

By Streamberry & Sons

http://streamberry.net

Adam Harwood  
Jake Saunders  
James Lumsden  
Robert Pamely  
Vitaliy Kishikov  
Yordan Byalkov

Streamberry Final Report

# Project Overview

## Introduction

Streamberry is an innovative social media player, designed for use in shared houses, or any circumstance where computers share a local area network. It is able to present every computer’s media library together in a seamless, unified interface, allowing users to stream media from others on their local network as easily as they would play a file from their own computer. Streamberry provides intelligent auto-discovery and synchronisation of other clients’ media libraries, removing all the hassle of setting up audio/video sharing manually using iTunes or Windows Media Player. Written in C++ using the Qt development libraries, Streamberry supports all 3 major operating systems: Windows, Mac OS X and Linux.

## 

While there are many media players on the market, we found none that met all our specific goals for Streamberry, which boiled down to:

* Cross-platform – Mac, Windows & Linux
* Zero-configuration streaming
* Auto-discovery of other clients
* Seamless merging of all libraries – no distinction between local and remote files

The completely zero-configuration approach to streaming makes Streamberry unique. Very little setup is involved - simply enter a nickname and choose which directories you would like to share and you’re ready to go. All remote libraries on the network are then automatically merged into your own, with no distinction between your own files and the files on the network. This is contrasted with Windows Media Player, where access from a non-windows machine requires configuration of Windows proprietary sharing on the non-windows side, and advanced permissions on the Windows computer. Other peoples’ libraries are also not automatically discovered. iTunes does a better job of auto-discovery, but this requires you to install Apple’s “Bonjour” service - something we do not require with Streamberry. iTunes does not however support library merging, where all libraries are shown in a single view like Streamberry. Winamp contains built-in streaming using a SHOUTcast server - which requires more configuration than an average user is able or willing to do.

We are releasing Streamberry as free, open source software. It is available to download now from our website at http://streamberry.net. The website is shown below.

*The Streamberry Website. http://streamberry.net*

## Justification Analysis

We decided early on that as most media players are free, the success of a media player is measured by the size of its user base. No other company has really made a media player that takes advantage of the millions of home networks available around the world. Other solutions are starting to take notice that streaming media across your home is something people want to do, but none of them make it particularly easy or enjoyable to do.

We hope that Streamberry’s unique features will encourage users to switch from their usual application, as there is definitely a gap in the market for a media player like Streamberry, especially on Linux.

Entering the already over-saturated media player market is difficult and only a free product can really survive, so we never expected to make a profit. Also, our streaming backend is based on libVLC, which is available under the GNU General Public License v3, which meant that we would have to release the source code of our application. When we decided to use the Qt libraries, we were also forced to abide by their non-commercial license.

Despite the difficulty of creating a software product that is competing against big names like Microsoft and Apple, we believe we have found a gap in the market that no other company caters for. Streamberry is primarily targeted at student houses, as this a very common situation leading to several different music libraries being in the same household. Being students ourselves, our experience with this spurred our inspiration for the project. However, Streamberry has use in a variety of other situations, for example small clubs or festivals where artists could showcase their media to other Streamberry users.

The student audience influenced our design and branding - inspiring the informal logo and tongue-in-cheek name. For the graphical user interface, we chose to make something sleek and individual, yet familiar, so new users would feel comfortable using controls similar to players they already use. This paid off, as a common piece of feedback we got from testers was, ‘the program feels like Spotify’ with some original touches. Throughout the project, we were aiming to produce a media player that was fun to use, not dull and corporate like some others.

To market Streamberry, we will use an entirely online, social-media based campaign, we will be marketing our application through our Twitter account, @streamberrysons, our website, via word of mouth and through press coverage. This fits in with our target demographic of students in shared accommodation, who would be more receptive to this type of advertising.

We made this decision after looking at how two relatively new media players, Spotify and Songbird, gained momentum in the market place. Songbird became popular when some big media names gave it attention due to some innovative features and iPod support. Spotify gained a large user base, especially amongst young people, after gaining a lot of media attention and releasing its mobile application. A mobile application would possibly be a good route to go down in the future for Streamberry.

## Feasibility Assessment

Our group was pre-selected, meaning none of us had worked in a team together before. It also meant we could not pick 6 people with different specialisations, and so we had to make do with the skills that we had. That said, the skill set of the group was pretty diverse, which helped as there was always someone willing to step up to complete the next required task. We are all competent programmers also, and so together we had the skill set and ability to implement the program we designed.

Although there was no budget, a budget was not actually required, apart from £6 or so for the domain name and for printing the final report. The only problem we encountered was underestimating the amount of time things would take, meaning a lot of the tasks were completed in the second half of the project where it ought to have been more spread.

# Project Design

## Original Specification

In the design stage of the project, we set out a list of requirements that Streamberry at a minimum must achieve. These were:

* Product can play media stored locally
* Product can play media stored on another machine on the local network
* Product can run on machines running Windows, Linux or Mac OSX and interaction between the operating systems is seamless
* Product can display and search through media stored on other machines on the local network. There will be no distinction between local and remote media.
* Product can automatically detect other machines running the software and will update the media list accordingly
* Product will be stable and not fail unexpectedly
* Unique GUI/library view

We also had some less important extras that we were hoping to include. We had to drop these however due to time constraints as we decided to perfect other parts of the system instead. These were:

* Media buttons on keyboard
* IMDB integration
* Last.fm integration
* On-the-fly transcoding to deal with slow networks
* Implement a web front-end and play files from a web page
* Messaging service
* Duplicate files detection
* Have a caching system so that a client can finish listening to the song if the remote computer goes offline
* Multiple ways to view media in your library

We made sure to refer to the minimum specification list whenever implementing new features to ensure that we were always within the design specification while at the same time keeping in line with the other, less fundamental, ideas that we set out to include.

Being a media player, the first and foremost requirement was to be able to play music and video. We knew from the very start that there are libraries available to build up on. We needed something that was easy to integrate into the project, which had to be cross platform to reduce development time. The libraries would also have to have streaming capabilities to enable us to play music and media over the network. We decided to use LibVLC for media playback and streaming. Although VLC documentation was sparse, audio, video playing and streaming to computers on the local network were fully implemented. Currently, a wide range of media formats can be played both locally and through the network. Also, because Streamberry relies on VLC, that means keeping up with the latest releases of VLC give the most up to date range of media playing capabilities.

An important part of Streamberry is the fact that everything should happen seamlessly whether it’s searching through the library containing remote files, listening to your own music or listening to music on a remote computer. This is generally done successfully, however one of the shortcomings of the VLC library that we found was that because data is being streamed, there can be slight delays when switching songs, playing or pausing. This is an issue that cannot be easily fixed because this is how the VLC library is designed. Ideally we would prefer to write our own streaming protocol, but this would be an entire years project in itself and we simply did not have the time. Overall, our test users concluded that the delays are to an acceptable level and to be expected due to the fact that it is coming from another computer.

One of the important aims of Streamberry was its cross-platform compatibility. With the major audience being students, we realise that people use different operating systems, so we decided support all of the three main operating systems: Windows, Linux and Mac OSX. We chose to use the Qt framework so that the program would not have to be re-implemented on each platform. When we started coding, we thought that same code could be used to make the same application on each platform without any modification, but we later discovered that we needed to tweak and debug several things due to how things had to be implemented on different platforms. These were generally small details, such as file paths and how video is displayed, but it took some extra care to make sure the GUI looked the same on each platform. Currently, the program interacts well between the 3 operating systems and works the same whether it is two similar or two dissimilar operating systems.

