UNIVERSITY OF OSLO
Department of Informatics

SQSI - Search Queries for Sysadmin Information

Eirik Caspari

Network and System Administration

Oslo University College

May 21, 2012

SQSI - Search Queries for Sysadmin Information

Eirik Caspari

Network and System Administration
Oslo University College

May 21, 2012

Abstract

In this master thesis the modular architecture SQSI (Search Queries for Sysadmin Information) aimed at facilitating efficient information retrieval, unification and personalization, for system administrators is designed. The architecture allows users to search for information related to servers and services, and receive search results ranked by personal preferences, from diverse and distributed information sources at a single point. Based on the architecture, a prototype is also developed as a proof of concept.

Acknowledgements

First and foremost I would like to thank Kyrre Begnum who suggested the topic for this master thesis. His support, enthusiasm and contributions through several discussions has helped shape this project.

Next I would like to thank Tore M. Jonassen for helping with practical matters related to my thesis, and for being so quick to respond whenever something went wrong.

I would also like to thank the people at Norske Systemarkitekter AS for giving me access to their facilities, providing me with a place to work during my thesis, and giving me helpful insight into how their information sources are organized, which in turn helped me in the design of SQSI.

Lastly, I would like to thank my parents for their continued support, and taking interest in my work.

Contents

1 Introduction **6**
1.1 Problem statement 8

2 Background **10**
2.1 System Administration and Automation 10
2.2 Information Classification 12
2.3 Information Retrieval 12
2.4 Information Filtering 15
2.5 Systems Monitoring 15
2.6 Presenting Information 16
2.7 Message-oriented Middleware 17

3 Approach **19**
3.1 Designing the Architecture 19
3.1.1 Classification 20
3.1.2 Developing a Prioritization Scheme 21
3.1.3 Data Aggregation and Module Design 21
3.1.4 The Presentation Layer 21
3.2 The Prototype 22
3.2.1 The Core System 22
3.2.2 Modules 22
3.2.3 Presentation Layer Implementation 23
3.2.4 Verification 23
3.3 Task Summary 23

4 Results: Architecture **25**
4.1 Domain Classification 28
4.2 Prioritization Scheme and Preference Profiles 28
4.3 Module Design, The Query Broker and Data Aggregation 32
4.3.1 Communication and Data Transportation 33

4.4 Presentation Layer Design 37
4.5 The Meta Model 38
4.6 Extensibility 39

5 Results: Prototype **43**
5.1 Overview 43
5.2 Development Environment and Technologies 44
5.3 Data Transfer Formats 47
5.4 Queue Naming Scheme 50
5.5 Query Broker 50
5.5.1 Query Distribution 51
5.5.2 Handling Incoming Results 52
5.6 SQSI-CLI: The Client 54
5.7 The Modules 55
5.7.1 Munin Module 55
5.7.2 Request Tracker Module 58
5.7.3 DokuWiki Module 60
5.8 Functionality Testing 62

6 Discussion **68**
6.1 The Prototype 68
6.1.1 Changes to the Data Transfer Formats 69
6.1.2 Functionality Testing 69
6.2 Architecture 70
6.2.1 Caching 71
6.2.2 Prioritization 71
6.2.3 Generating Preference Profiles 75
6.2.4 Presentation Layer 76
6.2.5 Error Handling 77
6.2.6 Expanding Search Possibilities 78
6.2.7 Scalability 78
6.3 In The Server Room 79
6.4 Problem Statement Revisited 79
6.5 Source Code 80

7 Conclusion **81**
7.1 Future Work 82

8 Appendix **83**
8.1 Preference Profiles 83
8.1.1 Alice 83

8.1.2 Bob 84
8.2 Query Broker 85
8.2.1 querybroker.pl 85
8.2.2 QueryResult.pm 95
8.2.3 Result.pm 97
8.2.4 modules.dat 99
8.3 SQSI-CLI 99
8.4 Munin Module 103
8.4.1 munin-module.pl 103
8.4.2 classifications.dat 112
8.4.3 services.dat 113
8.5 Request Tracker Module 114
8.5.1 rt-module.pl 114
8.5.2 queue_classifications.dat 120
8.6 DokuWiki Module 120
8.6.1 dokuwiki-module.pl 120
8.6.2 classifications.dat 127

Chapter 1

Introduction

One of the key parts of system administration is to make sure systems and services perform according to their necessary service levels and specifications. The enterprises that utilize these systems, as well as their employees and customers depend on the systems in their daily work. It is therefore essential that these systems are reliable. In addition to that, any errors occurring needs be dealt with in an efficient manner. Furthermore, there may also be formal requirements specified by customers in service level agreements that impose even stricter demands on the systems and their administrators. This thesis concerns itself with helping system administrators reach the goal of keeping their systems and services running according to plan.

The complex and dynamic environment of systems administration:
Upholding the requirements of reliability, quick error resolution and other additional specifications is a challenge due to the great number of complex systems that can be part of the responsibilities of the system administrator, e.g. ticketing systems, monitoring, backup systems, managing web-servers, e-mail and so on. For example, in the case where a service is not behaving as it should, the administrator will often need different types of information from multiple systems like, a monitoring system that may have information about the state of the service, a configuration management system to look at the configuration of the service, a documentation system, a ticketing system to see any relevant support cases, an inventory management system to get information about the hardware and other services running on the same server.

Some of the systems that system administrators are responsible for display dynamic behaviour. For instance, when utilizing infrastructure in the cloud, where the number of servers you have at any one time may differ. Monitoring information and alerts therefore needs to be carefully correlated with the current production state in order to avoid misunderstandings. The same issue also arises when virtualization technology is used. In this case multiple virtual machines may be run on the same server. This type of dynamic behaviour induces further complexity when it comes to keeping track of the systems and their behaviour.

Information diversity and stakeholders:
An important aspect is the fact that there are different stakeholders involved with the systems that the system administrators maintain, that have different needs and demands. The fact that the structure of the information that these systems and services produce is not sensitive to the different needs of the stakeholders, can potentially introduce disorganization and inefficiency. Examples of relevant stakeholders are system administrators, service level managers, customers and the users of the systems. Being that system administrators have different areas of responsibility, the feedback they need from the systems will differ. Again, the customers have needs different from that of the system administrators, they may for instance be more interested in cost and the amount of resources they have used.

In search of control:
Observing the complex environment in which the system administrator works, with the large number of different systems and services and the wealth of information they produce, the need for control can easily be discerned.

The first step in regaining control, is to identify the reasons that may lead to a lack of control. First of all, the sheer number of systems and the amount of information generated by the systems that an enterprise is responsible for, can be overwhelming. Also, since system administrators deal with different systems, the information they generate is distributed, meaning that the output from these systems is not tied together in any way. In larger organisations and enterprises the duty of maintaining the systems and the information they produce is performed by teams of system administrators. Each of the teams will have their specific area of responsibility e.g. networking, storage, security and so on. Generally, the information generated by the systems is not sorted or adapted to individual roles, which means it can be difficult to get the complete picture of a situation within a sensible amount of time. Another issue arise in large datacenters with large numbers of servers. How does the system administrator know which services run on a particular server, and more importantly, how does the system administrator know which services are running on a particular server, that are important him or her.

Control cannot be achieved unless the wealth of information is sorted, correlated and organized, as well as presented to the system administrator in a manner which is not only sensitive to the context, but also to the individual system administrator. Also, aggregation alone will only lead to an overflow of information with a possibly high level of redundant or irrelevant information. The information will need to be classified, sorted and prioritized in order to eliminate noise and information overload.

1.1 Problem statement

The fact that system administrators already have systems in place that both they and their enterprises depend on, can not be ignored. Therefore, a sensible approach is to utilize a modular architecture, and to employ a strategy in which the already existing systems can contribute with their information. A modular architecture in this sense, is an architecture where separate modules are responsible for each subsystem. Such an approach should ensure minimal intrusion in the already existing systems, as well as minimize overhead.

The goal of this thesis is to facilitate efficient information retrieval for system administrators by designing a modular architecture which

1. *aggregates and correlates information from a variety of sources*
2. *can classify, sort and prioritize information according to the user and systems*
3. *is capable of presenting information in an efficient manner*

An architecture is a blueprint which encompasses the design and specifications of an implementation, the collaboration and interconnection between system parts, as well as the extensibility of the system.

Aggregation and correlation of information refers to the act of gathering data and linking related data together. Establishing relations between data from different sources will allow a unified view of information relevant to a particular area of responsibility or situation. These data could be collected from different sources such as a monitoring system, a ticketing system or a documentation management system.

To be able to correlate information a classification system is needed; a meta description, a categorization and classification of the information. The development of a classification enables not only correlation of information, but also gives the ability to prioritize information. Prioritization can be seen as the utilization of rules on a classified set of data. Prioritization of information is a crucial tool when dealing with large quantities of data. When prioritization is done based on user preferences or job responsibilities, it ensures that the user of the system is presented with information that is considered important to the user. This keeps the user from being overloaded with noise.

Presenting information in an efficient manner naturally entails presenting only the information that is useful to the user, but also *how* the information is presented. Using HTML (Hypertext Markup Language) to present information will ensure that the information can be viewed on any device as long as it has a web browser. This enables viewing the information on devices like tablets and smartphones, as well as computers without the need for developing specialized platform dependent applications. Also, having a system that supports many different types of clients to connect to it, receive information from it, and display that information in various ways is a great advantage. A system possessing such flexibility allows for great diversity and customizability.

Chapter 2

Background

This thesis encompasses multiple topics; how one deals with complex systems and large amounts of data, how data can be classified, information retrieval, information filtering, systems monitoring as well as presentation of information. In this chapter these topics will be explained and some of the relevant research in these fields will be presented.

2.1 System Administration and Automation

As mentioned in the introduction chapter the task of system administration involves dealing with complex systems and services, and making sure they are reliable and perform well. An important factor in the context of complex systems is how we, as humans, interact with such systems.

Mica R. Endsly [7] talks in her article "Automation and Situation Awareness" about the changing role of the human operator from direct system control towards the role of a monitor of automated systems. She points to the fact that humans play a critical part of automated systems, specifically to monitor for failures of the systems and conditions that the systems are not designed to handle. She stresses the importance of having an accurate mental model of a system in order to perform well. She specifically emphasizes the importance of situation awareness when monitoring complex systems. Situation awareness was defined by Endsley as *"the perception of the elements in the environment within a volume of time and space, the comprehension of their meaning and the projection of their status in the near future"*[8].

When designing systems for any profession it is naturally essential to have a good understanding of how the users of the system work, and how the system will fit in with the workflow of those users. Haber and Bailey [12] investigated how system administrators work by conducting a series of ethnographic field studies. The goal for these studies was to better understand the reasons behind management dominating the cost of IT systems [14]. Haber and Bailey found in their observations that the tools used by system administrators were often not well aligned with their work practices. They emphasize that system administration tools may often be developed without a sufficient understanding of the full context of system administration work. Haber and Bailey stress that sysadmins are different from other computer users, specifically with respect to collaboration between each other and the size and complexities of the systems they deal with. Furthermore, they maintain that given the serious consequences related to failure in the system administration profession, it is essential for system administrators to maintain situational awareness. Also, given the heterogeneous nature of many systems, their exist no single tool to monitor everything. They observed many instances where sysadmins created their own tools in order to enhance situational awareness. Haber and Bailey designed a set of design guidelines for system administration tools, in order to better support how system administrators work. However, they emphasize that not all of the guidelines are relevant to every situation or tool. One of the guidelines that is applicable in the context of an information gathering system, is to enable integration of the system into system-wide monitoring and management tools. Another applicable guideline is to allow sharing of system views or system state.

Woods [32] explains in his article "Decomposing automation: Apparent simplicity, real complexity" about how changes in level of automation transform systems, and that understanding these transformations *"allows one to identify and treat the many post-conditions necessary for skillful use of technological possibilities"* [32]. He emphasizes that when designing new automated systems we are not just concerned with a software or hardware object, but we are also designing a dynamic visualization that shows the current state of activities and what may happen in the system, a tool that helps the user adaptively respond to changes, and a team of people and machine agents that coordinate their activities according a situation.

This illustrates how automation of tasks introduce new complexities, and as more and more systems get automated, there is a need for systems that can help in the task of keeping the automated systems operating as they should.

2.2 Information Classification

When faced with large amounts of information of various types, the need to sort this information quickly arise. In order to sort information it is necessary to have some basic criteria for how the information is going to be sorted. So, to be able to do this, having a way of classifying the data is crucial. Furthermore, having a unified way of classifying data will allow for a uniform way of interpreting data from different sources. An ontology is such a classification.

A key feature of an ontology is the concept of shared understanding. Having a shared understanding of a domain enables inter-operability, and a potential for re-use and sharing. It also reduces conceptual and terminological confusion by providing a unified way of interpreting and describing information. An ontology entails a world view with respect to a particular domain. This world view consists of a set of concepts, their definitions and the relationships between these concepts. [29] According to S. Staab and R. Studer [26] the nowadays most frequently seen definition of an ontology is that *"An ontology is a formal, explicit specification of a shared conceptualization"* [28].

Ontologies can be used to solve the *interoperability problem*, which is the problem of bringing together heterogeneous and distributed computer systems. This is accomplished through their ability to describe semantics of information sources and to make content explicit, which is turn can be used in the context of integrating systems. [30]

2.3 Information Retrieval

Information Retrieval is an area of study concerned with retrieval of unstructured data, often in the form of textual documents. The retrieval is based on a query or topic, which may be either structured or unstructured. [11]

In the article *"A vector space model for automatic indexing"*, Salton et. al [23] describes how vector space models can be used in the context of document retrieval, where documents are compared with incoming patterns (queries). We define a document space consisting of documents D_i, with each of them consisting of one or more index terms T_j. The terms may be weighted according to importance, or they may be restricted to 0 and 1. Now each document can be represented by a t-dimensional vector:

$$D_i = d_{i1}, d_{i2}, ..., d_{it}$$

d_{ij} represents the weight of the *j*th term.

Salton et. al then explains how it is now possible to compute a similarity coefficient between two documents D_i, D_j. That is, $s(D_i, D_J)$ shows the degree of similarity in the corresponding terms and term weights. A similarity measure that can be used is the inner product of the two vectors, or an inverse function of the angle between corresponding pairs.

According to Greengrass, [11] their are multiple different ways to compute the similarity between two vectors used in information retrieval, in the context of the vector space model. Some of these are the inner product, cosine similarity and a family of distance metrics. The inner product and cosine similarity are the most widely used.

The inner product of a query vector QT and a document vector DT is computed by multiplying each component (term weight) in the query vector by the corresponding component in the document vector and summing these products:

$$\Sigma_{i=1}^{N} QT_i \cdot DT_i \tag{2.1}$$

The cosine similarity represents the cosine of the angle between the two vectors. Cosine similarity is a way of normalizing document length.

$$\cos(\theta) = \frac{QT \cdot DT}{||QT||||DT||} = \frac{\Sigma_{i=1}^{N} QT_i \times DT_i}{\sqrt{\Sigma_{i=1}^{N} (QT_i)^2} \times \sqrt{\Sigma_{i=1}^{N} (DT_i)^2}} \tag{2.2}$$

The distance metrics are given by the following equation:

$$L_p(QT, DT) = [\Sigma_i |qt_i - dt_i|^p]^{1/p} \tag{2.3}$$

The distance metrics compute the distance in vector space between the vectors QT and DT. The different metrics are chosen by the parameter p. If $p = 1$, the formulae calculates the Manhattan Distance or city block distance, that is the distance in number of city blocks from one street intersection to another, if the layout of the streets is a rectangular grid. If $p = 2$, then the formulae calculate the Euclidean distance, which is the straight line distance in the vector space. If $p = \infty$, the formulae calculates the maximal direction distance.

The importance of context and personalization in information retrieval has already been established, especially in relation to web search [Lawrence [16], Finkelstein [9], Haveliwala [13]].

According to Myaeng and Korfhage [19] it is widely recognised that different users expects different sets of data from the same query, and that they make different relevance judgements based on the same retrieved items. Myaeng and Korfhage state that in order to take into account the various preferences of users in the context of an information retrieval system, it is obvious that some form of user models needs to be developed. Myaeng and Korfhage performed a study aimed at demonstrating the superiority of information retrieval systems with user profiles, to those without user profiles, and investigating what they call the query/profile "model space". This "model space" consisted of the existing query/profile models and extensions of them.

They explain that there are three distinct ways that user profiles can be used, depending on when and how the profiles are applied to the retrieval process:

1. The profile can be used to pre-process a query to produce a modified query.
2. The profile and query can be considered as one entity to direct the retrieval process.
3. The profile can be used as a filter to post-process the results of a query.

Myaeng and Korfhage focused on the first two methods. They identified 396 different models in the model space. These models where organized along three dimensions:

1. modes of query/profile interaction
2. parameters embedded in the interaction modes
3. metrics used to discriminate among documents

They chose a *vector model* as the representational scheme, were documents, queries and profiles can be regarded as points in an n-dimensional space, where n is the number of terms describing the information object.

A system called PBS (profile-based system) was developed for the purpose of the research. It could accept queries, search a database, retrieve documents, handle profiles and evaluate different models based on the query and profile. The database consisted of 3703 different documents.

They conducted a series of experiments in a laboratory setting. The following sequence was followed in the experiments. For each of the models, retrieve a document set and select the top 25. Then merge the documents into a final set. Finally randomize the order of the documents in the final set. There were a total of 11 subjects, and a total of 30 queries were made. Each of the experimental subjects had to review at least 60 documents from the final set. Each of the subjects had to construct a user profile. The subjects had to evaluate the quality of each of the documents with respect to relevance, pertinence and usefulness.

Myaeng and Korfhage found in the results of their experiments that there were always some models which outperformed the model that consisted of a query alone. This indicates the usefulness of integrating user profiles into information retrieval systems.

2.4 Information Filtering

Information filtering and information retrieval are closely related. However, in contrast to information retrieval, information filtering is often concerned with repeated uses of a system, by persons with specific goals and interests. This means that user profiles are an integral part of an information filtering system. Information filtering systems are used as mediators between the user and the information sources, and they typically deal with streams of incoming data from multiple sources. [24][3]

Albayrak et al. [1] developed a situation aware agent-based personal information system called PIA. The system is aimed at collecting and filtering information, integrating the information at a common point, as well as presenting the information to the user. PIA uses push and pull techniques in order to allow the user to both explicitly search for information, and automatically inform the user of relevant information. The PIA system incorporates multiple information-extracting agents that continuously gather data from a variety of different sources (internet, databases, web-services, files etc.). Each user of the system has a personal agent that manages information provisioning tailored to the user, by knowing the user profile, the situation and learning from feedback.

2.5 Systems Monitoring

The field of system monitoring is about presenting the state of a system in a form that is comprehensible to humans. Furthermore, it involves collecting information from diverse and possibly distributed components, and displaying this compound information in suitable manner. The data should be displayed in such a way that it occupies a suitable portion of the operator's attention based on the operator's interests and the state of the system. [2]

2.6 Presenting Information

When presenting information in an information seeking context, there are several aspects that needs to be considered. The perhaps most important question to ask, is what type of presentation will help the users achieve their goals in the information seeking process. Also, assuming some knowledge of the user, how can the way information is presented impact the usefulness of the system in a broader context.

Mica R. Endsly [7] stresses that in order to have an accurate mental model of a system or situation, it is crucial that all relevant information is at hand, and that the amount of information is not overwhelming. Therefore, regarding the design of interfaces she emphasizes that it's important that all information that is needed is always present. Also, she expresses that at the same time one needs to avoid displaying volumes of low-level data.

When presenting information to a user there is naturally a limit to how long the user is willing to wait for that information to appear. It is therefore important to take this into account in the design of information retrieval systems. Nah [20] performed a study of tolerable waiting times in relation to web users and their willingness to wait for web pages to appear. The two main questions posed by Nah was *"How long are users willing to wait for downloading a web page before abandoning it?"* and *"Does providing feedback during the wait prolong web users' tolerable waiting time?"*. They conducted an exploratory experiment in order to determine the tolerable waiting time for web users, both with and without feedback. In this study seventy undergraduate students participated as subjects. The subjects had to visit 10 specific web pages in order to answer a set of questions, where only 7 of the websites were working. The other websites would not load which would result in the subjects eventually having to click the stop button in the browser. The time from accessing a non-working page to clicking the stop button would then be measured. The subjects were divided into two groups. One that was provided with a feedback bar signifying that the system was carrying out their requests, and one that was provided no such feedback. The results of the study showed that the inclusion of a feedback bar significantly prolonged how long a user would wait before abandoning the web page. This is in line with other research on the subject [15]. Furthermore, they found that overall their results suggest that web users tolerable waiting time peaks at about 2 seconds.

2.7 Message-oriented Middleware

When designing a modular architecture where functionality is distributed to different components, perhaps written in different languages and running in separate locations, there is need for a way to communicate between the different modules. As mentioned earlier, having a shared understanding of the domain in question, facilitates communication by providing a common language. However, in addition to that, a communication channel is needed. Such a communication channel can be provided by Message-Oriented Middleware (MOM).

The job of a MOM is to provide message-based communication between distributed applications. A MOM acts as a message mediator between senders and receivers. Messages are sent and received via the MOM and what is known as *destinations*. The messages are addressed to receivers, and they are received by subscribing to destinations. [25] An illustration of the communication flow between sender, MOM and receiver can be seen in figure 2.1.

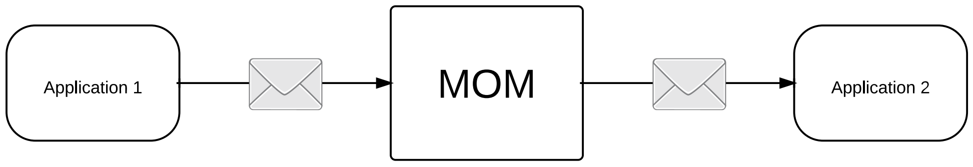

Figure 2.1: Message-oriented Middleware communication

1. *Application 1* sends a message to a destination/queue on the MOM server.
2. The message is stored on the MOM server.
3. *Application 2* which subscribes to the destination, collects the message from the queue and acknowledges it to the MOM.
4. When the MOM has received the acknowledgement, the message can be removed from the queue.

Having the MOM as a mediator means that the sender and receiver does not need to know about each other. The advantages of using a MOM is that it provides the opportunity for asynchronous communication between the different applications, and it facilitates loose coupling. *"Coupling refers to the interdependence of two or more applications or systems"* [25]. Coupling can be thought of as how changes in one application necessitates changes in other applications. If no changes are required in the other applications involved, then the applications can be said to be loosely coupled, otherwise they are tightly coupled.

An alternative form of communication in distributed systems is the use of Remote Procedure Calls (RPC). RPCs are procedure calls that operate across a network. When a remote procedure call is issued, the caller will send parameters across the network and wait. When the procedure has been run on the remote system the result is sent back to the callee and execution resumes. [4] The major disadvantage of RPC compared to MOM is that it is tightly coupled. It also does not provide asynchronous communication. [25] Asynchronous communication enables components in a distributed system to operate in an autonomous manner, in the sense that the components does not have to wait for each other to communicate. Each component can independently decide when to initiate communication. This means that each component can go about its own business, and only initiate communication when it is needed by the component itself.

Chapter 3

Approach

In order to fulfil the goals stated in the problem statement, an architecture must be designed. This architecture can be said to be satisfactory when it facilitates fulfilling the three requirements in the problem statement. Secondly, a prototype based on the architecture in question should be developed as a proof of concept. Such a prototype will show that the architecture can be applied to a real life situation, and that it can in fact facilitate efficient information retrieval for system administrators. The functionality of the prototype will need to be verified through functionality testing.

