University of Westminster

Final Year Project

LIEBERT server monitoring platform

Requirements report

Martin Kukura

w1499573

December 2016

Supervisor: Girish Lukka

ECSC697 - Computer Science And Software Engineering Project

Contents

List of Figures

1 Statement of purpose

1.1 Synopsis

With the evolution of computer systems in the recent years, the cost of processing power has been dropping rapidly. This allows everyone from big corporations to ordinary people to possess vast computing resources, be it in the form of a regular computer, Raspberry Pi or a virtual server instance running in the cloud provided by the likes of *Amazon*. Most companies, as well as many tech savvy consumers utilize these resources to run long-lived continuous tasks such as video encoding or data processing. Without proper monitoring tools, large amounts of computing resources can be wasted, as many instances are provisioned with needlessly large amounts of CPU and RAM due to their relatively low cost. Looking at the bigger picture however, under-utilizing, especially cloud resources, becomes a costly nightmare when the operation being supported scales unexpectedly. On the other hand, many instances are over-utilized and would be able to complete their tasks more efficiently if they were properly dimensioned.

Server monitoring tools evolved from the need to track system performance. They not only allow system administrators to detect over and under-utilized systems, but also monitor system health. Properly setup advanced monitoring system will not only raise an alert when an issue has arised, but also warn of any impending issues (e.g. disk space running out). There are many monitoring systems out there used to fulfil different requirements. From large complex all-in-one solutions such as *Nagios*, through easy to setup tools like *Munin*, to ones that are focused on performance akin to *collectd*.

LIEBERT's main goal will be to end up in the last category. There is a vast array of system monitoring tools, however very few of them focus on performance and memory footprint and instead sacrifice these parameters in favour of more features, many of which are, more often than not, unused. **LIEBERT** will be written from the ground up to offer superior performance and avoid hogging up the system, and will instead sacrifice features if required.

1.2 Project scope

LIEBERT will be pretty limited in scope, however it will make up for that fact by going more in-depth in its core features. The minimal basic version will deliver a highly

performant monitoring solution with minimal resource usage. There will be two parts (applications), one will be the endpoint metric gatherer that will reside on each monitored node, the other one will be a *controller* to which all the endpoint applications will send their data over the network to store the metrics to persistent storage. The core monitored metrics will be CPU, RAM, HDD and network utilisation. The core storage methods will be plaintext and RRD (round-robin database). *LIEBERT*'s focus is on performant metric retrieval, therefore there will not be any sort of dashboard with fancy charts to display the data. One of the core storage options, RRD, is however designed for easy storage and visualization of timeseries data, and it will therefore be very easy to create visualizations without the need for external software (except for *RRDtool* itself). RRD is also widely used in the industry to store this sort of data, and will therefore be familiar to most people dealing with **LIEBERT** (Oetiker 2014). If time restrictions allow for further development after the initial version is finished, there are numerous extra features that can be added, most notably a plugin system that will allow for collection of any user-supplied metric. Note however that the performance of the plugin system would be considerably lower than what core **LIEBERT** will be able to offer. The initial version of the package will aim for compatibility with the Linux operating system, because it is the most widely used server operating system (W3Techs 2016)[1].

1.3 Goals

The primary goal of this project is a learning experience. Creating **LIEBERT** will require good understanding of all the possible programming languages that could be used to ensure its performance, as well as proper understanding of the chosen programming language to ensure best practices leading to higher performance. It will also require deeper understanding of the targeted operating systems (basic version being aimed at Linux) to minimize resource requirements for the metric gathering itself, as well as networking to enable cross-application communication through the network.

Secondary goal is delivery of the actual monitoring system, with all of the basic, and potentially some of the extra, features described. Altough the software package itself will be a proof-of-concept, maximum care will be taken to deliver as production-ready

[1]Non-public facing server OS market share is impossible to determine, however from personal experience and blogs of leading technology companies Linux seems to be the most popular

product as possible.

1.4 Stakeholders

Due to the relatively closed and autonomous nature of the system, there aren't many direct stakeholders. However, indirectly the system can affect a broad range of entities, as the operation of a monitoring suite and thus healthy state of the underlying system (or rather an unhealthy state of the underlying system) will affect any user consuming services running on top of it. The system will therefore have 2 major stakeholders - the system administrators directly interacting with our monitoring solution, and consumers that depend on the system that is being monitored.

