An Investigation into Network Monitoring with Natural Language Processing

By Connor Bell

Student ID: 657010

Course: Cyber Security

School: Liverpool John Moores University

Submission Date: 21/04/2017

Contents

Abstract

Issues with a network can happen at any time, whether system administrators or security professionals are at work, off-site or asleep. There are many security applications that produce thousands of lines of logs per second and even programs to compile them into dashboards or email reports, but many of these solutions can only inform the right person of the problem if they are actively at their work computer, or via a method that will have them see it far too late. This project explores the idea of monitoring a network via data acquisition & compilation and informing remote persons of issues, then allowing them to directly work on those issues using tools they are familiar with by parsing their commands with Natural Language Processing.

Introduction

The aim of this project is to investigate network monitoring with elements of natural language processing by firstly researching literature related to security, data visualisation and natural language processing. We will then analyse this research to look for potential problems that can be solved in our research area, and then specify them precisely. Following our specification, we can then design a system which will hopefully combat the problem, which we can then implement. The project will end with a critical analysis of the work done.

The motivations behind this project stem from an interest in Cyber Security, cloud based natural language services and finally systems administration. It was personally identified that a common theme in systems administration is not knowing when an attack is occurring, and even when the administrator was made aware they would then not have access to any of the tools required to fix it. Added onto this problem an interest in natural language, chat bots seemed to be an interesting solution to a complex issue. While the scope of this problem can be considered very large, our problem specification addresses this issue and gives us a smaller scope to work with.

Literature Reviews

As part of this project, we researched into various pieces of literature to look for potential problems or opportunities within our areas of research that we could cover. Our areas of interest were Cyber Security, Natural Language Processing and Data Visualisation. This seemed to be a good range of research areas that could bring up some interesting results to talk about in our problem analysis.

Cyber Security

We started with Cyber Security and, according to a paper by Khan et al. 2013[1], large organizations require fast and efficient network monitoring systems that reports to a network administrator via email or SMS as soon as a problem arises, with details of the problem and locations affected. They go on to explain the merits of Nagios[2], a network monitoring tool, and its role in their system. It is extremely important, they mention, that the system be essentially autonomous in operation, because in a large company manual monitoring is very difficult due to both cost and time. The paper contains some basic instruction for configuring a Nagios setup and defines several ways for the software to check the status of various servers and services, and could act as a good guide for somebody new to Nagios configuration. However, the paper does not explore additional ways of informing administrators of issues, nor does it compare other software that may have similar features. The method used to set up Nagios means that their system interacts with a Request Tracker, and as Nagios detects faults in the network it will send affected nodes plus other information via the Request Tracker as a ticket to the network admin. If the ticket isn't resolved in an hour, the ticket is send to the second responsible network person. This method means that all persons are informed one by one until the ticket is marked as resolved. This could mean some dissonance may exist at times between different responsible persons, but this is not addressed in the paper.

The Cyber Security in the UK paper[3] explains, in detail, the British governments approaches to cyber security. It also describes various different types of attacks, such as data theft, attacks on critical information infrastructure, and attacks on physical infrastructure. The paper does well to inform the reader of terminology related to the field, such as the concept of air-gapping – network isolation, and zero-day attacks – previously unknown and unprotected attacks. The paper suggests that common cyber security measures include methods such as the deployment of firewalls, using up to date anti-virus software, regular software patching, access management, encryption, and use of intrusion detection software. It also stresses the importance of security in industrial control systems, such as smart metering of gas and electricity consumption in homes, to avoid data falsification or damage to systems. While there are some examples of major high profile attacks such

as the Stuxnet virus[4] and the data thefts at Lockheed Martin[5], the details of these attacks are very lacking and there are few sources to follow for more information. The paper offers no real scrutiny or analysis, and merely informs the reader, where it would have been nice to see a comparison between other countries cyber security plans.

A study in the *International Journey of Scientific & Engineering Research*[6] shows new trends in cyber security based on the adoption of new technologies. It claims that due to the Windows 8[7] and onwards unified architecture between devices, attacks will be easier than ever between a range of systems. It also claims that due to this, somehow it would be possible to develop malicious applications like those for Android[8]. The study makes numerous claims but a good number seem to be not fully backed up by their referenced media, or lacks any sort of study that can be referenced at all. The study also claims via its abstract to discuss lack of coordination between security agencies and critical IT infrastructure, though this was not covered in detail.

Security Data Visualisation

These three papers on Cyber Security gave us a good idea how important security systems can be, and how being out of date or misinformed can be a severe hindrance in the security of a system. From this point, we decided to move on to some papers regarding the visualisation of these issues.

'Visualizing Cyber Security: Usable Workspaces'[9] provides an insight into an experiment involving adding more visualisations into cyber security, and phasing out 'primitive' command line tools. They replaced eight cyber analysts screen setups with a 4x2 monitor configuration and recorded results when faced with generic visualisations of Net-flow[10] and Snort[11] alert data, which were met with mixed opinions. The paper quotes the opinions of many analysts, even stating broadly that cyber analysts in general dislike visualisations, and prefer command line due to its flexibility and expressive power. While they did find several situations where visualisations helped in finding complex patterns in real world data, they found it very hard to sell the idea of visualisation to seasoned analysts. This distrust may stem from poor performance of instruction detection systems, which attempt to automate and "simplify" the process, as the number of false positives emitted is supposedly very high.

Much of the experimentation done in the paper was met with harsh response by the analysts, with comments that their original approaches, using grep or SQL queries, were *"considerably faster"* than the visualised equivalent. These comments are met with unconvincing defences by the paper, stating that is was not a fault of the visualisation tool in use, but rather bad database management. The paper could have spent much more time discussing possible alternatives rather than trying to defend

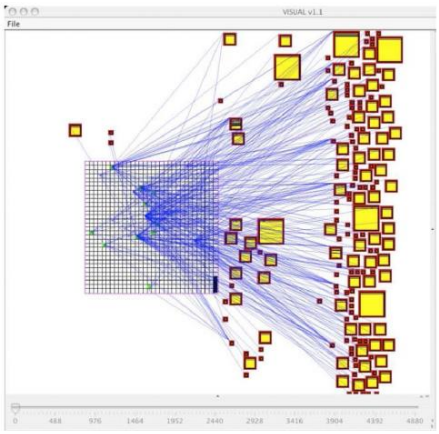

Figure 1 - 'A Survey of Visualisation Systems for Network Security'[12] *showing packet percentage data to endpoints*

its choices when met with criticism.

Security Visualization is a very young term, and many common visualisation techniques are not designed for security related data [12]. Manually traversing textual logs is not only frustrating and time consuming, but may result in important details being overlooked. *'A Survey of Visualisation Systems for Network Security'*[12] explores methods of showing administrators quantitative data in meaningful ways to better look for anomalies or patterns from sources such as intrusion detection systems, port scanning tools and firewalls.

The paper shows interesting ways of converting vast quantities of event types into graphs and other visual representations. This included splitting the events into different types: Network traces, security events, network activity context, user/asset context, network events and application logs. The image in Figure 1 shows 80 hours of network data on a network of 1020 hosts. The internal network is represented by the grid on the left, and external servers by squares on the right, with square size denoting the level of activity. This is not extremely obvious when first viewed.

One thing the paper does not do is really discuss the requirements of visualisation compared to merely parsing and reformatting text data. While the image above does a good job of showing that large amounts of data may be transferred to external servers from internal sources, we have no easy way of knowing if any of this data is malicious or coming from potentially unwelcome sources. If this data were formatted as an excel table for example, we could apply filters to look for patterns in

data. This is a common theme in the paper, where visualisations show what at first seems to be useful information, but allows little to no exploration of potential anomalies once identified short of digging through raw data.

Natural Language Processing

Having read about several visualisation systems we noticed what seemed to be a common theme – All these systems required an experienced administrator to be looking directly at the data. We then started looking at automatic systems with notifications, which brought us into the idea of chat bots and natural language processing to make those chat bots easier to use.

No consensus has emerged whether a piece of software will ever be able to convert English text into a programmer friendly data structure that describes the meaning of the text reliably, according to a paper by Collobert et al. 2011[13]. The paper was written to accompany an attempt to build a natural language parser using a huge database of training data, and documenting the process of machine learning. Their approach was benchmarked using four standard NLP tests:

- Part of Speech tagging – This aims at labelling each word with a tag that indicates its *syntactic role*, such as plural, noun, adverb etc.
- Chunking (or shallow parsing) – This aims at labelling segments of a sentence with syntactic constituents such as noun or verb phrases, where each word is assigned a tag and encoded as a 'begin-chunk' or 'inside-chunk' tag.
- Named Entity Recognition – This aims to label elements into categories such as "PERSON" or "LOCATION".
- Semantic Role Labelling – This aims at giving a semantic role to a syntactic constituent of a sentence.

Their experiments followed the standard evaluation procedure of the CoNLL challenges, a set of tasks with the goal of challenging the computer science community to create machine learning strategies which address proposed natural language processing problems.

The paper criticizes itself, noting that they used multilayer neutral networks, a 20-year-old technology, rather than something more modern, though they also note that the training algorithm used was only possible because of the tremendous progress in computer hardware. Due to their unique approach of trying to build from scratch rather than using work already established, much potentially relevant information from other papers and previous experiments could be construed as missing.

Natural Language Processing: An Introduction[14] acts as an overview of common machine-learning approaches currently being used and possible future directions of NLP – Natural language processing, as well as some of the associations with IR – Information Retrieval. One of the first things defined is that of statistical NLP –NLP based on machine learning methods, learning via large annotated bodies of text which provided the standard they were looking to achieve. The paper quickly becomes complex, looking into data driven approaches to NLP and their drawbacks such as Hidden Markov Models (HMMs), which is a system where variables can switch between several states and generate possible outputs. The issue with HMM's is that we can only see the output, not the process that it takes to comes to that output.

While this paper primarily focuses NLP as its research topic, it also reviews methods of machine learning that could be applied to NLP. However, it still lists and explains a good number of NLP sub-problems which are primarily aimed toward the medical field but are still relevant in computer science, such as sentence boundary detection and morphological decomposition – The act of separating words into smaller words.

The paper also has an interesting section focussing on the future of artificial intelligence and NLP, quoting heavily IBM's Watson[15] supercomputer and its attempt at beating humans in the game Jeopardy. With 16TB ram, Watson is designed to hold all of reference content in memory making its seek time fast, as opposed to being disk-I/O-bound, which makes seek time exceptionally slow. However, Watson can be easily misled with certain questions – Asking it "Which US city has two airports, one named after a World War II Battle, the other after a World War II Hero?" would be a multi-step process which Watson could not answer, as the reference content used for machine learning was structured as one sentence question and answers ("What/who is/are X?").

The Anatomy of A.L.I.C.E[16] is a paper that explains and represents the technical side of the Artificial Linguistic Internet Computer Entity, the winner of the Loebner Prize[17] as "the most human computer". The paper goes into detail on Turing's 1950's paper on the Original Imitation Game (OIG)[18] where a man and woman are asked questions remotely, via text, by an interrogator. The man is instructed to lie and ensure the interrogator is not able to find out that the man is a woman. Turing proposes replacing the man with a robot, and so came the description of the Turing Test. The paper goes on to criticise ALICE's winning of the Loebner Prize, as it is purely designed to beat the Turing Test and is not designed for use in real world applications. ALICE consists of a huge database of AIML (XML style) elements, each combining questions and answers or stimulus and responses in an attempt to match any potential question with a reasonable response.

In this way, ALICE is based on the original ELIZA program[19], an earlier natural language processor in development from 1964-1966. The original ELIZA program was released under the guise of 'Doctor' for use by nontechnical staff in MIT(Massachusetts Institute of Technology[20]). The issue with this was that nontechnical staff thought that this 'Doctor' was a real therapist and spent hours revealing personal problems to a script that merely either rewrote what they had said with the pronouns reversed, or answered with pre-prepared statements based on simple pattern matching. The AIML elements are designed to symbolically reduce sentences into simpler formats to understand, or match elements and patterns. For example, ALICE programmers preferred to reduce sentences to the simplest possible form, which would be referred to as the template.

It then attempts to "divide and conquer" sentences by reducing it to sub sentences. For example, if a sentence begins with "Yes" (to answer a question), and then has more words, then both sections would be treated as sub sentences and would go down separate paths in the AIML tree.

```
<category>
<pattern>DO YOU KNOW WHO * IS</pattern>
<template><srai>WHO IS <star/></srai></template
</category>
```

'SRAI' allows AIML to commit recursion to further simplify patterns.

Figure 2 - ALICE template example

The paper talks at length of how the ALICE bot continues to try and build its responses from the context of conversation, including previous responses, and how it is capable of remembering pronoun bindings using predicates. However, the paper does not conjecture on possible improvements to the ALICE bot nor what other works may be in process. It also does not describe the computational requirements or any examples of how the code that runs the AIML works, which is disappointing.

Chatbots: Are they really useful?[21] is essentially a review of many different chat bots and their uses to date, many of which use AIML elements and are based on ALICE. It has basic explanation of ALICE's pattern matching algorithm and how it deals with incoming input. For example, all input has punctuation removed and is split into multiple sentences as needed. It will then try and match word by word to obtain the longest pattern match, which would be expected to be the best one. Finally it will then use those matched patterns to determine which templated response to use. The current public-domain ALICE "brain" has been built up by Dr Richard Wallace[22], and contains more than 50,000 categories of possible patterns. All these categories were hand coded, which is extremely time consuming if one were interested in building a chat bot for themselves. To combat this idea, the paper also includes their attempt at a Java program that attempts to automatically turn text into

AIML style scripting. However, this was found to not be satisfactory to trial users when compared to manually created templates.

While the paper does include some original work, many of the reviews and comparisons of the chatbot systems are references to the papers writer in other papers, and there is little evidence from other authors. The paper also rarely reflects upon bots that do not use the AIML system.

Using Dialogue Corpora to Train a Chatbot[23] presents two chatbot systems, ALICE and Elizabeth and attempts to compare and pattern matching techniques of each system. Using the Dialog Diversity Corpus, a database of formatted human texts that can be used for human interaction research, the researchers attempt to convert natural language into AIML format. After a section explaining symbolic reduction and how the AIML format handles recursion, which interestingly seems to be taken straight from the original paper by Dr Richard Wallace, with no direct reference, the paper moves on to the Elizabeth chat bot.

Elizabeth[24] is a much-improved adaptation of the Eliza program, but much simpler in terms of database management than ALICE. Elizabeth uses a single script to run the bot, and may contain at most four parts. The first part deals with welcoming the user, and what to do in the case of an empty message, or a message which it does not understand. The second part of the script deals with transformations, such as “MUM -> mother”. This can be seen as a rudimentary form of sentence reduction, as it attempts to give the meaning of multiple words into one “intent”. The third part of the script deals with output transformation. This, again, is a simplified version of pronoun reversal such as “my -> your” and “you is -> you are”.

The final part of the script deals with keyword transformation, which is effectively the picking out of keywords to attempt to create a cohesive answer. For example, if the script read:
K I LIKE [string]ING
R HAVE YOU [string]ED AT ALL RECENTLY?

(With K and R being keyword and response rule starts for the Elizabeth interpreter)
The user would be able to enter “I like gaming” and the response would be “Have you gamed at all recently?”.

Elizabeth can get very complex, including implementing grammatical rules and advancing its pattern matching algorithm, but the fact remains that Elizabeth is unable to use any sort of “big data” to automatically train itself without assistance from a third-party application, nor can it stray particularly far from exact pattern matches without extremely advanced programming. The paper ends with a list of problems encountered when dealing with their chosen text database, the Dialog

Diversity Corpus, noting that there are no standard formats to distinguish between speakers, sub-standard annotation, and scanned images not being converted into text correctly.

A paper[25] by a BRAC University[26] student follows the implementation of a chat bot with domain-specific knowledge about a university, essentially operating as a frequently asked question bot. Their goal is to show how accurate such a system can be, and how it can be improved based on a specific domain. Domain-specific knowledge is a knowledge base relating to a particular system. In this case, frequently asked questions about BRAC University in Bangladesh. This information was formatted with a modified conversational base already implemented into ALICE, allowing the bot to have its conversation focused only on topics related to the domain.

Creating the bots knowledge bases consisted of extensive brainstorming and writing down as many questions in as many different formats or layouts as possible, to assist in the bot being able to intelligently match patterns, and to make it less likely that a relevant question would be misunderstood or ignored entirely. This included the use of wildcards, for example, "* about cse370" would assume you were talking about a course, cse370, and would ignore other text. The bot follows an interesting system for getting an idea about a topic. A user can advance into another topic by mentioning a certain thing, such as "admission", and the bot will then move on to reading data from the "admission" AIML file instead of the "general" AIML file.

The paper compares between two potential bot setups. The first setup includes only very basic general responses, but the full range of FAQ responses. The second setup includes all general responses from the ALICE source, and the full range of FAQ responses. The first setup intentionally limits conversation from straying too far from the knowledge domain, but potentially could not seem as real or be as comfortable of a conversation for the end user. It was found that in fact the limited setup was preferred, as it came up with much more satisfactory answers a higher percentage of the time, as questions were more to do with domain specific knowledge. The paper does not explore outside of the ALICE chatbot, nor does it look at any other ways of increasing the size of the knowledge base aside from manually writing new AIML.

According to a paper on today's AI[27], the evolution of AI has been limited around some key points, such as databases being fed-in manually, or targeting beating the Turing test. Today's AI systems only "pretend" to act like intelligent entities instead of actually being one. Rule based systems just follow a large set of if-else based logic, rather than applying actual reasoning.

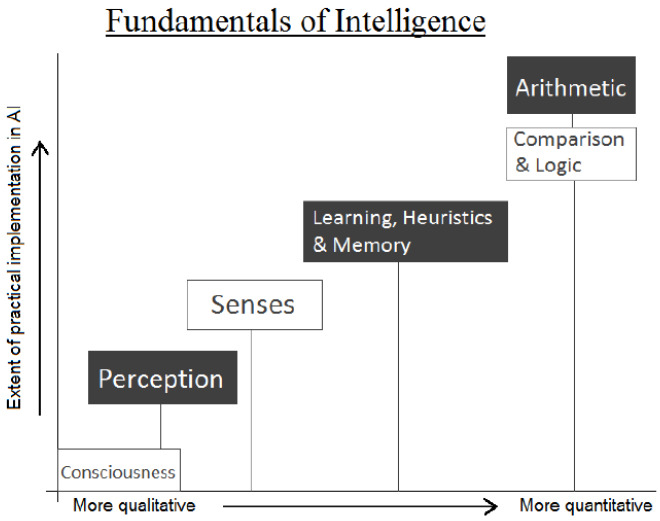

Figure 3 - 'Fundamentals of Intelligence' according to [27]

To experiment, they created two chatbot programs. One using traditional AIML, which also gave them the advantage of also using ALICE's AIML set directly, and one they created in C++ called FUTURE. FUTURE had an advantage over AIML – Being written in C++, it could reference other functions and programs when required, including offering basic arithmetic, trigonometric functions and even differentiating simple expressions.

It was stated that making the AIML bot was significantly easier and faster due to it already having many AI-oriented features. The AIML bot was tested for about 1500 queries, and gave suitable replies to around 1200, for an accuracy of 80%. However, the FUTURE bot was not tested in a similar fashion, for unknown reasons.

The end goal of the paper was to redefine "Machine Intelligence" and came to the conclusion that an "Intelligent" system must have all the fundamentals of intelligence (Figure 3). A partially intelligent system could exhibit some of the fundamentals, such as Chat Bots, which can compare, have logic & reasoning, heuristics, and memory. It can even include the ability to perform arithmetic operations.