3.1 Designing the Architecture

In order for the architecture to provide a successful solution to the problem, it should include the following features:

- *A classification of the domain*
- *A prioritization scheme*
- *A data aggregation strategy*
- *An approach to the design of modules*
- *A design for the presentation layer*

Additionally, a meta model describing the entities involved, where data is processed, and how the information flows, should be designed.

3.1.1 Classification

As a basis for further data manipulation, the different types of relevant information needs to be classified. As explained in the background chapter, an ontology is such a classification. However, an ontology is something more than just a classification. Even if there are multiple definitions of an ontology [31], the common elements in the definitions is that an ontology is a formal description of a domain, intended for sharing among different applications, and expressed in a language that can be used to reason[21]. Ontologies also deals with the relations between the concepts defined in the ontology. However, if one instead limits the classification to consist of a set of well defined concepts, it allows the application of traditional Information Retrieval methods in prioritization of classified information. Therefore, an ontology will not be developed, but rather a classification that entails a categorization and their descriptions. However, the concept of shared understanding related to ontologies, as explained in the background chapter, is still very much valid in relation to this classification. This is because a shared understanding of the domain is what will allow the modules in the architecture to be decoupled from the core system.

The domain that the classification is being based, on is that of system administration. Classifying an entire domain, and especially an evolving domain, is a task that can likely never be finished. That means that the classification developed has to be extensible. This also means that the classification developed will not cover all possible terms within the domain, but it should cover the most important ones, and be considered a good starting point. In order get started on the classification, looking at what types of information that a system administrator is after can be a sensible approach. Looking at the problem from an even more general level one can ask the question of *what* it is that the system administrator wants information about. On a general level, information that helps system administrators perform their job of managing complex systems can be tied to two terms; *services* and *servers*. For example, an alert from a monitoring system can be related to a service that is not performing normally, or problems with a server, e.g. hardware related. This means that a query sent to a system designed to give relevant information could be based on the two terms service and server, and furthermore, the classification should be geared towards categories related to these two terms. That is, the classification should cover terms related to services and servers, the information they produce, and information about them.

3.1.2 Developing a Prioritization Scheme

Prioritization can, as mentioned in the introduction chapter, be seen as utilizing rules on a classified set of data. Developing a set of rules that ensures that information is sorted and filtered according to some preference is therefore essential. For the purpose of this architecture, a ranking policy that ranks the information based on a set of preferences should suffice. It is therefore necessary to find a way to calculate some form of rank, as well as a way of collecting and storing the preferences that the prioritization is based on. Furthermore, a way to generate the preference profiles needs to be designed. The ranking policy can be considered a good policy when it enables prioritizing the data.

3.1.3 Data Aggregation and Module Design

The intention of the architecture is that the information gathered will be aggregated from multiple different systems and services. In order to be able collect the data, a system based on the architecture needs a way to communicate with the different services. Being that access to the data from these services will be granted in different ways, and that communication will likely involve different languages and commands, developing individual modules is the only feasible solution. Therefore, a module design, detailing the functionality and requirements of modules will be designed. Also, a strategy for collecting data from the modules needs to be developed. Furthermore, a model of the component that will communicate with the modules and presentation layer will be designed. This component will be referred to as the *core system*. The way modules can be included into the system also needs to be considered.

3.1.4 The Presentation Layer

To ensure separation of presentation and content, a separate presentation layer will be designed. The presentation layer should be separated from the business logic in such a way that adding new modules to the system should *not* impose changes to the presentation layer.

3.2 The Prototype

Based on the architecture a prototype will be developed. This prototype should include an implementation of the core system, a different set of service modules as well as a presentation layer implementation. The prototype will be based on freely available technologies and open source software.

3.2.1 The Core System

Developing the core system involves implementing the system according to the architecture, as well as choosing technologies that allows the implementation to follow the ideas and concepts described in the architecture.

3.2.2 Modules

To demonstrate the functionality of the modular design, modules for three different services will be developed. These services should be different with respect to the type of content they provide, such that the architectures independence from content type can be demonstrated. However, due to the time frame of this thesis, these modules will not cover all functionality related to their services. They will instead cover a subset as a proof of concept. Also, it can be argued that developing full functionality for these modules will not lend any additional validity to the architecture.

Modules will be developed for the following services:

- *Munin: an open source networked resource monitoring tool*
  Munin is an open source resource monitoring tool providing trend information in the form of graphs through a web interface. Munin has a master/node architecture, and the master gathers data from all of the nodes. This means that the munin-node software needs to be installed on any computer that is to be monitored. By default Munin provides a lot of graphs about the system that is monitored. Also, it relatively easy to write plugins for Munin to generate graphs for virtually any resource. [18]

- *Request Tracker: A widely used open source issue tracking system*
  Request tracker is used for bug tracking, help desk ticketing, customer service, workflow processes, change management and more. It provides access through a web interface which allows creating tickets, managing tickets, managing users and groups, and much more. Additionally it provides

a command line client which allows other programs to communicate with RT, and provides integration with other systems. [17]

- *DokuWiki: an open source Wiki*
  DokuWiki is a wiki targeted at developer teams, work groups and small companies. It allows creation of documents using a simple syntax. DokuWiki stores all of its documents as plain text files, which means there is no need for a database. [10]

3.2.3 Presentation Layer Implementation

The presentation layer (client) will be implemented as a Command Line Interface (CLI) script. The CLI script will adhere to the requirements of the architecture, and communicate with the system in the same manner that any client would, for example an HTML/JavaScript client. This will demonstrate how clients can be developed by third parties as long as they adhere to the architecture.

3.2.4 Verification

In order to verify the correct functioning of the prototype, the functionality of the prototype developed should be tested. Performing the functionality testing will involve specifying the functionality expected of the prototype, creating test data to allow the functionality to be properly tested, executing the tests and comparing the results of the tests with the expected functionality.

3.3 Task Summary

Table 3.1 summarizes the tasks defined in the approach for the architecture, the prototype and the verification of the prototype.

Table 3.1: Task Summary

Category	Task	Comments
Architecture	**A.1:** Define a classification based on the domain of system administration, geared towards categories related to services and servers	
	A.2: Define a ranking mechanism capable of ranking information based on a set of preferences (preference profiles).	The preference profiles should be based on the classification from **A.1**
	A.3: Design a way to generate preference profiles.	The preference profiles should be based on the classification from **A.1**
	A.4: Define a design for the core system.	
	A.5: Define a module design, detailing the functionality and requirements.	
	A.6: Define a communication protocol for communication between the different components.	
	A.7: Design the presentation layer	
	A.8: Define how the architecture, and systems based on the architecture can be expanded	
Prototype	**P.1:** Develop an implementation of the core system.	Based on the requirements from **A.2**, **A.3**, **A.4** and **A.6**.
	P.2: Develop a module for Munin	Based on the requirements from **A.5** and **A.6**
	P.3: Develop a module for Request Tracker	Based on the requirements from **A.5** and **A.6**
	P.4: Develop a module for DokuWiki	Based on the requirements from **A.5** and **A.6**
	P.5: Develop CLI client implementation	Based on the requirements from **A.6** and **A.7**
Verfication	**V.1:** Specify the expected functionality	
	V.2: Create test data	
	V.3: Execute tests and compare the results with the expected functionality	

Chapter 4

Results: Architecture

The architecture presented in this chapter is aimed at providing a way for system administrators to get relevant personalized information from multiple sources, at a single entry point. Furthermore, the architecture is focused on being flexible, extensible, and allowing for easy inclusion of information sources.

A system based on this architecture will allow users to search for information related to servers and services. Searches are performed through a client which sends queries with the search information to a query broker. The job of the query broker is to distribute queries to the modules responsible for the different information sources. The query broker is also responsible for prioritizing the results to the queries coming from the modules, based on the preferences of the user that issued the query, and sending the results back to the client. The query broker was refereed to as the core system in the approach chapter. Figure 4.1 on page 26 shows the key components and concepts in the architecture, how they relate to each other, and their basic interaction pattern.

The following list gives a short explanation of the different components and their role in the architecture:

- *Classification*

  The classification is used to describe important concepts in the domain of system administration. It enables a common language which facilitates the creation of preference profiles, classifying search results, and performing prioritization of results.

- *Prioritization*

  Prioritization is done with the help of the preference profiles, and the clas-

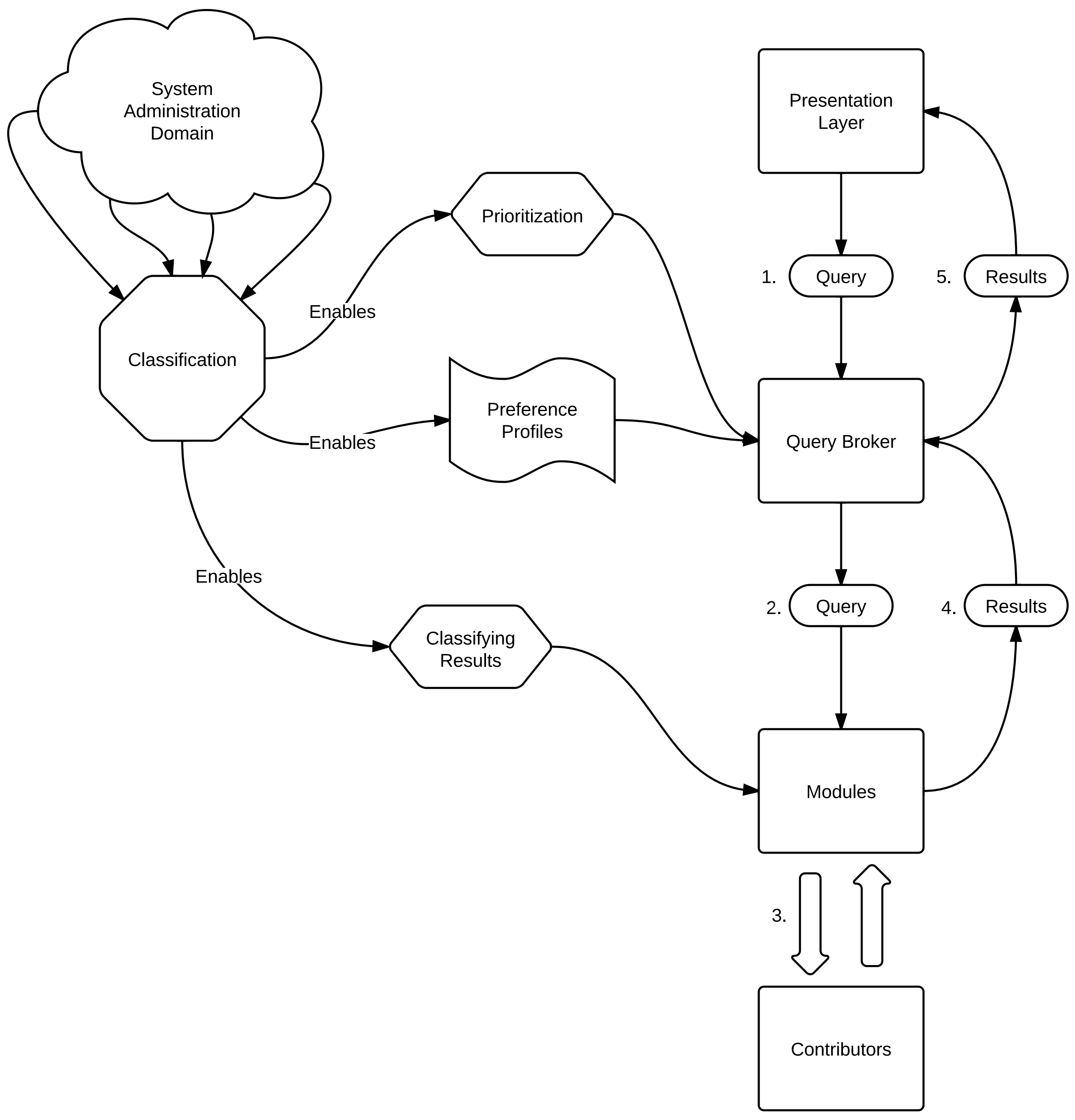

Figure 4.1: Key components and concepts in the architecture

sification applied to the search results.

- *Preference Profiles*

  The preference profiles are used to store the preferences of the users of the system, and is the basis for prioritizing search results.

- *Presentation Layer*

  The presentation layer is where searches for information are issued, and where the results from the searches are presented to the user. Searches are issued by sending queries to the query broker containing the search parameters.

- *Query Broker*

  The query broker is responsible for handling queries from the presentation layer, distributing the queries to the modules, gathering results from the modules, performing the prioritization according to the preference profiles, and sending the ranked results to the presentation layer.

- *Modules*

  Modules provide the search results, and classifies the results they provide. A module is a component responsible for a contributor. That is, a module is considered the expert in the workings and information of its contributor. Modules gather search results by communicating with their contributors.

- *Contributors*

  Contributors are services that provide data to be used in the system. Examples of possible contributors are monitoring systems, ticketing systems, documentation systems and so on.

- *Queries*

  Queries are the entities which contain all information related to a specific search.

- *Results*

  The results are entities which contain the data relevant to a specific query.

The next four sections are focused on explaining the different components in detail. Then a meta model of the architecture is presented, giving an overview of the interaction of the components involved. The last section explains how the architecture and systems based on the architecture can be expanded.

4.1 Domain Classification

The purpose of the classification is to enable a shared understanding of the problem domain, and give a foundation for prioritizing data. In order for the classification to facilitate a shared understanding it is imperative that the terms are interpreted in the same way by all users of the classification. Therefore, all terms in the classification should have a corresponding explanation, stating what is specifically meant by the term and in what context it should be used, unless it is clearly evident. This should ensure that the terms will be used in a uniform way, and the way they were intended.

The terms in the classification were determined by investigating the terms used in different types of monitoring software, ticketing systems and documentation systems, as well as through discussions with professional system administrators. The classification is divided among two tables. It can be found in table 4.1 on page 29, and continues in table 4.2 on page 30.

4.2 Prioritization Scheme and Preference Profiles

The prioritization of the results gathered from the modules, is based on the classification and preference profiles. Being that the classification consists of a set of terms, creating a preference profile amounts to applying priorities (weights) to all the different terms in the classification. In the case of weights, a higher number indicates a higher priority of the related term. The preference profiles can then be stored in a key value type format, and be identified by the name of the user.

Preference profiles can be generated by having the users apply weights to all the terms in the classification, for instance through a user interface. The specific number range allowed for the weights is not essential, but it should be large enough to allow the user to differentiate between the different terms. For instance, the weights could be in the range between 0 and 1, and allow decimal numbers. The following figure shows an example preference profile based on a small classification:

Example preference profile

```
1.0 Alert
0.3 Hardware
0.1 Network
0.6 Ticket
0.0 Trend
```

Alert	A term used to classify alerts. For example an alert coming from a monitoring system.
Backup	A term used to classify any type of backup related information.
Configuration	A term used to classify any type of configuration information.
Customer	A term used to classify any type of customer related information.
Database	A term used to classify any type of database related information.
Firewall	A term used to classify any type of firewall related information.
Hard Drive	A term used to classify any type of information related to hard drives.
Hardware	A term used to classify any type of hardware related information.
Documentation	A term used to classify any type of documentation.
Guarantee/Licence	A term used to classify any type of information related to guarantees or licences.
Hostname	A term used to classify hostname information
Installed Software	A term used to classify information describing what software is installed on a computer.
Intrusion Detection	A term used to classify any type of information related to intrusion detection.
Linux	A term that can be used to classify information related to any flavours of Linux based operating systems.
Mail	A term used to classify any type of information related to e-mail.
Mac OS X	A term that can be used to classify information related to any flavours of the Mac OS X operating systems.
Middleware	A term used to classify any type of information related to software that provides services to software applications.
Monitoring	A term used to classify monitoring information that is *not* trend information. See term Trend.
Network	A term that can be used to classify any network related information.
Network Interface	A term that can be used to classify any information regarding network interfaces.

Table 4.1: Classification part 1

Order	A term used to classify orders. An order is a commission or instruction to produce or supply something in return for payment.
OS	A term used to classify operating system related information.
Process	A term that can be used to classify any information related to processes. A process is a running instance of a computer program.
SLA	A term used to classify service level agreement information.
Support	A term that can be used to classify any information related to support.
System	A general term that can be linked to any system related information.
Ticket	A term used to classify a case supplied by a ticketing system.
Trend	A term used to classify monitoring information related to tracking changes occurring over time.
Uptime	A term used to classify any kind of uptime related information.
Vendor	A term used to classify any kind of vendor related information.
Web	A term used to classify any type of web related information.
Windows	A term that can be used to classify information related to any flavours of Microsoft Windows operating systems.

Table 4.2: Classification part 2

The results arriving from the modules will include a classification vector. The classification vector consists of 0's and 1's indicating which terms in the classification the result is related to. This means that a single result can be related to multiple terms. For instance a ticket from a ticketing system about a server that has gone down, can be related to both the *"Ticket"* term and the *"Alert"* term. Allowing multiple terms to be applied a specific result allows a more fine grained and precise description of results in comparison to sticking with singular terms.

A classification vector for a result related to the terms *"Ticket"* and *"Alert"*, with the same classification as in the preference profile example, would look like this:

$$C = (1, 0, 0, 1, 0)$$

Each result will be dealt a rank based on the preference profile and the classification vector. The system reads the preference profile of a user and stores that in a vector as well; from now on referred to as a profile vector.

The idea is to apply an approach used in Information Retrieval (IR) to calculate the similarity between the two vectors. This approach is the *vector space approach* explained in the background chapter. However, what is compared is not documents and queries as in the case of the vector space approach for IR, rather the profile vector and the classification vector of a result is what is being compared.

Three types of similarity measures where explained in the background chapter in relation to the vector space approach. These where inner product (equation 2.1), cosine similarity (equation 2.2) and the distance metrics (equation 2.3). The choice of which similarity measure to use fell on *inner product*. This was based on the fact that it is one of the two most widely used according to Greengrass [11] (the other one being cosine similarity), their is no need to normalize document length (which is what cosine similarity does), and it is the least computationally demanding. The following formula shows the calculation of rank (priority) with a classification vector C, a profile vector P, and N being the number of terms in the vectors:

$$R = s(C, P) = C \cdot P = \Sigma_{i=1}^{N} C_i \times P_i \tag{4.1}$$

Suppose there exists a classification consisting of the following 5 terms:

```
Example classification
Alert
Hardware
Network
Ticket
Trend
```

Let $P1$ and $P2$ be two profile vectors:

$$P1 = (1.0, 0.3, 0.1, 0.6, 0.0) \tag{4.2}$$

$$P2 = (0.2, 0.8, 0.7, 0.0, 1.0) \tag{4.3}$$

As can be seen, $P1$ is mostly concerned with information dealing with *Alert* and *Ticket*, where as $P2$ is mostly concerned with *Hardware*, *Network* and *Trend* information. Let $C1$ and $C2$ be the classification vectors belonging to two different results:

$$C1 = (1, 0, 0, 1, 0) \tag{4.4}$$

$$C2 = (0, 0, 1, 0, 1) \tag{4.5}$$

$C1$ could be a classification vector belonging to an alert from a ticketing system, where as $C2$ could be a result containing network related trend information. Now calculating the ranking for both profile vectors and both classification vectors according to formula 4.1, yields the following results:

$$R_{1,1} = C1 \cdot P1 = 1.6 \tag{4.6}$$

$$R_{2,1} = C2 \cdot P1 = 0.1 \tag{4.7}$$

$$R_{1,2} = C1 \cdot P2 = 0.2 \tag{4.8}$$

$$R_{2,2} = C2 \cdot P2 = 1.7 \tag{4.9}$$

As can be seen for $P1$, $C1$ yields a higher rank than $C2$, which corresponds with the weights assigned in $P1$. As for $P2$, $C2$ yields a higher rank than $C1$, which again corresponds with the weights assigned in $P2$. The result of the ranking means that if performing the same search (query) with the two different profiles $P1$ and $P2$, then for $P1$ the result with classification vector $C1$ would be prioritized higher than the result with classification vector $C2$. On the other hand, for $P2$, the result with classification vector $C2$ would be prioritized higher than the result with classification vector $C1$.

4.3 Module Design, The Query Broker and Data Aggregation

A key feature of the modular architecture is that responsibility for the contributors is delegated to the different modules. The idea behind delegating responsibility in this way, is that each module should be considered the expert in its field, or more precise its service. Dividing responsibility in this way will ensure that the query broker can treat all modules in the same way. So, each module is responsible for knowing how to communicate with and acquire data from its related service. The module also has to be able to deliver that data back to the query broker.

The way data is aggregated is closely related to the module design. Since each module is responsible for knowing how to acquire data from its related service, it means that a single message asking for data can be sent to all modules. This message will be in the form of a single identical query. It is therefore necessary that all modules understand this query, and are able to translate the meaning of the query into commands that their respective services can understand. It is, however, important to mention that such a query may not be relevant for a particular module. In this case the module should simply return an empty result indicating that it has no relevant data for that particular query. Figure 4.2 on page 34 illustrates the interaction between the query broker, modules and contributors.

4.3.1 Communication and Data Transportation

As explained in the background chapter, Message-oriented Middleware or MOM enables asynchronous message based communication among distributed software components. Utilizing MOM as the communication medium allows each module to run as a separate process with its own query queue and its own result queue. This allows a complete decoupling of the modules from the query broker. The query broker will also have its own query queue to which the clients will send their queries.

Queries

In order for the query broker to be able to distinguish among different queries it needs some way of identifying them. Firstly, it is obvious that a query needs to be linked to the user issuing it. This will enable a query to be associated with the correct preference profile. Also, since dealing with queues as means of communication, there is a need to know which result queue to send the results to. Secondly, adding a device specification to the query enables a single user to perform different searches from different devices at the same time. Such devices can be different physical devices, or simply different tabs in a web browser. This means that each user and device combination will get a result queue, to which the results from a query are delivered. As mentioned earlier, the actual search is performed per service, server or both, so that has to be part of the query as well. Based on these observations a query from a client could look like this, using XML type notation to identify the different components:

Example XML-formated query sent from the client to the query brokers query queue

```
<query>
    <user>Bob</user>
    <device>Device01</device>
    <service>Apache2</service>
    <server>Webserver01</server>
</query>
```

In addition to the above mentioned requirements for a query, the query broker

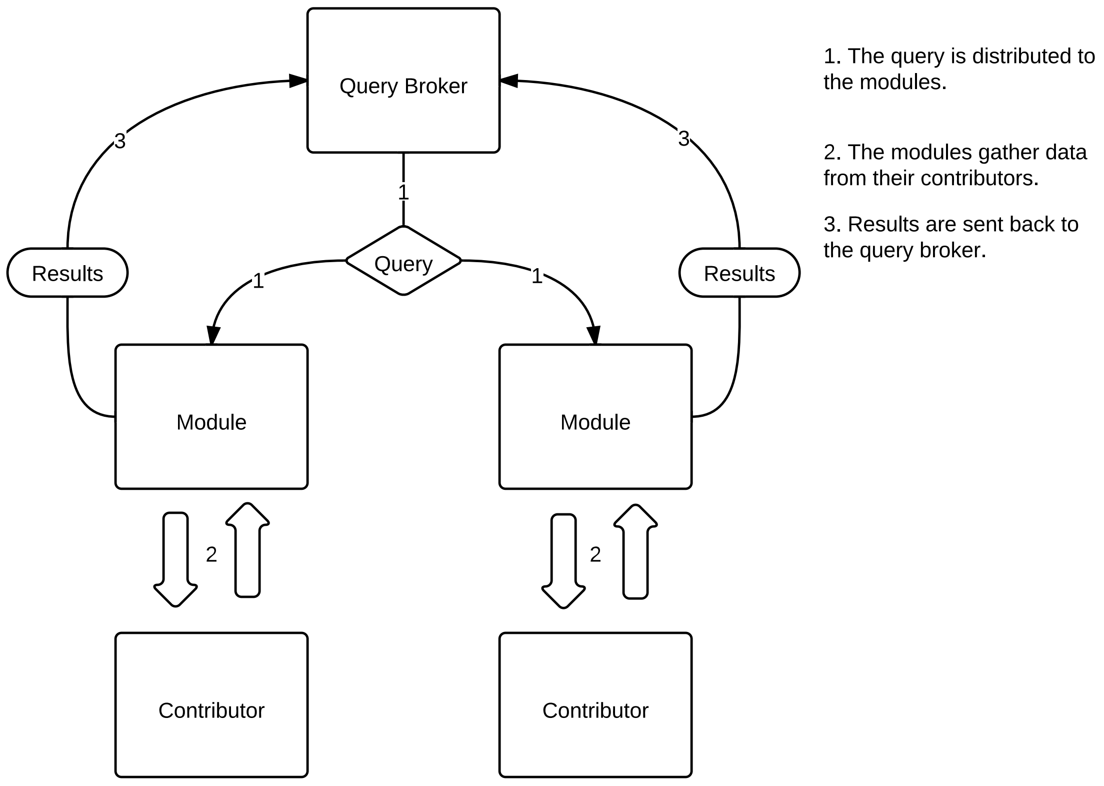
Query Broker
3
3
1
Query
1
1
Results
Results
Module
Module
2
2
Contributor
Contributor
1. The query is distributed to the modules.
2. The modules gather data from their contributors.
3. Results are sent back to the query broker.

