Jarkko Laitinen

QCloud API for synchronous and asynchronous file transfer in Qt

Master's Thesis
in Information Technology
December 18, 2012

University of Jyväskylä

Department of Mathematical Information Technology

Jyväskylä

Author: Jarkko Laitinen
Contact information: jarkko.p.laitinen@jyu.fi
Title: QCloud API for synchronous and asynchronous file transfer in Qt
Työn nimi: QCloud API synkroniseen ja asynkroniseen tiedonsiirtoon Qtlla
Project: Master's Thesis in Information Technology
Page count: 91
Abstract: There is growing need for using services provided by the Cloud in applications. The differences between the largest service providers make the integration of applications difficult. Developers need to make different implementations to each service provider and it hinders the ability to use the services. In this thesis I compare the differences between Windows Azure and Amazon S3. I implemented an API according good API design guidelines that enables the developer to use different Cloud providers in their applications with minimal changes. The final API is functional, but there is still much to be done. In the study I discovered many minute, and not so minute, differences between the two Cloud providers, and it restricted the amount of functionality that could be placed in the API.
Suomenkielinen tiivistelmä: Pilvipalveluiden käyttö ohjelmissa on lisääntynyt, sillä niiden avulla voidaan tarjota uusia ominaisuuksia suhteellisen pienellä vaivalla. Kuitenkin eri palveluntarjoajien eroista johtuen pilvipalvelusta toiseen siirtyminen on vaikeaa, ja usein vaatii mittavaa uudelleen koodausta ohjelmassa. Tämän gradun tarkoitus oli yhdistää eri pilvipalveluntarjoajien palvelut saman APIn taakse, ja näin tarjota helpompi tapa siirtyä yhdestä palvelusta toiseen. Esitelty QCloud-API tarjoaa tarpeelliset ominaisuudet Windows Azuren ja Amazonin S3 palveluiden käyttöön yhden APIn kautta, jolloin palvelusta toiseen ei vaadi ohjelmassa kuin korkeintaan yhden rivin muutoksen.
Keywords: Qt, C++, Cloud, Amazon S3, Windows Azure, REST, API design
Avainsanat: Qt, C++, Pilvipalvelut, Amazon S3, Windows Azure, REST, APIn suunnittelu

Glossary

Qt: C++ framework developed by TrollTech, later purchased by Nokia, and finally by Digia.
API: Application Programming Interface, is a solution to a certain problem by providing a easier way to do it.
C++: Programming language developed by Bjarne Stroustrup in 1979.
REST: Representational state transfer, a stateless architecture for communication between servers and clients.
PaaS: Platform as a Service, is a service that provides a computing platform.
SaaS: Software as a Service, provides software on-demand in the Cloud.
IaaS: Infrastructure as a Service, offers virtual machines or physical computers as a service.
SLA: Service level agreement, is an document where the service provider describes the properties of the service.
QoS: Quality of Service, is requirements on a metric that reflects the subjectively experienced quality.
SP: Service Provider, The provider of a service in the Cloud, for example, Amazon, and Windows Azure.
Blob: Any kind of file that is located in the Cloud. Resides inside a bucket/container.
Container/Bucket: Top level directory in the Cloud.
AWS: Amazon Web Services, a collection of services that together make up a cloud computing platform.
EC2: Amazon Elastic Compute Cloud, scalable private virtual services.
S3: Amazon Simple Storage Service, service providing storage. SimpleDB: Amazon SimpleDB, service providing distributed database.
SWF: Amazon Simple Workflow, a workflow service providing control on distributed services.
VM: Virtual Machine, a software implementation of computer.
Azure: Windows Azure, Microsoft's cloud computing platform.
IIS: Internet Information Services, a web server application.

RESTful: Conforming to REST constraits.
OpenGL: Open Graphics Library, a cross-language multiplatform graphics API.
GPL: GNU General Public License, software license written by Richard Stallman.
LGPL: GNU Lesser General Public License, software license written by Free Software Foundation.
JVM: Java Virtual Machine, virtual machine for executing Java bytecode.
Qt Quick: Declarative application framework.
MOC: Meta-Object Compiler, generates C++ code from Qt specific macros.
IDE: Integrated Development Enviroment, provides tools for software development.
Qt Creator: a cross-platform C++ IDE.
MinGW: Minimalist GNU for Windows, native port of GCC for Microsoft Windows.
HMAC: Hash-based message authentication code, an algorithm that can be used to verify authenticity and data integrity.
QDoc: a documentation tool to generate documentation from notation in source code.
QTestLib: a unit test framework provided by Nokia to Qt framework.

Contents

1 Introduction

In my thesis I will introduce the concept of Cloud API (Application Programming Interface) to the Qt framework. Cloud computing (from now on Cloud) can be considered as a diversion of computing from the small scale to the large scale. The transformation can be attributed to large e-commerce companies and their need to provide the best possible customer experience in their stores. This led to building of huge datacenters around the world. Amazon was the first to commercialize their existing infrastructure as they began renting their services in 2007 [61]. Cloud can be seen as the next paradigm shift in the IT and the reprecussions will echo through the whole industry [40]. From how a developer develops his application to how a user uses the application, Cloud can change it all.

The goal of this thesis is to provide a sample software using an early version of the QCloud API that I have created. I will also go through the different iterations of the API and the programs that helped me in creating it. The program shows one example how cloud computing can help the software developer. I will concentrate how the different providers differ and how their implementations of REST (Representational state transfer) can be used together. At the moment there are two competing implementations to provide communications from clients to the server, REST and SOAP (Simple object access protocol). REST has a much lower overhead and the implementation of the protocol is much easier, so I will concentrate on the REST implementations. The need for QCloud API comes from the fact that Qt is lacking an uniform way to incorporate the different cloud services in applications. QCloud will support Amazons AWS and Microsofts Azure at first but the support for different service providers can be inspected later. Most of the providers offer Java toolkits, but none offer Qt/C++. The ability to use cloud services is a must in future and so API providing these features is needed.

This thesis consists of 5 general chapters. In the first chapter I will go through the history of the Cloud, how it developed and cloud providers that I will use in the later parts of the thesis. The second part is on Qt framework and how it developed. Also I'll go through the basic improvements that Qt gives compared to basic C++ frameworks. In the third chapter I will try to explain how software development in

the cloud differs from software development for local machines. I will also compare the REST-API implementations of the service providers and present two examples on how the cloud can enhance programs. The last chapter will be on QCloud that is the prototype of a general Cloud API for Qt. I will go through the steps that I took when designing the API and provide a working example how to integrate the Amazon S3 cloud file storage and Windows Azure to Qt applications.

2 Cloud computing

Cloud computing is one of the buzzwords of the 2010 decade. It has been hailed as the next big thing in the IT-sector. Cloud can help a starting company to publish their software without the need for an expensive server acquisitions. Or a company can first see how their application starts off and then buy their own servers according to the usage levels seen in the cloud. These providers strive to provide the best possible QoS (Quality of Service) in order to make the service always available.

In this chapter I will go through the history and phases of cloud computing. The origins of peer-to-peer computing, grid computing and the transformation to Cloud. I will compare grid and cloud and after that one should have a good understanding of the differences and similarities of these. After explaining the terms I will go through three different cloud providers: Amazon, Microsoft Azure, and Google AppEngine. I will compare the features, pricing, and technologies that these three provide. Lastly, I explain the aspects of security in cloud computing and how the security affects the transformation of the current paradigm to the cloud paradigm. After this chapter one should have a good understanding of the cloud paradigm and in which ways the service providers can differ from each other.

2.1 From Distributed Applications to Cloud

The first ancestor of cloud can be thought as Peer-to-Peer (P2P) computing. P2P is defined by Dejan S. Milojicic et al. as follows [56]: "a class of systems and applications that employ distributed resources to perform a function in a decentralized manner." P2P computing is an evolution of the basic concept of distributed computing and deepens the possibilities of sharing and cooperation between heterogenious entities. Distributed computing can be thought as the distribution of solving a problem, program, or resources on different computers connected via network. David Barkai brings forth the three aspects of P2P computing in his article "Technologies for Sharing and Collaborating on the Net" [41]:

- *The "action"*: computing, file sharing, communication, etc.., is taking place at the edge of the Net, where users and devices are.

- *Resources are being shared*: They might be computing cycles, storage, network bandwith, content and more.
- *Direct communication between peers*: PCs or other devices at the edge of the Net, is almost always present.

P2P is ancestor of Cloud in a manner that entities use distributed resources towards a central goal.

Grid computing can be thought as another anchestor of the Cloud. In the book "The Grid: blueprint for new computing infrastructure" [48] Ian Foster and Carl Kesselman talked about computing being an utility. The ability to have almost unlimited resources available to everyone on demand. They were not the first because the first real mention about the paradigm of computing as utility comes from a press release [54] put out by UCLA on the founding of the internet (at the time ARPANET). The first paragraph was as follows: "As of now, computer networks are still in their infancy. But as they grow up and become more sophisticated, we will probably see the spread of "computer utilities" which, like present electric and telephone utilities, will service individual homes and offices across the country". This vision has proven to be the driving force behind the birth of ARPANET and later the Internet. One of the most cited grid computing definitions comes from Ian Foster and Adriana Iamnitchi [47] : "Grids are sharing environments implemented via the deployment of a persistent, standards-based service infrastructure that supports the creation of, and resource sharing within, distributed communities. Resources can be computers, storage space, sensors, software applications, and data, all connected through the Internet and a middleware software layer that provides basic services for security, monitoring, resource management, and so forth." A grid is not useful for time constrained operations because the fact that the resources are connected to eachother via the Internet brings latency to the equation.

There are many public grids where users download an application that then can use the computer's resources when the user doesn't need them. Good examples are BOINC-project [8] and SETI@Home [28]. BOINC is a collection of different calculation projects of which a users can choose a favourite. BOINC has approximately 2.3 million users [9]. SETI@Home on the other hand analyzes radio telescope data and has over 3 million users. Grids can also be private. Private grids differ from public grids as the resources come from the same provider. Notable private grids are the ones at Cern [16]. Cern is the European organization of nuclear research and they do experiments with particle accelerators in Switzerland. The need for grids came from

the huge amounts of data that the measuring devices produce in the experiments. With the grid Cern can handle the data and analyze it in a reasonable amount of time.

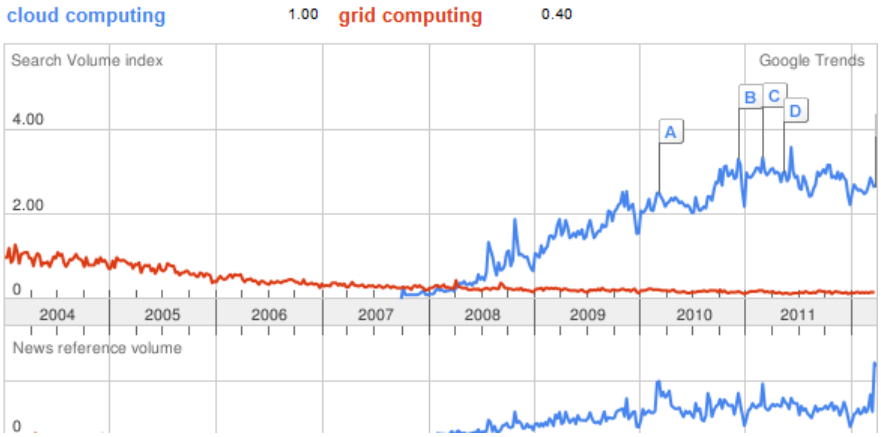

Figure 2.1: Search frequency of Grid and Cloud from Google Trends [15]

Like we can see from Figure 2.1, in early 2008 "cloud computing" was searched more than "grid computing". The change in search frequencies was quite fast since the term "cloud computing" was conveinced in the late 2007. There is no consensus regarding who coined up the term, but the first real appearance of it can be found in a paper by Sharon Eisner Gillett and Mitchell Kapor [50]. The paper "The Self-Governing Internet: Coordination by design" has many of the key concepts of cloud explained and detailed. The invention of the term has also been credited to Google's Eric Schmidt [11], but he just popularisized it. One could say that the difference between cloud and grid is the means of usage and the distribution of the resources. The available grid computing platforms usually are within a company and are used for a certain project, usually calculations. This differs from the now prevailing cloud computing. Vaquero, Rodero-Merino, Caceres and Lindner [60] suggested a definition for cloud computing as follows: "Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load(scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Service Provider (SP) by means of customized SLAs (Service Level Argeements)."

Cloud can be thought to be the unification of the P2P and Grid computing paradigms. Where in P2P heterogeneous computers are connected to each other via the Internet and share resources, in a (private) grid homogeneous resources are connected to each other via the ethernet. Public grids can be thought to be centralized P2P systems that use resources from the peers towards a central task. Clouds provide an agreement between the provider and the developer which defines the way that the developer wants the service to be provided.

One of the key factors in the growth of cloud services is the building of huge datacenters. It is not news that many of the cloud providers have history in e-trade or other services that require the service to be always available. But the usage of the datacenters can vary because the load is not always the same. This can lead to the possibility of renting these machines to a third party in the manner of cloud services. The renting has grown easier with the usage of virtualization [61]. Virtualization means that an operating system runs inside a virtual machine which mimics the properties of a computer. Providing a virtualized computer allows the SP's to run many images on one machine and thus making the utilization of the machine better. Virtualization provides homogenous slices of the infrastructure and makes the renting of unused resources easier. Amazon was one of the first to commercialize the cloud with their EC2 service in 2006 [35]. One of the real advantages of the cloud is the QoS requirements of the SP's: the service is almost always online and available.

2.2 The Cloud Stack

The abbreviations SaaS (Software as a Service), PaaS (Platform as a Service), and IaaS (Infrastructure as a Service) are used in conjuction with cloud. In Figure 2.2 these abbreviations have been placed on the cloud stack. The following explanations of the abbreviations is based on the National Institute of Standards and Technology (NIST)'s definitions of Cloud Computing [55].

2.2.1 SaaS

NIST defines [55] SaaS as follows: "The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such

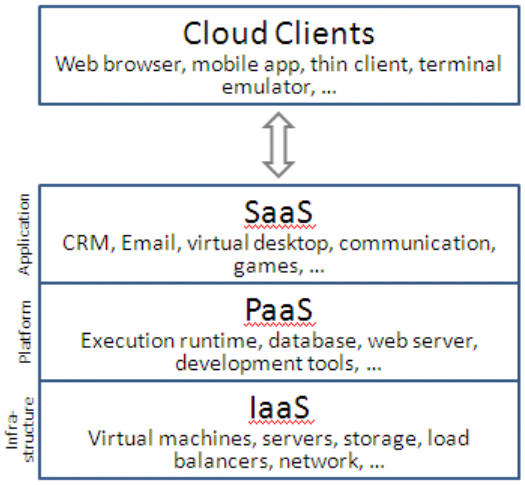

Figure 2.2: The cloud stack, picture from Wikipedia [10]

as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited userspecific application configuration settings." A good example of a SaaS service is the Google's Apps [14]. They provide typical office tools needed for text editing, making presentations, making spreadsheets, etc.

2.2.2 PaaS

NIST defines PaaS as follows [55]: "The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment." Microsofts Windows Azure and Googles App Engine are good examples of PaaS services. These provide, like the definition implies, the deployment platform for programs that are then used as SaaS. The development is usually done by provided APIs and using various languages.

2.2.3 IaaS

NIST defines [55] IaaS as follows: "The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls)." A good example of IaaS provider is Amazon with their EC2 instances.

2.2.4 Scalability

Armburst et al. [40] mentioned scalability as one of the obstacles of cloud computing in their paper "Above the Clouds: A Berkeley View of Cloud Computing". They point out that one of the biggest obstacles to the software developer is to implement parallelism and scalability to their programs efficiently. They also point out that the scalability should be two-way, which means that the amount of instances should subtract when the need is low. The implementation of scaling and parallelization is a challenge yes, but cloud SP's usually provide a well thought out APIs so that the infrastructure side of the equation would be as easy as possible. The implementations of the API's are different between all of the cloud SPs so it is again a hindering fact when choosing a SP.