2 Context diagram

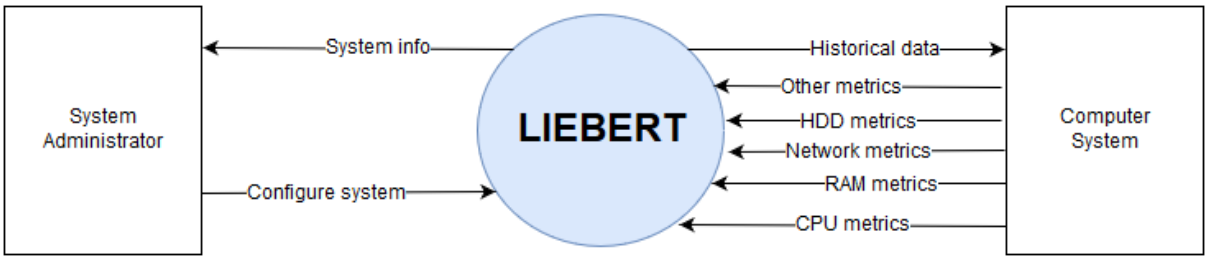

Figure 1: The context diagram

The context diagram provided in Figure 1 displays the information flow to and from the system. As said before, the autonomous nature of our system limits its reach, therefore information will only be exchanged with the computer systems that are running the application, and with potential system administrators that are either engaging in maintenance tasks or are checking on the status of the application itself. This type of diagram is also known as the *Data Flow Diagram*.

3 Requirements

3.1 Similar software

To better understand the requirements for system monitoring software, a survey of some of the most popular existing solutions was performed. Exploring their features and promi-

nent characteristics allowed for a more precise specification on what shall be the focus of this project.

3.1.1 Nagios

Nagios (`https://www.nagios.org/`), '*the industry standard in IT infrastructure monitoring*' as it proclaims itself, is an extremely extensible monitoring solution. It doesn't come with any actual data gathering capabilities out of the box, however its plugin architecture allows users to write extensions to monitor anything from system metrics to complex applications. Nagios offers excellent alerting and early warning capabilities atop the user supplied data as well as logging, however its configuration is rather complex. It also comes with a basic but very powerful web dashboard allowing control over the software. The storage backend of *Nagios* is handled by *RRDtool*.

3.1.2 Munin

Munin (`http://munin-monitoring.org/`) is an alerting solution similar to *Nagios*, however its main focus is on ease of use. It provides near plug-and-play capabilities and simple configuration and plugin creation. *Munin* uses a master - nodes architecture where the master node polls all child nodes for data. Plugins can be written in any language and only require that they be executable and their text output matches *Munin* specifications. It also offers a simple web interface, and stores it's data using *RRDtool*.

3.1.3 collectd

Collectd (`https://collectd.org/`) is a minimal filtering and relaying Unix daemon with extensible architecture. Core of *collectd* only provides high performance 'routing' capabilities between various input and output plugins. By default it ships with plugins for collecting cpu, network, load and memory data, as well as output plugins for CSV, RRD, and network relaying. *Collectd* plugins have to be written in C and are loaded as shared modules - this makes them highly efficient and performant, however makes them incredibly difficult to make for someone who is not well-versed in C.

3.1.4 Feature table

To better understand the differences between the aforementioned monitoring solutions, a feature comparison table has been created. It shows clear difference between the features, and allows cherry-picking characteristics of each solution that should be part of **LIEBERT**.

Features	**Nagios**	**Munin**	**collectd**	**LIEBERT**
Extensible	Yes	Yes	Yes	Extra feature
Easy to extend	No	Yes	No	Extra feature - Yes
Core monitoring included	No	Yes	Yes	Yes
Web dashboard	Yes	Yes	No	No
Easy to use	No	Yes	No	Yes
Focused on performance	No	No	Yes	Yes
Cross-platform	No	No	No	Partially

Extensibility Extensibility is not planned as an essential part of **LIEBERT**. This is due to the fact that important metric collectors will be included as part of the system core, and therefore optimized. Extensibility, if implemented will be working on the same principle as *Munin*, where **LIEBERT** will call user supplied executables that will provide data in specific format recognizable by the system.

Web dashboard The aim of this project is to create a highly performant system with minimal memory footprint. In order to achieve that, certain features need to be sacrificed. Web accessibility (and advanced user interaction in general) is not included in the project as such features would most probably be more resource intensive than **LIEBERT** itself.

Easy to use Ease of use is probably the second most important requirement after performance. This will enable users to make use of high performance monitoring without the steep learning curve of tools such as *collectd*. **LIEBERT** will strive to provide an easy setup and configuration akin to *Munin*, where users should be able to get basic monitoring started just by changing a few lines in a configuration file.

Performance Primary requirement of this project is performance. Features will be sacrificed to provide high performance and small memory footprint (compared to other

full-featured solutions). The system will be implemented in a low level language to ensure no performance overhead arising from complex modern languages.

Cross-platform Altough **LIEBERT** will not be intentionally developed for cross-platform compatibility, it will be very easy to port to other platforms due to the chosen programming language (read more in subsection 3.2). In fact, most of it will be written in cross-platform compatible code and the only parts that will need rewriting are the gatherer modules which utilize native OS resources.

3.2 Programming language

In order to maximize performance of the system, care needs to be taken when choosing the language it will be implemented in. Choice of an unsuitable language might effectively 'undo' any optimizations done within the code, as the language's execution environment can be a source of large resource utilization. This is especially true for hybrid *JIT* compiled languages such as Java or C#. In the following subsections the results of surveying for potentially suitable languages can be found.

