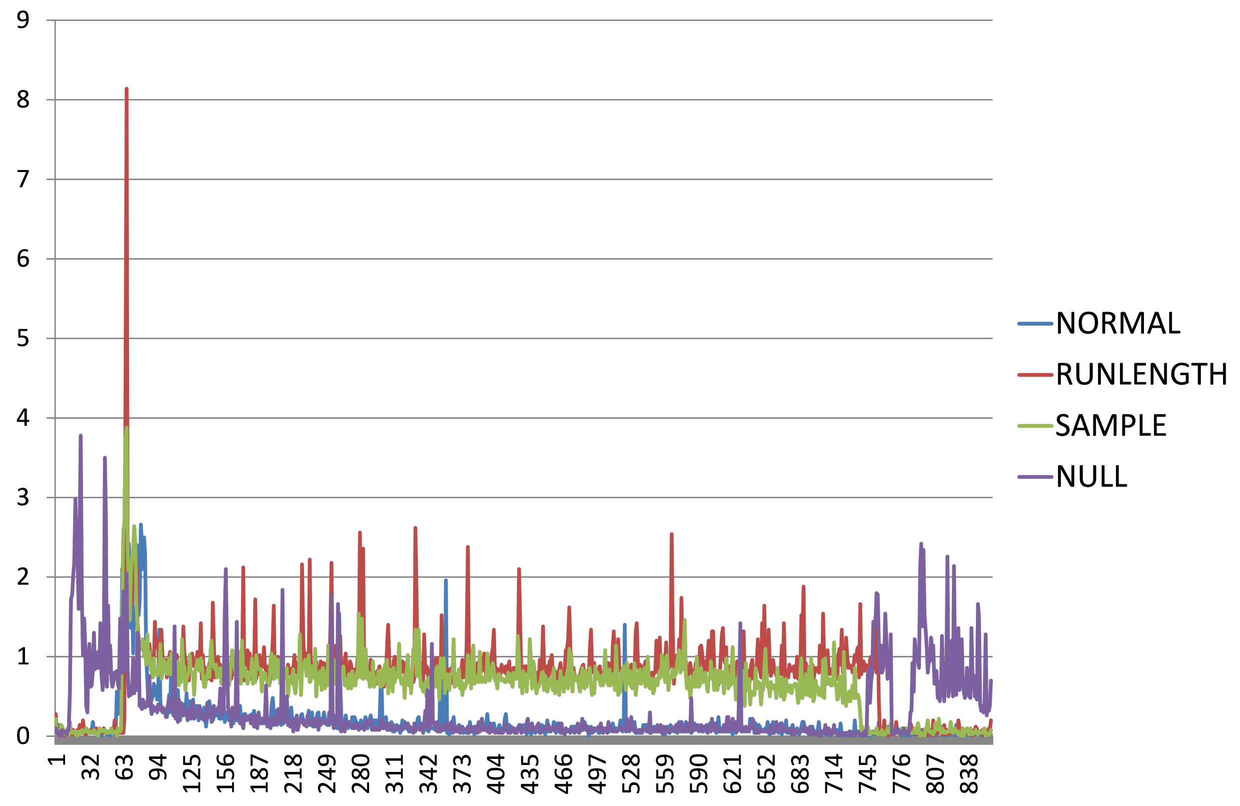


Figure B.8: *Server: Test 1 : CPU Usage*