Figure 4.2: Query Broker, Modules and Contributors interaction pattern

needs a way of differentiating between different queries from the same user on the same device. This is important in the case where a user performs multiple searches in rapid sequence. Doing this could result in the query broker having multiple queries to handle from the same user on the same device at the same time, and only the newest query will be relevant. This can be solved by adding a query number to the queries sent to the modules. In this way the query broker can keep track of query numbers, and discard any results coming from the modules related to outdated queries. The following figure shows how such a query could look, again using XML type notation:

Example XML-formated query sent from the query broker to the modules query queues

```
<query>
      <query-number>1</query-number>
      <user>Bob</user>
      <device>Device01</device>
      <service>Apache2</service>
      <server>Webserver01</server>
</query>
```

Results

Each module providing data to the query broker, is responsible for tagging the data gathered from their respective services with classification vectors before returning it to the query broker. This has to be done to enable the query broker to prioritize the data.

A sensible way to keep the amount of data being transferred low, is to not actually transfer the contents of the data, but rather information about where that data can be found. Being that many information systems provide access through web interfaces, passing urls in the results is a good way to provide access to the resource. In cases where this is not possible, actual data could be transferred instead.

A result needs to include some sort of description, enabling the user to distinguish among different search results. Therefore, each result will include a title clearly describing the contents of the result.

Summing up the requirements for a result, it is clear that each individual result needs a title, a classification vector and a url or data tag. The following figure shows how a result could be tagged using XML notation:

Example XML-formated single result

```
<result>
      <title>My Title</title>
      <classification>0000000000000000000000000000001000010000</classification>
      <url>http://someurl.html</url>
</result>
```

The modules will have to provide some meta data in the results sent back to the query broker. This is to enable the query broker to recognize which query the result belongs to. Being that a query often will result in multiple results from the same module, encapsulating the results in a meta data container is a good idea. This will ensure that the meta data is only sent once for each set of results as a response to a query, instead of introducing redundancy by tagging each individual result with the same meta data. The necessary meta data needed, includes of course the meta data sent in the query, which is the query number, user name and device name. In addition to that the source of the data will be the same for all results, and should also be included. Passing the data source, that is the name of the service providing the data, should enable the user to more easily distinguish among different types of data. Also, providing a timeout which tells how long a result should remain in the clients result queue, will ensure that in the case where data retrieval is aborted at the client side, results already delivered to the queue will not linger forever and disrupt new searches. This of course means that the underlying MOM has to support this as a feature. The following figure shows two results encapsulated in a module result using XML notation:

Example XML-formated module result as sent from the modules to the module result queue

```
<module-result user="Bob" device="Device01" query-number="1" source="Some Service" expiry="20000">
    <result>
        <title>Title 01</title>
        <classification>000000000000000000000001000010000</classification>
        <url>http://someurl.html</url>
    </result>
    <result>
        <title>Title 02</title>
        <classification>000000000000000000000000000010000</classification>
        <url>http://someotherurl.html</url>
    </result>
</module-result>
```

When a module-result has been processed by the query broker and the priorities calculated for the individual results, the results are sent to the result queue of the client identified by user and device. The result has to contain the priority of the result, so that it can be sorted, in addition to the other information contained the result delivered from the module. The following figure shows what a result delivered to the client result queue could look like, using XML notation:

Example XML-formated individual result sent to the client

```
<result>
    <priority>1.0</priority>
    <title>Webserver01 is down</title>
    <url>http://some.url.org</url>
    <source>Request Tracker</source>
</result>
```

Finishing a search
Letting the user issuing a query know when the search is finished, and all results are received, prevents the user from waiting in vain thinking more results may arrive. For instance, if a query produces no results, then the user should be made aware of this as soon as possible. Therefore, the query broker needs to keep track of which modules that have delivered their results for a particular query. When all modules have delivered their results, the query broker should send a message to the result queue of the client, indicating that no more results will arrive. The following figure shows an example of an XML tagged message letting the client know that the search is finished:

Example XML-formated search finished message

```
<search-finished></search-finished>
```

Module Integration
Making the query broker communicate with a new module will be a simple matter of providing the name of the module in question. Due to the decoupled nature of the architecture via MOM, the query broker simply needs to know the names of the query and result queue for the module. By following a naming scheme for the queues based on module names, the name of the module will be all that is needed for the query broker to communicate with the module.

4.4 Presentation Layer Design

A key aspect of the presentation layer is to make sure that the information is displayed in a user friendly manner. This also includes how long it takes for the system to display that information. Being that in this architecture data is gathered from multiple sources, it is only natural that the data will arrive at different times. The presentation layer should therefore be able to display partial results. That is, the results from the different sources should be displayed as they arrive, and reordered according to the prioritization. This will ensure that the user will not have to wait until all the information is gathered before seeing results, which should help alleviate user impatience. This is in line with the research by Nah [20] explained in the background chapter, where it was found that feedback prolonged the time a user would wait before abandoning a webpage.

The next issue to address is how information should be displayed. As mentioned in the approach, using web technology will allow for platform independence. The use of a web interface means that queries can be stored as urls and integrated into other systems. For instance several monitoring systems send out emails notifying system administrators of alerts. Being that queries can be stored as urls, these urls can be included into the alert emails providing a fast way of getting information relevant to the alert. The decoupled nature of the architecture enables different types of clients to be developed, so clients can be developed to cater to different needs.

How the information should be displayed is of course dependent of what kind of information is available, and the amount of information that is available. As described earlier, the following information is available to the presentation layer.

Type of information:

- Priority
- Title
- Url
- Source

The priority ranking number is likely of little value to the user, and should therefore not be displayed, but rather used to sort the results. The url, also does not necessarily need to be displayed to the user. The title could be made into a link, linking to the url, depending on the client. The title and source on the other hand should be shown. As the results are to be sorted, the results should be displayed as a list.

Being that the query broker will send all results relevant to the specified query, the amount of results received by the presentation layer can be large. Therefore, a mechanism limiting the amount of results shown to the user could be implemented. For example, in the case of a web client, one could limit the number of results to 20 per page.

Finally, the user interface should give the user some form of feedback when all the results from the search has arrived, meaning that the search is finished. This can be accomplished with the help of the special *search-finished* message explained earlier.

4.5 The Meta Model

In this section the meta model is presented. The model shows the components involved in the architecture, and the flow of data between them.

Figure 4.3 on page 40 shows the different components in the architecture and how they are related to each other. Also, the basic flow of information between the different components is depicted. The information flow follows the following pattern:

1. A user performs a search for information. User, device, service and/or server information is sent to the Query Broker in the form of a query.
2. In the Query Broker a query is generated based on the information from the Presentation Layer, given a query-number, and sent to all Modules.
3. The Modules translates the query into something the Contributors understand, and requests data from them.
4. Data is returned from the Contributors to the Modules.
5. The Modules tags the data according to the Classification and sends the results to the Query Broker.
6. The Query Broker utilizes the Preference Profiles to prioritize the results from the modules, and then sends the prioritized results to the Presentation Layer.

What figure 4.3 does not show is how information is transferred between the different components. This is as explained earlier solved with the introduction of MOM as the data transportation method. The result of introducing MOM into the architecture can be seen in figure 4.4 on page 41. This involves:

- Having a query queue for the Query Broker, to which all queries from the clients are sent.
- Having a query queue for each Module, to which the Query Broker distributes queries.
- Having a result queue for each Module, to which each Module sends their results, and the Query Broker collects results from.
- Having separate result queues for each user and device combination, to which the Query Broker sends the ranked results for the queries.

4.6 Extensibility

The extensibility of a system based on the architecture is handled through the development of modules. The fact that modules can be developed for virtually

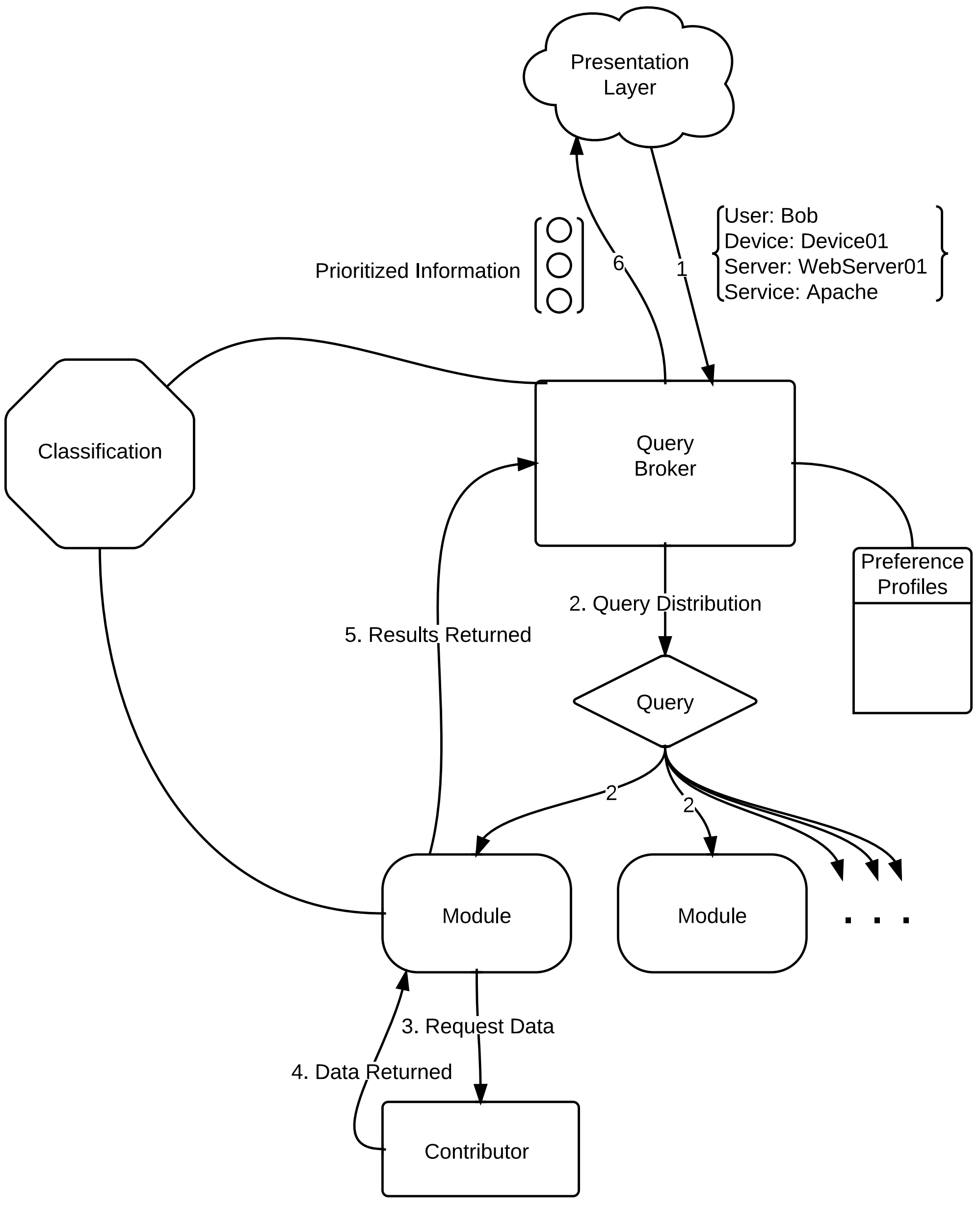

Figure 4.3: Meta Model

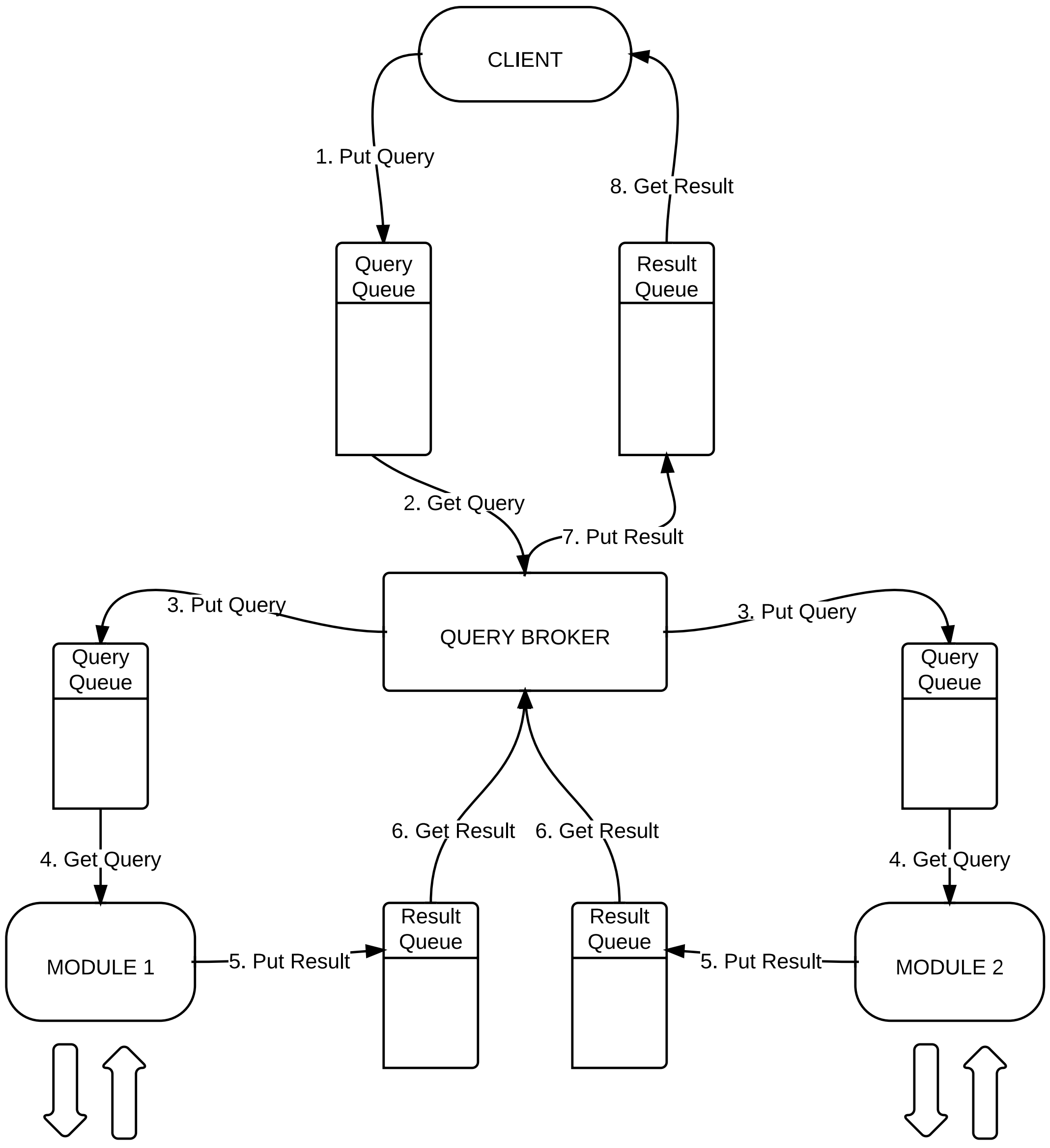

Figure 4.4: Meta Model including MOM

any service providing data, allows the system to be extended and adapted to the needs of the organization using it. However, with new modules, it is likely that new terms may need to be included in the classification. Hence, the extensibility of the classification also impacts extensibility through new modules.

One key issue to look at, is how extending the system impacts the already existing components in the system. Adding new modules will not impact the system in any other way than that there will be more data to handle by the Query Broker. However, changes made to the classification, if not done correctly can cause the other modules classification schemes to fail. Therefore, adding to the classification has to be done in the following way:

Any new term must be added to the end of the classification. This will ensure that there will be no shifting of indexes in the classification vectors causing existing modules classification schemes to fail. Furthermore, the Query Broker needs to check whether the length of the classification vectors received in the results from the modules, matches the length of the classification rules. If not, trailing 0's should be added to the classification vectors of the results. This is to ensure that the prioritization rank can be calculated, and it will also remove the need to update all other modules to adhere to the updated classification rules.

As with the classification vectors, when the classification has been extended, the query broker needs to take into account the fact that previously created preference profiles will not contain the new terms. In this case trailing 0's can be added to the profile vectors by the query broker as a temporary solution until the preference profiles are updated.

Chapter 5

Results: Prototype

This chapter explains how the prototype was developed, the technologies used to build it, and how it relates to the architecture. The last section of the chapter details the functionality testing of the prototype.

It is worth noting that this being a prototype, robustness and security has not been a priority. Therefore, the provided code is not production ready. However, all provided scripts are fully functional as long as they are used correctly. Also, direct data transfer from the modules is not supported by the prototype. That is, the data tag in the results and related functionality has not been implemented. This choice was made due to time constraints, and the fact that it is not essential in demonstrating the validity of the architecture.

5.1 Overview

The prototype addresses the five tasks specified in the prototype section of the task summary table (table 3.1 on page 24). The prototype includes implementations of:

- The Query Broker (referred to as the core system in the task summary table).
- A CLI client based on the presentation layer design
- A module for Munin
- A module for Request Tracker
- A module for DokuWiki

Figure 5.1 on page 45 illustrates how all the components in the prototype interact in order to get from a query being sent from a client, to relevant information being returned.

1. In the first step *Client 1* sends a query containing the search parameters and meta information to the query queue of the Query Broker.
2. Next *Client 2* also sends a query to the Query Broker.
3. When the Query Broker is ready it fetches a query from its query queue; in this case the query from *Client 1*.
4. The query broker adds a query number to the query, and sends this query to the Munin Module, Request Tracker Module and DokuWiki Module. The query broker is then free to fetch another query from its query queue.
5. The modules will when they are ready, each fetch their copy of the query from *Client 1* from their respective query queues.
6. When the query has been handled by a module, and data extracted from their contributors and classified, the results are sent as a *module result* to the module's result queue.
7. When the Query Broker is ready, it will fetch a *module result* from one of the module's result queues. The individual results are then extracted from *module result* and given a rank according to the preference profile of the user and classification of the results.
8. Then the results are sent to the result queue matching the user and device that issued the query.
9. The client will continuously check for results arriving in its result queue, and fetch them as they arrive. When results arrive the will be sorted according to their rank, and presented to the user.

5.2 Development Environment and Technologies

This section explains the various technologies involved in the prototype, as well as the infrastructure used in the development, and how the different components in the system (modules and query broker) where distributed.

Message-oriented Middleware (MOM)
The choice of Message-oriented Middleware for the prototype fell on Apache ActiveMQ. ActiveMQ is an open source MOM from the Apache Software Founda-

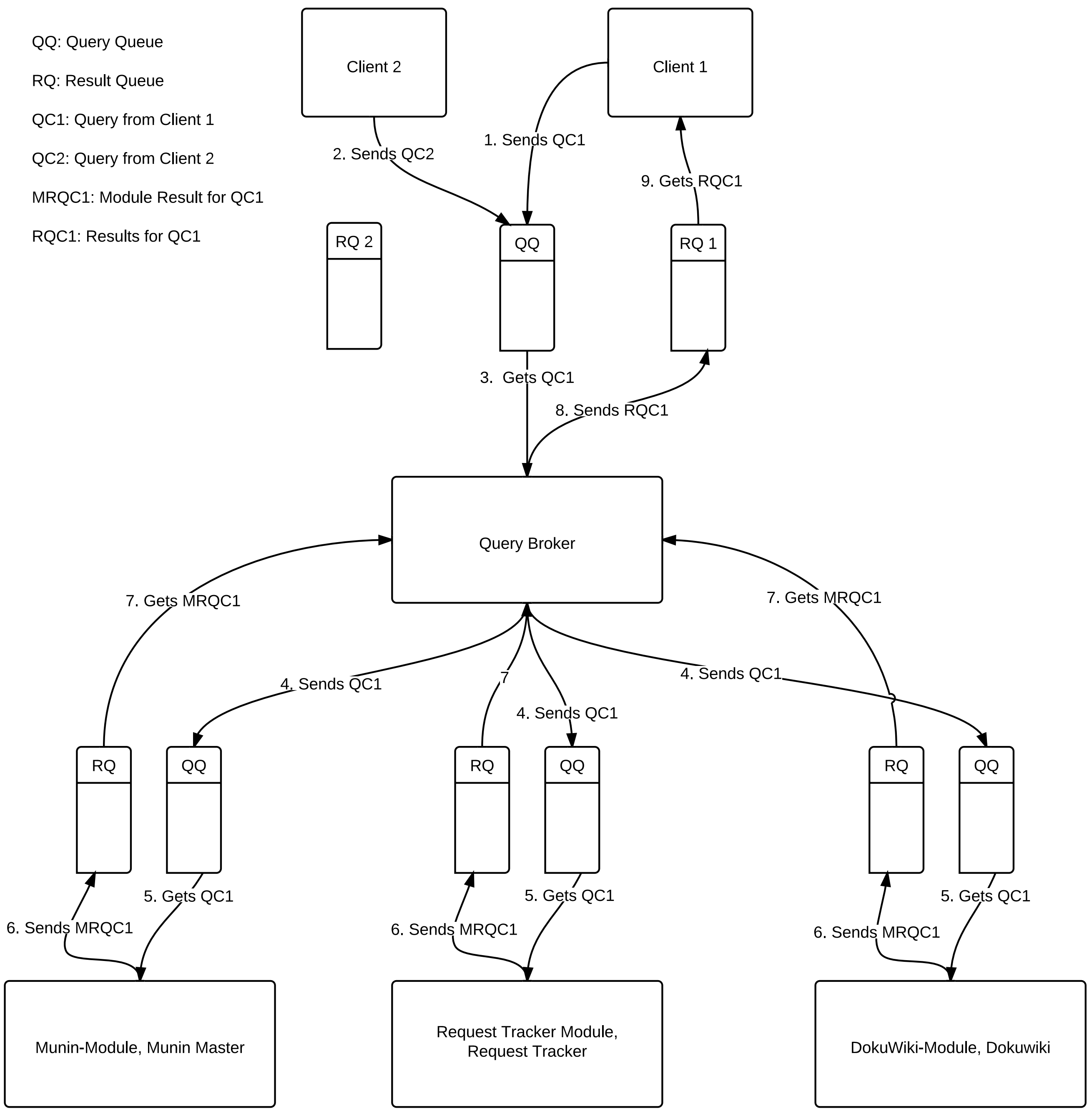

Figure 5.1: Prototype Information Flow

tion. It provides client APIs for multiple programming languages including Java, C/C++, .NET, Perl, PHP, Python, Ruby and more. This allows for applications/-modules that are to communicate via the MOM to be written in many different languages. [25]

Simple Text Oriented Messaging Protocol (STOMP)
The STOMP protocol is a simple text oriented protocol that is designed for asynchronous messaging between clients via a mediating server. [27] Using STOMP in conjunction with ActiveMQ simplifies writing clients for ActiveMQ in multiple different languages. ActiveMQ can easily be configured to use the STOMP protocol.

