# Intro

Based upon our previous section detailing 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 differed from the plan.

# Design

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 natural language processing service with Facebook’s Wit.ai would not require an entire system rewrite, but would only require the replacement of the Natural Language Processing module.

It is intended that the application be semi-autonomous in nature, 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 HTTP Server Access Logs** 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** show records of all error conditions reported by the server, which in some cases will need urgent attention.

**Authorization Logs** 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** 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** is also non-human-readable, and shows the last login of users.[1]

However, there is no reason why extra modules could 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 its connection is like.

**Traceroute** shows the user the route packets are taking to a server.

**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

Todo

# Technical Prototype Proposal

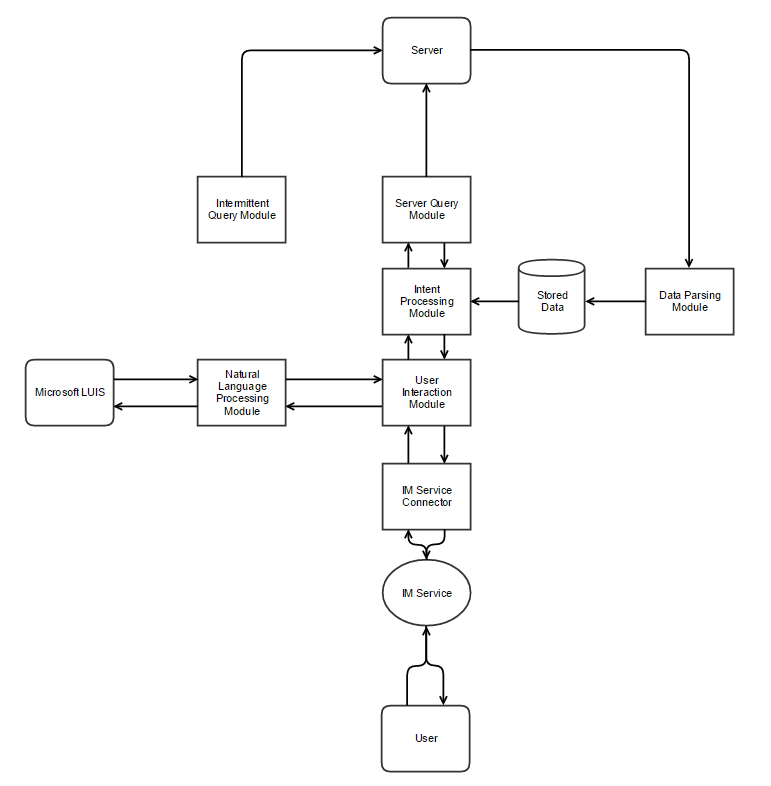


Figure 1 - Technical Prototype Flow Chart

## IM Service Connector

This module connects the User Interaction module to the IM service. This simply manages the connection to the instant messaging server, be it IRC, Jabber, Telegram or any other. As we already know the Microsoft Bot Builder[2] acts as a proxy between many IM services and our application, some of the work here is done for us. 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.

## User Interaction Module(UIM)

This module transports incoming information from the IM Service Connector to the Natural Language Processing module, and then 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 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.

## 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[3] running on an Azure instance.

At program startup, the NLP module queries the UIM 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 information the same way, then call an intent handler from the UIM.

## Intent Processing Module(IPM)

This module takes the ‘intent’ of the user from the NLP Module, plus any associated data, and gathers the information that was requested. This module pulls from stored data or live data as required through other modules. This module also manages thresholds and comparison of incoming data while monitoring servers.

The IPM only handles more complex requests such as server queries.

## Server Query Module

This module queries data from servers. (Incomplete)

## Intermittent Query Module

This module queries data from servers based on a timer, and is related primarily to the “monitor” mode. (Incomplete)

## Data Parsing Module

This module parses incoming data from servers into data more easily readable by other modules. This attempts to keep data consistent. (Incomplete, necessary?)

## Stored Data

This is a database containing data that may need to retrieved in future. It will be in JSON and stored on disk. However, this could be replaced with another system if needed for expansion, such as MySQL. (Incomplete, necessary?)