Imagine a scenario where your service is featured on a technical blog or in twitter. The traffic to a site can grow exponentially compared to normal levels. If the infrastructure is located on company's own computers, then the load can cause the service to become unavailable. All of the cloud providers that I have mentioned provide on-demand scalability of the deployed applications. This is one of the best examples of the power of scalable resources that the cloud provides.

The scalability also applies to storage of which the cloud can provide almost unlimitedly. For example, the highest pricing group of Amazons S3 is over 5000TB [5], so the amount that one can store in the cloud is almost unlimited. When considering using that much space on the Cloud one of the biggest challenges is how to get the data to the SP. Armburst et al. [40] present the option of sending hard drives overnight to the cloud provider. This would provide larger bandwidth when comparing to the 5-18Mbits/s that S3 can usually provide [49]. The transfer of data

is usually billed by the SP so the overnight option would also be cheaper. I will go through the scalability issues again when going through each of the SPs.

2.2.5 Security

Security is one of the biggest questions in cloud. For example, in study done in 2009 [30] IT executives said that despite the potential benefits, they still trust existing internal systems over cloud-based systems due to fear about security threats and loss of control of data and systems. The fact is that when you take your services to cloud you really have to trust the company from which you lease the service. The data is located on their devices and there is the possiblity that some not-so-friendly person can access it leading to security breaches. Jon Brodkin mentions in his article the seven key security risks [44] that the user should raise with the vendor before committing. They are as follows:

- *Privileged user access*: Sensitive data processed outside the enterprise needs the assurance that they are only accessible and propagated to priviledged users.
- *Regulatory compliance*: A customer needs to verify if a Cloud provider has external audits and security certifications and if their infrastructure complies with some regulatory security requirements.
- *Data location*: Since a customer will not know where her data will be stored, it is important that the Cloud provider commit to storing and processing data in specific jurisdictions and to obey local privacy requirements on behalf of the customer.
- *Data segregation*: One needs to ensure that one customer's data is fully segregated from another customer's data.
- *Recovery*: It is important that the Cloud provider has an efficient replication and recovery mechanism to restore data if a disaster occurs.
- *Investigative support*: Cloud services are especially difficult to investigate, if this is important for a customer, then such support needs to be ensured with a contractual commitment.
- *Long-term viability*: Your data should be viable even when the Cloud provider is acquired by another company.

The data location is something that one should ponder before choosing a service. The location of the service will decide what kind of laws should be complied with. Encryption of the data sent to the cloud is one of the choices that an individiual can take, thus providing better safety for his/her data. I will go through the SP specific details in the later chapters.

The risks that Jon Brodkin mentioned in his article are not the best frame of reference, they are risks not facts. Joshi et al. [52] list the main goals of information security in their article "Security Models for Web-Based Applications". They are as follows:

- *Confidentiality*: Information can be accessed only by authorized entities.
- *Integrity*: Data can be modified only by authorized entities.
- *Availability*: Ensures the information is available when needed and is not made inaccessible by malicious data-denial activities.
- *Accountability*: Every action taken by an entity can be traced back to the entity.
- *Assurance*: Degree of confidence in the security of the system with respect to predefined security goals.

When comparing the risks that Brodkin mentioned to the goals from Joshi et al we can see that the basic ideas are quite similar. I will use the Joshie et al. frame of reference when inspecting the SP's security aspects. The two first items from the frame are quite similar and can be thought of as authentication. From now on, the Joshi et al. frame will be referred to as 4A.

2.3 Service providers

There are many different service providers in the cloud ecosphere. The user should select cloud SP that is best suited to the user's needs as there can be differences in the provided services. The differences between cloud SP's can hinder the generalizability of the services [40]. The offered API's differ between almost all of the SP's and it leads to the fact that moving from one cloud SP to another is really difficult. This is one of the ten risks that Armburst et al mention in their article [40] and they recommend a standardized API to be adopted. Pricing of the offered services can be

different from each other so comparison between SP's can be difficult. In this chapter, I will go through three different SPs: Google's AppEngine, Microsoft's Azure, and Amazon's AWS. I will go through the offered API's, services, and pricing of the service.

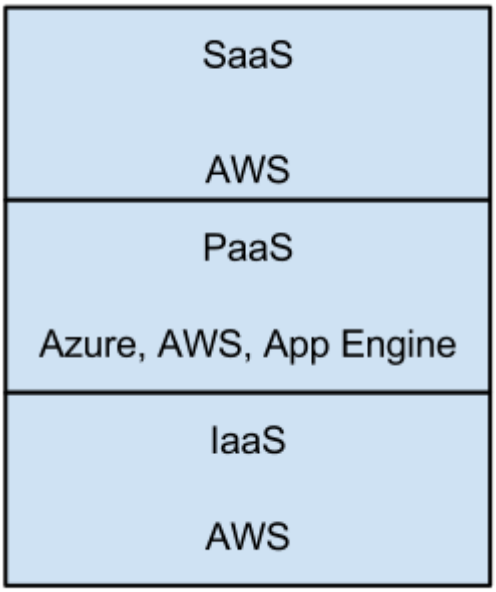

Figure 2.3: SP's location on the cloud stack

2.4 Amazon

Amazon was probably the first SP in cloud computing as they started Amazon Web Services (AWS) in 2006 [35]. AWS offered IT infrastructure to businesses in the form of web services. This model is now known as the cloud. Amazon's services are mostly in the IaaS level but they do offer PaaS and SaaS level services as well. Amazon offers many different services today, and they have data centers in U.S., Europe, Singapore, and Japan. Amazon offers a total of 34 different services in the following categories:

- Computing
- Networking
- Content Delivery

- Payments & Billing
- Database
- Storage
- Deployment & Management
- Support
- Web Traffic
- Application Services
- Workforce

From these categories I will concentrate on computing and storage. Amazon Elastic Compute Cloud or EC2 is one of the best examples of an IaaS offering. EC2 is web service that provides resizable computing capacity in the cloud. EC2 is a virtual machine in which you can choose an own image or one of the ready-made Amazon Machine Images (AMI) to run in the cloud. The instances boot in minutes and there are APIs which help you to do that programmatically. Amazon provides means to start new instances on-demand so usage spikes won't harm the availability of the service. The almost endless amount of computational power has led to interesting revelations. For example, one can use 100 computers to do calculations and it is the same price as if one had used one computer for 100 hours. Development can be done using almost any programming language because the cloud is running on a VM that you have complete control over. One can choose which Linux distribution to use and what the distribution contains.

2.4.1 Pricing

Amazon provides six types of general virtual machines that have different properties and prices. The micro instance which is in the free-tier of usage has 613MB of memory and offers 2 Elastic Compute Units (ECU) of cpu power. ECU is Amazon's own unit and it is equivalent to an 1.0-1.2GHz 2007 Opteron or Xeon processor [3]. On the other end of the spectrum is the cluster computing instances. The biggest rentable cluster has 60.5GB of memory, 88 ECU of cpu, and 3370GB of local storage. The price difference between these is (in Linux usage and in US East) $0.020/h in the micro and $2.400/h in the grid.

Amazon Simple Storage Service or Amazon S3 is the storage service that Amazon provides in the cloud. The stored objects are stored in buckets that have their own adresses and can be accessed via REST interface. The default protocol for downloading the files is HTTP but Amazon provides BitTorrent alternative for large files. The pricing of these services is by storage and request amounts. The pricing of storage starts from $0.125/month per GB if using under 1TB to $0.055/month per GB if using over 5000TB. The requests are $0.01 per 1,000 requests for PUT, COPY, POST or LIST and $0.01 per 10,000 GET and other requests.

The free-tier of AWS has the following properties [4]:

- 750 hours of EC2 Linux or Windows instance usage with load balancing,
- 30GB of Amazon Elastic Block Storage, plus 2 million I/Os and 1GB of snapshot storage.
- 5GB of S3 storage with 20,000 get and 2,000 put requests.
- 100 MB of storage, 5 units of write capacity, and 10 units of read capacity for Amazon DynamoDB.
- 25 Amazon SimpleDB Machine Hours and 1 GB of storage.
- 1,000 Amazon SWF workflow executions can be initiated for free. A total of 10,000 activity tasks, signals, timers and markers, and 30,000 workflow-days can also be used for free.
- 100,000 Requests of Amazon Simple Queue Service.
- 100,000 Requests, 100,000 HTTP notifications and 1,000 email notifications for Amazon Simple Notification Service.
- 10 Amazon Cloudwatch metrics, 10 alarms, and 1,000,000 API requests.
- 15 GB of bandwidth out aggregated across all AWS services.

Anything going over these amounts is billed on regular prices. The free-tier can be used to host own web sites or run your own SaaS applications or just as a storage device.

2.4.2 Scalability

Scalability can be implemented by the user with the EC2 API, S3 API, or the general Amazon REST API. Amazon also provides a service called CloudWatch that offers seven pre-selected metrics to monitor the usage of different resources (storage, computation, bandwidth) for free [2]. CloudWatch can be automated to take action if the metrics show a rapid change in the usage of resources. This can be done manually via the developer implementing a load balancing algorithm using the metrics and API's or using the Auto Scaling feature. Auto Scaling does what the name implies; enables automatic scaling of the EC2 instances. S3 services scale automatically as the amount of data is increased.

2.4.3 Security

Amazon has published a white paper [6] on their approach for security in their services. I will not go through the whole document, but inspect what they promise in the 4A frame of reference.

Amazon has a four-pronged approach to security on high level [7]:

- *Certifications and Accreditations*: Amazon has audits performed by outside entities.
- *Physical Security*: The location of data centers are on a need to know basis. Data centers are also protected with many physical security measures to prevent unauthorized access.
- *Secure Services*: All of the services in AWS have been designed to be secure and restrict unauthorized access.
- *Data Privacy*: Users can enable encryption of data in the AWS Cloud.

When reflecting the Amazon security model against the 4A frame of reference, one notices that they go well together. Amazon provides confidentiality via the authorization needed to access services. Information integrity is provided also by the authorization process as only authorized entities can access data. Amazon provides availability via decentralizing the locations of their data centers. If a data center serving certain entity goes cold the service is transferred to another data center. Amazon provides accountability in their S3 service via ability to log every request

done to access data. Amazon has external audits done regularly on their AWS-services so assurances are made that the system will comply to predefined security goals.

2.4.4 REST implementation

Amazon offers RESTful interface to control and access the S3 storage service. The basic operations of PUT and GET can provide almost everything that one needs in file handling between physical storage and cloud storage.

The basic flow of getting a file from a bucket is as follows: getting the listing of buckets in the users account, getting the contents of a certain bucket, and after that getting a file from the bucket. In each request, an authorization header, date-header (seconds since epoch), and a HMAC-SHA1 hash of the request are included. The HMAC-SHA1 hash includes the user's secret access key so Amazon knows that the user is allowed to access the files. The specific structure of the requests can be found from Amazon's S3 REST API documents [18]. I will compare the structures of the requests in detail in Section 4.3.

2.5 Windows Azure

Azure is the Microsoft's cloud service. Microsoft describes it like this: "Windows Azure is an open and flexible cloud platform that enables you to quickly build, deploy and manage applications across a global network of Microsoft-managed datacenters. You can build applications using any language, tool or framework. And you can integrate your public cloud applications with your existing IT environment." [39] There are three different types of instances, which Microsoft calls compute roles, in Azure:

- Web applications
- Backend applications
- Legacy applications

Web applications are defined as follows: "Web roles in Windows Azure are special purpose, and provide a dedicated Internet Information Services (IIS) web-server used for hosting front-end web applications. You can quickly and easily deploy web

applications to Web Roles and then scale your Compute capabilities up or down to meet demand" [39]. So web roles can be thought as the SaaS offering of Azure.

Backend applications are defined as [39] "Applications hosted within Worker roles can run asynchronous, long-running or perpetual tasks independent of user interaction or input. When you separate your application's background processes in a Worker role and host the front-end in a Web role, you can better distribute your application logic and have more fine grain control over how your application scales." One could say that the worker roled applications are similar to the PaaS level of Azure.

The final offering is legacy applications and they are defined as: "Virtual Machine (VM) roles, now in Beta, enable you to deploy a custom Windows Server 2008 R2 (Enterprise or Standard) image to Windows Azure. You can use the VM role when your application requires a large number of server OS customizations and cannot be automated. The VM Role gives you full control over your application environment and lets you migrate existing applications to the cloud." [39] This can be thought as the IaaS offering because you get a fully working VM powered OS to use. Azure offers services from all levels of the cloud stack. The development for Azure is done primaly with .NET, node.js, Java, PHP or C++. Azure supports also other languages via Azure service API's (HTTP/REST).

2.5.1 Pricing

Pricing on Azure is similar to other providers. Microsoft offers a three month free trial after which the service has to be re-uploaded to the cloud and the developer has to accept a paid subscription. There are five different sizes of compute instances: extra small, small, medium, large, and extra large. These differ from one another on the amount of storage and prosessors. For example, the extra small has no own prosessors but they are shared with other instances and the amount of memory is 768BM. Cost of that is $0.02/h. On the other hand, the extra large has 8 cpu cores, 14GB of RAM, and costs $0.96 an hour.

Azure offers also cloud storage that is available via RESTful interfaces. The pricing of the space is $0.125 per GB stored per month for amounts under 1TB and $0.01 for 10,000 storage transactions.

2.5.2 Scalability

Scalability on Azure is not as elastic as on the other SP's. The amount of servers is controlled by the user via service configurations. If the user notices that the utilization of the server is high, he/she can start a new server by changing the service configurations. There is no ready scaling engine but the user has to code it him/herself. Microsoft provides examples [38] on scaling engines that survey the usage metrics and then change the amount of needed servers.

2.5.3 Security

Microsoft provides a white paper [53] that goes through the security aspects of Azure and it's service. Microsoft provides the following assurances [53]:

- *Confidentiality*: Customers data can be accessed only by the allowed entities.
- *Integrity*: Customeres can trust that their data won't be changed in the Cloud.
- *Availability*: Che data is replicated in three different nodes so if one goes down there is a backup.
- *Accountability*: Azure provides tools to monitor the clients data so everything can be monitored.

Azure also provides an authentication service called Access Control Service 2.0 that can help the developer to include access control on the application or service. ACS includes support for Active Directory Federation Services (AD FS 2.0), and popular web identity providers such as Google, Windows Live, Yahoo, and Facebook [1]. When registering to Azure, the developer identifies with the Windows Live account.

The four points presented in the white paper and ACS provide assurances that all of the points of the 4A frame of reference are taken seriously. Azure has received security certifications from external auditors [26].

2.5.4 REST implementation

Azure provides RESTful interfaces for service management and storage management. I will concentrate on the storage management. The GET request is formed with the following parameters: the storage accounts name, container, and the wanted

file. The request needs also Authorization-header, Date-header, and x-ms-version-header. The Authorization header is made from the request itself taking the verb, context, date, and canonized resource and then encoding the string with the HMAC-SHA256 algorithm. PUT-requests have the same basic structure as GET-requests, but add a Context-Length field to the packet. A more thorough examination can be found in Section 4.3.

2.6 Google App Engine

Google and their App Engine is located on the PaaS level of the cloud architecture as shown in Figure 2.3. App Engine is defined as "a system that exposes various pieces of Google's scalable infrastructure so that you can write server-side applications on top of them" [13]. AppEngine offers several critical features that are required to make application that can run under heavy load and need large amounts of data. The features are as follows [36]:

- Dynamic web serving, with full support for common web technologies.
- Persistent storage with queries, sorting, and transactions.
- Automatic scaling and load balancing.
- APIs for authenticating users and sending email using Google Accounts.
- A fully featured local development enviroment that simulates Google App Engine on user's computer.
- Task queues for performing work outside of the scope of a web request.
- Scheduled tasks for triggering events at specified times and regular intervals.