Natural Language Processing – By Service

It was at this point that it was decided that while natural language processing may be an interesting subject, it was not what our project should focus on. Instead, we were interested in studying already existing services and researching how cloud computing could affect a potential chat bot.

Human teachers guiding learning machines presents numerous benefits and challenges according to research by Microsoft[28]. If a teacher can provide additional information and metadata as a

learning task progresses, it becomes an *interactive learning* problem. The paper explains the need and benefits of ICE, a platform that accommodates the two-way communication channel needed for efficient interactive learning. The platform has two goals, two produce valuable & deployable models, and to support research on both learning and user interface challenges of the interactive learning problem.

A “lopsided” problem[29] is defined as a problem where not all the data is available. For example, a book review not being correctly labelled as a book review. It could take hundreds of thousands, if not millions of pages for learning machine to correctly predict what a book review is, which is incredibly inefficient. To overcome the issue, they turned to interactive machine learning, which allows human input, training, scoring, and machine feedback in a real-time loop. Using this method, a single teacher performs all the functions of a domain expert, the labeller and the machine learning expert. At each step, the teacher learns from the machine how the machine is functioning, and machine benefits from the human guidance. The feedback from the machine allows the teacher to gain expertise to best guide the training process without possessing machine learning expertise.

The paper goes on to describe the using of the ICE program, and how your average user would update labels and system states. The paper also describes system architectures, including the extremely powerful servers needed for sub-second delays for the teacher. While the paper goes into detail on the inner-workings of their own system, they do not really compare to other systems that exist, even though they mention several, such as Information Retrieval. However, their argument is that ICE differs so greatly from other work that it would be erroneous to compare them, as it can be applied to effectively any data type.

The Language Understanding Intelligent Service (LUIS) by Microsoft[30] is an attempt at allowing the easier creation of bot dialog systems for developers without machine learning expertise or experience. Being entirely cloud based, developers can set up their model and deploy straight to a HTTP endpoint, where almost anything can interact with its API. LUIS operates on the idea of “intents”, where the text inputted is compared against various possible programmed “intents” of the user. For example, if you were to enter “start tracking a run”, the possible output from the API may be that the “Start activity” intent with a score of 99%, and the “stop activity” intent with a score of 1%. This meaning that the service is 99% certain that the user wishes to start a new activity, with the entity type of “run”. This information can then be parsed by whatever program is interacting with the API.

The paper is slightly out of date, stating that the service is still in invitation-only beta, though now it is in open beta, but it still has several seemingly up to date images and a relatively descriptive

tutorial of how to use the service. However, it does not show any interactions with other software to see how that information can be used or what problems could be faced.

The Internet of Things consists of a massive number of devices, ranging from sensors, to motors, to communication devices and more. All of these devices report information to, or can be controlled by services. According to *Applying Chatbots to the Internet of Things*[31] there is no reason why these cannot all be controlled in a natural way through chatbots.

The paper defines chatbots as a form of software agent(SA), based on the following key properties

Figure 4 - Simple 'Chatbot' example from [31]

that have been associated with SA's:

- Reactive
- Proactive & Goal Oriented
- Deliberative
- Continual
- Adaptive

- Communicative
- Mobile

The paper quickly makes the connection between chatbots and IoT concerning their use of RESTful Web API's, citing this as an advantage as developers can take an API or service-oriented approach to development for both IoT and chatbots. Chatbots applications can be deployed side by side on cloud platforms with IoT applications, allowing them to easily communicate with each other and the outside world without worrying about the underlying technologies such as storage and processing.

Several use cases are presented, with the user asking a question and the chatbot giving an example answer. "How much is my car charged" could reply with "The Tesla Model S[32] is currently 40% charged. 3 hours 10 minutes to full charge.". The Model S already has a smartphone app that allows you to see the current state of the battery, and it would likely not be difficult to view this API from a different source.

Another use case in the image to the right shows a continuous dialog with intent. The user first asks what the status of the dining room is. The bot reports back temperature and lighting conditions. The user continues the conversation without exiting that particular conversation path, so the bot is aware that the user is continuing and applies the command accordingly. With the current advances in smart home automation such as the NEST[33] thermostat, developers could easily hook into the NEST's API[34] to make this kind of activity possible.

Problem Analysis

Large organisations require fast and efficient network monitoring systems, and many current systems use slow or difficult to set up methods for informing persons of issues such as services offline or environment issues[1]. Security is also a relevant factor, but dealing with potential security breaches or even methods of working out if they are happening are conspicuously missing from many reports and documents[3]. A paper on visualising Cyber Security is very much based around post-analysis, though its favour of colourful visualisations is not shared by many Cyber Security analysts in the field, who would much rather work with “archaic” command line tools[9]. This lack of interest in the visualisation of data makes it more difficult to identify trends in live data, as traversing textual logs can be both time consuming and frustrating[12].

Due to the unification of operating systems over devices, such as the Universal Windows Platform[35] or many Linux variants and Android[8] sharing similar codebases, cross-platform attacks are more popular than ever[6]. This means that monitoring and intrusion detection systems should be take this into account, keeping an eye on as much of the site as possible and not just servers or desktops.

The idea behind Natural Language Processing is to convert text into a programmer friendly data structure that describes the meaning of the text, allowing it to be used programmatically[13].

Chat bots started out as fairly non-intelligent programs, able to only parse very basic and very specific text out of incoming data[21]. Even the more intelligence chat bots such as Eliza and ALICE are still just simple adaptations of the “regular expression” style of chat bots, often written in a style of XML called AIML[16]. While you could still have almost meaningful or semi-useful information gathering conversations with chatbots in the early days of them[16][25], things slowly started to improve with projects taking advantage of large databases of documented human interaction such as the Dialog Diversity Corpus[36]. These databases allowed for much larger keyword recognition and transformations of incoming text to output human-like responses[23]. However, while conversations seemed more natural, this was still only a basic extension of the original “regular expression” style.

As machine learning started to enter the realm of computer science problems, it started to merge with the idea of being able to process natural language, and so began the field of statistical natural language processing[14]. Machine learning allowed chat bots to take interactions like the Dialog Diversity Corpus[36] and learn from them, being able to “guess” at new phrases or sentences rather

than just having a list of stock inputs and outputs. Machine learning is also not bound by its reference material, and is able to learn from the input it receives.

Attempts have been made to start the idea of natural language processing and machine learning almost from scratch, but so far it seems that it is almost always worth referring back to technologies that already exist and building upon them[13]. For example, a study comparing traditional AIML with the ALICE chat bot and a new chat bot program called FUTURE showed that the AIML based bot was significantly easier and faster to develop for due to already having many AI-oriented features, and ended up performing fairly well under human testing[27].

“Machine Intelligence” was defined one paper[27] as a list of fundamental requirements for a machine to be considered “intelligent”, such as having perception of a situation, working sensors and a working memory that can be used to retrieve data logically. Many chat bots can be considered partially intelligent due to being able to apply logic & reasoning, heuristics and a dynamic memory[27].

Recently, technologies like Microsoft’s Language Understanding Intelligent Service (LUIS)[37] allow for non-machine learning experts to build models for things like natural language processing without complex programming or large-scale database management[28]. Machine learning can suffer from what as being called a “lopsided” problem, where not all the data is available for a system to fully realise a piece of data. A human “teacher” can help rectify this and act as the middle-man, translating language easily into machine-readable code until the machine has learned enough to continue for itself[28]. After systems like these achieve the desired accuracy, they can be accessed via external API’s which will accept an input and output a much more machine-useable output, with information separated as requested by the teacher[30].

This table briefly compares IT Infrastructure monitoring tools to see where they get their information, how they notify administrators of potential issues, and if they can be scaled past a small set of servers.

Name of Tool	**OS Support**	**Data Sources**	**Notification Methods**	**Scalability**
Nagios[2]	Windows, any REHL Linux and most other OS’s[38]	Servers send log data via JSON to log server[39]	Email SMS Pager Windows Popup Yahoo/ICQ/MSN	Small-medium size infrastructure, not suited for large or

		Automated host/service checks[40]	Audio alert Web-based Custom third-party packages[41]	extremely large corporations.[42]
Webmin[43]	Linux Only[43]	Installed on every server, gathers information. Can be clustered to manage multiple servers at once.[44]	Email Can get notifications via API[45] Web-based	Small scale only[46]
Microsoft Remote Server Administration Tools[47]	Windows Only	Plug-in tools to control remote servers, uses Windows Management Instrumentation and Powershell to gather data automatically	Tool-based only	Small scale only
Icinga[48]	Windows, any REHL Linux and most other OS's[49]	Servers send log data to log server "Satellite" monitoring servers provide data to the master logging server[49] Automated host/service uptime checks	Provides addons for notifications to almost any service, and allows templating to add new services[50] Provides a query-able API	Large scale support, with distributed log servers and high customisability in terms of notifications[51]
OpenNMS[52]	REHL Linux, Debian, Windows, Vagrant	Runs regular tests against services to check uptime	XMPP SNMP Email Paging Arbitrary HTTP GET/POST's[53]	Designed to be split across multiple nodes – Very scalable if managed properly[54]

This table shows the advantages and disadvantages of various notification methods from the previous list of tools. This is assuming a tool has picked up on the issue and is trying to contact a person able to attend to it.

Notification Method	**Advantages**	**Disadvantages**
E-Mail	Can provide a lot of information at once Can format information well Is almost instantaneous Uses service probably already paid for by the company Can receive commands in response	Unlikely to be seen immediately due to lack of push notifications on most email services
SMS	Fast and cheap Likely to be seen quickly, even if the receiver has no internet Can receive commands in response	Limited information able to be given Would likely have to use outside services
Phone Call (Automated)	Gets attention of receiver immediately if phone is on	Limited information able to be given Receiver has to actively answer their phone to receive information
Web page	Can provide a lot of information at once	Must be checked manually

	Can format information well Is almost instantaneous Can receive commands in response	
Log files	Contains all information	Information could be badly formatted Must be checked manually
Instant Message(XMPP or other)	Can provide a medium amount of information Can format information in some cases, depending on service Is almost instantaneous – Push notifications available through major platforms Likely to be seen quickly Can receive commands in response	Requires receiver to have internet Requires receiver to have IM service application installed

This table shows advantages and disadvantages of various response types – How a receiver of a notification can then affect a problem with their infrastructure while off site.

Response	**Advantages**	**Disadvantages**
Remote desktop	Can see whole server and access tools or logs	Requires VPN or public facing remote desktop server Laggy/unstable connection can cause issues in operating server

SSH	Can see whole server and access tools or logs	Requires VPN or public facing SSH server Can't see graphical information, might be hard to understand data at a glance
Responding to notification (Email, IM, SMS)	Instant action based on the request	Requires receiver of response to be able to understand and act upon request
Web page	Can see and act upon information on the page Instantaneous	Requires VPN or public facing web server

At this point, there are many working IT infrastructure monitoring tools, and this is not where the issue lies. The issue lies in informing the right people of the right information, and giving them the tools to deal with the information quickly.

While many of the existing tools show promise in their ability to send notifications to remote users, via a variety of different methods, many of them then require a large number of steps to attempt to rectify an issue. If an administrator gets a notification via SMS, they must then get onto a PC, VPN into their workplace, SSH into the problem server and can only then begin to figure out the scope of the problem. This is assuming the notification made it to them in the first place, they were able to read it in a timely manner, they are able to get to a work-station, and if notifications were even set up correctly.

Many of the tools rely on technology that is either not suited to the task or out of date. For example, sending an email with an issue report – It could be hours before the receivers' phone checks in to the mail server, and even then requires the receiver to drop whatever they are doing to respond to the issue.

The major issues highlighted in this analysis are firstly the difficulty of connecting infrastructure monitoring services to notification services, and secondly the inability for infrastructure maintainers to reliably manage and fix issues in their infrastructure remotely in a timely manner.

Problem Specification

In our problem analysis, the first major issue that stood out was the inability of connecting infrastructure monitoring services such as firewalls and intrusion detection systems into global notification services. The second major issue was the difficulty of administrators remotely administering their services once they had been notified of an issue. This project seeks to make headway in both those areas by producing a scalable application able to interface with popular security solutions and provide easy to use remote administration.

The goals the project are intended to meet are as follows:

- Review literature related to chat bots, natural language processing, cyber security and visualisation methods.
- Address the inability for administrators to be able to access commands from within a network remotely.
- Address connections between notification systems and data generation systems.

The problem with remote infrastructure administration is that generally, you are away from your tools. Having a dashboard with all your servers' status is useful as long as you can access it. It is very important that administrators know how their infrastructure is operating, as outages may affect a business very quickly. While it is possible that somebody will notice the outage and call the person responsible, it is always better to be informed by trustworthy sources of the scenario occurring. Being away from your tools also limits your ability to work on an issue once notified, and having remote access to tools even as simple as ping can be essential in troubleshooting.

To be more specific with aims, there are several parts to the project that need to work for the project to work as a whole:

- Create a framework for a chat bot and connect it to a popular IM(Instant Messaging) service, and ensuring its style remains modular to allow for easy replacement of services.
- Connect the chat bot to a natural language processing service to allow it to respond certain ways for certain commands, such as LUIS[37]. Give access to commands such as ping, or traceroute.
- Test both the framework and bot to see if it works in an example scenario.
- Critically analyse the framework and bot to see if it lives up to the problem specifications.

Allowing the chat bot to communicate via different IM services is vital. Some companies use Cisco Jabber[55] and some have no IM communication at all. For the sake of having messages that can be

formatted, Telegram[56] is a good choice for testing. However, the bot should be able to be modified to work with most IM services with minimal effort. Natural Language Processing services, like Microsoft's LUIS[37], Google's Natural Language API[57] and Facebook's Wit.ai[58] are all viable to be used for chat bots. Having an easy to train platform that is able to parse requests without complex regular expressions will make it much easier to expand the bot in the future.

While it is not a requirement to have the bot connected to a production server, we believe it is important for the bot to encounter live data so it can be taught how to deal with it. Being able to remotely access logs via SSH or FTP and parse them for data will likely be a major part of the security monitoring aspect of the bot.

One of the most important parts of the whole project is the alerts and notifications system. Once an alarm is tripped, a specified user should be informed via their IM service. While we do not expect this aspect to be as powerful as the tools of a well-equipped network, this should at least give them access to part of their arsenal in which to troubleshoot and assess the situation. This could be anywhere from ping, to dig, to a fully-fledged remote shell.

The bot should be scalable to support as many different users, IM services, notification types, API reads and remote servers as possible. The bot should never be the bottleneck, only the hardware on which it runs.

To complete these objectives, we intend to use an array of utilities ranging from online services to open source tools. This will likely include utilising tools on both Windows and Linux for a more realistic approximation of real administration. It is hoped that this tool will be used in day to day administration of a real network, but it should not be specific to that network.

Design

Based upon our previous section detailing the specific problems we would like to address; this section will contain the design and implementation phases of the project based upon our previously identified objectives. The design phase will focus on detailing the individual modules and how they can interconnect to form a working solution. The implementation phase will look at the final product, as well as any issues encountered along the way that may have differed from the plan.

The general design ethic for the project is for it to be modular. This allows for the replacement of entire services with others as long as the data output remains consistent with other modules. For example, replacing Microsoft's LUIS[59] natural language processing service with Facebook's Wit.ai[58] would not require an entire system rewrite, but would only require the replacement of the Natural Language Processing module.

As an addition to the modular design ethic, the project is intended to be a framework that can be refined, worked on and added to over time. It is not intended to be a complete monitoring system for a network – rather it is a framework with a various proof of concept and example uses pre-programmed.

It is intended that the application be semi-autonomous in nature and able to be extremely low maintenance once configured. The application will have two operational modes:

Interactive mode is a state where the user leads the conversation. A user will query the app, which will cause the app to produce a result and serve it back to the user.

Monitor mode is the applications default state, where it is analysing incoming information from sources such as servers and comparing it to known thresholds. If a threshold is met, a user responsible for that threshold is informed – The application is then in interactive mode once a user replies.

Data Sources

For our testing we will only be taking data from a production Linux Ubuntu operating system. This will include the following data sources:

Apache[60] **HTTP Server Access Logs**[61] show records of pages served and files loaded by the webserver. This can be valuable information if formatted correctly. This information will be used to generate statistics.

Apache HTTP Server Error Logs[61] show records of all error conditions reported by the server, which in some cases will need urgent attention.

Authorization Logs[62] track usage of authorization systems such as *sudo* and remote logins over SSH. These can be used for both statistical purposes and for showing login attempts.

Login Failures Log[62] is designed to be non-human-readable, and contains all login failures. This may be better for machine parsing than the authorization logs.

Last Logins Log[62] is also non-human-readable, and shows the last login of users.

However, there is no reason why extra modules could not be added with more commands and to accept more sources.

Other Commands

Remote administration means being away from tools, so adding tools that are often in an administrators' arsenal to the application can be very useful in troubleshooting. Having remote access to a machine inside the network means that certain commands can be piped to the user:

Ping can be used to ping both internal and external resources, telling the user if that resource is online and what the delay is.

Traceroute shows the user the route packets are taking to a server, through other networks and servers.

Nslookup allows a user to look the DNS information of a resource either internal or external as reported by their DNS server.

SSH allows an interactive shell to another computer. This may or may not be possible with our planned setup and will have to be tested during the implementation.

Security and Other Considerations

Security is a major point in this project, as this application may have access to logs or passwords for servers. To address this issue, we will be using the security offered by the various chat services by associating access to the bot with access to instant messaging services.

In our case, BotBuilder[63] provides a user ID along with every message. We can compare this user ID to a hardcoded whitelist that allows a user access to functions in the bot if they are whitelisted. A user should take steps such as two-factor authentication on their IM account in order to protect it, and by extension, any data the project may be able to give out.

It would be possible to add more security considerations to the modules such as passwords, but this was deemed unnecessary for this example framework.