As well as being able to play music and videos, the program had to include a feature to view and search though a list of the media files. The main view of the program is where all the files in the database are displayed. These files can be sorted either in ascending or descending order by the headings that are in the table, namely Title, Artist, Album, Genre and Track length. Although these are the only headings that are currently available, other fields are also stored in the database. The original plan was to make these headings customisable to be able to add or delete the ones a user would want, but due to the time constraints, we had to prioritise tasks and decided to drop support for this for the time being. The search bar allows the user to search through the media library by Title, Artist, Album and Genre. Both searching and sorting are very useful to users and give a much more pleasant experience when managing a database with thousands of tracks, which is what we expect a majority of people would have. Both searching and sorting are fast and work well on small and large media libraries due to the database back-end, although there is a slight noticeable delay performing these operations when there are more than a few thousand tracks in the library.

For the program to be able to stream files from other computers on the network, there had to be an efficient way of communicating and receiving a list of files that are available on the remote machine. Communication is established between the 2 computers and a newest version of the database is sent over. Initially, this can take some time for a very large library, but afterwards only incremental changes are sent over and if nothing has changed in the database, as soon as the client machine comes online, the files appear. As one of Streamberry’s global aims is to make usage feel seamless, files on other computers are displayed mixed in the same library as those stored locally on the computer.

A big motivation for making Streamberry was the fact that the other solutions that already exist are either difficult to set up and maintain or they are plagued by compatibility issues, so ease of use and setup had to be done just right to differentiate ourselves from our competition. We felt strongly that a media player should be one of the easiest to use programs on the computer, available to all computer uses regardless of technical ability and so some of the most important objectives were simplicity, ease of use and zero-configuration. This would enable a wide range of users to be able to quickly get going with Streamberry. This meant making every part of the program as intuitive as possible. To achieve this, a lot of the program was made to be pre-configured without needing much information or input from the end user. However, some customisation options such as custom headers had to be omitted because there was not enough time available in the final parts of the project. By default, a lot of the internal settings would never need to be changed. Some advanced users might have the need to tweak some settings, such as port numbers for sending the beacons, communication between Streamberry clients or streaming, but there is currently no option to do that. As it stands, the only configuration that is needed is to pick a nickname, then choose which folders to import. This is much easier to set up than many of the other software available packages; therefore this aim has also been achieved.

To appeal to students, we designed a unique GUI that would be nice looking and pleasant to the eye. The feedback we got from our testers indicated that the GUI looks similar to Spotify, but with a lighter theme. In the early stages of planning, we were also going to come up with a new way of organising thousands of tracks but as before, due to time constraints, in the end we had to settle for a simple list of files with the sorting and searching options. There are some interesting aspects in the GUI however, such as the radial progress bar and preview pane, which shows a preview of items in playlists.

When writing up a list of requirements, we also designated a number of hours that we thought it would take for each task. As the project went on, we realised that we underestimated the time it would take to do all the tasks. Due to this, some of those features had to be dropped. These tasks are mostly not difficult, but time consuming, so we decided that we should instead focus on the core functionality of the program. One of the functions that we were going to implement was IMDB and Last.FM integration. We understood that users do not always have media files with all the tags present. This is why Streamberry could have used the filename of the media files to look up the information on the internet, guess which track or film it is, give the user a choice if the name is ambiguous and label the files with reliable tags.

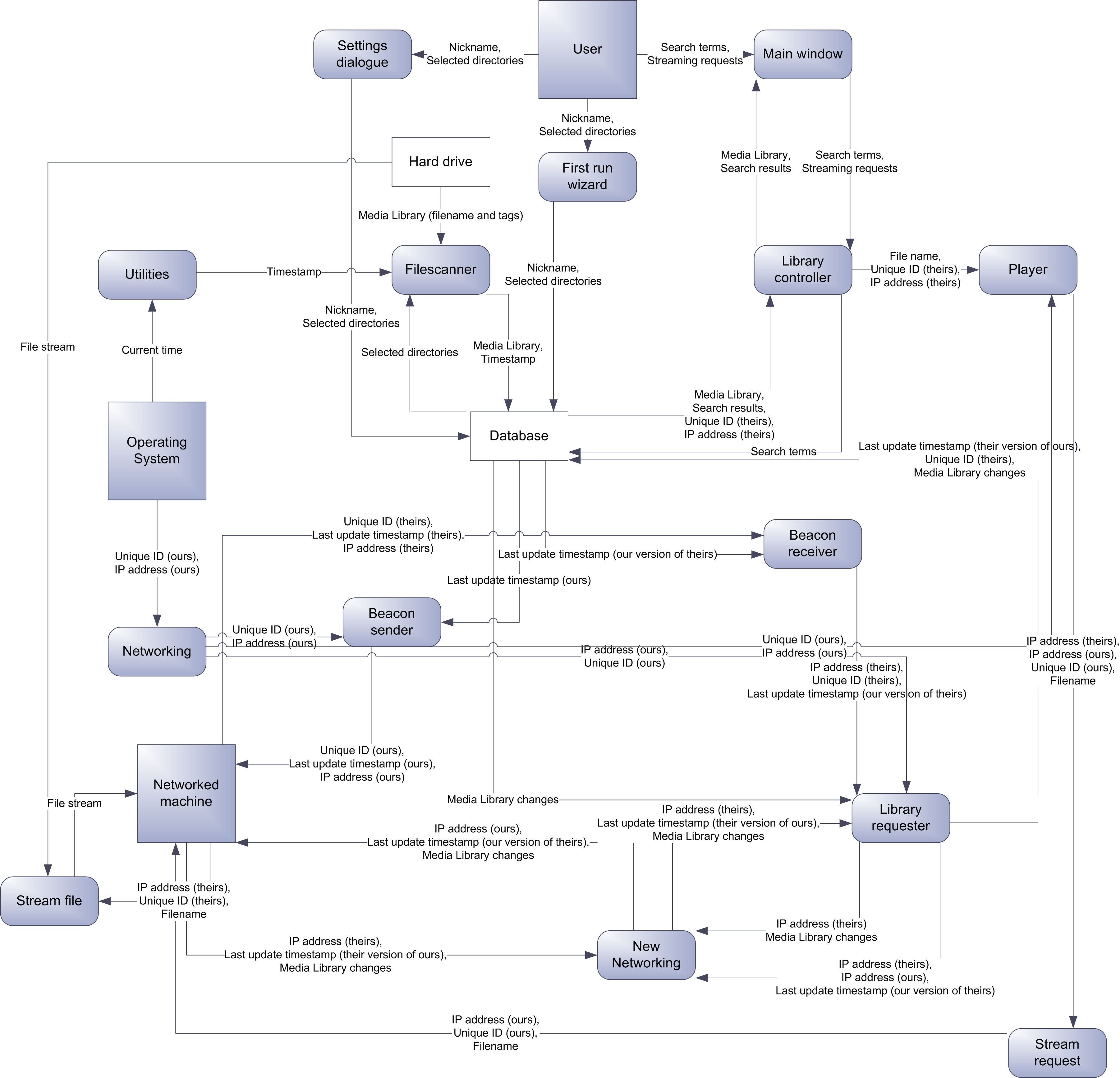
A feature that we thought would be useful on slow networks is real time transcoding of files when they are being streamed to another computer. This option was investigated and it soon became apparent that wireless networks can handle music and some video streaming and the fact that transcoding video is very CPU intensive meant that the gains of transcoding would be negligible when taking into account the fact that the source computer would be very slow and if it cannot keep up with the video, the client requesting the stream would see a stuttering video. Therefore we decided not to implement this feature and instead make the user aware that results of streaming will vary based on how fast and stable the network is.