3.2.1 C / C++

C, or "The closest you can get to assembler"[2], is a low-level, immensly powerful and versatile language with practically zero overhead. It enables programmers to create highly optimized applications, however its 'freedom' can (and will) lead to negative side-effects such as no memory safety. *C++* is an evolution of *C* that added modern features on top of the old language, such as objects. These two languages would be suitable for the project, however their inherent memory unsafety can be a problem.

3.2.2 Go

Go is a modern systems programming language created by *Google*. It has seen surge of popularity since its public release in 2009, especially in recent years. It offers similar low-level system access as the *C* family, however was designed for concurrent applications and therefore contains built-in concurrency and memory safety features. It also features

[2]Quote from an old coworker

a garbage collector found in many modern high-level languages. *Go* would be a fantastic fit for **LIEBERT**, however its use of the garbage collector is a downside.

3.2.3 Rust

Mozilla's very own programming language *Rust*, publicly announced in 2010, has seen recent surge in popularity in the same way as *Go*. Rust is essentially a very low-level language akin to *C*, without any additional overhead of a garbage collector that *Go* has. It is also focused on concurrency and memory safety, and additionally contains so called *zero-cost abstractions*. *Zero-cost abstractions* are compile-time features that allow the language to have powerful features often found in higher level languages, however with practically zero overhead. This is accomplished by the *Rust* compiler that performs complicated analysis and optimization, that's additionally able to perform actions such as unrolling highly complex loops (van Asseldonk 2016). Notable features of the language is the lack of objects, native support for creating and distributing libraries, memory safety (mostly accomplished by the so called *borrow checker*), type inference, and its focus on concurrency.

3.2.4 Python

Python is the most popular interpreted programming language (TIOBE 2016). Years of development surprisingly allowed it to mature into a powerful and incredibly fast interpreter. It doesn't have a set goal or focus in mind, and is instead a general programming language capable of producing nearly any sort of application. *Python* provides convenience features found in many high-level programming languages, such as dynamic typing or a garbage collector. Altough it is considerably fast, it's still going to suffer from overhead compared to low-level compiled programming languages.

3.2.5 Feature table

Features	**C / C++**	**Go**	**Rust**	**Python**	**Desired?**
Low-level	Yes	Yes	Yes	No	Yes
Garbage collector	No	Yes	No	Yes	No
Dynamic typing	No	No	No	Yes	No
Conveniece features	No	Somewhat	Yes	Yes	Yes
Runtime overhead	No	Small	No	Large	No
Cross-platform	Yes	Yes	Yes	Yes	Yes
Memory safety	No	Yes	Yes	Partially	Yes

3.2.6 Conclusion

Programming language The implementation language chosen for this project is **Rust**. It features many high-level language convenience features through its *zero-cost abstraction* system, focuses on concurrency, and does not come with any added overhead of the likes of a garbage collector.

Development methodology Traditional methodologies such as *waterfall* are becoming less and less popular in modern software development. Altough still present, they are often remnants of the old times in large corporations with established workflows. Evolving progressive teams and new companies alike are usually not in favour of such methodologies, and instead prefer modern ones such as *Agile* to enable rapid feature development, leveraging continuos integration and deployment to push new releases into production as quickly as possible. This project will embrace the *agile-scrum* methodology, focusing on design, development and release of features in small batches. This is relevant to the Gantt chart found in section 6, which indicates a longer running design and development cycle in the first phase of the project that focuses on delivering a minimal working version, with short-release feature development cycles in the second phase.

3.3 Functional requirements

The system will consist of 2 applications, an *Agent* and a *Controller*. Each has its own set of requirements.

3.3.1 Agent

	The application shall collect CPU state metrics
R1	The application should extract CPU load counters from the underlying operating system with as much granularity as possible. These should at least include user and system load.
	Importance: **essential**

	The application shall collect RAM state metrics
R2	The application should extract current RAM usage from the underlying operating system. This should include utilized RAM as a minimum.
	Importance: **essential**

	The application shall collect HDD state metrics
R3	The application should extract current disk usage from the underlying operating system. This should include the primary partition as a minimum.
	Importance: **essential**

	The application shall collect network interface metrics
R4	The application should extract network throughput and interface information from the underlying operating system.
	Importance: **essential**

	The application shall store collected data in an internal buffer
R5	The application should accumulate gathered metrics in an internal structure until they are ready to be sent over the network to the connected *Controller* instance.
	Importance: **essential**

	The application shall connect to a controller instance
R6	The application should establish a TCP link with a relevant controller on the network. This connection will be used to forward metric data and receive control commands.
	Importance: **desired**

R7	**The application shall forward the stored data to a connected Controller instance**
	The application should periodically forward accumulated metric data to a connected *Controller* instance. In case the *Controller* is unavailable, it should keep accumulating data internally until the *Controller* becomes available.
	Importance: **desired**

	The application shall be extendable
R8	Users should be able to extend the gathering capabilities with custom executables providing output recognizable by the system.
	Importance: **luxury**