Modules

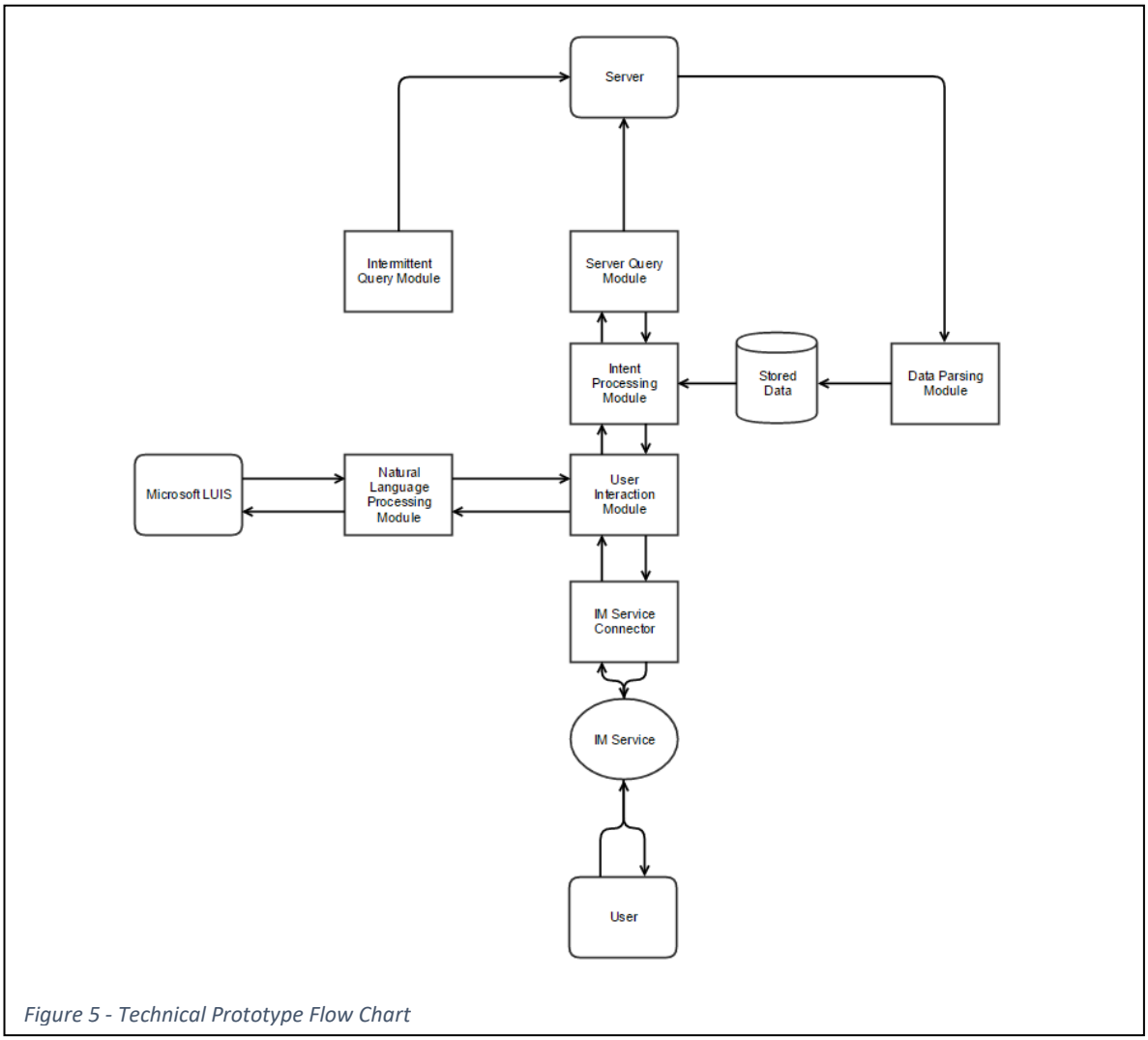

Figure 5 - Technical Prototype Flow Chart

Our initial plan is shown in Figure 5 - Technical Prototype Flow Chart, and shows each individual module. The following section will go into detail of each module and discuss its uses, as well as how they will interact or be swapped out as necessary.

IM Service Connector

This module connects the User Interaction module to the IM (Instant messaging) service. This simply manages the connection to the instant messaging server, be it IRC[64], Jabber[55], Telegram[56] or any other. As we already know the Microsoft Bot Builder[63] acts as a proxy between many IM services and our application, some of the work here is done for us, described below. However, in the case that a client wishes to use a different service, our User Interaction Module must be fed data in the same format.

Bot Builder does the work of connecting to the IM service and then forwarding user data from the IM to our API endpoint. Following Bot Builder conventions, we will use their library to open a

Restify[65] web service on our server, which will receive the data and handle replies to the Bot Builder server. We can expect the data to be very similar irrelevant of what IM service the user is using. This data is handed directly off to the NLP service without modification, and contains the session information such as the user's ID, what conversation it is part of, other data required for a reply, and the users message.

Bot Builder was chosen due to it's also modular nature, it allows for different NLP back-ends to be replaced with relative ease as well as not using one entirely. It also allows for many different IM services to use the same bot back-end, so as not to limit the user in their choices and it can be easily integrated into existing company instant messaging systems.

Natural Language Processing Module

This module takes the formatted IM messages and sends them to an NLP service, and formats the information to be understood by the User Interaction Module. For the sake of our testing, this module will be tuned to Microsoft LUIS[59] running on an Azure instance.

Intents are what the user 'intends' to do. These intents, as well as what triggers them are defined in the LUIS NLP website[37], and will be planned later.

At program startup, the NLP module queries the User Interaction(UI) Module to get a list of known intents. The NLP module then generates templates to allow it to format the information in a way the UIM understands. No matter what service the NLP module connects to, it will format the

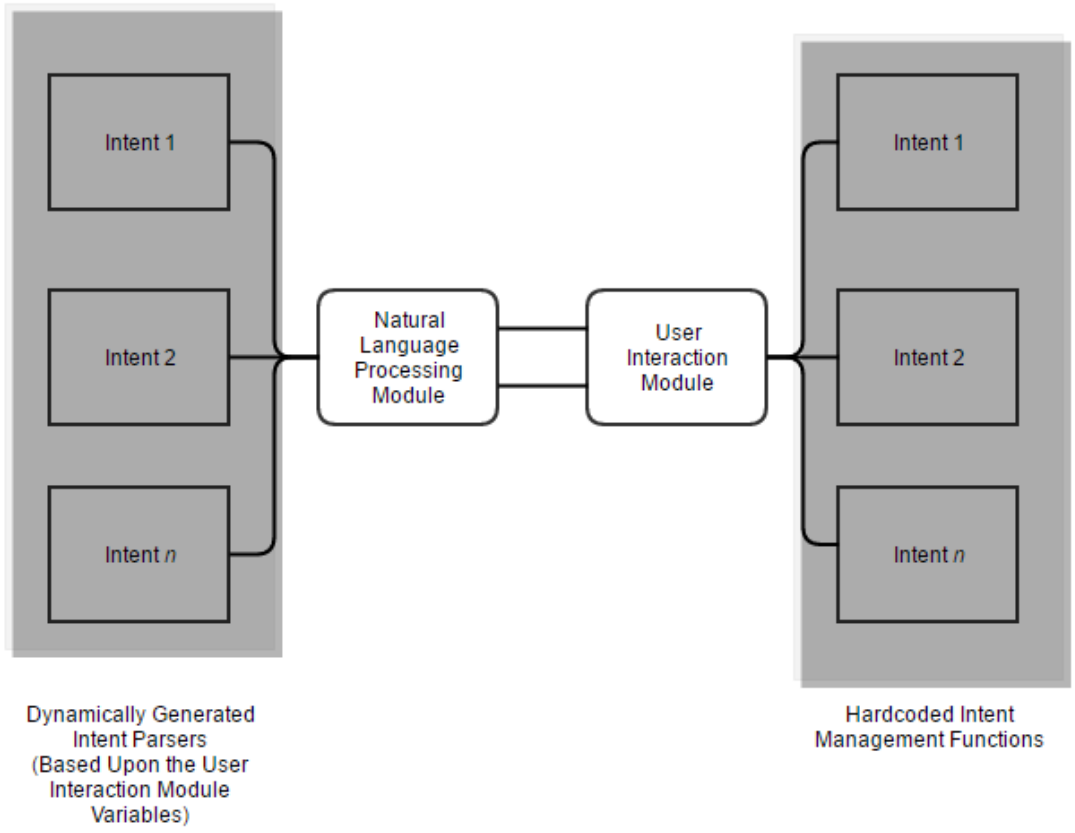

Figure 6 - Showing the relationship between the NLP Module and UI Module

information the same way. This generation of functions means that all data coming to the UI module will be in a constant format irrelevant of the intent, and as long as the NLP module sends that data in the same way it is irrelevant which NLP service is in use. We preferred this as it meant significantly less hard-coding of intent functions within the NLP module, making it easier to keep consistency in the event that the NLP module is replaced with one for a different service.

Shown in Figure 6, the intent management functions in the UI Module are hard-coded. The NLP Module then dynamically generates the functions required to handle these incoming intents. The functions here also handle the reply to the user, so it is important to make good use of callbacks between the two modules so the session data is not lost.

Upon receiving a message, the NLP Module will receive the data and immediately send it to our NLP Service (LUIS) to be analysed, which will then be returned and associated with a function related to the intent that the NLP Service deemed the user was trying to activate. The function will then strip

the arguments from the formatted message provided by the NLP Service into a more generic format, shown in Figure 7.

```
{
"application1":"value1",
"argument1":"value2",
"server1":"value3",
"argumentN":"value4"
}
```

Figure 7 - Generic argument format

This generic format will contain the values of all arguments in the users' message. For example, if the user were to type "Show me the log files for Apache on the Ubuntu server". The intent would be 'log', and the two arguments would be 'Apache' and 'Ubuntu'. The NLP Module sends this data, along with the session data in case it is needed later, plus a callback to the UI Module to be handled. The callback, when triggered by the UI module, will accept three arguments. The first argument will be either a string or array – If a string is sent, it is sent directly to the user. If an array is sent, it will be looped and each part of the array will be sent as a separate message to the user. The second argument specifies to the function that the dialog should be handed off to another function that handles multi-message dialogs.

The function that handles multi-message dialogs will be hardcoded, but still will call the intent handler in the UI module for instructions with a specified keyword 'Convo', which states that the conversation has entered a multi-message dialog state. This function will loop, receiving data based on how many times it has looped from the UI module with what to send to the user. The UI module may call back with a variable dictating the type of message to send the user, either a string, a prompt or an exit. A string will simply send a message to the user and loop again. An exit will exit the dialog.

However, a prompt is a special situation. It first prompts the user with a question, and then hands off to another handler function using Bot Builder's Waterfall method, which allows for conversations to be handled differently depending on the flow of a conversation. This final function then adds the users' response to an array, and loops back to the original dialog function in order to call and send that data to the UI Module for continued processing.

User Interaction(UI) Module

This module transports incoming information from the Natural Language Processing module and forwards formatted data of the users' intent to the Intent Processing Module. This module also

formats data intended for the user. In some cases, Intent Processing isn't necessary and a simple reply can be managed by the User Interaction Module. This module is entirely based on the 'Interactive Mode' idea, where it is only in use when the user is directly using the application.

This module can be left static between changes of the NLP and IM modules, as it relies on those modules handling formatting and simply has a list of known intents and how it reacts to them.

The UI Module acts as the middleware between the raw data of the NLP Module and the Intent Processing module. It contains a hard-coded list of intents which should be identical to those specified and trained on the NLP service website, which will act as the basis for the NLP modules functions. As by this point the intent has already been figured out, and the arguments have been correctly parsed, the UI module is where the real human-written responses start to take place.

In the UI module will be an intent handler, containing a function for each known intent plus conversation function for any intents requiring question and answer type conversations. Each function will accept 2-3 arguments, the first being the callback to the NLP Module that sends messages to the user, the second being any arguments that came from the user and formatted by the NLP Module, and finally the session variable which contains all the user data. These functions will then either then call the callback directly to reply to the user with a predefined variable with

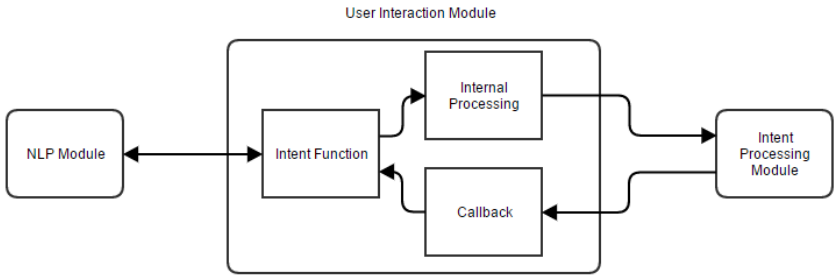

Figure 8 - User interaction module data flow

minor processing such as picking out the user ID and sending it back, or will pass on data to the Intent Processing module with another callback to return back when finished. The flow of data is shown in Figure 8.

As an example of a simple intent, such as the 'help' intent, the UI module would ignore any arguments and simply call the callback with a simple, hardcoded reply. In the case of an intent requiring responses such as the 'SSH' intent, the function would immediately call back to the NLP module requesting a change to the conversation handling function. The conversation handler then loops, incrementing a counter that calls the SSHConvo intent, getting a different prompt or text

response each time until all the required information is obtained. This seems like a reliable method for dynamically generating conversations without a complex hardcoding system.

To prove this framework is viable, we will implement several simple responses and 1-2 complex responses. The list of planned user interactions, and therefore intents, are:

- None: Will respond that no intent has been detected.
- Version: Will respond with the systems version.
- Help: Wil respond with a hard-coded ‘help’ text.
- Identify: Will respond with identifying features of the user and the current conversation, such as user ID and platform in use.
- Ping: Will tell the user if a given IP address or endpoint is online and replying to pings.
- Traceroute: Will perform a traceroute to a given endpoint, and reply with individual messages to the user with each hop.
- SSH: Will open up a live shell session with a given server, asking for username, password, and port and then forwarding messages to and from the server.
- Logs: Will connect to a hard-coded server and allow the user to choose (with natural language) between several different logs, before tailing the chosen log to the user.
- Associate: Will associate this conversation with a hardcoded query, which operates on a hardcoded timer. If an ‘alert’ is triggered by this query, the user will be informed and the alert data will be sent to them.

It is believed that these intents are sufficient examples of what the framework can do and how it can be worked with.

Intent Processing(IP) Module

The IP module manages all complex processing of information, and is called upon by several modules. At its core, the IP module receives data to the required function, processes information and then returns that information via callback to whichever module requested it. Not all intents require extended processing, but it makes sense to separate the majority of the processing that

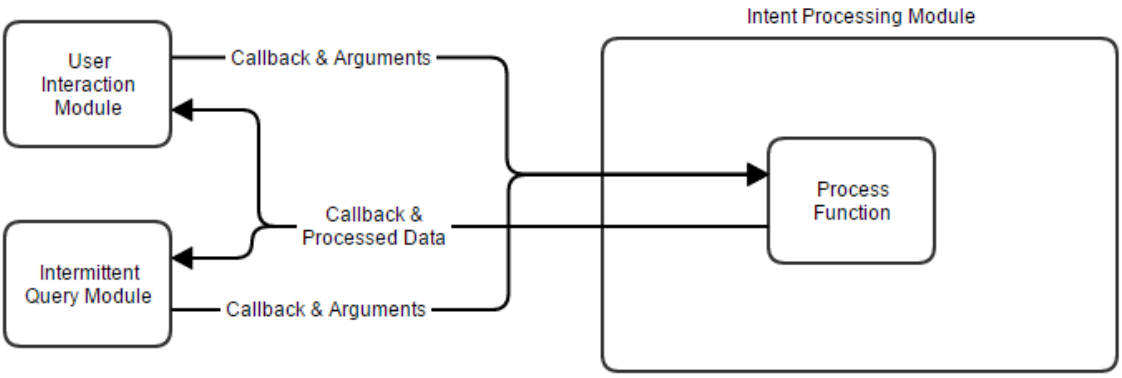

Figure 9 - Intent Processing Data Flow Example

happens away from direct user responses.

Figure 9 shows an example of the flow of data from two modules to the Intent Processing module. In this case, either module can request a function from the IP module as long as it has the correct arguments and provides a callback, the IP module will then reply with the processed data.

Referring back to our list in the UI module planning, only the Ping, Traceroute, SSH and Associate intents will require processing in the IP module. Additionally, there will be a 'helper' file downloading function, which will download data from at least SSH servers for log retrieval. Each function will require specific data from the UI module, and will return specific processed data.

Ping can expect simply an address and a callback. It will send a request to the given address and call back whether the address is alive or not. This is our simple test to show that the framework is able to take a command, process it, and return information to the user. Traceroute, similarly, can expect only an address and a callback, returning the call for every single hop. This shows that the application has the ability to return multiple times to send the user better formatted data. Associate will expect three arguments: Some data referring to which query the user intends to associate the conversation with, the session variable which allows the function to save it against the query, and the callback in order to inform the user of the success or failure. Finally, the SSH function will be able to take the previously compiled connection data to connect to a server, and receive new data once a connection is established to forward messages from the user to the server, as well as reply with

server messages through the callback. SSH will be a complex example of multi-stage conversation and will be a challenge to see if it is even possible with the planned configuration.

Data Parsing Module

This module parses incoming data from servers into data more easily readable by other modules, and attempts to keep data consistent in the case of module replacement. The Parsing module will have functions added as required, and will primarily use Regular Expressions and replacement functions to make data more consistent. For example, the Server Query module may receive log data from a server containing line by line date and login time data, it could then send this data to the Data Parsing module to format this data into an object with a more Javascript friendly date system plus having the data predictably located in a variable as part of the object tree. However, this function would have to be hardcoded in as part of the Data Parsing module.

This module will be used when we deem necessary while developing, rather than being rigorously planned at this stage. An example data flow is shown in Figure 10, where various modules send

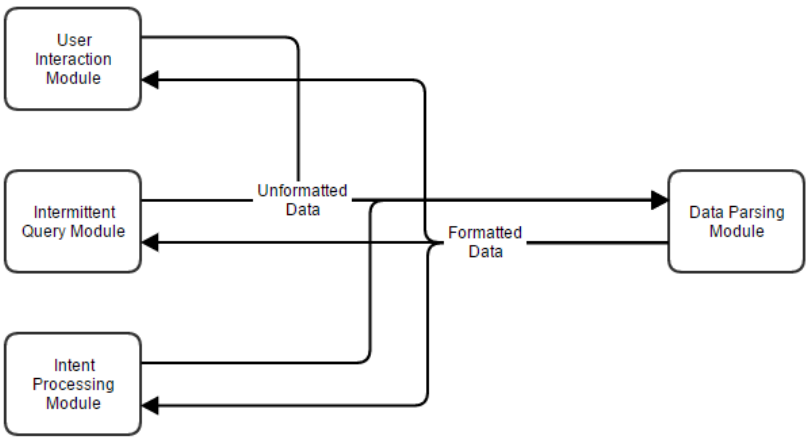

Figure 10 - Data Parsing Module example data flow

unformatted data to the Data Parsing module, and the module returns back formatted data. This makes it significantly easier to maintain cleaner code when it comes to managing data and text by keeping all major text manipulation in one place.

Server Query Module

The Server Query module manages queries to and from servers. It will have a function for each major file transfer protocol, such as SCP and FTP, which will both accept connection information and file locations to be downloaded. These functions will regularly be called by the Intermittent Query Module in order to make a request to a server, and this data will parse through the Data Parsing module to enter a consistent format.

In our example intents, the only functions that actually do regular queries to server are the 'log' intent and the Intermittent Query Module, which contains a hardcoded list of queries the application does regularly. The Server Query module will be quite simple in practice, just being functions able to download files using data provided by other modules.

An example function for SCP would import an SCP downloading Node module, and expect an object containing the server address, file location and a local location where to save that file, and would call back to the function that called it with the downloaded file location. This function can then retrieve the data and send it to the Data Parsing module if required.

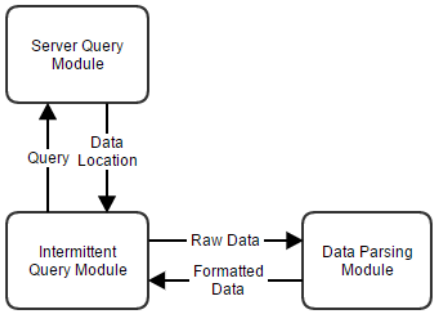

Figure 11 - Intermittent Query, Server Query & Data Parsing Interaction Example

As shown in Figure 11, the Intermittent Query module will send a query to the Server Query module containing required data, such as credentials and address information, and the Server Query module will call back with the location of the data. The Intermittent Query module can then read that data and send it to the Data Parsing module if it needs to.

Separating Server Queries from the other module makes sense at this stage as it can be highly customised. It is likely that server queries will be written from scratch depending on differing systems and are unlikely to be shared between installations, unless a large number of generic queries can be created. It is important therefore that the Server Query module be very generic in its

```
{
"name":"serverQuery1",
"data":"values",
"other":"OtherInfo"
}
```

Figure 12 - Server Query Example Output

structured responses, so it can be easily replicated when creating more queries. Figure 12 contains a snippet of the proposed structure of data output from the module. Very generic, containing just the name of the query (for confirmation), the data requested and any other information just in case it is relevant.

