



UNIVERSITY OF AGDER

BACHELOR REPORT

Global Online Traffic Hacking and Monitoring

Authors:

Per-Arne ANDERSEN

Paul Richard LILLELAND

Supervisors:

Christian AUBY

Sigurd K. BRINCH

May 26, 2015

This page is left intentionally blank.

Abstract

In 2001 Strategy First released a game called Uplink. This game featured you as the hacker with the whole world as your potential target. By buying hardware, software and tools you were able to accomplish missions given to you by an anonymous host. This game briefly simulates a global infrastructure where you are able to establish proxy chains when connecting to different targets. Even though this does not give a detailed overview of the traffic, it suggests that the structural overview can be a good approach for making an educational application.

As the global digital infrastructure becomes larger and more complex, the necessity for management, maintenance and knowledge becomes greater. The vastness and routines of these massive networks can be hard to understand by words.

The human brain has a better ability to understand structured information rather than pure logic and partial information. This makes it important for a student to see the structure as complete as possible and hopefully connect the pieces needed to get a good understanding of the subject.

This project is intended to solve this issue making it easy for students to see the connections and structure of the world infrastructure as a whole. The application features variable global traffic load, packets with header information, personal hosts and networks, an emulated Linux like OS, missions, Traffic-path illustration, live representation of DDoS attacks and more. With an open-source application and framework, the project is built for easy extensibility.

Preface

GOTHAM is a bachelor project in Computer Engineering by Per-Arne Andersen and Paul R. Lilleland at the University of Agder. The project assignment is of the group's own design approved by the faculty. The project use GIT as versioning control and JIRA for work logging and issue tracking. These services are provided by the university at tools.uia.no

The supervisors Christian Auby and Sigurd Brinch have previously lectured and supervised the members of the group, both in programming and ethical hacking.

The game is found at: <http://www.gotham.no>

Grimstad May 2015 Paul R. Lilleland and Per-Arne Andersen

Contents

1	Introduction	1
1.1	Background	1
1.2	Problem Statement	2
1.3	Priority Overview	3
1.4	Priority Specifications	4
1.5	Expected Results	11
1.6	Source Review	13
1.7	Problem Solution	14
2	Application Design	21
2.1	Main Menu	21
2.2	Settings Menu	22
2.3	Routing	23
2.4	World Map	24
2.5	Missions	25
2.6	Terminal	26
2.7	Software	27
2.8	Hardware	28
2.9	Help Central	29
2.10	Pedagogical Aspects	29
3	Technical Design	30
3.1	Language Choice	31
3.2	Structural Design	32
3.3	Database Design	36
3.4	Network Communication	37
3.5	Limitations	38
4	Implementation	39
4.1	Game Framework	39
4.2	Front-end Application	45
4.3	Backend Server	48
4.4	Data Management	53
4.5	World Map Generation	56
4.6	Pathfinding	60

5	Documentation And Testing	61
5.1	Coffee-Script	62
5.2	C#	62
5.3	Test Overview	64
6	Results	65
6.1	Priority A	65
6.2	Priority B	66
6.3	Priority C	68
6.4	Priority D	69
6.5	Summary	70
6.6	Future Work	70
7	Group Evaluation	73
7.1	Discussion	73
7.2	Conclusion	74
A	Appendix	A1
A.1	Code standard list	A1
A.2	Project Architecture	A2
A.3	Traffic structure diagram	A3
A.4	Database structure	A4
A.5	Python Scripts	A5
A.6	Group contract	A6
A.7	License	A7
A.8	Explanations	A8
A.9	Gantt	A9
A.10	Team Member Contribution	A10
A.11	Press Release	A11
A.12	Test Sheet	A12
A.13	Time Logs	A12
A.14	Issue List	A12
A.15	Meeting Logs	A12
A.16	Pre Project Report	A12
A.17	Project Description	A12

List of Figures

1	Average network load	9
2	Main Menu	21
3	Settings Menu	22
4	Routing Path Example	23
5	World Map	24
6	Mission Window	25
7	Terminal	26
8	GothShark Window	27
9	Hardware Store	28
10	Help Central	29
11	Namespace hierarchy defined in the Gotham back-end.	32
12	Front-End Structure	33
13	Game Framework Structure.	34
14	Debug clients view.	35
15	Table structure of the back-end database.	36
16	Traffic Flow Between Modules	37
17	Front-End Game Loop	44
18	Networking - Room Structure	50
19	Networking - Emit.	50
20	Debug Client	54
21	The process of sterilizing the image	57
22	Beginning of a flood fill process	58
23	A finished flood fill process	58
24	Example of loops in a polygon	59

List of Tables

1	Application specifications	30
2	Tween.cs benchmark comparison	43
3	Mission Requirement example	46
4	Parsing a requirement string to a external ip	53
5	List of all unit-tests created for this project	64

1 Introduction

1.1 Background

The idea for this project came from *Uplink*, where you take on the role as a blackhat hacker. The game featured criminal missions like altering social security statuses, or penetrating advanced network technology. This sparked the idea to develop a similar application based on new ideas.

This idea initially started at a course called *Advanced Programming* at the University of Agder, but was left dead after realizing that the project would be too large for the course. Planning the bachelor project the, hacking idea was revisited and after some discussion and planning the project was settled. The beginning of a *Global Online Traffic Hacking And Monitoring* application was made with the project codename GOTHAM.

The goal of the project was to adopt features from *Uplink* and develop a realistic simulation with the same user friendly interface. To this date, education in this field have done been by books and illustrations of theoretical events. The goal of GOTHAM is to set a perspective on these network events and help students understand how the Internet works in a gamified setting.

There are different reasons why this task was chosen. Primarily due to the interest from the group participants. The task requires a good amount of coding, the development and implementation of different algorithms and a good amount of background information on how the global digital infrastructure works. Due to the fact that this is a learning tool, the gamification on top of the simulation hopefully will make the application more compelling.

1.2 Problem Statement

Providing students an application making it easier to understand how the Internet infrastructure and the different layers of the of the OSI model works together.

To achieve this, an application is to be developed to present the relevant details, structure and context of the global infrastructure. This is initially intended for students learning how traffic, packets and the Internet work.

The emulated traffic should be affected by different high-load variables such as streaming, file transfer and other protocols. To be able to simulate this amount of traffic without a supercomputer, the traffic between the nodes should be generated by a server using statistical data if possible.

In order to generate detailed traffic information for the layers in the OSI model, a traffic session is queried and initialized when requested. This session should exist between two hosts. With the stream open, the student will be able to see the traffics physical path through the world, along with the packet headers at the different layers.

Traffic paths should be generated from algorithms like A-Star, Dijkstra or Ant Colony Optimization. The application should draw the path and display some information about the different algorithms and how they work.

The front-end should feature a dashboard with a world map for a good overview with necessary tools and functionality available from a side menu. For the application to be more than just a simulation, a gamification is planned to feature objectives and tools to manipulate data. This would be presented in the form of a hacking game. Every network operator should be aware of the risk of an attack. This will be featured by some common security issues such as old version exploits or weak passwords. This will hopefully give an insight of the importance keeping a system as impenetrable as possible.

1.3 Priority Overview

The goals in this project are prioritized in the following order.

A is the vital features of the project

B is important features of the project

C is secondary features that is planned for implementation.

D is secondary features that are not directly planned for the first version.

These goals can however be implemented if there is time, or they are implemented as a dependency of higher priority goal. D goals contains additional details as they are subject for future work.

A - Internet Emulation

- Node and Cable Generation
- Traffic Simulation
- Front-End
- Realistic Data (static)

B - Detailed Simulation

- Realistic Data (dynamic)
- Real ISP Names
- Algorithms
- Protocols and Services
- Missions
- Money, Hardware and Software
- Internet site

C - Interaction with Traffic and Gamification

- Creative Mode
- Packet header emulation
- Hacking OS

D - Alternative goals

- Advanced Missions
- Rivals
- Dynamic Traffic
- More Detailed Traffic
- Botnets

1.4 Priority Specifications

1.4.1 Priority A

Node and Cable Generation The application require information about nodes and cables located around the world. The primary source for this information should be from a realistic provider. Should this prove difficult, algorithms should be developed to generate the required data.

Traffic Simulation The simulation of traffic is an important feature of the project. Like in the real world infrastructure, traffic should exist between network nodes at all times. This requires a traffic engine to process and update the load of nodes and cables. If possible there should be an individual load for common protocols like HTTP, Torrent or RTSP.

The traffic simulation exists of two types of traffic, namely *Micro and Macro* traffic. Micro traffic is traffic defined on a packet level while Macro traffic is statistical data. These are explained better in A.8

When traffic is generated, the packet information should, if possible exist on each node in the path. The packet data will remain the same but as in reality, header information will change during the jumps from the different nodes.

Front-End Front-end should manage data served by the back-end application. The data should be presented in a way that it's easy to understand, having a user-friendly interface.

Its purpose is to present a world map containing all nodes and cables loaded from the back-end server. The user should be able to see how the traffic flows on the global infrastructure and be able to read packet information on the different nodes.

The front-end should also feature a menu system for *Networks and Identities*, *Terminal*, *Missions* and *Hardware*. The interaction with these menus should trigger events through communication with the back-end server, for example accepting a mission.

Realistic Data (static) Attempts to gather real data for the simulation in order to achieve the most realistic simulation possible. This information include:

- Node information
- Cable information
- Fake identities
- ISP Names

Should some of the information be difficult to obtain, it will be generated.

1.4.2 Priority B

Realistic Data (dynamic) Statistic of the real world traffic should be included in the calculations if possible. This may be problematic if no consistent source of information is provided.

Multiple sites provides live DDOS maps which may prove useful. This can be used to display and simulate real world DDOS attacks in the application.

Real ISP Names Use of real world ISP names should be implemented if possible. These would be used to name the nodes.

Algorithms The plan is to implement Dijkstra's algorithm, A-Star and if possible Ant Colony Optimization for path-finding. These algorithms should be presented and compared, describing how one each work.

The generation of traffic statistics (Macro traffic) should be done by the back-end. Using as much real data as possible generating traffic between the different countries and nodes.

Protocols and Services A session should be created when establishing a connection between two hosts. Traffic in the session is defined as packets. These packets contains a structure of protocols and layers from the OSI Model.

This should feature the most used protocols like HTTP, FTP, SSH and similar. By structuring the different layers individually, presentation and manipulation of the packet should be trivial.

Missions To be able to present the features of the application in a good way, a mission system will be implemented generating random missions. These missions require the usage of tools like ping, trace-route and packet sniffing.

The player will be awarded with experience points to unlock additional missions after completion. If possible, awarding currency will also be implemented and used to purchase hardware and software.

Money, Hardware and Software The plan is to implement a currency system and a shop for purchasing hardware and software. Money is earned by completing missions.

Internet site A website under the domain `http://gotham.no` will be created to serve the application. Additionally, source code and documentation should be included.

1.4.3 Priority C

Creative Mode A god mode, where the user is able to for example cut a cable and see how the traffic is redirected. As there are already a game aspect of the application, this goal mainly features actions not requiring any prerequisites.

Packet header Simulation By expanding Micro traffic packets, the presentation of the package headers could be added to the already existing packet.

As some headers are changed when jumping between nodes, different data must be presented on each node in a path.

Examples of header information is IP, MAC, port, ICMP type, versions or encryption.

Hacking OS Each of the hosts should feature an operating system. The operating system should be a UNIX-like environment with basic command line functionality.

The user will be able to use ping, trace-route and other tools. This directly impacts the missions and world map, displaying for example the path of a trace-route.

1.4.4 Priority D

Advanced Missions The mission system should already be flexible and able to generate random missions with multiple requirements. Further development of the mission systems should include multilevel mission chains. These chains are requirements chained together in a way that they have to be done in sequence to be successfully completed.

Example - Mission Sequence

1. Ping a host at 192.168.1.10
2. Attempt to SSH login to 192.168.1.10
3. Brute Force 192.168.1.10
4. Steal a file via SFTP

Each of these would have to be done in the specific order they are described, or the requirement would not be completed. In order to implement this, the system would have to be expanded in the Mission and Requirement classes. Additionally the requirement table would require a self referencing parent column to support the chain.

Rivals The game should already feature a basic multiplayer as all users are able to see the generated traffic. The planned requirements for this priority is:

- Being able to interact with other users of the simulation
- CO-OP Missions. These would typically be simple missions joint DDOS operations and hacking assignments
- World Authority, NSA. When a player is influenced in the simulated world, he would be subject to be arrested by the World Authorities.
- A functional communication platform between players.

Dynamic Traffic An algorithm to control the network load based on time of the day. This would increase the realism of macro traffic flow. This is a step towards implementing a AI to simulate additional micro traffic.

There are multiple metrics and statistics regarding time and date based network load. Research suggests that most traffic are statistically equal based on time of the day. This is proven by looking at the different types of traffic used throughout the average day.

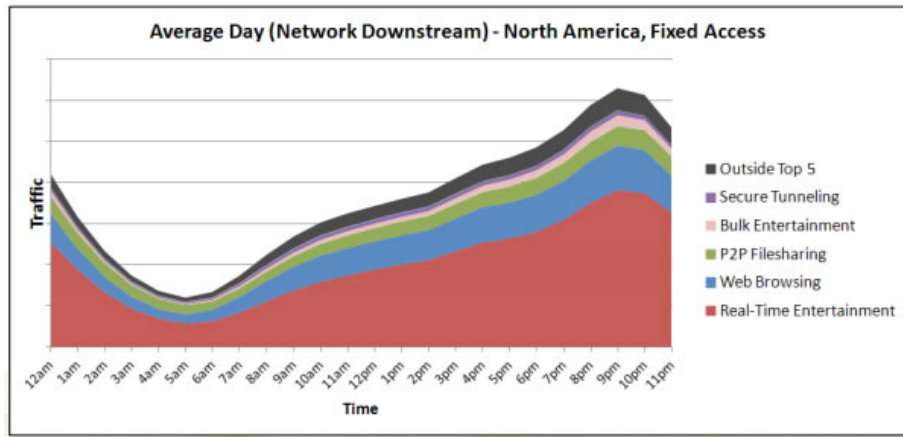


Figure 1: Average network load

By using this data, a relatively accurate network load simulation can be made.

More Detailed Traffic Implement a feature to alter micro traffic between nodes. This should primarily be altering of packet headers between the nodes. Example of this is the MAC address change between each hop in a ICMP request.

Additional information like sequencing of the packets should also be possible with a flexible framework.

Botnets A botnet is by definition a collection of computers. This would be possible to do by creating a new engine for pairing up hosts to a cluster. This cluster would sum up to a single host with increased processing power. The botnet should have enough functionality to:

- Execute a DDOS attack against any host or network
- Run coordinated brute force attacks against any host
- Be able to drop and add new hosts to the botnet
- Have a generic interface to be able to manually communicate with each of the member hosts of the botnet

As an addition, botnet traffic should be reflected as micro traffic.

1.5 Expected Results

1.5.1 Features

The project is expected to have all primary priorities implemented before the project end. Some features is expected to be implemented as a bare minimum because of the complexity involved in creating a generic application interface. Example of this is the Mission system, which should be as abstract as possible, in order to feature more complex tasks.

The project is expected to have some kind of validation, namely unit tests. It is not expected that the project will be written by BDD¹ or TDD² principles.