Programming Language
The query broker, the client (sqsi-cli) and all modules were written in the Perl programming language. Perl provides easy integration with ActiveMQ and use of the STOMP protocol through the Net::Stomp library.

Non-standard Perl libraries needed to run the prototype:

- Query Broker
  - Net::Stomp
  - XML::Simple
- Munin Module
  - Net::Telnet
  - Net::Stomp
  - XML::Simple
- Request Tracker Module
  - Net::Stomp
  - XML::Simple
  - Scalar::Util
- DokuWiki Module
  - Net::Stomp
  - XML::Simple
- Client

  - Net::Stomp
  - JSON

Testing environment
The prototype query broker, and the three modules were distributed among 4 Dell Optiplex 745 machines, running Debian "squeeze" 6.0.4.

5.3 Data Transfer Formats

This section explains the various ways that data are represented and tagged before passing between the different components in the system; be that the Query Broker, the Modules or clients. Two different ways of tagging data have been used in the prototype. These are XML (Extensible Markup Language) and JSON (JavaScript Object Notation).

XML: is a widely used markup language for structured information. The markup of information is done through the use of tags, with start tags <*some-tag*> and end tags <*/some-tag*>. The tags can have child elements, which allows arranging information in an hierachical fashion. [5]

JSON: is a text-based format for serialization of structured data that can represent strings, numbers, booleans, null, objects and arrays. [6]

The first data transfer format that will be explained, is that of the query sent from the client to the queue containing queries for the Query Broker. It is tagged using XML. The root element of the query is <query>. The <query> element has 4 child elements:

- <user>: the name of the user issuing the query
- <device>: the name of the device the query is issued from
- <service>: the name of the service to search for (can be empty)
- <server>: the name of the server to search for (can be empty)

Here follows an example of such a query:

XML-formated query as sent from the client to the Query Broker

```
<query>
    <user>Bob</user>
    <device>Device01</device>
    <service>Apache2</service>
    <server>Webserver01</server>
</query>
```

The query that is sent from the client to the Query Broker needs to be distributed to all modules. As explained in the architecture chapter, the Query Broker needs to differentiate between different queries from the same user on the same device. This is to avoid old outdated results from being sent to the client. Therefore, in addition to the elements described for the query above, an element <query-number> is added to be able to uniquely identify a query. This is shown in the following example:

XML-formated query sent from the query broker to the modules query queues

```
<query>
    <query-number>1</query-number>
    <user>Bob</user>
    <device>Device01</device>
    <service>Apache2</service>
    <server>Webserver01</server>
</query>
```

The results sent from the modules are also tagged as XML. The individual results are encapsulated in a <module-result> root element. The <module-result> element has the following parameters:

- *user*: the name of the user that issued the query
- *device*: the name of the device the query was issued from
- *query-number*: the number identifying the query
- *source*: the source that provides the data (e.g Munin)
- *expiry*: the number of milliseconds a single result should remain in the result queue of the client, before being removed from the queue.

Each <module-result> can contain an arbitrary amount of results. These results are identified by the <result> element. The <result> element has 3 child elements:

- <title>: a title describing the result
- <classification>: the classification string consisting of 0's and 1's
- <url>: the link to the result content

The following example shows a module-result containing two results:

XML-formated module result as sent from the modules to the module result queue

```
<module-result user="Bob" device="Device01" query-number="1" source="Munin" expiry="20000">
    <result>
        <title>Trend: apache_processes - webserver01</title>
        <classification>0000000000000000000000001000010000</classification>
        <url>http://sysadmin15.iu.hio.no/munin/iu.hio.no/webserver01/apache_processes.html</url>
    </result>
    <result>
        <title>Trend: apache_accesses - webserver01</title>
        <classification>0000000000000000000000000000010000</classification>
        <url>http://sysadmin15.iu.hio.no/munin/iu.hio.no/webserver01/apache_accesses.html</url>
    </result>
</module-result>
```

The next data transfer format represents a single result as received by the client. This format is tagged using JSON. The reason for using JSON when sending results to the client, is its limited overhead, as well as its close connection to JavaScript which is a standard scripting language for displaying dynamic content on the web. The result contains the following elements:

- *priority*: gives the importance of the result, and is used by the client to sort the results
- *url*: the link to the result content
- *title*: a title describing the result
- *source*: the source that provided the data (e.g. Munin)

The following example shows how a such a result would be tagged:

JSON-formated result as sent from the query broker to the client result queue

```
{
    "priority": "1",
    "url": "http://sysadmin15.iu.hio.no/munin/iu.hio.no/webserver01/apache_accesses.html",
    "title": "Trend: apache_accesses - webserver01",
    "source": "Munin"
}
```

The last of the data transfer format is used to let the client know that the search is finished, and that their will be no more incoming results. This format is also tagged using JSON for the same reasons are described earlier. It contains only one element *search-finished*. This element will always contain the value *true* as the message containing this data transfer format will only be sent when the search is finished. The following figure shows the data transfer format:

JSON-formated result as sent from the query broker to the client result queue

```
{ "search-finished": "true" }
```

5.4 Queue Naming Scheme

In order to minimize the configuration needed to allow the components in the prototype to communicate with each other, it is important to have a pre-defined way of naming queues. This ensures that all the components involved will be able to deliver data to each other as long as they follow the naming scheme. The following naming scheme has been employed:

- The queue containing queries from the clients is named *queryQueue*.
- For the modules all queues containing queries for the modules are named *queryQueue* followed by the module name. E.g. for the Munin module the name of its query queue is *queryQueueMunin-Module*.
- Likewise the result queues for the modules are named *resultQueue* followed by the module name. E.g. *resultQueueMunin-Module*.
- The result queues for the clients are named *resultQueue* followed by user name and device name. E.g. the result queue for user *"Bob"* on device *"Device01"*, would be named *resultQueueBobDevice01*.

5.5 Query Broker

This section explains the implementation of the query broker. The source code and related files can be found in the Query Broker section of the appendix.

The query broker is started by running the file *"querybroker.pl"*. It takes three options:

- *-h* Prints a help menu explaining how to run the script
- *-v* Enables verbose mode, which makes the script print some extra information
- *-d* Enables debug mode, which makes the script print debug messages

When the script is started the names of the modules that the system should communicate with are read from the file *"modules.dat"*. This file contains a list of module names separated by new-lines. The following figure shows the content of *"modules.dat"*:

modules.dat

```
DokuWiki-Module
Munin-Module
Request-Tracker-Module
```

Next two connections are made to the ActiveMQ server. One for the incoming queries, sending queries to modules and results to clients, and one for the incoming module results. Then the script subscribes to the *queryQueue*, and all the result queues of the modules. In the context of ActiveMQ, subscribing to a queue/destination entails that the subscriber can get messages from that queue.

At this point the script enters a loop performing the following two tasks until the script is terminated:

- Check if there is a message in the *queryQueue*.

  If this is the case, the message is taken from the queue, decoded, and put in to a hash which is passed to a function that handles query distribution.

- Check if there is a message in one of the result queues.

  If this is the case, a function dealing with incoming results will be called.

5.5.1 Query Distribution

When a query is received from a client, it needs to be distributed to all modules. This is handled by the "send_query" function. However, as specified in the architecture a query-number has to be added to the query before passing it on.

The first thing "send_query()" does is checking whether the their exists a *QueryResult* object for the user and device combination specified in the query. The *QueryResult* is responsible for keeping track of the the query-number for the user and device combination, as well as the number of modules that have sent their results for the query (*moduleResReceived*). If a *QueryResult* does not already exist for the user and device combination, one is created and given a query number. However, if there already exists a *QueryResult* its query number is increased by 1, and the *moduleResReceived* is set to 0. The reason for the query-number is as explained in the architecture, to be able to distinguish among new and outdated module results. The number keeping track of how many modules have sent their results (*moduleResReceived*), is used to determine when all results for a query have been received. This enables the query broker to know when to send the special *search-finished* message to the client.

Next the query is tagged according to the XML data transfer format, and sent to all modules.

5.5.2 Handling Incoming Results

When a message can be read from one of the result queues. The module result is taken out of the queue and converted into a hash. Next the query-number in the module result is checked against the query number stored in the *QueryResult* object for the same user and device combination. If the query number in the module result is lower than that stored in the *QueryResult*, it means that the module result is outdated, and it is simply discarded. Otherwise, if the query numbers match, *moduleResReceived* is increased by 1 and the module result is processed further.

Next all the results contained in the module result are converted into *Result* objects. The *Result* object stores all the information contained in the XML tagged result, and can also hold priority information. Then a profile vector is set based on the user name included in the module result. If the profile vector has been used before it will be stored in a hash, mapping users to profile vectors. Otherwise, the profile vector is created from file, and stored for later use.

All preference profiles are stored in *".profile"* files. E.g. the profile of user "Bob" will be stored in a file named *"Bob.profile"*. The profile files consist of lines of term names and their weights. The following figure shows an example *".profile"* file:

An example .profile file
```
Alert 1.0
Backup 0.0
Configuration 0.8
Customer 0.2
Database 0.0
Firewall 0.0
Hard Drive 0.1
Hardware 0.0
Documentation 1.0
Guarantee 0.0
Hostname 0.0
Installed Software 0.8
Intrusion Detection 0.0
Linux 0.5
Mail 0.0
Mac OS X 0.0
Middleware 0.0
Monitoring 0.3
Network 0.0
Network Interface 0.3
Order 0.0
OS 0.4
Process 0.0
SLA 0.5
Support 1.0
System 0.2
Ticket 1.0
Trend 0.0
Uptime 0.0
Vendor0.5
Web 0.4
Windows 0.0
```

Next priorities are calculated and added to all the *Result* objects.

The priorities are, as specified in the architecture, found by calculating the inner product of the classification vector of each result, and the profile vector matching the name of the user.

Finally the results are tagged according to the JSON data transfer format, and sent to the result queue matching the user and device combination. If results for the query have been received from all the modules, the *search-finished* message will be sent as well. The expiry time provided in the module result is used to calculate the time when the result should expire. The result of the calculation is sent as a parameter when sending the results, this ensures that the results will remain in the result queue for the specified time.

5.6 SQSI-CLI: The Client

As announced in the approach chapter, the client was developed as a CLI script. This section explains the implementation of the the CLI client named SQSI-CLI. The source code can be found in the SQSI-CLI section of the appendix.

Implementation and functionality

The script is started by running the file *"sqsi-cli"*. It takes eight options:

- *-h* Prints a help menu explaining how to run the script
- *-v* Enables verbose mode, which makes the script print some extra information
- *-d* Enables debug mode, which makes the script print debug messages
- *-u <username>* the name used to find the preference profile
- *-s <server name>* the server to search for
- *-c <service name>* the service to search for
- *-i <device name>* the name of the device
- *-r* Toggles including the result ranks as part of the output

As an example, searching for the service *apache*, at server *webserver01*, as user *Bob*, on device *iPad01* will require running the script in the following way:

Running the client
```
./sqsi-cli -c "apache" -s "webserver01" -u "Bob" -i "iPad01"
```

The client starts by connecting to the ActiveMQ server and subscribing to its result queue. Next a query message is sent to the query queue of the query broker with data taken from the parameters supplied by the user. Then the script enters a loop continuously checking whether there are any messages in the result queue.

When there is a message available to read from the result queue, the script takes the message out of the queue, converts the JSON coded message to a hash, inserts the result hash into an array, sorts the array based on the priorities of the results, and finally prints all results contained in the array. In the case that the message is not a result, but rather a *search-finished* message, a *SEARCH_FINISHED* flag is set, and the process of collecting messages from the result queue is continued until no more messages can be read from the result queue.

The following examples shows the resulting output from running the sqsi-cli script searching for the service *apache* at server *sysadmin14.iu.hio.no*. The *-r* option has been set to include the result ranks in the output.

Example output from sqsi-cli

```
1.7   Documentation: apache-sysadmin14.iu.hio.no                http://someurl01.com   DokuWiki
0.7   Trend: apache_processes - sysadmin14.iu.hio.no            http://someurl02.com   Munin
0.5   Trend: apache_accesses - sysadmin14.iu.hio.no             http://someurl03.com   Munin
0.5   Trend: apache_volume - sysadmin14.iu.hio.no               http://someurl04.com   Munin
0.1   Support: Apache poor performance sysadmin14.iu.hio.no     http://someurl05.com   Request Tracker
```

The first column displays the rank for the individual results. The second column displays the title. The third column the url (the urls have been exchanged in the example in order to fit the page). The fourth column shows the source from which the data originates.

5.7 The Modules

This section features detailed explanations of the three modules that were developed for the prototype. That is, how they work, how they gather information from their respective service/contributor, how they match information with queries, and how they classify that information.

All the modules need to fulfil the requirements of the architecture. That is, they need to be able to perform the following tasks:

- Get a query from their query queue
- Interpret the query
- Gather information from the their service/contributor
- Find relevant information according to the query
- Classify the results
- Tag the results according to the specified format
- Send the set of results to their result queue

5.7.1 Munin Module

As explained in the Approach chapter, Munin is a networked resource monitoring tool. It has a master/node architecture, which means that the master gathers data from all nodes. The job of the Munin module is to provide functionality that allows users to search for and get information from a Munin master and its nodes. The source code and related files can be found in the Munin Module section of the appendix.

Module Implementation and Functionality

The module is started by running the file *"munin-module.pl"*. It takes three options:

- *-h* Prints a help menu explaining how to run the module
- *-v* Enables verbose mode, which makes the module print some extra information
- *-d* Enables debug mode, which makes the module print debug messages

When the module is started, it first reads the file "classifications.dat" which contains a list of plugin names and classification strings. The following is an excerpt from classifications.dat:

Excerpt of classfications.dat

```
if_err_eth0          0000000000000000001000000010000
if_eth0              0000000000000000001000000010000
interrupts           0000000000000000000000001010000
iostat               0000010000000000000000000010000
iostat_ios           0000010000000000000000000010000
irqstats             0000000000000000000000001010000
load                 0000000000000000000000001010000
memory               0000000000000000000000001010000
munin_stats          0000000000000000000000000010000
mysql_bin_relay_log 0000100000000000000000000010000
```

Each of the plugins corresponds to a web page with graphs in the Munin web interface at the Munin server. The contents of the file is then stored in a hash map in order to later allow looking up plugin names and finding the corresponding classification string.

In order to find which nodes the Munin master is responsible for, the "munin.conf" file is read. This file contains descriptions of the Munin nodes, among other things.

Being that different Munin nodes will run different plugins depending on the services they run, the module need to check which plugins are running on each of the nodes. This is accomplished through telnet communication with the nodes. The munin-node software installed on each of the nodes, allows telnet connections to be made to them, and has a set of commands that can be run to get information about them. One of these commands allows listing which plugins are running. In combinations with the node information from "munin.conf", this allows creating a map of node names and their plugins.

Next, in order to allow the module to respond to searches for services the file *"services.dat"* was made. This file contains a list of service names and their related plugins. Here follows an excerpt of *"services.dat"*:

Excerpt of services.dat

```
apache,apache_accesses,apache_processes,apache_volume
munin,munin_stats
exim,exim_mailstats,exim_mailqueue
```

The "services.dat" file is read and a hash map allowing looking up services and finding their related plugins is created.

Next a connection is made to the server running ActiveMQ and the module subscribes to the queue containing its queries.

From this point on the module performs the following tasks in a loop until the module is stopped:

1. Gets a query from the query queue and converts it into a hash
2. Creates the <module-result> meta data container based on data from the query and module specific data.
3. Finds the plugins relevant to the query, generates urls based on plugin names, gets the classification vectors by plugin name, and tags each result according to the data transfer format.
4. Sends the results encapsulated in the <module-result> container to the result queue.

Finding relevant plugins:

The way that the relevant plugins for a query are found, needs further explanation. A query can have 4 different states:

1. *The query asks for both a server and a service that the module has relevant data for.*

   In this case the module checks which of the plugins for the service that are available at the specified server. This is done by cross checking the hash containing the service to plugins mapping, with the hash containing the server to plugins mapping. The result of this cross check is a set of plugins which should be tagged as results.

2. *The query asks for just a server, and the module has relevant data.*

   In this case the hash containing the server to plugins mapping contains the name of all the plugins which should be tagged as results.

3. *The query asks for just a service, and the module has relevant data.*

   In this case the hash containing the service to plugins mapping contains the name of all the plugins which should be tagged as results.

4. *The query asks for unknown server and service, or has not specified any server and service.*

   In this case the module will not have any results to tag.

5.7.2 Request Tracker Module

The Request Tracker Module provides the functionality to allow searching for and accessing the tickets on a Request Tracker installation. The source code and related files can be found in the Request Tracker Module section of the appendix.

Module Implementation and functionality

The module is started by running the file *"rt-module.pl"*. It takes three options:

- *-h* Prints a help menu explaining how to run the module
- *-v* Enables verbose mode, which makes the module print some extra information
- *-d* Enables debug mode, which makes the module print debug messages

In order to allow the module to classify different types of tickets, different queues can be created in the Request Tracker installation. Request Tracker has its own queues that can be used to place different types of tickets into different categories. For instance, a Request Tracker queue dedicated to alerts may be created, and all tickets that are alerts can be inserted into this queue. Therefore, when the module is started it first reads the file *"queue_classifications.dat"*. This file contains the names of the Request Tracker queues and their classification vectors. The data in the file is stored in a hash, mapping Request Tracker queues to their classification vectors. The following figure shows the contents of *"queue_classifications.dat"*:

queue_classifications.dat
```
Alert      100000000000000000000000100000
General  000000000000000000000000100000
Order     000000000000000000100000100000
Support   000000000000000000000010100000
```

Next the module connects to the ActiveMQ server and subscribes to its query queue. From this point on the module performs the following tasks in a loop until the module is stopped:

1. Gets a query from the query queue and converts it into a hash
2. Creates the <module-result> meta data container based on data from the query and module specific data.
3. Finds the tickets relevant to the query, generates urls based on ticket ids, gets the classification vectors by Request Tracker queue name, and tags each result according to the data transfer format.
4. Sends the results encapsulated in the <module-result> container to the result queue.

Communicating with Request Tracker:

Communication with Request Tracker is handled through the Request Tracker Command Line Interface (CLI). The user running the module needs to set up a .rtrc file in order to allow running the CLI. This .rtrc file has to contain the url to the Request Tracker web interface, and the Request Tracker username and password. The following shows an example .rtrc file:

example .rtrc file
```
server http://sysadmin16.iu.hio.no/rt
user root
passwd thepassword
```

The CLI allows queries to be made to the Request Tracker database. The module issues the following command in order to get a list of all tickets, and show their ticket ids, subjects and queue names:

CLI command
```
rt ls -f queue,subject
```

Finding relevant tickets:

When the ticket information has been collected from Request Tracker, the tickets matching the query are found by searching the subject fields of the tickets for matching words. Extending the search to include the ticket bodies could easily be done, however it would slow the search down and is therefore not implemented.

5.7.3 DokuWiki Module

The DokuWiki module provides access to, and classification of the documents stored in a DokuWiki installation. The source code and related files can be found in the DokuWiki Module section of the appendix.

Module Implementation and functionality

The module is started by running the file *"dokuwiki-module.pl"*. It takes three options:

- *-h* Prints a help menu explaining how to run the module
- *-v* Enables verbose mode, which makes the module print some extra information
- *-d* Enables debug mode, which makes the module print debug messages

The module starts by reading the file *"classifications.dat"*. This file contains a set of terms and their classification vectors. The following figure shows the contents of *"classifications.dat"*:

classifications.dat

```
hardware  0000000110000000000000000000000
software  0000000010010000000000000000000
mysql     0000100010000000000000000000000
apache    0000000010000000000000000000010
```

The terms in *"classifications.dat"* are used in slightly different ways. The *hardware* and *software* terms are general purpose terms used to describe hardware and software related documentation respectively. The last two terms *mysql* and *apache* are used to describe documentation related to their respective services. For each service that the DokuWiki server has stored information about, a corresponding term and its classification vector should be added to *"classifications.dat"*.

Next the module connects to the ActiveMQ server and subscribes to its query queue. From this point on the module performs the following tasks in a loop until the module is stopped:

1. Gets a query from the query queue and converts it into a hash
2. Creates the <module-result> meta data container based on data from the query and module specific data.
3. Finds the documents relevant to the query, generates urls based on document names, gets the classification vectors based on document names, and tags each result according to the data transfer format.
4. Sends the results encapsulated in the <module-result> container to the result queue.

Finding relevant documents:

As explained earlier, DokuWiki stores its files as text files, without the use of a database. When a new document is created through the DokuWiki web interface, it is stored as a *".txt"* file in a folder on the DokuWiki server. This allows the module to check for the existence of a document by checking if a *".txt"* file with a corresponding name exists in the document folder. This means that as long as documents follow a strict naming scheme, the module is able to find documents relevant to a query. The naming scheme chosen for the module and DokuWiki is as follows:

- General information documents about a service are simply given the name of the service in lower case letters. E.g. *apache*.
- Documents regarding a specific service running on a specific server, are given names in lower case letters by service name followed by a *"-"* followed by the server name. E.g. *apache-webserver01*.
- Documents regarding hardware for a server, are given names in lower case letters starting with *hardware* followed bye a *"-"* followed by the server name. E.g. *hardware-webserver01*.
- Documents regarding the software installed on a server, are given names in lower case letters starting with *software* followed bye a *"-"* followed by the server name. E.g. *software-webserver01*.

If both *server* and *service* is specified by the query. The module checks for the existence of a document with a name composed of the server name and service names in line with the naming scheme. If just the server is specified the module will check for any document containing the server name. Lastly if only the service name is specified, the module will check for a document matching the service name.

5.8 Functionality Testing

In order to verify the correct functioning of the prototype, functionality testing was performed. The functionality testing was done from the perspective of the client (sqsi-cli). The following list details the required functionality:

- Users should be able to search for information related to a service, a server or both, and receive relevant information from all modules that have information relevant to the search.
- The results should be prioritized in accordance with the preference profile of the user.
- The results should be presented to the user according to rank of the results, and give access to the information via urls.

To allow the functionality of the prototype to be tested, test data was created in the three information source installations: Munin, Request Tracker, and DokuWiki.

To be able to observe how the system is able to prioritize information according to different user preferences, two fictional users, *Bob* and *Alice* were created.

The preference profile for *Bob* was designed for a person working with support cases and responding to alerts. The following terms were given high priorities (0.8 - 1.0):

- Alert
- Configuration
- Documentation
- Installed Software
- Support
- Ticket

The other terms were given weights of 0.5 or lower. Bob's entire preference profile can be found in the appendix.