Also, a problem that we saw with playing remote media file was that we needed a way to deal with the remote machine going offline when listening to a file. One of the ideas that we had was to have a caching system that would download the full file into the buffer and play it from there. The problem with this is that if this was implemented, the best solution would be to also remove streaming and develop our own protocol to transfer files and play them. This would be a major task that would add a lot of hours for a little gain over the approach we took, so we did not implement this feature either.

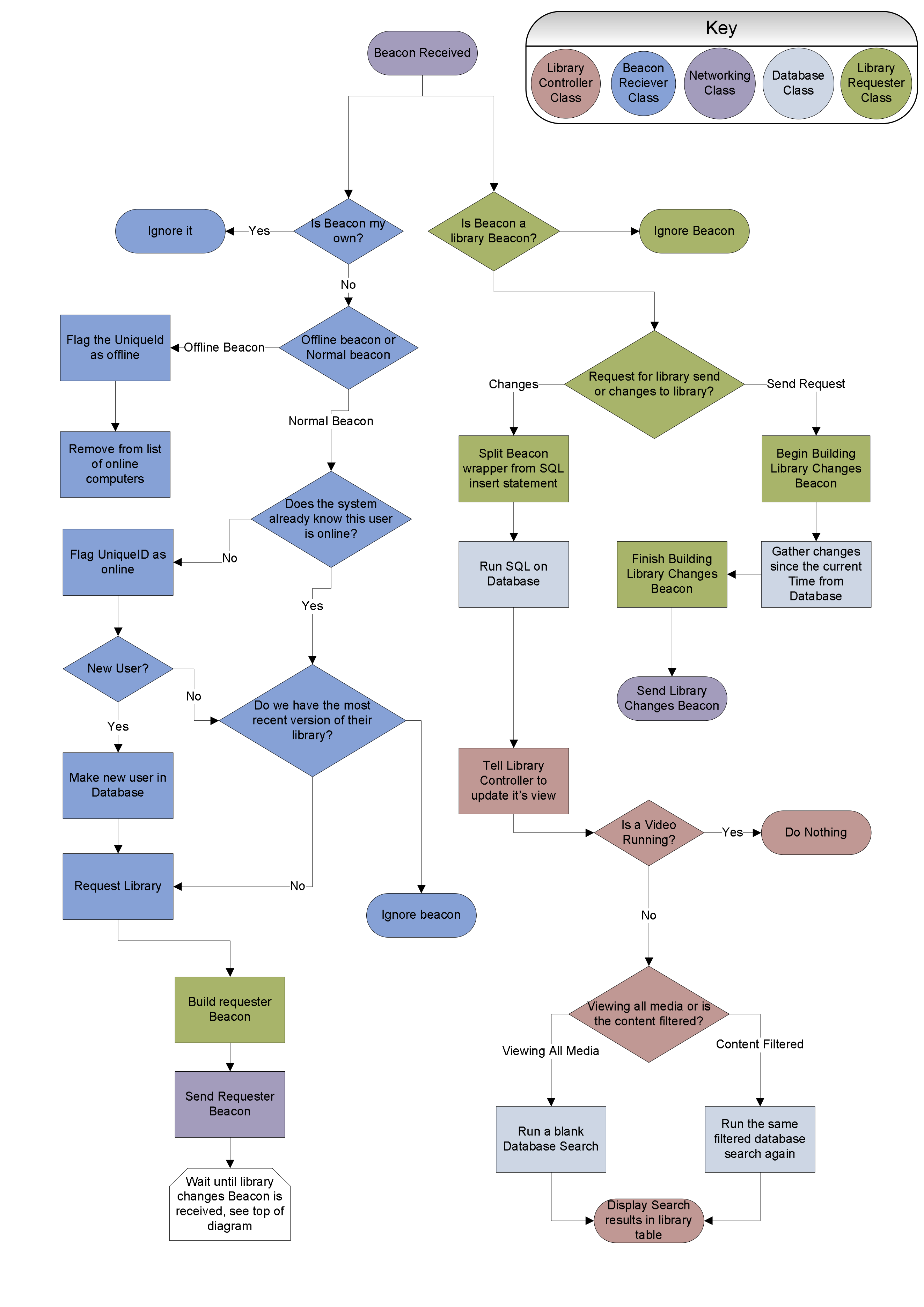
Overall, the defining features of Streamberry, such as easy configuration and setup, cross-platform compatibility, seamless sharing and organisation of media, have been implemented. The features that were not implemented are optional extras which would have been nice to be included in the final product, but unfortunately due to the lack of time had to be missed out in favour of making the core features as best as they can be.