1.5.2 Priorities

In terms of priorities, it is expected that *A* and *B* priorities are fully implemented. These priorities should not offer any problems that require changes. Implementation of *C* priorities may start whenever *A* and *B* is considered partially done.

C priorities and below are not required for the application to function properly, but is important for gameplay and gamification.

D priorities are not expected to reach v1.0. The priority list is structured so that these implementations are not important to for the project quality. The plan is to include *D* priorities as future work if not completed in the project period.

Features not fit for production by the project end should be deactivated or removed.

¹Behaviour driven development

²Test driven development

1.5.3 Summary

By the end of the project period, GOTHAM is expected to be a fully working network emulator. This is defined by the priorities set for this project period. Even though the project is a proof of concept, it should be possible to interact with all complete features.

Work done is expected to be logged and referenced in JIRA, which makes it possible to backtrace the progress of this project period. The group expects the final result to feature possibilities for further extensions, namely the D priorities.

Some features may have limited functionality due to high complexity. These features should not be required for proving the concept of this project. It is not expected that the product is error free and thoroughly tested due to the same reasons.

It is expected that as much realistic data as possible is used. As there are limitations, the features requiring unavailable data will have it generated by algorithms.

1.6 Source Review

1.6.1 Stack overflow

Stack overflow features questions and answers on a wide range of topics in computer programming. This offers a vast library of problems with ranked solutions. [1]

1.6.2 GoodBoyDigital

GoodBoyDigital is the home of pixiJS, the rendering library for the game framework. This is mainly used for documentation and examples. PixiJS is a relatively low-level framework, and is mostly used by other libraries to build game frameworks. [2]

1.6.3 Microsoft

Microsoft provides the documentation for all Microsoft products used. These products include for instance Visual Studio, IIS, .NET and ASP.NET [3]

1.6.4 HTML5 GAME DEVS

A forum focusing on the development of games in HTML5 with focus on several JavaScript graphic libraries. Featuring news, discussions and problem solutions. [4]

1.6.5 Official Documentation

nHibernate - <http://nhibernate.info> [5]

Fluent NHibernate - <https://github.com/jagregory/fluent-nhibernate> [6]

Loki.js - <http://lokijs.org/docs> [7]

TaffyDB - <http://www.taffydb.com/workingwithdata> [8]

Sequelize - <http://docs.sequelizejs.com/en/latest> [9]

Howler.js - <https://github.com/goldfire/howler.js> [10]

Socket.IO - <http://socket.io/docs> [11]

jQuery - <https://api.jquery.com> [12]

1.7 Problem Solution

By making a structured list of issues in Jira and planning the work stages of front and back-end, the group should have a general development plan to follow. Dependencies should be prioritized and issue handling adaptive to the current state of the project to prevent stagnation.

1.7.1 Project Strategy

Division of Labor As the group only consists of two members, no detailed division of labour seems necessary as the members mostly is in contact while working. If a member have problems finding a good solution, both members often cooperate on the same feature to get a more refined result.

The project is split into three parts, back-end, front-end and game framework. Paul Richard Lilleland will be in charge of the back-end server and Per-Arne Andersen of the front-end and game-framework.

Each member of the team is responsible that his part of the code follows code standards and is functional.

Meetings Each Tuesday at 13:00 the group meets with the main supervisor to give an update on the project and to get feedback on their work. Two additional supervisors have agreed to assist the group, each having different areas of expertise relevant to the project. Meeting with the other supervisors is scheduled if needed.

The group are responsible for documenting each meeting with a simple report.

Code standard The group members have agreed on a list of code standards to keep the code structured and consistent throughout the project period.

A list of requirements may be found in the appendix

This is generally what's described in the C# coding convention guide from Microsoft [13]

1.7.2 General Approach

The main approach for back-end development was using C# as this features a consistent and strongly typed code structure. Difficulties occurred when implementing layers from the OSI-model. Using a typed language proved to be problematic, and migrating to a weak typed language seemed necessary. This resulted in a migration to Node.JS as the back-end server.

As most back-end features do not require the front-end client for testing, the back-end development usually is done in the console. Graphics is a large part of the front-end development. Therefore the dependency of the back-end is reduced, making it ideal for development of both modules simultaneously. Tasks, structure and execution are being discussed by the team and then implemented by the assigned member. A review phase then starts when implementation is finished to ensure code standard consistency and quality.

Developing will start by creating a client console and server software, which should be able to communicate with each other. When communication is operable, testing and adapting the modules (back-end/front-end) will start.

Code is shared between the developers through GIT. The code is also available on Dropbox for quick look-up and continuous change if required. This would typically be static web content on the other members server.

This is mainly how the group will proceed to complete this project and experience shows that this method have been productive for the group consisting of the current members.

1.7.3 Phases

The estimated time requirement for each phase is described in the Gantt diagram in the Appendix. This chapter features a short description for each phase.

Research This is the research and prototyping phase where programming languages, libraries and tools are determined. Some of these are determined without the need for prototyping but others like front-end, graphics, communication library requires testing to see if they meet the requirements. This includes licensing.

Development The project consists of two primary modules, front-end and back-end. Starting with back-end, a database of nodes, cables, persons and other data is generated with the possibility to communicate with the front-end. The front-end consists of a structured overview, menus, data presentation and graphical elements.

Improvements After the main features are implemented and working, the phase of improvement and polishing starts. This phase depends on the time required to implement the primary goals. Elements of this phase is typically algorithms, process and graphical optimisation.

Report and Presentation When the project is close to its end the group will start writing a report which is due 27th of May 2015. This should include research and documentation of the work done in the project. A group will then have a presentation of the project at the 4th of June 2015.

1.7.4 Tools and Software

Development is done in Visual Studio and IntelliJ. For database management, an application called Navicat is used. Debugging is done in Google Chrome for the front-end client. The back-end uses the self made debug client in C#. Node.js serves as the back-end server and MariaDB as the projects database. A more detailed list of the projects tools and dependencies follows.

Experience shows that the use of CoffeeScript and PIXI as the Front-End would be a good choice. The use of C# as the Back-End have proved useful in the past but did not suffice for this project as described later in the report.

The architecture of the application and how it uses the various tools and libraries is illustrated in the appendix A.2

Development:

Visual Studio 2013 is the main leading IDE for development in .NET. This is officially supported by Microsoft and features the required tools for the development of this project.

IntelliJ IDEA is a Java integrated development environment (IDE) for developing computer software. For this project it is mainly used for developing the front-end and back-end software.

Navicat Premium is a database management tool that supports numerous types of databases with a user friendly interface.

Source Tree is Atlassian's Git management tool with an user friendly interface. The alternative would be Git Bash but since all group members are familiar with Source Tree, this is probably a more effective tool.

Chrome is one of the leading Internet browsers. This features a great tool for debugging javascript code which is used in the front-end.

JIRA features issue and time tracking amongst other productivity related tools. Using the kanban module for managing the project.

Stash is Atlassian's source code management and version control tool for Git.

Services:

MariaDB is a database management system (DBMS). This is a relational based database used to structure all data of the project.

nginx is a fairly new web server which is excellent for fast and static content with low resource requirement.

Google Maps is the leading global map providing service with an excellent API for gathering geo information.

Node.js “is a platform built on Chrome’s JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.”

LucidChart is an online diagram creator that provides sharing functionality with a user friendly interface.

Dropbox is a world leading cloud storage provider with file sharing features. It is used for data not contained in Git, like reports, diagrams and similar.

Libraries:

nHibernate is the .NET equivalent of Hibernate which is a library for database management in Java. nHibernate features great support for communication and database mapping.

Fluent NHibernate “XML-less, compile safe, automated, convention-based mappings for NHibernate”

Pixi.js is a low level, high performance graphics library for javascript.

log4net features a framework for structured logging in a .NET application.

LokiJS “In-memory Javascript Datastore with Persistence”. Servers as a client database cache.

Sequelize is a Node.js ORM for numerous types of databases.

Socket.IO “enables real-time bidirectional event-based communication.”

When.js is a promises implementation used to control the asynchronous code execution of javascript.

2 Application Design

There has been a high focus on the design of the application. Attempting to make the design as digital as possible, still being simplistic and easy to understand.

2.1 Main Menu

The main menu will appear after the loading screen, which is the first interactive window. This consist of a animated background with a set of menu buttons, *Play Game*, *Settings*, *About* and *Exit*.



Figure 2: Main Menu

When the game is loading, the progress is displayed as floating elements.. There are multiple background images on the main menu, rotating after a set duration.

The *Play Game* button starts the game, logging the player into the world map. Currently there is no way of logging in manually, but this is documented under future work.

The *Settings* button displays the settings menu, explained below.

Credits shows all contributed parties in the project, either it is developer, supervisor or a third party vendor.

2.2 Settings Menu

The settings panel currently only features sound configuration. The original plan was to also support resolution settings, but this is automatically handled by an implementation in the renderer.

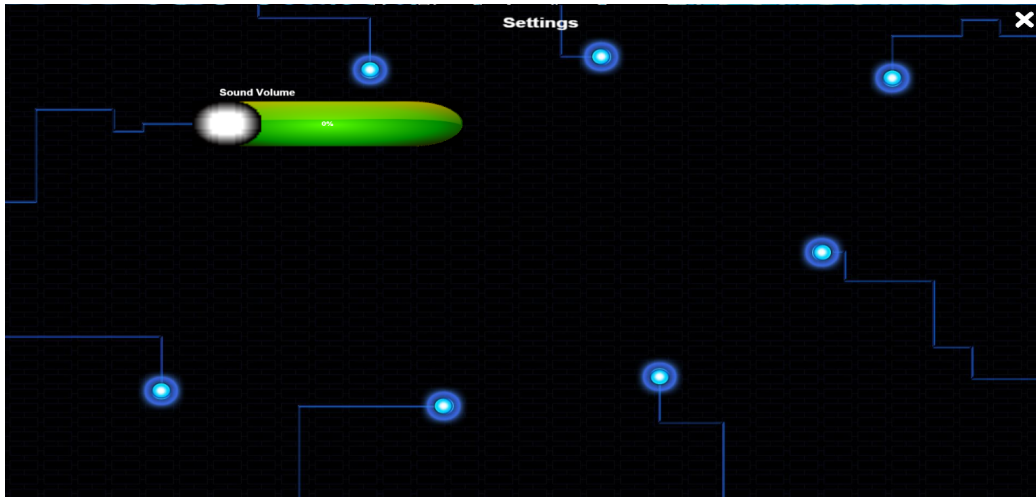


Figure 3: Settings Menu

The panel itself is very basic, but it has capabilities of adding additional settings if required.

2.3 Routing

Traffic routing is visualized by creating a path between all affected nodes. When a user trace-routes a target, lines with a animation are drawn inside the path which represents the direction.

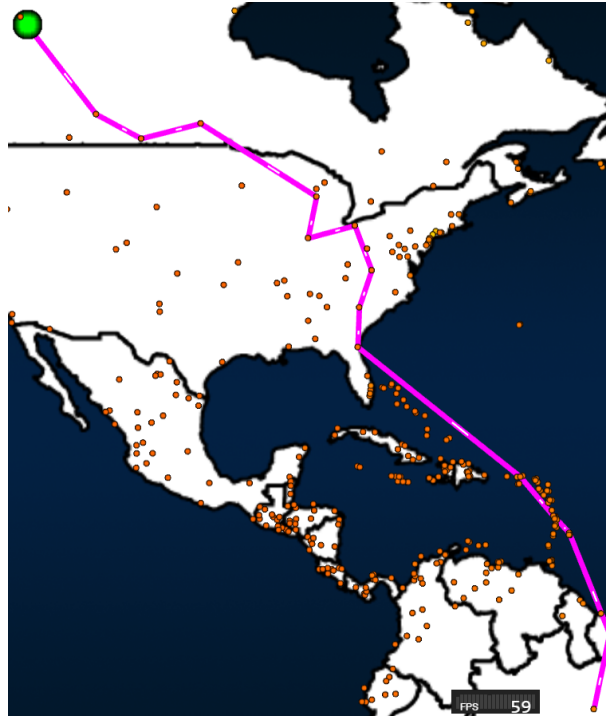


Figure 4: Routing Path Example

The path can be modified by right clicking, ultimately blacklisting that node from path-finding

2.4 World Map

The overview is the main view of the game, consisting of a world map and menu bars on the left and right. These menus are namely *Host and Networks*, *Identities*, *Missions*, *Help*, *Settings*, *Main Menu* on the left side and *Quick Action Tools* on the right.

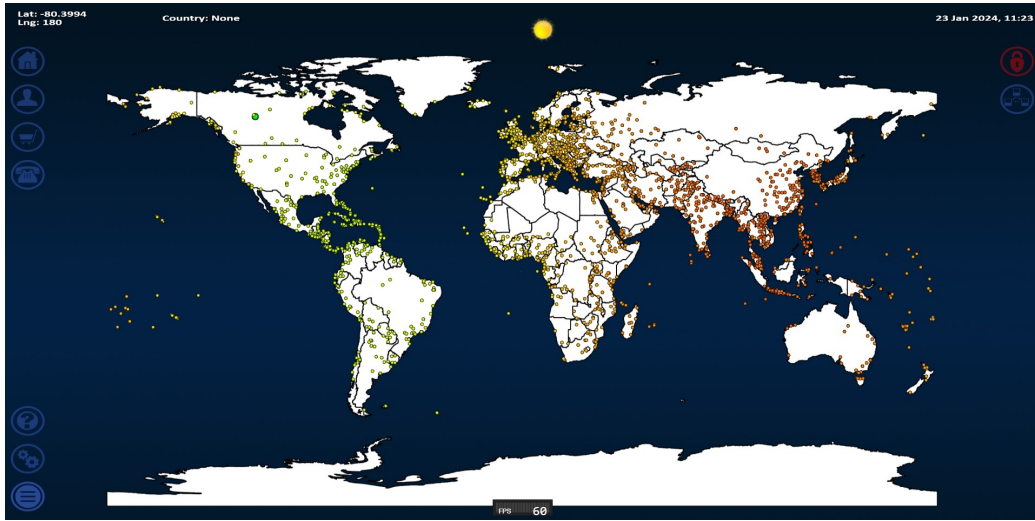


Figure 5: World Map

When hovering a country or any position on the map, current latitude, longitude and country is displayed in the upper left corner. *Cables* and *Nodes* are drawn onto the map, whereas cables are by default hidden. Hovering a node displays its connected cables while all cables can be enabled by activating it on the *Quick Action Toolbar*.

The user can deactivate nodes by right clicking it. This is useful for when the user wants to experiment with the path-finding algorithm.

Networks are displayed as a green dot on the map. These are not intractable, but can be managed from the *Host and Network* menu.

The sun represents current time throughout the world. The sun's position is always 12:00 pm in the current longitude. Node loads are affected by time as seen on the map. Load is displayed as node color, where green is the least load and red is the highest possible load.

2.5 Missions

The mission frame is split into three segments. *Available missions*, *Ongoing missions* and *Mission description*. When selecting an available mission, the description will be displayed allowing to accept the mission if the experience requirements are met.