The preference profile for *Alice* was designed for a person mainly interested in trend, hardware, and system related information. The following terms were given high priorities (0.8 - 1.0):

- Hardware
- Documentation
- Guarantee/Licence
- Process
- System
- Trend

The other terms were given weights of 0.5 or lower. Alice's entire preference profile can be found in the appendix.

The Query Broker and the modules where all run on different servers, similar to the layout seen in figure 5.1 on page 45.

The following searches were performed for both users:

- **S.1** A search for the service *"apache"*

```
.\sqsi-cli -u Username -s "" -c "apache" -i Device01 -r
```

- **S.2** A search for the service *"apache"* and the server *"sysadmin14.iu.hio.no"*.

```
.\sqsi-cli -u Username -s "sysadmin14.iu.hio.no" -c "apache" -i Device01 -r
```

- **S.3** A search for the server *"sysadmin14.iu.hio.no"*

```
.\sqsi-cli -u Username -s "sysadmin14.iu.hio.no" -c "" -i Device01 -r
```

The output from sqsi-cli for **S.1** is displayed below. All urls have been replaced by *"http://url"* in order for the output to fit the page.

S.1 Bob

```
2     Support: Apache poor performance sysadmin14.iu.hio.no   http://url Request Tracker
2     Support: Please update the apache documentation: sysadmin17.iu.hio.no  http://url Request Tracker
2     Alert: sysadmin15.iu.hio.no running apache is down!    http://url Request Tracker
1.8    Documentation: apache   http://url DokuWiki
1     Order: Server running apache    http://url Request Tracker
0     Trend: apache_accesses - sysadmin15.iu.hio.no   http://url   Munin
0     Trend: apache_processes - sysadmin15.iu.hio.no  http://url  Munin
0     Trend: apache_volume - sysadmin15.iu.hio.no     http://url Munin
0     Trend: apache_accesses - sysadmin14.iu.hio.no   http://url   Munin
0     Trend: apache_processes - sysadmin14.iu.hio.no  http://url Munin
0     Trend: apache_volume - sysadmin14.iu.hio.no     http://url Munin
0     Trend: apache_accesses - sysadmin16.iu.hio.no   http://url Munin
0     Trend: apache_processes - sysadmin16.iu.hio.no  http://url Munin
0     Trend: apache_volume - sysadmin16.iu.hio.no     http://url Munin
0     Trend: apache_accesses - sysadmin17.iu.hio.no   http://url   Munin
0     Trend: apache_processes - sysadmin17.iu.hio.no  http://url Munin
0     Trend: apache_volume - sysadmin17.iu.hio.no     http://url Munin
```

S.1 Alice

```
8     Trend: apache_processes - sysadmin15.iu.hio.no  http://url Munin
8     Trend: apache_processes - sysadmin14.iu.hio.no  http:/url Munin
8     Trend: apache_processes - sysadmin16.iu.hio.no  url Munin
8     Trend: apache_processes - sysadmin17.iu.hio.no  http://url Munin
1     Documentation: apache   http://sysadmin17.iu.hio.no/dokuwiki/doku.php?id=apache DokuWiki
1     Trend: apache_accesses - sysadmin15.iu.hio.no   http://url   Munin
1     Trend: apache_volume - sysadmin15.iu.hio.no     http://url Munin
1     Trend: apache_accesses - sysadmin14.iu.hio.no   http://url Munin
1     Trend: apache_volume - sysadmin14.iu.hio.no     http://url Munin
1     Trend: apache_accesses - sysadmin16.iu.hio.no   http://url Munin
1     Trend: apache_volume - sysadmin16.iu.hio.no     http://url Munin
1     Trend: apache_accesses - sysadmin17.iu.hio.no   http://url Munin
1     Trend: apache_volume - sysadmin17.iu.hio.no     http://url Munin
0.5    Alert: sysadmin15.iu.hio.no running apache is down!    http://url Request Tracker
0     Support: Apache poor performance sysadmin14.iu.hio.no   http://url Request Tracker
0     Order: Server running apache    http://url Request Tracker
0     Support: Please update the apache documentation: sysadmin17.iu.hio.no  http:/url Request Tracker
```

From the output from **S.1** it can be seen that results where received from all modules (Munin, DokuWiki and Request Tracker), and that all results are related to the service *"apache"*. Furthermore, the results are sorted by rank, as can be seen in the leftmost column of the output.

Comparing the output of **S.1** for *Bob* and *Alice* it can clearly be seen that while Bob gets the results related to support and alerts highest, Alice have these same results at the bottom, while trend and process related results get the highest priority in her case. This is in line with the preference profiles of both Bob and Alice.

The output from sqsi-cli for **S.2** is displayed below. All urls have been replaced by *"http://url"* in order for the output to fit the page.

S.2 Bob

```
2     Support: Apache poor performance sysadmin14.iu.hio.no   http://url Request Tracker
1.8   Documentation: apache-sysadmin14.iu.hio.no     http://url    DokuWiki
0     Trend: apache_accesses - sysadmin14.iu.hio.no   http://url Munin
0     Trend: apache_processes - sysadmin14.iu.hio.no  http://url Munin
0     Trend: apache_volume - sysadmin14.iu.hio.no     http://url Munin
```

S.2 Alice

```
8     Trend: apache_processes - sysadmin14.iu.hio.no  http://url Munin
1     Documentation: apache-sysadmin14.iu.hio.no     http://url    DokuWiki
1     Trend: apache_accesses - sysadmin14.iu.hio.no   http://url   Munin
1     Trend: apache_volume - sysadmin14.iu.hio.no     http://url Munin
0     Support: Apache poor performance sysadmin14.iu.hio.no   http://url Request Tracker
```

As with the output from **S.1**, from the output of **S.2** it can be seen that results where received from all modules. Also, all results are related to the service *"apache"* and the server *"sysadmin14.iu.hio.no"*.

Comparing the output of **S.2** for *Bob* and *Alice* the same ranking as in **S.1** can be observed. This is due to the fact that the results in **S.2** are a subset of the results from **S.1**.

The output from sqsi-cli for **S.3** is displayed below. All urls have been replaced by *"http://url"* in order for the output to fit the page.

S.3 Bob

```
2      Alert: Server down: sysadmin14.iu.hio.no        http://url Request Tracker
2      Support: Apache poor performance sysadmin14.iu.hio.no   http://url Request Tracker
1.8     Documentation: software-sysadmin14.iu.hio.no    http://url  DokuWiki
1.8     Documentation: apache-sysadmin14.iu.hio.no      http://url    DokuWiki
1      Documentation: hardware-sysadmin14.iu.hio.no    http://url  DokuWiki
0.8     Trend: open_inodes - sysadmin14.iu.hio.no       http://url Munin
0.8     Trend: irqstats - sysadmin14.iu.hio.no  http://url  Munin
0.8     Trend: if_eth0 - sysadmin14.iu.hio.no   http://url   Munin
0.8     Trend: swap - sysadmin14.iu.hio.no      http://url      Munin
0.8     Trend: uptime - sysadmin14.iu.hio.no    http://url    Munin
0.8     Trend: load - sysadmin14.iu.hio.no      http://url      Munin
0.8     Trend: cpu - sysadmin14.iu.hio.no       http://url       Munin
0.8     Trend: open_files - sysadmin14.iu.hio.no        http://url Munin
0.8     Trend: memory - sysadmin14.iu.hio.no    http://url    Munin
0.8     Trend: if_err_eth0 - sysadmin14.iu.hio.no       http://url       Munin
0.8     Trend: entropy - sysadmin14.iu.hio.no   http://url   Munin
0.8     Trend: users - sysadmin14.iu.hio.no     http://url     Munin
0.8     Trend: interrupts - sysadmin14.iu.hio.no        http://url        Munin
0.8     Trend: proc_pri - sysadmin14.iu.hio.no  http://url  Munin
0      Trend: http_loadtime - sysadmin14.iu.hio.no     http://url Munin
0      Trend: apache_accesses - sysadmin14.iu.hio.no   http://url Munin
0      Trend: df - sysadmin14.iu.hio.no        http://url        Munin
0      Trend: df_inode - sysadmin14.iu.hio.no  http://url  Munin
0      Trend: iostat - sysadmin14.iu.hio.no    http://url    Munin
0      Trend: forks - sysadmin14.iu.hio.no     http://url     Munin
0      Trend: exim_mailqueue - sysadmin14.iu.hio.no    http://url    Munin
0      Trend: vmstat - sysadmin14.iu.hio.no    http://url    Munin
0      Trend: fw_packets - sysadmin14.iu.hio.no        http://url        Munin
0      Trend: apache_processes - sysadmin14.iu.hio.no  http://url Munin
0      Trend: processes - sysadmin14.iu.hio.no http://url Munin
0      Trend: apache_volume - sysadmin14.iu.hio.no     http://url     Munin
0      Trend: iostat_ios - sysadmin14.iu.hio.no        http://url        Munin
0      Trend: threads - sysadmin14.iu.hio.no   http://url   Munin
0      Trend: exim_mailstats - sysadmin14.iu.hio.no    http://url    Munin
```

S.3 Alice

```
8.8   Trend: proc_pri - sysadmin14.iu.hio.no  http://url  Munin
8     Trend: forks - sysadmin14.iu.hio.no     http://url     Munin
8     Trend: vmstat - sysadmin14.iu.hio.no    http://url    Munin
8     Trend: apache_processes - sysadmin14.iu.hio.no  http://url Munin
8     Trend: processes - sysadmin14.iu.hio.no http://url Munin
8     Trend: threads - sysadmin14.iu.hio.no   http://url   Munin
2     Documentation: hardware-sysadmin14.iu.hio.no    http://url  DokuWiki
2     Trend: uptime - sysadmin14.iu.hio.no    http://url    Munin
1.8   Trend: open_inodes - sysadmin14.iu.hio.no      http://url Munin
1.8   Trend: irqstats - sysadmin14.iu.hio.no  http://url  Munin
1.8   Trend: if_eth0 - sysadmin14.iu.hio.no   http://url   Munin
1.8   Trend: swap - sysadmin14.iu.hio.no      http://url      Munin
1.8   Trend: load - sysadmin14.iu.hio.no      http://url     Munin
1.8   Trend: cpu - sysadmin14.iu.hio.no       http://url       Munin
1.8   Trend: open_files - sysadmin14.iu.hio.no       http://url Munin
1.8   Trend: memory - sysadmin14.iu.hio.no    http://url    Munin
1.8   Trend: if_err_eth0 - sysadmin14.iu.hio.no      http://url Munin
1.8   Trend: entropy - sysadmin14.iu.hio.no   http://url   Munin
1.8   Trend: users - sysadmin14.iu.hio.no     http://url     Munin
1.8   Trend: interrupts - sysadmin14.iu.hio.no       http://url        Munin
1.1   Documentation: software-sysadmin14.iu.hio.no    http://url  DokuWiki
1     Documentation: apache-sysadmin14.iu.hio.no     http://url    DokuWiki
1     Trend: http_loadtime - sysadmin14.iu.hio.no     http://url    Munin
1     Trend: apache_accesses - sysadmin14.iu.hio.no   http://url Munin
1     Trend: df - sysadmin14.iu.hio.no        http://url        Munin
1     Trend: df_inode - sysadmin14.iu.hio.no  http://url  Munin
1     Trend: iostat - sysadmin14.iu.hio.no    http://url    Munin
1     Trend: exim_mailqueue - sysadmin14.iu.hio.no    http://url    Munin
1     Trend: fw_packets - sysadmin14.iu.hio.no       http://url Munin
1     Trend: apache_volume - sysadmin14.iu.hio.no    http://url     Munin
1     Trend: iostat_ios - sysadmin14.iu.hio.no       http://url Munin
1     Trend: exim_mailstats - sysadmin14.iu.hio.no    http://url Munin
0.5   Alert: Server down: sysadmin14.iu.hio.no       http://url Request Tracker
0     Support: Apache poor performance sysadmin14.iu.hio.no   http://url Request Tracker
```

From the output from **S.3** it can be seen that all the results are related to the server *"sysadmin14.iu.hio.no"*, and that all modules have contributed with results. Again, as with **S.1** and **S.2** it can be observed that the ranking of the results for Bob and Alice are in accordance with their preference profiles.

It should be mentioned that all urls for all results were functional, and linked to the correct resources.

Chapter 6

Discussion

In this chapter the resulting architecture, prototype and results from the functionality testing are discussed. Possible modifications to the architecture and future work is suggested, and some of the choices made in the architecture are discussed in relation to the research introduced in the background chapter.

6.1 The Prototype

In the approach chapter of this thesis, a set of tasks (seen in figure 3.1 on page 24) were defined that needed to be completed in order for the architecture to fulfil its purpose. The results of these tasks constitute the architecture, which forms the basis for the prototype. The purpose of developing the prototype was to demonstrate the validity of the architecture, that it can be applied in a real environment, and that it facilitates efficient information retrieval for system administrators.

The results from developing the prototype showed that the different elements in the architecture can be implemented to create a working system. Furthermore, it demonstrated how modules can be developed for services used by system administrators. It also showed the ease of which modules can be included into the system when following the requirements of the architecture. This was made possible by the use of Message-oriented Middleware, which enables asynchronous communication, and the loose coupling between the client, the query broker and the modules. The loose coupling does not only facilitate easy integration of modules, it also allows different *types* of clients to communicate with the system.

6.1.1 Changes to the Data Transfer Formats

In retrospect, when looking at the different xml data transfer formats suggested in the architecture and those implemented in the prototype, it becomes clear that some of the names applied to the different elements might not be ideal. That is, some of the names may introduce confusion as to the meaning of the elements. Therefore, the following changes are proposed:

For the queries:

- <user> should be changed into a parameter for the <query> tag.
- <device> should be renamed to client-device, and changed to a parameter for the <query> tag.
- <query-number> should be changed to a parameter for the <query> tag.
- <service> should be renamed to <target-service>
- <server> should be renamed to <target-server>

These changes should make the role of the different elements more obvious, and help distinguish between meta data and content.

Revised query format

```
<query user="Bob" client-device="Device01" query-number="1" >
      <target-service>Apache2</target-service>
      <target-server>Webserver01</target-server>
</query>
```

These same changes also apply to the query sent from the client. The only difference being that *query-number* is not part of that query. For the *module-result* data transfer format, the parameter *device* should be renamed to *client-device*.

6.1.2 Functionality Testing

Looking at the results from the functionality testing of the prototype, it is evident that the prototype is able to gather information from multiple sources, prioritize the information according to preference profiles, and present the results to the user. This gives a clear indication of the validity of the architecture designed in this thesis, and shows how it can be used to help system administrators gather information in an efficient manner. However, what has not been addressed is how *well* the prioritization relates to user preferences. This was not investigated due to time constraints. In the future, this would be a valuable aspect to investigate. This could for instance be done with the help of user testing, where users create their own preference profiles, perform a series of searches, and report of the relevance of the results in accordance with the ranking.

6.2 Architecture

The architecture designed in this thesis, is aimed at facilitating efficient information retrieval for system administrators. One of the key features of the architecture utilized to achieve this, is the ability to gather information from the multitude of information sources that system administrators are reliant on, and allowing access to and presenting this information at a single point. Secondly, personalizing information is achieved through ranking search results based on user preferences. In order to design an architecture which allows for easy expansion and integration of information sources, a modular design was chosen. This was done to allow for reuse of the information already available in existing systems, rather than developing an all encompassing system.

PIA
The idea of collecting data from multiple sources and presenting the information at a common point is not new. As explained in the background chapter, Albayrak et al. [1] developed an agent-based personal information system called PIA. Their system uses agents to collect data from different sources. However, as opposed to the architecture designed in this thesis, PIA deals with documents, and the documents collected by PIA extraction agents is constantly collected and stored in a database. The database is then accessed by filtering agents, which filter the documents. Personal agents which manages individual information provisioning, communicates with the filtering agents and the presentation layer.

The PIA approach to information retrieval does not translate directly to the domain of system administration. This is partly due to the fact that the systems that deliver information to system administrators, do not necessarily provide that information in the form of documents. Also, the dynamic nature of the field of system administration entails that information can change rapidly. Therefore, it is imperative that the information available to the retrieval system is always up to date, and that the system is able to keep up with changes in the information space. For instance, alerts from monitoring systems and tickets can be of critical importance, and it is therefore a necessity that they are included in the information provided by the retrieval system within a relatively short period of time. Therefore, the base approach in the architecture designed in this thesis, is to directly access the information sources each time a query is issued. However, to alleviate the demands on the modules and query broker, caching can be implemented in the system, as long as care is taken with regards to keeping information up to date.

6.2.1 Caching

The use of caching is not an inherent part of the architecture itself, however the way the architecture is designed enables caching to be implemented in an implementation based on the architecture. Being that prioritization is done after a set of results have been returned by a module, caching can be done independently of preference profiles. This means that results can be cached and indexed by the queries that produced them. That is, a set of search results would be indexed by a service, a server or a combination of the two. For example: *"Apache2-Webserver01"*. This will allow reuse of results, which should increase search speed, and decrease the use of resources in modules and their related services. This is because the only thing that needs to be recomputed is the actual prioritization. The query broker should be responsible for handling caching functionality being that it can keep track of the results from all modules. However, to make caching work in favour of the system, it is imperative that results are updated relatively often to avoid getting outdated results, and to make sure new information is included in the results. This can be achieved by adding timestamps to the cached results, and removing them from the cache after a certain period of time has elapsed. The inclusion of caching functionality in the query broker is illustrated in figure 6.1 on page 72.

6.2.2 Prioritization

What has not been mentioned previously regarding the prioritization scheme in the architecture, is the underlying assumption that the more terms a result is related to, the more important the result is considered to be. Put another way, the more terms a result is related to, the higher is the likelihood that it will be ranked high, simply due to the fact that it covers more of the classification, and hence more of the preference profiles. While the assumption may be true in a lot of cases, it is important to note that there may be results which can not naturally be linked to many terms, but still are inherently important, and may therefore get an unfair chance. This could possibly be alleviated through the introduction of local importance.

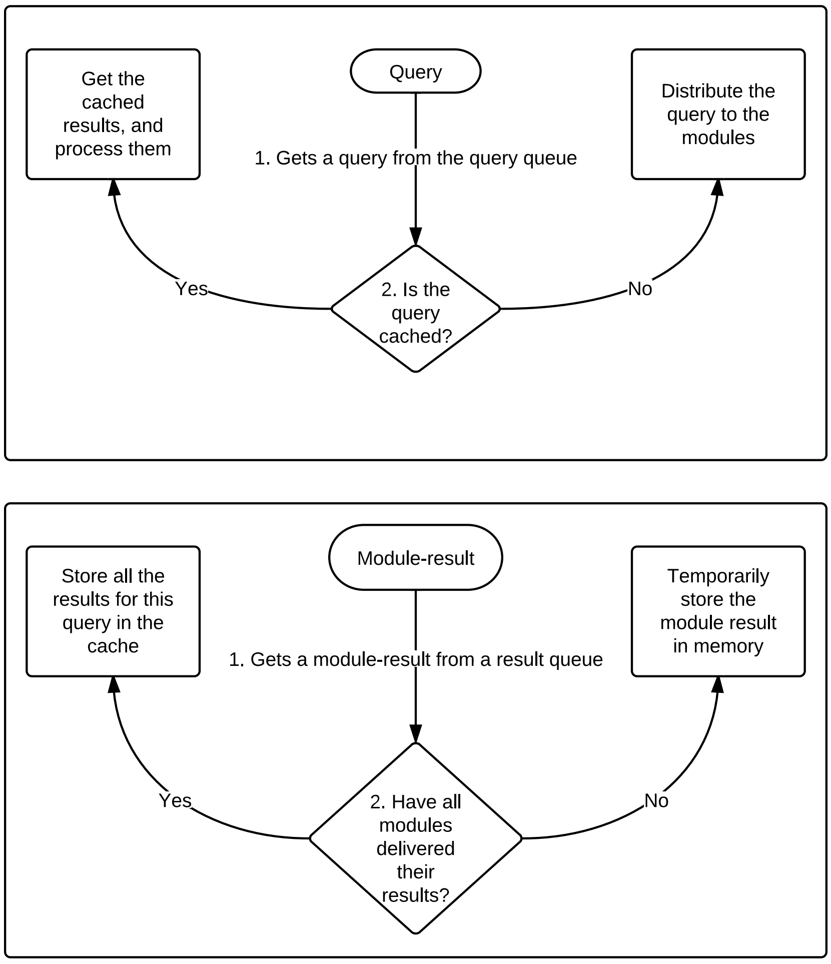
Get the cached results, and process them
Query
Distribute the query to the modules
1. Gets a query from the query queue
Yes
2. Is the query cached?
No
Store all the results for this query in the cache
Module-result
Temporarily store the module result in memory
1. Gets a module-result from a result queue
Yes
2. Have all modules delivered their results?
No

Figure 6.1: Caching functionality in the Query Broker

Introducing local importance

The way the architecture is designed, the modules have very limited power when it comes to adjusting importance of results. The modules are limited to classifying the results according the relevant terms. An interesting prospect to explore is to allow modules to signify the local importance of results. That is, the importance of a result compared to the importance of other results provided by a module, independent of user preferences. This could give the potential of for instance newly updated documentation or a specifically important alert to stand out. One way this could be implemented, is to allow modules to flag results as important. The query broker could then add a small weight to the rank of the results that are flagged as important. This should give the opportunity of differentiating between otherwise equally classified results. Another possibility is to apply a weight instead of a flag to allow more differentiation. However, giving the modules much power in determining the importance of results, may introduce an imbalance between the results from the different modules. Investigating different strategies for introducing local importance, their value and impact is an interesting topic for future research.

Prioritization on different layers

The architecture designed in this thesis can be viewed as a three tier architecture. With a presentation layer, a Business Logic Layer (BLL) (the query broker), and a Data Access Layer (DAL) (the modules). In practice the prioritization of results could be performed in any of these layers.

Performing the prioritization in the presentation layer would mean that the presentation layer would need to have information about the user (preference profiles), and contain the logic needed to perform the prioritization. This would lead to a thick client, where presentation and logic is not separated, and developing different types of clients would be much more complex. Prioritization does not necessarily need to be performed after results have arrived however. Some prioritization can be done beforehand by having knowledge of the user, and more advanced queries. These queries can then contain more detailed information about the preferences of the users, and be used to limit what information is sent back to the user. However, this form of query modification can also be performed in the business logic layer, so there is no real advantage to performing prioritization in the presentation layer.

In the architecture prioritization is performed in the BLL. This entails that only the query broker needs to have knowledge of the users. Query modification could be implemented here. The advantage of query modification is that potentially, less data may be transferred between the modules and the query broker since the modules can eliminate data that is of no interest to the user. However, this means that the user does not have access to all data for a search, as opposed to having the data available but ranked low and presented last in a list. Also, implementing query modification renders caching independent of preference profiles impossible, which means caching results looses its benefits. In addition to that, it increases the complexity of module development.

Lastly prioritization can be done in the DAL. In the architecture the DAL consists of the modules. Performing the prioritization in the modules has the same benefit as query modification of leading to less data being sent to the query broker, but also the same drawbacks. It also means that every module needs access to the preference profiles. Most importantly, having the prioritization in the modules leads to higher complexity in module development.

With this knowledge in mind, it becomes quite evident that performing the prioritization in the query broker is a good choice.

User feedback and additional importance vectors

With the current design of the architecture user preferences are static, meaning that they do not change unless explicitly modified in the preference profiles. It is however possible to let the system learn from user behaviour, either through explicit or implicit feedback. Explicit feedback could be implemented by allowing users to rate the results they get from searching. For instance by having a simple +/- button for each result, allowing the user to indicate that the result is either ranked to low or to high. Implicit feedback could be implemented by recording which result links a user clicks on.