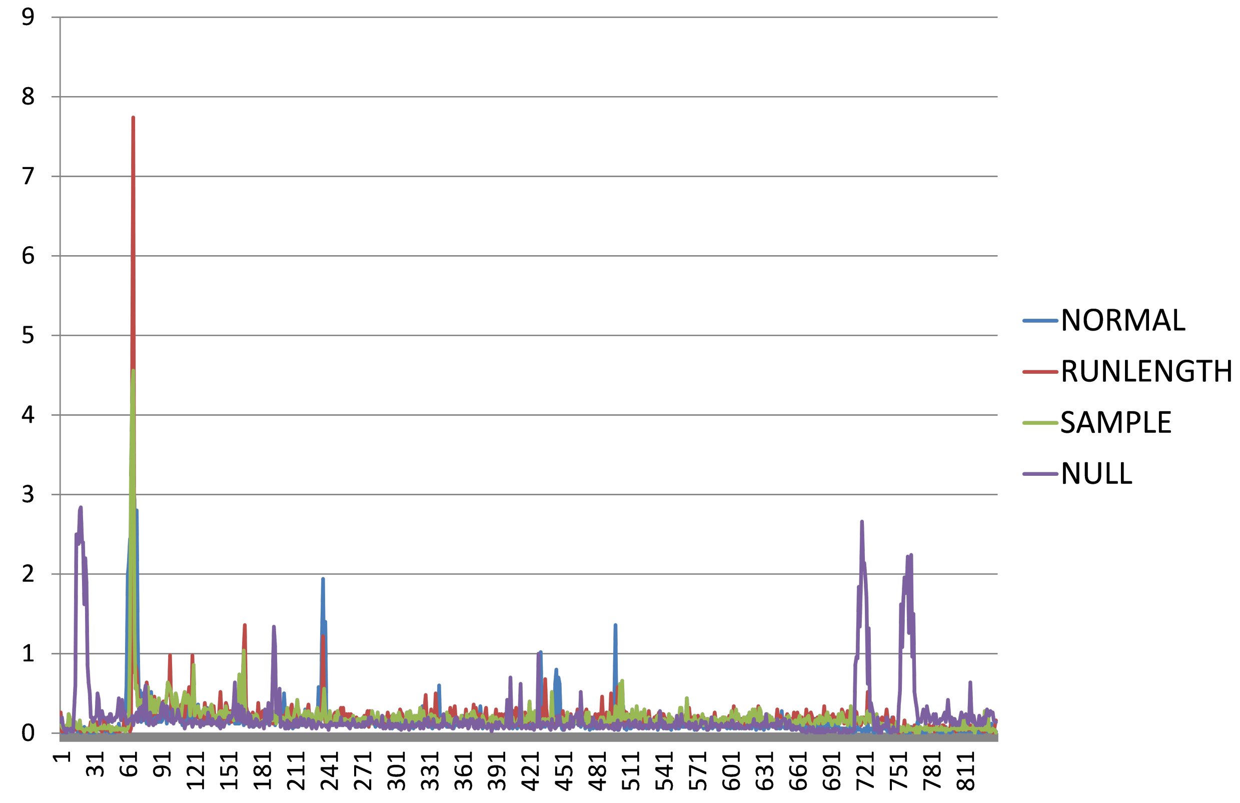


Figure B.9: *Server: Test 2 : CPU Usage*

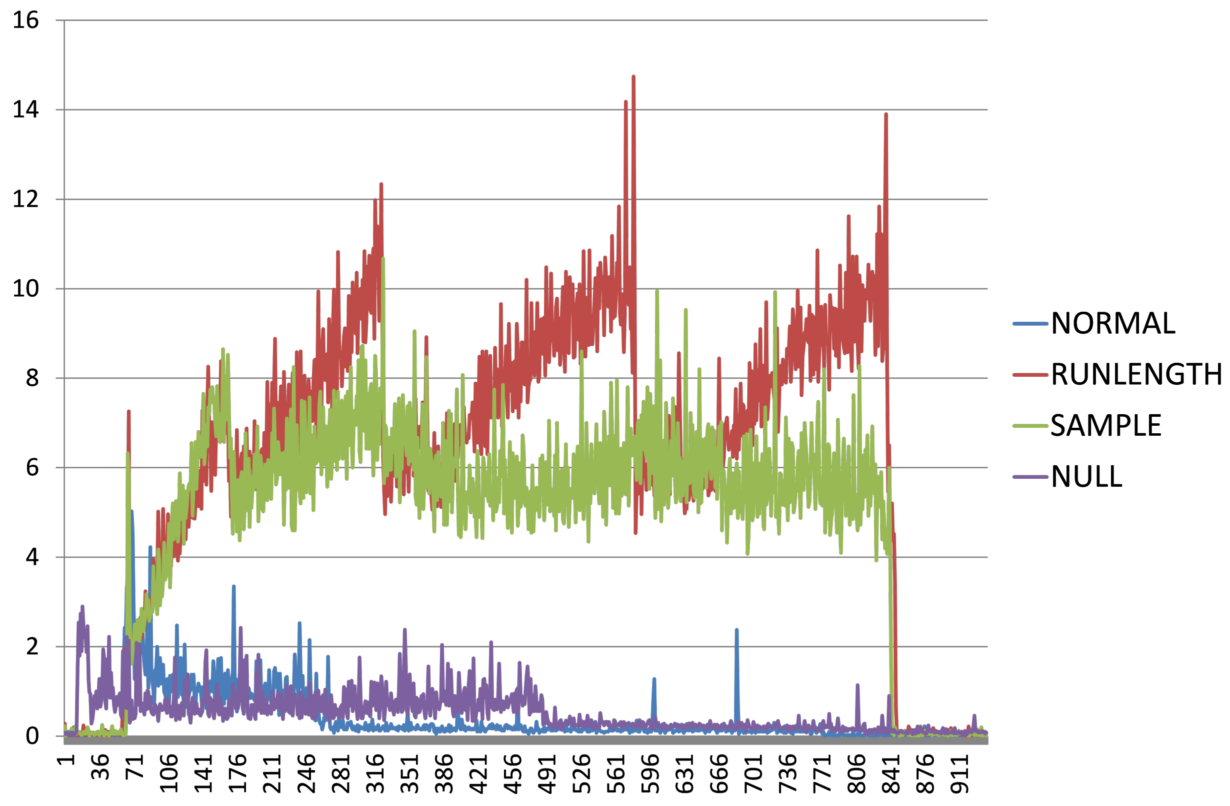