At the moment, App Engine supports applications written in several different languages: Java, Python, and Go are among the supported ones. App Engine also features REST-APIs that a developer can use. In Java runtime enviroment developer can develop the program with standard Java technologies, including JVM, Java servlets, and the Java programming language - or any other language using a JVM-based interpreter or compiler, such as JavaScript or Ruby. App Engine features two Python runtime enviroments which both include a fast Python interpreter

and Python standard library [36]. Go language is supported in the Go runtime enviroment that runs natively compiled Go code. In this context, the runtime enviroment is a sandboxed secure enviroment that limits the access of the programs to the underlying operating system. The limitiations are as follows [36]:

- Communication with the server and the client is done only via HTTP (or HTTPS) requests.
- Applications cannot write data to file system in any of the runtime enviroments. An application can read files, but only files uploaded with the application code. The app must use the App Engine datastore, memcache or other services for all data that persists between the requests.
- Application code only runs in response to a web request, a queued task, or a scheduled task, and must return response data within 60 seconds in any case. A request handler cannot spawn a sub-process or execute code after the response has been sent.

2.6.1 Pricing

The pricing of App Engine is similar to other PaaS services. There exists a free tier where an application can use up to 1GB of storage and up to 5 million page views in a month. If user wants to use more resources he has to enable billing and after that the resources are billed on usage. The usage is defined by a daily quota. User sets a daily max amount of usage and the App Engine allocates the resources up to that amount.

The real costs of App Engine is divided between the frontend instances, backend instances, bandwidth, stored data, and datastore operations (write, read, small). The daily quota is consumed by all of these. For example, the cheapest frontend instance (F1) is $0.12/h and cheapest backend instance is $0.08/h. The most expensive frontend instance (F4) is $0.32/h and the most expensive backend instance (B8) is $0.64/h. Stored data in blobstore is $0.13/GB/month and stored data in datastore is $0.24/GB/month. Datastore operations are between $0.07 and $0.10 for 100,000 operations.

2.6.2 Scaling

There is little information available on how scaling is implemented in App Engine. The only thing that Google provides is the following sentence [37] : "Automatic scaling is built in with App Engine, all you have to do is write your application code and we'll do the rest. No matter how many users you have or how much data your application stores, App Engine can scale to meet your needs."

2.6.3 Security

There is not a lot of information published about Google's perspective on information security in App Engine. Security assurance is provided by audition [27] of a third party auditer. App Engine is also tested against the SSAE-16 standard [27], thus making sure that the security is not just trusting Google's word. The authentication process is done with a Google account.

2.6.4 REST implementation

The context of this paper revolves around the construction of QCloud API that provides the basic file storage API's for Azure and S3. The decision to exclude App Engine from the comparison comes from the completely different nature of the service. App Engine requires the user to have the program already in the cloud in order to access the storage services.

2.6.5 Summary of the service providers

Now that I have gone through the SP's that I will concentrate on a short summary is in order. The SP's differ from one other on almost all of the details. On one hand Amazon is a free-for-all provider that gives you a virtualized computer to work on and release your applications, and on the other end of the spectrum is GAE which provides scalability via restricting the developers abilities to a certain degree. Azure is the middle ground between these two providing services from all of the levels of the cloud stack, but the free-tier does not exist.

There are open source services that provide their own PaaS implementation that one can deploy to a Cloud of their liking. For example, Open Stack [57] is an open source Cloud computing platform that provides different implementations for compute, storage, and networking. Apache's CloudStack [59] is another good example.

These open source alternatives can be deployed in already existing infrastructure and thus provide the benefits of the Cloud, for example elasticity. CloudStack supports the EC2 and S3 API's and thus applications using those services can be transferred to private Cloud's easily.

3 Qt

In this chapter I will go through the history and initial development of the Qt-framework. I will explain the special features of Qt and why it is used today. Also I will explain the basic development flow on Qt. Qt has an original view on event handling called signals and slots so those will be explained in this section. After looking in the past I will go through the future of Qt and how Qt 5.0 will change the development paradigm that is used today. Then I will present the example programs found in Qt 4.8 and the tools for testing and documenting. After this chapter you should have a good understanding on the history, development, program structure, testing and documenting Qt programs.

3.1 History

The history is based on the chapter "A Brief History of Qt" in the book C++ GUI Programming with Qt 4 by Jasmin Blanchette and Mark Summerfield [42].

Haavard Nord and Eirik Chambe-Eng began creating an object oriented display system in 1991 that would eventually develop into Qt. The main goal was to make it easier and faster to develop applications to the X11-window manager. Three years later they founded Quasar Technologies which was later renamed as Trolltech. Qt-framework was developed in Trolltech until the year 2008 when Nokia bought the company. At the same time Trolltech was renamed to Qt Software.

The first official release of Qt-framework happened in 1995. At this time Qt was ported to Unix and Windows. The reception of Qt was positive, but the initial usage was limited. At this time Qt was licenced under two different licences: a commercial one that required payments to Trolltech and a free one called FreeQt. One of the biggest early adopters was KDE. KDE is an international free software community that also develops the KDE Plasma Workspaces graphical enviroment. As the time passed the support for different platforms was improved in Qt, and in the version 3 the support for Mac Os X was added.

A notable milestone in the history of Qt came in 2008 when Nokia bought Trolltech and the Qt-framework. After the switch of ownership there were changes done

to the framework to match Nokia's vision. Less used parts of the framework were deprecated and the development of QTopia was cancelled. QTopia was a framework intended for mobile devices and as Nokia was building their own implementation the need for QTopia reduced. Qt was also licenced under LGPL-licence. As the QTopia project was cancelled the development began on a new API for mobile devices. The API was called Qt Mobility. Qt Mobility offers general components that are needed to make an application for a mobile device.

Another notable milestone came when Nokia decided to change their mobile operating system to Windows Phone. This lead to the sale of the Qt commercial side to Digia. Nokia is still a valued member of the Qt-project which is the open source and free part of the Qt-platform. Commercial licences are now bought from Digia.

Originally Qt supported just *nix and Windows, but throughout the years many platforms have been added to the support. For example, Java (JVM), embedded Linux, Linux-based mobile platforms (Meego, Maemo), and S60-Symbian have been included. Development language with Qt is C++ but many bindings for different programming languages have been made. For example, Java, Python, C#, Haskell, and many others are supported officially or via community made bindings.

Today Qt is a comprehensive framework that includes everything needed to make platform independent software. Because Qt is offered free via the LGPL license, many open source projects have adopted Qt. For example, Google Earth, Skype, Opera, and VLC media player have been created using Qt.

3.2 License

When Qt was originally released it was under Trolltechs proprietary FreeQt-licence and Windows development was closed and required a license from Trolltech. The free licence applied only to software written to *nix and that made the source code available after release. Also commercialization of software was forbidden under the FreeQt licence. Trolltech was also the highest power when considering changes to the Qt-framework. This kind of licencing was frowned upon by the open source community and they required changes to the licence.

Trolltech changed the FreeQt-licence to comply closer to GPL (Gnu Public Licence) and renamed the license to QPL (Qt Public Licence). This was not enought to please Free Software Foundation (FSF) and as a compromise Trolltech announced that if they ever stopped developing the open source version of Qt, the open source

community could take it under their wing. In the end of 2000 Trolltech moved Qt under GPL and after Nokia purchased them, the license was changed to LGPL.

Today, there are 3 different types of licences used in conjuction with Qt: Commercial, GPL, and LGPL. Commercial licence is needed to make commercial software and the licences are bought from Digia. The license also brings full support from Digia and offers extra components to be used in the applications. The free lincenses (LGPL and GPL) have their own requirements on how the source code is to be handled after the application has been released.

3.3 Qt framework

This chapter is based on Qt modular class library site [19].

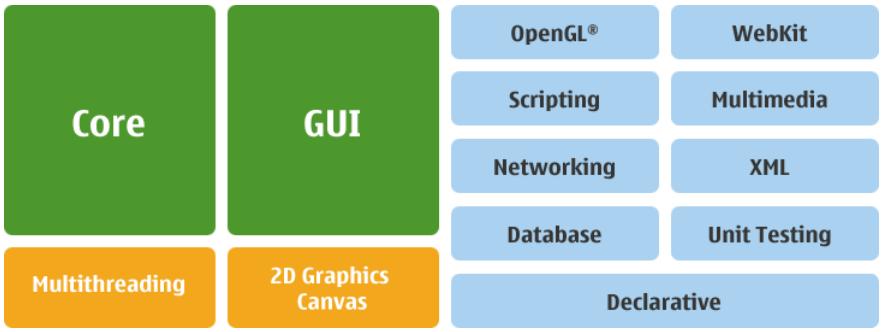

Figure 3.1: The Qt Class Library, from qt.nokia.com [19]

In Figure 3.3, the Qt frameworks class library is depicted. Qt has components to make full-fledged software that is platform independent. The framework is modular so that the developer can include only the parts that are needed. I will go through the framework and give examples of the contents of each module [19].

- *Core module* includes all of the basic classes needed for development. The classes include file I/O, event/object handling, multi-threading and concurrency, plugin and setting management and the Signal & Slot interobject communication mechanism.

- *GUI module* contains functions needed to create GUI (graphical user interface)'s. The functions include widgets, 2D-canvas with OpenGL-integration, Style engine, and other functions that help shape the interface. With the style

engine developer can change the programs style to match his/her vision. With the GUI-module developer can change the look and feel of the UI very liberally. The structure of the user interface resides in a *.ui file that is in XML-format.

- *Declarative module* is is the key component of QtQuick user interface creation kit that will be the preferred means of UI creation in Qt5. It provides classes for creating highly dynamic, custom user interfaces for touch-enabled and mobile devices.

- *Qt Webkit module* is an implementation of the WebKit web browser engine to Qt. It allows easy integration of web pages to applications.

- *Qt Script module* is an ECMA standard scripting engine based on JavaScript-Core back-end. Qt Script provides QObject integration, brings Qt's Signal & Slot mechanism to scripting, and allows integration between C++ and scripting.

- *Networking module* provides provides complete client and server socket abstraction, and implements common protocols such as HTTP, FTP, and DNS, i ncluding support for asynchronous HTTP 1.1.

- *Database module* brings database integration to Qt applications. It supports all major database drivers, and lets the developer send SQL to the database, or have the Qt SQL classes generate queries automatically.

- *Unit testing module* provides the abilities that are normally found on testing frameworks to Qt applications and graphical user interfaces.

- *XML module* provides a stream reader and writer for XML documents, C++ implementations of SAX, and DOM and an XQuery & XPath engine. XQuery is a simple SQL-like query language for traversing XML documents to select and aggregate XML elements of interest and transform them for output as XML or in some other format. XQuery simplifies query tasks by eliminating the need for large amounts of procedural programming in C++.

- *OpenGL module* offers classes that make it easy to incorporate 3D graphics with OpenGL and OpenGL ES in Qt applications. OpenGL is the standard graphics library for building cross-platform, hardware-accelerated, high performance

visualization applications. While OpenGL is ideal for 3D visualization, it offers little support for creating application user interfaces.

- *Multithreading features* Qt's cross-platform multi-threading functionality simplifying parallel programming, while added concurrency features make it easier to take advantage of multi-core architecture.

- *Qt Graphics View module* provides a surface for managing and interacting with a large number of custom-made 2D graphical items, and a view widget for visualizing the items, with support for zooming and rotation. Graphics View uses a BSP (Binary Space Partitioning) tree to provide very fast item discovery, and as a result of this, it can visualize large scenes in real-time, even with millions of items.

3.4 Compilation

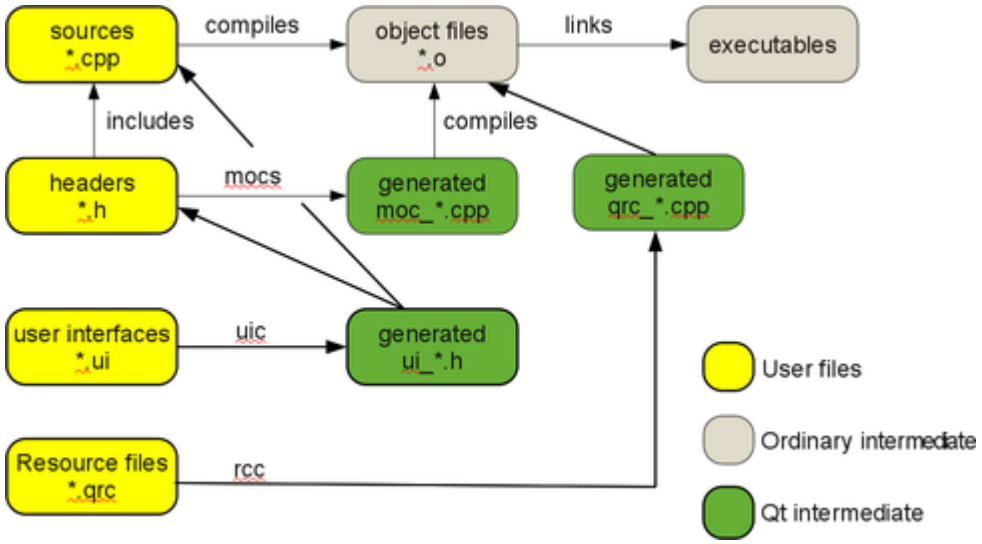

Figure 3.2: The compilation process of Qt software, from qt-project.org [33]

Compilation of a program that uses Qt framework depends on which language binding is used. Because Qt framework is implemented in native C++, I will concentrate on the compilation process of C++ language programs. The compiling of a program is automated in Qt, but the process is quite complicated behind the automation. The process can be seen in the Figure 3.2. Because Qt is a C++ framework

the developer can choose the compiler according to the platform and personal preference. For example, MinGW for Windows or GCC for *nix systems. After compiling the program the developer still has to link the object files and include the Qt platform files. In this section I will go through the tools that a developer uses and the Qt specific steps.

3.5 qmake

While Qt-programs can be compiled manually there are many required steps. That's why in Qt there is the **qmake** tool. **qmake** is a Qt version of the regular makefiles. With the help of **qmake** the developer only has to write the source code and**qmake** will take care of the rest. **qmake** will create a makefile according to the information in the project file. Project files are manually made files that include the instructions how to compile and what to include in the application. The instructions can be made to take into account the platform that the program is compiled in, so a program written with Qt can be compiled by the user on any platform. Project files can also contain instructions to the compiler including enabling debugging and optimizations.

qmake and project files are meant to help the development of Qt applications, but changes in the application can cause the developer to change the project files. The changes are minuscule but they need to be made. If the developer is using the Qt Creator IDE (Intelligent Development Enviroment), Qt Creator does the project files and changes to them automatically.

3.6 UIC - User Interface Compiler

In Qt there are two methods for building an UI: writing the code yourself or using an editor. The compilation of a self-written UI is easy, just compiling it, but when an UI made with an editor is compiled, it has to go through the **uic**. UI-editor makes an xml-file that has the definitions for the structure and items in the UI. This is then driven through the **uic** that makes the needed C++ files from the UI files.

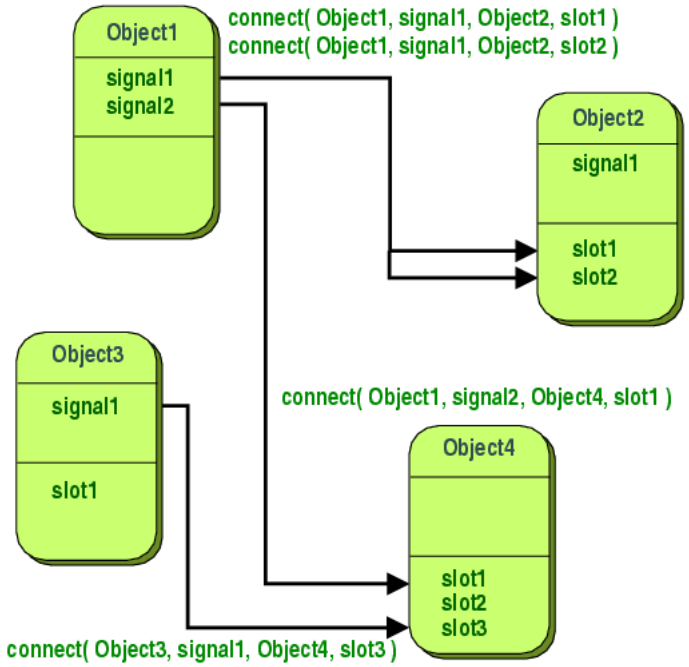

Figure 3.3: Signal & Slots from Qt-project [29]

3.7 Signals & Slots

Signals and slots is one of the key aspects of the Qt-framework, and they allow communication between objects as seen in Figure 3.3. For example, in GUI programming the developer wants a widget to be notified of changes to another widget when the user clicks a button. In other toolkits this is achieved via callbacks. Callback is a pointer to a function so if the function wants to be notified on changes elsewhere, it passes a function pointer to the wanted function. The wanted function then calls the function to be notified via the function pointer when something happens.