	The user shall be able to configure all parts of the application
R9	The application should be configurable in all areas where it makes sense. This would include network, gathering, and storage configuration at minimum.
	Importance: **desirable**

	The application shall log important runtime information
R10	The application should print *debug* / *warning* / *info* messages either to to *stdout* or a file.
	Importance: **essential**

	The user shall be able to check the application's current status
R11	There should be a way to 'tap into' the application and review its current status. This should probably be implemented as an additional protocol over the network.
	Importance: **luxury**

3.3.2 Controller

	The application shall receive connections from *Agents*
R12	The application should be able to establish TCP connections with *Agents* on the network.
	Importance: **essential**

	The application shall authenticate connecting *Agents*
R13	The application should implement security protocols to only allow connections from authorized *Agents*.
	Importance: **luxury**

	The application shall receive data from connected *Agents*
R14	The application should be able to handle historical metric data received from all connected *Agents*
	Importance: **essential**

	The application shall store data into persistent storage
R15	The application should store the historical metric data received from *Agents* to persistent storage (e.g. hard drive) in one of the supported formats.
	Importance: **essential**

	The application shall store data using plaintext format
R16	The application should be able to store data to persistent storage using a suitable plaintext format.
	Importance: **essential**

	The application shall store data using *RRD* format
R17	The application should be able to store data to persistent storage using the format supported by *RRDtool*.
	Importance: **desired**

	The application shall store data into a database
R18	The application should be able to store data into a relational database such as *SQLite* or *MySQL*.
	Importance: **luxury**

	The application shall be extendable
R19	The output formats of the application should be extendable using user supplied executables that accept a predefined format of the system.
	Importance: **luxury**

	The user shall be able to configure all parts of the application
R20	The application should be configurable in all areas where it makes sense. This would include network and storage configuration at minimum.
	Importance: **desired**

	The user shall be able to check the application's current status
R21	There should be a way to 'tap into' the application and review its current status. This should probably be implemented as an additional protocol over the network.
	Importance: **luxury**

	The user shall be able to relay commands to all connected *Agents*
R22	The user should be able to utilize the *Controller's* connections to relay commands to connected *Agent* instances. This requirement is closely related to '*The user shall be able to check the application's current status*'.
	Importance: **luxury**

	The application shall log important runtime information
R23	The application should print *debug* / *warning* / *info* messages either to to *stdout* or a file.
	Importance: **essential**

3.4 Non-functional requirements

	The system shall be built with focus on performance
NF1	Maximum care should be taken when designing and developing the project to ensure maximum performance of the end result.
	Importance: **essential**

	The system shall be written in the *Rust* programming language
NF2	After careful consideration of multiple programming languages (discussed in subsection 3.2) the *Rust* programming language has been chosen due to its best feature-to-performance ratio.
	Importance: **essential**

	The system shall target the Linux operating system
NF3	As discussed in subsection 1.2, primary support for Linux has been chosen due to its popularity with servers.
	Importance: **essential**

	The various system parts shall communicate through a network
NF4	Due to the nature of *Agent* and *Controller*, network communication will be required to utilize all the features of the system.
	Importance: **essential**

	The system shall be built with focus on ease of use
NF5	The system should be as easy to use as possible, ideally being able to be utilized with only a few changed lines in a configuration file.
	Importance: **essential**

	The system shall be secure
NF6	The system should make use of good modern security practices to not open the network it's residing on to unauthorized entities and prevent tampering with its data.
	Importance: **luxury**

	The system shall have minimal memory footprint
NF7	Due to the system's focus on performance, it should also consume minimum resource on the system it is running on.
	Importance: **essential**

	The system's data shall be accurate
NF8	Maximum care should be taken to ensure the data retrieved by the system accurately represents the real state.
	Importance: **essential**

4 Use cases

Closed and autonomous nature of the system makes regular use case modelling rather hard, however use case modelling still remains important to uncover the various flows within an application. Therefore, instead of focusing on traditional user-oriented use cases, this section will also contain internal system flows such as metric gathering or network communication.