Salton et al. [22] explains how relevance feedback has been used in information retrieval for query reformulation of vector queries. With vector queries, queries consist of sets of weighted search terms. The weight of the terms in the queries can then be modified by feedback from the user in order to better the search.

In the architecture the queries are limited to servers and services, so query reformulation is not applicable. However, if the results sent to the clients were to contain the classification vectors, the preferences of a user could be modified by changing the weights of the terms corresponding to those in the classification vector of the results. These changes could be applied directly to a preference profile. However, this may lead to the preference profiles being imbalanced until enough searches have been performed by the user for the feedback to be representational of the average usage pattern of the user. An alternative to directly modifying the preference profiles is to introduce a new vector; a learning vector. So instead of modifying the preference profile the learning vector could be modified by relevance feedback, and when enough data has been collected about the usage pattern of the user, the learning vector can be applied in the prioritization process. The inclusion of relevance feedback is a topic that could be explored in future research.

The possibility to include a learning vector, illustrates how the prioritization scheme implemented in the architecture can be extended to take into account multiple dimensions of importance by representing them as vectors.

6.2.3 Generating Preference Profiles

Generating preference profiles is a subject that has not been explored to its full extent in this thesis. As described in the architecture, preference profiles can be generated by allowing the users to directly assign weights to the different terms in the classification. This process could potentially be made to require less effort from the user, by providing profile templates for different user types or job responsibilities. Then a user could start out with a profile template, and modify that if needed.

An alternate way of generating preference profiles is to devise a profile generation game. Such a game could be based on recognition of terms, and comparing terms for importance. One possible approach is to create a knock-out tournament game. In such a game two and two terms will be set up against each other, and it will be up to the user to determine which of the terms are most important. The 'winning' terms will go on to a winners round, and the losers to a losers round with the same structure of setting two and two terms up against each other. This will then continue until all winners and losers have been determined. This allows all terms to be ranked from first to last, and weights can then be assigned accordingly. However, the knock-out tournament approach has some drawbacks. Firstly the number of terms has to be a power of 2, which means the classification has to be adjusted to fit the profile generation game, which is by no means optimal. Also, there's the issue of which terms to match against each other. If chosen randomly, one runs the risk of matching two terms that very important to the user against each other early in the game. If this happens in the first round of the game, then the looser of the match will instantly be ranked lower than half of the terms in the classification. If the competing terms are to be predetermined, then there's the challenge of choosing the correct terms. This very difficult, since it will depend on the preferences of the user.

A game which does not have the drawbacks of the knock-out tournament, is the round-robin tournament. In this type of tournament all terms will be matched against each other. However, the major drawback of this approach is that it is likely that profile generation will take a long time. Therefore, it is not a user friendly approach.

Looking into alternative ways of generating preference profiles to make the process both accurate and user friendly, is an interesting topic for future research.

6.2.4 Presentation Layer

In the architecture, it is always the case that all results related to a search are returned to the presentation layer, meaning that no results are removed on the basis of user preferences. This means that all the information is available to the user, it is ranked, and low ranked information will be displayed last. This is in line with Endsly [7] stating that all information that is needed should always be present in order to allow the user to form an accurate mental model of a system or situation. However, Endsly also emphasizes that the amount of information should not be overwhelming. This can be handled by not presenting all results from a search, but instead allow the user to choose whether or not to display more results.

As explained in the background chapter Nah [20] performed a study of tolerable waiting times in relation to web users and their willingness to wait for web pages to appear. Nah found that providing feedback to the user could prolong the time a user is willing to wait. In the architecture feedback is provided by allowing the results from the different modules to be processed and sent to the user independently of each other. This means that possibly slow modules does not impact the time a user has to wait to see some results. As soon as some results are available, they will be presented to the user, which should help alleviate user impatience. Nah also found that their results indicated that web users tolerable waiting time peaks at about 2 seconds. During the functionality testing, waiting time was not an issue, being that results appeared almost instantaneously. However, this does not mean that waiting time may not be an issue. Rigorous testing in a representative environment with multiple concurrent users is required before one can say anything definitive regarding waiting times.

Part of the presentation layer design is that the presented results should be sorted when new results arrive. An issue that has not been explored is the impact this live sorting has on the user. For instance, it may be frustrating for the user, when a user wants to click a link, and then the results are rearranged, and possibly the result being that the wrong link is clicked. How big an issue this is, is likely dependent on how often results arrive, and how often the displayed results are updated. It may be valuable to investigate this through user testing.

One of the guidelines that Haber and Bailey [12] proposed, as explained in the background chapter, is that the tools that system administrators use should have the possibility to be integrated into system-wide monitoring and management tools. With the architecture, it is possible to design a client that could be integrated into such systems. Haber and Bailey also stress the importance of allowing sharing of system views. A client can be developed that support queries being embedded entirely in urls, which means that urls can be passed between sysadmins to get the same view. However, the ranking of the results will of course depend on the user. Also, since the search results are mainly based on urls, these urls can easily be shared among users. Haber and Bailey mentions that there exists no single system to monitor everything, due to the heterogeneous nature of many systems. Even though the architecture does not provide a single place to monitor all activities, it does provide a single entry point for getting information on demand, which may help enhance situational awareness.

An aspect that has not been mentioned is the possibility that the urls in the search result may need to contain personal information, such as user names, in order for the urls to give access to the information source. In such cases, a possible solution is to remove this information from the urls, and exchange it with place holders. The users can then be informed that additional information is required in the url, in order for it to work.

6.2.5 Error Handling

In retrospect, error handling may not have been given enough attention when designing the architecture. More specifically, the error handling related to communication between the query broker and the modules. How can the query broker know whether a module is not responding, or simply slow? A solution to this issue, can be to include an additional queue for each of the modules that the query broker can send messages to, asking the module to reply with a message indicating that it is operational. The modules can then check this queue regularly and respond to a different queue belonging to the query broker. The query broker will then have a way to check whether a module is down. If the module does not respond within some predetermined time frame, the module may be flagged as non-operational, the error can be logged, and the query broker can stop sending queries to that module's query queue.

6.2.6 Expanding Search Possibilities

The search options, or query parameters, in the architecture are very limited in contrast to for example free text search, available in search engines on the web. The reason for the search limitation is of course that it is specialized towards the domain of system administration. An advantage of having such limited search possibilities is that the user will likely not have any trouble formulating a search, which can be more difficult in the context of free text searching.

In retrospect, one might ask the question of whether the option of searching for servers and services is sufficient to cover the information need of system administrators. It may be that there are additional query parameters that could cover broader areas of information, or help in getting information about other entities useful to system administrators. On such entity is the project. Adding the option of searching for a project could allow the user to get information about for instance which servers are involved in a project, the documentation related to a project, Service Level Agreements related to a project, information about which customers are involved, and so on. Another aspect is that there may be parties other than the system administrators, that may have interest in the information provided by the information sources. For instance service level managers or even some customers. There may be additional query parameters that could help their information seeking needs as well. Looking into the value of additional query parameters is a topic for future research.

6.2.7 Scalability

The topic of scalability has not been mentioned earlier in this thesis. This is mainly due to the fact that scalability is largely dependent of how a system based on the architecture is implemented. There are two main dimensions which could impact the performance of the system.

The first is the number of users that use the system. This can impact the the modules, the MOM and the query broker. In order to alleviate stress on the modules, multiple modules can be run against the same service as long as they use the same queues. In the context of MOM, ApacheMQ which is the MOM used in the prototype, supports creating networks of message brokers in order to scale to many clients. If the query broker is under stress, one could implement a form of load balancing by having a component responsible for distributing queries from the clients to different query brokers, and having identical sets of modules communicating with their own query broker. However, one might ask the question if scalability with regards to users is even an issue. There is a limit to how many system administrators work even in a large organization.

The second dimension is the amount of information sources to collect information from, and thereby the amount of modules that are part of the system. A large amount of modules will increase the amount of information that the query broker has to process. However, the amount of information to process will always be dependent of the number of users. This means that a load balancing solution could be applied to the situation where a system includes many modules as well.

6.3 In The Server Room

System administrators do not always sit at their desks. A part of being a system administrator involves dealing with servers and other hardware. System administrators sometimes have to disconnect or reconnect different hardware in the server room, and it is essential to know what impact this will have. Therefore, it could be useful for system administrators to have a way to get information about the different servers while they are in the server room. The architecture designed in this thesis allows system administrators to search for information about specific servers. As mentioned earlier it is possible to develop a client that is capable of specifying a search by a simple url. This allows having predefined searches related to specific servers. These urls can then be embedded in QR-codes, and hand-held devices like smart-phones or tablets can be used to scan the QR-codes and perform the search. Then it is simply a matter of sticking a qr-code onto every server to allow easy access to information. In the case of virtualization, multiple machines may run on a single server, and a single qr-code will not be sufficient. In this case, one could possibly have a screen displaying qr-codes for the different machines running on the physical server.

6.4 Problem Statement Revisited

The problem statement for this thesis was defined the following way:

The goal of this thesis is to facilitate efficient information retrieval for system administrators by designing a modular architecture which

1. *aggregates and correlates information from a variety of sources*

2. *can classify, sort and prioritize information according to the user and systems*

3. *is capable of presenting information in an efficient manner*

In retrospect, one might consider this problem statement to be a bit too ambitious, given the time frame of this master thesis. Not in the sense that it is not possible to design such an architecture in the given time, this has clearly been done. Rather, the available time limited the amount of testing that could be done. Therefore it may be difficult to make definitive conclusions as to in what *degree* the architecture, and systems developed according to the architecture, help system administrators with their information demands. An alternative problem statement could instead have specified the functionality required by a system developed according to such an architecture. However, as the prototype demonstrated, systems developed according to the architecture can be capable of aggregating and correlating information from multiple sources; classify, sort and prioritize the information according to user preferences; and present this information to the user in a manner which reflects the relevance of the information.

One may say that the architecture designed in this thesis, is a step in the right direction for solving the *interoperability problem* in the context of system administration. Also, it is clear that this architecture represents a new way for system administrators to gather knowledge about the systems they maintain.

The architecture designed, and the prototype developed in this thesis is functional, relevant and easily expandable. The system can in principle be used by anyone, and new modules can be developed if needed. System administrators can use it, develop it further, and share knowledge about the system. New clients can also be developed to cater to different needs. The system is usable today, and modifications have been suggested to possibly make it better in the future.

6.5 Source Code

In order to allow for repeatability of the tests performed in this thesis, and help in future research, all source code and related files, have been included in the appendix. The source code has been carefully documented to allow for easy understanding.

Chapter 7

Conclusion

This master thesis is aimed at helping system administrators with their information needs. The goal was to design a modular architecture to help in this endeavour, that is capable of aggregating and correlating information from different sources; classify, sort and prioritize the information according to user preferences; and present this information to the user in an efficient manner.

To design this architecture several key elements where needed. These were a classification of the domain of system administration, a prioritization scheme, a strategy for data aggregation, a module design and a design for the presentation layer. As a proof of concept a prototype based on the architecture was to be developed. Verification of the prototype's functionality was to be accomplished through the use of functionality testing.

An architecture named SQSI (Search Queries for Sysadmin Information) was designed according to the aforementioned principles, and a prototype based on SQSI was developed. The prototype included modules for three different services; Munin, Request Tracker and DokuWiki. The prototype allows users to search for information about servers and services, and receive information from the three information sources Munin, Request Tracker and DokuWiki. The results from testing the prototype showed that it is able to gather information from multiple sources, prioritize the information according to preference profiles, and present the results to the user in a manner reflecting the priorities of the user.

Based on the experiences made during the design of the architecture and the development of the prototype, several improvements have been discussed and added to the design.

SQSI is a functional and easily expandable architecture, that represents a new way for system administrators to gather knowledge about the systems the maintain.

7.1 Future Work

Possible future work and modifications to the architecture have been suggested. This involves user testing of the system, performance testing of the system, investigating the value of local importance to the prioritization scheme, looking into how preference profiles can be generated in a more user friendly and accurate manner, expanding the search possibilities by additional search parameters, and investigating the inclusion of relevance feedback in the architecture.

Chapter 8

Appendix

8.1 Preference Profiles

8.1.1 Alice

Alice.profile

```
Alert 0.5
Backup 0.2
Configuration 0.2
Customer 0.2
Database 0.0
Firewall 0.0
Hard Drive 0.7
Hardware 1.0
Documentation 1.0
Guarantee 0.8
Hostname 0.6
Installed Software 0.1
Intrusion Detection 0.0
Linux 0.5
Mail 0.0
Mac OS X 0.0
Middleware 0.0
Monitoring 0.8
Network 0.0
Network Interface 0.8
Order 0.0
OS 0.6
Process 7.0
SLA 0.5
```

```
Support 0.0
System 0.8
Ticket 0.0
Trend 1.0
Uptime 0.2
Vendor 0.5
Web 0.0
Windows 0.0
```

;

8.1.2 Bob

Bob.profile

```
Alert 1.0
Backup 0.0
Configuration 0.8
Customer 0.2
Database 0.0
Firewall 0.0
Hard Drive 0.1
Hardware 0.0
Documentation 1.0
Guarantee 0.0
Hostname 0.0
Installed Software 0.8
Intrusion Detection 0.0
Linux 0.5
Mail 0.0
Mac OS X 0.0
Middleware 0.0
Monitoring 0.3
Network 0.0
Network Interface 0.3
Order 0.0
OS 0.4
Process 0.0
SLA 0.5
Support 1.0
System 0.2
Ticket 1.0
Trend 0.0
Uptime 0.0
Vendor0.5
Web 0.4
Windows 0.0
```

;

8.2 Query Broker

8.2.1 querybroker.pl

querybroker.pl

```perl
#!/usr/bin/perl -w

# Needed packages
use Getopt::Std;
use Net::Stomp;
use XML::Simple;
use Data::Dumper;
use strict "vars";
use UNIVERSAL 'isa';
use POSIX;
use QueryResult;
use Result;

### Global variables

# turns verbose mode on or off
my $VERBOSE = 0;
# turns debug mode on or off
my $DEBUG = 0;
# the directory containing the user profiles
my $PROFILEDIR = "profiles";
# the stomp query connection
my $STOMP;
# the stomp incoming module connection
my $STOMPIN;
# sets the maximum number of profiles to keep in memory
my $MAXPROFILES = 500;
# a hash that maps usernames to QueryResults
my %QUERYRESULTS;
# a hash that maps usernames to profiles
my %PROFILES;
# the name of the file containing the modules to use
my $MODULES_FILE = "modules.dat";
# an array containing the modules to subscribe to
my @MODULES;

### End Global variable

# Handle flags and arguments
# Example: c == "-c", c: == "-c argument"
```

```
my $opt_string = 'vdh';
getopts("$opt_string",\my %opt) or usage() and exit 1;

# Print help message if -h is invoked
if($opt{'h'}){
  usage();
  exit 0;
}

# Handle other user input
$VERBOSE = 1 if $opt{'v'};
$DEBUG = 1 if $opt{'d'};

###### Main script content
#

verbose("Verbose is enabled\n");
debug("Debug is enabled\n");

set_modules();
connect_to_server();
subscribe_query("queryQueue");
foreach my $module (@MODULES){
  subscribe_module("resultQueue$module");
}
while(1){
  if($STOMP->can_read({timeout => 0})){
    my %query = get_query();
    send_query(\%query);
  }
  if($STOMPIN->can_read({timeout => 1})){
    handle_incoming_result();
  }
}

disconnect_from_server();

#
######

##
## Sets the modules to be used as specified in $MODULES_FILE
##
sub set_modules{
  open(MOD,$MODULES_FILE) or die "unable to open file
      $MODULES_FILE\n";
  verbose("Loading modules:\n");
  while(my $line = <MOD>){
    push(@MODULES, $line);
```

```perl
        verbose("$line");
    }
}

##
## Handles an incoming module result. Decoding it, checking if it's outdated,
## adds priorities according to a user profile, encodes the results, and makes
## sure the results are sent to the correct queue.
## This is accomplished through the use of variouse sub routines.
##
sub handle_incoming_result{

    my $xml_data = get_module_result();
    my @result_info = get_result_info_from_xml($xml_data);
    my $user = $result_info[0];
    my $user_device = $result_info[1];
    my $query_number = $result_info[2];
    my $expiry = $result_info[3];
    my $qr = ${$QUERYRESULTS{$user_device}};

    # need to check if the result is outdated
    if($query_number != $qr->getQueryNumber()){
        # the result is outdated, so it is ignored
        debug("Result outdated, user_device: $user_device, query number: $query_number\n");
        return;
    }
    $qr->incResReceived();

    my @results = convert_module_result($xml_data);
    my @profile = get_profile($user);
    addPriorities(\@results,\@profile);
    my @output_results;
    foreach my $r (@results){
        push(@output_results, resultToJSON($r));
    }

    # need to check whether the search is finished
    my $mod_rec = $qr->getResReceived();
    if($mod_rec == scalar(@MODULES)){
        $qr->resetResReceived();
        debug("SEARCH FINISHED!\n");
        my $res = "{\"search-finished\": \"true\"}";
        push(@output_results, $res);
    }
    send_results(\@output_results, $user_device, $expiry);
```

```perl
}

##
## Sends an array of results to a specfied queue
## @param: ARRAY of encoded results
## @param: string containg user and device (concatinated)
##
sub send_results{
  my @results = @{shift()};
  my $user_device = shift();
  my $expiry = shift();
  my $queue = "/queue/resultQueue$user_device";

  foreach my $result (@results){
    my $current_time = time * 1000;
    my $deadline = $current_time + $expiry;
    $STOMP->send({
       destination => $queue,
       body     => $result,
       expires    => $deadline
    });
  }
}

##
## Used to extract data from an xml tagged module-result
## @param: a string containing the xml data
## @return: ARRAY containing the following data as strings:
##        user,
##        user_device (user and device concatenated),
##        query number,
##        expiry, (the number of milliseconds the results should
    remain in
##        the result queue before being deleted)
##
sub get_result_info_from_xml{
  my $xml_data = shift();
  my $xml = new XML::Simple;
  my $data = $xml->XMLin($xml_data);
  my $user = $data->{'user'};
  my $user_device = $user . $data->{'device'};
  my $query_number = $data->{'query-number'};
  my $expiry = $data->{'expiry'};
  my @res_array = ($user, $user_device, $query_number, $expiry);
  return @res_array;
}

##
## Used to get a user profile based on user name.
## Will call a method to create the profile and store it in %
    PROFILES if it
## does not already exist.
## @param: string user name
## @return: ARRAY containing the profile (priorities)
##
sub get_profile{
  my $username = shift();
  my @profile;
  if(exists($PROFILES{$username})){
    debug("PROFILE EXISTS!\n");
    @profile = @{$PROFILES{$username}};
  }else{
    debug("PROFILE DOES NOT EXIST!\n");
    @profile = create_profile_array($username);
    $PROFILES{$username} = \@profile;
  }
  return @profile;
}


##
## Used to send a query to all modules
## @param: HASH containing the query
##
sub send_query{
  my %query = %{shift()};
  my $user = $query{'user'};
  my $device = $query{'device'};
  my $user_device = $user . $device;
  # check max int for query number?
  # increase the number if there already exists a QueryResult
      object
  # for the user
  my $qr;
  if(!exists($QUERYRESULTS{$user_device})){
    $qr = new QueryResult($user_device);
    $QUERYRESULTS{$user_device} = \$qr;
  }else{
    $qr = ${$QUERYRESULTS{$user_device}};
    if($qr->getQueryNumber() == INT_MAX){
      $qr->resetQueryNumber();
    }else{
      $qr->incQueryNumber();
    }
    $qr->resetResReceived();
  }
  my $query_xml = "<query>\n\t".
          "<query-number>" . $qr->getQueryNumber() .
```

```
            "</query-number>\n\t" .
            "<user>$query{'user'}</user>\n\t" .
            "<device>$query{'device'}</device>\n\t" .
            "<service>$query{'service'}</service>\n\t" .
            "<server>$query{'server'}</server>\n" .
            "</query>\n";

  foreach my $module (@MODULES){
    debug("Sending to $module\n");
    my $queue = "/queue/queryQueue$module";

    $STOMP->send({
      destination => $queue,
      body        => $query_xml
    });
  }
}

##
## Used to connect to the ActiveMQ server
## Instantiates the two stomp connections STOMP and STOMPIN
##
sub connect_to_server{
  $STOMP = Net::Stomp->new({hostname => 'localhost', port => '
      61613'});
  $STOMP->connect({login => 'admin', passcode => 'activemq'});
  $STOMPIN = Net::Stomp->new({hostname => 'localhost', port => '
      61613'});
  $STOMPIN->connect({login => 'admin', passcode => 'activemq'});
}

##
## Disconnectes both stomp connections
##
sub disconnect_from_server{
  $STOMP->disconnect;
  $STOMPIN->disconnect;
}

##
## Used to subscribe to the query queue
## @param: queue name
##
sub subscribe_query{
  my $queueName = shift();
  $STOMP->subscribe(
    { destination           => "/queue/$queueName",
      'ack'                 => 'client',
      'activemq.prefetchSize' => 1
    }
  );
}

##
## Used to subscribe to the a modules result queue
## @param: queue name
##
sub subscribe_module{
  my $queueName = shift();
  $STOMPIN->subscribe(
    { destination        => "/queue/$queueName",
      'ack'              => 'client',
      'activemq.prefetchSize' => 1
    }
  );
}

##
## Used to get a query from the query queue
## @return: HASH containing the query
##
sub get_query{
  my $frame = $STOMP->receive_frame;
  my $xml = new XML::Simple;
  my $data = $xml->XMLin($frame->body);
  my %query = %{$data};
  $STOMP->ack({frame => $frame});
  return %query;
}

##
## Converts a Result object into a JSON string
## @param: Result object
## @return: JSON string
##
sub resultToJSON{
  my $res = shift();
  my $title = $res->getTitle();
  my $source = $res->getSource();
  my $priority = $res->getPriority();
  my $url = $res->getUrl();
  my $json =  "{\"priority\": \"$priority\", " .
         "\"url\": \"$url\", \"title\": \"$title\", " .
         "\"source\": \"$source\"}";
  return $json;
}

##
```

```
## Creates an array of priorities based on a user profile file
## @param: user name
## @return: ARRAY containing the priorities
##
sub create_profile_array{
  my $name = shift;
  my @profile;
  open(PROFILE,"$PROFILEDIR/$name.profile") or die "Missing
      profile: $name\n";
  while(my $line = <PROFILE>){
    $line =~ /([\d|\.]+)/g;
    my $priority = $1;
    push(@profile, $priority);
    debug($priority . "\n");
  }
  close(PROFILE);
  return @profile;
}

##
## Calculates the inner product of two arrays of the same length
## @param: ARRAYREF
## @param: ARRAYREF
## @return: Number (the inner product)
##
sub inner_product{
  my @v1 = @{shift()};
  my @v2 = @{shift()};
  my $sp = 0;
  my $length = scalar(@v1);
  for(my $i = 0; $i<$length;$i++){
    $sp += $v1[$i] * $v2[$i];
  }
  return $sp;
}

##
## Gets a module result from a result queue
## @return: string containing xml data
##
sub get_module_result{
  my $frame = $STOMPIN->receive_frame;
  my $xml_data = $frame->body;
  $STOMPIN->ack({frame => $frame});
  return $xml_data;
}

##
```

```
## Converts a xml tagged module result to an array of Result
    objects
## @param: string of xml data
## @return: ARRAY containing the Result objects
##
sub convert_module_result{
  my $xml_data = shift();
  my $xml = new XML::Simple;
  my $data = $xml->XMLin($xml_data);
  my @res_array;

  my $user = $data->{'user'};
  my $device = $data->{'device'};
  my $user_device = $user . $device;
  my $queryResult = ${$QUERYRESULTS{$user_device}};
  my $source = $data->{'source'};

  if(isa($data->{'result'}, 'ARRAY')){
    my @results = @{$data->{'result'}};
    foreach my $res (@results){
      my @classification = create_classification_array(
        $res->{'classification'}
      );
      my $result = new Result(
        \@classification,
        $res->{'url'},
        $res->{'title'},
        $source);
      push(@res_array, $result);
    }
  }elsif(isa($data->{'result'}, 'HASH')){
    my %res = %{$data->{'result'}};
    my @classification = create_classification_array(
      $res{'classification'}
    );
    my $result = new Result(
      \@classification,
      $res{'url'},
      $res{'title'},
      $source);
    push(@res_array, $result);
  } # else there are no results
  return @res_array;
}

##
## Converts a string of numbers (classification string) into an
    array
## containing these numbers
## @param: string
## @return: ARRAY
##
sub create_classification_array{
  my $c_string = shift;
  my @c_array = ();
  for(my $i=0;$i<length($c_string);$i++){
    my $c = substr($c_string,$i,1);
    push(@c_array, $c);
  }
  return @c_array;
}

##
## Adds priorities to an array of Result objects according to a
    profile
## @param: ARRAY of Result objects
## @param: ARRAY of priorities (the profile)
##
sub addPriorities{
  my @results = @{shift()};
  my @profile = @{shift()};
  foreach my $result (@results){
    my @classification = @{$result->getClassification()};
    my $priority = inner_product(\@profile, \@classification);
    $result->addPriority($priority);
  }
}

##
## Prints the correct use of this script
##
sub usage{
  print "Usage:\n";
  print "-h Usage\n";
  print "-v Verbose\n";
  print "-d Debug\n";
  print "./script [-d] [-v] [-h]\n";
}

##
## Used to print verbose messages, when in verbose mode
## @param: string to print
##
sub verbose{
  print $_[0] if $VERBOSE;
}

##
```

```
## Used to print debug messages, when in debug mode
## @param: string to print
##
sub debug{
  print "DEBUG: " . $_[0] if $DEBUG;
}
```