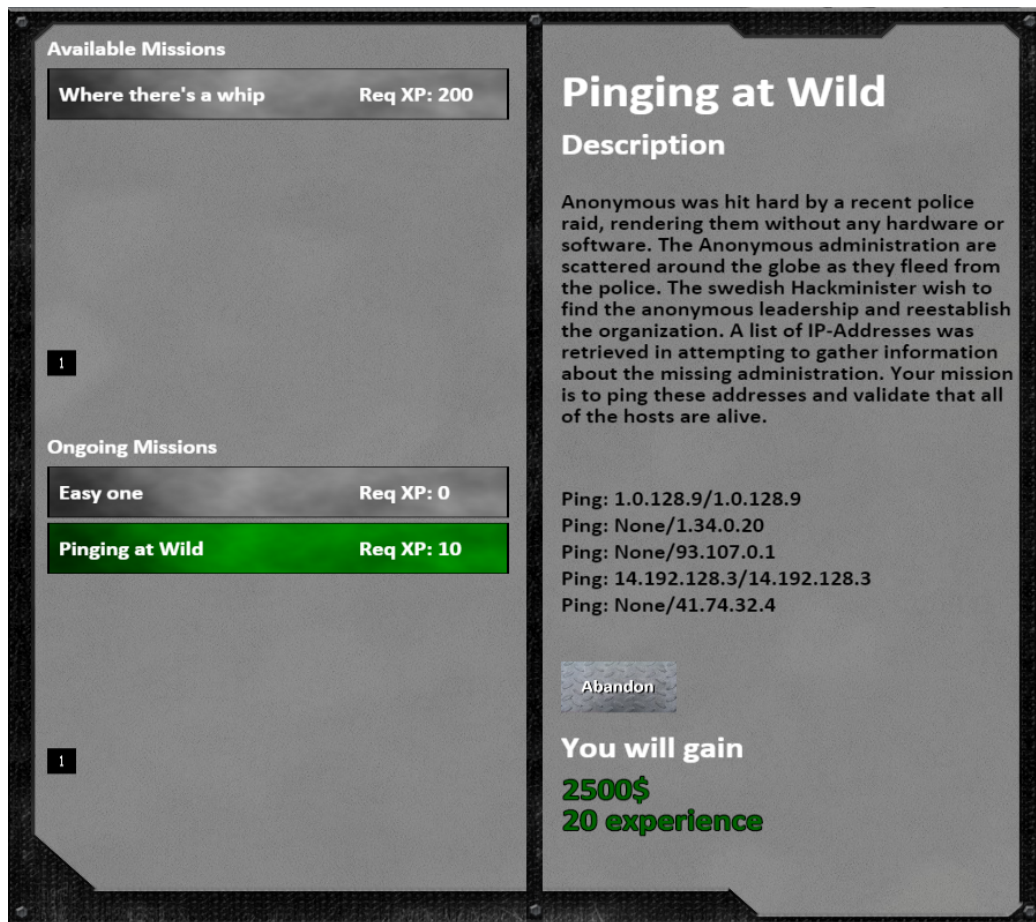
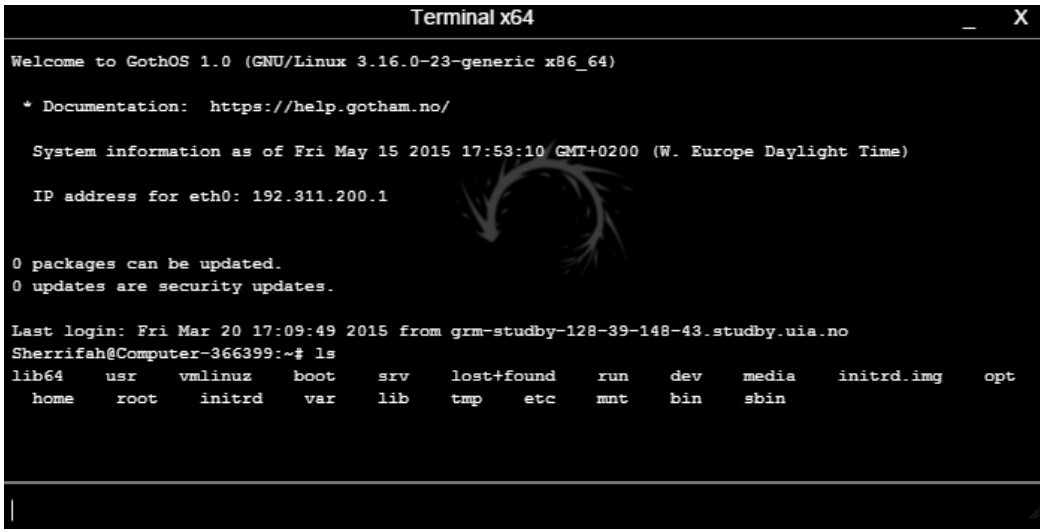


Figure 6: Mission Window

The mission can be abandoned at any time. Mission description features a background story followed by the requirements to complete the mission. When all requirements are completed, the mission can be completed and the user will gain money and experience.

2.6 Terminal

The terminal is opened by clicking on one of the hosts in the *Hosts and Networks* menu. The design is based on Ubuntu Linux and accepts input commands like ping and trace-route. Commands in the terminal supports input parameters.



```
Terminal x64
Welcome to GothOS 1.0 (GNU/Linux 3.16.0-23-generic x86_64)

* Documentation:  https://help.gotham.no/

System information as of Fri May 15 2015 17:53:10 GMT+0200 (W. Europe Daylight Time)

IP address for eth0: 192.311.200.1

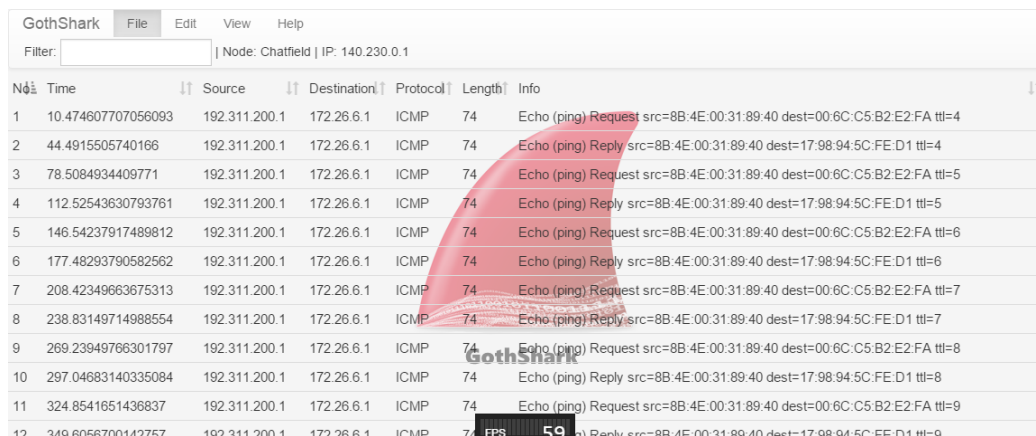
0 packages can be updated.
0 updates are security updates.

Last login: Fri Mar 20 17:09:49 2015 from grm-studby-128-39-148-43.studby.uia.no
Sherrifah@Computer-366399:~$ ls
lib64  usr  vmlinuz  boot  srv  lost+found  run  dev  media  initrd.img  opt
home  root  initrd  var  lib  tmp  etc  mnt  bin  sbin
```

Figure 7: Terminal

2.7 Software

An essential part of the game is software. This features tools for system penetration, monitoring and other helpful applications. In the first version of the application the software will consist of ping, trace-route and a basic sniffing tool. The traffic generated will exist on the nodes the traffic has traversed through. This could be for instance ICMP as shown below.



No.	Time	Source	Destination	Protocol	Length	Info
1	10.474607707056093	192.311.200.1	172.26.6.1	ICMP	74	Echo (ping) Request src=8B:4E:00:31:89:40 dest=00:6C:C5:B2:E2:FA ttl=4
2	44.4915505740166	192.311.200.1	172.26.6.1	ICMP	74	Echo (ping) Reply src=8B:4E:00:31:89:40 dest=17:98:94:5C:FE:D1 ttl=4
3	78.5084934409771	192.311.200.1	172.26.6.1	ICMP	74	Echo (ping) Request src=8B:4E:00:31:89:40 dest=00:6C:C5:B2:E2:FA ttl=5
4	112.52543630793761	192.311.200.1	172.26.6.1	ICMP	74	Echo (ping) Reply src=8B:4E:00:31:89:40 dest=17:98:94:5C:FE:D1 ttl=5
5	146.54237917489812	192.311.200.1	172.26.6.1	ICMP	74	Echo (ping) Request src=8B:4E:00:31:89:40 dest=00:6C:C5:B2:E2:FA ttl=6
6	177.48293790582562	192.311.200.1	172.26.6.1	ICMP	74	Echo (ping) Reply src=8B:4E:00:31:89:40 dest=17:98:94:5C:FE:D1 ttl=6
7	208.42349663675313	192.311.200.1	172.26.6.1	ICMP	74	Echo (ping) Request src=8B:4E:00:31:89:40 dest=00:6C:C5:B2:E2:FA ttl=7
8	238.83149714988554	192.311.200.1	172.26.6.1	ICMP	74	Echo (ping) Reply src=8B:4E:00:31:89:40 dest=17:98:94:5C:FE:D1 ttl=7
9	269.23949766301797	192.311.200.1	172.26.6.1	ICMP	74	Echo (ping) Request src=8B:4E:00:31:89:40 dest=00:6C:C5:B2:E2:FA ttl=8
10	297.04683140335084	192.311.200.1	172.26.6.1	ICMP	74	Echo (ping) Reply src=8B:4E:00:31:89:40 dest=17:98:94:5C:FE:D1 ttl=8
11	324.8541651436837	192.311.200.1	172.26.6.1	ICMP	74	Echo (ping) Request src=8B:4E:00:31:89:40 dest=00:6C:C5:B2:E2:FA ttl=9
12	349.6056700142757	192.311.200.1	172.26.6.1	ICMP	74	Echo (ping) Reply src=8B:4E:00:31:89:40 dest=17:98:94:5C:FE:D1 ttl=9

Figure 8: GothShark Window

GothShark features a similar interface as Wireshark. Providing the user with information about MAC, IP, Data and other relevant information. *GothShark* currently supports the most basic information Wireshark provides.

2.8 Hardware

The original idea was to implement hardware components like CPU, RAM, HDDs and Network adapters. As this would not have any effect on the application's features, it was replaced with the possibility of buying new networks and nodes. Buying a new network enables the user to choose a new location to operate from.



Figure 9: Hardware Store

2.9 Help Central

The Help Central features a tree-style structure of game components. Clicking one will present useful information about the features and options.

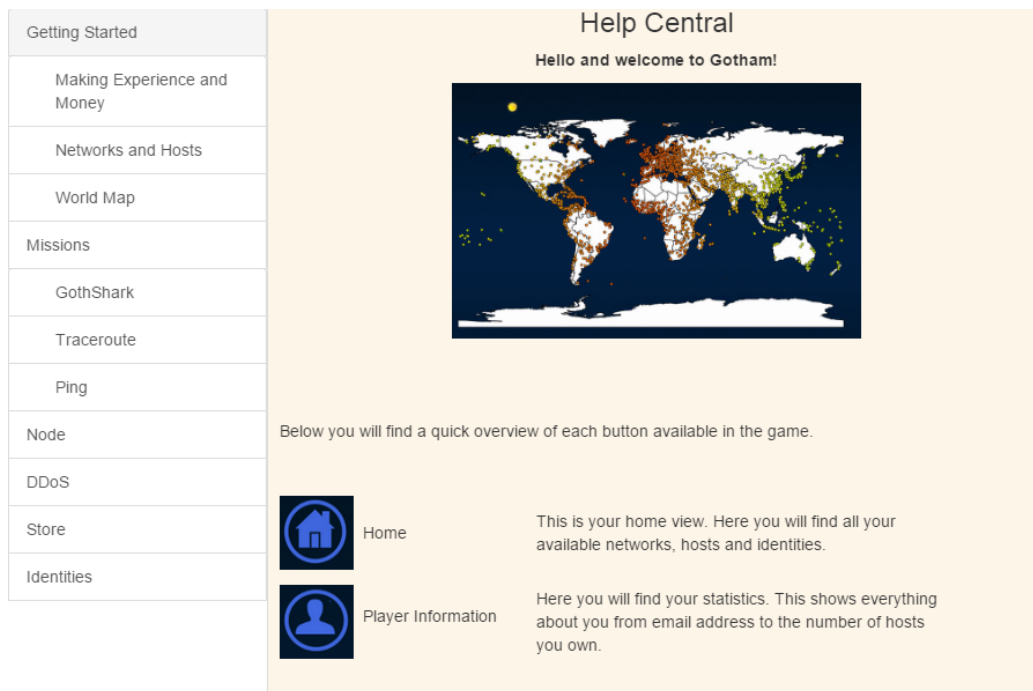


Figure 10: Help Central

2.10 Pedagogical Aspects

Due to this project's focus on learning, it requires some entertaining aspects instead of pure network simulation. What makes most games entertaining is progression and the feeling of accomplishment. As progression unlocks new features, there will always be the excitement for what is next. The goal is to make learning a good experience by gamifying the application. This is done by making missions rewards like experience points and money. Making enough experience points unlocks new missions.

The other form of gamification is storytelling. Each mission has its own background story. The goal of this story is to create a more complete mission experience.

3 Technical Design

Designing the technical requirements of an application is a demanding task, defining the implementation. This chapter describes how the Front-end and Back-end communicates together, representing the product as a whole.

The following specifications are defined for the Gotham Project

Project	Front-end	Back-end
Language	CoffeeScript	
Platform	Modern Browsers	Node.JS
Code Pattern	Modular, HMVC	Namespaced / Modular
Namespace	GothamGame	GothamBackend
Release Version	1.0	
Stability State	Stable	

Table 1: Application specifications

This table is based on research in the pre-project period. Some alterations have been done in order to better support the required implementations.

See appendix for *Architecture diagram A.2*

3.1 Language Choice

Coffee-script is programming language that transcompiles to Java-Script. Transcompiling is the process of compiling one language to another on the same level of abstraction. Coffee-script simplifies the syntax of Java-script by enhancing readability and efficiency of writing code.

Designing this project, it was required to write dynamic based code. This means to dynamically alter a object/class during runtime. This would not have been possible in for example C#.

Because of the structural complexity of networking, the flexibility of dynamic typing comes handy when defining generic functions for the application.

Coffee-script is a functional programming language that supports prototyping. Prototyping can simulate classes, but are in reality objects. These objects can be modified at runtime which makes packet handling and network management trivial.

3.2 Structural Design

3.2.1 Back-end

The back-end is designed as a module based dependency hierarchy. This design is based on having a object bound to the global scope which can be accessible everywhere in the NodeJS runtime. A good example of this is the two database instances of the back-end, namely *Database* and *LocalDatabase*. First the Database caches the remote mysql database, then the Local Database is synchronized via the global namespace.