## Diagrams

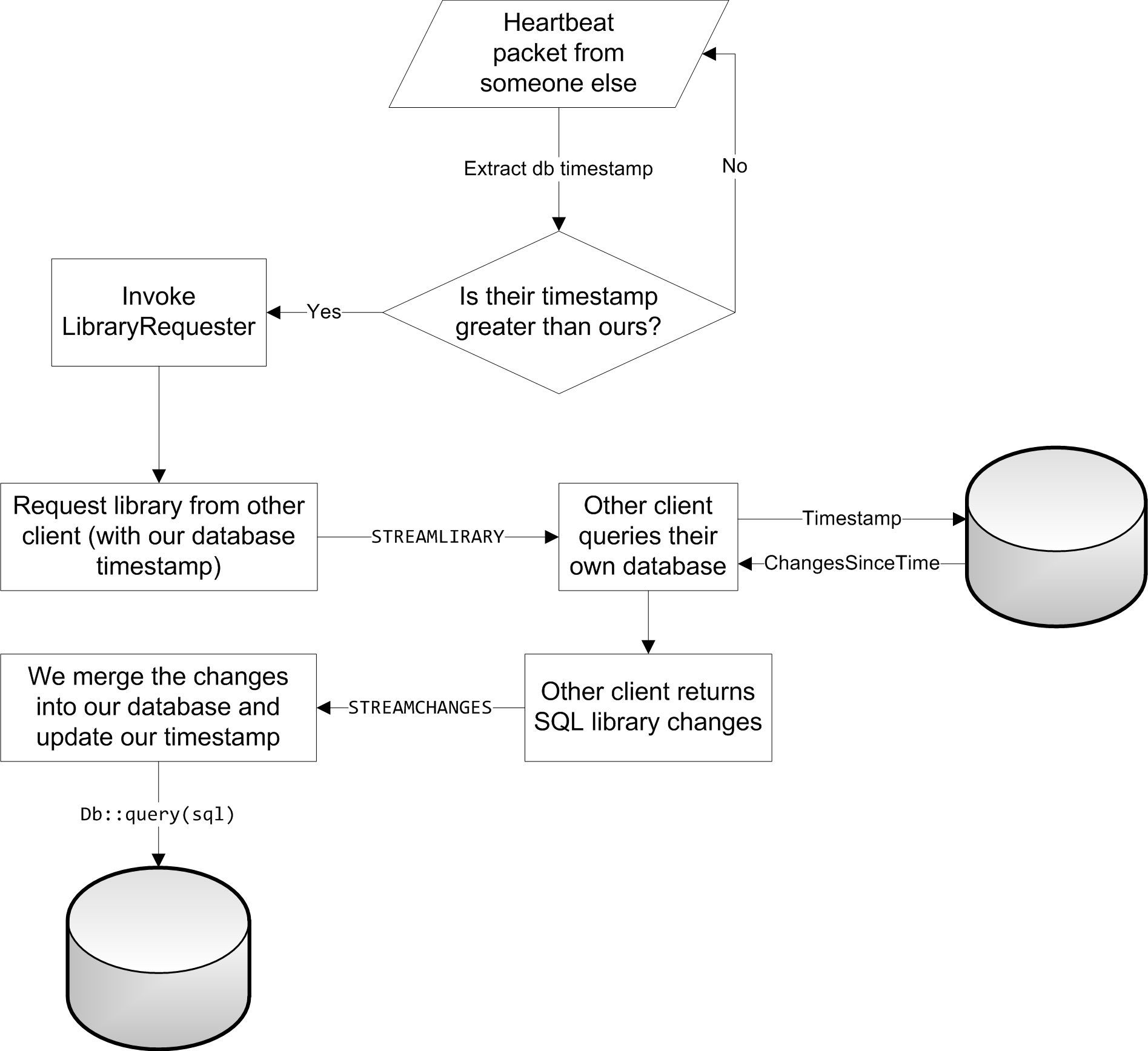
### Data Flow Diagram



### Library Send/Receive Control Flow Chart

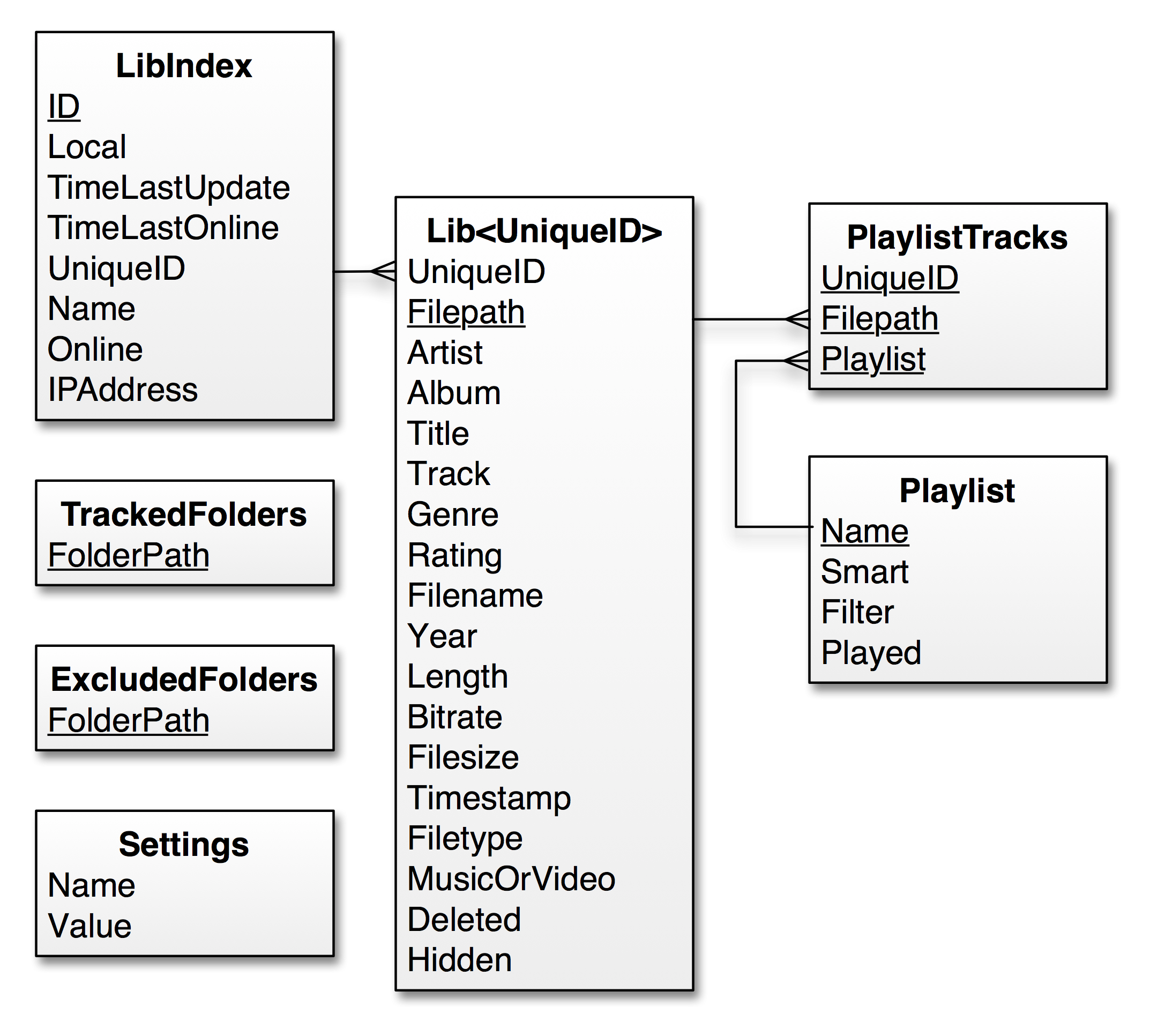


### Library Synchronisation Flow Chart



## Database Design

Streamberry uses an SQLite database to store data. The entity relationship for the database is as follows:



Below is a list of all tables in the database and a description of their fields.

|  |  |  |  |
| --- | --- | --- | --- |
| Table name: **LibIndex** | | Table description: | Stores the list of users every seen on the network and their information. |
| **Field Name** | **Field Description** | **Data Type** | **Key?** |
| ID | A unique ID for each user ever seen. | Integer | Primary |
| Local | Flag saying if this is the record of the local user. | Boolean |  |
| TimeLastUpdated | Timestamp of latest database version. | Integer |  |
| TimeLastOnline | Timestamp of the last time user was seen online. | Integer |  |
| UniqueID | Their unique Streamberry ID. Note that for the local user this is not their real ID and is instead the word ‘Local’ | Varchar |  |
| Name | Friendly name for other users to identify the user with. | Varchar |  |
| Online | Flag stating whether user is currently online. | Boolean |  |
| IPAddress | Stores the last known IP address of the user. | Varchar |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Table name: **Lib<UniqueID>** | | Table description: | The library table stores all of the track information. Every user on the network has their own library table named Lib<UniqueID>, where the UniqueID is the same as the one stored in LibIndex. The number of library tables therefore varies massively as the number of users who could be on a network is unknown and the tables are dynamically created at run-time. LibLocal is always present however. |
| **Field Name** | **Field Description** | **Data Type** | **Key?** |
| UniqueID | The unique Streamberry ID for the user. Same as the table name. | Varchar | Foreign |
| Filepath | Stores the filepath of the media file. | Varchar | Primary |
| Artist | Media file artist name. | Varchar |  |
| Album | Media file album name. | Varchar |  |
| Title | Media file title. | Varchar |  |
| Track | Media file track number. | Varchar |  |
| Genre | Media file genre. | Varchar |  |
| Rating | Media file rating. | Integer |  |
| Filename | Media file filename. | Varchar |  |
| Year | Media file year published. | Integer |  |
| Length | Media file length in seconds. | Integer |  |
| Bitrate | Media file bitrate. | Integer |  |
| Filesize | Media file filesize. | Integer |  |
| Timestamp | Timestamp of when record was last updated. | Integer |  |
| Filetype | Media file filetype. | Varchar |  |
| MusicOrVideo | Whether the media file is music or video. 0 indicates music, 1 indicates video. | Integer |  |
| Deleted | Flag stating whether the media file has been deleted. | Boolean |  |
| Hidden | Flag stating whether media file should not be synced to other computers. | Boolean |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Table name: **Playlist** | | Table description: | Stores information about playlists, except the tracks that are in it. |
| **Field Name** | **Field Description** | **Data Type** | **Key?** |
| Name | The name of the playlist to show in the GUI. | Varchar | Primary |
| Smart | Is the playlist a smart/filtered playlist. | Boolean |  |
| Filter | If the playlist is a filtered playlist, it will use this field. | Varchar |  |
| Played | Timestamp of when the playlist was played. | Integer |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Table name: **PlaylistTracks** | | Table description: | Stores information about which tracks are in the playlists. |
| **Field Name** | **Field Description** | **Data Type** | **Key?** |
| UniqueID | The UniqueID of the user who owns the track. Note that this is not a relationship as this data is used to generate dynamic table name, not to link with other table data. | Varchar |  |
| ID | Filepath of the media file. | Varchar | Primary, Foreign |
| Playlist | Name of the Playlist | Varchar | Primary, Foreign |