4.1 Use case diagram

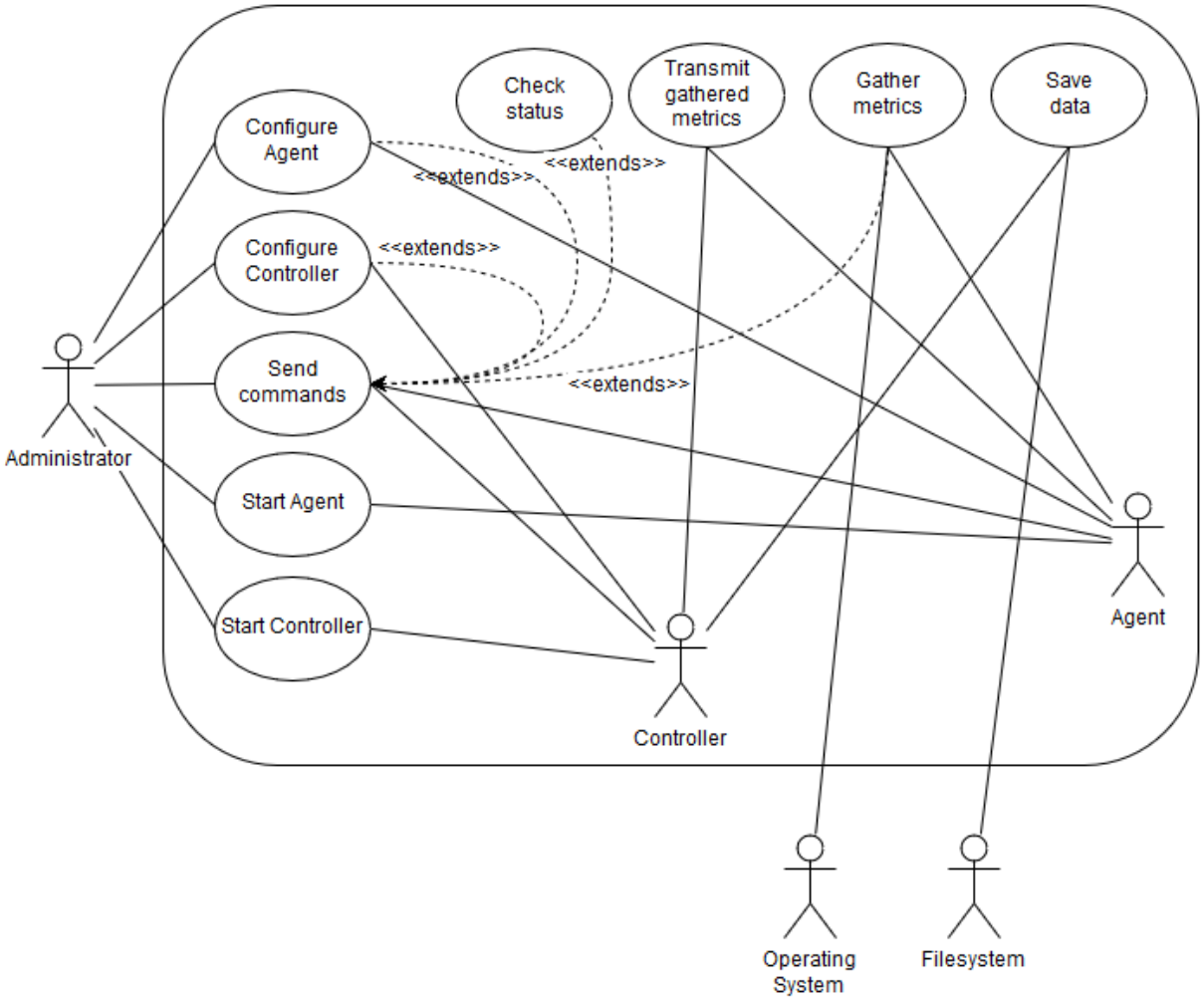

Figure 2: The use case diagram

4.2 Use case specifications

UC_1	
Use case	Start *Agent*
Description	Administrator attempts to start the *Agent* application using the operating system's appropriate method
Actors	Administrator, *Agent*
Pre-conditions	Application is present on target machine and target machine is running a supported platform
Flow of events	
Normal flow	
1. Administrator launches application 2. Application parses command line options 3. Application loads and parses relevant configuration file 4. Application attempts to connect to a configured controller 5. Application starts all metric gathering subsystems 6. Application enters normal run loop	
Alternative flow	
• N/A	

Exceptions	• If in step 3 the application cannot find a relevant configuration file it exits with an error • If in step 3 the application cannot successfuly parse the configuration file it exits with an error • If maximum retries are exceeded in step 4 the application exits with an error • If a subsystem fails to load in step 5, the application moves it to disabled state and continues
Post-conditions	The application is successfuly started and running

UC_2	
Use case	Start *Controller*
Description	Administrator attempts to start the *Controller* application using the operating system's appropriate method
Actors	Administrator, *Controller*
Pre-conditions	Application is present on target machine and target machine is running a supported platform
Flow of events	
Normal flow	

1. Administrator launches application
2. Application parses command line options
3. Application loads and parses relevant configuration file
4. Application starts all configured storage subsystems
5. Application starts listening for *Agent* connections
6. Application starts listening for control connections
7. Application enters normal run loop

Alternative flow

- N/A

Exceptions	• If in step 3 the application cannot find a relevant configuration file it exits with an error • If in step 3 the application cannot successfuly parse the configuration file it exits with an error • If a subsystem fails to load in step 4, the application moves it to disabled state and continues; if all subsystems fail to load, the application exits with an error • If the application fails to initialize a TCP listener in step 5 or 6, it exits with an error
Post-conditions	The application is successfuly started and running