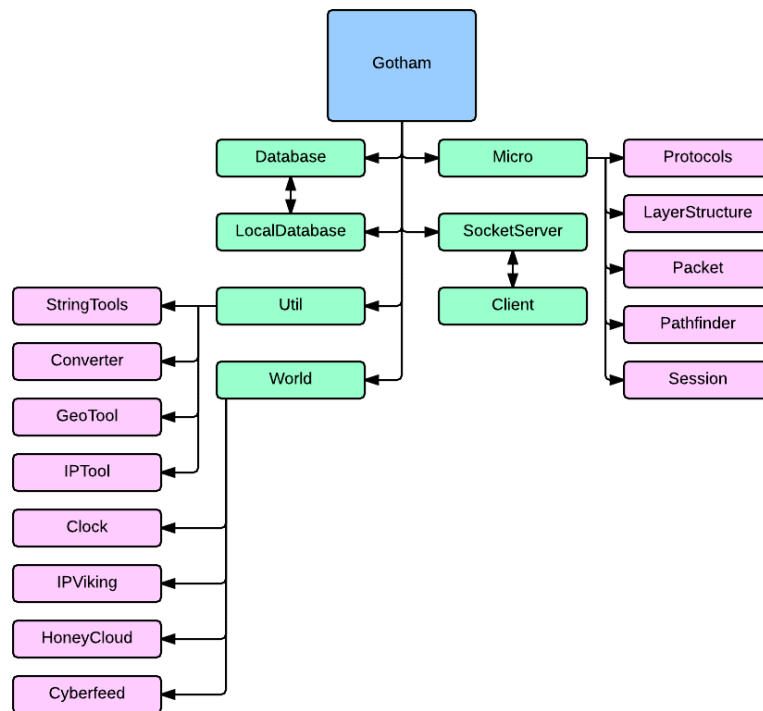


Figure 11: Namespace hierarchy defined in the Gotham back-end.

This design allows developers to rapidly implement unit tests, and at the same time develop in an accelerated rate comparing to using *dependency injection*. The downside with this pattern is that it's easier to get memory leaks. This is because the code has no way of knowing when the all scopes have been closed.

3.2.2 Front-end

The front-end is developed with a customized Hierarchical-Model-View-Controller (HMVC) pattern. Usually the HMVC pattern is layered, which means that the loading scene and world scene is not able to communicate. The revised pattern used in GothamGame allows all scenes to communicate with each other. This adds flexibility like *cross scene observing* variables [14]

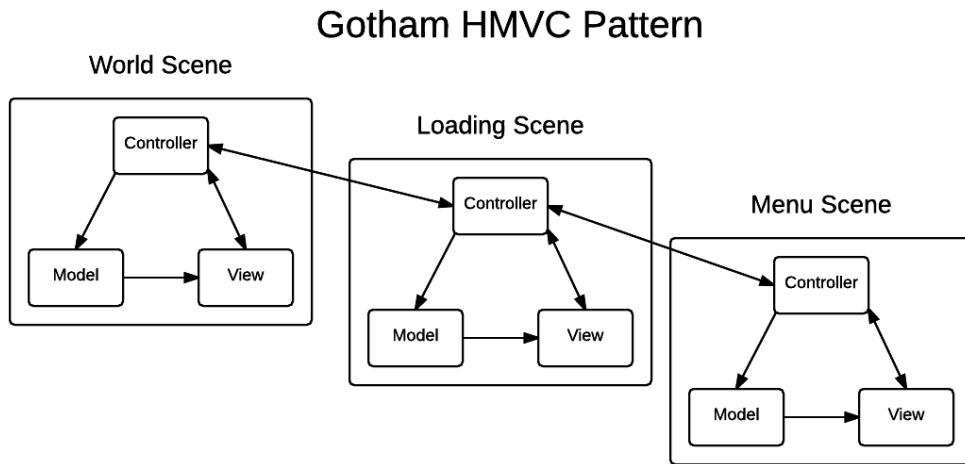


Figure 12: Front-End Structure

This is possible because all objects are added to the renderer, and can be retrieved on request. These are all properties defined by the Game Framework and makes it easy for later expansions if required.

The modular structure of the game framework makes it trivial to deprecate and create new features. The main class *Gotham.coffee* is the namespace class of the framework. All modules are included here and can be accessed through the Gotham namespace. Gotham is injected into the window scope of the browser, which gives access throughout the application. Pollution of the global scope is considered bad practice [15], and it is therefore wise to use the Gotham namespace for all code.

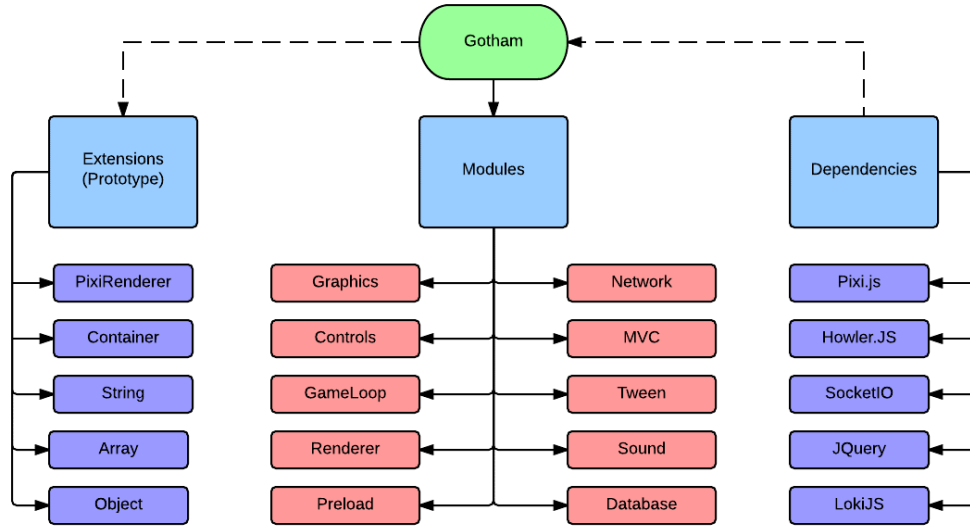


Figure 13: Game Framework Structure.

This illustration does not include depth level 4 which is classes contained inside *Controls* and *Graphics*. *Graphics* contains all drawable elements, while *Controls* is for example an interactive button.

3.2.3 Data Manager

This was primarily the back-end written in C#, now serving as the data manager and testing application.

The data manager consists of two parts, the debug client and data manipulation software. The debug client and data manipulation tools are in the same application, but are considered two separate parts due to their different tasks.

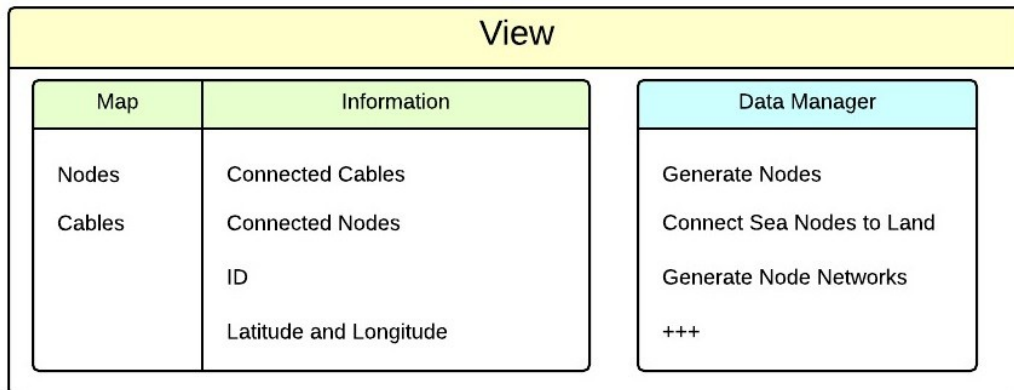


Figure 14: Debug clients view.

The debug client used a Google Maps library for drawing nodes and cables. This made it trivial to check data consistency compared to using the front-end world map. Mode details of this in the Implementation chapter.

The other important function is the data management tools. Some data is gathered from the Internet. Tools was made to parse this information into specific nHibernate entities required to write it to the database. The debug client also features buttons for the most common data management processes.

3.3 Database Design

The database tables can be categorized into four parts, *cable*, *node*, *network* and *mission*. The figure below represents the final structure of the database.

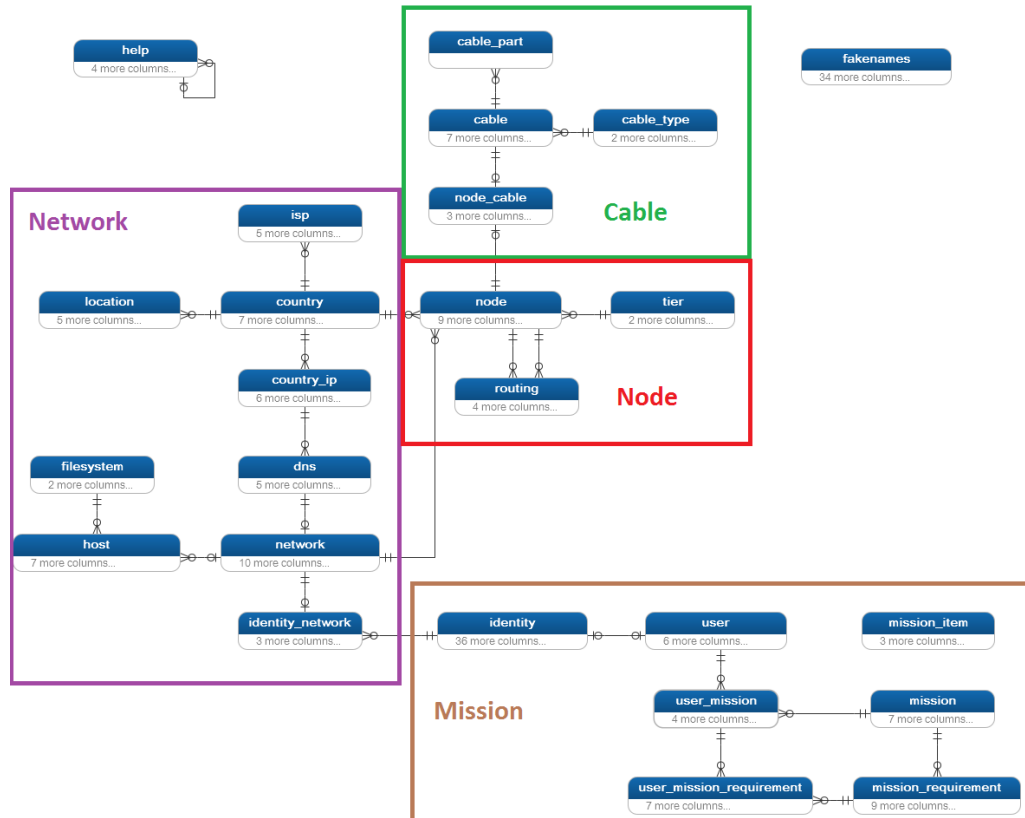


Figure 15: Table structure of the back-end database.

A more detailed version of the database structure can be found in the appendix[A.4]

3.4 Network Communication

The network module of the project is build on top of Socket.IO Socket.IO makes it possible to easily transmit packets between the front-end and back-end.

The back-end serves processed data to the front-end client through *rooms*. Rooms are an abstraction of the already existing Socket.IO room system whereas a room was equal to an emit. The back-end is designed so that one room can hold multiple emits. This makes it trivial to separate traffic in the application.

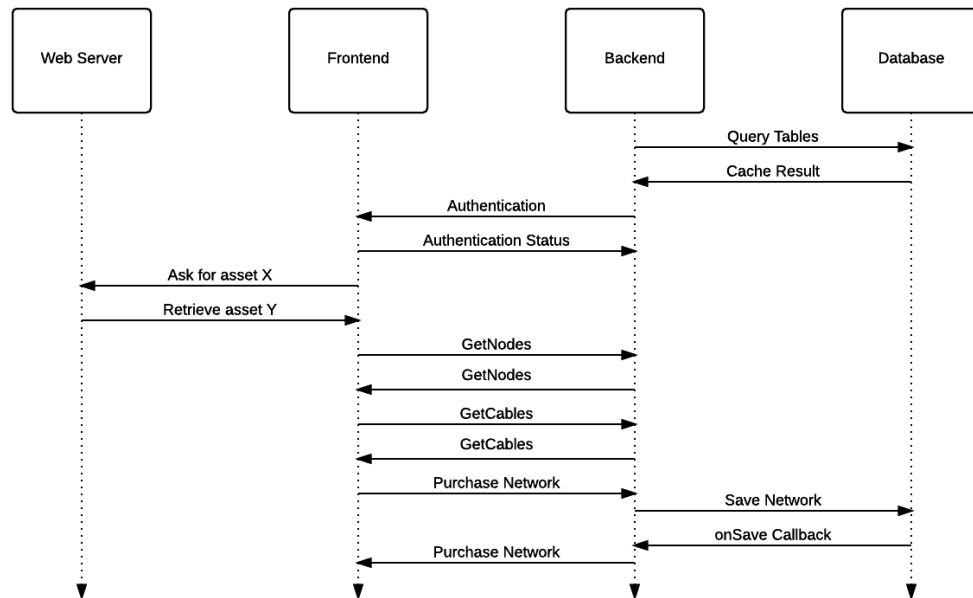


Figure 16: Traffic Flow Between Modules

This is all possible due to the callback-based room design made specifically for this project.

3.5 Limitations

The technical design is currently limited to a module pattern in the back-end and HMVC in the front-end. Both these designs are sufficient for the application and it is possible to scale the application parameters with this technology.

Some limitations exists as the front-end is run in the browser. There are libraries to fix some of these limitation, but there are no official solution. The first is multithreading. Java-Script does not have the ability to thread processes. Most issues requiring multithreading is solved by Java-Script's asynchronous code execution but limits the use of CPU cores to one per process.

Browser compatibility. Should the browser be out of date or not support certain parts of the Java-Script ECMA [16], the application may have difficulty executing properly.

The back-end is written in Coffee-script, using Node.js as execution platform. Coffee-script is not optimal for CPU heavy tasks. For example, a task taking 400 milliseconds in C# would prove to take 14 seconds in Node.js. This can however be optimized by utilizing Node.JS asynchronous properties.

The old back-end, now serving as data manager features one severe limitation resulting in the migration to Node.js. The strongly typed structure of C# would prove to be a issue when generating OSI-layer based traffic headers. The creation of dynamic objects is better handled in Coffee-Script.

4 Implementation

4.1 Game Framework

The game framework is a module based, easy to use framework, focusing on simplicity. Some modules are bare wrappers for other libraries, just to include it into the Gotham game framework. There have been some discussion in the group whether the definition Game Framework is the correct term. According to DocForge [17], “A *software framework* is a set of source code or libraries which provide functionality common to a whole class of applications.”, which means that the software can be categorized as a game framework.

The most important module of the Game Framework is the MVC bootstrap, which the frontend application heavily rely upon. The MVC bootstrap makes it quick and easy to divide logic and design code. The abstraction layer given by the Game Framework has made development of Gotham considerably easier, and the group estimates that recurring code is reduced by approximately 30 percent.

Some modules are not mentioned here because they currently only acts as namespace wrappers. A namespace wrapper is a way of adopting library functionality to a new framework. The adopted classes is primarily located in the *Gotham.Graphics* namespace. All adopted code is planned to be extended in an later version of the game framework.

4.1.1 Database

Current implementation the database is based of *LokiJS*, an in-memory database. The Database is an static instance, which only allows for one instance but an infinite number tables can be created if needed. To make it as simple as possible, new tables are automatically created if they were not previously accessed.

```
1  # Retrieve table with name "TableName"  
2  # Creates this table if first access  
3  table = Gotham.Database.table "TableName"
```

Listing 1: Creating table instance

Furthermore, the database can be exported to the browser's local storage if loading times become a problem.

The database is used specifically for storing assets in the application. Current database implementation can also be used for storage of alternative, namely temporary storage for user created data.