Signals can be emitted by an object when its internal state has changed and other objects would like to know about that. The restriction is that only a class that defines the signal, or a subclass, can emit the signal. When the signal is emitted the functions connected to it are executed immediately and the execution takes place outside the GUI event loop. The execution of code will continue after all of the functions in slots have finished. If several slots have been connected to one signal they will be called in the order they have been connected.

Slots are called when signal connected to them are emitted. Slots are normal C++ functions and can be called like functions are called. The only difference is that also

signals can connect to them. Slots have no restrictions on what can be connected to them. Slots can be also defined to be virtual. Virtual is a keyword of C++ related to inheritance. The implementation of a virtual function can be replaced by the derivided class.

The Signal & Slot model is slightly slower compared to callbacks because the emitting of a signal causes the following steps:

1. locating the connecting object
2. checking that receivers have not been destroyed during the emission
3. marshalling the parameters of the signal

The whole process is 10 non-virtual function calls, but the overhead is till less than any **new** or **delete** operation.

The S&S model is provided by the Meta-Object System (MOS) inside Qt. It also provides run-time type information and dynamic property system. The MOS is based on three things:

1. The QObject class provides base class for objects that can take advantage of the MOS.
2. The Q_OBJECT macro inside the private section of the class definition is used to enable meta-object features.
3. The Meta-Object Compiler (moc) supplies each QObject subclass with the necessary code to implement meta-object features.

Next I will go through the moc and it's purpose.

3.7.1 MOC - Meta-Object Compiler

MOC is the tool that handles the C++ extensions of Qt. **moc** analyzes the header files of the project and if it finds **Q_OBJECT** macro from it, it produces another C++ file that includes the meta-object code for the class. The generated file must be then compiled and linked with the original file. The meta-object code is required for the extra properties that Qt has when compared to standard C++. The calls to **moc** are usually done automaticly by the build system so it does not require any extra trouble from the developer. With the usage of **moc** and **uic** the Qt-specific properties can be compiled using a 'regular' C++ compiler.

3.8 Qt5

In this chapter, I will talk about what kind of changes the new Qt5 brings to Qt, and how the development paradigm will change.

Qt4 was published in June 28, 2005, and it is over 7 years old. Application development has changed much after that and new devices have emerged that has to be taken into account. For example, the smartphone revolution has lead to the rise of touchscreens in mobile phones.

Qt5 will change the paradigm of developing with Qt. Qt Quick will be the driving force in app development as it allows much more state-of-the-art interfaces to be designed and implemented. The C++ aspects will be hidden in the data layer doing the intensive tasks as the Qt Quick layer will take care of the interface logic. Qt5 allows OpenGL/OpengGL ES to be used in the GUI design.

Lars Knoll who is R&D (Research and Development) leader at Qt Software business unit wrote a blog post of the changes in the architecture of Qt that was done in transition to Qt5 [23]:

- Base all Qt ports on Qt Platform Abstraction layer (QPA).
- Re-architech the Qt graphics stack.
- Modularize the Qt repository structure.
- Separate QWidget functionality to own library.

Qt5 includes also at least the following new features [24]:

- *Qt Quick:*
  - See Section 3.9.2
- *Qt Qml*
  - New module containing the QML engine.
  - Performance improvements and enchancements to the language.
  - Some changes required to old QML items written in C++ to comply with the new SceneGraph.
- *Qt 3D*

  - Now belongs to the Essentials module.

- *Qt Webkit*
  - Qt Webkit essential module is now based on WebKit2. The C++ APIs have not changed.
  - Better performance and HTML5 compliance.
  - The module based on WebKit1 as in Qt 4.x is now called Qt WebKit Widget and available as an add-on.

- *Qt Core*
  - QStandardPaths class giving standard locations for files.
  - JSON parser and speed optimized binary format for JSON.
  - MIME type recognition.
  - New compile-time check of signal/slot connection syntax.
  - New Perl-compatible regular expression engine.
  - Many data structures have been rewritten and optimized for better performance.
  - C++11 support where it makes sense (but Qt continues to compile and work with C++98 compilers).

- *Qt Gui*
  - Support for top-level surfaces through the QWindow class.
  - Built-in OpenGL support.

- *Qt Network*
  - Support for DNS lookups.
  - QHttp and QFtp classes removed (they are available stand-alone for those that need them).

- *Qt Location*
  - Maps and geolocation-related classes formerly part of Qt Mobility are now in their own essential module.

- *Qt Widgets*
  - All former QWidget based classes in Qt Gui have been separated out into the new Qt Widgets library.
  - Ported over to the new Qt Platform Abstraction architecture.

The most essential additions in the light of this thesis are the support for JSON and the removal of QHttp. The QHttp has been replaced with QNetworkAccessManager which is used to handle the communications between internet and application. There has been many delays in the release of Qt5 Beta, and the first beta was delayed for almost 3 months. Qt5 Beta1 was released on August 31st, and Qt5 RC1 was released on December 6th.

3.9 QtQuick and QML

QtQuick is a collection of modules that helps the developers to develop modern fluid user interfaces. It was added to Qt after Nokia aquired Trolltech and they needed another way to create UIs. The desktop world differs from the mobile and the use cases are also different. QtQuick consists of a rich set of user interface elements, a declarative language for describing user interfaces (QML), and a language runtime. A collection of C++ APIs are used to connect the high level functions to classic Qt applications [17]. Next I will describe the declarative language and talk a bit about the differences of QtQuick 1.0/1.1 and QtQuick 2.0 that Qt5 brings to the table.

3.9.1 QML

QML or Qt Modeling Language is a high level declarative language. It was designed to describe the user interface of the application. It can define the look and application logic. JavaScript is used as the scripting language of QML so any implementation is done with it.

3.9.2 From QtQuick 1.0 to QtQuick 2.0

QtQuick 1.0 was intended for building applications for mobile devices and not so much for desktop. QtQuick 2.0 will bring the same development paradigm to the

desktop applications. QtQuick 2.0 includes also many other improvements and new features and I will now shortly review the most important ones.

- *New JS engine*: QtQuick 1.x used the JavaScriptCore engine to handle JS. QtQuick 2.0 changed that to Google's V8 engine that should bring better performance.
- *New Canvas Item*: A canvas item that is similar to the HTML5 Canvas.
- *OpenGL integration*: The graphics stack has been remade and the new implementation uses OpenGL 2.0. This should bring performance improvements compared to the QtQuick 1.x.

QtQuick 1.0 will be available as a separate library and a module.

3.9.3 QtCreator

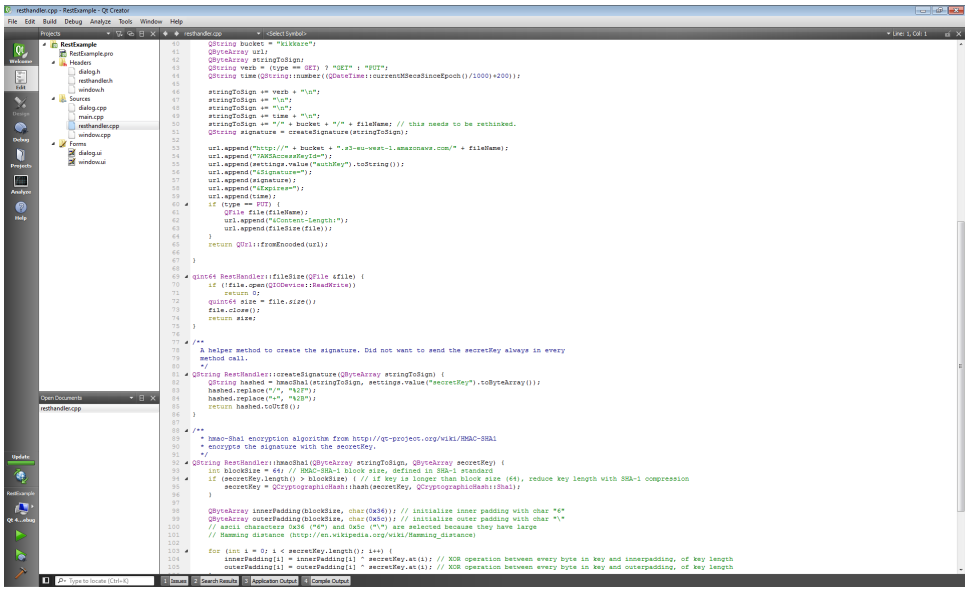

Figure 3.4: Qt Creator IDE

QtCreator (QtC) is a cross-platform IDE (Integrated Development Enviroment) that is provided with Qt. It has been tailored to ease the Qt application development. QtC has features that are almost standard in the IDE's of today including syntax highlighting, code completion, easy source refactoring, parenthesis matching, and a visual debugger. QtC also has an integrated UI designer and simulator for mobile devices. Languages supported in QtC are C++, JS, and QML.

In Figure 3.4 is basic structure of QtC is illustrated. On the left side are all of the different modes of the IDE (edit, design, debug, analyze etc..) and on the center is the code view. Qt SDK includes the examples that can be also found from the qt-project.org website. These samples can be viewed at the right side of the designer, allowing easy learning.

3.10 QDoc

Qt documentation presents QDoc with the following sentence [21]: "QDoc is a tool used by Qt Developers to generate documentation for software projects. It works by extracting qdoc comments from project source files and then formatting these comments as HTML pages or DITA XML documents, etc." QDoc comments always begin with /!* and end with */. QDoc tool only searches .cpp and .doc files from the project and comments placed in header files are disregarded. QDoc is mainly used in the automatic documentation creation for Qt framework, wider adoptation has not yet happened.

```
/*!
 \class QCloudConnection
 \brief Cloud connection Interface. Provides nesesary
        functions for basic operations in file storage
        that is located in the cloud.

  QCloudConnection is the interface for a cloud connection
  object. These methods are required and should be
  implemented as documented.
*/
```

Figure 3.5: Documentation in QDoc format

3.11 QTestLib

QTestLib is a unit testing framework created by Nokia [25], and it is meant for testing Qt's libraries and software that uses them. QTestLib has the following proper-

ties, as mentioned by QTestLib documentation [25]:

- *Lightweight*: QTestLib consists of about 6000 lines of code and 60 exported symbols.
- *Self-contained*: QTestLib requires only few symbols from the Qt Core library for non-GUI testing.
- *Rapid-testing*: QTestLib needs no special test-runners; no special registration for tests.
- *Data-driven testing*: A test can be executed multiple times with different test data.
- *Basic GUI testing*: QTestLib offers functionality for mouse and keyboard simulation.
- *Benchmarking*: QTestLib supports benchmarking and provides several measurement back-ends.
- *IDE friendly*: QTestLib outputs messages that can be interpreted by Visual Studio and KDevelop.
- *Thread-safety*: The error reporting is thread safe and atomic.
- *Type-safety*: Extensive use of templates prevent errors introduced by implicit type casting.
- *Easily extendable*: Custom types can easily be added to the test data and test output.

3.11.1 Usage

Using QTestLib is easy, all developer needs to do is to create a new class that is a subclass of QObject and add one or more private slots to it [25]. Private slots represent the test functions in the class. There is four specified functions that can be used to initialize and clean up the test structure:

- *initTestCase()*: is called before the first test function is called. If this fails no test functions will be ran.

```
class Test : public QObject {
  Q_OBJECT
private slots:
  void initTestCase() { qDebug("Ran before anything else"); }
  void firstTest() { int a = 1; int b = 1; QVERIFY(a == b); }
  void secondTest() { int a = 1; int b = 1; QCOMPARE(a, b); }
  void cleanupTestCase() { qDebug("Ran after everything else") }
};
```

Figure 3.6: Example test program using QTestLib

- *cleanupTestCase()*: is called after the last test function.
- *init()*: is called before every test function. If this fails the next test function will not be ran.
- *cleanup()*: is called after every test function.

The testing is done with macros that QTestLib provides. There are 15 different macros that provide different types of checks for the data. For example, QVERIFY(x) checks if something is true (QVERIFY(1==1)), and QCOMPARE(x,y) checks if the values x and y differ. QCOMPARE expects the values to be same and returns an error if they differ. The macros also provide QEXPECT_FAIL that marks the next QVERIFY or QCOMPARE to fail, thus not braking the testing structure. In Figure 3.6 an example test program that uses QTestLib is provided.

3.12 Example Qt program

Qt-project has an extensive archive of example programs that present how to do a certain kind of operation. They range from simple Ftp client that shows how QFtp is used to larger database examples that show how SQL and other relational databases can be used in Qt. I will now present the QFtp example program as it was an initial starting point to the QCloud API. I will also compare QFtp and QNetworkAccess-Manager.

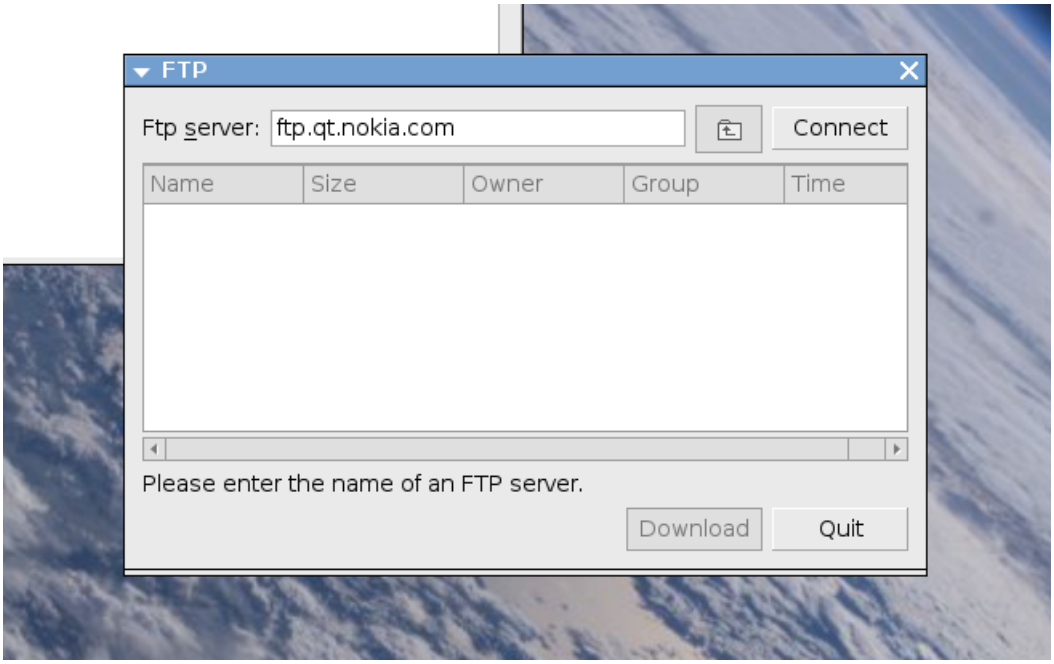

Figure 3.7: QFtp GUI

3.12.1 QFtp example

In Figure 3.7 one can see the basic GUI that provides needed functionality for the program. The user first supplies an address to which the program then connects. The connection is done via QFtp. QFtp has been part of Qt since Qt3, and it's usage is not recommended [22]. In the documentation the usage of QNetworkAccessManager (QNAM) is recommended for new programs as it provides simpler but more powerful API [22]. I will now present the parts that connect to the given server and how QFtp uses signals to give the results of the query. I will not go through the whole example as it is not nessesary. The example can be found from qt-project [12].

```
ftp = new QFtp(parent);
connect(ftp, SIGNAL(commandFinished(int, bool)), this,
        SLOT(ftpCommandFinished(int, bool)));
connect(ftp, SIGNAL(listInfo(QUrlInfo)), this,
        SLOT(addToList(QUrlInfo)));
connect(ftp, SIGNAL(dataTransferProgress(qint64,qint64)),
       this, SLOT(updateDataTransferProgress(qint64,qint64)));
```

```
QUrl url(ftpServerLineEdit->text());
ftp->connectToHost(url.host(), url.port(21));
```

In the listing above one can see the usage of QFtp. The first thing is creating new QFtp object. After that it's singals are connected to local slots that perform certain actions. QFtp's replacement QNAM is also used like this. After that new QUrl is made from the address the user has inputted to LineEdit, and after that the connection is made using the URL. Again this same type of syntax is used with QNAM. As Qt uses signals to indicate that the operation has finished, it is easy to make asynchronious calls to Internet.

```
QString fileName = fileList->currentItem()->text(0);
file = new QFile(fileName);
...
ftp->get(fileList->currentItem()->text(0), file);
```

The get request is made in the listing above, and when the download has finished QFtp emits ftpCommandFinished(int, bool)-signal.The structure of first creating the connection, connecting the needed signals, doing requests, and finally getting the file can be used with QNAM. I will present a small example of QNAM's usage as it will show the similar nature of the two.

```
QUrl url("http://sisuguild.fi/~jage/examplefile.zip");
QNetworkAccessManager *qnam = new QNetworkAccessManager(this);
QNetworkReply *reply;
connect(reply, SIGNAL(finished), this, SLOT(requestFinished()));
reply = qnam->get(url);
```

In the listing above zip file is downloaded from the given address. QNAM returns a pointer to the reply object that contains the downloaded file. When the download has finished the reply object emits finished()-signal and after that the data can be read from it.