Intermittent Query Module

This module queries data from servers based on a timer, and is related primarily to the “monitor” mode of the program. Server queries will be stored as objects inside of an array, and hard coded into the program.

An example would be an intermittent query that checks, every minute, the ‘auth’ log from a Linux server. In this case, we would need to SCP to the server every minute and download the auth log,

```
{
        name: 'LinuxAuthLogs',
        server: 'linux', //Defined in the servers file and looked up later
        type: 'logs',
        application: 'system',
        specific: 'auth’,
        delay: 60000, //How often between queries
        run: function(callback) {
                //Call the server query module to get the file
                serverQuery.scp(this.server, this.type, this.file,
function(fileLocation) {
                        //Call back with file location and read the file
                        readfile(fileLocation, function(file) {
                                //Call back with data and parse it
                                var parsedFile = dataParse(file);
                                //Inform the user using their saved session
of the alert and its details
                                informUser(parsedFile, userSavedSession);
                        })
                });
        }
}
```

Figure 13 - Example layout of an intermittent query

then parse it for information we cared about such as failed or successful logins, being careful to not include information of our own connection to get data, and then inform the user if it meets certain criteria. This also means the user must associate their conversation with the query using the ‘associate’ intent before receiving alerts.

Figure 13 shows an example layout of an intermittent query in pseudocode, detailing all information related to the server aside from credentials, which is saved in the servers file shown later. It contains a function that can be called, which handles the gathering of data using other modules and then finally responding to the user. The name is used to associate users, where a user can simply type ‘associate LinuxAuthLogs’ to receive alerts for that query. The function itself will be called by a separate part of the module, which loops through queries on start-up and configures the intervals at which the queries will be run automatically based upon the definitions above.

Stored Data

Data that may be required to be stored between sessions will be stored on disk in the form of JSON. Data that may be relevant to store include conversational sessions so users do not have to re-associate themselves with a query, and server data such as log history. If data needs to be stored, we can simply call JSON.stringify() on it and save it to disk, then load it to a variable on startup. This will be done as needed and is not a directly planned occurrence.

Server File

The server file is simply a file containing objects of known servers, their logins and known paths to logs. These objects can be navigated by the Server Query module to obtain information required to retrieve logs and other data.

```
{
        "linuxServer": {
                address: 'linux.local',
                protocol: 'scp',
                port: 22,
                username: 'root',
                password: 'toor',
                logs: {
                        system: {
                                auth: {
                                        path: "/var/log/",
                                        file: "auth.log"
                                }
                        }
                }
        }
}
```

Figure 14 - Server object example

Figure 14 shows an example of a simple server layout. A correctly configured Intermittent Query would be able to access the full path of the log file, as well as the address, protocol, port, and login by simply following the object. For example, *servers[this.server][this.type][this.application][this.specific].path* when referencing from the Intermittent Query module would give the module access to the path for the log file on that server.

Setup File

The setup file will not be included in the final project paper as it contains private information such as passwords to servers, private keys for the LUIS application and lists of whitelisted IP addresses. This seemed necessary to address as the mysterious ‘setup’ variable is reference often in the code appendices but never makes an appearance itself. It is simply an object containing information not directly relevant to the code.

LUIS

The choice of using the LUIS NLP[37] service between others was a tough one at first, but it was decided that it in fact did not really matter which NLP service was chosen. As the project is deemed to be a framework rather than a fully defined software application, LUIS can theoretically be replaced with any one of many NLP services as long as the NLP module is tuned to output the same information to other modules.

However, LUIS ended up being the chosen service for testing due to the project lead having prior experience with it, and it seemed fairly fast to train and export learning data to a remote Azure server. The choice of Azure was easy, as it naturally connects to LUIS and Bot Builder, allowing for a much-simplified approach to hosting the project.

The plan for training LUIS is to come up with a base phrase, such as “ping {address}” and then working off of that manually to add extra training phrases, like “Please ping the address {address}” and “Ping the IP address {address}”. Training will be continuous as the project is being developed until testing starts.

Implementation

This section follows on from our design section to show the actual development of the application and any issues we had while developing it, as well as outlining any differences from the design.

Forwarding Ports

The first step in producing the application was met with the issue of forwarding ports. Because Bot Framework[63] expects you to have a HTTP endpoint for it to send information to, and one had not yet been set up on a production server, we have to open ports locally. This is a problem, as our test environment does not allow for port forwarding.

To fix this we used an application called Ngrok[66]. Ngrok allows us to expose local servers to the internet from behind a NAT when we are not able to port forward. Using this tunnel allowed us to

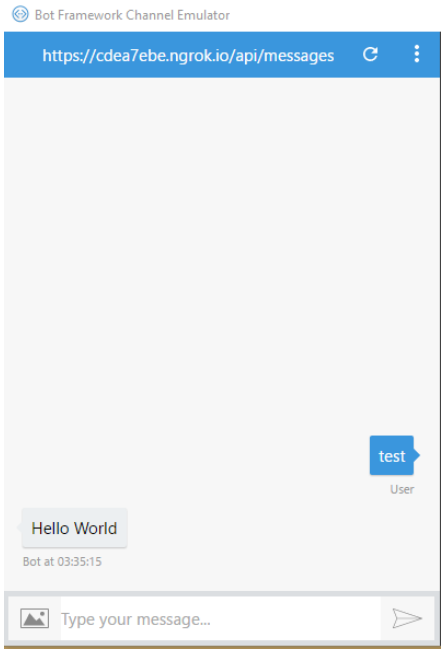

Figure 15 - Successful remote call and response

continue our test deployment. At this point, we have an internet-connected chat bot that we can test with the Bot Framework Emulator[67] Shown in Figure 15 is a basic 'Hello World' response.

IM Connection

Next we need to connect our application to an IM service. In this case Telegram[56], as it has an abundance of formatting styles and rich messaging capability that can be used to make data look better for the user, if required. This is as simple as registering a bot with the Telegram service by messaging the 'BotFather' user and following its instructions. Entering the given HTTP API tokens

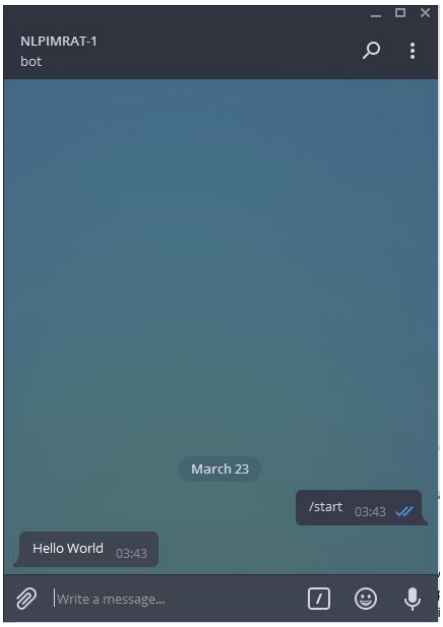

Figure 16 - Call and response from within Telegram

into Microsoft's API allows the bot to connect directly to the Telegram service, so we can access our bots commands from within Telegram, shown in Figure 16.

Implementing Natural Language Processing

NLP must be added early, even prior to training being started. It is important that the module is at least configured so it can be expanded upon easily. All programming for Luis.ai[37] is done via their website, but when finished is hosted on an Azure[68] endpoint. After setting up a basic NLP app and publishing it to Azure, we have access to the 'intents' that we configured. Intents are defined as "What the user intended to do based on what was said", and data is formatted in a way that is understandable in code rather than in natural language. It also separates the users intent from any potential arguments in their message, as well as giving them names, making it much easier to pick out values and arguments from messages. These intents allow us test our connections between Azure, BotBuilder, and Telegram.

In this case, some very simple training had to be added for Luis to understand the 'ping' command.

Training is a fairly simple process, but can become very intensive. To add a command first you must break the command down into its most bare components. In this case, the smallest possible rendition of the command is “ping {$address}”. But in natural language, it could be “Could you please ping the address {$address} for me?”. We must think up as many different forms of this as possible to train LUIS into understanding what it receives.

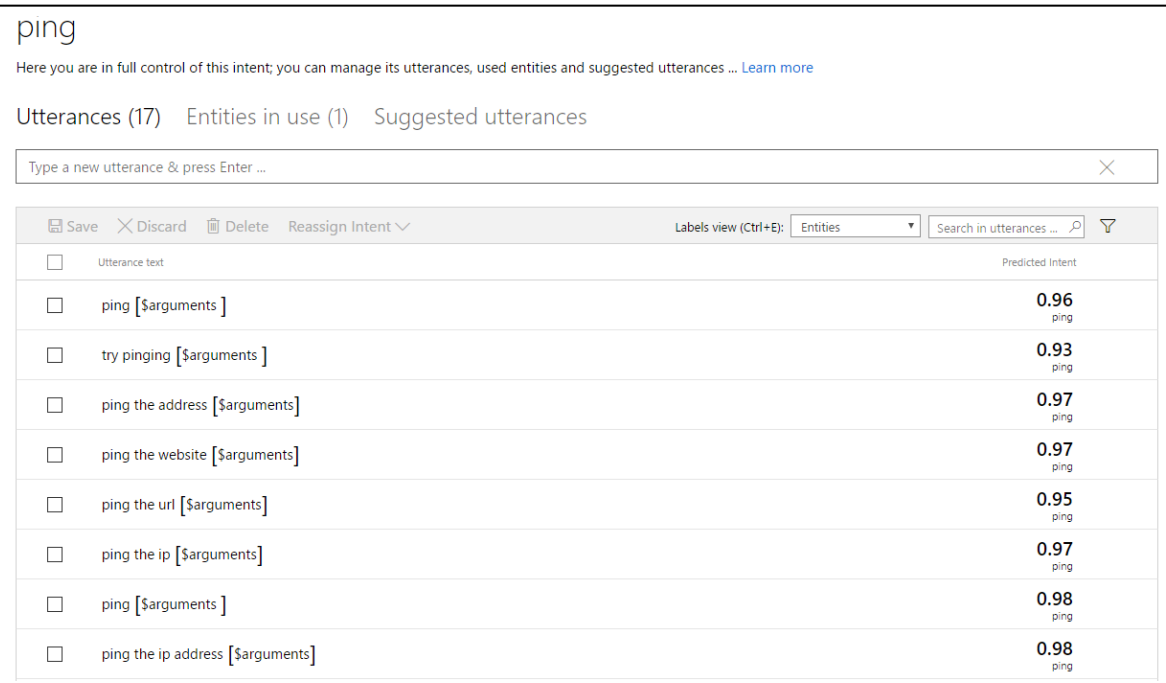

Figure 17 - Training the 'ping' intent

We can quickly come up with some basic ‘utterances’, which are defined as the different ways of saying the command in natural language[69]. Luis then attempts to parse it based on what it knows so far. In Figure 17 we can see on the right-hand side that Luis successfully guessed that our utterances were related to the ping intent with a 93-98% accuracy. With regards to the *[$arguments}*, this variable is manually set to begin with, but after further training Luis is able to guess where the arguments will appear in a given utterance.

In Luis, we named our intent “ping”, and we can easily tell if a user’s intent was to ping by the response we receive from LUIS. For example, if it DID match ping, we can tell the user what arguments we managed to pull from their command. Failing that, we can give a default error message. At the same time, we can also program in the intents for querying the version number, and some templates for a help system. At this point the first rendition of the dynamic NLP module started taking shape, an early and simplified example of which is shown in Figure 18, where the known intents are being looped and a simple argument parser is put in place.

```
//Grabs known intents from user interaction module, creates a template
handler for each by looping through them
for (var i = 0; i < knownIntents.length; i++) {
    var match = knownIntents[i];
    var intentGenerator = `
    dialog.matches('${match}', [
        function (session, args) {
            console.log("Handling intent: ${match}")
            var arguments = [];
            //Simplifies arguments for the intent handler
            for (var i = 0; i < args.entities.length; i++) {
                arguments.push({
                    name: args.entities[i].type,
                    value: args.entities[i].entity
                });
            }
            //Sends intent plus data to the UIM for processing, then returns
back with the
            //text for the user
            intentHandler['${match}'](arguments, function(response) {
                session.send(response);
            });        }
    ]); `;
    eval(intentGenerator);
}
```

Figure 18 - NLP Module template generator

To confirm that the NLP Module template generator was working correctly, we created a very simple example of the 'ping' command. We know that 'ping' is an intent that might be returned by the NLP module, so just for testing we can reply with a response that shows we know the user requested the

```
//List of known intents
//This allows us to have a static list of expected data even in the case
that the NLP module changes
var knownIntents = ['ping'];

//Handling of each intent, NLP module agnostic
var intentHandler = {
        ping: function(arg) {
                console.log(arg);
                var response = "You wish me to ping " +
arg[0].value.replace(/\s/g, "");
                return response;
        }
};
```

Figure 19 - User Interaction Module(UIM) example code

ping intent, and what address they were looking to ping. This snippet of code is shown in Figure 19.

It turns out that Luis separates punctuation from arguments when extracting them from text, so the address "google.com" becomes "google . com". Thankfully this can be easily fixed by deleting whitespace before sending the data off for processing.

Implementing Commands

Now that we have confirmed that Luis can be trained and that the bot can receive not only intent data, but also argument data, we can start to actually implement our intents. Our intents are separated into ‘basic’ and ‘advanced’ commands, basic being defined as not requiring intervention by the Intent Processing(IP) module and advanced being mostly handled by the IP module. At this point we are only really interested in the User Interaction and NLP modules, so basic commands will come first.

Our plan states that we have 9 intents, the basic ones being ‘None’, ‘Version’, ‘Help’, and ‘Identify’. These all came together incredibly quickly, as they are all planned to be functions that respond with either text provided by the user or hardcoded text inside the User Interaction module. First, the NLP Module function generator was modified to send the ‘session’ argument along with arguments and callbacks to the User Interaction module. Then a simple function was made for each, returning their text to the NLP Module for sending to the user. The ‘none’ function simply replied that no intent was

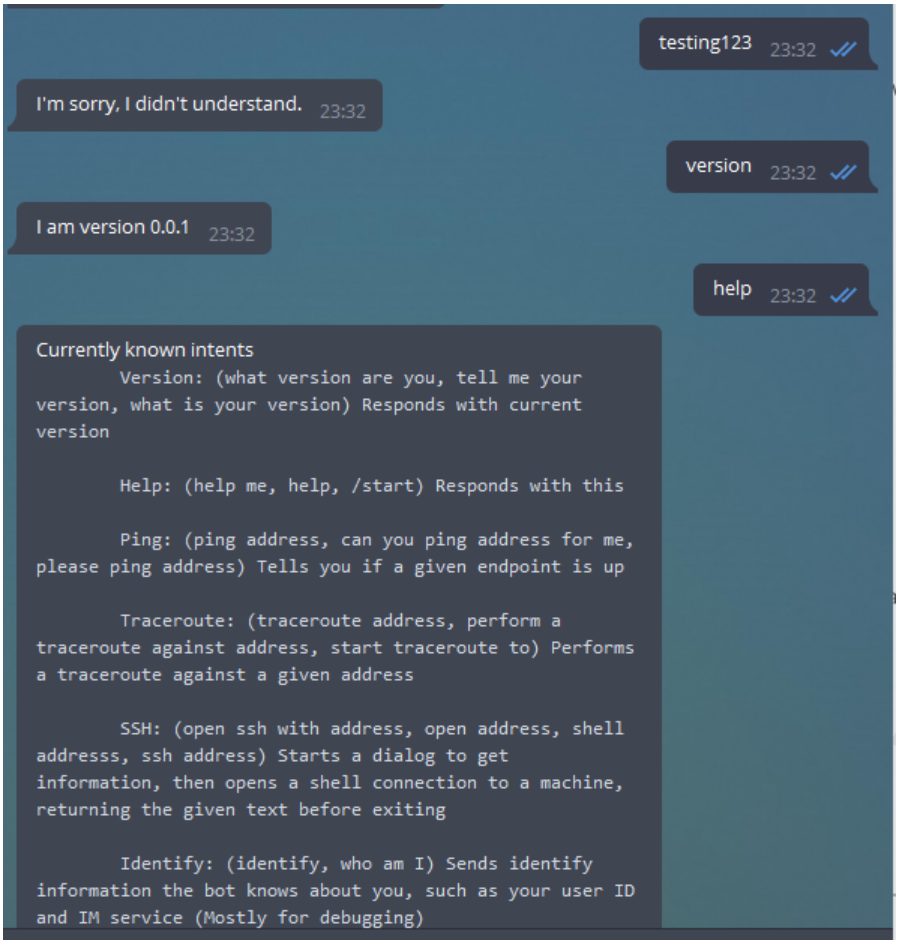

Figure 20 - Initial basic intent processing

detected, ‘version’ took the version number from the package file, ‘help’ replied with a predefined

help text, and ‘identify’ took the Channel ID, User ID, Address ID and Username from the session variable and returned them to the user for debugging. Figure 20 shows examples of the help, none and version intents.

The more complex commands required some extra work, and the implementation of the Intent Processing module.

Ping

Instead of using the systems in-built ping command by piping user input straight to a console, which would make it ripe for command injection, we instead opted to use a powerful but small library simply named ‘ping’[70]. It sanitises inputs and simply checks if a given IP is online or not. Using Node’s in-built DNS tools combined with *ping* allows us to quickly check if a given address is ping-able, as well as using our previously trained NLP service, we can set up a small alive-or-not host checker.

The reason behind doing a DNS lookup prior to pinging is because the ping library does not support the pinging of Windows hosts by computer name. The simple fix was to use DNS lookup to first obtain the IP address of the machine, and then have the ping library check if it is online.

The Intent Processing module was created similarly to the User Interaction module, with an object containing all of the functions that would be called. The ping function accepted the address, did a DNS lookup upon it (in the case that the user provided an IP address, the DNS lookup will just reply with the IP address, so this isn’t an issue), pinged the address, and had a simple response telling the user if the host was alive or not and what IP address the address resolved to. It then calls back down the chain, first to the User Interaction Module.

The User Interaction module function for the Ping intent is simple, it takes the argument from the NLP Module, replaces any spaces, and sends it to the Intent Processing module. Once the IP module calls back, the UI module also calls back to the NLP module, which finally sends the message that was passed down the chain to the user. Figure 21 shows an example of several successful and failed

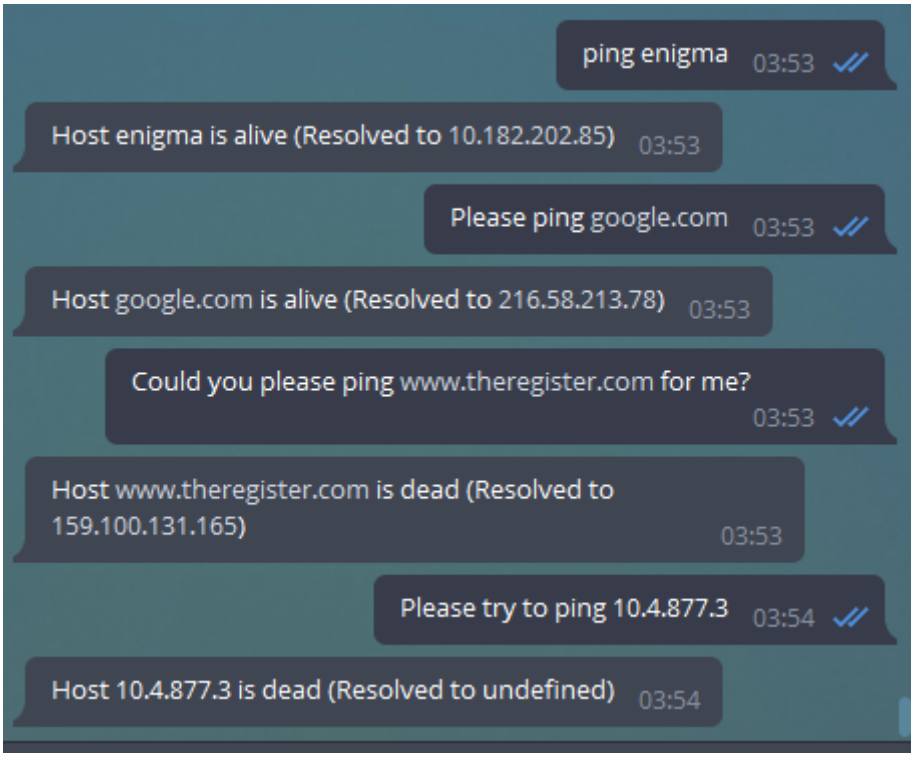

Figure 21 - Simple host PING example

pings, and it is interesting to note that the final phrase, "Please try to ping *x*" was not directly trained into Luis, and it in fact correctly guessed the intent and argument based on its training thus far.