# Implementation

This section will show all the trouble I had building my application, such as NLP training and SSH not acting like I expected at all.

## Forwarding Ports

The first step in producing the application was met with the issue of forwarding ports. Because Bot Framework 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[4]. 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 continue our test deployment.

At this point, we have an internet-connected chat bot that we can test with the Bot Framework Emulator[5].

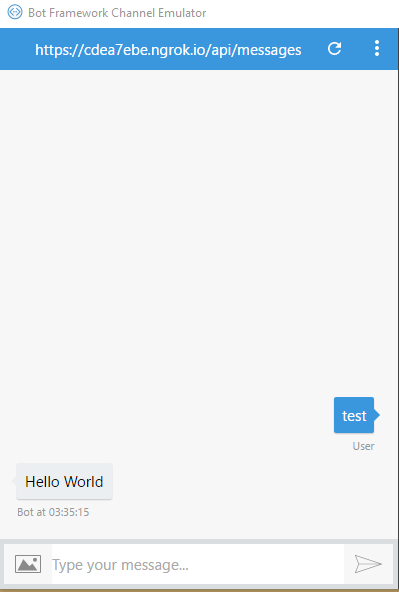


Figure 2 - Successful remote call and response

## IM Connection

Next we need to connect our application to an IM service. In this case Telegram, 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 user on Telegram and then entering it into Microsoft’s API.

We now have access to our bots’ commands from within Telegram.

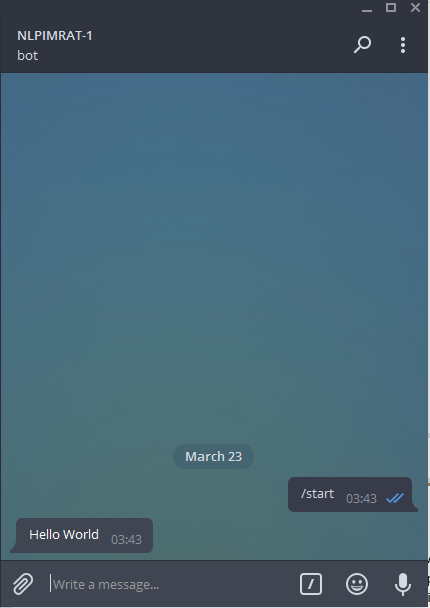


Figure 3 - Call and response from within Telegram

## 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 is done via their website, but when finished is hosted on an Azure 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. 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.