4.1.2 Resource Handling

Resource handling is done by the *Preload* class. The preload class uses the database module as an dependency. The preload module's task is to load assets before the game starts and give easy access to these at runtime. See figure 16. It currently supports *json*, *image*, *mp3*, *network streams*. Each of these methods are based on downloading a file from a remote server. When this is done, it is stored in the database with a name chosen by the developer.

```
1  downloadJSON(url , callback)
2  downloadImage(url , callback)
3  downloadSound(url , options , callback)
4  network(name, table , socket)
```

Listing 2: Preload download handlers

Each of the download handlers takes in a url and runs a callback on completion. When this callback is executed, the retrieved file is stored in the appropriate database table.

The preload class keeps tracks of current number of loaded assets versus the total amount to load. When all assets are loaded, an *onComplete* callback is sent to the application.

In Gotham front-end this *onComplete* callback is wrapped around a loading screen. The game is then started when the preloader calls the *onComplete* callback.

4.1.3 Tweening

Inbetweening or *Tweening* is a way of generating frames between a start and end point. This is used in graphic animation to give the illusion that something is moving smoothly. Already existing implementations of tweening in java-script was not sufficient this project. This is why an implementation of such Tweening engine was created.

Tweening is done by calculating a position based on current execution time of the application. When a tween is created, current execution time is recorded as the *startTime* and in addition, the *endTime* is calculated based on the *startTime*.

```
1
2  # Create tween instance
3  tween = new Gotham.Tween(object)
4
5  # Waits 0.5 Seconds
6  tween.startDelay 500
7
8  # Repeat tween Infinitely
9  tween.repeat(Infinity)
10
11 # Easing method
12 tween.easing Tween.Easing.Circular.InOut
13
14 # Move object to x: 20 in 2 seconds
15 tween.to {x: 20}, 2000
16
17 # Move objects y position to 20, on 20 seconds
18 tween.to {position: y: 20}, 20000
19
20 # onStart callback
21 tween.onStart ->
22   console.log "Started"
23
24 # onUpdate callback. Every iteration
25 tween.onUpdate ->
26   console.log "Updated!"
27
28 # onComplete callback, when complete
29 tween.onComplete ->
30   console.log "Completed!"
31
32 # Start the tween
33 tween.start()
```

Listing 3: Tweening usage

The tween implementation made for this project is rather special. This is because it can support tweening of nested objects without any drastic performance loss.

Following is a output from benchmark test done between Tween.cs and the popular Tween.js [18]

Configuration	Result
[Deep] Tween.cs	110.11540000035893ms
[Shallow] Tween.cs	16.299000000253727ms
[N/A] Tween.js	16.071699999884004ms
Parameters: 100x10000 Iterations	

Table 2: Tween.cs benchmark comparison

As one can see, the nested version (Marked with *[DEEP]*) is slower than the shallow version. Per iteration the deep version uses 0.11 microseconds versus 0.0163 microseconds for the shallow one.

In order to get this performance, the tween loop must be as simple as possible. The tween loop first checks if there is any tweens currently running, if this is the case it continues to loop through the list of tweens. For each of these tweens, checks are done to validate the tween for further processing. It checks if the tween is started, has any chain items left and is initialized. If all these conditions are true a new tween action can be executed.

The tween engine made for this project is used in all animations, and is proven to be sufficient for current precision needs.

4.1.4 Patterns

The game framework currently have extensive support for the MVC pattern. The MVC pattern is made up of two base classes, *View* and *Controller*. The model can be freely chosen by the developer but will in many cases be the Database module in the framework.

For the View, the same *create* method must be overridden.

A controller consist of one function, called *create*. This is a function that must be overridden by the new controller class in order to work. The Controller

also takes in the View as a parameter in the constructor. This implementation allows for *hot-swapping* views using the same controller. This is often desirable if the same data is used in different views.

4.1.5 Game Loop

The game loop implementation is based on PIXI.js bootstrap version of `requestAnimationFrame`. This is a built-in redraw function for java-script. Whenever this function is called, the browser is instructed to redraw whenever possible.

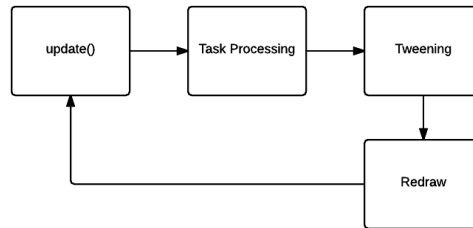


Figure 17: Front-End Game Loop

By calling the update function, logic is updated followed by a redraw.

4.2 Front-end Application

Rendering and designing is often more time consuming than logical implementation. This is because today's consumer market demand good visuals and special effects in their applications. The goal of the front-end implementation was to make an abstraction layer which allowed for further expansions after this project. Also keeping the code clean and simple was a priority. All implementations done in the front-end are based on the game framework, made specifically for this project.

4.2.1 Mission Engine

The mission engine is arguably the most advanced logic based feature in GothamGame. This engine consists of three classes, *MissionEngine*, *Mission* and *Requirement*. *MissionEngine* is the handler for all missions. It keeps track of all ongoing, as well as available missions. The *MissionEngine* is based on an emitting system. Throughout the application, different emit triggers are placed. For example when the user pings an IP-address, an emit is sent with the code "Ping" and the value "IP-ADDRESS". The *MissionEngine* then iterates through all ongoing missions to see if there are any missions with a matching requirement.

All missions contain *requirements*. A requirement is a simple task that needs to be completed before the mission status can be set to complete. A requirement is defined with the following variables: *requirement*, *default*, *expected*, *emit*, *emit_value*, *emit_input* and *emit_behaviour*.

requirement variable is the name of the requirement.

default is the requirements default value when first started.

expected is the value that has to be met for the requirement to be complete

emit is the trigger for the requirement. For example "Ping"

emit_input is the value that has to be emitted for the requirement to register the change

emit_value is the value the set to current value when emit is valid

emit_behaviour is which behaviour that should happen when a valid emit happens.

Example:

Consider a requirement with the following values:

Name	Value
requirement	Ping
emit	Ping
default	None
expected	81.166.17.74
emit_value	81.166.17.74
emit_input	81.166.17.74
emit_behaviour	set

Table 3: Mission Requirement example

Case 1: The client pings 81.166.17.75.

The MissionEngine loops through all missions and checks for a requirement with the parameter *emit: Ping* and *emit_input: 81.166.17.75*.

No matches are found and nothing happens.

Case 2: The client pings 81.166.17.74.

The MissionEngine loops through and finds that there are a requirement with *emit: Ping* and *emit_input: 81.166.17.74*. Since the *emit_behaviour* is “set”, the *emit_value* is set to *current*. In this case the *expected* and *current* value will be equal, which triggers *onComplete* callback on the requirement.

The client sends a network emit to the back-end, updating the requirement state in the database. If all requirements for this mission is complete, the mission itself can be completed.

4.2.2 Scenes

The front-end client consist of three scenes, *Loading*, *Menu* and *World*. These extend *Gotham.Graphics.Scene* which are in reality a PIXI.js container. A scene can contain graphic elements like sprites and textures.

All scenes overrides *create* which initializes the scene. A scene contains multiple objects. These object are in the client represented as MVC objects. The scene abstraction makes it possible to extend the MVC pattern to HMVC (*Explained in technical design*).

4.2.3 Objects

In early implementation of the front-end, no design pattern was used for graphical elements. Later on, an implementation of the MVC pattern was introduced to the game framework, which made it possible to better separate the graphic elements from the business code.

The scene allows objects to access sibling objects. For example; the world map can access the terminal, using `@scene.getObject("Terminal")` call. This makes it trivial for objects to share information.

4.2.4 World Map

The world map is the biggest object in the application. The class itself is a total of approximately 1100 lines of code. The task of the WorldMap is to visualize all events bound to the world. Example of such events are *Traceroute* and *Ping*. This is done by drawing graphics onto the map and animate using the *Gotham.Tween* library. The graphics library used is only a renderer. This makes it challenging to rotate, scale and translate graphic elements.

In reality the World map is a bunch of polygons converted to sprites for better performance. The performance gained by doing this is approximately 50 fps. The downside by using sprites is that scaled objects becomes blurry. There was no implementation of the sprite method which could remove the blur. The implementation of the cables also gave a performance hit. Cables are drawn as primitives, which are expensive compared to a *sprite* element. It was however not possible to convert cables into sprites because it was no way of determining the correct cable position afterwards. Also the hitbox would no longer work.

Current implementation of the world map may seem messy, but in reality its not. This is because there is no abstract way of managing graphics elements in PIXI.js. This means that every translation, rotation and scale has to be done manually.

4.2.5 Terminal

This object is the unix-like terminal made for interacting with the world map. The implementation of software are abstract so that new applications can be made later versions. Current version of the terminal support *ping*, *traceroute* and a visual version of *slowloris*. In addition, a unix-filesystem has been parsed to json and implemented as the terminal filesystem. The filesystem itself support *cd* and *ls*.

All applications extends the *Application* base class. This class takes in a *command* object and has two overridable functions; *execute* and *switches*. The switches method is a implementation of an argument parser. This argument parser allows for commands with parameters ex: *ping 192.168.10.1 -c 0.2*. Execute is the function called whenever a user writes the command. Its task is to do whatever action that is desired.

4.3 Backend Server

Using C# as the back-end server was found to be inferior because of the lacking of dynamic type support for classes and objects. While it is possible to create dynamic objects in C#, this was not sufficient for our needs. Because of this, the planned micro traffic module suffered and it was not possible to get the flexibility required for the implementation. Attempts to implement the OSI Model in a object oriented way was therefore not possible. Node.js was a suitable replacement, which had all the requirements for this project and became the new back-end server. It features dynamic typing, good IO throughput, but suffers from poor processing abilities. This is caused by the event-driven architecture [19], which in general are slower at computation tasks. However there was no need for heavy computations in the project. Node.js was not perfect because it is designed to run asynchronously. Much of the already existing code was written synchronously, which blocked Node.js's event loop. A workaround for this was to implement code as promises. [20]

Additional details are found in the Data Management chapter.

4.3.1 Pathfinding / Routing

When establishing a connection with a remote host one wants to keep the path the data travels as optimal as possible. In the real world, routing tables built by advanced algorithms are used. As this is not an option for this project, some more common algorithms are used to determine a path.

Try-Random was the first pathfinding algorithm implemented. This is generally a bad algorithm trying a specified amount of random paths and picking the shortest. To get decent results about 100'000 random paths had to be made each time, resulting in 400 ms execution time in C# and 14 seconds on Node.js. This was a good reason to start working on the next pathfinding algorithm.

A* (A-Star) was the next algorithm implemented and assumed the best one for this project. This is generally a heuristic algorithm constantly working its way towards the goal, traveling around obstacles in its way. This was one of two versions of this algorithm implemented. This version did not have any form of error correction. This algorithm works good in most cases but could have problems finding complex paths.

B* is the last implemented algorithm and is used for all pathfinding if nothing else is specified. This algorithm has no relation to the official B*. This name is used because it is an extension of the A* algorithm above. B* works generally the same as A* but with a lot more features and optimizations. B* is able to backtrack, recognize shortcuts, check the path in both directions and expand itself should the path be difficult to find. This is definitely the best and most refined algorithm in this project.

4.3.2 Networking (I/O)

The back-end serves as a controller between the client and database caching all relevant data and distributing it. In addition all processing of tasks and information like pathfinding, traffic handling, time and traffic load on cables and nodes are processed on the back-end server.

Networking is done by utilizing Socket.IO's room architecture [21]. A wrapper class was created for this purpose which in Gotham renames a room to an event, whilst a room is rather a collection of events.

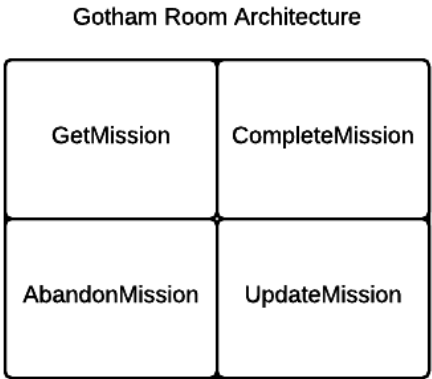


Figure 18: Networking - Room Structure

Each of the squares represent an emit/event. Whenever a user interacts with the front-end application, these emits will be sent back and forth depending on the user's action. Each of the rooms are registered to the socket server and by doing this, all new clients will be subscribed to this room. This system is flexible because it automatically subscribes the user to appropriate rooms.

Using Socket.IO, much of the routing work is done automatically. This is because of the flexible emit system.

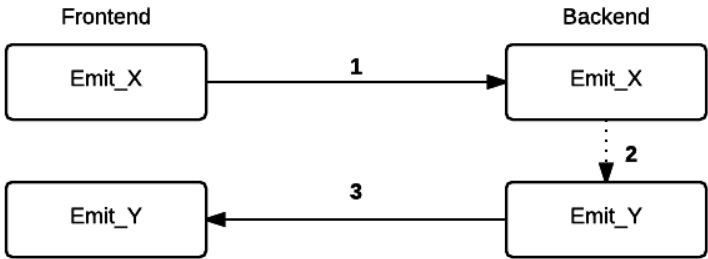


Figure 19: Networking - Emit.

Front-end client emits a signal X to the back-end. This signal is received in a

callback function which parses data to for example; retrieve something from the database. The back-end sends an emit to the front-end containing the response data. The front-end client then receives the emit response in its own callback method. The data can now be used in the front-end application.

The emit system have more flexibility than this. For example a single request emit can return unlimited number of response emits.

4.3.3 Traffic Structure and Sessions (Micro)

There is need for a packet structure (protocols and headers), packet data and session when emulating traffic. The packet structure is built from 5 out of 7 layers in the OSI model. *Link*, *Network*, *Transport*, *Encryption* and *Application* layer. As a session normally uses the same protocols during the whole session, the structure remains the same. The session object contains a structure, packets and a list of node information for each packet. As the content of some layers change when jumping from node to node, the difference is calculated when the session is generated and parsed on the client. This gives the front-end a template of the headers and a list of changes applied for each node. For example:

```
1
2  # Packet Template
3  template =
4      sourceMAC: 05:05:05:05:05:05
5      targetIP: 123.32.66.34
6
7  # Source
8  node1 =
9      sourceMAC: 15:15:15:15:15:15
10
11 # Target
12 node2 =
13     sourceMAC: 25:25:25:25:25:25
```

Listing 4: Packet structure

The session object supports calculation of variables like TTL and TCP sequence numbering. Should a packet with TTL 3 be applied to the session, the packet will only exist on the first three nodes. TCP sequence numbers are generated by the length of the packets. TCP packets now creates SYN and SYNACK packets but a solution with congestion control and detailed sequencing is not implemented.

A session may contain several packets. A packet is made of *data*, *direction* and *tll*. Data is a string containing the application layer data. Direction is a boolean telling if the packet is traversing from the target or towards the target. TTL is the Time To Live, decrementing on each jump and dropping the packet if it reaches zero.

In reality each packet has its own copy of layer structure but as this is an emulation, optimizations like described above can be done to reduce processing time and traffic.

4.3.4 Traffic Handling (Macro)

As the application features global traffic, some algorithms has to be developed to calculate the load on each node and cable. A general calculation is done by calculating the load based on the time in the current zone. When it is 4 O'clock in the night at any point in the world, the traffic is less than for example 6 O'clock in the evening.