;

8.2.2 QueryResult.pm

QueryResult.pm

```
#!/usr/bin/perl

use strict 'vars';

package QueryResult;

##
## Constructor
##
sub new{
  my $class = shift;
  my $self = { _user_device => shift};
  $self->{'_queryNumber'} = 1;

  # The number of modules that have sent results
  $self->{'_moduleResReceived'} = 0;

  bless $self, $class;
  return $self;
}

##
## Returns the user_device name
## @return: user_device name
##
sub getUserDevice{
  my ($self) = @_;
  return $self->{'_user_device'};
}

##
## Increases the modulesResReceived by 1
##
sub incResReceived{
  my ($self) = @_;
  return $self->{'_moduleResReceived'}++;
```

```perl
}

##
## Returns the number of module results received
## @return: number of module results received
##
sub getResReceived{
  my ($self) = @_;
  return $self->{'_moduleResReceived'};
}

##
## Sets modulesResReceived to 0
##
sub resetResReceived{
  my ($self) = @_;
  $self->{'_moduleResReceived'} = 0;
}

##
## Sets the user_device name
## @param: user_device name
## @return: user_device name that was set
##
sub setUserDevice{
  my ($self, $user_device) = @_;
  $self->{'_user_device'} = $user_device if defined($user_device
      );
  return $self->{'_user_device'};
}

##
## Returns the queryNumber
## @return: query number
##
sub getQueryNumber{
  my ($self) = @_;
  return $self->{'_queryNumber'};
}

##
## Increases the queryNumber by 1
##
sub incQueryNumber{
  my ($self) = @_;
  return $self->{'_queryNumber'}++;
}

##
```

```
## Resets the query number to 1
##
sub resetQueryNumber{
  my ($self) = @_;
  return $self->{'_queryNumber'} = 1;
}

1;
```

;

8.2.3 Result.pm

Result.pm

```
#!/usr/bin/perl

use strict 'vars';

package Result;

sub new{
  my $class = shift;
  my $self = { _classification => shift, _url => shift, _title
      => shift, _source => shift};
  $self->{'_priority'} = 0;

  bless $self, $class;
  return $self;
}

##
## Returns the classification vector
## @return: ARRAY classification vector
##
sub getClassification{
  my ($self) = @_;
  return $self->{'_classification'};
}

##
## Used to add priority
## @param: number, priority to add
##
sub addPriority{
  my ($self, $priority) = @_;
  $self->{'_priority'} += $priority;
}

```

```
##
## Returns the priority
## @return: number, priority
##
sub getPriority{
  my ($self) = @_;
  return $self->{'_priority'};
}

##
## Returns the url
## @return: string, url
##
sub getUrl{
  my ($self) = @_;
  return $self->{'_url'};
}

##
## Returns the title
## @return: string, title
##
sub getTitle{
  my ($self) = @_;
  return $self->{'_title'};
}

##
## Returns the source name
## @return: string, source name
##
sub getSource{
  my ($self) = @_;
  return $self->{'_source'};
}

##
## Returns the expiry time
## @return: number of milliseconds
##
sub getExpiry{
  my ($self) = @_;
  return $self->{'_expiry'};
}

1;
```

;

8.2.4 modules.dat

modules.dat

```
DokuWiki-Module
Munin-Module
Request-Tracker-Module
```

;

8.3 SQSI-CLI

sqsi-cli

```
#!/usr/bin/perl

# Needed packages
use Getopt::Std;
use Net::Stomp;
use Data::Dumper;
use JSON;
use strict "vars";

## Global variables

# turns verbose mode on or off
my $VERBOSE = 0;
# turns debug mode on or of
my $DEBUG = 0;
# the stomp connection
my $STOMP;
# the hostname of the activemq server
my $HOSTNAME = 'localhost';
# the port used to connect to the activemq server
my $PORT = '61613';
# the username used when logging in on the activemq server
my $LOGIN = 'admin';
# the password used when logging in on the activemq server
my $PASSCODE = 'activemq';
# the destination to subscribe to
my $DESTINATION;
# the username
my $USER;
# the device name
my $DEVICE;
# the service name
```

```
my $SERVICE;
# the server name
my $SERVER;
# array holding results
my @RESULTS;
# the time to wait before updating the user interface
my $SLEEPTIME = 0.01;
# flag set when the a search is finished
my $SEARCH_FINISHED = 0;
# signifies whether rank should be displayed in the output or
    not
my $SHOW_RANK = 0;

## End Global variables

# Handle flags and arguments
# Example: c == "-c", c: == "-c argument"
my $opt_string = 'vdhu:s:c:i:r';
getopts("$opt_string",\my %opt) or usage() and exit 1;

# Print help message if -h is invoked
if($opt{'h'}){
  usage();
  exit 0;
}

# Handle other user input
$VERBOSE = 1 if $opt{'v'};
$DEBUG = 1 if $opt{'d'};
$SHOW_RANK = 1 if $opt{'r'};
$SERVER = $opt{'s'};
$SERVICE = $opt{'c'};
$USER = $opt{'u'};
$DEVICE = $opt{'i'};
$DESTINATION = 'resultQueue' . $USER . $DEVICE;

###### Main script content
#

verbose("Verbose is enabled\n");
debug("Debug is enabled\n");

$STOMP = Net::Stomp->new({ hostname => $HOSTNAME, port => $PORT})
    ;
$STOMP->connect({ login => $LOGIN, passcode => $PASSCODE});
$STOMP->subscribe({
  destination => "/queue/$DESTINATION",
  'ack' => 'client',
  'activemq.prefetchSize' => 1
});

send_query();
my $counter = 0;
while(1){
  if($STOMP->can_read({timeout => 0})){
    my %result = %{receive_frame()};
    if(not exists $result{'search-finished'}){
      $counter++;
      push(@RESULTS, \%result);
      @RESULTS = sort_results(\@RESULTS);
      system('clear');
      foreach my $res (@RESULTS){
        my %result = %{$res};
        print $result{'priority'} . "\t" if $SHOW_RANK;
        print $result{'title'} . "\t" . $result{'url'}.
            "\t" . $result{'source'} . "\n";
      }
      print "counter: $counter\n";
    }
  } elsif($SEARCH_FINISHED){
    last;
  }
  select(undef, undef, undef, $SLEEPTIME);
}

#
######

##
## Sorts an array of results contained in hashes in descending
    order by
## their priority.
## @param: array of result hashes
## @return: array of sorted result hashes
##
sub sort_results{
  my @unsorted_res = @{shift()};
  my @sorted_res = map {$_->[1]}
                   sort {$b->[0] <=> $a->[0]}
                   map {[$_->{'priority'},$_]}
                   @unsorted_res;
  return @sorted_res;
}

##
## Gets a message from the result queue and returns it in the
    form of a hash.
```

```
## If the message is a "search-finished" frame, $SEARCH_FINISHED
     is set to 1.
## @return: result hash
##
sub receive_frame{
  my $frame = $STOMP->receive_frame;
  my $decoded_frame = decode_json $frame->body;
  $STOMP->ack({frame => $frame});
  my %tmp = %{$decoded_frame};
  if($tmp{"search-finished"} eq "true"){
    $SEARCH_FINISHED = 1;
  }
  return $decoded_frame;
}

##
## Sends a query message to the query queue with command line
## specified parameters.
##
sub send_query{
  my $queue = "/queue/queryQueue";
  my $query = "<query>\n\t" .
         "<user>$USER</user>\n\t" .
         "<device>$DEVICE</device>\n\t" .
         "<server>$SERVER</server>\n\t" .
         "<service>$SERVICE</service>\n" .
         "</query>";

  $STOMP->send({
    destination => $queue,
    body => $query
  });
}

##
## Prints the correct use of this script
##
sub usage{
  print "Usage:\n";
  print "-h Usage\n";
  print "-v Verbose\n";
  print "-d Debug\n";
  print "-u   <username> username\n";
  print "-s   <server name> server name\n";
  print "-c   <service name> service name\n";
  print "-i   <device name> device name\n";
  print "-r   Toggles printing rank in output\n\n";
  print "./client.pl -u <username> -s <server name> " .
```

```
      " -c <service name> -i <device name> [-r] [-d] [-v] [-h]\n
         \n";
}

##
## Used to print verbose messages when in verbose mode
## @param: string to print
##
sub verbose{
  print $_[0] if $VERBOSE;
}

##
## Used to print debug messages when in debug mode
## @param: string to print
##
sub debug{
  print "DEBUG: " . $_[0] if $DEBUG;
}
```

;

8.4 Munin Module

8.4.1 munin-module.pl

munin–module.pl

```
#!/usr/bin/perl

# Needed packages
use Getopt::Std;
use strict "vars";
use Net::Telnet;
use Net::Stomp;
use XML::Simple;
use Data::Dumper;

### Global variables

# turns on or off verbose mode
my $VERBOSE = 0;
# turns on or off debug mode
my $DEBUG = 0;
# the port used to connect to telnet on the munin nodes
my $TELNET_PORT = 4949;
# the path to the munin configuration file
```

```
my $MUNIN_CONF = "/etc/munin/munin.conf";
# an array containing the names of all munin nodes
my @NODENAMES;
# a hash maping service names to their relevant plugins
my %SERVICES_TO_PLUGINS;
# a hash maping node names and to their plugins
my %NODE_PLUGINS;
# a hash maping plugin names to their classification vectors (
    strings)
my %PLUGIN_CLASSIFICATIONS;
# the name of the file containing the classification strings
my $CLASSIFICATIONS = "classifications.dat";
# the name of the file containing the service names and their
    plugins
my $SERVICES = 'services.dat';
# the stomp connection object
my $STOMP;
# the name of the ActiveMQ server
my $MQSERVER = "sysadmin14.iu.hio.no";
# the name of the munin server
my $MUNINSERVER = "sysadmin15.iu.hio.no";
# the name of the munin domain. Used in url generation
my $MUNINDOMAIN = "iu.hio.no";
# the port used to connect to the ActiveMQ server
my $MQPORT = 61613;
# the name of this module
my $MODULENAME = "Munin-Module";
# the name of the source being provided by this module
my $SOURCENAME = "Munin";
# the name of the query queue
my $QUERYQUEUE = "queryQueue$MODULENAME";
# the name of the result queue
my $RESULTQUEUE = "resultQueue$MODULENAME";
# the number of milliseconds a result from this module should
    remain in the
# result queue of the client before being removed
my $EXPIRY = 20000;

### End Global variables

# Handle flags and arguments
# Example: c == "-c", c: == "-c argument"
my $opt_string = 'vdh';
getopts("$opt_string",\my %opt) or usage() and exit 1;

# Print help message if -h is invoked
if($opt{'h'}){
  usage();
  exit 0;
```

```perl
}

# Handle other user input
$VERBOSE = 1 if $opt{'v'};
$DEBUG = 1 if $opt{'d'};

###### Main script content
#

verbose("Verbose is enabled\n");
debug("Debug is enabled\n");

verbose("Loading...\n");
set_plugin_classifications();
set_nodenames();
register_node_data();
set_services();
connect_to_server();
verbose("$MODULENAME running\n");
subscribe($QUERYQUEUE);

while(1){
  my %query = get_query();
  my $result = get_module_result(\%query);
   send_result($result);
}

#
######

##
## Used to read the $SERVICES file and store the contents in
## %SERVICER_TO_PLUGINS indexed by service
##
sub set_services{
  open(SER,$SERVICES) or die "unable to open $SERVICES\n";
  while(my $line = <SER>){
    chomp($line);
    my @input = split(/,/,$line);
    my $service = $input[0];
    my $length = scalar(@input) - 1;
    my @plugins =  @input[1..$length];
    @{$SERVICES_TO_PLUGINS{$service}} = @plugins;
  }
  close(SER);
}

##
```

```
## Gets the results from a query based on the server and service
    name
## @param: service name
## @param: server name
## @return: a string containing all xml tagged results from the
    query
##
sub get_results{

  my $server = lc shift;
  my $service = lc shift;
  my $server_exists = 0;
  my $service_exists = 0;
  $server_exists = 1 if exists $NODE_PLUGINS{$server};
  $service_exists = 1 if exists $SERVICES_TO_PLUGINS{$service};
  my $results = "";

  if($server_exists and $service_exists){
    my %server_plugins = %{$NODE_PLUGINS{$server}};
    my @service_plugins = @{$SERVICES_TO_PLUGINS{$service}};
    my @available_plugins;

    # Need to check which of the plugins for the service that
       are
    # available at the specified server

    foreach my $service_plugin (@service_plugins){
      if(exists $server_plugins{$service_plugin}){
        push(@available_plugins, $service_plugin);
      }
    }

    foreach my  $plugin (@available_plugins){
      my $title = "Trend: $plugin - $server";
      my $c = $PLUGIN_CLASSIFICATIONS{$plugin};
      my $url = generate_url($server,$plugin);
      my $xml_result = generate_xml_result($title,$c,$url);
      $results .= $xml_result;
    }

  }elsif($server_exists and !$service_exists and length($service
    ) == 0){
    my %server_plugins = %{$NODE_PLUGINS{$server}};
    foreach my $plugin (values %server_plugins){
      my $title = "Trend: $plugin - $server";
      my $c = $PLUGIN_CLASSIFICATIONS{$plugin};
      my $url = generate_url($server,$plugin);
      my $xml_result = generate_xml_result($title,$c,$url);
      $results .= $xml_result;
    }
  }elsif(!$server_exists and $service_exists and length($server)
      == 0){
    my @service_plugins = @{$SERVICES_TO_PLUGINS{$service}};
    my %servers;

    # Need to find all servers that run the relevant service
    foreach my $server (keys %NODE_PLUGINS){
      foreach my $service_plugin (@service_plugins){
        if(exists $NODE_PLUGINS{$server}{$service_plugin}){
          push(@{$servers{$server}},$service_plugin);
        }
      }
    }

    foreach my $server (keys %servers){
      my @plugins = @{$servers{$server}};
      foreach my $plugin (@plugins){
        my $title = "Trend: $plugin - $server";
        my $c = $PLUGIN_CLASSIFICATIONS{$plugin};
        my $url = generate_url($server,$plugin);
        my $xml_result = generate_xml_result($title,$c,$url);
        $results .= $xml_result;
      }
    }

  }# else there are no results for this query

  return $results;
}

##
## Used to generate a url to a munin plugin based on server and
    plugin
## @param: server name
## @param: service name
##
sub generate_url{
  my $server = shift;
  my $plugin = shift;
  my $url = "http://$MUNINSERVER/munin/$MUNINDOMAIN/".
      "$server/$plugin.html";
  return $url;
}

##
## Used to generate an xml tagged single result
## @param: the title
## @param: the classification vector (string)
## @param: the url linking to the resource
## @return: a single xml tagged result
##
sub generate_xml_result{
  my $title = shift;
  my $c = shift;
  my $url = shift;
  my $xml_result =  "\t<result>\n".
                "\t\t<title>$title</title>\n".
                "\t\t<classification>$c</classification>\n".
                "\t\t<url>$url</url>\n".
                "\t</result>\n";
  return $xml_result;
}

##
## Used to build the module result based on a query
## @param: Hash containing the query data
## @return: An xml tagged result
##
sub get_module_result{
  my %query = %{shift()};
  my $user =  $query{'user'};
  my $device = $query{'device'};
  my $query_number = $query{'query-number'};
  my $xml_module_result = "";
  my $xml_header =  "<module-result user=\"$user\" ".
                "device=\"$device\" ".
                "query-number=\"$query_number\" ".
                "source=\"$SOURCENAME\" ".
                "expiry=\"$EXPIRY\">\n";
  my $xml_footer = "</module-result>\n";
  $xml_module_result .= $xml_header;

  my $server = $query{'server'};
  if($server =~ /^HASH\(.*/){
    $server = "";
  }
  my $service = $query{'service'};
  if($service =~ /^HASH\(.*/){
    $service = "";
  }
  my $results = get_results($server, $service);
  $xml_module_result .= $results;

  $xml_module_result .= $xml_footer;
  return $xml_module_result;
}

```

```
##
## Used to send a module result to the result queue
## @param: an xml tagged module result
##
sub send_result{
  my $result = shift();
  $STOMP->send({
     destination => "/queue/$RESULTQUEUE",
     body => $result
  });
}

##
## Used to get a query message from the query queue
## @return: Hash containing the query data
##
sub get_query{
  my $frame = $STOMP->receive_frame;
  my $xml = new XML::Simple;
  my $data = $xml->XMLin($frame->body);
  my %query = %{$data};
  debug($query{'user'} . "\n");
  debug($query{'query-number'} . "\n");
  debug($query{'server'} . "\n");
  debug($query{'service'} . "\n");
  $STOMP->ack({ frame => $frame });
  return %query;
}

##
## Connects to the ActiveMQ server
##
sub connect_to_server{
  $STOMP = Net::Stomp->new({
     hostname   => $MQSERVER,
     port       => $MQPORT
  });

  $STOMP->connect({
     login      => 'admin',
     passcode   => 'activemq'
  });
}

##
## Used to subscribe to a certain queue
## @param: queue name
##
sub subscribe{
  my $queueName = shift();
  $STOMP->subscribe({
     destination           => "/queue/$queueName",
     'ack'                 => 'client',
     'activemq.prefetchSize' => 1
  });
}

##
## Used to read the $CLASSIFICATIONS file and store the
     classification strings
## of the plugins by name in %PLUGIN_CLASSIFICATIONS
##
sub set_plugin_classifications{
  open(IN, $CLASSIFICATIONS) or die "unable to open
      $CLASSIFICATIONS\n";
  while(my $line = <IN>){
     $line =~ /^(\w+)[\s|\t]*(\d+)/;
     my $plugin = $1;
     my $c_string = $2;
     $PLUGIN_CLASSIFICATIONS{$plugin} = $c_string;
  }
  close(IN);
}

##
## Used to store the plugins active for each munin node in %
     NODE_PLUGINS
## by node name. The data from the nodes is collected by calling
## get_plugin_info()
##
sub register_node_data{
  my $output;
  my @plugins;
  foreach my $node (@NODENAMES){
     $output = get_plugin_info($node);
     @plugins = split(/ /,$output);
     foreach my $p (@plugins){
        chomp($p);
        $NODE_PLUGINS{$node}{$p} = $p;
     }
  }
}

##
## Reads the $MUNIN_CONF file and stores the names of all the
     munin nodes
## in @NODENAMES
##
```

```
sub set_nodenames{
  open(MCONF,$MUNIN_CONF) or die "unable to open $MUNIN_CONF\n";
  while(my $line = <MCONF>){
    if($line =~ /^\[(.*)\]$/){
      my $nodename = $1;
      push(@NODENAMES, $nodename);
    }
  }
  close(MCONF);
}

##
## Collects the names of all active plugins for a munin node
## @param: node name
## @return: string separated by space containing plugin names
##
sub get_plugin_info{
  my $node = shift();
  my $output;
  my $telnet = new Net::Telnet(
    Timeout => 10,
    Port => $TELNET_PORT,
    Prompt => '/\$ $/',
    Errmode => 'die'
  );
  $telnet->open($node);
  $telnet->waitfor('/# munin.*$/');
  $telnet->print('list');
  $telnet->getline;
  $output = $telnet->getline;
  return $output;
}

##
## Prints the correct use of this script
##
sub usage{
  print "Usage:\n";
  print "-h Usage\n";
  print "-v Verbose\n";
  print "-d Debug\n";
  print "./script [-d] [-v] [-h]\n";
}

##
## Used to print verbose messages, when in verbose mode
## @param: string to print
##
sub verbose{
  print $_[0] if $VERBOSE;
}

##
## Used to print debug messages, when in debug mode
## @param: string to print
##
sub debug{
  print "DEBUG: " . $_[0] if $DEBUG;
}
```

;

8.4.2 classifications.dat

classifications.dat

```
apache_accesses        00000000000000000000000000010000
apache_processes       00000000000000000000001000010000
apache_volume        00000000000000000000000000010000
cpu             00000000000000000000000001010000
df              00000100000000000000000000010000
df_inode          00000100000000000000000000010000
diskstats_iops        00000100000000000000000000010000
diskstats_latency      00000100000000000000000000010000
diskstats_throughput    00000100000000000000000000010000
diskstats_utilization   00000100000000000000000000010000
entropy           00000000000000000000000001010000
exim_mailqueue        00000000000000100000000000010000
exim_mailstats        00000000000000100000000000010000
forks           00000000000000000000001000010000
fw_packets         00001000000000000010000000010000
http_loadtime        00000000000000000010000000010000
if_err_eth0         00000000000000000001000000010000
if_eth0           00000000000000000001000000010000
interrupts         00000000000000000000000001010000
iostat           00000100000000000000000000010000
iostat_ios         00000100000000000000000000010000
irqstats          00000000000000000000000001010000
load            00000000000000000000000001010000
memory           00000000000000000000000001010000
munin_stats         00000000000000000000000000010000
mysql_bin_relay_log     00001000000000000000000000010000
mysql_commands        00001000000000000000000000010000
mysql_connections      00001000000000000000000000010000
mysql_filetables       00001000000000000000000000010000
mysql_innodb_bpool      00001000000000000000000000010000
mysql_innodb_bpool_act    00001000000000000000000000010000
mysql_innodb_inser_buf    00001000000000000000000000010000
```

```
mysql_innodb_io          0000100000000000000000000010000
mysql_innodb_io_pend     0000100000000000000000000010000
mysql_innodb_log         0000100000000000000000000010000
mysql_innodb_rows        0000100000000000000000000010000
mysql_innodb_semaphores  0000100000000000000000000010000
mysql_innodb_tnx         0000100000000000000000000010000
mysql_myisam_indexes     0000100000000000000000000010000
mysql_network_traffic    0000100000000001000000000010000
mysql_qcache             0000100000000000000000000010000
mysql_qcache_mem         0000100000000000000000000010000
mysql_replication        0000100000000000000000000010000
mysql_select_types       0000100000000000000000000010000
mysql_slow               0000100000000000000000000010000
mysql_sorts              0000100000000000000000000010000
mysql_table_locks        0000100000000000000000000010000
mysql_tmp_tables         0000100000000000000000000010000
open_files               0000000000000000000000001010000
open_inodes              0000000000000000000000001010000
proc_pri                 0000000000000000000001001010000
processes                0000000000000000000001000010000
swap                     0000100000000000000000001010000
threads                  0000000000000000000001000010000
uptime                   0000000000000000000000001011000
users                    0000000000000000000000001010000
vmstat                   0000000000000000000001000010000
```