Figure B.10: *Server: Test 3 : CPU Usage*

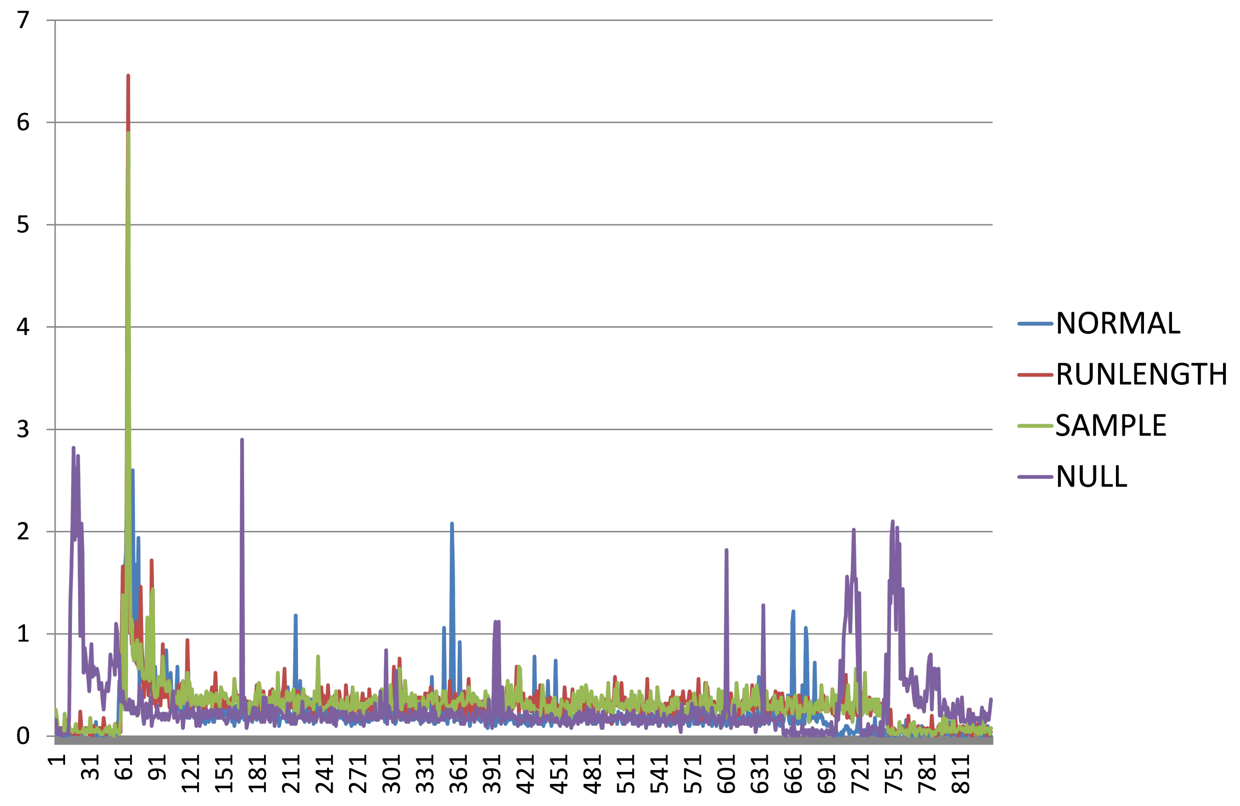