This algorithm uses a calculation that returns the current time from a longitude coordinate. It then splits the time into seconds and by using the following code calculates the load.

```
1  # 1440 Minutes per day
2  deltaMinute = ((2 * Math.PI) / 1440)
3
4  # minutes += (currentHour * 60)
5  loadValue = Math.sin(deltaMinute * minutes)
```

Listing 5: Load calculation

4.3.5 Mission Generation

The mission system is as earlier described, implemented as abstract as possible. This is done by creating a specific syntax for generating missions. A mission is defined with requirements. These requirements uses the following syntax in order to generate dynamic missions:

Syntax: [`<Entity>|<Property>,<Index>`] This makes it trivial to create missions that require pinging numerous hosts:

[`Network|external ip v4, 0`] is an example, where it will be translated to a random network's IPv4 address.

Command	Result
[<code>Network external ip v4, 0</code>]	81.166.17.74
[<code>Network external ip v4, 1</code>]	10.43.39.124

Table 4: Parsing a requirement string to a external ip

This can be done for all entities defined in the project

4.4 Data Management

As this code was originally designed to be the back-end of this project the existing code and database connections are solid. Not all code was migrated to Node.js, specifically the consistency checks and unit tests. This is because Node.JS does not offer any advantages over C# in unit testing.

4.4.1 Debug Client

Before the front-end was developed for debugging of data, a small debug client was developed using the GMap library and google maps to draw nodes and cables. This library is very restricted to drawing only nodes and paths and would not seem fit as a client for this project. As a debugger it was sufficient. With an already existing world map, small amounts of code was necessary to generate a view of the nodes and cables, outputting the relevant information from the database.

The debugger was later extended to handle node searches and listing of connected nodes to specific cables and vice versa. This made it easy for developers to detect errors and check if algorithms like pathfinding functioned as intended. The debug-client was used to detect errors in the front-end client by validating node locations and cable connections.

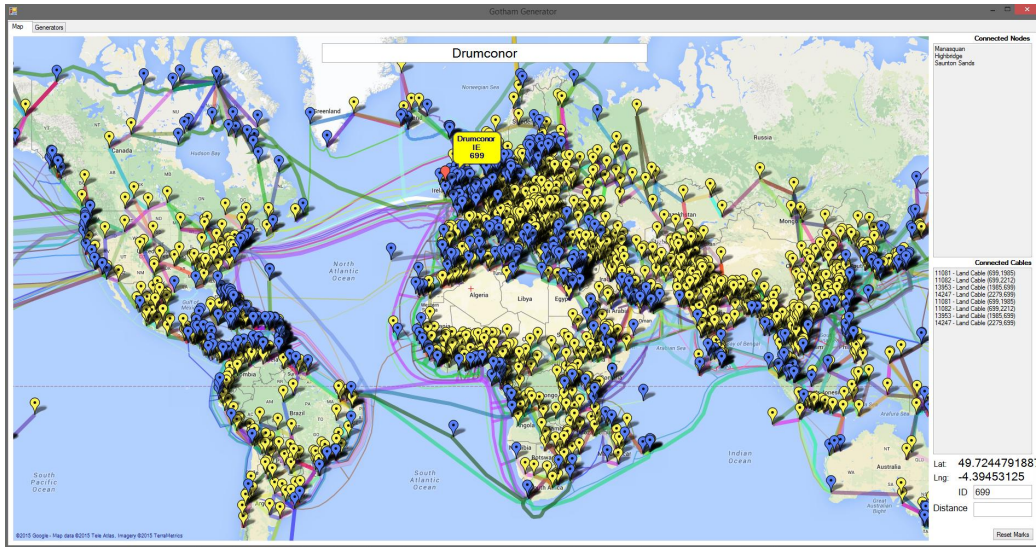


Figure 20: Debug Client

4.4.2 Python Scripts

Python was used for data management when small and non invasive modification of the database was required. These scripts mainly consists of web-scrapers and data manipulation tools. This project has a total of 10 python scripts which all have different tasks in the application. The benefit of using python is that it is very fast, and easy to implement in. Very few lines of code is required to complete the job. Scripts made for this project can be found in the appendix section A.5

4.4.3 Node Generation

The generation of nodes is a part that required large amounts of experimentation. As there is only partial realistic data of these nodes, the solution was to generate these by algorithms.

The first idea was to create nodes randomly around the world and drop them if they were generated in the sea. This generates an even distribution of nodes round the world but did not feature much realism.

Another idea was to gather locations from live DDoS services as IPViking or HoneyNet. This featured realistic locations. The problem was that most of these attacks happened between United States and China, excluding locations from Africa and smaller countries.

As there exists lists of all cities in the world with coordinates, it seemed possible to generate nodes from these coordinates. The problem would be that a massive amount of nodes would be generated.

The final and current solution was to use the previous "*all cities in the world*" idea and combine it with population. Generating a program iterating through all countries, using an algorithm taking population and returning random cities in that country seemed to work. The algorithm had to be customized to limit nodes should the country's population be very high. The algorithm also features some limitations on continents reducing the amount of nodes.

Some research has been done on mapping endpoint connections by doing ICMP requests via a botnet. [22] This gives some insight on traffic density. The node density are not exactly realistic, but this method prevents from having nodes in unpopulated areas. As this application is focused on understanding how the global network works, the generation of a more realistic network was not pursued.

4.4.4 Cable Generation

The generation of sea cables was not necessary as Greg's Cable Map already provided coordinates on existing sea cables and their nodes. These are parsed and written to the database. The issue was how to structure the data in a relational database as the original data was used in Coffeescript. This was solved by splitting the cables into parts.

After the nodes was generated, connecting them with cables required some thinking. The first attempt was to connect each node to nodes closer than a specified distance but as the density of the nodes vary by a great amount, this was not an option.

The second idea was to connect each node to its closest node, unless they were already connected. This featured a better network but had holes and illogical connections.

At last by iterating from left to right and connecting current node to two of the closest nodes to the right, the network seemed consistent and usable. This resulted in every node getting connections and small amounts of illogical parts.

As cables and nodes generally exists separately, they are connected in a table independent of their locations. This table is made by using a method that connects cable parts to nodes if they are close to each other.

Methods were added for repeated use like making of a single cable, getting the closest node or connect a cable to a node.

4.4.5 Data Parsing

All external data had to be parsed to be put into the database. Most information was comma separated and easy to parse. Some information was edited using Notepad++ macro functionality.

Information parsed was Greg's cable coordinates and different kinds of data containing countries and cities. This was done by splitting each lines into an array. Then splitting the line into an array of partial information using split-string.

4.5 World Map Generation

In the pre-project it was discovered that it was not suitable to use D3.JS for the World Map rendering because HTML DOM objects was too expensive to do transformations on. This triggered research on Canvas/WEBGL rendering which led to PIXI.js. PIXI.js is said to be the fastest WEBGL renderer out on the open source market [23]. A benchmark done by GoodBoyDigital supports our finding that DOM objects are simply not suitable for heavy graphic manipulation [24]

PIXI.js does not ship with a world map, and it was found that the best solution was to create a tool with the purpose of generating the map. A tool

for generating polygons from a world map image was required. Given this map, the following procedure was done to create polygons for each of the world countries:

1. Extract country borders from a world map image
2. Flood fill all countries with a unique color
3. Add border pixels for a country to a polygon array
4. Sort the polygon points with a sorting algorithm

4.5.1 Border Extraction

This algorithm removes all noise from further processes by removing all colors except from the black border surrounding countries. The generated image is saved to file for further processing.

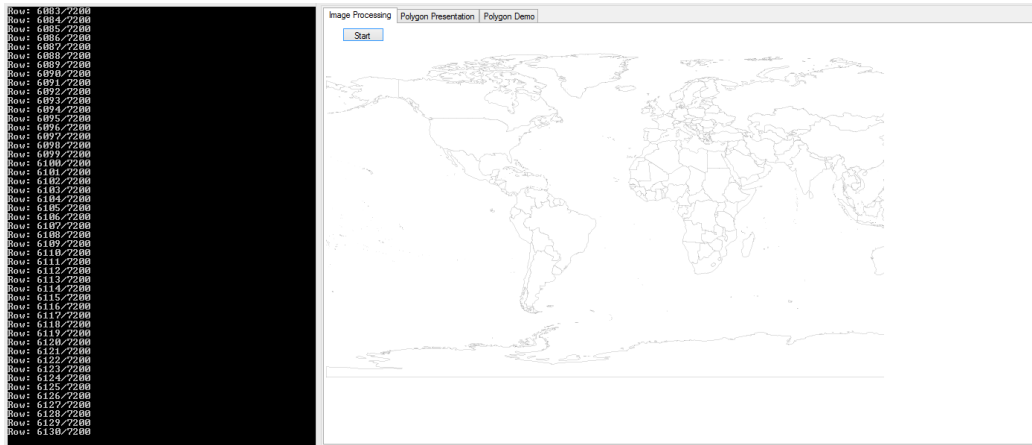


Figure 21: The process of sterilizing the image

4.5.2 Flood Fill

The flood filling algorithm finds the first available white pixel in the image, and starts to fill in all directions recursively. The algorithm used is very slow and use approximately one minute to fill a 7200x3600 image. This is because the algorithm loses big portions of CPU cycles to function calls.

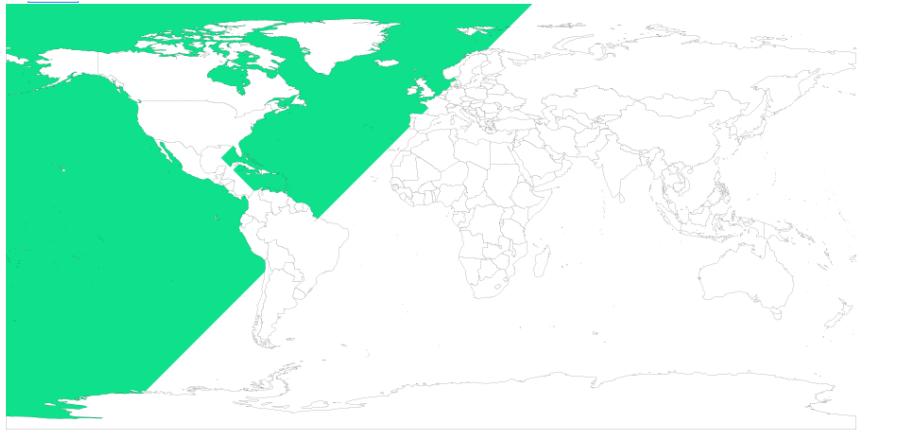


Figure 22: Beginning of a flood fill process

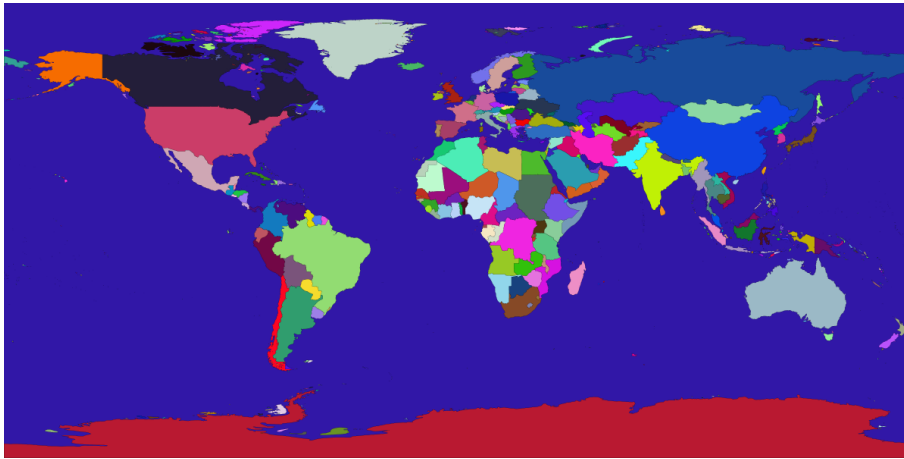


Figure 23: A finished flood fill process

When the flood fill process is done a unsorted polygon list have been created for each of the countries on the map.

4.5.3 Sorting the Polygon

Sorting a concave polygon is not trivial to do because a concave polygon can sometimes have more than one possible shape. [25]. The group could not find any existing algorithms suitable for this task, so an alternative algorithm was developed to solve this problem.

The algorithm does the following steps:

1. Start at a random point
2. Find closest point
3. Loop Detection
4. Add closest point to sorted list - Back to step 2.

The problem with this algorithm is that sometimes, looping pixels can be encountered, these loops causes the sorting process to halt as it enters a never-ending loop.

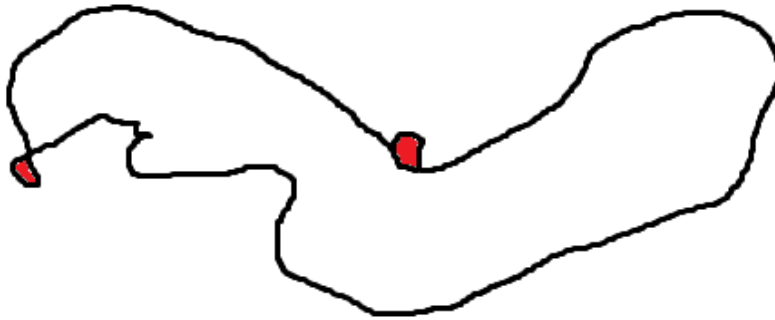


Figure 24: Example of loops in a polygon

To counter this problem, a backtrack mechanism was added. Backtracking adds a blacklist which triggered each time a pixel had been visited. If a pixel was visited twice, the backtracking initiated, and it reversed to last working point.

When the algorithm is done processing the image, an array of points is created for each of the countries. These are directly parsed into PIXI.js as a polygon.

4.6 Pathfinding

The pathfinding is a sequential operation of traversing through nodes. The most simple and least effective is *Try Random*. This works the following way:

1. Get start and goal nodes
2. Get a random sibling from current node and add it to path
3. Check if the sibling from step 2 is the goal node
4. Check if the path is longer than 100 or number of tries exceed 100'000
5. If not then repeat from step 2

The next algorithm is A* and works the following way:

1. Get start and goal nodes
2. Get all siblings from current node
3. Find the sibling closest to goal node and add to path
4. Set last node in path as current and return to step 2

The last algorithm (B*) functions the same way as A* but includes heuristics and optimizations.

1. Get start and goal nodes and various parameters
2. Run pre-check to fix invalid parameters
3. Get all siblings from current node and return closest.
4. Check for loops
5. Check if the path is traversing away from the goal
6. Check if a shortcut can be made if current node is close to a previous position
7. Save result, reverse the path, reset and return to step 3
8. If the pathfinding is executed after parameter specifications then return

The algorithms includes several details not provided in this description. This is documented in the code.

5 Documentation And Testing

The project has been documented by including code comments, and *yuidoc* code blocks. Documenting code is important because it allows for new developers to understand its purpose. Github features a method of hosting static content via a repository named *gh-pages*. These repositories are populated with the documentation of the project. The documentation can be accessed at <http://perara.github.io/Gotham/> for the whole project and <http://perara.github.io/Gotham-GF> for the game framework. Current state of the documentation is *partially complete* as it still lacks example code for some important modules.

Testing is important when developing new software. There was originally no plans for automated tests, but some were added to relieve developers for repeated testing. This project consists of two types of testing, *Unit Testing* and *User Testing*.