In this chapter I have explained the history of Qt framework, what it contains, what is the special features about it, how Qt5 will change the development paradigm,

and different tools that help the developer. The last part was an example that shows how network connections are created with QFtp and QNAM, and how they inform the developer when the action has been completed.

4 Software development for cloud

Developing software to cloud is similar to developing programs to 'regular' computers. The API's can differ from SP to SP so there is a need to find a best suited offering for the program before deciding where to deploy it. In this chapter I will give a couple of examples how the Cloud can help the regular programs. After that I will explain how Qt can be integrated to cloud services. The different levels of the software stack are placed on different levels and I will present use cases of each of the possiblities. After this the reader should have a knowledge of how software development for cloud is done and how the integration of Qt to Cloud can be achieved.

4.1 Differences in architectural views

The real inspection of the Cloud requires an inspection of how the Cloud differs from the traditional local software development. Software development has been regularly modeled using the three-tier model (Data, Logic, View). In this model the lowest level is the data-level where data is stored. The separation of data from the software logic helps to keep the data neutral since it is not related to the upper levels of the model. The separation also brings better scalability and performance. In the Cloud this level can be thought to be on the border of PaaS and IaaS.

The second level of the model is logic. This level refers to the application logic and controls the program execution. This level is the controller of everything the program does. With the separation of data, this level can exist without needing to know where the data is located.

The third and the highest level is the view. View is the user interface of the program. It does not have any application logic in it but it communicates with the logic-layer to provide the user with the wanted features.

With the separation of data, logic, and view gives the program an elasticisty, scalability, and performance that is hard to achieve with all of the aspects being in the same layer. In the cloud, developer can create a new data layer object for every user and place them in different instances. This way the user will see an

improvement in the performance. With the Cloud the separation is clearer because the Cloud stack is divided according to the three-tier-model.

The implementations of API's differ from SP to SP. This can be a hindering fact to the developer as he/she has to change a portion of the codebase as the program is transferred to another SP. Armstrong et al. [40] mentioned the lack of standardized APIs as one of the risks of the Cloud. To alleviate the different implementations in this thesis I will concentrate on the REST (Representational state transfer). REST allows the ability to use the API's via standardized protocol making an implementation of a general Cloud API possible. The REST-implementations are similar for all of the SP's, but they do have some differences.

When developer decides to incorporate elements from the Cloud to the application, there are some questions that the developer to consider: what level of integration is needed to provide the wanted features, and how does the application work if the Cloud is not accessible. I will now analyze these two questions a bit.

The three-tier model presented earlier provides a nice frame of reference to inspect the different levels of integration between the software and the Cloud. I present the following model to represent the cloud integration:

- *Integrating the data layer*: The developer decides to use the storage(table, blob), or computational services of the SPs.
- *Integrating the logic layer*: The developer decides to use the two lowest layers of the cloud stack (IaaS, PaaS) and implementing the view on the application.
- *Integrating the view layer*: The developer deploys the application to the Cloud and it is used via browsers. This includes all of the layers of the cloud stack (SaaS)

In the first scenario the developer uses only the storage aspects of the SP's. Developer can take the stored application data to the Cloud, thus providing accessibility to it almost universally. If the data is not also kept locally, there is the possibility that the application can not access the data at a given point and time. This could lead to malfunctions, of course depending on the type of application. This is the stepping stone of the Cloud as it requires access to the Internet, and it is still not available everywhere. The other data layer service would be to integrate calculations done in the Cloud to an application. This can help if the used device is starved on computational resources, or the calculation jobs would be really intensive. Of

course the calculation can not be used in a program that requires the calculations done instantly as the transfer of data will take some amount of time. This would be the developers job to decide if the gains of using the Cloud are larger compared to the transfer times.

In the second scenario the developer uses the two lowest levels that the SP's provide. In this scenario the developer only has to do a 'stupid terminal' on the device that then connects to the cloud to use the software logic and data. This could be the most versatile approach as the developer can choose via design choices that something could be done in the cloud and something could be done in the device. Size of the application would also be small and the required resources on the device are small, almost the only crucial thing is somekind of a way to access the web.

The last scenario is that the developer uses the SaaS level. This means that the whole software is located in the Cloud and it would not be necessary to install any extra applications on the device.

4.2 Development using Qt

In this section I will inspect the Qt framework and how it could be integrated with different cloud services and different SP's. The inspection will be on the three-tier model and how these three tiers could be integrated in the Cloud.

Like I have previously mentioned, Qt is a platform independent framework, and it has bindings for almost all of the major operating systems. The ability to code once and run everywhere is a major bonus in Qt. In the second chapter I introduced cloud computing and the different service providers, but I did not address how the Cloud paradigm changes the developers view. I will now go through the chosen service providers and ponder, how the Qt applications could be deployed there. There are two different scenarios that can be thought to be realistic: placing the data layer to the cloud, or placing data and logic layers to the cloud.

4.2.1 Amazon

The best possiblity to integrate Qt applications is to make them in the Cloud. Amazon provides IaaS-services so that the developer can start an instance in the Cloud and then place the developed program there. The developer can choose from Linux and Windows operating systems to which Qt provides support. As Amazon pro-

vides fully fledged operating systems in the cloud there is a possiblity to do the software development in there. In the case of QtCreator this would require graphical user interface, but, for example in Linux, the command line is enough (qmake etc...). It is possible to start a window manager in Linux, and Windows Server provides it out-of-the-box.

The integration of data layer to the cloud is the simplest way to integrate the Cloud to software. Application would still be run on the users device, but the stored data would be in the Cloud. Amazon provides blob and table storage to incorporate in the programs. The connection to the cloud can be done via RESTful interfaces that Amazon provides. Amazon provides also computation in the cloud and it also could be integrated in the software.

Amazon instances could be used to host the whole program and then provide a user interface on a website. In this scenario everything should be in the cloud. This possibility is only available on the Amazon services at the moment, as Microsoft's virtualization services are in the beta stage.

The open source implementations of private clouds are similar to Amazon's APIs so there is a way to move from the corporate cloud to a private cloud if the need arises [57, 59].

4.2.2 Azure

Like I mentioned in the Azure section of the Cloud computing chapter, Azure has three different compute roles: web-, backend-, and, in the beta stage, legacy. Web roles are ran on the IIS7 (Internet Information Service) web server. Integration between Qt software and this role is not feasible because the web technology nature of the service. As QML and JS is used on Qt applications the integration of the user interface can someday be possible, but at this moment its not.

Azure provides data storage that is usable via RESTful interfaces. Thus the scenario of integrating the data layer is possible in Azure. The use cases are similar to Amazon.

Azure has a rentable VM service, but it is still in the beta stage. The service offers a customizable Windows Server 2008 instance in the Cloud. Microsoft requires that the developer keeps the VM up to date so there is some extra work required from time to time. As the VM offers a full operating system, there is the possibility of running Qt applications there. However there are some restrictions to the VM role. Azure requires that the application running in the VM complies to the stateless

requirement shared by all of the roles. For example, SharePoint, SQL Server, Small Business Server, and Terminal Server are applications that can not be used in the VM role [34]. The requirement can be complied via using Azure's own services in the application.

4.2.3 Google App Engine

GAE is the most strict of the chosen SPs and the integration of Qt applications can prove to be too hard. Applications running in the GAE are made with the GAE SDK and the language support of the SDK is small. Google provides support for Java, Go, and Python. A Qt application could use the GAE as a backend, but it should be written in the aforementioned languages. The extra work makes this option worse than the others. Google does provide storage services that could be used from a Qt application, but, in this thesis, I won't take this into account. This decision comes from the fact that further integration is at the moment not possible.

4.3 REST API comparison between Azure and Amazon

In earlier sections I presented the general properties of each SP, and now I will compare them in the context of this work. The integration of different SP's can be challenging because almost all of them have differences in their API implementations. For example, App Engine is made so that one can not use it in the same manner as the Azure and Amazon to save files. File storage is the first feature of the QCloud API so this leads to the fact that App Engine will be left out of the later comparisons. Both of the remaining SPs offer REST interfaces in conjuction with their own propietary interfaces so when designing a general API that would work on both, REST is the way to go.

For the sake of this example I presume that the storage account/account name is **exampleaccount** and there exists the following file structure in the cloud:

- Bucket/Container: example

  Blob: examplefile.txt

In Azure all of the containers are within a certain storage account that has its own namespace and user can have many active storage accounts at the same time. In Amazon the user has an account that has the buckets inside of it and the namespace

is shared with all of the users of S3. This can create interesting situations as the most used bucket names have already been taken. Next I will introduce the structures of the signature strings.

```
StringToSign = VERB + "\n" +
Content-Encoding + "\n"
Content-Language + "\n"
Content-Length + "\n"
Content-MD5 + "\n" +
Content-Type + "\n" +
Date + "\n" +
If-Modified-Since + "\n"
If-Match + "\n"
If-None-Match + "\n"
If-Unmodified-Since + "\n"
Range + "\n"
CanonicalizedHeaders +
CanonicalizedResource;
```

(a) Azure

```
StringToSign = HTTP-Verb + "\n" +
Content-MD5 + "\n" +
Content-Type + "\n" +
Date + "\n" +
CanonicalizedAmzHeaders +
CanonicalizedResource;
```

(b) Amazon

Figure 4.1: Strings to sign

In Figure 4.1 one can see the differences of the strings that are used to compute the signature to the request. These are created and after that hashed with HMAC (Hash-based message authentication code) and the chosen SHA-algorithm (1 or 256). In the QCloud API I used SHA1 for Amazon and SHA256 for Azure. The difference in hash algorithms is explained later in the Section 5.2.

As the created signatures differ also the required headers in the requests differ. Next I will present the structure of the requests.

Amazon

- *Request URL*:

  `http://example.s3.amazonaws.com/examplefile.txt`

- *Required headers*:

AWSAccessKeyID: The unique access key of the user.

Signature: The result of hashing the string to sign with HMAC-SHA1.

Expires: Seconds from Epoch added the amount of time one wants the request to be valid.

Azure

- *Request URL*:

```
http://exampleaccount.blob.core.windows.net/
                                example/examplefile.txt
```

- *Required headers*:

  Authorization: Includes the used authentication (SharedKey or SharedKeyLite), storage account name and the HMAC-SHA256 hash.

  x-ms-date: RFC1123 formatted date string.

  x-ms-version: The API version used.

I will explain the differences from the Qt point of view in Section 5.2, as the minute differences brought a lot of troubles in the creation of API. Azure also requires that one specifies the wanted operation in the request. If one wants to list all of the files in a certain container the following operation specifier is added to the URL: ?comp=list. In Amazon this would be done with GET operation on the bucket, and it would need no further specification.

PUT-requests differ from the GET- and HEAD-operations in both services. In Amazon PUT-operation adds Content-Type-header that is the MIME-type of the sent file and Content-Length which is the length of the message without the headers, as specified in RFC 2616. Azure, on the other hand, needs the same Content-Length, but it is needed also in the signature string. Also Azure needs x-ms-blob-type-header which specifies if the file is page blob or block blob. The differences of these two blob types are as follows [32]:

- *Page blob*: Page blobs are a collection of 512-byte pages optimized for random read and write operations. When creating page blob one needs to specify the maximum size of the page and also initialize it. When writing to a page blob the user can write over the old page or add to the page by continuing from the end. The maximum size of a page blob is 1TB.

- *Block blob*: Block blobs are comprised of blocks, each of which is identified by a block-ID. Blocks can be different sizes but the maximum size is 4MB. The maximum total size for a block blob is 200GB and it cannot contain no more than 50000 blocks. Files larger than 64MB cannot be uploaded in a single PUT-operation. For larger files the developer needs to chop the file in to pieces and send them one by one. If the file is sent in a one go, the file is uploaded and the change is committed to Azure. If the file is uploaded in many parts, the developer needs to first upload all of the part using the same operation, and after that call the Put Block List-request. In the request the developer specifies the changed blobs and thus making the commit to the service. If the developer does not commit the blobs, the service discards the uncommitted ones in one week.

Amazons service differs from Azure as it does not have two different types of blobs, but it has only one type: object. The size of object an can range from 1 byte to 5TB. The user can upload a object in a one go or in many different parts. When dividing the upload Amazon requires user to take the following steps [20]:

- *Initiation*: The developer initiates the upload, and Amazon returns a unique multipart upload id which is required in each of the requests.
- *Part upload*: The developer specifies the upload id of the part and also specifies a part number. Part number can be any number between 1 and 10000 and it specifies the position of part in the whole object. If the part number is already used the new upload overrides the old part.
- *Completion*: Complete multipart upload request contains the upload id, object metadata if it was not provided in the initiation, the part numbers uploaded and the corresponding E-Tags of the objects. After the request is sent Amazon constructs the final object from the parts and the parts cease to exist.

4.3.1 Comparison of REST API's using example

Next I will compare the REST-implementations of Azure and Amazon from the storage point of view. I will compare the required headers of the requests, the differences in the requests, and the authentication strings. I will elaborate the differences using the following example:

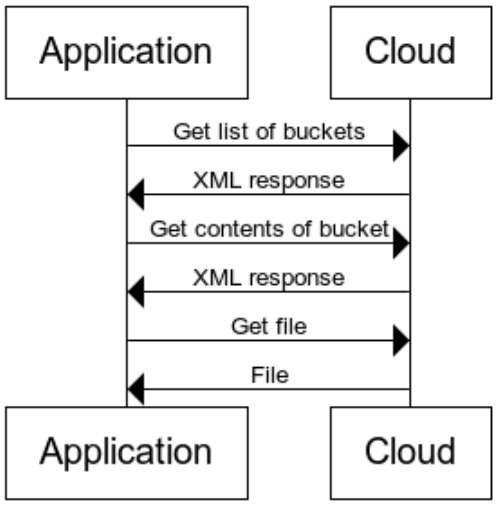

Figure 4.2: Sequence and responses

- Getting the list of buckets/containers,
- getting the file listing of a bucket/container,
- getting a file from a certain bucket/container.

This is decipted in Figure 4.2. From now on I will refer both containers and buckets as CloudDir. This is the name that they are called in the QCloud API. Getting the list of containers and buckets is similar in both SP's. The basic structure is a GET-request on the root of the account.