|  |  |  |  |
| --- | --- | --- | --- |
| Table name: Settings | | Table description: | Stores settings and a value for that setting. |
| **Field Name** | **Field Description** | **Data Type** | **Key?** |
| Name | Setting name. | Varchar | Unique |
| Value | Setting value. | Varchar |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Table name: **TrackedFolders** | | Table description: | Stores a list of folder paths that the file scanner should scan. |
| **Field Name** | **Field Description** | **Data Type** | **Key?** |
| Folderpath | Folder path. | Varchar | Primary |

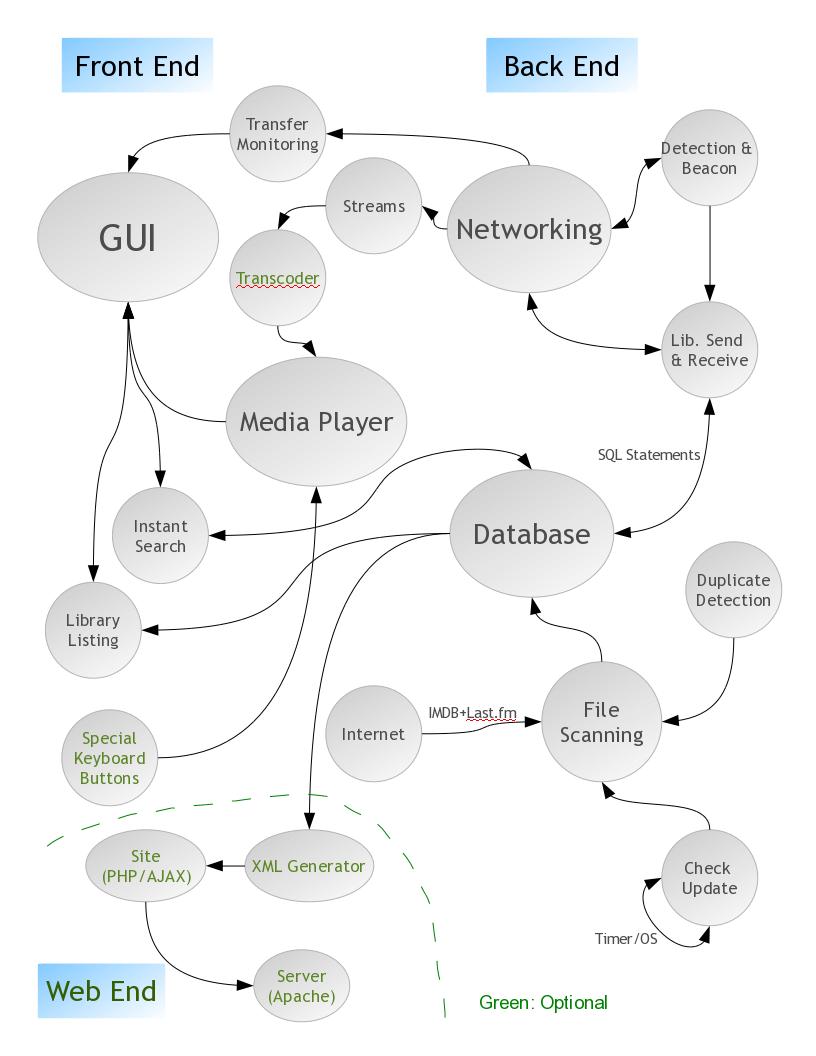
|  |  |  |  |
| --- | --- | --- | --- |
| Table name: **ExcludedFolders** | | Table description: | Stores a list of folder paths that the file scanner should never scan. Used when you want to scan a folder, but ignore a certain sub-folder. |
| **Field Name** | **Field Description** | **Data Type** | **Key?** |
| Folderpath | Folder path. | Varchar | Primary |

# Implementation & Testing

## Implementation Overview

The initial idea for the implementation of our project was to have a clear design of all stages and modules of the program and after to simply implement them. In the first few group meetings we defined what the main features of the program would be. The whole program is divided into two main parts, which are the front-end and back-end. The front-end contains for everything that the program will display, in this case that is the program’s GUI. The back-end is composed of the modules that drive the GUI and work behind the scenes, the two main modules being the database, file scanning and the networking modules. Following our design, these are hidden from the user in order to keep the program more user-friendly.

The following diagram demonstrates our initial implementation plan, with the rough classes shown as circles, and the arrows showing how they are connected to each other:



For the Database we chose to use SQLite. SQLite stood out because it doesn’t require the user to install any extra database software, is fast and is contained into a single file. The file scanner populates the database with information about the user’s music and video library, including media tags and file paths. The networking module was devised from the ground up by our group, and allows for the sending and receiving of beacons as well as the syncing of media libraries.

## Software Maintenance

To make it easier to work on numerous parts concurrently, Streamberry was designed to be built in a modular manner from several smaller parts, all interconnected. The goal that we set for ourselves was to have a working back-end of the program for the Alpha release. Therefore the first major program modules that were implemented were database, networking, file scanning and media playing, which were done simultaneously.

As we started the project we were planning to use the Java language. When we came across libVLC, it seemed like the best choice of media library, but there were no Java bindings available for Mac OSX. Rather than use a less proficient media library, we decided to switch to C++ language instead, as this supported libVLC on all platforms.

Paired programming techniques were used for networking and database, as with two people you are more likely to have errors highlighted, especially on a difficult task like networking. The file scanner and media player parts were built separately.

After the Alpha release, we were hoping to be able to integrate all of the modules together, but at this stage it was clear we had over looked a lot in the design of the program. We had all written parts separately and a lot of extra coding was needed to make them interact with each other. While these parts were being rewritten, Robert and Yordan started on our next goal, which was to have a fully working GUI; ready in time for the Beta release. As explained in the GUI development section below, this was actually not ready in time until Gamma release, with only a prototype in place for Beta.

The Beta prototype was enough to allow integration of the back-end however, and so after this deadline we were able to work on properly integrating the back-end into the front-end. At this point we started on the task of playlists. We had overlooked this rather vital feature before, and so a lot of restructuring was required in the GUI, especially in the library view.

This was also the first point where the libVLC player was integrated into the main program and taken away from its own proof of concept program in its own separate Git branch. Local playback was not too much of an issue, although perfecting streaming took up until Gamma release.

