**PowerGrid documentation**

**An overview of the PowerGrid project, how it works, and what the differences are with other Runescape bot clients.**

****

Table of Contents

[Chapter 1 – Introduction 3](#_Toc357190395)

[Chapter 2 – Overview of PowerGrid 4](#_Toc357190396)

[2.1 – Artificial Intelligence as a way of playing Runescape automatically 4](#_Toc357190397)

[2.2 – Communication with the Runescape client 4](#_Toc357190398)

[Chapter 3 – Structural overview 5](#_Toc357190399)

[3.1 – Structural diagram of PowerGrid 5](#_Toc357190400)

[3.2 – Summary of each of the modules in PowerGrid 6](#_Toc357190401)

[3.3 – The PowerGrid execution cycle 7](#_Toc357190402)

[Chapter 4 – Implementation 8](#_Toc357190403)

[4.1 – The JNI module 8](#_Toc357190404)

[4.1.1 – Problems with Java’s JNI 8](#_Toc357190405)

[4.1.2 – Mimicking Java’s reflection engine in C++ 8](#_Toc357190406)

[4.1.3 – Summary of JNI classes in PowerGrid 8](#_Toc357190407)

[4.2 – The Monitor module 9](#_Toc357190408)

[4.3 – The Caching module 10](#_Toc357190409)

[4.4 – The GUI module 10](#_Toc357190410)

[4.4.1 – Requirements for the PowerGrid user interface 10](#_Toc357190411)

[4.4.2 – Implementation of the GUI requirements 10](#_Toc357190412)

[4.4.3 – Summary of GUI components in PowerGrid 10](#_Toc357190413)

[4.5 – The AI module 10](#_Toc357190414)

[4.6 – The Injection module 10](#_Toc357190415)

[4.7 – Summary of modules and their relations 10](#_Toc357190416)

# Chapter 1 – Introduction

Many bot clients already exist for the MMORPG Runescape. Each with their own benefits and downsides. However, almost all bot clients have one thing in common: they offer little to no functionality by themselves, and only provide that which is required to perform basic operations on the bot. A large downside of this is, that it results in bot clients not being able to operate in a dynamic environment, and such clients often also do not provide tools to recognise certain object types, leading to manually checking and comparing id values over and over again.

A solution to the abovementioned problem would be, to provide the missing tools as an extension to an existing bot client, but such an approach also has a price in the form of a very high memory footprint, since much data needs to be cached locally too. This is mainly because almost all clients are closed-source, and getting the necessary data often requires an impractical detour through a variety of method calls.

So the best solution would be to create a bot client from scratch, focussed on solving the above problems and to optimize user experience in this way. However, there is another problem a large number of bot client cope with: It is only possible to play the game in fixed resolutions, because of the way the widget system in these bot clients work. Since most bot clients essentially operate on the same core, these bot clients all deal with the same problems, and as such an entirely new and revolutionary approach must be taken in order to avoid the same problems and issues.

In order to optimize speed and to make it easier to remain undetected as a botting client, it would be a good design choice to implement the project in a programming language other than Java. As such, a programming language was chosen that resembles Java in many ways, but is a lot faster and more dynamic. This chosen programming language was C++.

Putting all the above together, we present you PowerGrid. A revolutionary, open-source bot client that intends to make life easier for everyone by providing functionality that automatically classifies and stores the data from the runescape world in native (C++) objects. Because of this caching behavior, it suddenly becomes possible to plan routes across the entire world, or find the nearest object matching certain criteria even if such an object is far away.

The final goal of PowerGrid is to provide users with a tool that can play Runescape completely by itself, automatically deciding on the tasks to perform based on changes in the environment. PowerGrid will even be able to perform abstract tasks like leveling a certain skill to a certain level, or making a certain amount of money. PowerGrid should then automatically decide on the concrete tasks (what methods of money-making to use, or what method to use to train the requested skill) by effiency.

Please do note, however, that the abovementioned behavior is merely an indication, and exact functionality may change over time.

# Chapter 2 – Overview of PowerGrid

This chapter describes the two main components of PowerGrid and how these contribute to each other to make PowerGrid work the way it does. PowerGrid basically consists of the following main components: the Artficial Intelligence and the communication between the Runescape client and PowerGrid. A description of the Artificial Intelligence module is provided in paragraph 2.1, whereas the communitication with the Runescape client is described in paragraph 2.2.