- *Amazon request URL*: http://s3.amazonaws.com/
- *Azure request URL*: http://exampleaccount.blob.core.windows.net /?comp=list

In both services this will return an XML-file that contains information of the clouddirs in the account for example: creation date, size, owner, name and many other. The structure of the XML responses are quite similar and the names of the clouddirs are contained in the Name-field. The next step is to get the content listing of a certain CloudDir called example:

- *Amazon request URL*: http://example.s3.amazonaws.com/

- *Azure request URL*: http://exampleaccount.blob.core.windows.net /example?comp=list&restype=container

These requests return the contents of CloudDir example. Again the response is XML formatted and contain many more fields than just the names of the contained files. After this the, last step is to get a certain file from the CloudDir. In this example the file is named examplefile.txt.

- *Amazon request URL*: http://example.s3.amazonaws.com/examplefile.txt
- *Azure request URL*: http://exampleaccount.blob.core.windows.net /example/examplefile.txt

The return message from SP's contain the contents of the examplefile.txt. In every step, except getting the listing of CloudDirs, if the requeted blob/CloudDir does not exist, the service returns an error message. As this example shows, the basic structure of the REST API's is quite similar, only the requests differ in some sense.

In this chapter I explained the differences and the similarities of the REST API implementations of the chosen SP's. I also explained how the services differ from each other in the sense of the storage aspect, and how cloud storage could enhance programs.

5 QCloud API

The end result of this thesis is QCloud API which brings the Cloud to the developer. The driving idea behind QCloud was that the developer would have an easy way to add cloud storage to programs with minimal changes and without many new dependencies. In this chapter I will first explain generally on API design and what qualifies as a good API. After that I will present the QCloud API and the design choices behind it. Last I will evaluate the QCloud API according the good API design priciples I presented.

5.1 On API design

In this Section I will present four different views on what constitutes a good API. After that I will try to find the similarities in these four views. After this section the reader should have a good understanding on the basic priciples of API design and what makes a API good.

In the book "API design for C++", Martin Reddy describes the design of a good API as follows [58] : "An API is written to solve a particular problem of perform a specific task. So, first and foremost, the API should provide a coherent solution for that problem and should be formulated in such a way that models the actual domain of the problem." This is a strict view of thinking about API's. It constricts an API to do one thing and do the one thing without difficulties and side effects. Reddy also talks about that the API should use the same kind of terms as the domain where the API would deployed, as it would help the users to fantom the concepts easier.

Michi Henning gives another good explanation in his article "API Design Matters" [51]: "Good APIs are joy to use. They work without friction and almost disappear from sight: the right call for a particular job is available at just the right time, can be found and memorized easily, is well documented, has an interface that is intuitive to use, and deals correctly with boundary conditions." Even though Hennings definition is broader, the same underlying issues are present here. Both authors have an idea that the API should have just the needed functions and tools for a certain job, nothing more. Henning's view mentions intuitive use and cor-

rect boundary condition handling. These can be thought to be the logical match to Reddy's coherent solution.

In his presentation: "How to design a good API and why it matters", Joshua Bloch [43] presents the following qualifications for a good API: easy to learn, easy to use even without documentation, hard to misuse, easy to read and maintain code that uses it, sufficiently powerful to satisfy requirements, easy to extend, and appropriate to the audience. There are again many similarities in Bloch's definition and the two earlier ones. All of them mention that the API's should be easy to use to a level where one can learn the API just by using it from the IDE. Also all of them mention that an API should just do one thing well and not include useless methods. Bloch and Henning mention documentation that Reddy does not. Documentation is important as it is the place where the developers learn how to use the API and if the documentation does not tell them how to use the API they wont use it.

From these three one can extract the following rules for a good API: it should do one thing and one thing well, it should be documented so that one could learn to use the API just by reading the documentation, it should be tested well, and it should be easy to use.

Qt has it's own API design instructions [46] that I will now go through. Qt's API design guidelines mention six properties of a good API: be minimal, be complete, have clear and simple semantics, be intuitive, be easy to memorize, lead to writeable code. As one can see from the previously mentioned definitions, Qt API's guidelines have the same principles behind them. The guideline has also other rules for things to avoid. These include:

- *The Convenience trap*: Less code is not always better. Code can be written once, but it is read many times after that.
- *The Boolean Parameter trap*: Boolean values can lead to unreadable code.
- *Static polymorphishm*: Similar classes should have the same kind of API and inheritance should be used when possible.
- *The Art of Naming*: The naming should reflect the conventions of the problems domain and naming can make or brake an API.
- *Pointers or References*: In Qt pointers are preferred, because it can make the user code more readable.

These four different views on API design are quite similar and I will evaluate the QCloud API with these in mind in Section 5.5.

5.2 Implementation

In this Section I will present the construct QCloud API. First I will explain what services could be integrated to the API and after that I will explain what services I chose to include. Then I will explain why I chose to build the API the way I did. I will go through the basic structure, explain the functions and present the difficulties I had with certain parts. After this chapter the reader should have a good understanding what QCloud API is, how is it used and what it can give to a software developer.

The design of the API started with inspection of the QFtp example that I presented earlier in Section 3.12. It provided a nice logical match to the operation of putting and getting files from the Cloud, and the only real difference would be that the communication would be little more complicated. In the beginning I also looked at the different database services in the Cloud and how they could be integrated to the API. They were left out of the first prototypes, as they differ from file storage in many ways. Earlier in the Section 4.1 I presented the three different levels of Cloud integration. The API would be placed on the lowest levels, as it uses the Cloud only as a storage device.

As the development of the QCloud API began the first step was to compare the different REST API implementations of Amazon and Azure. As I have previously mentioned the basic structure is almost same, but there are couple of differences: Amazon accepts the signature to be done with HMAC-SHA1 and HMAC-SHA256, but Azure accepts only SHA-256. As Qt did not provide SHA-256 implementation at that moment, I decided to start the implementation process with Amazon.

Qt provided QNetworkAccessManager (QNAM) that contained the needed functionality to send GET-, PUT-, and HEAD-requests to the service. QNAM is a asynchronious, and thus the responses from the service are not usable until the response object, QNetworkReply, emits a signal finished. This requires a certain type of structure to the program as the parsing and overall usage of the response can only begin after the response is finished.

The development of the API continued with an example program demostrating the possibilities that the API would offer. The first draft of the implementation included getting the list of buckets, and their contents. The draft revolved around a

class named QRestHelper, that took care of everything. As the good design principles require that a class should have minimal responsibilities, QRestHelper would next be sliced in to pieces. QRestHelper contained the basic structure of the encoding methods that would later take care of creating the requests and signatures. As the development progressed, first alpha versions of Qt5 began to surface. Qt5 contained SHA256-algorithm that was required to start working on Azure. I ported the algorithm from Qt5 to Qt 4.8.3 that I used for the development. This allowed me to really start looking at the differences between the two SP's.

The most striking difference was that Azure required more headers in every request, and the operation had to be included in every request. Amazon's API did not require a operation as the request itself contained the information needed to differentiate the requests. Signature strings were constructed in a similar fashion from the headers of the requests. This led to the current iteration of the API structure.

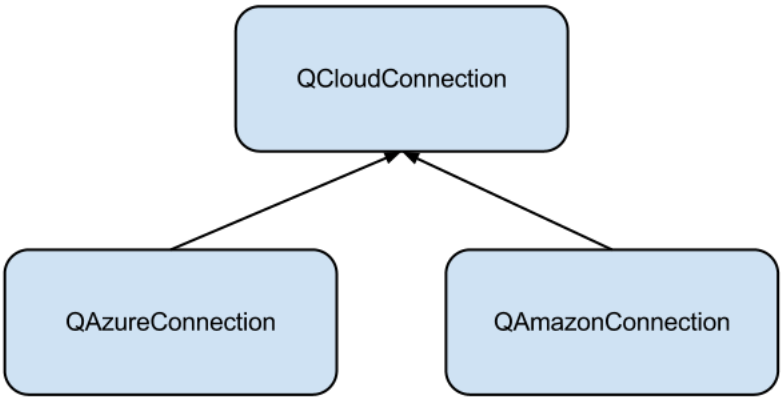

Figure 5.1: QCloudConnection and it's subclasses

In Figure 5.1 the inheritance from interface QCloudConnection is shown. QAmazon- and QAzureConnection implement the interface and thus provide the same functions for the developer. I will go through the functions in the next Section. As the basic operations for both services did not differ too much this approach could be used. The defining use-case of the QCloudAPI would be handling the data transfer to and from the cloud.

During the development I realized that there should be a corresponding class for a item that would be located in the Cloud. Using the regular QFile and QDir did not

seem possible as the items had the following three different states:

1. *Item was purely local*: The item was created on the computer and there weren't a corresponding item in the Cloud.
2. *Item was purely in the Cloud*: The item was in the Cloud, but a local copy did not exist.
3. *Item existed in both*: The item existed in the Cloud and locally.

I created a container class called interface called QCloudItem that would help with handling of these states.

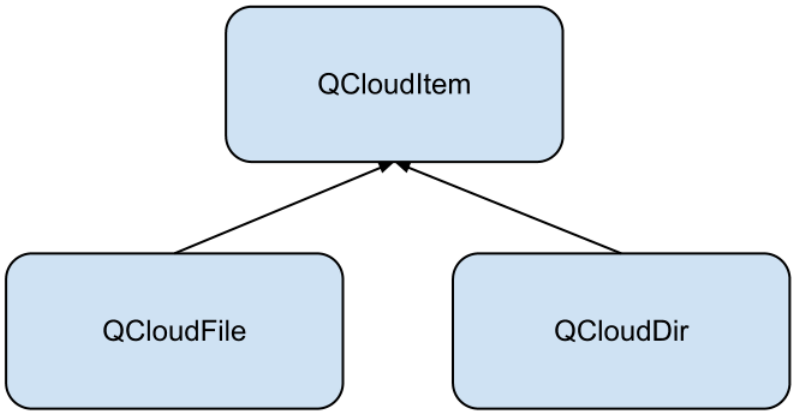

Figure 5.2: QCloudItem inheritance

In Figure 5.2 the inheritance of QCloudItem is shown. QCloudFile and QCloudDir are wrapper classes for QFile And QDir that also contain information on if they are local or just in the Cloud. They implement the QCloudItem interface that only has one function, isLocal(), that returns a boolean value containing the information that where the file was created from.

There is also three different types of response-classes that are used with the asynchronous methods (asyncGetCloudDirs, asyncGetCloudDirContents, and asyncGetCloudFile): QCloudFileResponse, QCloudResponse, and QCloudListResponse. QCloudListResponse is returned when the operation is a listing, for example, when getting the list of clouddirs or when getting the contents of a clouddir. The QCloudFileResponse is returned when the operation deals with a file, for example, getting

a cloudfile from the Cloud. QCloudResponse is returned when the return value of the operation can be considered to be a boolean, for example, putting a cloudfile to the Cloud. I will explain the need for response-classes later in Section 5.5.

5.3 QCloud API

In this Section I will present the API and it's structure. First I will present the structure of the QCloudConnection-interface, and after that I will go through an example case of usage. After that I will explain the difficulties that were encountered in the construction and last I will reflect on the construction and critically review the API.

In Appendix A you can see the structure of QCloudConnection. There are total of 20 functions and four different signals that are required to provide the needed functionality. The design choice of creating an interface that then would be inherited was made because the overall structure of the calls are so similar. This is because the QCloudConnection is basicly a program that translates the users requests to REST requests. As I have previously mentioned the REST implementations are similar. For example getting a certain file from a certain bucket would have the following operations:

- *Create new connection*: Create new connection object for the wanted service provider
- *Get list of buckets*: Get the list of buckets with QCloudConnection::getCloudDirs()
- *Get contents of bucket*: Get the contents of certain bucket with QCloudConnection::getCloudDirContents(QString bucketName)
- *Get the file*: Get the file with get(QString bucketName, QString fileName)

The same structure works for both services and the only real difference to the developer is in the creation of the connection. The scenario presented above happens when the location, in which bucket the file recides, is not know. If the developer knows where the files is, he/she can call get(bucketName, fileName) after the creation of the CloudConnection. The biggest differences in the implementation are in the encode(const Request &r)-functions. Encode function creates the signature strings that the services uses to autenticate the request. The flow of operations inside the encode function is similar in both services:

- *Create stringToSign*: StringToSign contains the structure of the request.
- *Create signature*: Signature is created using HMAC-SHA1/256 where the stringToSign is the message and the secret key of the user is the key.
- *Create request*: The request URL is created inside a QUrl, and it is placed inside QNetworkRequest.
- *Return request*: Return the created QNetworkRequest.

As the encode functions are the places where the real differences can be found, I have included them in Appendices. Amazon's implementation can be found from Appendix D, and Azure's from Appendix E. Otherwise the request creation is done with the struct Request. The wanted operations are added to the struct and encode functions make the QNetworkRequest's from them. The signature is created by HmacSHA-class, that implements the HMAC hashing algorithm. The class contains a enum called HmacSHAType that has two values: SHA1, and SHA256. The only function in the class is hash that takes three parameters: HmacSHAType, the stringToSign, and the users secret key. With these three parameters the class creates the hash and returns it.

There are minute differences in the request creation between the services: Amazon does not recognize requests if just the request URL is place inside QUrl and the needed headers are then added to QNetworkRequest. This is the way that Azure requires the request to be formed. Amazon requires the headers to be placed in the QUrl object and that then placed in the QNetworkRequest.

5.3.1 Testing

QCloud API is tested using the QTest-unit test library that I presented in Section 3.11. The tests test every function both in Amazon and in Azure. The functions are tested for a positive result (the operation was a success), and for a negative result (the operation failed). The tests presume that there is a certain file structure present in the Cloud. The creation of the structure is not automated at the moment, and it really should be made so that when the tests are ran, initialization creates the file structure to the Cloud. As all of the operations require valid credentials, the tests included in the project cannot be ran before inserting one's own credentials. Next I will present the structure of tests and what functions they test.

There are a total of 27 tests: 5 for the QCloudFile and QCloudDir, 10 for QAmazonConnection, and 10 for QAzureConnection. The tested functions in the classes inherting QCloudConnections are:

- *getCloudDirs()*: Positive and negative result.
- *getCloudDirContents(QString cloudDir)*: Positive and negative result.
- *get(QString bucket, QString file)*: Positive and negative result
- *asyncGetCloudDirs()*: Positive result.
- *asyncGetCloudDirContents(QString cloudDir)*: Positive result.
- *asyncGetCloudFile(QString bucket, QString file)*: Positive result.

The lack of negative case testing for the async-functions, is based on the fact that the response-objects contain the QNetworkReply pointer, and thus the errors can be seen from it. With these basic tests the functionality of the API can be proven. Next I will present couple of example tests both for the positive and negative results.

```
void QCloudTest::testAzureGetCloudDirs() {
   QCloudConnection *conn = new QAzureConnection("","","");
   QList<QString> buckets = conn->getCloudDir();
   QCOMPARE(buckets.at(0), QString("test"));
}

void QCloudTest::testAmazonGetCloudDirs() {
   QCloudConnection *conn = new QAmazonConnection("","","");
   QList<QString> buckets = conn->getCloudDir();
   QCOMPARE(buckets.at(0), QString("test"));
}
```

Above you can see the basic structure of a test that gets the listing of buckets in the Cloud. The test presumes that the first bucket in the list is named test. The test are almost indentical, and the only difference is the creation of the QCloudConnection-object. Both of the tests test for the positive case of the synchronous getCloudDir()-methods. Next I will present how I tested the negative results.

```
void QCloudTest::testAzureGetCloudDirsFail() {
    QCloudConnection *conn = new QAzureConnection("","","");
    QList<QString> buckets = conn->getCloudDir();
    QCOMPARE(0, buckets.size());
}
```

The test above shows how I tested the negative case usually. There is one kind of error that can happen in the listing of clouddirs: the credentials are false. In the test I test with wrong credentials and expect conn to return an empty list. I have tested some functions using the QSignalSpy-class. QSignalSpy takes two parameters, the object that you want to listen, and the signal you are listening for. As QCloudConnection's emit cloudError()-signal when the operation was not a success, testing can also be done this way. Next I will present the way I tested the asynchronous functions.

```
void QCloudTest::testAsyncAzureGetCloudDirContents() {
    QCloudConnection *conn = new QAzureConnection("","","");
    QCloudListResponse *resp = conn->asyncGetCloudDirs();
    QEventLoop l;
    connect(resp, SIGNAL(finished()), &l, SLOT(quit()));
    l.exec();
    QCOMPARE(resp->getParsed().at(0), QString("test));
    resp->deleteLater();
}
```

Above one can see how I tested the asynchronous functions. The tests differ from the synchronous functions by the creation of a QEventLoop. There is a event loop also in the synchronous functions, but it is 'hidden' in the sendGet(), sendPut(), and sendHead()-functions. I connect the QEventLoop's slot quit() to the signal finished() of the QCloudListResponse, that is emitted when the operation has finished. QCloudListResponse contains a parser that parses the XML-message received from the Cloud to a list of QString's. This test again presumes that there is a certain structure already present in the Cloud.