Unit testing are automated tests, which in term validate a block of code. These are often run automatically by task runners such as *Grunt* and *Team-City*. *Back-end*, *Front-end* and the *C#* project has unit test, with varying quality. The tests done in Coffee-script has not been kept up to date due to lack of time. The unit test made for the *C#* project are primarily to validate the database, keeping all components like cables, networks and nodes consistent.

The tests done in this project are not considered conventional. Developing conventional tests for all use-cases was not something that was prioritized for this version.

Both *C#* and Coffee-script features good tools for unit testing. It is important to take advantage of these to keep the application as bug-free as possible.

User Testing is the biggest test framework used. A checklist document was created for this purpose, making it possible to do extensive testing in a matter of minutes.

This document is located in the appendix. ??

5.1 Coffee-Script

The test runner used in the coffee-script projects are called Mocha. In addition this runner has multiple dependencies. Chai, a BDD/TDD assertion library for java-script was used to do assertion in the test logic. In addition a plugin called *Sinon.js* was used together with *Chai*. This made it possible to do more extensive mocking and spying on variables.

5.1.1 Traffic

As earlier described, the emulation of a layer structure is consistent through a session. Not all layers needs to be used. To verify that a layer structure contains a valid series of layers, a consistency check is implemented to throw an error should inconsistency occur. The current layer structure is [*Layer 2*, *Layer 3*, *Layer 4*, *Layer 6*, *Layer 7*]

An example of failed and passed tests follows:

Pass: [*Ethernet*, *IP*, *TCP*, *NoEncryption*, *HTTP*]

Pass: [*Wifi*, *ICMP*, *null*, *null*, *null*]

Failed: [*Wifi*, *IP*, *null*, *null*, *HTTP*]

As seen above, there can not exist a layer above an empty layer. Having a layer 7 without a layer 6 would result in a failed consistency check.

5.2 C#

5.2.1 Node Testing

To keep a consistent structure of nodes and cables, each are loaded from database and tested by the Data Management Application (C#).

- Unit tests are run on each node to check the following:
- Nodes cables are not null
- Nodes cables are greater or equal to 1

- Name is not empty
- Length of country code equals 2
- MAC is not null

By keeping these these consistent, it should not feature any fatal errors during runtime.

5.2.2 Cable Testing

The tests for cables are similar to nodes as some columns contain data critical for the application to run properly.

Unit tests are run on each cable to check the following:

- Cables nodes are not null
- Cables nodes are greater or equal to 1
- Name is not empty
- Cables cable parts are greater or equal to 1
- Type is not null

5.2.3 Coordinate Test

To verify that there is no errors in the coordinate conversion method, a random coordinate is converted between the two types 1000 times. Should the start and end coordinate differ, the test fails.

5.3 Test Overview

Testing is important, therefore the group made unit tests to validate the most important features of the application.

These tests are as follows:

Suite	Test Name	Quality	Language	Module
Networking	Connection OK	Good	CS	Back-end
Networking	Room Callbacks	Good	CS	Back-end
Networking	LayerStructure	Good	CS	Back-end
Networking	Pathfinding	Good	CS	Back-end
Mission	Check properties	Good	CS	Front-end
Mission	Requirement Check	Good	CS	Front-end
Mission	Trigger/Emit	Good	CS	Front-end
Mission	HackMission	Good	CS	Front-end
Cable	Consistency	Good	C#	Data Manager
Coordinate	Precision	Good	C#	Data Manager
Node	Consistency	Good	C#	Data Manager
Node	Have Network	Good	C#	Data Manager
Node	Bandwidth Consistency	Poor	C#	Data Manager

Table 5: List of all unit-tests created for this project

6 Results

6.1 Priority A

6.1.1 Node and Cable Generation

It proved difficult to find global realistic data of nodes and land cables. Even though some information was available this did not suffice as it was only partial. The data on sea cables exceeded expectations as a large list of sea cables around the world was available, featuring detailed coordinates of cables and their connected nodes.

The land nodes and cables was generated from algorithms making them as realistic as possible given the current timeframe. Sea cables was directly imported from *Greg's Cable Map* and ported to the application.

6.1.2 Traffic Simulation

As planned, traffic simulation happens on both Micro and Macro levels. The most basic part is the Macro traffic generating load depending on time of day in the current timezone. Micro traffic has the ability to simulate some of the most used protocols, most importantly ICMP and TCP. Packet content and header information are automatically generated by the back-end. MAC, Time and sequence numbers are dynamic values. These change depending on the packet number and current node the packet are traversing through.

6.1.3 Front-End

All required functionality was included in the end result of the front-end. The front-end features a user friendly interface. With descriptive icon based menu, and instructions in help menu. is expected that the majority of the implementation done can be recognized by an average computer science student.

Nodes and Cables is presented as markers on a world map. Nodes change color based in its network load, which gives a good representation on the

world traffic flow. Furthermore, a micro traffic flow is visualized by creating lines between nodes.

The required packet inspection tool was added with limited features in this version. However these features was adequate for this version.

6.1.4 Realistic Data (static)

To find large amounts of realistic data on the Internet consistent enough to be used was not expected. Luckily there was multiple sources of usable information. Greg's Cable Map proved to be a great source.

The most valuable data would be information about nodes and land cables, understandably this is not available for all countries. Fakenames.com provided with generated identities for the social engineering mechanism planned for further version of the application.

6.2 Priority B

6.2.1 Realistic Data (dynamic)

To be able to find live traffic load data would be a great advantage, this is however not available. The only form of live data available is DDoS traffic, from multiple sources.

This kind of traffic is highly relevant for this project and is streamed into the application with the option to toggle it on and off.

6.2.2 Real ISP Names

Several sites featured real world ISP names, some with IP ranges and country code. These were imported to the project database.

6.2.3 Algorithms

The plan was to implement alternative algorithms to feature possibility to see the difference between them. The first algorithm is the brute force style pathfinder. This uses several seconds to complete and does not necessarily find the best route. The next algorithm is A* constantly trying to get closer to the target with distance checks. This is effective, but does not necessarily find a path should the route be complex. The last algorithm is an extension of A* and provides multiple improvements and heuristics to find a good route even if the route should be complex. Ant Colony Optimization was hoped to be implemented but this would require a lot of time.

The generation of nodes and land cables was necessary as this information was not available. The result is satisfactory and are explained more detailed in the Implementations chapter.

6.2.4 Protocols and Services

The support for the most used protocols have been implemented as planned. It was also planned to implement different kinds of services that could exist on a host, like FTP. This was not accomplished as protocols did take a substantial amount of time to implement.

6.2.5 Missions

Missions are implemented as planned. More advanced missions was hoped to be generated, like accessing hosts and networks. Advanced missions was not prioritized as it was not the main focus of this version.

6.2.6 Money, Hardware and Software

All these aspects are implemented. Money can be used to buy new hardware. Software and hardware is used to earn money. The main plan with hardware was upgradable hosts. As this did not serve any purpose this early in the application development, the hardware components shop was changed to a network and host shop. Software is mainly used through a terminal. Examples of software is ping, traceroute and general linux commands.

6.2.7 Internet site

The internet site for the application is `http://gotham.no`. This features a link to the source code and the application.

6.3 Priority C

6.3.1 Creative Mode

Features like cutting cables was originally planned to give the user the possibility to manipulate traffic and see how it changes. The application features the possibility of disabling nodes. This reroutes traffic that would normally flow through it. There is also the possibility of reading the traffic traversing through each of the nodes in the world through a application called GothShark.

6.3.2 Header Simulation

Each Micro traffic packet generated has its own headers. Link, Network, Transport, Encryption and Application. These 5 layers contains values like IP, MAC, Port, Hash, Data and more. Applications in the front-end like GothShark have the ability to read this information and present it to the user. The structure of the headers are generated in a way that is expandable. Theoretically every value in a header could exist in these.

6.3.3 Hacking OS

The terminal functions as the user's base of running software. Current version features a unix-like environment with limited command-line tools. The tools implement for this version is `cd`, `ls`, `ping` and `traceroute`. The application system is developed in a way that it should be trivial to implement new software.

6.4 Priority D

6.4.1 Advanced Missions

The existing missions are more advanced than planned, making it a part of this goal. The missions have their general requirements that are random generated when queried. One mission may have multiple requirements making it trivial expand.

6.4.2 Rivals

Rivals are not implemented. The only multiplayer functionality is publicly shared packet information. Should a user accept a mission, another user can see the packets generated.

6.4.3 Dynamic Traffic

The traffic flows dynamically as the time changes in the world. At 18:00 the load is higher than for example 04:00. This change is featured as load on the different nodes. A red node has more traffic through it than a green node.

6.4.4 More Detailed Traffic

As Micro traffic has proven to be one of the most challenging aspects of this project. Traffic details beyond the header information is implemented in this version.

6.4.5 Botnets

Botnets are not implemented in this version.

6.5 Summary

Goals that were planned for the first version of the application were implemented without any mentionable discrepancy.

As expected, not all data needed were available but was generated without difficulties. The front and back-end works as planned with the exception of the migration of the back-end from C# to Node.js.

The application features hardware, identities, currency and different kinds of software . To be able to look closer on a packet stream, an application named GothShark features the packet information. The application has missions requiring the user to utilize different functions of the application together with tools for overview, like DDoS traffic. To experiment with the traffic, the user has the possibility of disabling nodes on right click.

Some goals were altered throughout the project period to better fit the application specifications. The most significant discrepancy of the original plan was the migration of the back-end from C# to Node.js. This did not cause any noticeable time loss as the merge was done without problems. Because of this change, the original planned goals became less difficult to implement. Parts of the plan were therefore re-evaluated to possibly include additional features.

Some goals had a secondary alternative of implementation. If the primary alternative was not possible to achieve, the secondary method was used. An example of this is node information which had no available information.

In addition, some of the D goals were *partially implemented*.

6.6 Future Work

The plan for most features was to start with implementing basic functionality followed by more advanced feature. The possibility of upgradable hosts was originally planned but as the developers currently could not see how this should affect any tools or features, it were redesigned to offer new networks and nodes instead.

Another part is Macro traffic. Even though it is dynamic it was hoped to be built with random variables that all affected each other. This is a good

example of a feature that was scaled down due to high complexity.

A project like this can be developed far past the main tasks and priorities described in this report. Some examples of future developing is presented by the *D* goals. Should the project continue after the project period this would probably be the main priority for improvements and implementations.

Since this project has virtually no limitations when it comes to features, some future priorities have been defined if the project were to continue. These priorities are categorized by the expected version number of that feature.

6.6.1 Version 1.5

Implement all goals that did not reach Version 1.0 of the application. Expand the unit test environment to have broader code coverage in overall. A code review stage should be initiated to check quality of the existing code, ensuring that it can support a expansion.

6.6.2 Version 2.0

The main goal of version 2.0 is to improve the macro traffic flow throughout the application. In order to do this, the plan is to implement a basic human AI to account for this traffic. This AI should also be to react to user interactions. The plan for the AI is to implement it as an behaviour-tree. This tree should include abilities to handle social-engineering missions.

Mission system is also planned to be expanded in this version. Already having a multi-depth mission system, new missions should be added to include social-engineering and AI interaction.

6.6.3 Version 3.0

Version 2.0 improved the macro traffic level of the application, which now allows for expansion of the micro traffic scope. This version is planned to include data-centers with the possibility to interact with each of the network components in the data-center. A datacenter is planned to feature router

configuration, host configuration and clustering of hosts. This version prepares for a expansion of the micro-traffic on a global level.

6.6.4 Version 4.0

This version is planned to feature new micro-traffic details, specifically for new protocols like HTTP. It should be possible to browse through all micro-traffic, using the GothShark application.

7 Group Evaluation

7.1 Discussion

As expected the project has taken a significant amount of time. It has been a focus to develop and keep the application abstract and easy to implement to new features. This results in a more time consuming development.

Should the application be developed using an existing game framework, development would be faster but the end product would not fulfill the required abstraction. This would make the application code limited and harder to develop further.

No existing graphics library is designed for this purpose and no similar program is known to exist. Making it challenging to create a developed structure and approach for this project.

By developing an abstract and dynamic graphics framework, the application and framework is now able to be developed further. The game framework is additionally ready to be used in new projects as required documentation and issue tracking is available.

The project participants are satisfied with the product considering that all goals were achieved by two people.

7.2 Conclusion

The developers had many ideas for the project, much more than the project period allowed. Should all ideas for this project be implemented, the project period would have to be extended drastically.

Careful planning resulted in a priority list of goals, where several of these had possibility of improvements. This made it easier to follow the plan with the time given. Some features took longer than expected and was implemented on a more basic level. The features that would prove to be easier than anticipated to implement was improved where possible.

The project have been educative and interesting to work on for the participants.

Would this product be usable for students? At this point, some features is simulated sufficient enough to give the student an introduction in the global digital infrastructure. The application is not suitable to give a deep understanding in networking but has capabilities to be expanded to implement required features.

References

- [1] “Stack overflow,” <http://stackoverflow.com/>.
- [2] “Goodboydigital, the home of pixi.js,” <http://www.goodboydigital.com/>.
- [3] “Microsoft, msdn,” <https://msdn.microsoft.com>.
- [4] “Html5 game devs,” <http://www.html5gamedevs.com/>.
- [5] “nhibernate,” <http://nhibernate.info/>.
- [6] “Fluent nhibernate,” <https://github.com/jagregory/fluent-nhibernate>.
- [7] “Loki.js,” <http://lokijs.org/docs>.
- [8] “Taffy.js,” <http://www.taffydb.com/workingwithdata>.
- [9] “Sequelize,” <http://docs.sequelizejs.com/en/latest/>.
- [10] “Howler.js,” <https://github.com/goldfire/howler.js>.
- [11] “Socket.io,” <http://socket.io/docs/>.
- [12] “jQuery,” <https://api.jquery.com/>.
- [13] “Msdn - microsoft code standards,” <http://msdn.microsoft.com/en-us/library/ff926074.aspx>.
- [14] “Tutorialspoint - design patterns - observer pattern,” http://www.tutorialspoint.com/design_pattern/observer_pattern.htm.
- [15] “Devspoint - pollution of global namespace,” <https://devspoint.wordpress.com/2014/06/16/js-link-avoid-polluting-the-global-namespace/>.
- [16] “Ecma standard,” <http://www.ecma-international.org/>.
- [17] “Docforge,” <http://docforge.com/wiki/Framework>.
- [18] “Tween.js,” <https://github.com/tweenjs/tween.js>.
- [19] “Node.js,” <https://nodejs.org/about/>.
- [20] “Mozilla - promises system,” https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.
- [21] “Socket.io - rooms and namespaces,” <http://socket.io/docs/rooms-and-namespaces/>.
- [22] “Spiegel - carna botnet,” <http://www.spiegel.de/international/world/hacker-measures-the-internet-illegally-with-carna-botnet-a-890413.html>.
- [23] “Pixi.js,” <http://www.pixijs.com/>.
- [24] “Goodboydigital - to dom or not to dom,” <http://www.goodboydigital.com/to-dom-or-not-to-dom/>.
- [25] “Stack overflow - ordering concave polygons,” <http://stackoverflow.com/questions/20141812/ordering-concave-polygon-vertices-in-counterclockwise>.