;

8.4.3 services.dat

services.dat

```
apache , apache_accesses , apache_processes , apache_volume
munin , munin_stats
exim , exim_mailstats , exim_mailqueue
mysql , mysql_bin_relay_log , mysql_commands , mysql_connections ,
    mysql_filetables , mysql_innodb_bpool , mysql_innodb_bpool_act ,
    mysql_innodb_inser_buf , mysql_innodb_io , mysql_innodb_io_pend ,
    mysql_innodb_log , mysql_innodb_rows , mysql_innodb_semaphores ,
    mysql_innodb_tnx , mysql_myisam_indexes , mysql_network_traffic ,
    mysql_qcache , mysql_qcache_mem , mysql_replication ,
    mysql_select_types , mysql_slow , mysql_sorts , mysql_table_locks ,
    mysql_tmp_tables
```

;

8.5 Request Tracker Module

8.5.1 rt-module.pl

rt–module.pl

```
#!/usr/bin/perl

# Needed packages
use Getopt::Std;
use strict "vars";
use Net::Stomp;
use XML::Simple;
use Data::Dumper;
use UNIVERSAL 'isa';
use Scalar::Util 'reftype';

### Global variables

# turn on or off verbose mode
my $VERBOSE = 0;
# turn on or off debug mode
my $DEBUG = 0;
# the stomp connection object
my $STOMP;
# the name of the ActiveMQ server
my $MQSERVER = "sysadmin14.iu.hio.no";
# the name of the Request Tracker server
my $RTSERVER = "sysadmin16.iu.hio.no";
# the port used to connecto the ActiveMQ server
my $MQPORT = 61613;
# the name of this module
my $MODULENAME = "Request-Tracker-Module";
# the name of the source being provided by this module
my $SOURCENAME = "Request Tracker";
# the name of the query queue
my $QUERYQUEUE = "queryQueue$MODULENAME";
# the name of the result queue
my $RESULTQUEUE = "resultQueue$MODULENAME";
# the file containing the classfications strings for the RT
      queues
my $CLASSIFICATIONS = "queue_classifications.dat";
# Hash mapping RT queue names to classifications strings
my %QUEUE_CLASSIFICATIONS;
# the number of milliseconds a result should remain in the
# result queue of the client before being removed
```

```
my $EXPIRY = 40000;

### End Global variables

# Handle flags and arguments
# Example: c == "-c", c: == "-c argument"
my $opt_string = 'vdh';
getopts("$opt_string",\my %opt) or usage() and exit 1;

# Print help message if -h is invoked
if($opt{'h'}){
  usage();
  exit 0;
}

# Handle other user input
$VERBOSE = 1 if $opt{'v'};
$DEBUG = 1 if $opt{'d'};

###### Main script content
#

verbose("Verbose is enabled\n");
debug("Debug is enabled\n");
set_queue_classifications();
connect_to_server();
subscribe($QUERYQUEUE);
while(1){
  my %query = get_query();
  my $result = get_module_result(\%query);
  send_result($result);
}

#
######

##
## Used to get a query message from the query queue
## @return: Hash containing the query data
##
sub get_query{
  my $frame = $STOMP->receive_frame;
  my $xml = new XML::Simple;
  my $data = $xml->XMLin($frame->body);
  my %query = %{$data};
  debug($query{'user'} . "\n");
  debug($query{'query-number'} . "\n");
  debug($query{'server'} . "\n");
  debug($query{'service'} . "\n");
```

```perl
  $STOMP->ack({ frame => $frame });
  return %query;
}

##
## Used to send a module result to the result queue
## @param: an xml tagged module result
##
sub send_result{
  my $result = shift();
  $STOMP->send({
     destination => "/queue/$RESULTQUEUE",
     body => $result
  });
}

##
## Connects to the ActiveMQ server
##
sub connect_to_server{
  $STOMP = Net::Stomp->new({
     hostname   => $MQSERVER,
     port       => $MQPORT
  });

  $STOMP->connect({
     login      => 'admin',
     passcode   => 'activemq'
  });
}

##
## Used to subscribe to a certain queue
## @param: queue name
##
sub subscribe{
  my $queueName = shift();
  $STOMP->subscribe({
     destination             => "/queue/$queueName",
     'ack'                   => 'client',
     'activemq.prefetchSize' => 1
  });
}

##
## Used to read the $CLASSIFICATIONS file and store the
     classifications strings
## in %QUEUE_CLASSIFICATIONS by RT queue names
##
```

```
sub set_queue_classifications{
  open(CLASS,$CLASSIFICATIONS) or die "unable to open
     $CLASSIFICATIONS\n";
  while(my $line = <CLASS>){
    $line =~ /^(\w+)[\s|\t]*(\d+)/;
    my $queue = $1;
    my $c_string = $2;
    $QUEUE_CLASSIFICATIONS{$queue} = $c_string;
  }
  close(CLASS);
}

##
## Used to build a module result based on a query
## @param: Hash containing the query data
## @return: An xml tagged module result
##
sub get_module_result{
  my %query = %{shift()};
  my $user =  $query{'user'};
  my $device = $query{'device'};
  my $query_number = $query{'query-number'};
  my $xml_module_result = "";
  my $xml_header =  "<module-result user=\"$user\" ".
            "device=\"$device\" ".
            "query-number=\"$query_number\" ".
            "source=\"$SOURCENAME\" ".
            "expiry=\"$EXPIRY\">\n";
  my $xml_footer = "</module-result>\n";
  $xml_module_result .= $xml_header;

  my $server = $query{'server'};
  if($server =~ /^HASH\(.*/){
    $server = "";
  }
  my $service = $query{'service'};
  if($service =~ /^HASH\(.*/){
    $service = "";
  }

  my $results = get_results($server, $service);
  $xml_module_result .= $results;

  $xml_module_result .= $xml_footer;
  return $xml_module_result;
}

##
```

```
## Gets the results from a query based on server and service
    names
## @param: server name
## @param: service name
## @return: a string containing all xml tagged results from the
    query
##
sub get_results{
  my $server = lc shift;
  my $service = lc shift;
  my $server_not_empty = 0;
  my $service_not_empty = 0;
  $server_not_empty = 1 if $server ne "";
  $service_not_empty = 1 if $service ne "";
  my $results = "";

  # return empty result if the search parameters are missing
  return $results if (!$server_not_empty and !$service_not_empty
      );

  open(RT,"rt ls -f queue,subject|");
  my $skip_line = 1;
  while(my $line = <RT>){
    if($skip_line){
      $line = <RT>;
      $skip_line = 0;
    }
    $line =~ /^(\d+)[\s|\t]+(\S+)[\s|\t]+(.*)/;
    my $ticket_number = $1;
    my $queue = $2;
    my $subject = $3;
    if($server_not_empty and $service_not_empty){
      if( $subject =~ /.*\b$service\b.*/i and
        $subject =~ /.*\b$server\b.*/i){
        my $xml_result = generate_xml_result(
          "$queue: $subject",
          $QUEUE_CLASSIFICATIONS{$queue},
          $ticket_number);
        $results .= $xml_result;
      }
    }elsif(!$server_not_empty and $service_not_empty){
      if($subject =~ /.*\b$service\b.*/i){
        my $xml_result = generate_xml_result(
          "$queue: $subject",
          $QUEUE_CLASSIFICATIONS{$queue},
          $ticket_number);
        $results .= $xml_result;
      }
    }elsif($server_not_empty and !$service_not_empty){
      if($subject =~ /.*\b$server\b.*/i){
        my $xml_result = generate_xml_result(
          "$queue: $subject",
          $QUEUE_CLASSIFICATIONS{$queue},
          $ticket_number);
        $results .= $xml_result;
      }
    }
  }
  close(RT);
  return $results;
}

##
## Used to generate an xml tagged single result
## @param: the title
## @param: the classification vector (a string)
## @param: the ticket number
## @return: a single xml tagged result
##
sub generate_xml_result{
  my $title = shift;
  my $c = shift;
  my $ticket_number = shift;
  my $url = "http://$RTSERVER/rt/Ticket/Display.html?id=
      $ticket_number";
  my $xml_result =  "\t<result>\n".
              "\t\t<title>$title</title>\n".
              "\t\t<classification>$c</classification>\n".
              "\t\t<url>$url</url>\n".
              "\t</result>\n";
  return $xml_result;
}

##
## Prints the correct use of this script
##
sub usage{
  print "Usage:\n";
  print "-h Usage\n";
  print "-v Verbose\n";
  print "-d Debug\n";
  print "./script [-d] [-v] [-h]\n";
}

##
## Used to print verbose messages, when in verbose mode
## @param: string to print
##
```

```perl
sub verbose{
  print $_[0] if $VERBOSE;
}

##
## Used to print debug messages, when in debug mode
## @param: string to print
##
sub debug{
  print "DEBUG: " . $_[0] if $DEBUG;
}
```

;

8.5.2 queue_classifications.dat

queue_classifications.dat

```
Alert 1000000000000000000000000100000
General 0000000000000000000000000100000
Order 0000000000000000000100000100000
Support 0000000000000000000000010100000
```

;

8.6 DokuWiki Module

8.6.1 dokuwiki-module.pl

dokuwiki–module.pl

```perl
#!/usr/bin/perl

# Needed packages
use Getopt::Std;
use strict "vars";
use Net::Stomp;
use XML::Simple;
use Data::Dumper;

### Global variables

# turn on or off verbose mode
my $VERBOSE = 0;
# turn on or off debug mode
my $DEBUG = 0;
# file containing the classification vectors
```

```
my $CLASSIFICATIONS = "classifications.dat";
# hash, mapping classifications
my %CLASSIFICATION_MAP;
# the path to directory containing all wiki pages
my $PAGEPATH = '/var/lib/dokuwiki/data/pages';
# the stomp connection object
my $STOMP;
# the name of the ActiveMQ server
my $MQSERVER = "sysadmin14.iu.hio.no";
# the name of the dokuwiki server
my $DOKUWIKISERVER = "sysadmin17.iu.hio.no";
# the port used to connect to the ActiveMQ server
my $MQPORT = 61613;
# the name of this module
my $MODULENAME = "DokuWiki-Module";
# the name of the source being provided by this module
my $SOURCENAME = "DokuWiki";
# the name of the query queue
my $QUERYQUEUE = "queryQueue$MODULENAME";
# the name of the result queue
my $RESULTQUEUE = "resultQueue$MODULENAME";
# the number of milliseconds a result from this module should
    remain in the
# result queue of the client before being removed
my $EXPIRY = 40000;

### End global variables

# Handle flags and arguments
# Example: c == "-c", c: == "-c argument"
my $opt_string = 'vdh';
getopts("$opt_string",\my %opt) or usage() and exit 1;

# Print help message if -h is invoked
if($opt{'h'}){
  usage();
  exit 0;
}

# Handle other user inpul
$VERBOSE = 1 if $opt{'v'};
$DEBUG = 1 if $opt{'d'};

###### Main script content
#

verbose("Verbose is enabled\n");
debug("Debug is enabled\n");

```

```
verbose("Loading...\n");
set_classifications();
verbose("$MODULENAME running\n");
connect_to_server();
subscribe($QUERYQUEUE);
while(1){
  my %query = get_query();
  my $result = get_module_result(\%query);
   send_result($result);
}

#
######

##
## Gets the results from a query based on the server and service
     names
## @param: server name
## @param: service name
## @return: A string containing all xml tagged results from the
     query
##
sub get_results{

  my $server = lc shift;
  my $service =  lc shift;
  my $server_in_query = 0;
  my $service_in_query = 0;
   $server_in_query = 1 if length($server) > 0;
   $service_in_query = 1 if length($service) > 0;
  my $results = "";

   if($server_in_query and $service_in_query){

      if(-e "$PAGEPATH/$service-$server.txt"){
        my $title = "Documentation: $service-$server";
        my $c = get_classification_vector($service);
        my $url = generate_url("$service-$server");
        my $xml_result = generate_xml_result($title ,$c, $url);
         $results .= $xml_result;
      }

   }elsif($server_in_query and !$service_in_query and length(
     $service) == 0){

      opendir(DIR, "$PAGEPATH");
      my @files = readdir(DIR);
      foreach my $file (@files){
         if($file =~ /.*$server.*/){
                my $id = $file;
                $id =~ s/.txt//;
                my $title = "Documentation: $id";
                my $c = get_classification_vector($file);
                my $url = generate_url($id);
                my $xml_result = generate_xml_result($title ,$c,$url);
                $results .= $xml_result;
            }
        }

    }elsif(!$server_in_query and $service_in_query and length(
        $server) == 0){

        if(-e "$PAGEPATH/$service.txt"){
            my $title = "Documentation: $service";
            my $c = get_classification_vector($service);
            my $url = generate_url($service);
            my $xml_result = generate_xml_result($title ,$c,$url);
            $results .= $xml_result;
        }

    }# else there are no results for this query

    return $results;
}

##
## Used to get the classification vector of a term
## @param: term name
## @return: string, classification vector
##
sub get_classification_vector{
    my $page_name = shift;
    if(exists $CLASSIFICATION_MAP{$page_name}){
        return $CLASSIFICATION_MAP{$page_name};
    }
    $page_name =~ /(.*)-.*/;
    my $class = $1;
    return $CLASSIFICATION_MAP{$class};
}

##
## Used to generate the url to a wiki page based on an id
## @param: id
## @return: url
##
sub generate_url{
    my $id = shift;
    my $url = "http://$DOKUWIKISERVER/dokuwiki/doku.php?id=$id";
```

```perl
  return $url;
}

##
## Used to generate an xml tagged single result
## @param: the title
## @param: the classifcation vector (a string)
## @param: the url linking to the resource
## @return: a single xml tagged result
##
sub generate_xml_result{
  my $title = shift;
  my $c = shift;
  my $url = shift;
  my $xml_result =  "\t<result>\n".
            "\t\t<title>$title</title>\n".
            "\t\t<classification>$c</classification>\n".
            "\t\t<url>$url</url>\n".
            "\t</result>\n";
  return $xml_result;
}

##
## Used to build the module result based on a query
## @param:  HASH containing the query data
## @return: An xml tagged module result
##
sub get_module_result{
  my %query = %{shift()};
  my $user =  $query{'user'};
  my $device = $query{'device'};
  my $query_number = $query{'query-number'};
  my $xml_module_result = "";
  my $xml_header =  "<module-result user=\"$user\" ".
            "device=\"$device\" ".
            "query-number=\"$query_number\" ".
            "source=\"$SOURCENAME\" ".
            "expiry=\"$EXPIRY\">\n";
  my $xml_footer = "</module-result>\n";
  $xml_module_result .= $xml_header;

  my $server = $query{'server'};
  if($server =~ /^HASH\(.*/){
    $server = "";
  }
  my $service = $query{'service'};
  if($service =~ /^HASH\(.*/){
    $service = "";
  }
  my $results = get_results($server, $service);
  $xml_module_result .= $results;

  $xml_module_result .= $xml_footer;
  return $xml_module_result;
}

##
## Used to send a module result to the result queue
## @param: an xml tagged module result
##
sub send_result{
  my $result = shift();
  $STOMP->send({
     destination => "/queue/$RESULTQUEUE",
     body => $result
  });
}


##
## Used to get a query message from the query queue
## @return: HASH containing the query data
##
sub get_query{
  my $frame = $STOMP->receive_frame;
  my $xml = new XML::Simple;
  my $data = $xml->XMLin($frame->body);
  my %query = %{$data};
  debug($query{'user'} . "\n");
  debug($query{'query-number'} . "\n");
  debug($query{'server'} . "\n");
  debug($query{'service'} . "\n");
  $STOMP->ack({ frame => $frame });
  return %query;
}

##
## Connects to the ActiveMQ server
##
sub connect_to_server{
  $STOMP = Net::Stomp->new({
     hostname   => $MQSERVER,
     port       => $MQPORT
  });

  $STOMP->connect({
     login      => 'admin',
     passcode   => 'activemq'
  });
}

##
## Used to subscribe to a certain queue
## @param: queue name
##
sub subscribe{
  my $queueName = shift();
  $STOMP->subscribe({
     destination        => "/queue/$queueName",
     'ack'              => 'client',
     'activemq.prefetchSize' => 1
  });
}

##
##
##
sub set_classifications{
  open(IN, $CLASSIFICATIONS) or die "unable to open
      $CLASSIFICATIONS\n";
  while(my $line = <IN>){
    $line =~ /^(\w+)[\s|\t]*(\d+)/;
    my $name = $1;
    my $c_string = $2;
    $CLASSIFICATION_MAP{$name} = $c_string;
  }
  close(IN);
}

##
## Prints the correct use of this script
##
sub usage{
  print "Usage:\n";
  print "-h Usage\n";
  print "-v Verbose\n";
  print "-d Debug\n";
  print "./script [-d] [-v] [-h]\n";
}

##
## Used to print verbose messages, when in verbose mode
## @param: string to print
##
sub verbose{
  print $_[0] if $VERBOSE;
}
```

```

##
## Used to print debug messages, when in verbose mode
## @param: string to print
##
sub debug{
  print "DEBUG: " . $_[0] if $DEBUG;
}
```

;

8.6.2 classifications.dat

classifications.dat

```
hardware    00000001100000000000000000000000
software    00000000100100000000000000000000
mysql   00001000100000000000000000000000
apache      00000000100000000000000000000010
```

;

Bibliography

[1] S. Albayrak et al. “Agent technology for personalized information filtering: the PIA-system”. In: *Proceedings of the 2005 ACM symposium on Applied computing*. ACM. 2005, pp. 54–59.

[2] R. Barrett, Y.Y.M. Chen, and P.P. Maglio. “System administrators are users, too: designing workspaces for managing internet-scale systems”. In: *CHI’03 extended abstracts on Human factors in computing systems*. ACM. 2003, pp. 1068–1069.

[3] N.J. Belkin and W.B. Croft. “Information filtering and information retrieval: two sides of the same coin?” In: *Communications of the ACM* 35.12 (1992), pp. 29–38.

[4] A.D. Birrell and B.J. Nelson. “Implementing remote procedure calls”. In: *ACM Transactions on Computer Systems (TOCS)* 2.1 (1984), pp. 39–59.

[5] T. Bray et al. “Extensible markup language (XML)”. In: *World Wide Web Journal* 2.4 (1997), pp. 27–66.

[6] D. Crockford. *The application/json media type for javascript object notation (json)*. RFC 4627. IETF, 2006. URL: **https://tools.ietf.org/html/rfc4627**.

[7] M.R. Endsley. “Automation and situation awareness”. In: *Automation and human performance: Theory and applications* (1996), pp. 163–181.

[8] M.R. Endsley. “Design and evaluation for situation awareness enhancement”. In: *Human Factors and Ergonomics Society Annual Meeting Proceedings*. Vol. 32. 2. Human Factors and Ergonomics Society. 1988, pp. 97–101.

[9] L. Finkelstein et al. “Placing search in context: The concept revisited”. In: *Proceedings of the 10th international conference on World Wide Web*. ACM. 2001, pp. 406–414.

[10] A. Gohr. *DokuWiki*. Last visited on 21/03/2012. 2012. URL: **http://www.dokuwiki.org/dokuwiki**.

[11] E. Greengrass. “Information retrieval: A survey”. In: *DOD Technical Report TR-R52-008-001* (2000).

[12] E.M. Haber and J. Bailey. “Design guidelines for system administration tools developed through ethnographic field studies”. In: *Proceedings of the 2007 symposium on Computer human interaction for the management of information technology*. ACM. 2007, p. 1.

[13] T.H. Haveliwala. “Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search”. In: *Knowledge and Data Engineering, IEEE Transactions on* 15.4 (2003), pp. 784–796.

[14] P. Horn. “Autonomic computing: IBM’s perspective on the state of information technology”. In: *Computing Systems* 15.Jan (2001), pp. 1–40.

[15] M.K. Hui and D.K. Tse. “What to tell consumers in waits of different lengths: An integrative model of service evaluation”. In: *The Journal of Marketing* (1996), pp. 81–90.

[16] S. Lawrence. “Context in web search”. In: *IEEE Data Engineering Bulletin* 23.3 (2000), pp. 25–32.

[17] Best Practical Solutions LLC. *RT: Request Tracker*. Last visited on 21/03/2012. 2012. URL: **http://bestpractical.com/rt**.

[18] Munin-monitoring. *Munin-monitoring*. Last visited on 20/03/2012. 2012. URL: **http://munin-monitoring.org**.

[19] S.H. Myaeng and R.R. Korfhage. “Integration of user profiles: Models and experiments in information retrieval”. In: *Information processing & management* 26.6 (1990), pp. 719–738.

[20] F.F.H. Nah. “A study on tolerable waiting time: how long are Web users willing to wait?” In: *Behaviour & Information Technology* 23.3 (2004), pp. 153–163.

[21] N.F. Noy. “Semantic integration: a survey of ontology-based approaches”. In: *ACM Sigmod Record* 33.4 (2004), pp. 65–70.

[22] G. Salton and C. Buckley. “Improving retrieval performance by relevance feedback”. In: *Readings in information retrieval* (1997), pp. 355–364.

[23] G. Salton, A. Wong, and C.S. Yang. “A vector space model for automatic indexing”. In: *Communications of the ACM* 18.11 (1975), pp. 613–620.

[24] B. Sheth and P. Maes. “Evolving agents for personalized information filtering”. In: *Artificial Intelligence for Applications, 1993. Proceedings., Ninth Conference on*. IEEE. 1993, pp. 345–352.

[25] B. Snyder, D. Bosnanac, and R. Davies. *ActiveMQ in action*. Manning, 2011.

[26] S. Staab and R. Studer. *Handbook on ontologies*. Springer Verlag, 2004.

[27] *STOMP Protocol Specification, Version 1.1*. Last visited on 02/04/2012. 2011. URL: **http://stomp.github.com/stomp-specification-1.1.html**.

[28] R. Studer, V.R. Benjamins, and D. Fensel. “Knowledge engineering: principles and methods”. In: *Data & knowledge engineering* 25.1-2 (1998), pp. 161–197.

[29] M. Uschold and M. Gruninger. “Ontologies: Principles, methods and applications”. In: *Knowledge engineering review* 11.2 (1996), pp. 93–136.

[30] H. Wache et al. “Ontology-based integration of information-a survey of existing approaches”. In: *IJCAI-01 workshop: ontologies and information sharing*. Vol. 2001. Citeseer. 2001, pp. 108–117.

[31] C. Welty. “Ontology research”. In: *AI magazine* 24.3 (2003), p. 11.

[32] D.D. Woods. “Decomposing automation: Apparent simplicity, real complexity”. In: *Automation and human performance: Theory and applications* (1996), pp. 3–17.