With the existing tests QCloud API is tested on a minimal level, as the error handling that would be needed for input checking is totally missing. There is still a lot to do in this part. The greatest defect is still that there is no initialization in the test project. This is the first thing that should be done, as otherwise the developers

need to make the same file structure to their account before running the tests. There should also be a test to cover the HmacSHA-class, but as the API is working, the hash algorithm also works. All of the tests can be found from YouSource [31].

5.3.2 Documentation

The documentation for QCloud API has been done with QDoc tool that I presented in Section 3.10. Every function, signal, and slot is documented and just by reading the documentation the usage of the API should be clear. Documenting the API was not easy as I was not familiar with the QDoc tool. The documentation for the functions came easily, but when trying to use QDoc, a wall rose. Configuring QDoc was not easy and there were almost none ready made configurations for my case. I first used Doxygen [45] (a documentation system for C++, C, Java, and more), but as the project is intended to be integrated to Qt, the change to QDoc was necessary. Next I will present an example of the documentation from QCloud API.

```
/*!
  \class QAzureConnection
  \brief Implementation of the interface QCloudConnection for
         Azure.

  Constructor to create new QAzureConnections. This contains
  three parameters and all should be in the right format. The
  first parameter (\a url) is QByteArray containing the url of
  the service i.e. "kikkare.blob.core.windows.net" where kikkare
  is the storage accounts name and also the next parameter
  \a storageAccountName. The last parameter \a storageKey is
  the authentication key that one can get from Azure. It is
  important to enter the key as it is in the web, as the
  connection presumes that it is Base64-encoded.

  \sa QCloudConnection
*/
```

Above is the documentation of QAzureConnection's constructor. The important information about the parameters is explained and one should be able to understand how it works from the comments. The project also contains a readme-file that has all

of the different differences documented. The readme can be found from Appendix C.

5.3.3 Summary

The whole project contains the following parts: two examples, documentation, tests, and the QCloud API. QCloud API has automated unit tests made with QTestLib and they confirm that the API works as intented. At the moment they only test the API with positive tests: they expect that the inputs are always right. The project has total of 3000 lines of code. All in all during the project, total line count is closer to 6000, as there are many prototypes and prior example programs. The largest files are the QAzureConnection and QAmazonConnection, as they contain almost all of the logic needed to communicate with the services. I will now go through an example program that uses QCloud API.

5.4 Example application

QCloudTransfer (QCT) is a simple program that takes input from the user in the command line. The commands are what the wanted operation is (put, get) and if the operation is put then the file that is wanted to transfer to the Cloud. The source code for the program can be found from Appendix B. The first thing done is checking the input parameters and making decisions based on them. If the param is get the execution continues to getFile(), if the parameter is put then the putFile(QString name) is called. The parameter name is second input parameter to the program.

In getFile() the program first creates a new QAmazonConnection object using the credentials read from the auth.txt file. After that it gets the list of buckets in the users account and displays them. Then the user picks one bucket from the list and the program gets it's contents. Then the program prints the contents to the user and the chosen file is downloaded to the computer.

In putFile() the program first creates a new QAmazonConnection with the credentials. After that it gets the list of buckets in the users account and displays them. Then the user chooses in which bucket he/she wants the file to be uploaded. Then the file is sent to the Cloud. The example program is trivial in nature but showcases the usage of QCloud API nicely.

QCloudExplorer which is an GUI-program that resembles a FTP-application can

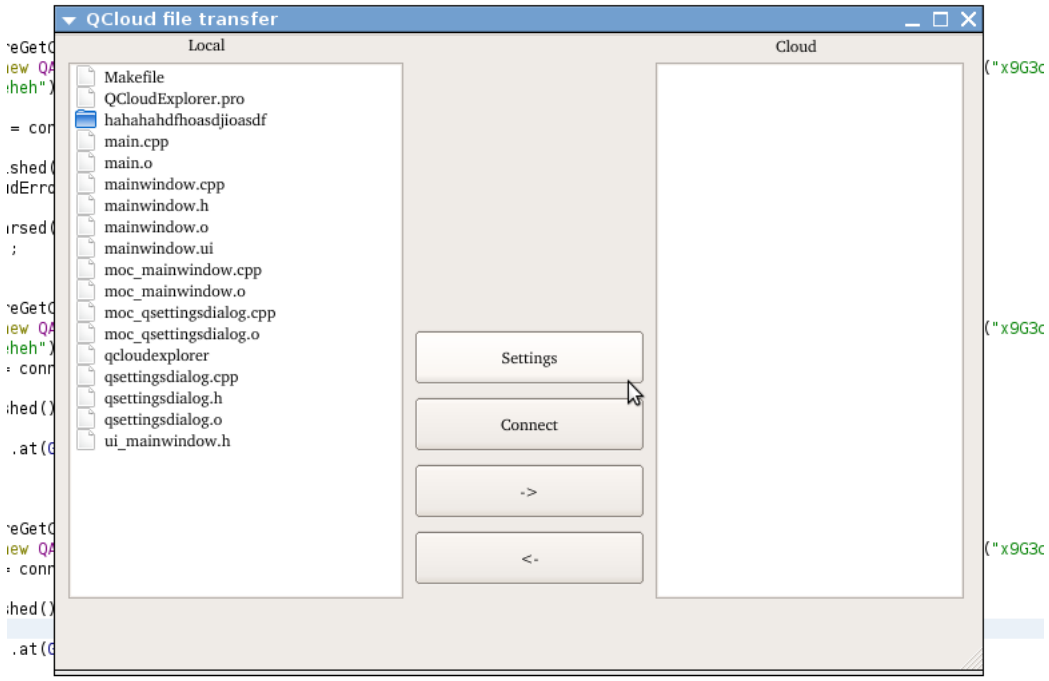

Figure 5.3: QCloudExplorer

be seen in Figure 5.3. The program takes the users authentication credentials in the settings menu, and also to which service the user wants to connect. After that the connection can be created. The difference between QCE and QCT is that QCE supports uploading full directories at once. Both of the example programs use the API in similar fashion, but the QCE uses the signals from the API to its full advantage. The advantage of signals is really noticeable in GUI programming as the GUI can respond to the events happening. For example, if the user decides to upload a directory, QCloudConnection emits a signal containing the amount of files in the folder. GUI can then update the progress as the files are uploaded one by one, and when the operation is finished QCloudConnection emits finished() and the GUI can update the directory listing in the Cloud. The full source code for QCloudExplorer can be found from YouSource [31].

5.5 Evaluation of the implementation

Evaluating something that one has done himself is hard, but I will try to be as non-partial as possible. I will use the criteria for good API's I presented in Section 5.1 as a guideline and compare the design to them. This will give me a way to evaluate the API. The API implementation has its good points, and it does comply to most of the good API design guidelines. Anyhow there are many things that one could have done differently. For example, the function that takes care of the creation of the requests, QCloudConnection::encode(), could have been made in to its own class. At the moment it is hidden so that the end user can not use it for him/herself. If the encode()-function resided in its own class the API-user could easily make own requests, and not be forced to use the ready functions.

Another big thing in the API is the sendHead(), sendPut(), and sendGet()-functions are at the moment synchronous, as there is an eventloop running until the Cloud has replied. This was a design choice that came from the fact that I wanted the getCloud-Dir() and getCloudDirContents() return a list of their contents in a parsed form. If there were no eventloop running in the send-functions, it would return the pointer to the QNetworkReply before the reply would be finished. This would lead to exceptions in the parsing as it was not ready to be parsed. I started to incorporate the asynchronous side with the functions that have async in their name. The next big change will be to change the API to use them, thus providing a truly asynchronous operation. When starting the development I looked at QNAM for guidance and I cannot make the comparison unless the QCloud is truly asynchronous.

The three different QCloudResponse-classes provide the needed functionality, as the functions can return a pointer to the object before the operation has finished. Then the user can ask from the response is it ready to be used. Nothing is done to the response before it is ready so there is less chance for errors. QNetworkReply contains a well designed error handling features so with including the reply in the response brings to the table the ability to use them in conjunction with my own errors. The user can also decide in which format the response is returned: parsed or unparsed. Parsed messages leave a lot of information out, so the ability to use unparsed messages is nice. Asynchronous functions could have returned the QNetworkReply pointer that QNAM returns, but then the user would have to create a parser for the XML-formatted response. This design provides the user much more usability, but is also more error prone.

All of the writers that I named in the Section 5.1 mention the readability and the

documentation as a huge deal in API quality. When an developer can straight use the API by just reading the function declarations and a little bit of documentation, one could presume that he would recommend the API to others. I have documented the classes and functions to a certain degree, and there is an readme file where I explain the usage, and needed tools for compilation. There is still need for more documentation and propably a third example that would show how to use the API to store settings in the Cloud. The two examples that exist are pretty similar in the usage.

In Section 5.1 all of the writers mention that the API should do one thing and that well. Henning [51] also mentions that the API should handle the border conditions well. For really being sure that these conditions are fullfilled, the API should be tested well. Tests have been written with care and they should test every possible combination of inputs and errors. At the moment the API is tested with 27 unit tests, but the test coverage is not wide enough to be sure. The tests prove that the API works as intended, but the error handling is still under construction.

The greatest problem with the API at the moment is that it is pretty much made for the one task: sending and receiving files from the Cloud. The API should have been designed so that it offered the tools needed. At the moment it does it, but for example get and put functions should have been made for QFile and QDir, so that there would be a chance to use the API without the QCloudFile and QCloudDir. As I mentioned in the earlier, files transferred have many different states. The handling of these states would have been easier if the API provided tangible version control. This is what I tried to provide with the QCloudItem's isLocal()-function. It does provide information, but I think that it should be thinked through again. At the moment it does not provide the needed functionality as there are three states and isLocal() can only provide two. There should be a enumeration that handles the different states of each QCloudItem.

Amazon's decision that every user shares namespace for buckets has been a enigma for me. As it now is implemented, QCloud API provides a way to transfer a full folder to the Cloud. When using the put(QCloudDir*)-function it first creates a new bucket with the name of the folder. This will lead to errors most of the times as the most used folder names are already taken in the S3. Azure's solution for this is storage accounts. Every user has own namespace inside the storage account.

All-in-all there is still much to be done for the API to be complete, but it can already be used in the way intended. In the beginning there was talk that the ta-

ble storage should be integrated also in the API. Tables are not supported at the moment, but there is examples that show how the API could be used with tables. Amazon's table storage can be used in the same fashion as files, but there are some differences. These differences make the generalization of the API a bit tougher, for example synchronization between Cloud tables and locals can be really difficult as Cloud databases are almost without exception key-value stores. The API could be, with some modification, used to send tables to the Cloud and then make queries from there.

6 Conclusion

The purpose for this thesis was to inspect the different service offerings of Cloud providers and make an unified API to cover them. The services were later limited to storage aspects. Both table and blob storage was inspected, but only blob storage did make it to the final API. The complete exclusion of Qt and C++ in the official API support from the SP's made this study neccesary and acute, as easy integration with the Cloud is a needed property in modern software development.

In this thesis I first presented the paradigm shift from distributed computing, to grid computing and finally to cloud computing. I went through the history of Cloud and how the first SP's started offering their services. Then I presented the different service providers and how they sell their Cloud services. This provided a much needed comparison of the pricings and properties received from the service. After that I presented the Qt framework and it's history. Now the release of Qt 5 is even closer as Digia released the Qt5 RC already and the final release should be out soon. I also presented the needed addons I used in testing (QTest) and documentation (QDoc). After that I presented the differences in general software development for the desktop and Cloud. And finally I presented the end product of the thesis: QCloud API.

QCloud API does provide the wanted functionality, but there is still things to be done before it can be thought to be complete. At the moment there is conversation going on in the development mailing list of qt-project, and QCloud API will be accepted to the Qt playground. There has also been some interest on continuing the development of the API so that it someday could be integrated to the Qt framework. I will continue to be active in the conversation and provide reasonings behind my design choices.

There is still much to be done with Cloud API's as the different service providers have different implementations and different rules to comply to. This can make the development for many Cloud platforms a cumbersome effort. With a general API that would provide a abstraction between the developer and the Cloud, the usage of different Cloud services would be easier. Now the developer needs to make sure what Cloud provider to use and how their API's differ from the others. QCloud API

can provide a real competitive edge for Qt as it can ease the move between SP's, as the needed changes on code level are minimal.

7 References

[1] Access Control Service 2.0. http://msdn.microsoft.com/en-us/library/windowsazure/gg429786.aspx. Acessed 7.5.2012.

[2] Amazon cloudwatch. http://aws.amazon.com/cloudwatch/. Accessed 13.4.2012.

[3] Amazon elastic cloud compute(ec2). http://aws.amazon.com/ec2/. Accessed 10.4.2012.

[4] Amazon free usage tier. http://aws.amazon.com/free/. Accessed 4.4.2012.

[5] Amazon s3 pricing. http://aws.amazon.com/s3/pricing/. Accessed 12.4.2012.

[6] Amazon web services: Overview of security processes. http://d36cz9buwru1tt.cloudfront.net/pdf/AWS_Security_Whitepaper.pdf. Accessed 13.4.2012.

[7] Aws security and compliance center. http://aws.amazon.com/security/. Accessed 13.4.2012.

[8] BOINC. http://boinc.berkeley.edu/. Accessed 29.3.2012.

[9] Boinc stats. http://boincstats.com/. Accessed 29.3.2012.

[10] Cloud computing layers. http://upload.wikimedia.org/wikipedia/commons/3/3c/Cloud_computing_layers.png. Accessed 30.3.2012.

[11] Did Google's Eric Schmidt Coin "Cloud Computing" ? http://cloudcomputing.sys-con.com/node/795054. Accessed 2.4.2012.

[12] Ftp Example. http://doc.qt.digia.com/latest/network-qftp.html. Accessed 12.11.2012.

[13] Google app engine campfire one transcript. https://developers.google.com/appengine/articles/cf1-text. Accessed 4.4.2012.

[14] Google Apps. http://www.google.com/apps. Accessed 19.4.2012.

[15] Google Trends: cloud computing, grid computing. http://www.google.com/trends/?q=cloud+computing,+grid+computing. Accessed 20.4.2012.

[16] Grid: More bytes for science. http://public.web.cern.ch/public/en/spotlight/SpotlightGrid-en.html. Accessed 29.3.2012.

[17] Introduction to Qt Quick. http://doc.qt.nokia.com/4.7/qml-intro.html. Accessed 3.5.2012.

[18] Making requests using the rest api. http://docs.amazonwebservices.com/AmazonS3/latest/dev/RESTAPI.html. Accessed 11.4.2012.

[19] Modular Class Library. http://qt.nokia.com/products/library. Accessed 24.4.2012.

[20] Multipart Upload Overview. http://docs.amazonwebservices.com/AmazonS3/latest/dev/mpuoverview.html. Accessed 22.10.2012.

[21] Qdoc Reference Documentation. http://doc.qt.digia.com/qdoc/01-qdoc-manual.html. Accessed 9.11.2012.

[22] Qftp Class Reference. http://doc.qt.digia.com/latest/qftp.html#details. Accessed 12.11.2012.