A Appendix

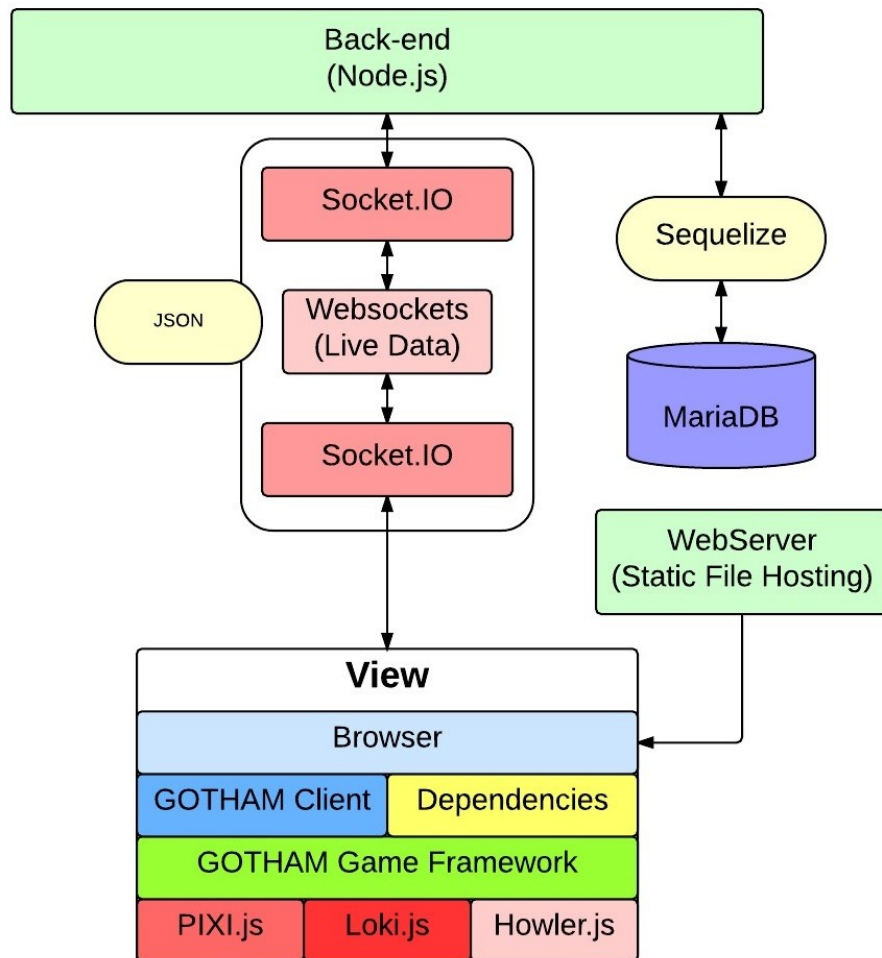
A.1 Code standard list

The following programming standards and guidelines applies for the project:

- Keep the complexity as low as possible where applicable
- Code comments before the code
- Function names in capital CamelCase in C#, normal camelCase in JavaScript
- “Useful” variable name wherever applicable
- Class names start with capital letters.
- Enums and static values: Capital letter only (ex. static final int TOP = 1)
- General code formatting using the tool standard (Fix indentations, remove blank lines etc)
- Class, method or function start bracket on the next line
- Names of methods and classes should be as self-descriptive as possible
- One statement per line (Wrong: a = 1; b = a*a)
- Flags are boolean only (Not Number).
- Recycling: If the same “large” function is used with various variables in different places, that function has to be written as a method to limit copy-paste
- No spaces in variable and method names. If spaces have to be used in other places, “_” is used instead.

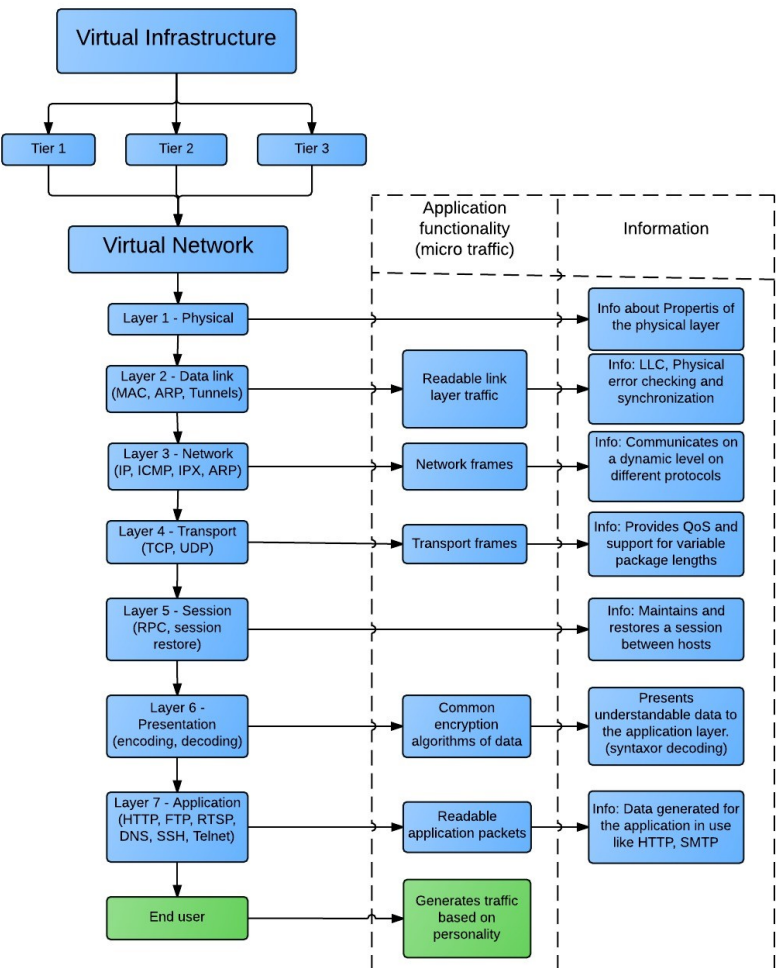
A.2 Project Architecture

Architecture



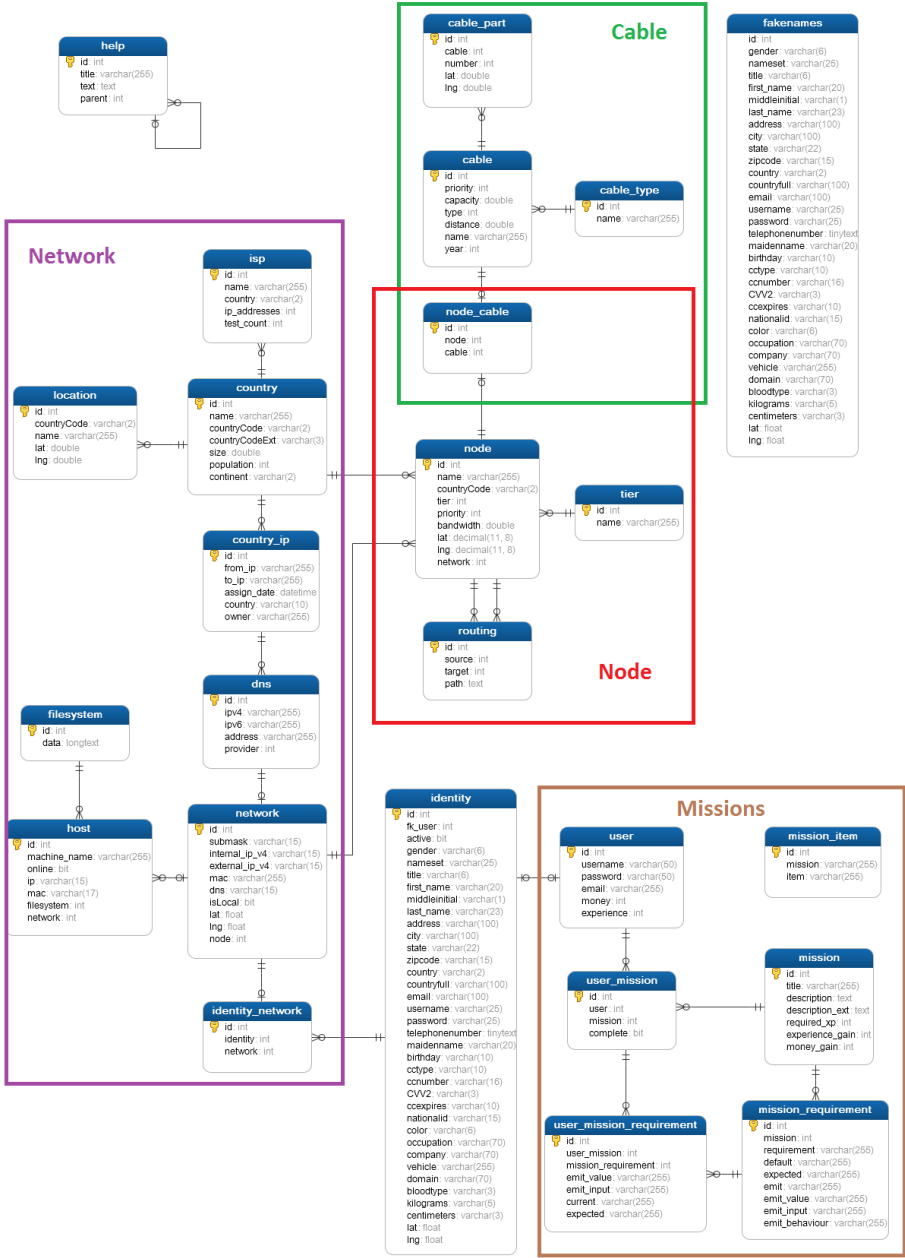
Architecture of services and libraries

A.3 Traffic structure diagram



Traffic and layer structure

A.4 Database structure



Database structure including all fields

A.5 Python Scripts

All python scripts can be found in `./source/python` directory of the project hand-in

Assign-ISP-country-ip.py Assigns a ISP Provider to a IP-Block in the database

Country-IP-Blocks a scraper for retrieving a country's IP block reservation

Level3-Node-Parser a parser for Level3's nodes in the world.

filesystem.py a linux filesystem parser. Output is a json formatted filesystem

fetch-isps.py Fetch ISP data from speedtest

generate-property-documentation.py Generate YUIDOC documentation for tables in the database

isp-load-statistics.py File with LOAD statistics for ISPS

node-distance-measurement.py Removes nodes which are closer than 10 km to eachother. Transfers node 1 to node 2

push-isps.py Saves ISPs to the database

SMALLFIX-NODE.py Synchronizes network and node references

A.6 Group contract

Main Goal:

Develop a well-tested and structured application within the project time-frame. Deliver a product containing the specified A, B, C goals and hopefully more. In order to fulfill this goal the group has agreed to the following:

1. Participation

It is expected that the project tasks are equally distributed and that the person tries to complete their tasks in time. If a group member can't meet at the designated time he or she is required to inform at least another group member.

2. Responsibilities:

Every group member is expected to complete their task with their best effort and ask if there are uncertainties or difficult tasks.

3. No conflict policy:

Listen and respect the other group members. Only constructive criticism. On disagreements the group member in charge of the part being disagreed on will have the final word. The other members are obligated to respect this and adapt the code to this choice by best effort.

4. Other:

1. Changes must be agreed upon by all members.
2. Short report after each meeting containing work done and work to be done.

A.7 License

The following is a copy of this products MIT License:

“The MIT License (MIT)

Copyright 2015 Per Arne Andersen and Paul R Lilleland

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

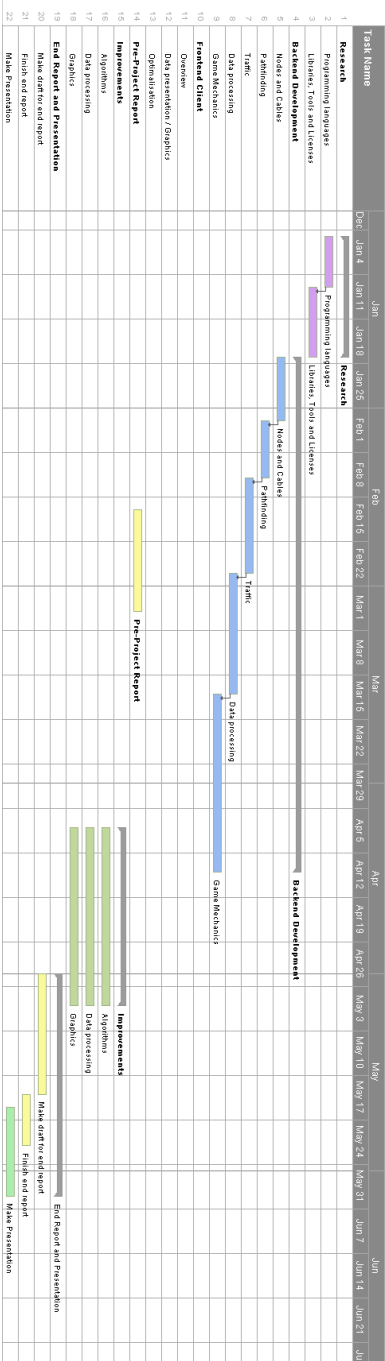
THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.”

A.8 Explanations

Macro traffic is used for presenting large amounts of traffic in the simulation. This uses only statistical information on each node and cable. This is an requirement as it allows the application to simulate most of the network traffic with minimal overhead. Macro traffic does not contain any low level information like packet data.

Micro traffic is when detailed data simulation is required. This type of traffic should be recognized as how packets behave in reality. Micro Traffic contains packet information like protocol, destination and source. This traffic requires additional processing by the server, and it is therefore important to not overuse this type of traffic in the simulation.

A.9 Gantt



Gantt Diagram

A.10 Team Member Contribution

As explained in chapter [1.7.1], each group member had their own areas of responsibility. Per-Arne have been in charge of the Game-Framework and Front-End, Paul R. was in charge of the Back-End including the Data Manager (originally a part of the Back-End but was split into its own solution).

Per-Arne started of with the game-framework, at the same time developing the front-end. When front-end was ready for network communication, work on the back-end began with developing database and network communication.

While the front-end was under development, Paul R. started working on collecting data and making it presentable. This required a small debug client to draw nodes and cables. Experimentation of node and cable generation was performed simultaneously.

When front-end was partially-complete, the phase of improving the product began. Fixing bugs, implementing animation, sounds and graphic. Paul R. improved the algorithms while Per-Arne implemented the last goals defined for this project.

A.11 Press Release

Ask a network administrator if they found it easy to understand how packets, traffic and the Internet works. Your answer will probably be "No". GOTHAM is an application to give students the opportunity to learn it the easy way.

GOTHAM contains a user friendly interface with a lot of features. Presenting realistic cables and routers located all around the world, with network load varying every minute. Should the user establish a connection to another host, the traffic path and detailed packet information is presented on each router in a realistic format.

The application also features a gaming aspect consisting of missions completed from multiple identities, networks and hosts. By being a hacker working in an emulated Linux like environment, the whole world is your target (for efficient learning). You even have a live, real-time DDoS map of attacks happening in the world at this very moment.

Educate your students more efficiently with GOTHAM, the Global Online Traffic Hacking And Monitoring application.

A.12 Test Sheet

External file: GOTHAM_Test_Sheet.pdf

A.13 Time Logs

External file: GOTHAM_Timesheets.html

A.14 Issue List

External file: GOTHAM_JIRA_Issue_List.html

A.15 Meeting Logs

External file: GOTHAM_Meeting_Logs.pdf

A.16 Pre Project Report

External file: GOTHAM_Pre_Project_Report.pdf

A.17 Project Description

External file: GOTHAM_Project_Description.pdf