During the Gamma phase several major components of Streamberry were completed, including the managing and displaying of playlists. A few major changes to file scanning and the database had to be performed due to some serious bugs in library syncing.

Considerable development time was also invested in the automatic album art downloading class; this never made it to release. The main reason that album art failed to meet standards was that it had been designed incorrectly and had no way of coping with hundreds of pictures needing to be downloaded at once. As the user would have scrolled through the library the class would have crashed or hung under the extreme demand. The entire class was rewritten for the final release, and now works as intended, but due to complications in integration it still remains to be seen whether album art will appear in the final release or not.

Around the time of the Gamma Release we also instantiated Bug Genie, as mentioned in more detail later, and this helped us coordinate our bug-fixing efforts. As a result we reached the Gamma release with quite a stable build.

As the final release approached we knew we had to give the almost finished program to testers. There were still quite a few vital features that needed to be implemented for the program to be test-ready however. With time running short, extraneous features were cut from the plan and we worked full out on those elements that provided basic and essential functionality, like adding tracks to playlists, setting your music folders, shuffling the tracks as they played. Up until this point we had only worked towards the Alpha, Beta and Gamma milestones but due to the large amount needed to be done in such a short space of time, we decided that we would design an agile development plan. This was semi-successful and helped us get the program ready for our user testing on time.

For the final release, the SQLite database was restructured and made much faster and file scanning was also optimized. The GUI was completely finished, having context menus, a settings panel, dynamic preview panel, intelligent playlists, video streaming and more.

Once given this version of the program testers gave us valuable feedback on usage issues as well as bugs. When a bug was found, the person that had worked on the particular method causing it was notified and it was left to him to fix. Overall this proved to be a good technique, resulting in the discovery of about ten bugs, which were successfully dealt with from user testing alone.

## Graphical User Interface (GUI) Development

Over a couple of group meetings, the design of the GUI was discussed and a mock-up was drawn.

GUI1

From there onwards the GUI was thoroughly designed and thought out. We worked both on paper and in Adobe Photoshop to visualise and mock-up different GUI designs. This rapid drafting allowed us to explore how different approaches would look. Here is the first digital ‘flat’ of the GUI:

GUI2

After going through several changes in the design and layout of the GUI we settled for a GUI design as shown below, which we implemented into the beta build.

GUI3

We really liked the sleek look of the GUI; it was simple, clean and modern looking. The GUI was split in several sections, which were separately implemented. By having the GUI divided into parts, we could ensure every object had the designed functionality in a more organized way. The main sections were the LibraryController, the Sidebar, the Play-Controls, the top bar and the menu bar. The Library controller was responsible for the filling of the track table in the central space, as well as managing the back and forward views and the current ‘play queue’. The sidebar handled general navigation and playlists. The GUI proved to be the most time consuming part of this project, because it was it was the glue that bound all of the backend modules together. There was a huge amount of communication between GUI elements and the backend, even across threads. Qt helped in this regard, and their robust Signals and Slots mechanism made cross-thread communication possible without too much difficulty.

While the GUI was coded into the main program build all of the back-end algorithms were also revised in order to meet a certain requirements of the GUI. Most of the components also need to  be connected to the GUI or optimized in some way so the GUI didn’t hang when a backend function was run. The first streaming methods were implemented and connected to the rest of the program too, allowing us to finally access the ‘core’ functionality that Streamberry aimed to provide and begin in-house testing of the networking aspects of Streamberry. The Beta release met the set goal of having a GUI and also we achieved implementing basic streaming and playing of media files.

After the beta release we were still not content with the GUI however, as the GUI is is most vital part of the user experience of our program and so kept experimenting. One of the main things we found that separates a bad GUI from a good GUI is the size of the text. As we began to use smaller text for our titles, the GUI looked better, and, along with a slight rearrangement of the play-controls and the progress bar we came to a design we were happy with. For the Gamma release we redesigned and implemented the GUI, with new features such as new playback controls, a new library and generally a more polished feel.

Below is a picture the final GUI design:

GUI3

## Algorithm Descriptions

### Networking Beacons

A core component of Streamberry is the ability to see other people’s media libraries and be able to play their media. To enable this, each client needed to be able to communicate with clients running on other machines on the local network. We decided not to create separate client and server programs, as this would make it more complicated for the end user and increase the barrier to entry. All clients must communicate frequently to check that others are still online, but they must also occasionally send large amounts of data when libraries need to be synchronised. As such, we decided to split the networking module into a component that would deal with bulk transfers and another to handle frequent communication; it is this component we shall discuss here.

The component is responsible for allowing clients to detect when a machine comes online and maintaining a list of all currently online machines. It also has responsibility for ensuring libraries are kept up to date. Our design uses beacons that are frequently sent out by all clients that all other clients can detect, we implemented this using UDP multicast. The beacons are sent out every x seconds depending on when the program was launched, beacon sending is not synchronised between clients as this would cause large amounts of network activity at regular intervals if the number of clients were scaled up.

Normal beacons contain 3 pieces of information and a header that identifies them as belonging to Streamberry. The information included is the unique ID of the machine, a database timestamp and the IP address of the machine. When the client is closing, Streamberry sends a beacon with a different header indicating to all receiving clients that the machine should be marked as offline.

When a client receives a beacon, it checks which type of beacon it has received and takes the appropriate action. For a normal beacon, the client checks that it already knows the sending machine is online using the unique ID and the list of currently online machines, if not then the sender is added to the list of online machines and set as online in the database. The receiver first checks that the sender already has a database entry and creates a new one if not. Once the sender has been set as online in the database, other components will be used to perform actions such as displaying the other machine’s media library in the GUI. The receiving client then checks the timestamp in the beacon against the one stored in the database for the sending machine. This timestamp indicates when the sender last updated their library. If this is different to the one stored by the receiving machine then the library requester is invoked, but that is a separate communication. Finally, the receiving client updates the list of online machines with the current timestamp, indicating when the sending machine was last seen. If an offline beacon is received, the receiving client removes the sender from the list of online machines, sets the machine to offline in the database and takes no further action.

The receiving client periodically checks for timeouts using the list of online machines and the timestamps it contains. If a supposedly online machine has not been seen for some time, it is removed from the list and set offline in the database. The timeout value is set to just over the time between 2 beacons, this means that if a client doesn’t receive any beacons from another machine in the time it should have received 2 (taking into account network latency), then the machine will be believed to have gone offline.

Both the sender and receiver are in their own threads, this is to allow them to work concurrently with each other and other components. This is necessary as both are required to perform actions regularly without waiting for another component.

The beacons algorithm, along with all networking code, changed quite a lot during development. Initially the whole networking module was written in Java, however had to be changed to C++ when we decided to switch languages. In addition C++ implementation had to be rewritten in around February, mostly due threading issues. Both of these rewrites were mainly debugging and few changes were made to the algorithms themselves, though each component was implemented in a way that allowed greater concurrency.

### File Scanner

The file scanner is initiated with the ‘scan’ method. The file scanner then pulls the list of folders to scan from the database, stores them in an array and then recursively scans through them one at a time. In every folder it looks at each file individually, checking their file types are a media file type accepted by the program. If the file type is acceptable then it stores the file information, such as file name, tags and media type (music or video) into a buffer.

Once the buffer size reaches ‘BUFFERSIZE’, currently set at 1500, or the file scanner has finished, then it loops through the list adding each file to the database. It is more efficient to buffer the file information instead of adding one at a time as SQLite gives huge speed improvements when performing actions in bulk. The files are added to a temporary table the database with a simple SQL ‘INSERT’ statement in the database class.