In terms of the actual training, Figure 22 shows the 14 base examples used to train Luis, where '*x*' refers to a random IP address or website address. After even those few examples, Luis was guessing intent with a 95% accuracy, which was considered close enough for testing. At any point, more training can be added to improve the accuracy further.

ping x
ping the ip address x
ping the address x
can you ping the domain x
can you ping the address x
please try pinging x
please ping x
ping the url x
ping the ip x
try pinging x
can you ping x for me
can you ping the website x for me
is x online
is x up

Figure 22 - Base training examples for the 'ping' intent

Traceroute

The traceroute intent was achieved with a similar library, simply named 'traceroute'[71]. This library takes an address as an argument, and returns an array of 'hops' containing each IP and three millisecond timings of the ping to that hop.

As with the ping intent, the User Interaction list of known intents was updated with the new intent, and a new function made to handle it, which simply replaces whitespace in the argument and hands it over to the Intent Processing module. The IP module function for the traceroute intent takes the address and calls the traceroute library, which then returns the hops array. We then cycle through this array to clean up the data and also average the milliseconds for the pings to each hop, it then calls back all the way to the NLP Module for each hop, sending a message each time it loops. This function is an example showing that the framework is capable of managing multiple callbacks to the user.

For training LUIS, we used much less training compared to the ping intent, primarily due to the difficulty of finding alternative ways to say 'traceroute *x*'. We ended up with only 6 examples to train Luis with, but this seemed to be enough for Luis to claim a 97-100% accuracy.

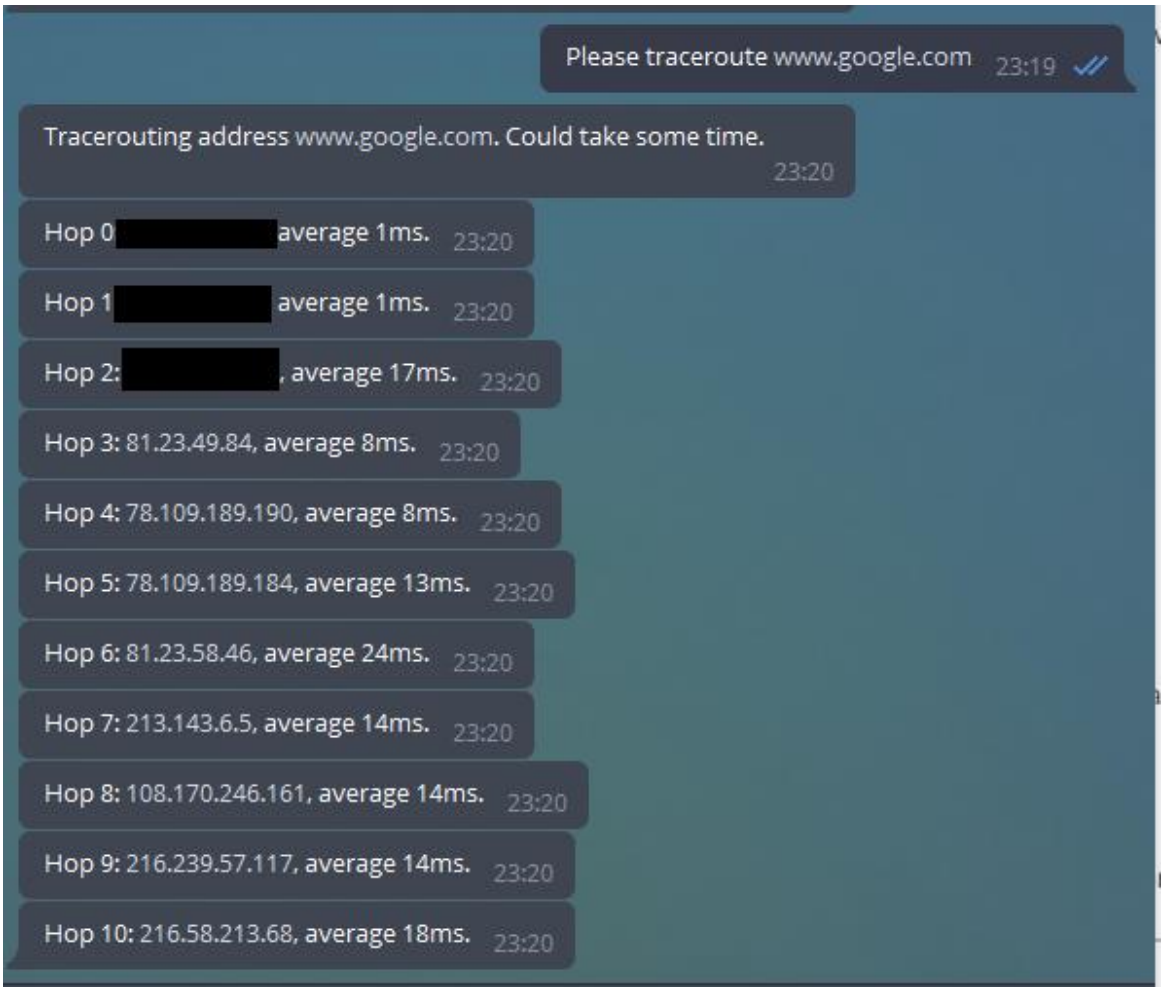

Figure 23 - Successful example of tracerouting

Shown in Figure 23 is the response to a traceroute command, with the calculated average time in milliseconds afterwards the IP address of the hop.

SSH

The implementation of a live shell session was much more complex than any intents so far, and was the first and only intent to require a conversational dialog with multiple questions and answers. Unfortunately, due to issues with how Bot Builder managed callbacks and data, the implementation did not fully follow the planned design and several key features had to be left out, which will be described later. While it may certainly be possible to do exactly as planned with Bot Builder, it would have required major changes to how the dynamic generation of functions in the NLP Module

worked, which would likely have changed the modular aspect of the framework. We chose to instead reduce functionality of the SSH intent rather than stray from the overall project goals.

To start with the implementation, we first had to modify our existing generator in the NLP module to be able to handle a variable telling it to swap functions, as per our design. We then had to make another function which handled question and answer style dialogs. It was fairly easy to follow the design up until it came to maintaining data while looping, and as global variables were not a clean option and Bot Builder had no easy way to maintain data between dialog changes, it was decided that we would try to include relevant data and saved data when calling the dialog again.

This idea ended up making the code difficult to work with as it made it quite untidy, but with few options it continued to be developed on in the User Interaction module, where a function was made containing an *if* statement that would reply differently dependant on what iteration we were on. Lastly, the Intent Processing module gained a function that connected to an SSH server, streamed incoming data to the user, then disconnected.

It was at this point we came to realisation that there was no safe and clean way to pass further variables to the server and receive the response, and so it was decided to not complete this

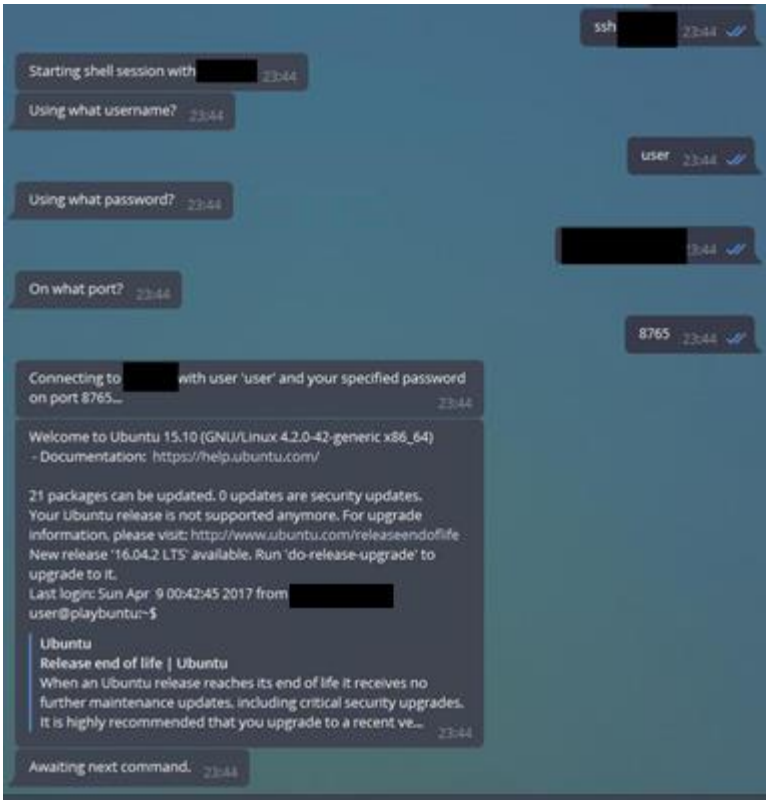

Figure 24 - Dialog-based SSH connection

function. However, it still operates well up to this point, as shown in Figure 24, but unfortunately the 'next command' simply exits the connection.

Log Handling

Other features such as the alert association features cannot be completed without functions that can gather logs, so this is the obvious next step.

Getting the right combination of words to be able to traverse our planned server object list was quite difficult, but eventually the approximate format "$*application $arguments* logs $*server*" was decided on. This by itself doesn't make a lot of sense, but when expanded into "Get the System Auth logs for linuxServer" it makes a bit more sense. Using Figure 14 as an example, we can actually traverse this object using the collection of arguments a user sends by extracting the variables from that message and applying it to the servers variable in the format *servers.$server.$arguments.$application*, or using our message as an example *servers."linuxServer".logs."system"."auth"*, and this returns the path of the log file the user would like.

With this decided, around 10 sentences were trained into Luis following the same format, as it proved difficult to be able to swap variables around (for example *$server*, *$application* then *$arguments*) without Luis starting to assign variables incorrectly. It will be interesting to see during testing to see how badly this may affect understanding.

After training was completed, the User Interaction module was updated with a logs function that used the above object traversal technique to get data such as address, login, port and filenames from the server object, which it then builds into a cleaner object to send to the Intent Processing object. The Intent Processing module then gained it's getFile function, which accepts the above object and uses it, plus the any-file[72] library, to create a download link for the given log file. The last 20 lines of this logfile is then read and sent back down the line to the user with minor text processing.

It was at this point that it was realised that the plan for the Data Parsing module seemed pointless. All text modification to this point has been fairly minor and in-line, and so it was decided that an extra module just to manage it would be extraneous. There is no reason why it could not be added later as an additional module, but for the sake of our application it did not seem necessary.

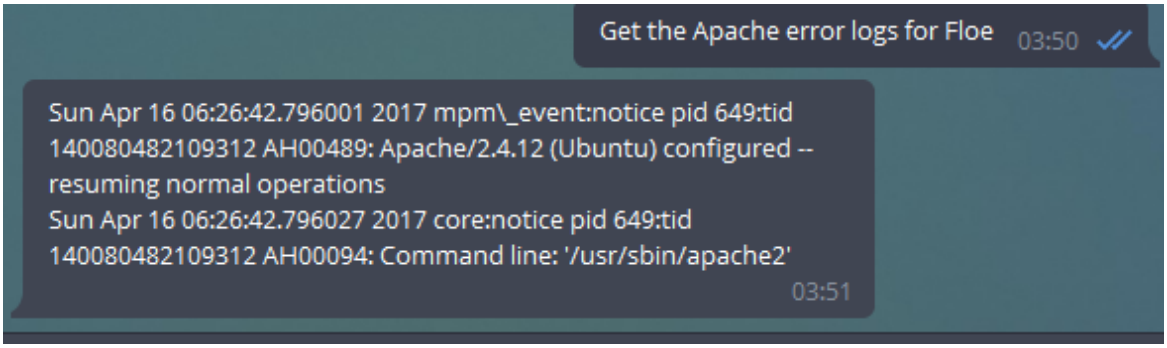

Figure 25 - Example of server object traversal and simple log parsing

Figure 21 shows an example of the log grabbing in action, with a short error file. The application is able to parse the request into the application, the argument and the server, and traverse the Servers object to figure out which log the user is looking for, then use other information to download and show that log.

Automation

As log handling is now complete, we are able to re-use those functions and information to create our final module, the Intermittent Query module. The format for intermittent queries is fairly simple, as shown in our design, it effectively contains the same information as a request from the user, an extra few variables with its name plus a delay, and finally two functions that handle the data. The first function, 'run' contains the call to the getFile function, as well as gathering any details required for it, and also manages sending users messages when it has the data. The second function, 'format' acts much as the Data Parsing module would, formatting the data as required for the user and stripping useless information.

The 'run' function is fairly self-explanatory, but the format function has some interesting uses. Firstly, it is important to note that our example use of the Intermittent Query module is to check the 'auth' logs on a Linux machine, which shows SSH connection attempts and uses of the 'sudo' command. The first interesting use is the 'IP Whitelist', which allows any IP addresses in the array without informing the user. This is useful, as the getFile process actually uses SSH to connect to the server to download logs, and we are not particularly interested in spamming the user with alerts about the application checking for problems. Secondly, we have a 'blacklist' which are text that, when detected in a line of this log file, are deleted. This is just to remove some of the more

uninteresting logs, for example removed sessions or disconnected sessions, as we are only interested in new or failed connections from IP addresses that aren't our own.

The third interesting part of the format function is the regular expression used for splitting text. Due to how Linux outputs linefeeds, it was being incorrectly formatted through messages, so it was decided to use a regular expression to split to new lines based on the date stamp. However, attempting to split on the date stamp caused it to be deleted. This was solved using a negative lookahead expression, which allows us to confirm that something exists within a string without actually matching it. The code used is shown in Figure 26.

```
text.split(/(?=[A-Z][a-z][a-z] \d\d \d\d:\d\d:\d\d)/g);
```

Figure 26 - Split text regular expression with negative lookahead

The run function uses the format function to deal with incoming data before sending it to the user. However, data cannot be sent directly to the user, we must have a saved conversation with full address information in order to do so, this is where the 'associate' intent comes into play. Firstly though, to start this function running, an automatically running function was placed inside the index file for the application, which looped through all programmed queries, and set them up to run based on their delay. Saved queries will now run automatically irrelevant of whether a user is associated with them, which would be useful in the case of needing to save data or retrieve statistics over time.

The 'associate' intent, as described above, is used to attach a user to a query, so they receive all alerts that that query is programmed to give. Similar to the other commands, we first had to teach Luis how to get arguments from them. This was a simple process as we decided on the base command "associate *x*", where 'x' related to the name of a particular Intermittent Query.

This was a simple intent, requiring a function in the User Interaction module that forwarded the users session straight into the Intent Processing module, which first checks if the query exists, and then saves their address to a global variable against the query name. At this point, the user is considered associated and will then be notified once the query they are associated with loops and has an alert.

Real World Example

During our testing, we left the application online with our example intents running, and our Telegram conversation associated with the Auth alert. After a time, it was noticed that we were getting very persistent messages of failed login attempts which escalated to over a hundred times per minute, and it became obvious that our test server was being password bruteforced.

Due to it being a test server, only designed to be up for a few days at maximum and shut down when not in use, it was not expected that this machine would be attacked and so security was not deemed top priority when configuring the server. The only reason we were able to identify that an attack was occurring was due to the application sending messages to the project lead via the IM service configured on all their devices, and they were quickly able to reconfigure the server to ban such attempts.

An example message from the program during this time is shown in Figure 27, with IP addresses blanked.

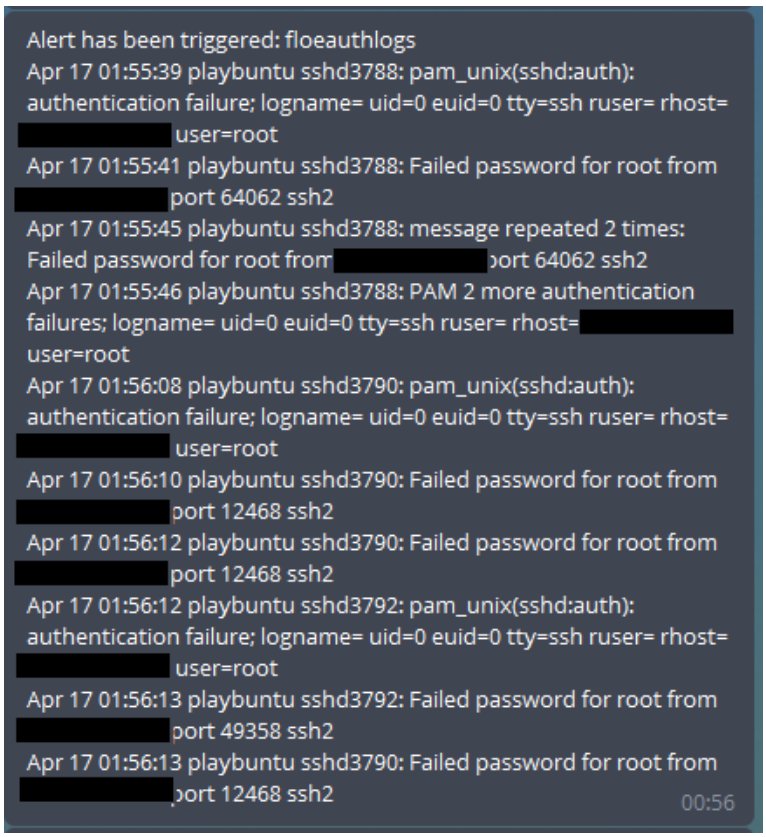

Figure 27 - Real bruteforce attack example output

Securing the Software

Securing software against use from unauthorised parties is often an important part of any application, but in this case is somewhat easier than expected. The method for securing it as described in the design seemed to work well, only allowing particular User ID's access to any commands. This was set as a simple if-else inside the NLP Module, and the whitelist set in the setup file. If the user's ID is in the setup file, then allow them to continue, else tell them they are not

Figure 28 - User failing to authenticate with the bot

permitted and give them their user ID for debugging purposes, as seen in Figure 28.

Finally, the completed implementation framework with examples flow is shown in Figure 29. Some modules are intentionally missing or have been modified from their original design, most of which is explained in the above section but will also be touched on in final overview of the project.