Figure B.11: *Server: Test 4 : CPU Usage*

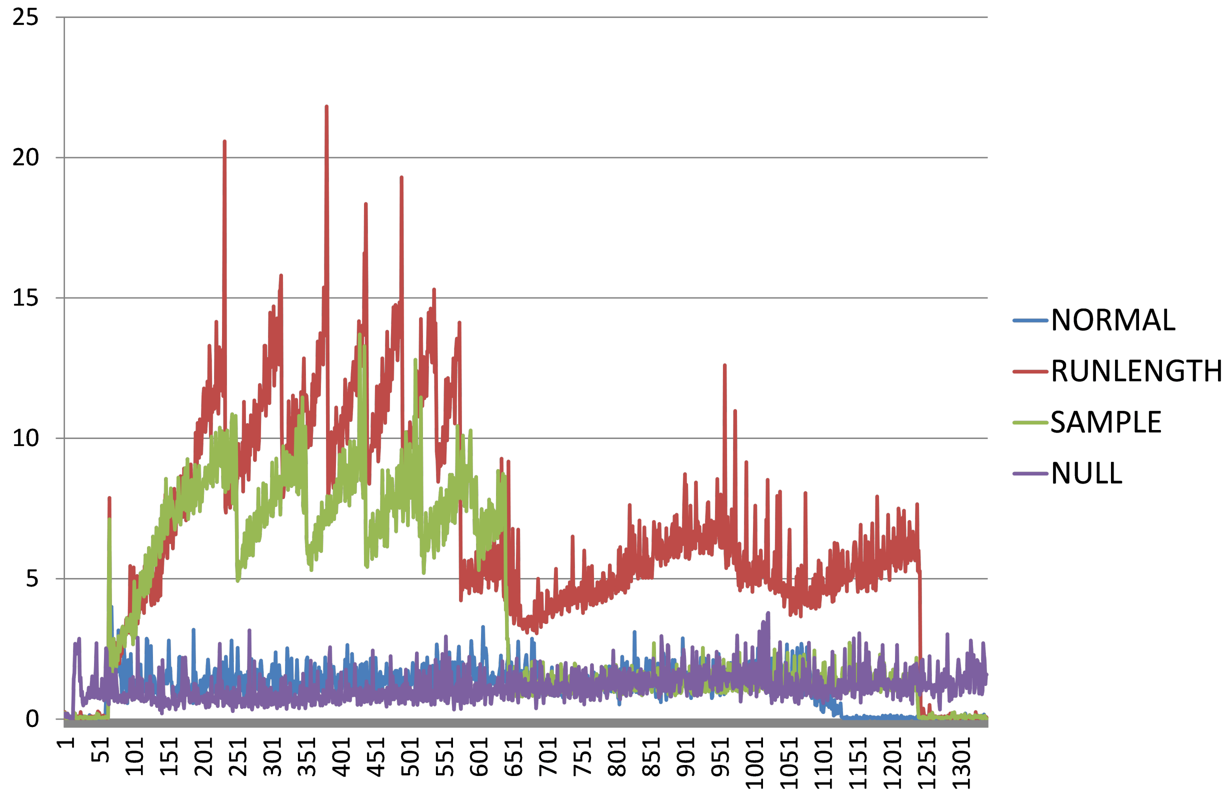