[23] Qt 5 Alpha. http://labs.qt.nokia.com/2012/04/03/qt-5-alpha/. Accessed 3.5.2012.

[24] Qt5 Alpha. http://qt-project.org/wiki/Qt-5-Alpha. Accessed 11.5.2012.

[25] Qtestlib Manual. http://doc.qt.digia.com/qt/qtestlib-manual.html. Accessed 9.11.2012.

[26] Securing the Cloud Infrastructure. http://cdn.globalfoundationservices.com/documents/Strategy_Brief_Securing_Cloud_Infrastructure.pdf. Accessed 29.5.2012.

[27] Security first: Google apps and google app engine complete ssae-16 audit. http://googleenterprise.blogspot.com/2011/08/security-first-google-apps-and-google.html. Accessed 12.4.2012.

[28] Seti@home webpage. http://setiathome.berkeley.edu/. Accessed 29.3.2012".

[29] Signals & Slots. http://qt-project.org/doc/qt-5.0/signalsandslots.html. Accessed 26.4.2012.

[30] Survey: Cloud computing 'no hype', but fear of security and control slowing adoption. http://www.circleid.com/posts/20090226_cloud_computing_hype_security. Accessed 11.4.2012.

[31] Thesis-code. http://yousource.it.jyu.fi/thesis-code/. Accessed 17.11.2012.

[32] Understanding Page Blobs and Block Blobs. http://msdn.microsoft.com/en-us/library/windowsazure/ee691964.aspx. Accessed 22.10.2012.

[33] Using CMake to Build Qt Projects. http://qt-project.org/quarterly/view/using_cmake_to_build_qt_projects. Accessed 27.4.2012.

[34] Virtual Machine Role. http://msdn.microsoft.com/en-us/gg502178. Accessed 11.5.2012.

[35] What is AWS. http://aws.amazon.com/what-is-aws/. Accessed 10.4.2012.

[36] What is google app engine. https://developers.google.com/appengine/docs/whatisgoogleappengine. Accessed 4.4.2012.

[37] Why app engine. https://developers.google.com/appengine/whyappengine. Accessed 12.4.2012.

[38] Windows azure dynamic scaling sample. http://archive.msdn.microsoft.com/azurescale. Accessed 12.4.2012.

[39] Windows azure tour: What is windows azure. https://www.windowsazure.com/en-us/home/features/overview/. Accessed 5.4.2012.

[40] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53:50–58, April 2010.

[41] D. Barkai. Technologies for sharing and collaborating on the net. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 13 –28, aug 2001.

[42] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[43] Joshua Bloch. How to Design a Good API and Why it Matters. `http://lcsd05.cs.tamu.edu/slides/keynote.pdf`. Accessed 21.11.2012.

[44] Jon Brodkin. Gartner: Seven cloud-computing security risks. *Network World*, July 2008.

[45] Dimitri van Heesch. Doxygen. `http://www.stack.nl/~dimitri/doxygen/`. Accessed 9.12.2012.

[46] Matthias Ettrich. Designing Qt-Style C++ APIs. `http://doc.qt.digia.com/qq/qq13-apis.html`. Accessed 21.11.2012.

[47] Ian Foster and Adriana Iamnitchi. *On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing*, volume 2735 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003.

[48] Ian Foster and Carl Kesselman. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[49] Simson L. Garfinkel and Simson L. Garfinkel. An evaluation of amazon.s grid computing services: Ec2, s3, and sqs. Technical report, Center for, 2007.

[50] Sharon Eisner Gillett and Mitchell Kapor. The self-governing internet: Coordination by design. Working Paper Series 197, MIT Center for Coordination Science, January 1997.

[51] Michi Henning. Api design matters. *Queue*, 5:24–36, May 2007.

[52] James B. D. Joshi, Walid G. Aref, Arif Ghafoor, and Eugene H. Spafford. Security models for web-based applications. *Commun. ACM*, 44(2):38–44, February 2001.

[53] Charlie Kaufman and Ramanathan Venkatapathy. Windows azure. security overview. 2010.

[54] Leonard Kleinrock. An internet vision: the invisible global infrastructure. *Ad Hoc Networks*, 1(1):3 – 11, 2003.

[55] Peter Mell and Tim Grance. The nist definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50, 2009.

[56] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja1, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing. Technical report, Hewlett-Packard Company.

[57] Open Stack. OpenStack - Open Source Cloud Computing Software. `http://www.openstack.org/`. Accessed 29.11.2012.

[58] Martin Reddy. Api design for C++. `http://dl.acm.org/citation.cfm?id=1971974`. Accessed 19.11.2012.

[59] The Apache Software Foundation. CloudStack - Open Source Cloud Computing Software. `http://incubator.apache.org/cloudstack/software.html`. Accessed 29.11.2012.

[60] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008.

[61] Aaron Weiss. Computing in the clouds. *netWorker*, 11(4):16–25, December 2007.

A API implementation

```
class QCloudConnection : public QObject
{
    Q_OBJECT
public:
    struct Request {
    QHash<QString, QString> headers;
    };

   virtual QCloudFile* get(QString bucket, QString fileName) = 0;
   virtual bool get(QCloudDir &d) = 0;
   virtual bool put(QCloudFile &file, QString bucket) = 0;
   virtual bool put(QCloudDir &dir) = 0;
   virtual QList<QString> getCloudDir() = 0;
   virtual QList<QString> getCloudDirContents(QString bucketName) =
   virtual bool deleteBlob(QString name, QString bucket) = 0;
   virtual bool deleteCloudDir(QString bucket) = 0;
   virtual QList<QString> parseCloudDirListings(QByteArray &data) =
   virtual bool cloudDirExists(const QString &dirName) = 0;
   virtual bool createCloudDir(const QString &dirName) = 0;
   virtual void setOverrideLocal(bool value) = 0;
   virtual void setOverrideCloud(bool value) = 0;
   virtual QCloudFileResponse* asyncGetCloudFile(QString &bucket, QS
   virtual QCloudListResponse* asyncGetCloudDir() = 0;
   virtual QCloudListResponse* asyncGetCloudDirContents(QString &clo

protected:
    QCloudConnection();

private:
    virtual QNetworkReply* sendGet(const QNetworkRequest &req) = 0;
```

```
    virtual QNetworkReply* sendPut(const QNetworkRequest &req, const
    virtual QNetworkReply* sendHead(const QNetworkRequest &req) = 0;
    virtual QNetworkRequest encode(const Request &r) = 0;

signals:
    void finished();
    void failed();
    void getCloudDirFinished();
    void putCloudDirFinished();
};
```

B Source of demo application

```
#include <QtCore/QCoreApplication>
#include "qamazonconnection.h"
#include <iostream>

QList<QString> readAuthFromFile() {
    QList<QString> auth;
    QFile f("auth.txt");
    if (!f.open(QIODevice::ReadWrite | QIODevice::Text)) {
        qDebug() << "could not open file auth.txt, you sure it exists
    }
    QByteArray line;
    while ((line = f.readLine()) != 0) {
        line.replace("\n", "");
        auth.append(line);
    }
    f.close();

    return auth;
}

void printList(QList<QString> list) {
    foreach(QString s, list) {
        qDebug() << s;
    }
}

int getFile() {
    qDebug() << "Doing get-operation";
    qDebug() << "Reading authentication from file auth.txt";
    QList<QString> auth = readAuthFromFile();
```

```
    QAmazonConnection conn(auth.at(0).toAscii(), auth.at(1).toAscii()
    qDebug() << "Connection initialized";
    QList<QString> buckets = conn.getCloudDir();
    qDebug() << "Your account contained the following buckets:";
    printList(buckets);
    qDebug() << "Which bucket would you like to inspect";
    std::string s;
    std::cin>>s;
    QString bucket(s.c_str());
    qDebug() << "Getting contents of " + bucket;
    QList<QString> files = conn.getCloudDirContents(bucket);
    qDebug() << "The bucket contained following files :";
    printList(files);
    qDebug() << "What would you like to get?";
    std::string file;
    std::cin>>file;
    QString qfile (file.c_str());
    qDebug() << "Downloading the file " + qfile;
    QCloudFile* down = conn.get(bucket, qfile);
    qDebug() << "File downloaded, thx";
    return 0;
}

int putFile(QString name) {
    qDebug() << name;
    qDebug() << "Doing put-operation";
    qDebug() << "Reading authentication from file auth.txt";
    QList<QString> auth = readAuthFromFile();
    QAmazonConnection conn(auth.at(0).toAscii(), auth.at(1).toAscii()
    qDebug() << "Connection initialized";
    QList<QString> buckets = conn.getCloudDir();
    qDebug() << "Your account contained the following buckets:";
    printList(buckets);
    qDebug() << "Please specify in which bucket would you like to pla
    std::string bucket;
```

```
    std::cin>>bucket;
    QString qbucket(bucket.c_str());
    if (!buckets.contains(qbucket)) {
        qDebug() << "The bucket did not exists. exitting";
        return -1;
    }
    qDebug() << QString("Putting file %1 to bucket %2").arg(name).arg
    QFile f(name);
    if (!f.exists()) {
        qDebug() << "File did not exist, exitting";
        return -1;
    }
    QCloudFile cf(f);
    if (conn.put(cf, qbucket)) {
        qDebug() << "File uploaded, exitting";
        return 0;
    }

}

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    if (argc < 2) {
        qDebug() << "You did not specify operation, please use qcloud
        return 0;
    }

    if (strcmp(argv[1], "put") == 0) {
        putFile(QString(argv[2]));
    } else if (argv[1]){
        getFile();
    } else {
       qDebug() << "Please specify operation, put or get" ;
    }
```

```
        //return a.exec();
    }
```

C Readme

QCloud–API

QCloud–API provides public API to call Windows Azure and Amazon S3.

Requirements:
Qt5 (tested with beta1)

Installation:

Linux: (Tested: Debian 2.6.32–45)
qmake
make

The user needs account on either Amazon or Windows Azure. The credentials to amazon can be found in the AWS dashboard (https://console.aws.amazon.com/). Azure's credentials can be found from https://manage.windowsazure.com/. Insert the credentials as cloudconnections params as they are presented in the sites. Do not do anything to them.

Note:

Amazon has a interesting naming contention for the buckets. The bucket names are shared between all of the users (a single namespace), so the most used are already taken. The rest of the naming rules are as follows:

Bucket names must be at least 3 and no more than

```
    63 characters long.
    Bucket name must be a series of one or more labels
    separated by a period (.), where each label:
     Must start with a lowercase letter or a number
     Must end with a lowercase letter or a number
     Can contain lowercase letters, numbers and dashes
     Bucket names must not be formatted as an
      IP address (e.g., 192.168.5.4)


Example usage.
 Getting a file from a certain bucket in the cloud:
    Include the lib in your project.
    Create a new CloudConnection for the service you want
    (Azure*, Amazon*)

    Get the list of buckets with QCloudConnection::getCloudDir()
    Get the contents of certain bucket with
        QCloudConection::getCloudDirContents(QString name)

    Get the file using
    QCloudConnection::get(QString bucket, QString fileName)

 Putting a file to a certain bucket in the cloud:
    Include the lib
    Create new connection for the service
    if (bucketname not known)
       Get bucket list with getCloudDir()
    create a QFile that contains the file
    Create qcloudfile from the qfile
    call put(qcloudfile, bucket)

 examples/QCloudTransfer/main.cpp is a good indication how
 to use the api.
```

TODO:

- Both services need implementation of the deleteBlob and deleteCloudDir -functions.

- As AmazonS3 has the requirement that buckets cannot contain uppercase letters, somekind of check should be placed somewhere. At the moment error is thrown.

- The content-type-header should be replaced according to the real type of the file sent. At the moment it is always text/plain.

- As using QCloud should be as similar to QNAM as possible, the structure should be changed to use the async methods.

- QCloudTransfer-example supports only AmazonS3 at the moment. There needs to be some change that can change the service provider according the params. Maybe in the auth.txt the first line could be the provider (aws, azure) etc..

- As a cloudconnection is created check if the credentials are correct

If bugs are found please contact me at jlaitinen@gmail.com, please include [qcloud] in the header.

D Amazon's encode function

```
/*!internal
  \brief Creates the request that is then sent to Amazon, everything
  general as possible and this should take care of getting buckets an
  */
QNetworkRequest QAmazonConnection::encode(const Request &r) {

    QString timeString = QString::number(QDateTime::currentMSecsSinceE
                                                                / 1000
    QByteArray stringToSign = r.headers.value("verb").toAscii() + "\n

    stringToSign += "\n\n";
    stringToSign += timeString + "\n";

    QString urlString = "";

    if (r.headers.contains("bucket")) {
        QString bucket = r.headers.value("bucket");
        stringToSign += "/" + bucket + "/";
        urlString = "http://" + bucket + "." + this->host + "/";

        if (r.headers.contains("filename")) {
            QString value = r.headers.value("filename");
            urlString += value;
            stringToSign += value;
        }
    } else {
        stringToSign += "/";
        urlString = "http://" + this->host + "/";
    }
```

```
    QUrl url(urlString);

    QByteArray hashedSignature = HmacSHA::hash(HmacSHA::SHA1, stringTo
                                                this->secret);


    replaceUnallowed(&hashedSignature);
    QNetworkRequest req;

    url.addEncodedQueryItem("AWSAccessKeyId", this->password);
    url.addEncodedQueryItem("Signature", hashedSignature);
    if(r.headers.value("verb") == "PUT") {
        QString value = r.headers.value("filesize");
        if (value != "0") url.addEncodedQueryItem("Content-Type",
                          r.headers.value("Content-Type").toAscii());
        url.addEncodedQueryItem("Content-Length", value.toAscii());
    }
    url.addEncodedQueryItem("Expires", timeString.toAscii());
    req.setUrl(url);
    return req;
}
```

E Azure's encode function

```
nternal
  */
QNetworkRequest QAzureConnection::encode(const Request &r) {
    QString urlString("http://" + this->url);

    if (r.headers.contains("path")) {
        urlString += r.headers.value("path");
    }

    if (r.headers.contains("operation")) {
        urlString += "?" + r.headers.value("operation");
    }

    QUrl url = QUrl::fromEncoded(urlString.toAscii());
    QNetworkRequest req;
    req.setUrl(url);

    QByteArray stringToSign;
    stringToSign.append(r.headers.value("verb"));
    stringToSign.append("\n");

    for (int i = 0; i < head.requiredHeaders.size(); i++) {
        if (head.requiredHeaders.at(i).first == "Content-MD5" && r.he
                                                == "PUT" && !r.header
            stringToSign += "0\n";
        } else if (r.headers.contains("size") &&
                  head.requiredHeaders.at(i).first == "Content-MD5")
            stringToSign += r.headers.value("size") + "\n";
        }
        else stringToSign += head.requiredHeaders.at(i).second;
```

```
    }

    QString date = dateInRFC1123();

    if (r.headers.contains("size") /*&&
        !r.headers.value("operation").contains("restype")*/) {
       stringToSign += "\nx-ms-blob-type:BlockBlob";
    }
    stringToSign += "\nx-ms-date:" + date;
    stringToSign += "\nx-ms-version:2009-09-19\n";

    stringToSign += "/" + this->storageAccountName + r.headers.value(
    if (r.headers.contains("operation")){
        QString temp = r.headers.value("operation");
        stringToSign += "\n" + temp.replace("=", ":").replace("&", "\
    }
    QByteArray key = QByteArray::fromBase64(this->storageKey);
    QByteArray hash = HmacSHA::hash(HmacSHA::SHA256, stringToSign, ke

    QString version = "2009-09-19";
    QByteArray test = this->authentication + " " + this->storageAccoun

    req.setRawHeader("Authorization", test);
    req.setRawHeader("x-ms-date", date.toAscii());
    req.setRawHeader("x-ms-version", version.toAscii());
    if(r.headers.value("verb") == "PUT")  {
        req.setRawHeader("Content-Length", r.headers.value("size").to
        req.setRawHeader("x-ms-blob-type", "BlockBlob");
    }

    return req;
}
```