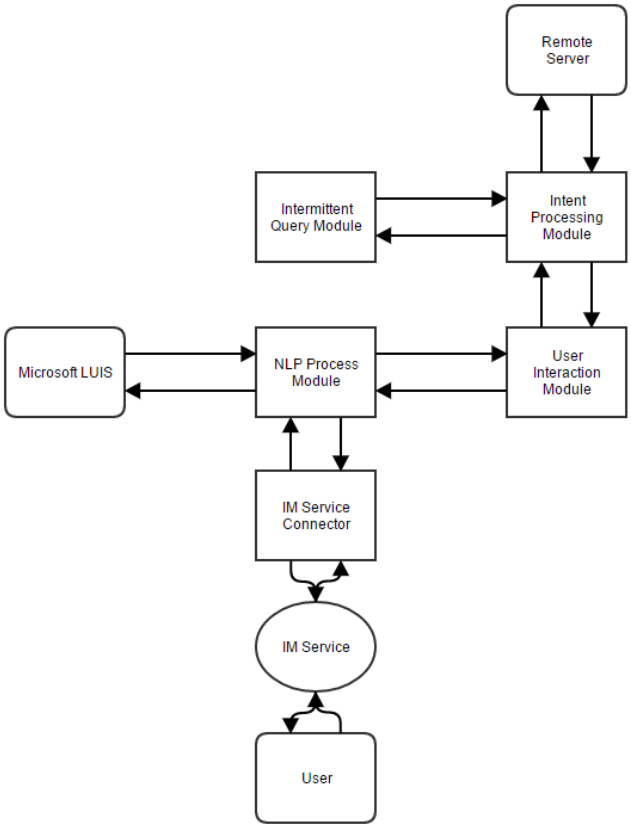

Figure 29 - Final implementation flow chart

Testing

It was decided that testing the application will be split into three separate parts, one to test LUIS and its training, one to test the application's reaction to commands based on the design and one to test the applications reaction based on commands not in the design.

1. Try three different ways of calling each intent and making sure that it picks the right intent, to test the LUIS portion of the application
2. Try three different requests for each intent, to ensure the application is handling requests properly when a correct intent is chosen.
3. Try several bogus requests, to ensure the application is handling non-intents. This will be combined with 2. to create an application testing sheet.

The full results of this can be seen in the appendix, under 'Tests', but while the application succeeded in most tests, and even succeeded in some tests that were designed to fail due to LUIS'

being able to make fairly good assumptions at intents, there were also a few failures due to oversights in the code or when training.

The testing methodology of splitting into two sections was chosen because technically, LUIS was a choice that could be replaced with any other NLP service as the application supports that. This means testing of LUIS is slightly less relevant than the testing of the actual application. A simple layout was chosen for both, Showing the test number, what the test is, the content of the message sent to the application, the expected and actual outcome and any notes. A few examples are shown in Figure 30 of the tests and the rest can be seen in the appendix.

#	Test	Expected Outcome	Actual Outcome	Message Content	Notes
1	Application handles 'version' intent correctly	Pass	Pass	"Version"	
2	Application handles 'version' intent correctly	Pass	Pass	"What is your version"	
3	Application handles 'version' intent correctly	Pass	Pass	"What version number are you at?"	
4	Application handles 'help' intent correctly	Pass	Pass	"Help"	
5	Application handles 'help' intent correctly	Pass	Pass	"Help me"	

Figure 30 - Example of tests performed on the application

Critical Analysis

The specification for this project was a fairly open framework style application, which allowed for a quite a bit of experimentation when it came to designing the final product. There were three high level goals that needed to be met to consider the project successful, and this analysis section will be a critical evaluation of the project versus its specification.

The first objective for the project was to review literature related to chat bots, natural language processing, cyber security and visualisation methods. This research showed the natural progression of chat bots and natural language, including uncovering some services that offered easy connections between the two technologies to have reactive and understanding conversations with bots. The research into cyber security showed how quickly things can go wrong and how people need to be informed of issues. This led onto the idea of parsing large quantities of log data in different ways to make it easier to consume for a system administrator. Furthering on the cyber security research, it was shown that a lot of security auditing and live alert tools require a user to be on-site and have direct access to the work network, or required a laptop or desktop to access the information. It was at this point that it was decided that a remote network administration tool using chat bots and natural language processing would be a worthy area of research.

The second objective was the one identified in the first – Address the inability for administrators to be able to access commands from within a network from a remote location. Specifically, we were interested in providing a framework that allowed an administrator to set up a remote dashboard away from the environment they are looking at, as a tool for accessing commands and the response to those commands.

The third objective was to address the connection between notification systems, such as our chat bot, and data generation systems, such as logs and security alerts. This means having a very open framework that allows various methods for gathering logs and other data. This objective acted as an extension to the second one – Where the second one allowed the idea of troubleshooting being done remotely by the use of commands, this objective allowed the gathering and analysis of data, the results of which to be delivered outside the network.

We believe that we did well at achieving our objectives, as well as the stated aims in our specification. We successfully reviewed literature to do with chat bots, natural language processing, cyber security and visualisation methods, and arrived at a conclusion based upon that research. However, more research could have been done relating to remote administration, and perhaps the

research was slightly too focussed on the idea of building a natural language processing tool rather than using a service that already existed.

We also created a framework capable of creating chat bots and easily expanding upon its functions in a modular fashion, and while there were some small deviations from the original design, the implementation worked well and we were able to use the bot to do some basic commands like pinging and tracerouting remotely. To address the third objective, we also created a log gathering function along with an alerts function that was able to inform us of an attack on a production server as it was happening, which was not even intended. This proves that the tool, even in its current example form is still capable of being a useful asset.

We were able to connect the framework to an Instant Messaging Service using Microsoft's Bot Builder, which allows us to connect to many different types of IM service from Telegram to Skype to Slack. However, we were not able to prove that the framework is as modular as it was designed to be, as because training takes a while, we could not set up the framework to another NLP service to see if it could be easily swapped out. This is unfortunate, but we are confident that the fairly service-ambiguous design of the framework would allow for easy replacement of modules, as long as the person writing them followed the design and existing data structures correctly.

There were several smaller requirements in the problem specification that the example application and framework did not meet, such as supporting API reads and Windows Management Instrumentation tools. However, we believe that this type of application will gain traction as a useful tool in the system administration world, and that this is a very good start in that area. The example functions are not incredibly important, but the framework on which it works is, as it allows for easy expansion of an already powerful toolset.

Conclusions

When starting this project we had several key areas of interest: Cyber Security, Data Visualisation, Natural Language Processing and System Administration. We expected to find some data to back up our theory that systems were hard to work with away from tools, but when researching these areas we found existing issues of visualising security data reliably, met with the inability of data visualisation even being possible outside of a dedicated work environment. Having identified this as the problem, we then created an application in an attempt to combat it.

We conclude that our design for a framework was successful, and while our implementation may need further refinement, using this type of system in a production environment is certainly possible and was proven in our testing to be able to inform users of an attack in a real-world scenario. The idea of reliable instant notification systems for important systems and services is an area of security that seems sorely neglected, and systems that allow users to deal with those issues instantly from wherever they are even more so.

Further Work

If more time were available for this project we would have liked to include support for the Windows operating system, as well as connection to the Windows Management Instrumentation framework. It also would have benefited from a real-time API of alert data and simply more data to process. We would like to see this type of system be used as part of an intrusion detection system in the future, to allow administrators to directly respond to an issue from anywhere once alerted of the problem.

Bibliography

[1] M. I. B. Rafiullah Khan, Sarmad Ullah Khan, Rifaqat Zaheer, “An Efficient Network Monitoring and Management System,” *Int. J. Inf. Electron. Eng.*, vol. 3, no. 1, p. 122, 2013.

[2] Nagios, “Nagios - The Industry Standard In IT Infrastructure Monitoring,” 2017. [Online]. Available: https://www.nagios.org/. [Accessed: 25-Feb-2017].

[3] N. Chandrika, “Cyber Security in the UK,” *POSTnote*, no. 389, pp. 1–4, 2011.

[4] Symantec, “W32.Stuxnet Dossier,” 2011.

[5] Rockerfeller, “A ‘ Kill Chain ’ Analysis of the 2013 Target Data Breach,” pp. 1–16, 2014.

[6] R. Sharma, “Study of Latest Emerging Trends on Cyber Security and its challenges to Society,” *Int. J. Sci. Eng. Res.*, vol. 3, no. 6, 2012.

[7] Microsoft, “Windows 8 Release Preview Product guide,” 2012.

[8] Android, “Android,” 2017. [Online]. Available: https://www.android.com/. [Accessed: 21-Apr-2017].

[9] G. A. Fink, C. L. North, A. Endert, and S. Rose, “Visualizing cyber security: Usable workspaces,” in *6th International Workshop on Visualization for Cyber Security 2009, VizSec 2009 - Proceedings*, 2009, pp. 45–56.

[10] Cisco, “Introduction to Cisco IOS NetFlow - A Technical Overview - Cisco,” 2012. [Online]. Available: http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html. [Accessed: 21-Apr-2017].

[11] Snort, “Snort - Network Intrusion Detection & Prevention System,” 2017. [Online]. Available: https://www.snort.org/. [Accessed: 21-Apr-2017].

[12] H. Shiravi, A. Shiravi, and A. a. Ghorbani, “006 A survey of visualization systems for network security,” *IEEE Trans. Vis. Comput. Graph.*, vol. 18, no. 8, pp. 1313–1329, 2012.

[13] R. Collobert, J. Weston, and L. Bottou, “Natural language processing (almost) from scratch,” *J. Mach. …*, vol. 12, pp. 2493–2537, 2011.

[14] P. M. Nadkarni, L. Ohno-Machado, and W. W. Chapman, “Natural language processing: an introduction.,” *J. Am. Med. Inform. Assoc.*, vol. 18, no. 5, pp. 544–51, 2011.

[15] D. A. Ferrucci, “Introduction to "This is Watson"”

[16] R. S. Wallace, “The Anatomy of A.L.I.C.E.,” 2002. [Online]. Available: http://www.alicebot.org/anatomy.html.

[17] H. Loebner, “Home Page of the Loebner Prize,” 2015. [Online]. Available: http://www.loebner.net/Prizef/loebner-prize.html. [Accessed: 21-Apr-2017].

[18] A. M. Turing, “Computing Machinery and Intelligence A.M. Turing,” 1950. [Online]. Available: http://www.loebner.net/Prizef/TuringArticle.html. [Accessed: 21-Apr-2017].

[19] J. Weizenbaum, “ELIZA--A Computer Program For the Study of Natural Language Communication Between Man and Machine,” *Commungicatins ACM*, vol. 9, no. 1, pp. 36–35, 1966.

[20] MIT, “MIT - Massachusetts Institute of Technology,” 2017. [Online]. Available: http://web.mit.edu/. [Accessed: 21-Apr-2017].

[21] B. A. Shawar and E. Atwell, “Chatbots: are they really useful?,” *LDV-Forum*, 2007.

[22] R. Wallace, “ALICE and AIML Software and Downloads - ALICE A. I. Foundation Natural Language Chat Robot (Chatterbot) Programming and Virtual Personality Development Tools,” 2017. [Online]. Available: http://www.alicebot.org/downloads/sets.html. [Accessed: 21-Apr-2017].

[23] B. A. Shawar and E. Atwell, “Using dialogue corpora to train a chatbot,” 2003.

[24] P. Millican, “Elizabeth,” 2015. [Online]. Available: http://www.philocomp.net/ai/elizabeth.htm. [Accessed: 21-Apr-2017].

[25] J. Rahman, “Implementation of ALICE chatbot as domain specific knowledge bot for BRAC U (FAQ bot),” 2012.

[26] Brac U, “BRAC University,” 2017. [Online]. Available: http://www.bracu.ac.bd/. [Accessed: 21-Apr-2017].

[27] A. Khanna, B. Pandey, K. Vashishta, K. Kalia, B. Pradeepkumar, and T. Das, “A Study of Today’s A.I. through Chatbots and Rediscovery of Machine Intelligence,” *Int. J. Sci. Technol.*, vol. 8, no. 7, pp. 277–284, 2015.

[28] P. Simard, D. Chickering, A. Lakshmiratan, D. Charles, L. Bottou, C. Garcia, J. Suarez, D. Grangier, S. Amershi, J. Verwey, and J. Suh, “ICE: Enabling Non-Experts to Build Models Interactively for Large-Scale Lopsided Problems,” 2014.

[29] K. Sanjiv and E. M. Reingold, “Optimum Lopsided Binary Trees,” *J. ACM*, vol. 36, no. 3, pp. 573–590, 1989.

[30] J. D. Williams, E. Kamal, M. Ashour, H. Amr, J. Miller, and G. Zweig, “Fast and easy language understanding for dialog systems with Microsoft Language Understanding Intelligent Service (LUIS),” 2015.

[31] R. Kar and R. Haldar, “Applying Chatbots to the Internet of Things: Opportunities and Architectural Elements,” 2016.

[32] Tesla, “Model S | Tesla UK,” 2017. [Online]. Available: https://www.tesla.com/en_GB/models. [Accessed: 21-Apr-2017].

[33] nest, “Meet the Nest Learning Thermostat | Nest,” 2017. [Online]. Available: https://nest.com/uk/thermostat/meet-nest-thermostat/. [Accessed: 21-Apr-2017].

[34] nest, “API | Nest Developers,” 2017. [Online]. Available: https://developers.nest.com/documentation/api-reference. [Accessed: 21-Apr-2017].

[35] Microsoft, “Intro to the Universal Windows Platform - UWP app developer | Microsoft Docs,” 2017. [Online]. Available: https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide. [Accessed: 21-Apr-2017].

[36] W. C. Mann, “Dialogue Diversity Corpus,” 2003. [Online]. Available: http://www-bcf.usc.edu/~billmann/diversity/DDivers-site.htm. [Accessed: 15-Dec-2016].

[37] Microsoft, “LUIS: Language Understanding Intelligent Service (beta),” 2017. [Online]. Available: https://www.luis.ai/. [Accessed: 15-Dec-2016].

[38] Nagios, “Operating Systems - Nagios Exchange,” 2017. [Online]. Available: https://exchange.nagios.org/directory/Plugins/Operating-Systems. [Accessed: 25-Feb-2017].

[39] Nagios, “Nagios Log Server - Monitoring a New Log Source,” 2016.

[40] Nagios, “Service Checks · Nagios Core Documentation,” 2017. [Online]. Available: https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/servicechecks.html. [Accessed: 25-Feb-2017].

[41] Nagios, “Notifications · Nagios Core Documentation,” 2017. [Online]. Available: https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/notifications.html. [Accessed: 25-Feb-2017].

[42] Opsview, “Why Scalability is Such a Big Concern When Using Nagios | Opsview,” 2016. [Online]. Available: https://www.opsview.com/resources/blog/why-scalability-such-big-concern-when-using-nagios. [Accessed: 25-Feb-2017].

[43] Webmin, “Webmin,” 2017. [Online]. Available: http://webmin.com/. [Accessed: 25-Feb-2017].

[44] Webmin, “Cluster Webmin Servers - Webmin Documentation,” 2017. [Online]. Available: http://doxfer.webmin.com/Webmin/Cluster_Webmin_Servers. [Accessed: 25-Feb-2017].

[45] Webmin, “API Webmin - Webmin Documentation,” 2017. [Online]. Available: http://doxfer.webmin.com/Webmin/API_Webmin#show_webmin_notifications.28.5Bno-updates.5D.29. [Accessed: 25-Feb-2017].

[46] Jack Wallen, “Webmin: One big drawback to using this data center management tool - TechRepublic,” 2016. [Online]. Available: http://www.techrepublic.com/article/webmin-one-big-drawback-to-using-this-data-center-management-tool/. [Accessed: 25-Feb-2017].

[47] Microsoft, “Remote Server Administration Tools (RSAT) for Windows operating systems,” 2016. [Online]. Available: https://support.microsoft.com/en-gb/help/2693643/remote-server-administration-tools-rsat-for-windows-operating-systems. [Accessed: 25-Feb-2017].

[48] Icinga, “Icinga – Open Source Monitoring,” 2017. [Online]. Available: https://www.icinga.com/. [Accessed: 25-Feb-2017].

[49] Icinga, “Service Monitoring :: Icinga Web,” 2017. [Online]. Available: https://docs.icinga.com/icinga2/latest/doc/module/icinga2/toc#!/icinga2/latest/doc/module/icinga2/chapter/service-monitoring#service-monitoring-windows. [Accessed: 25-Feb-2017].

[50] Icinga, “icinga2 Documentation - Notification Script and Interfaces,” 2017. [Online]. Available: https://docs.icinga.com/icinga2/latest/doc/module/icinga2/toc#!/icinga2/latest/doc/module/icinga2/chapter/addons#notification-scripts-interfaces. [Accessed: 25-Feb-2017].

[51] C. Haen, E. Bonaccorsi, and N. Neufeld, “DISTRIBUTED MONITORING SYSTEM BASED ON ICINGA,” 2011.

[52] OpenNMS, “Installation |,” 2017. [Online]. Available: https://www.opennms.org/en/install. [Accessed: 25-Feb-2017].

[53] OpenNMS, “Configuring notifications - OpenNMS,” 2017. [Online]. Available: https://wiki.opennms.org/wiki/Configuring_notifications. [Accessed: 25-Feb-2017].

[54] Jeff Gehlbach, “Scalability of opennms,” 2008. [Online]. Available: http://opennms-discuss.narkive.com/wIRXrWZk/scalability-of-opennms. [Accessed: 25-Feb-2017].

[55] Cisco, “Cisco Jabber - Cisco,” 2017. [Online]. Available: http://www.cisco.com/c/en/us/products/unified-communications/jabber/index.html. [Accessed: 21-Apr-2017].

[56] Telegram, “Telegram Messenger,” 2017. [Online]. Available: https://telegram.org/. [Accessed: 05-Mar-2017].

[57] Google, “Cloud Natural Language API | Google Cloud Platform,” 2017. [Online]. Available: https://cloud.google.com/natural-language/. [Accessed: 21-Apr-2017].

[58] wit.ai, “Wit.ai,” 2017. [Online]. Available: https://wit.ai/. [Accessed: 21-Apr-2017].

[59] LUIS, “LUIS: Homepage,” 2017. [Online]. Available: https://www.luis.ai/home/index. [Accessed: 05-Mar-2017].

[60] Apache Software Foundation, “Welcome! - The Apache HTTP Server Project,” 2017. [Online]. Available: https://httpd.apache.org/. [Accessed: 21-Apr-2017].

[61] Apache Software Foundation, “Log Files - Apache HTTP Server Version 2.4,” 2017. [Online]. Available: https://httpd.apache.org/docs/2.4/logs.html. [Accessed: 21-Apr-2017].

[62] Ubuntu, “LinuxLogFiles - Community Help Wiki,” 2015. [Online]. Available: https://help.ubuntu.com/community/LinuxLogFiles. [Accessed: 23-Mar-2017].

[63] Microsoft, “Microsoft Bot Framework,” 2017. [Online]. Available: https://dev.botframework.com/. [Accessed: 23-Mar-2017].

[64] Viha, “IRC.org - Home of IRC,” 2005. [Online]. Available: http://www.irc.org/. [Accessed: 21-Apr-2017].

[65] Restify, “API Guide | restify,” 2016. [Online]. Available: http://restify.com/. [Accessed: 14-Apr-2017].

[66] A. Shreve, “ngrok - secure introspectable tunnels to localhost,” 2017. [Online]. Available: https://ngrok.com/. [Accessed: 23-Mar-2017].

[67] Microsoft, “Bot Framework Emulator | Documentation | Bot Framework,” 2017. [Online]. Available: https://docs.botframework.com/en-us/tools/bot-framework-emulator/. [Accessed: 23-Mar-2017].

[68] Microsoft, “Microsoft Azure: Cloud Computing Platform & Services,” 2017. [Online]. Available: https://azure.microsoft.com/en-gb/. [Accessed: 23-Mar-2017].