## 2.1 – Artificial Intelligence as a way of playing Runescape automatically

PowerGrid has a sophisticated Artificial Intelligence (AI) module, which it uses to decide on actions from the Runescape environment. Because PowerGrid uses such an AI module, it is possible for PowerGrid to operate even in unknown environments and behave dynamically based on the state of the environment and the goals set by the user.

The goals the user can provide to the AI are high-level descriptions of tasks, such as to reach a certain location, or to acquire a certain amount of experience for a skill. The AI combines this information with the environment to decide on an action to perform.

For example, reaching a certain location requires the AI to execute a pathfinding algorithm and then follow the path this algorithm produced. As the AI navigates this path, it should still monitor the environment because some things are prone to change. Imagine that the path leads through a door but that door is closed. The AI should then automatically know that the door should be opened before continuing its path.

For gaining experience in various skills the amount of elements to monitor is even bigger, and the approach varies greatly between different skills. Training a skill such as crafting, for example, requires a completely different approach then training a skill such as woodcutting. Therefore an AI is required that can somehow decide on the action based on its goals.

## 2.2 – Communication with the Runescape client

The AI by itself cannot do much good if there is no connection to the Runescape environment itself. As such, a system is needed that ensures a correctly set up connection to the Runescape client. Since Runescape runs on Java, we need to communicate with the Java Virtual Machine (that runs the Runescape client) first. Java provides a system for this called JNI (Java Native Interface). However, the functionality provided by it is not sufficient, and JNI only understands very low-level instructions. As such, Powergrid contains a system that mediates between instructions and information the AI can understand, and the low-level instructions that the JNI can understand. This bridges the gap between C++, the language PowerGrid is written in, and Java, the language of the Runescape client.

# Chapter 3 – Structural overview

In this chapter, an overview is given of the basic structural components of the project and the relations between these components. First, in paragraph 3.1, a diagram is provided that shows the structure of PowerGrid while in action. This is followed by a summary of each module shown in the diagram (paragraph 3.2). Lastly, a description is given about PowerGrid’s execution cycle in paragraph 3.3

## 3.1 – Structural diagram of PowerGrid

The below diagram illustrates the structural diagram of PowerGrid. The diagram shows the different modules in PowerGrid and how these modules interact with each other.

The modules are split up in two groups: The Java Virtual Machine and the PowerGrid client. The Java Virtual Machine contains the running Runescape client as well as access to the JNI functions that PowerGrid uses. PowerGrid itself contains various modules to monitor, process and inject actions and information from and to the Java Virtual Machine.



Figure 3.1

## 3.2 – Summary of each of the modules in PowerGrid

Below a summary is provided of each of the modules in PowerGrid. For a diagram of how these modules fit together, please refer to figure 3.1.

**JNI module**

The JNI module handles basic interaction with the JVM. Can read from, and send data to the running Java Virtual Machine through JNI. Unlike JNI by itself, this module performs safety-checks to ensure the data put into the Java Virtual Machine is valid. JNI by itself does not do this, so without using an intermediate module to perform these checks, it becomes possible to send invalid data into the JVM that potentially crashes the entire program.

**Monitor module**

The monitor module handles incoming raw data from the JNI module and parses it as recognisable (native) objects. These objects are then sent to the cache module for quick retrieval. The monitor also provides functionality to the AI module that allows it to read real-time statistics from the Runescape world.

**Caching module**

**The caching module stores and manages the data from the Runescape environment for quick access. This module consists of a variety of caches with different structures for different purposes, and each cache has its own policy of storing and retrieving the data it contains. The caching module also allows storing and loading custom data from other modules. This data is then kept in a separate cache, also for quick lookup.**

**GUI module**

The GUI module handles parsing and configuring the AI module with the information provided by the user. Also, the GUI can check the status of each individual module and report its status to the user. This provides a overview of the state of the entire program.

**AI module**

The AI module decides on an action based on parameters from the GUI module and information from the Monitor module. This is the core of the bot itself that makes the decisions. The AI module may give directives to the Monitor module and the cache module for managing their data. This helps, because the AI module usually knows the semantics of the data it’s working with. By allowing the AI to provide these directives, it becomes possible to use the memory more efficiently.

**Injection module**

The injection module translates an action from the AI into one or more events that can be injected into the JVM. These event objects are then passed on to the JNI module that will put them into the JVM. Basically, the injection module provides more high-level functions to the AI for performing common actions, and the injection module will convert these actions into Java event objects which are then passed into the JVM.