UC_3	
Use case	Configure *Agent*
Description	The administrator attempts to configure the *Agent* application
Actors	Administrator, *Agent*, *Controller*
Pre-conditions	The *Agent* application has been successfuly started; the *Controller* application has been successfuly started and is connected to the relevant *Agent* instance (for alternative flow)
Flow of events	
Normal flow	
1. Administrator opens an existing or creates a new configuration file 2. Administrator edits the configuration file 3. Administrator stops the *Agent* application (if running) 4. Administrator starts the *Agent* application - continues in **UC_1**	
Alternative flow	
1. Administrator successfuly completes **UC_5** 2. Administrator sends configuration command to *Controller* 3. *Controller* relays configuration command to the relevant *Agent* 4. *Agent* performs a run-time reconfiguration based on the command received	
Exceptions	• If the command received by *Agent* in *Alternative flow step 3* is invalid, it is discarded and a warning is emitted

Post-conditions	The *Agent* application is successfuly running with the new configuration

UC_4	
Use case	Configure *Controller*
Description	The administrator attempts to configure the *Controller* application
Actors	Administrator, *Controller*
Pre-conditions	The *Controller* application has been successfuly started
Flow of events	
Normal flow	
1. Administrator opens an existing or creates a new configuration file 2. Administrator edits the configuration file 3. Administrator stops the *Controller* application (if running) 4. Administrator starts the *Controller* application - continues in **UC_2**	
Alternative flow	
1. Administrator successfuly completes **UC_5** 2. Administrator sends configuration command to *Controller* 3. *Controller* performs run-time reconfiguration based on the command received	

Exceptions	• If the command received by the *Controller* in *Alternative flow step 2* is invalid, it is discarded and a warning is emitted
Post-conditions	The *Controller* application is successfuly running with the new configuration

UC_5	
Use case	Send commands
Description	Administrator attempts to control the running *Controller* or *Agent* instances
Actors	Administrator, *Controller*, *Agent*
Pre-conditions	Administrator has correct version of the **LIEBERT** CLI tool; *Controller* has been successfuly started; *Agent* has been successfuly started (for alternative flow)
Flow of events	
Normal flow	
1. Administrator opens the **LIEBERT** CLI application and establishes a connection with a *Controller* 2. Administrator sends commands to controller using the CLI application	
Alternative flow	

1. Administrator opens the **LIEBERT** CLI application and establishes a connection with a *Controller*
2. Administrator sends a command telling the *Controller* to switch to *Agent* control model
3. Administrator sends commands to controller which get relayed to the relevant *Agent* instance

Exceptions	• If the connection fails in *Normal flow step 1* or *Alternative flow step 1*, the CLI tool exits with an error • If the *Agent* selected in *Alternative flow step 2* is invalid, *Controller* responds with an error • If the command sent in *Normal flow step 2* or *Alternative flow step 3* is invalid, it is discarded by the receiving node
Post-conditions	Administrator has established a connection with a *Controller* using the CLI tool and is able to send commands

UC_6	
Use case	Check status
Description	Administrator attempts to check the status of a specific *Agent* or *Controller* instance
Actors	Administrator, *Controller*, *Agent*
Pre-conditions	Relevant *Controller* and *Agent* instances have been successfuly started and are running
Flow of events	
Normal flow	

1. Administrator successfuly completes **UC_5**
2. Administrator sends a check status command
3. *Controller* relays the command to the specified node
4. *Controller* responds with the response received from the specified node

Alternative flow
• N/A

Exceptions	• If the node requested in step 2 is invalid, *Controller* responds with an error
Post-conditions	Administrator is presented with status of the relevant node

UC_7	
Use case	Transmit gathered metrics
Description	*Agent* attempts to transmit metric data held in its internal buffer to a connected *Controller* instance
Actors	*Agent*, *Controller*
Pre-conditions	The relevant *Agent* and *Controller* isntances have been successfuly started
Flow of events	
Normal flow	

1. *Agent* retrieves momentary metric data from its internal buffer
2. *Agent* removes the retrieved data from the internal buffer
3. *Agent* encodes the retrieved data into a format suitable for network transmission
4. *Agent* sends the encoded data to the connected *Controller* instance
5. *Agent* starts again from step 1 if the internal buffer is not empty

Alternative flow
• N/A

Exceptions	• If there is no data in the internat buffer in step 1, *Agent* halts this task until the internal buffer is supplied with data • If the transmission in step 3 fails, *Agent* stores the retrieved data back into the internal buffer and emits a warning
Post-conditions	X

UC_8	
Use case	Gather metrics
Description	The *Agent* application attempts to gather metric data using one of its gathering subsystems
Actors	*Agent*, Operating System, *Controller*

Pre-conditions	The *Agent* application has been successfuly started; the *Controller* application has been successfuly started and link with *Agent* established (for alternative flow 2)
Flow of events	
Normal flow	
1. The application invokes one of its core gathering subsystems 2. Gathering subsystems requests metric data from the underlying operating system 3. Gathering subsystem transforms data into format understood by the *Agent* 4. *Agent* stores the data into its internal buffer	
Alternative flow	
1. The application invokes a plugin gathering subsystem 2. Subsystem calls a user supplied executable and parses the output 3. Subsystem transforms the data and hands it to the *Agent* 4. *Agent* stores the data in its internal buffer	
Alternative flow 2	