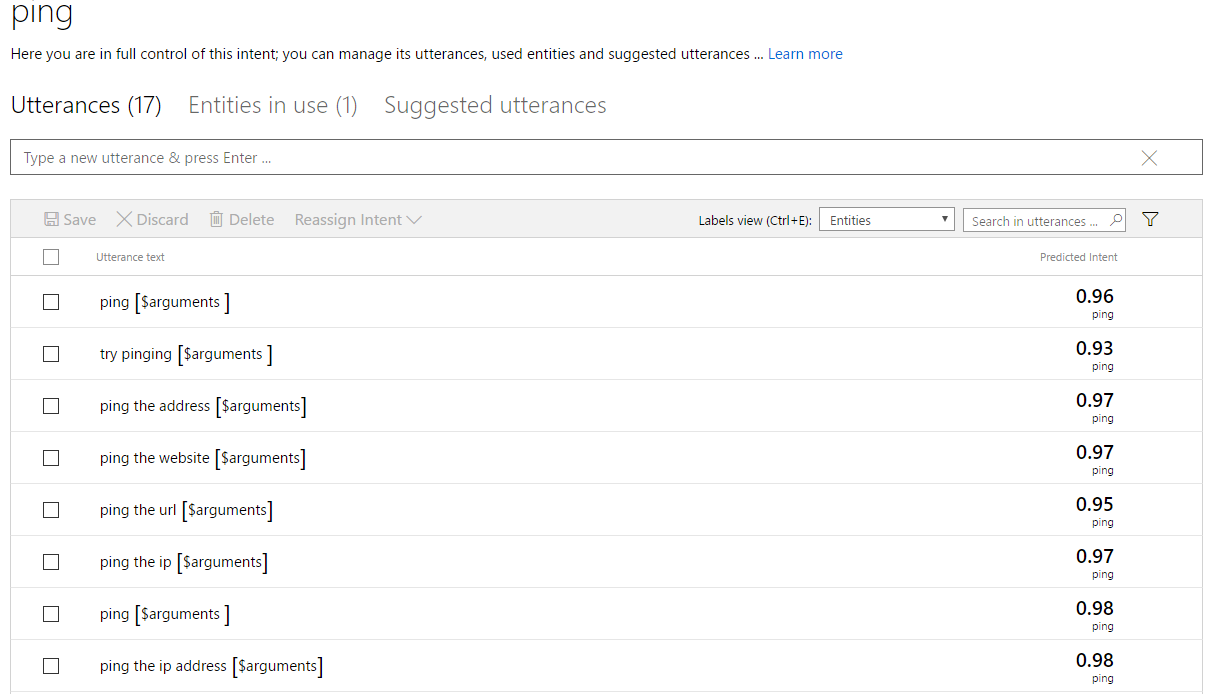


Figure 4 - Training the 'ping' intent

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.

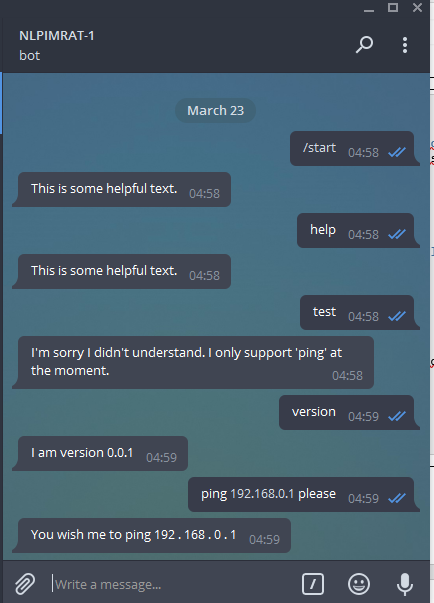


Figure 5 - Initial NLP training

After a successful conclusion to the NLP testing, point we started moving into a more modular and dynamic design. It is important that the NLP module be able to be replaced while leaving the intent processing intact, which is why the majority of the intent-processing capabilities of the module be generated by the UIM.

A simplified test version of the code can be seen in Figure 6 - User Interaction Module(UIM) example code. This code snippet shows a very simple example of the ‘ping’ command. We know that ‘ping’ is an intent that might be returned by the NLP module, so will respond with our formatted response of the data.

*//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 - NLP Module template generator

*//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 - User Interaction Module(UIM) example code

In the case of the NLP module, handling this data is slightly more complex, but infinitely expandable. As it generates all of handling templates and rules from the UIM, it can be replaced easily with an NLP module for another service, instead of using LUIS. Figure 6 - NLP Module template generator shows this templating engine in action, by generating the endpoints for all the intents from LUIS and formatting the data to a generic format to be managed by the UIM.

## Implementing Basic Commands

Commands such as ‘ping’ and ‘traceroute’ are very simple in theory, but can be incredibly powerful to a user who is not in the network. Our Intent Process Module (IPM) manages the more advanced intents that require interacting with other modules.

### 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’. 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.

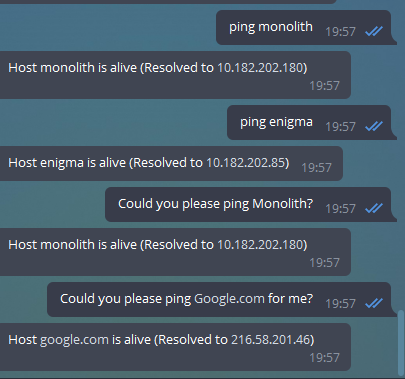


Figure 8 - Simple host PING example

### Traceroute

### NSLookup

### SSH

## Advanced Server Querying

### Log Handling

### Automation

## Alerts System

## Securing the Software

This section will show how I secured the software from use by unauthorised parties

## Testing

This section will show testing against general use.

# References

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

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

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

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

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