**JNI core**

The JNI core is the core of the Java Virtual Machine’s native API. It manages the basic functions of the running Java Environment. It also provides functions to the JNI module for accessing the running Runescape environment. Since this is a part of the JVM itself, PowerGrid has no control over its behavior.

**RS client**

The running Runescape client. Interaction with this environment can only be done using reflection functionality provided by the JNI core, which in turn can be accessed through the JNI module in PowerGrid. The RS client itself connects to the RS servers, allowing the player to perform actions. These actions are simulated by PowerGrid by injecting appropriate event objects into the RS client by pushing the required data into the JNI core through PowerGrid’s own JNI module.

## 3.3 – The PowerGrid execution cycle

PowerGrid has four modules (JNI, Monitor, AI and Injection modules) that pass information to each other in a cycle. This is the contiuous cycle that PowerGrid runs in. An overview of how this cycle works is given below.

First, information is fetched through JNI. This information is parsed by the monitor (and then stored in the caches). Then, the AI module retrieves this information and uses it to decide on an action. The AI may also offer directives to the monitor on what to monitor, what to remember, and what to delete. This improves the hit ratio of cached data. When the AI decided on an action, it is sent to the injector module, which converts (breaks up) the action into separate JNI calls and sends these commands to the JNI module that will execute them on the Java Virtual Machine.

More technically correct, though, would be to say that the AI module runs in a loop that requests the data it needs from the monitor module which in turn first performs the required JNI calls to fetch the required data (or fetches it from the cache, if this is feasible).

The GUI module provides end users with the possibility to modify how the AI works. When a setting is changed in the GUI, this change is propagated to the AI module, which will change its behavior accordingly.

# Chapter 4 – Implementation

This chapter will provide a more in-depth description of how each module is implemented. The first paragraph provides an analysis of the JNI module (4.1). After that the Monitor module will be discussed (4.2). Following the Monitor module, the implementation details of the Caching module are discussed (4.3). In paragraph 4.4, the design of the GUI module is explained. The next paragraph (4.5) describes the AI module. In the second-to-last paragraph the Injection module will be explained (4.6). Finally a summary is provided indicating the relations between the different modules.

## 4.1 – The JNI module

The JNI module provides functions that allow communication with the JNI in a safer way than JNI provides by itself, and the JNI module in PowerGrid also provides functions for commonly used operations that are not provided by the JNI itself. The problems of Java’s own JNI is described in paragraph 4.1.1, while the solution PowerGrid implements to counter these problems is depicted in paragraph 4.1.2. At last a summary of the revelant classes is given to give an impression of the design of the implementation of the JNI module in paragraph 4.1.3.

### 4.1.1 – Problems with Java’s JNI

Java’s own JNI was originally made for plain C, and has some syntactic additions for C++. However, Java’s JNI does not make efficient use of C++ objects and classes, even though Java is also an object-oriented language. To bridge this gap between C++ and Java, the JNI module of PowerGrid provides classes that model Java components such as classes and methods by representing them as C++ objects. This allows for a more fluent use of JNI and also provides a more solid foundation for the more complex JNI calls. The JNI module also provides satefy checking for the method invocations, something Java’s JNI also did not provide. As such, using a wrapper around Java’s JNI to do the safety checking increases the robustness of the entire program and ensures that the Java Virtual Machine does not crash or hang due to invalid method calls.

### 4.1.2 – Mimicking Java’s reflection engine in C++

The design of the JNI module mimicks the design of the Java reflection engine provided in the JVM itself. This is done because the Java reflection engine works as expected, and using a similar approach in C++ makes it easier for other developers to start using the JNI module’s functionality in C++. Also, Java’s reflection engine is fully object-oriented, just like Java itself, and that is what makes the reflection engine perform well enough in most situations. By mimicking the classes in the Java reflect package in C++, it becomes possible to gain access to JNI functionality in a more native way than when using Java’s own (functional) JNI.

### 4.1.3 – Summary of JNI classes in PowerGrid

The JNI module in PowerGrid provides a variety of classes quite similar to the reflection classes in Java. The relevant classes are mentioned below:

* JNIClass
  + Represents a class in the Java environment.
  + Contains functions to locate and retrieve methods and properties of the class it represents.
* JNIMethod
  + Represents a method (static or non-static) in the Java environment
  + Contains functions to check the return type as well as the parameter types
* JNIValue
  + Represents any value (primitive or object) from the Java environment.
  + Different from JNI’s own jvalue type, JNIValues know what type they are.
* JavaEnv
  + A general access point to the running Java environment.
  + Can be used instead of the other JNI classes for class and method retrieval, although using JNIClass and JNIMethod is usually more readable and easier to understand.
* OngoingInvocation
  + Represents an ongoing method call. An instance of this class is created for each invocation.
  + Supports executing the same method multiple times without having to recollect the required values.