1. A user has requested a run of a specific gathering subsystem using a remote command
2. *Agent* invokes the requested gathering subsystem
3. Gathering subsystem retrieves, transforms and hands data back to the *Agent*
4. *Agent* sends retrieved data back to the requesting administrator

Exceptions	• If a gathering subsystem fails to transform the retrieved data (*Normal flow step 3*, *Alternative flow step 3*, *Alternative flow 2 step 3*), it hands the *Agent* an error message; *Agent* emits a warning and stores a value indicating an error • If the executable in *Alternative flow step 2* cannot be executed, the specific plugin subsystem is moved to disabled state and a warning is emitted • If *Agent* cannot hold any more data in its internal buffer due to memory limits, the *Agent* application exits with an error
Post-conditions	The *Agent* application expanded its internal buffer with newly collected metric data; the administrator is presented with the requested metric data (alternative flow 2)

UC_9	
Use case	Save data
Description	The *Controller* attempts to save data into persistent storage
Actors	*Controller*, Filesystem

Pre-conditions	The *Controller* application and at least a single connected *Agent* application have been successfuly started
Flow of events	
Normal flow	
1. *Controller* receives metric data from a connected *Agent* instance 2. *Controller* invokes a configured storage subsystem to store the new data 3. Storage subsystem stores the data into persistent storage 4. Repeat from step 2 for each configured storage subsystem	
Alternative flow	
• N/A	
Exceptions	• If the metric data received in step 1 is in invalid format it is discarded and a warning is emitted • If an invoked storage subsystem experiences a failure in step 3 it is moved to disabled state and a warning is emitted • If there are no enabled storage subsystems the application exits with an error
Post-conditions	Metric data retrieved from a connected *Agent* instance is saved to persistent storage

5 Domain model

Rust, the chosen programming language (read more in subsection 3.2), is not object-oriented. Therefore, a standard class diagram-based domain model cannot be created. *Rust* still supports the notion of variable and function protection, however it implements these features with a file-scope, instead of a class-scope, approach. Due to this the domain model found in Figure 3 will mostly represent entities grouped together in files instead of classes.