Once all the files have been added to the temporary table in the database, it is now necessary to copy them into the main table where the local media is stored: ‘LibLocal’. The reason for storing the files in a temporary table is that files not found in the scan can be marked as deleted in the media library. Therefore the library always shows an up to date list of files in the folders you specify for a scan. The files cannot be deleted from the database, and must only be flagged as deleted, as they must be remembered so that they can be deleted from remote libraries while synchronising.

After marking missing files as deleted, updated records are deleted from the media library and then all records are copied across from the temporary table into ‘LibLocal’. All of the temporary tables can then be deleted. Finally, the library controller is alerted to the fact that the library has changed, and refreshes the library list from the database.

## Code Examples

### Playlists

Another key piece of functionality in any media player is being able to organise your music in playlists, allowing for quick access to a selection of music. Streamberry supports two types of playlists, normal playlists (explicit playlists) and filter playlists. An explicit playlist is created by making a new playlist and then adding single or groups of songs to it via a context menu. A filter playlist is essentially a saved search and works via the same mechanism as the search bar present at the top of the GUI.

Playlists are stored in the database, across two tables:

|  |  |
| --- | --- |
| Table: **Playlist** | **Field Description** |
| **Name** | This field stores the name of the playlist, which is not only used in the GUI but also acts a primary key for uniquely identifying playlists. As such it is impossible to name two playlists the same. |
| **Smart** | This is used to tell whether the playlist is a filter playlist or not. 1 for filter, 0 for normal |
| **Filter** | If the playlist is a filter playlist then this string contains the search terms required. |
| **Played** | This contains a timestamp that stores when the playlist was last played or saved, allowing the Recent Playlists section of the GUI to display the top five. |

|  |  |
| --- | --- |
| Table: **PlaylistTracks** | **Field Description** |
| **UniqueID** | This stores the uniqueID of the track in the playlist |
| **ID** | Stores the filepath of the playlist track |
| **Playlist** | Stores the name of the playlist that the track is in |

The playlist class has fairly obvious functionality, and acts mainly as a wrapped for more complex database functions. The main actions you can perform on a playlist are: Making a new playlist, saving the playlist, deleting the playlist, adding tracks, removing tracks, getting all tracks and altering the database fields, such as name, filter and type. I will go through roughly how these operations work below:

When a new playlist is created, of either type, a new Playlist object is created and is passed the new playlist’s name. The plnormalnew dialog box then saves the playlist, which adds a new record to the database and enters the playlist name:

void Database::PlaylistSave(QString name, int smart, QString filter)

{

QString time;

time.setNum(Utilities::getCurrentTimestamp());

QString value;

if(smart == 1)

value = "1";

else

value = "0";

QString sql = "INSERT OR REPLACE INTO Playlist (Name, Smart, Filter, Played) VALUES (\"";

sql += name; sql += "\","; sql += value; sql += ",\"";

sql += filter; sql += "\","; sql += time; sql += ");";

query(sql);

}

The code above shows how a playlist is saved into the database, with a dynamically built SQL statement in the database class. The new playlist then appears in the sidebar (update\_sidebar is called as the new playlist is created) and becomes playable. If you made an explicate playlist it will initially be empty, but right-clicking any track gives you the option to add it to a playlist. The addTo class is used to create a menu displaying all current playlists and controls adding tracks to this playlists. When initialised it connects to the database and draws a list of all playlists, it populates its menu with these playlists and connects the triggering of each menu item to a slot. Then, when a track is right-clicked, AddTo is updated once again, with a list of all the currently selected tracks. This allows it to know which tracks should be added to the playlist once a playlist is chosen. Finally, the playlist name is clicked, and the selected tracks looped through to add them to the playlist.

A code snippet showing setup is displayed below:

…

for(int i=0; i<playlistnumber; i++)

{

if(playlists->at(i).field(1).value().toString() == "0")

{

menuitems[i] = this->addAction(playlists->at(i).field(0).value().toString());

menuitems[i]->setObjectName(playlists->at(i).field(0).value().toString());

QObject::connect(menuitems[i], SIGNAL(triggered()), this, SLOT(Clicked()));

}

}

…

The code above shows the loop that runs when AddTo is updated. The number of playlists is held by playlistnumber. This creates the correct number of QMenuItems to populate the menu and names each one. The naming of each item is very important, so that later, when the QMenuItem is clicked, the slot function can identify the sender of the signal. Finally a connection is set up between the newly created QMenuItem and its slot handler.

Once the playlist has been populated it can be played by double clicking on it in the sidebar. A single click displays the playlist contents in the library view, this works via a specially filtered search. Once viewing a playlist the track-context menu gains another function, the ability to delete a track from a playlist. This works in exactly the same way as adding a track, except a different database function is called in the end.

Other playlist functions are accessed through the playlist context menu, e.g. renaming, deleting and editing (for filter playlists only). Calling up a context menu is a simple affair when using Qt; firstly you change a property of the table that holds the playlists in the GUI to cause it to display a custom context menu when right clicked:

playlistTableWidget->setContextMenuPolicy(Qt::CustomContextMenu);

QObject::connect(playlistTableWidget, SIGNAL(customContextMenuRequested(const QPoint&)), this, SLOT(ShowContextMenu(const QPoint&)));

Then you hook up the custom menu signal to a slot, which in turn will execute your newly created context menu class, in this case PlaylistContext.

As I mentioned above, some playlist functionality is provided using dialog boxes; since they are much the same I will go through only one of them here.

PLSmartNew is the class responsible for displaying a dialog box capable of creating a new filter playlist. Qt makes the creation of dialog boxes quite easy, although they still require as many elements to set them up as a more complex GUI. For example, despite only using four child-widgets a Grid Layout is still required to encase them all. When the user selects “New Filter Playlist” from the edit menu, the dialog box is set up and then displayed. Once text has been entered in both fields pressing enter or clicking ok calls the btnClicked() slot which closes the dialog box and saves the playlist.

### Networking Beacons

This is an extract of code from the BeaconSender constructor (the source code for the whole class is in the appendix):

timer = new QTimer(this);

//connect the timer signal to the send slot

connect(timer, SIGNAL(timeout()), this, SLOT(send()));

//tell the timer to fire every 5 seconds

timer->start(5000);

The first line of the code creates a timer, this timer will emit a signal every so often (this time has not been set yet). The signal the timer emits is then connected to a slot in the BeaconSender called send(). This means that every time the signal is emitted, the code within send() will be run. The timer is then started and made to emit the signal once every 5 seconds. The send function itself is quite basic, it merely gathers the necessary data, puts it all in a string and sends the string out to all other machines on the local network using UDP multicast. The structure of Streamberry beacons is "STREAMBEACON|<unique ID>|<database timestamp>|<ip address>" or "STREAMOFFLINE..." for offline beacons.

The BeaconReceiver also has a timer, when its signal fires a check is run to detect machines that have timed out. This code snippet is from the BeaconReceiver constructor:

//Creates a UDP socket and sets it to listen for streambeacons

udpsocket = new QUdpSocket(this);  
udpsocket->bind(45454, QUdpSocket::ShareAddress);  
connect(udpsocket, SIGNAL(readyRead()), this, SLOT(processPendingDatagrams()));

This code creates a new UDP socket and binds it to any host address on port 45454 using the bind mode “ShareAddress”. This means that other services can bind to the same address and port, allowing the BeaconSender to send beacons on it too. The UDP socket will emit a signal when there is a datagram ready to be read. The connect command ensures that when this signal is fired, the code in processPendingDatagrams will run, which will deal with the datagram.