[69] G. Pretty, “Using the Microsoft LUIS service to build a model to understand natural language input – Gary Pretty,” 2016. [Online]. Available: http://www.garypretty.co.uk/2016/07/20/using-microsoft-luis-service-understand-natural-language-input/. [Accessed: 21-Apr-2017].

[70] D. Zelisko, “ping - a ping wrapper for nodejs,” 2016. [Online]. Available: https://www.npmjs.com/package/ping. [Accessed: 16-Apr-2017].

[71] W. James, “traceroute - Simple wrapper around the native traceroute command,” 2016. [Online]. Available: https://www.npmjs.com/package/traceroute. [Accessed: 16-Apr-2017].

[72] M. Colomer, “any-file,” 2017. [Online]. Available: https://www.npmjs.com/package/any-file. [Accessed: 17-Apr-2017].

Appendix.

Code

Package.json

```
{
  "name": "nlp-im-rat",
  "version": "0.0.1",
  "description": "Natural Langue Parsing Instant Messaging Remote
Administration Tool - Connor's LJMU Final Year Project",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://Makeshift@github.com/Makeshift/NLP-IM-RAT.git"
  },
  "author": "Connor Bell",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/Makeshift/NLP-IM-RAT/issues"
  },
  "homepage": "https://github.com/Makeshift/NLP-IM-RAT#readme",
  "dependencies": {
    "any-file": "0.0.23",
    "botbuilder": "^3.2.3",
    "diff": "^3.2.0",
    "ping": "^0.2.2",
    "read-last-lines": "^1.1.2",
    "restify": "^4.1.1",
    "ssh2": "^0.5.4",
    "traceroute": "^1.0.0"
  }
}
```

Index.js

```
//=========================================================
// NLPIMRAT
// Description:
```

```
/*
    This file is the one you start to start the app. It contains all of the
required includes for it to work,
    as well as the initial bot setup and importing of all the modules.
*/
//=========================================================

//Required includes
var restify = require('restify'); //HTTP server
var builder = require('botbuilder'); //Microsoft Botbuilder
var fs = require('fs'); //Filesystem access

//Special setup file with all my hidden vars
var setup = require('./setup.js');

//=========================================================
// Bot Setup
//=========================================================

// Setup Restify Server
var server = restify.createServer();
server.listen(3978, function () { //Arbitrary port
   console.log('%s listening to %s', server.name, server.url);
});

// Create chat bot
var connector = new builder.ChatConnector({
    appId: setup.appId,
    appPassword: setup.appPassword
});
bot = new builder.UniversalBot(connector);
//Opens up a post endpoint for the connector to listen on
server.post('/api/messages', connector.listen());
//=========================================================
// Modules
//=========================================================

//User Interaction Module
var mUI = require('./module-User_Interaction.js');
//Natural Langauge Processing Module
//LUIS Module configuration
```

```
var recognizer = new builder.LuisRecognizer(setup.azureModel);
//Takes over from the inbuilt parser to configure intents
var dialog = new builder.IntentDialog({ recognizers: [recognizer] });
//Import the actual LUIS module as an eval for simplicity - There are so
many references
//to variables in other files it is a lot simpler to keep it as an eval
rather than converting it into a module.
eval(fs.readFileSync('module-LUIS_NLP.js')+'');
//Intent Processing Module
var mIP = require('./module-Intent_Processing.js');
//Server list
var serverDefs = require('./servers.js');
//Intermittent Server Query Module
var mIQ = require('./module-Intermittent_Query.js');
//Global timer for Intermittent Queries
(function() {
    for (var i = 0; i < mIQ.programmedQueries.length; i++) {
        setInterval(function(x) {
            mIQ.programmedQueries[x].run();
        }, mIQ.programmedQueries[i].delay, i);
    }
}());
```

Module-Intent_Processing.js

```
//=========================================================
// Intent Processing Module
// Description:
/*
    This file processes user requests and returns data back to the User
Interaction module.
*/
//=========================================================

//Various imports for functions
var ping = require('ping');
var dns = require('dns');
var traceroute = require('traceroute');
var Client = require('ssh2').Client;
var AnyFile = require('any-file');
var readLastLines = require('read-last-lines');
```

```
var setup = require('./setup.js');

module.exports = {
    //List of processes we can use
    process: {
        //Ping function
        ping: function(address, cb) {
            //Ping doesn't support using Windows internal DNS, so we do a
proper DNS lookup on incoming addresses
            //before sending it to ping. eg. we want it to be able to take
an internal DNS name like Computer001 and
            //convert it to a local IP.
            dns.lookup(address, function(err, dnsAddress) {
                ping.sys.probe(dnsAddress, function(isAlive) {
                    console.log("Pinging address: " + dnsAddress);
                    var msg = isAlive ? `Host ${address} is alive (Resolved
to ${dnsAddress})` : `Host ${address} is dead (Resolved to ${dnsAddress})`;
                    cb(msg);
                }, null, cb);
            });
        },
        traceroute: function(address, cb) {
            cb("Tracerouting address "+address+". Could take some time.")
            traceroute.trace(address, function(err, hops) {
                console.log("Tracerouting address: " + address);
                if (err) {
                    console.log(err);
                    cb(err);
                } else {
                    console.log(hops);
                    //For each hop
                    for (var i = 0; i < hops.length; i++) {
                        //Traceroute returns with an array containing
objects. Each object contains
                        //one key(named the IP) containing an array with
the millisecond timing of
                        //the ping to each hop (should be 3 pings);
                        //So we average the ping amount and grab the name
of the key, calling our callback
                        //each time to send another message.
                        if (typeof hops[i] == "object") {
```

```
                            var avg = 0;
                            var ip = Object.keys(hops[i])[0];
                            for (var x = 0; x < hops[i][ip].length; x++) {
                                avg += hops[i][ip][x];
                            }
                            avg = Math.floor(avg / hops[i][ip].length);
                            cb("Hop "+i+": "+ip + ", average " + avg +
"ms.");
                        } else {
                            cb("Hop "+i+" failed.");
                        }
                    }
                }
            });
        },
        ssh: function(connectionData, userID, cb) {
            //SSH connects to a server using the details given by the user,
waits for the text to buffer
            //and then returns it the user, exiting cleanly.
            var conn = new Client();
            conn.on('ready', function() {
              conn.shell(function(err, stream) {
                if (err) throw err;
                //Allow 500ms for data streaming, otherwise we spam the
user pretty hard
                //Note that setTimeout is considered synchronous so output
ordering is important
                //here.
                var streamed = "";
                setTimeout(function() {
                    console.log(streamed);
                    cb(streamed);
                    streamed = "";
                }, 500)
                stream.on('close', function() {
                  console.log('Connection exited');
                  conn.end();
                  cb("Connection ended.");
                }).on('data', function(data) {
                    streamed += data;
                }).stderr.on('data', function(data) {
```

```
                    streamed += data;
                });
                steam.end('exit\n');
            });
        }).connect({
            host: connectionData[0],
            port: connectionData[3],
            username: connectionData[1],
            password: connectionData[2]
        });
    },
    getFile: function(data, cb) {
        //Set our defaults early because scp2 doesn't allow us to set a
port during the initial connection
        //And set a couple of others at the same time because we can
        //Generate the SCP 'from' location as a single line
        var af = new AnyFile();
        var fileLoc =
`${data.protocol}:\/\/${data.username}:${data.password}@${data.address}${da
ta.file.path}${data.file.file}`;
        console.log("Grabbing data: " + fileLoc);
        var localFile = "./downloads/"+data.userid;
        //Download the file locally to a known location
        af.from(fileLoc).to(localFile, function(err, res) {
            if(res) {
                console.log("Copy complete, tailing file");
                readLastLines.read(localFile, 20).then(function(lines)
{
                    //console.log(lines);
                    cb(lines.replace("\n", "  \n"));
                });
            } else {
                console.log("Copy failed, informing user")
                console.log(err);
                cb(err);
            }
        });
    },
    //This function associates an IM user with a particular alert, so
they get alerts that get called by the intermittent query module.
    associate: function(data, session, cb) {
```

```
            var queryExists = false;
            //Checks if the query they want to associate with exists
            for (var i = 0; i < require('./module-
Intermittent_Query.js').programmedQueries.length; i++) {
                if (require('./module-
Intermittent_Query.js').programmedQueries[i].name.toLowerCase() ==
data.toLowerCase()) {
                    queryExists = true;
                }
            }
            //If it does, add them to the association variable and informs
them
            //Else, inform them that their query wasn't found.
            if (queryExists) {
                setup.globalAssociateVar.push([data.toLowerCase(),
session.message.address]);
                cb("This conversation has been associated with the query "
+ data + ". You will receive alerts based on the rules set.")
            } else {
                cb(data + " doesn't exist as a programmed query.")
            }
        }
    },
}
```

Module-Intermittent_Query.js

```
//=========================================================
// Server Intermittent Query Module: Intermittent_Query
// Description:
/*
    This module handles automatic queries to users based on incoming server
data.
    Has an associated .json file for persistent address saving for users.
*/
//=========================================================

//Imports
var jsdiff = require('diff');
var mIP = require('./module-Intent_Processing.js');
var serverDefs = require('./servers.js');
```

```
var setup = require('./setup.js');
var builder = require('botbuilder'); //Microsoft Botbuilder

module.exports = {
    programmedQueries: [
        {
            name: 'floeAuthLogs', //Used to attach users to the alert
            server: 'floe', //Must be defined in servers.js
            type: 'logs',
            application: 'system',
            logfile: 'auth',
            delay: 60000, //Time in milliseconds between checks of this
file
            format: function(text) { //Formats the data and only gets data
we are interested in
                var whitelist = setup.whitelist; //List of IP addresses
that are whitelisted and we don't want notices about failing
                //We still show all logins, regardless of whitelist
                var blacklist = ['Removed session', 'Disconnected from',
'Received disconnect', 'New session', 'session opened', 'session closed',
'connection closed'] //This is a list of strings we know might turn up in
our auth logs that we don't care about, and can discard
                for (var i = 0; i < whitelist.length; i++) {
                    //Adds 'accepted login' from our whitelisted IP's to be
on the blacklist so we don't spam the user with successful logins from the
application
                    blacklist.push('Accepted password.*'+whitelist[i])
                }
                var lineText = text.split(/(?=[A-Z][a-z][a-z] \d\d
\d\d:\d\d:\d\d)/g); //Regex that splits text based on what we know is on
each line - The date stamp. Uses a negative lookahead to not accidentally
remove that data.
                var finalText = "";
                for (var i = 0; i < lineText.length; i++) {
                    if (!new RegExp(blacklist.join("|")).test(lineText[i]))
{ //Little REGEX generator snippet
                        finalText += lineText[i]+"  \n";
                    }
                }
                return finalText;
            },
```

```
            run: function(cb) { //Handles getting the data and sending it
                //Make a few more variables here because inside the process
the 'this' keyword changes, so we can't access
                //object data anymore. Scoping.
                var longFileName =
serverDefs.servers[this.server][this.type][this.application][this.logfile];
                var name = this.name.toLowerCase();
                var format = this.format;
                //Generates a getfile object
                mIP.process.getFile(
                    {
                        protocol: serverDefs.servers[this.server].protocol,
                        username: serverDefs.servers[this.server].username,
                        password: serverDefs.servers[this.server].password,
                        address: serverDefs.servers[this.server].address,
                        file: longFileName,
                        userid: 'interm'
                    }, function(file) {
                        //Compares it against the previous file to get the
diff
                        if (typeof setup.globalCompVar[longFileName.file]
=== 'undefined') {
                            setup.globalCompVar[longFileName.file] = file;
                        } else {
                            var diff =
jsdiff.diffLines(setup.globalCompVar[longFileName.file], file);
                            diff.forEach(function(part) {

                                if (part.added) {
                                    //Formats it as requested in the above
object
                                    //Sends it to the user based on the
associated conversation
                                    part.value = format(part.value);
                                    for (var i = 0; i <
setup.globalAssociateVar.length; i++) {
                                        if (setup.globalAssociateVar[i][0]
=== name && part.value.length > 5) {
                                            var msgSetup = "Alert has been
triggered: " + name + "  \n  \n";
```

```
                                                var msg = new
builder.Message().text(msgSetup+part.value).address(setup.globalAssociateVa
r[i][1]);
                                                bot.send(msg);
                                            }
                                        }
                                    }
                                });
                                setup.globalCompVar[longFileName.file] = file;
                            }
                        });
                },
            }
        ],
}
```

Module-LUIS_NLP.js

```
//=========================================================
// NLP Module: LUIS_NLP
// Description:
/*
    This file is the NLP management module.
    This contains all 'matches' for the NLP system, formats incoming data
and sends it to the user interaction module.
*/
//=========================================================

bot.dialog('/', dialog);

//Grabs known intents from user interaction module, creates a template
handler for each
for (var i = 0; i < mUI.knownIntents.length; i++) {
    var match = mUI.knownIntents[i];
    //Because we can't create new dialog matches in this loop without
overwriting them each time
    //we need to make a text version of the intent template and then eval
it to create all our dialogs
    //without the need for the user to manually write them all.
    //As long as it follows the same template for the UI module, it will be
fine.
```

```
    var intentGenerator = `
    dialog.matches('${match}', [
        function (session, args) {
            //Ensure our user is whitelisted so not everybody can use the
system
            if (setup.userWhitelist.indexOf(session.message.user.id) > -1)
{
                console.log("Handling intent: ${match}")
                var arguments = {};
                //Handle all our arguments and format them in a predictable
way
                for (var i = 0; i < args.entities.length; i++) {
                    arguments[args.entities[i].type] =
args.entities[i].entity;
                }
                //Sends intent plus data to the UIM for processing, then
returns back with the
                //text for the user
                mUI.intentHandler['${match}'](arguments, function(response,
replaceMulti, data) {
                    //For standard messages
                    if (typeof response == "string") {
                        session.send(response);
                    } else if (typeof response == "array") {
                        for (var i = 0; i < response.length; i++) {
                            session.send(response[i]);
                        }
                    }
                    //For initiating a dialog tree
                    if (replaceMulti) {
                        console.log(arguments);
                        session.replaceDialog('/convo', {intent:
"${match}", internal: [data], count: 0})
                    }
                }, session);
            } else {
                //Deal with people who aren't meant to use the bottle
                session.send("You are not authorised to use this bot. ID: "
+ session.message.user.id);
            }
        }
```

```
    ]); `;
    eval(intentGenerator);
}
//Multi-layered conversation(Dialogue)
bot.dialog('/convo', [
    function(session, args) {
        mUI.intentHandler[args.intent+'Convo'](args.internal, args.count,
session.message.user.id, function(type, response, reset) {
            //If reset, we're deleting all of the saved data thus far to
pass new data onto the UI module
            if (!reset) {
                session.dialogData.internal = args.internal;
            }
            console.log(session.dialogData);
            //Handles the return from the UI module. If prompt then prompt
the user and move onto the next function to handle responses
            //If exit, kill the dialog
            //If else, assume we're sending a one-off message to the user
then restart the conversation to query the UI for more data
            if (type == "prompt") {
                session.dialogData.intent = args.intent;
                session.dialogData.count = args.count+1;
                builder.Prompts.text(session, response);
            } else if (type == "exit") {
                console.log("Dialog exited")
                session.cancelDialog();
            } else {
                session.send(response);
                console.log("IN CONVO: " + args.count);
                session.beginDialog('/convo', {intent: args.intent, count:
args.count+1, internal: session.dialogData.internal});
            }
        });
    },
    function(session, results) {
        //This function is called as a second part to a prompt
        //Quickly check if we have internal data stored and fix it if not
        if (typeof session.dialogData.internal != 'object') {
            session.dialogData.internal = [];
       }
       //Push incoming data into the internal data store
```

```
        session.dialogData.internal.push(results.response);
        //Hand back off to the start of the conversation to call the UI
module again.
        session.replaceDialog('/convo', {intent: session.dialogData.intent,
count: session.dialogData.count, internal: session.dialogData.internal})
    }
]);

//Default dialog in case something goes horrifically wrong and LUIS fails
to find any intent at all, including no intent
dialog.onDefault(builder.DialogAction.send("I'm sorry, I didn't
understand."));
```

Module-User_Interaction.js

```
//=========================================================
// User Interaction Module
// Description:
/*
    This file is the user interaction module. It recieves formatted data
from the NLP module and works interacts with
    other modules to make a response.
*/
//=========================================================
var mIP = require('./module-Intent_Processing.js');
var serverDefs = require('./servers.js');
module.exports = {
    //List of known intents
    //This allows us to have a static list of expected data even in the
case that the NLP module changes
    //This list does not include 'extended' intents, such as the +Convo
modifier, which allows for multi-stage dialogs
    knownIntents:
['none','version','help','ping','traceroute','ssh','identify','logs','assoc
iate'],

    //Handling of each intent, NLP module agnostic
    intentHandler: {
        //This handler is called if Luis detects the 'none' intent, which
is full of nonsense phrases designed to illicit a response
        //but not do anything meaningful.
```

```
        none: function(arg, cb) {
            var response = "No intent detected.";
            cb(response);
        },
        //This handler is called if Luis detects the 'version' intent, such as "What version are you?" or simply just "Version".
        //It replies with the verison as specified in the application package file.
        version: function(arg, cb) {
            var pjson = require('./package.json');
            var response = "I am version " + pjson.version;
            cb(response);
        },
        //This handler is called if Luis detects the 'help' intent, such as "What am I doing?" or simply just "help".
        //It replies with predefined help text.
        help: function(arg, cb) {
            var response = `Currently known intents  \n
            Version: (what version are you, tell me your version, what is your version) Responds with current version  \n
            Help: (help me, help, /start) Responds with this  \n
            Ping: (ping address, can you ping address for me, please ping address) Tells you if a given endpoint is up  \n
            Traceroute: (traceroute address, perform a traceroute against address, start traceroute to) Performs a traceroute against a given address \n
            SSH: (open ssh with address, open address, shell addresss, ssh address) Starts a dialog to get information, then opens a shell connection to a machine, returning the given text before exiting  \n
            Identify: (identify, who am I) Sends identify information the bot knows about you, such as your user ID and IM service (Mostly for debugging)  \n
            Logs: (get the aplication specific logs for server) Queries the known server list to get log data from a particular application on a server \n
            Associate: (associate x, associate me with x) Associates this conversation with a hardcoded intermittent query. You will then receive alerts from this query.
            `;
            cb(response);
        },
```

```
        //This handler is called if Luis detects the 'ping' intent, such as
"is -server- up?, "is -server- online?" or "ping server".
        //It defers to the intent processing module to deal with the actual
processing after sanitizing the input
        //and acts as middleware between the LUIS and proccessing modules.
        ping: function(arg, cb) {
            console.log(arg);
            mIP.process.ping(arg.arguments.replace(/\s/g, ""),
function(response){
                cb(response);
            });
        },
        //This handler is called if Luis detects the 'traceroute' intent,
and responds to 'traceroute server', 'trace a route to x server' or similar
        //Similarly, it defers processing to the processing module after
sanitizing input.
        traceroute: function(arg, cb) {
            console.log(arg);
            mIP.process.traceroute(arg.arguments.replace(/\s/g, ""),
function(response) {
                cb(response);
            });
        },
        //This handler is called if Luis detects the 'identify' intent. It
responds to "Who am I?", "identify", "identify me", and similar phrases.
        //This is primarily a debug handler, and just outputs data scraped
from the session related to the user.
        identify: function(arg, cb, session) {
            console.log("Identifying the user");
            cb(`Address ID: ${session.message.address.id}  \nUsing:
${session.message.address.channelId}  \nUser ID: ${session.message.user.id}
\nUsername: ${session.message.user.name}`);
            cb("Use the 'associate' command to associate this conversation
with an alert.");
        },
        //This handler is called if Luis detects the 'ssh' intent. It only
responds to 'ssh -server-'. It then forces the
        //conversation handler to hand off to another handler, designed
better for multi-message dialogue and response.
        ssh: function(arg, cb) {
            console.log(arg);
```

```
            var address = arg.arguments.replace(/\s/g, "");
            cb("Starting shell session with " + address, true, address);
        },
        //This handler is called by the previous SSH handler, and uses a
counter system to loop through the conversation to get
        //different data from the user. This data is then sent to the
intent processing module to open a connection.
        sshConvo: function(data, count, userID, cb) {
            console.log("SSHCONVO: " + count);
            console.log("USERID: " + userID);
            //Each time sshConvo is called it cycles to the next counter
            //This is a proof of concept to prove that a multi-layered
dialog plus waiting for callbacks can work
            //However it does not actually forward commands to the server,
it simply exits when the text has
            //finished buffering from the server
            if (count == 0) {
                cb("prompt", "Using what username?");
            } else if (count == 1) {
                cb("prompt", "Using what password?");
            } else if (count == 2) {
                cb("prompt", "On what port?");
            } else if (count == 3) {
                cb("text", `Connecting to ${data[0]} with user '${data[1]}'
and your specified password on port ${data[3]}...`);
                mIP.process.ssh(data, userID, function(response) {
                    cb("text", response);
                    cb("text", "Connected. Further commands will be
forwarded to the server. Type 'exit' to exit session.", true);
                });
            } else if (count == 4) {
                cb("prompt", "Buffering login text...", true);
            } else if (count >= 5) {
                cb("exit");
            }
        },
        //This handler is called if Luis detects the 'logs' intent. It
detects the $application, $argument and $server from the users
        //message and works out exactly what the user is trying to do,
before passing it off to intent processing.
        logs: function(arg, cb, session) {
```

```
            try {
                //Generate the getFile object so we don't have to deal with
it later
                mIP.process.getFile({
                    protocol: serverDefs.servers[arg.server].protocol,
                    address: serverDefs.servers[arg.server].address,
                    port: serverDefs.servers[arg.server].port,
                    username: serverDefs.servers[arg.server].username,
                    password: serverDefs.servers[arg.server].password,
                    file:
serverDefs.servers[arg.server].logs[arg.application][arg.arguments],
                    userid: session.message.user.id
                }, function(line) {
                    //Call back with the lines to inform the user
                    cb(line);
                });
            } catch (e) {
                //Try/catch to try and grab any specific errors - The big
one being an undefined when looking for a server that doesn't exist
                //So let's tell the user about it and see if they can fix
their syntax.
                console.log(e);
                cb(`We weren't able to find the correct path for the
specified server. Here's what we got from your message:  \nServer:
${arg.server}  \nApplication: ${arg.application}  \nSubsection:
${arg.arguments}`)
            }
        },
        //This handler is called if Luis detects the 'associate' intent.
Such as 'associate me with x' or 'associate x'. This sends
        //the argument and session off to the intent processor.
        associate: function(args, cb, session) {
            console.log(args);
            mIP.process.associate(args.arguments, session,
function(response) {
                cb(response);
            });
        }
    }
};
```