5.1 Diagram

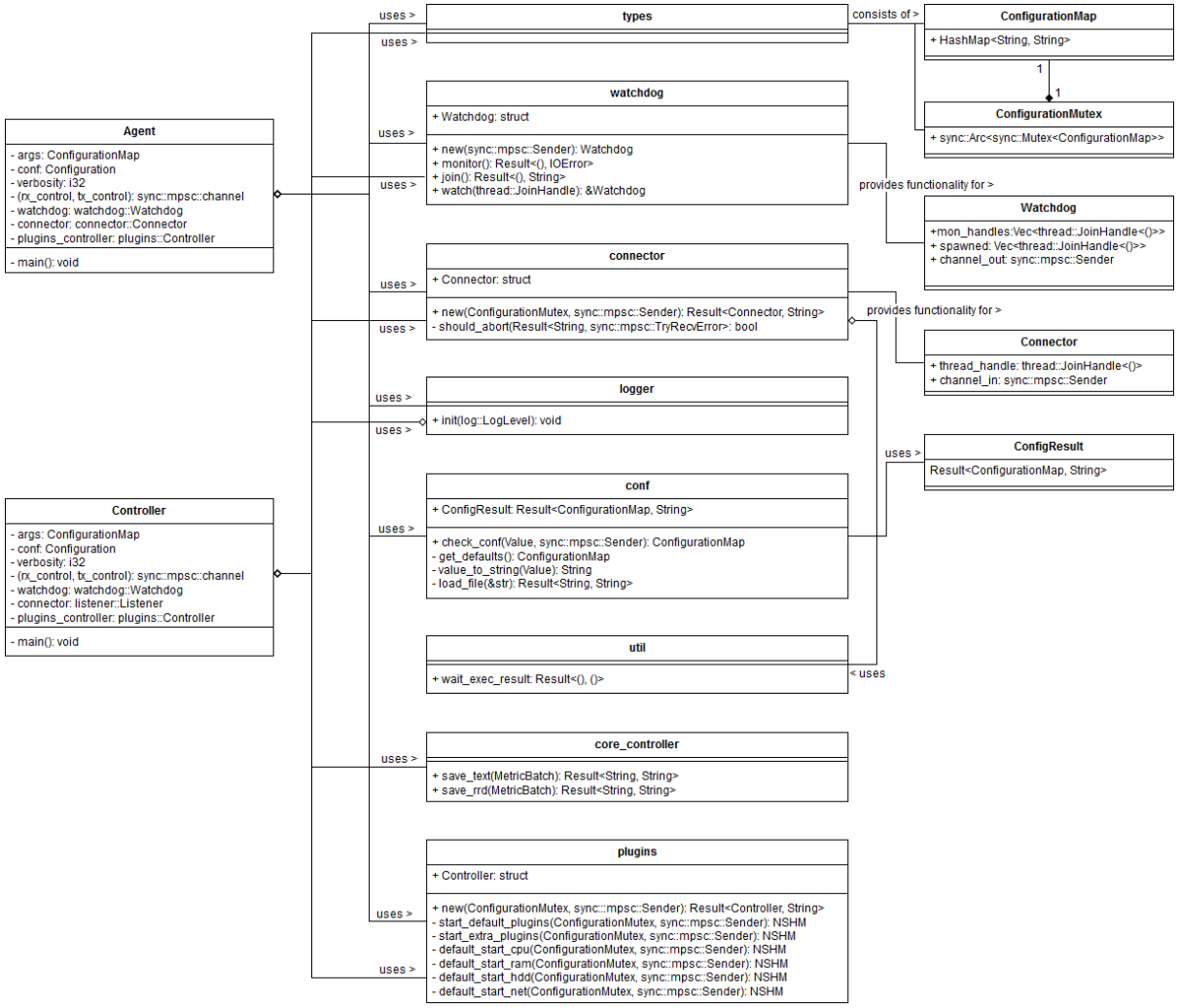

Figure 3: The domain model

6 Appendix A - Gantt chart

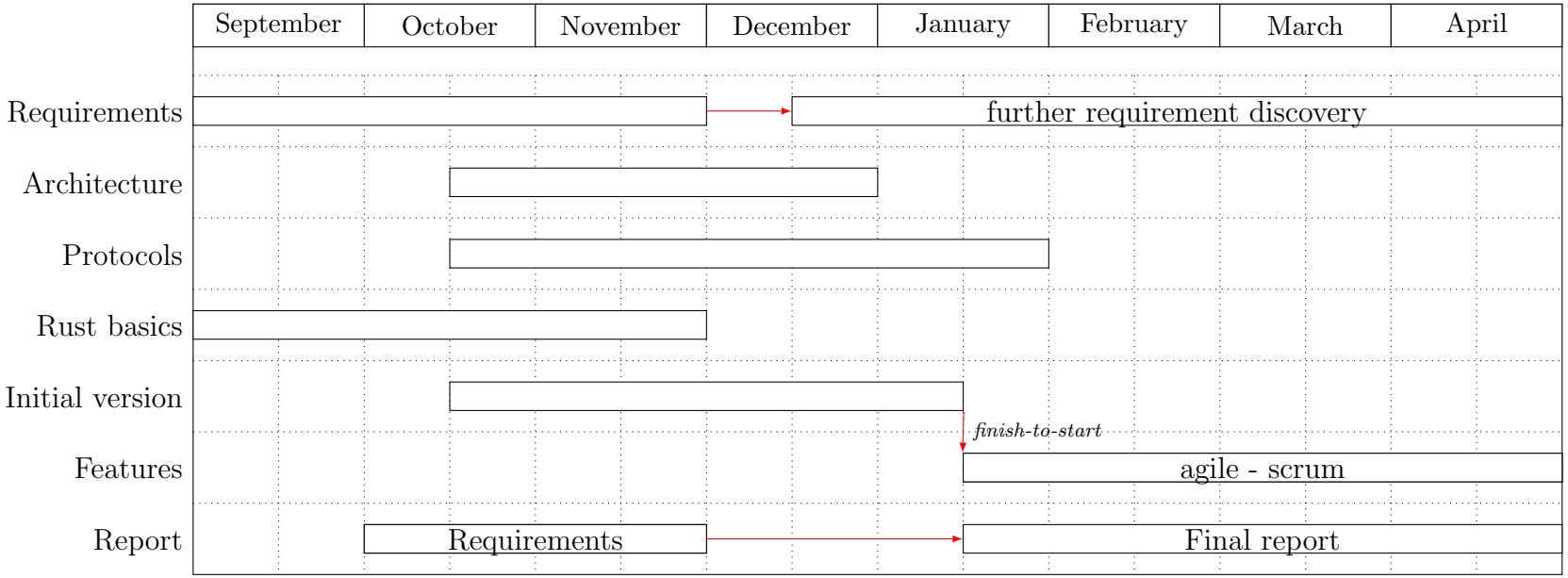

Figure 4: Gantt chart from the initial *project proposal*

References

Oetiker, T. (2014), 'About rrdtool', http://oss.oetiker.ch/rrdtool/. Accessed: 2016-12-02.

TIOBE (2016), 'Tiobe index for december 2016', http://www.tiobe.com/tiobe-index/. Accessed: 2016-12-05.

van Asseldonk, R. (2016), 'Zero-cost abstractions', https://ruudvanasseldonk.com/2016/11/30/zero-cost-abstractions. Accessed: 2016-12-02.

W3Techs (2016), 'Usage of operating systems for websites', https://w3techs.com/technologies/overview/operating_system/all. Accessed: 2016-12-02.