Figure B.12: *Server: Test 5 : CPU Usage*

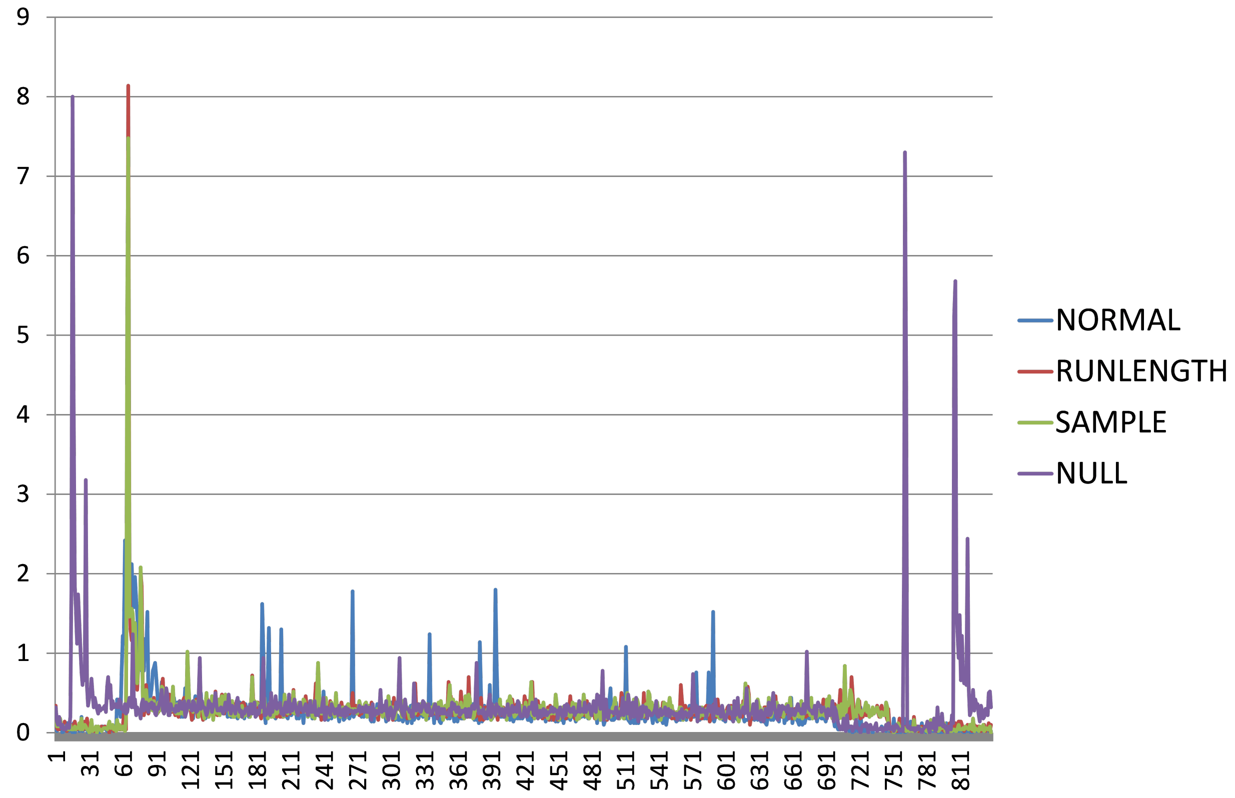


Figure B.13: *Server: Test 