Servers.js

```
//=========================================================
// Server Configuration File
// Description:
/*
    This file contains the connection data for services and servers that is
referenced in other parts of the application.
*/
//=========================================================
var setup = require('./setup.js');
module.exports = {
//For reasons of security, the actual logins for the servers are not
included in the code.
    servers: {
        //First server, floe
        "floe": {
            address: 'floe.xyz',
            protocol: 'scp',
            port: 8765,
            //Login
            username: setup.floeUser,
            password: setup.floePass,
            //Locations of interest
            logs: {
                //Application specific logs - Apache
                apache:  {
                    access: {
                        path: "/var/log/apache2/",
                        file: "access.log",
                    },
                    error: {
                        path: "/var/log/apache2/",
                        file: "error.log"
                    }
                },
                //Server specific logs
                system: {
                    auth: {
                        path: "/var/log/",
                        file: "auth.log"
                    }
```

```
                }
            },
            //Commands that give log-like output
            commands: {
                loginFailLog: "faillog",
                lastLoginLog: "lastlog"
            }
        }
    }
}
```

Testing

LUIS Testing

Number	Test	Expected Outcome	Actual Outcome	Message Content	Notes
1	LUIS understands the 'version' intent and provides correct arguments (If any)	Pass	Pass	"Version"	
2	LUIS understands the 'version' intent and provides correct arguments (If any)	Pass	Pass	"What is your version"	
3	LUIS understands the 'version' intent and provides correct arguments (If any)	Pass	Pass	"What version number are you at?"	
4	LUIS understands the 'help' intent and provides correct arguments (If any)	Pass	Pass	"Help"	
5	LUIS understands the 'help' intent and provides correct arguments (If any)	Pass	Pass	"Help me"	
6	LUIS understands the 'help' intent and provides correct arguments (If any)	Pass	Pass	"What do you do?"	

7	LUIS understands the 'ping' intent and provides correct arguments (If any)	Pass	Pass	"ping connor-bell.com"	
8	LUIS understands the 'ping' intent and provides correct arguments (If any)	Pass	Pass	"ping the address at www.google.com"	
9	LUIS understands the 'ping' intent and provides correct arguments (If any)	Pass	Pass	"please ping the url at domainthatisdefinitelydown.com"	
10	LUIS understands the 'traceroute' intent and provides correct arguments (If any)	Pass	Fail	"traceroute connor-bell.com"	Incorrect argument - LUIS returned "bell.com" as the address argument
11	LUIS understands the 'traceroute' intent and provides correct arguments (If any)	Pass	Pass	"trace a rotue to the address www.google.com"	Note: Intentional misspelling
12	LUIS understands the 'traceroute' intent and provides correct arguments (If any)	Pass	Pass	"please trace a route to domainthatisdefinitelydown.com"	
13	LUIS understands the 'ssh' intent and provides correct arguments (If any)	Pass	Pass	"ssh floe.xyz"	
14	LUIS understands the 'ssh' intent and provides correct arguments (If any)	Pass	Pass	"open a tunnel to floe.xyz"	
15	LUIS understands the 'ssh' intent and provides correct arguments (If any)	Fail	Pass	"Start a live shell with floe.xyz"	This particular sentence was not trained into LUIS, but it managed

					to gather the intent and arguments correctly.
16	LUIS understands the 'idenitfy' intent and provides correct arguments (If any)	Pass		"identify"	
17	LUIS understands the 'identify' intent and provides correct arguments (If any)	Pass		"Who am I?"	
18	LUIS understands the 'identify' intent and provides correct arguments (If any)	Fail	Pass	"What is my user ID?"	This particular sentence was not trained into LUIS, but it managed to gather the intent correctly.
19	LUIS understands the 'logs' intent and provides correct arguments (If any)	Pass	Pass	"apache access logs floe"	
20	LUIS understands the 'logs' intent and provides correct arguments (If any)	Pass	Pass	"Get the auth logs for system from the floe server"	
21	LUIS understands the 'logs' intent and provides correct arguments (If any)	Pass	Pass	"get the apache error logs from floe"	
22	LUIS understands the 'associate' intent and provides correct arguments (If any)	Pass	Pass	"associate floeAuthLogs"	
23	LUIS understands the 'associate' intent and provides correct arguments (If any)	Pass	Pass	"Associate me with floeauthLogs"	

24	LUIS understands the 'associate' intent and provides correct arguments (If any)	Fail	Fail	"Associate me with the intermittent query floeauthlogs"	LUIS was not trained to understand 'intermittent query', so this failure is expected. The intent was correct but no argument was sent.
25	LUIS understands the 'none' intent and provides correct arguments (If any)	Pass	Pass	"awodnawioduhwaidih"	
26	LUIS understands the 'none' intent and provides correct arguments (If any)	Pass	Pass	"There is no intent in this sentence."	
27	LUIS understands the 'none' intent and provides correct arguments (If any)	Pass	Fail	In this message we sent a message with 35,000 characters	The application did not crash, but outputted many errors and responded with 8 POST failures as the message was split into several messages by the Telegram service.

Application Testing

Number	**Test**	**Expected Outcome**	**Actual Outcome**	**Message Content**	**Notes**

1	Application handles 'version' intent correctly	Pass	Pass	"Version"	
2	Application handles 'version' intent correctly	Pass	Pass	"What is your version"	
3	Application handles 'version' intent correctly	Pass	Pass	"What version number are you at?"	
4	Application handles 'help' intent correctly	Pass	Pass	"Help"	
5	Application handles 'help' intent correctly	Pass	Pass	"Help me"	
6	Application handles 'help' intent correctly	Pass	Pass	"What do you do?"	
7	Application handles 'ping' intent correctly	Pass	Pass	"ping connor-bell.com"	
8	Application handles 'ping' intent correctly	Pass	Pass	"ping the address at www.google.com"	
9	Application handles 'ping' intent correctly	Pass	Pass	"please ping the url at domainthatisdefinitelydown.com"	
10	Application handles 'traceroute' intent correctly	Pass	Fail	"traceroute connor-bell.com"	An incorrect argument from LUIS (only giving 'bell.com') means this test fails
11	Application handles 'traceroute' intent correctly	Pass	Pass	"trace a rotue to the address www.google.com"	
12	Application handles 'traceroute' intent correctly	Pass	Pass	"please trace a route to domainthatisdefinitelydown.com"	
13	Application handles 'ssh' intent correctly	Pass	Pass	"ssh floe.xyz"	

14	Application handles 'ssh' intent correctly	Pass	Pass	"open a tunnel to floe.xyz"	
15	Application handles 'ssh' intent correctly	Fail	Pass	"Start a live shell with floe.xyz"	See LUIS testing for expected fail.
16	Application handles 'ssh' intent with incorrect responses correctly	Fail	Fail	Incorrectly answered all questions	This was not checked for and so fails, as expected. The application does nothing and does not inform the user of any mistakes, and the next command to be sent is not parsed.
17	Application handles 'idenitfy' intent correctly	Pass	Pass	"identify"	
18	Application handles 'identify' intent correctly	Pass	Pass	"Who am I?"	
19	Application handles 'identify' intent correctly	Fail	Pass	"What is my user ID?"	See LUIS testing for expected fail.
20	Application handles 'logs' intent correctly	Pass	Pass	"apache access logs floe"	
21	Application handles 'logs' intent correctly	Pass	Pass	"Get the auth logs for system from the floe server"	
22	Application handles 'logs' intent correctly	Pass	Pass	"get the apache error logs from floe"	
23	Application handles 'logs' intent with incorrect arguments correctly	Pass	Pass	"get the test test logs from test"	Informs the user that their query does not exist and shows them what information was

					gathered from their message.
24	Application handles 'associate' intent correctly	Pass	Pass	"associate floeAuthLogs"	
25	Application handles 'associate' intent correctly	Pass	Pass	"Associate me with floeauthLogs"	
26	Application handles 'associate' intent correctly	Pass	Fail	"Associate me with the intermittent query floeauthlogs"	LUIS training failure means the argument is invalid.
27	Application handles 'associate' with incorrect query	Pass	Pass	"Associate me with nonexistent"	Informs the user the query does not exist.
28	Application handles 'none' intent correctly	Pass	Pass	"awodnawioduhwaidih"	
29	Application handles 'none' intent correctly	Pass	Pass	"There is no intent in this sentence."	
30	Application handles 'none' intent correctly	Pass	Fail	In this message we sent a message with 35,000 characters	The application did not crash, but outputted many errors and responded with 8 POST failures as the message was split into several messages by the Telegram service.

Commit Logs

commit 60e90731c40a6f4a6137f016e1ac6e41d84b667d

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Fri Apr 21 03:02:50 2017 +0100

Description: Final modifications to appendicies

commit 8f9248fe4daef164752840a555fb11497a7a1d61

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Thu Apr 20 10:09:19 2017 +0100

Description: Moving stuff around, updating todo

commit 98638f30d389c3515859c0f1914037586f3cda18

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Thu Apr 20 08:20:38 2017 +0100

Description: Minor changes to formatting

commit bf48835876773c6e1a7f948409b035136243c00d

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Thu Apr 20 03:31:58 2017 +0100

Description: Combined all into one document, first draft of critical analysis and some general TODO's and refs.

commit e6b65074e3049644170b85eb7de489b8bfccdc72

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Mon Apr 17 04:25:44 2017 +0100

Description: Implementation writeup refactor, fixed some comments

commit 6a077a49acd0d813bdd5d658f40277e0dcc696b1

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Sun Apr 16 04:34:25 2017 +0100

Description: Refactoring implementation writeup

commit 68a1228f80be337a3f069fd966a4fc4c87728100

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Sat Apr 15 04:46:33 2017 +0100

Description: Design refactored

commit 222e253fad1a97696e393eda6cc849fcd7979d62

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Sat Apr 15 01:31:49 2017 +0100

Description: Refactoring design, cleaned up some comments

commit a5f51a436941953ad46d018e549a503524ac7743

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Fri Apr 14 02:11:28 2017 +0100

Description: Improved help output

commit 88e61091bcd0230dec7e280680fff30416e7d183

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Fri Apr 14 01:51:15 2017 +0100

Description: Implementation done?

commit 6dcc69deb9780168d5c9e98c7b0bf2c4d68c9656

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Thu Apr 13 04:17:52 2017 +0100

Description: Basic implementationsn of some intents, unfinished SSH, unfinished refactoring of eval to proper module requires

commit ebb02b8dc778cc35cb72cf99857a958b584977c3

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Tue Apr 4 20:18:01 2017 +0100

Description: Added modularisation, added ping command and intent generation, updated documentation

commit 70d9cd4906c9a45170e6e63b1e68afb6534f6088

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Mon Mar 27 06:37:19 2017 +0100

Description: I uploaded my private keys. Good job me. Now I have to regenerate them.

commit 642eba2f4ed1fdcb05885d55c096314f558b1df5

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Mon Mar 27 06:32:50 2017 +0100

Description: I deleted the readme by mistake apparently

commit 07f51b494eb43599d41cddb8e7dc8c0d9e4c4488

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Thu Mar 23 05:04:29 2017 +0000

Description: Drafted first half of design docs, basic telegram/luis/botbuilder integration

commit fe715e6fe265b4cfdfb2a8a1b76baf0732b26b89

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Sun Mar 5 23:13:51 2017 +0000

Description: Problem spec draft 1

commit 1b8743b5555901cfafd13214e4f59c8376133742

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Mon Feb 27 16:48:28 2017 +0000

Description: Problem Analysis update

commit 7ec128a07c20b69cb1c9440b8c83edb7b8c4f3d5

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Mon Feb 6 00:21:10 2017 +0000

Description: Cleaned up and added a bit more to the analysis, barebones info for spec

commit 0b9bf1ad80356c76342f1ff0409550d361cceefb

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Thu Dec 15 04:29:32 2016 +0000

Description: Moved motivation, did 2/3 of the actual problem analysis hopefully

commit 553e9e50456c964a14419d9c9e3ea0fb4e58cd4f

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Mon Dec 12 02:34:25 2016 +0000

Description: Restructured lit reviews, did problem analysis intro and templated some more stuff

commit 46cd3bd2644862f3ba72f81efe4e743b7f153ba3

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Mon Nov 28 12:20:34 2016 +0000

Description: 2 lit reviews

commit db8757650213e23ae7b1a7a092fa74e631cd8520

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Mon Nov 21 03:56:13 2016 +0000

Description: Added 6 reviews, plus templates for 3/4 more

commit 8621f80fffbbe3c776f2eeb220cc6b047b158acb

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Mon Oct 31 04:19:27 2016 +0000

Description: Rewrote some lit reviews

commit fcb294499456e7e5c54e7bc78abc568c1c899e5f

Author: Brett Lempereur <b.lempereur@outlook.com>

Date: Mon Oct 24 11:16:23 2016 +0100

Description: Added comments and feedback

commit 600fd40ee4ffaeb4e20919ec5118e61842b1e87b

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Sun Oct 23 23:34:06 2016 +0100

Description: Two new lit reviews

commit 3ddd23d9d9a50a940b22d43c2e6d40b83007e164

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Mon Oct 17 04:02:56 2016 +0100

Description: Lit reviews 3-6 added, plus a doc version for easy editing and printing. MD verison not checked so may be badly formatted.

commit d62a2a0355dfbc811ccde8a85ad0405763e6d9c1

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Sun Oct 2 02:24:03 2016 +0100

Description: Second literature review added, updated some formatting

commit ae8194b72c6ff4819e3a99157fb87851ecbd7820

Author: ENIGMA\Connor <makeshift@itch.com>

Date: Sun Oct 2 01:03:33 2016 +0100

Description: Added meeting notes and first lit review for checking Monday

commit b368dca383d55dd3c399d7ed44c06425840bbbdc

Author: Makeshift <makeshift@itch.com>

Date: Sat Oct 1 23:54:39 2016 +0100

Description: Readme update

commit 244573afb3ea25788a50ced83e9127f30f0ffb28

Author: Makeshift <makeshift@itch.com>

Date: Sat Oct 1 23:47:53 2016 +0100

Description: Initial commit

Monthly Reports

November

LIVERPOOL JOHN MOORES UNIVERSITY

6000PROJ / 6001PROJ Final Year Project
Monthly Supervision Meeting Record

Student: Connor Bell Date: 31/10/2016

Main issues / Points of discussion / Progress made
- Reasonable progress on literature review. - Much improved writing :)
Actions for the next month
- Complete literature review - Start considering problem analysis.
Deliverables for next time
" "
Other comments

Supervisor signature: .....

Student signature: CBell

December

LIVERPOOL JOHN MOORES UNIVERSITY

6000PROJ / 6001PROJ Final Year Project
Monthly Supervision Meeting Record

Student: Conor Bell Date: 28-11-2016

Main issues / Points of discussion / Progress made
- Good progress on Lit. review - Need to start organising into single document.
Actions for the next month
- Completed problem analysis - Begin problem requirements
Deliverables for next time
" "
Other comments

Supervisor signature: .....

Student signature: CBell

January

LIVERPOOL JOHN MOORES UNIVERSITY

6000PROJ Final Year Project
Monthly Supervision Meeting Record

Student: Connor Bell Date: 1/2017

Main issues / Points of discussion / Progress made
Actions for the next month
- Complete problem requirements - Study on methodology / implementation
Deliverables for next time
Other comments

Supervisor signature: .....

Student signature: .....

February

LIVERPOOL JOHN MOORES UNIVERSITY

6000PROJ Final Year Project
Monthly Supervision Meeting Record

Student: Connor Bell Date: 2/2017

Main issues / Points of discussion / Progress made
- Lossing abit behind, need to step up pace of delivery, deadline looming!
Actions for the next month
- Complete problem requirements - First draft of methodology/implementation
Deliverables for next time
//
Other comments
—

Supervisor signature: .....

Student signature: .....

March

LIVERPOOL JOHN MOORES UNIVERSITY

6000PROJ / 6001PROJ Final Year Project
Monthly Supervision Meeting Record

Student: Connor Bell Date: 17/5/2017

Main issues / Points of discussion / Progress made
- Due to timing get drafts written for Friday 7th. - Quite behind on Implementation/analysis/conclusions.
Actions for the next month
Finish your Project :)
Deliverables for next time
A completed Project! :)
Other comments
NEVER GONNA GIVE... arbitrary marks.

Supervisor signature: .....

Student signature: CBell