The JNIClass and JNIMethod classes above provide roughly the same functionality as the Class and Method classes from Java. JNIClass actually holds a reference to the Class object in the Java environment to request specific information.

All classes and all functionality directly related to the JNI module resides on the jni namespace.

## 4.2 – The Monitor module

The Monitor module collects relevant information from the Java Virtual Machine and stores it in the caches. At the same time, the information retrieved from the JVM is parsed and stored in C++ objects according to the type of information.

The classes modelling the Runescape environment are all stored on the world namespace. The monitor itself, as well as everything related to it, is stored on the monitor namespace.

The Monitor only collects world and collision data by default. Loading of other types of data is done on a need-to-know basis. This is to ensure no obsolete data is loaded and no obsolete information is cached. For example, wigets are only loaded once a certain widget is required that is not already cached. In that case, the monitor module will check the Runescape environment for the required widget and caches it, after which it will return a reference to the C++ object representing the widget.

The monitor module can also take directives on what to cache and what not. This is to optimize the monitor by only loading what is needed and discarding information that is no longer needed. In this way it is possible to optimize the memory usage.

## 4.3 – The Caching module

The caching module is responsible for storing information retrieved from the Runescape environment for quick access. This module is essential to ensure that the AI module has access to the information it needs. In paragraph 4.3.1, an overview is given of the data from the Runescape world that will be cached in this module, along with an description of each data type. In paragraph 4.3.2, the different methods of caching for the different types are mentioned and explained. A summary providing an explanation of how the different caches in this module benefit the other modules is given in paragraph 4.3.3.

### 4.3.1 – Objects suited to be cached

Runescape contains many different types of objects and data. Some types of data are very important for PowerGrid, while others are not. Requesting important data from the Runescape enviroment repeatedly causes massive delay, and as such should be prevented as much as possible. For this problem, PowerGrid contains a variety of caches to ‘remember’ the information it requested, thus eliminating the need to request it again. However, caching everything that Runescape provides results in a large memory footprint, which also reduces performance. The objects that are most suited to be cached by PowerGrid are mentioned below along with an explanation as to why it is important to cache this specific object type.

* The locations of walls and other types of barriers
  + This includes the locations and orientations, but excludes further details about the texture or height of the wall or barrier.
  + Caching this type of information is important for PowerGrid to be able to look up the locations of walls when computing a path between two points in the Runescape world. Also, if this information is not cached, PowerGrid cannot compute paths that go far away from the player’s location, because Runescape unloads far away areas, making them unavailable for reading data from them.
* Definitions of objects that can be interacted with.
  + This includes doors and gates, but also trees, ores, shortcuts and so on. However, objects that the player would normally never interact with, such as wall and floor tiles will not be cached because they are not interesting for PowerGrid.
  + The specific interaction options can also be cached if these are known to PowerGrid, allows
  + Caching this type of information allows PowerGrid to quickly look up information about objects in the neighbourhood of the player, and as such can compute the nearest object of a certain type, or identify certain situations based on the objects there are.

### 4.3.2 – Caching method for different object types

…

### 4.3.3 – Integration of the caching module into PowerGrid

…

## 4.4 – The GUI module

GUI stands for Graphical User Interface. Basically this refers to the windows that will be visible to the end user. The GUI module provides access to most of the functions of PowerGrid through such a window. The design of the GUI should meet certain parameters in order for the end users to find their way through the GUI. These parameters are identified in paragraph 4.4.1. In the next paragraph (4.4.2), an overview is given as to how the parameters discussed in paragraph 4.4.1 are reflected in the design of the GUI. Next, the different parts of the GUI are explained and shown in paragraph 4.4.3.

### 4.4.1 – Requirements for the PowerGrid user interface

…

### 4.4.2 – Implementation of the GUI requirements

…

### 4.4.3 – Summary of GUI components in PowerGrid

…

## 4.5 – The AI module

…

## 4.6 – The Injection module

…

## 4.7 – Summary of modules and their relations

…