6 : CPU Usage*

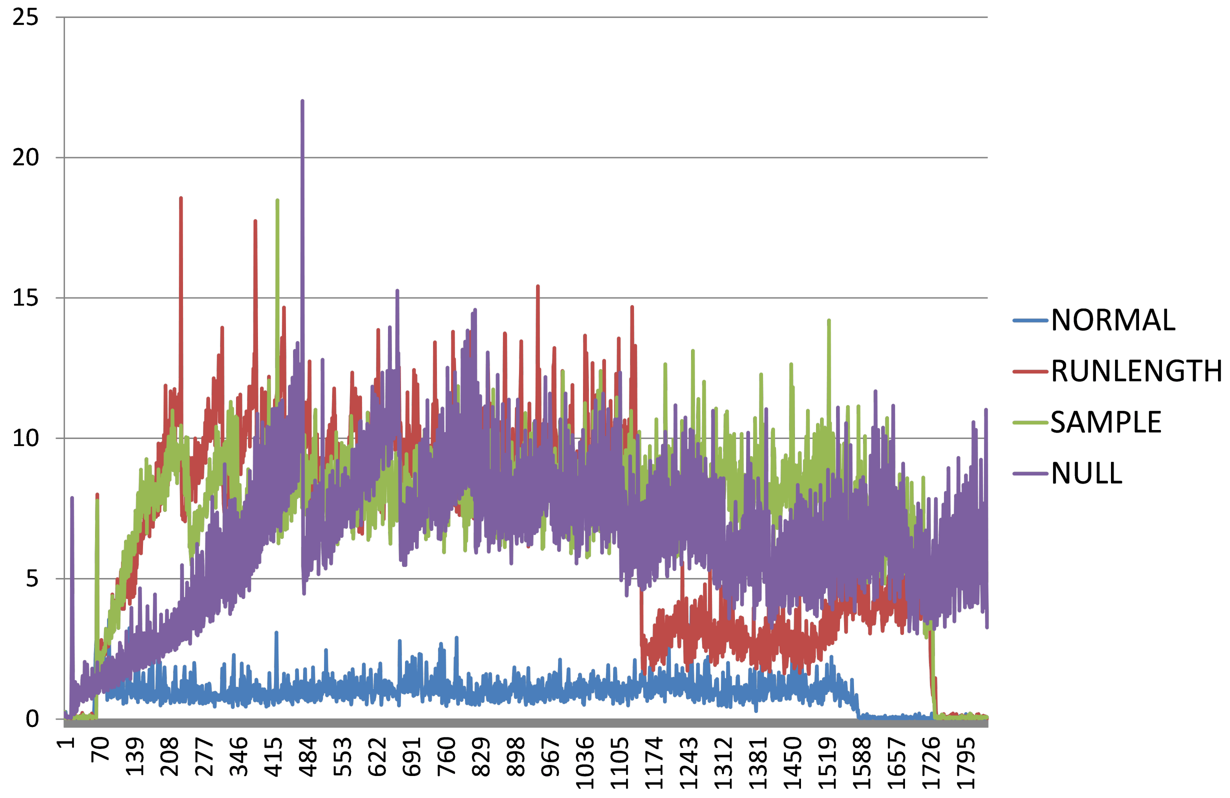


Figure B.14: *Server: Test 7 : CPU Usage*