QByteArray datagram;

datagram.resize(udpsocket->pendingDatagramSize());  
udpsocket->readDatagram(datagram.data(), datagram.size());  
QString datastring = (QString) datagram.data();

This is the code from processPendingDatagrams which takes the datagram from the socket and extracts the relevant contents. A datagram object is created and sized so it can fit the contents of the datagram waiting on the socket, which is then read into the object. The string contained in the datagram is then copied into a QString object so the string can be easily accessed.

QString dbtimestamp = n.parsebeacon(datastring, networking::timestamp);

checkID(id, dbtimestamp, QHostAddress(n.parsebeacon(datastring, networking::ip)));

This code is from the same function, it shows how particular information is actually extracted from the strings contained in beacons. The first line extracts the timestamp from the beacon using a networking object (n) and its parsebeacon function. The checkID function takes the extracted information from a beacon and performs all the necessary checks and updates. The next piece of code is from checkID:

QString lasttimestamp = db.lastUpdate(id);

…

//if their timestamp is newer, get new one

else if (lasttimestamp.toInt() < dbtimestamp.toInt())  
{  
 getLibrary(theirip, id, lasttimestamp);  
}

This code gets a timestamp from the database, which indicates the version of the other machine’s library we have. Whenever a library is updated, a timestamp is stored in the database and sent in beacons. Each machine stores a timestamp for each other machine, allowing them to detect if their version of the other machine’s library is outdated. The if statement in the above code checks this, lasttimestamp is the timestamp the current machine has stored, while dbtimestamp is the one received in the beacon. If the library is out of date, the getLibrary signal is emitted which instructs the LibraryRequester to obtain the new version of the library.

### Library View Searching

The search bar at the top of the gui acts as a direct method of input to one of the most powerful functions we use in our program, the searchdb function from the database class. See Appendix for the full code. This function, although very long, performs one simple function. It takes a string of search text and returns the set of tracks that match that search text in either their Artist, Title, Genre or Album field.

In actual fact searchDB takes several arguments:

QList<QSqlRecord>\* Database::searchDB(int type, QString playlist, QString searchtxt, QList<QString>& sortcols, QList<QString> order, int musicorvideo);

Firstly, you must pass an int determining the type of search you wish to run. A type 0 search runs on all fields listed above however a type 1,2,3,4 and 5 search runs on Artist, Title, Genre, Album and Length respectively. If a playlist is passed in as a string, then the search will be narrowed down to tracks within that playlist. This is in fact how the library controller displays playlists, by running a search with “” as the search text and PLAYLIST as the second parameter; therefore returning all the tracks in the playlist.

The next parameter is the search text, which as I mentioned is used to find the matching tracks from the database. The actual comparison of searchtext and field contents is done using the SQLite keyword LIKE. LIKE allows an SQL search not to be completely exact, through the use of wildcards. Therefore a search will return a track called “Never Gonna Give You Up” when the search text is only “Give You” because it is surrounded on either side by two wildcard strings.

An example SQL search statement is shown below:

"…. AND (Album LIKE \"%SEARCHTEXT%\") … "

The next parameter is a list of strings which define how the resultant search should be ordered; these strings must correspond to fields in the track sql record. The sorting is done automatically by SQLite with the aid of the fifth parameter, order. Order is another list of strings that dictates the order in which the fields defined by sortcols should be sorted; essentially either ascending or descending. For most purposes library search is sorted by Album descending, to group the tracks nicely.

Finally, search requires another *int* to tell it if a media toggle has been turned on or not (using the large buttons at the top of the GUI). If 0 then both music and video files are returned by the search, if 1 then only music and if 2 then only video.

Throughout searchdb a large SQL statement is gradually built up, containing elements relating to all of the parameters. When the SQL string is finally constructed it is executed by Qt using the query function in the Database class. Query returns a list of QSqlRecords containing all the tracks that matched the search parameters; however the job is not yet complete.

The SQL records require post-processing to remove any escaped characters and this is made more complex than a ‘find-replace’ through the record fields by the fact that any SqlRecords drawn by the database are ‘const’ and therefore read only. There is no quick way around this issue, it is built into qt. To resolve the situation, any records with fields containing unescaped characters are copied, the changes made, and then inserted back into the list of QSqlRecords ready to be returned by the function.

Below is an example of this procedure:

QSqlRecord copy(result.record());   
if(result.record().field("Filepath").value().toString().indexOf("\\;") != -1)  
    {  
      const QString fieldname = "Filepath";  
      QSqlField temp(fieldname, QVariant::String);         temp.setValue(result.record().field(fieldname).value().toString().replace("\\;",";"));  
      int pos = copy.indexOf(fieldname);  
      copy.remove(pos);  
      copy.insert(pos, temp);  
    }       
       files->append(copy);

Back/Forward views:  
The back and forward buttons situated at the top-left of the GUI allow easy navigation through the views you have recently looked at. The library view operates entirely using searchdb to fill its table. When ‘all media’ is being displayed a blank search can be run; when ‘video’ is toggled a video only search can be run. Since library controller operates using searches only, it is possible store the search parameters and therefore be able to recreate any past view. The back/forward mechanism stores the search parameters in specialist structs called ViewQueueItems.

typedef struct {  
    QString playlist;  
    QString playlisttitle;  
    QString searchtext;  
    QString smarttext;  
    int videoview;  
    QList<QString> sortcols;  
    QList<QString> orders;  
 } ViewQueueItem;

As you can see, a ViewQueueItem contains strings for storing the searchdb parameters, as well as two new fields; playlist and videoview. Playlist stores the filter for filter playlists, and the blank string if the view is not a filter playlist. Videoview is an *int* that is set to 1 when the VLC video display pane is showing and 0 when it is not.

The viewqueue operates by keeping list of ViewQueueItems and a viewqueueindex, an *int* representing where in the viewqueue the user currently is. The whole system works quite intuitively just like an internet browser. As the user ‘creates’ views, those views are saved in ViewQueueItems and the viewqueueindex is incremented. When back is pressed then the viewqueueindex is decremented and the librarycontroller grabs the previous search parameters from viewqueue[viewqueueindex], runs the search and displays the results. If a new view is then ‘created’ any forward-history is deleted and replaced by the new set of views. If a new view is not ‘created’ then clicking the forward button will move you back up the views list until you reach the most recently saved view. When the view is changed, either the relevant searchtext will appear in the searchbar, or the relevant item in the sidebar will be highlighted, making it easy for the user to tell what view they are currently looking at.

It is worth noting that although the music/video toggle buttons at the top of the GUI can be saved as search parameters, in the case of forward/back they are not. This is because we wanted them to act more like filters to your current content, and not as defining ‘categorisers’.

Below is a code snippet showing the goBack() function, as explained above, that is called when the back arrow is clicked:

void LibraryController::goBack()  
{  
 if(viewqueueindex>0)  
 {  
    viewqueueindex--;  
    emit setSearchBoxText(viewqueue[viewqueueindex].searchtext);  
    emit setSelectedPlaylist(viewqueue[viewqueueindex].playlisttitle);  
    updateLibrary();  
 }  
}

For the ease of the rest of the program, library controller also stores an *int* representing the state of the current view. Other classes can then request this *int* so they know how to react to the content in the main screen. A viewtype of 0 means that the library is displaying all media, a viewtype of 1 means a normal playlist is being displayed, and a viewtype of 2 means a filter playlist is in the library table.

## Versions & Bug Tracking

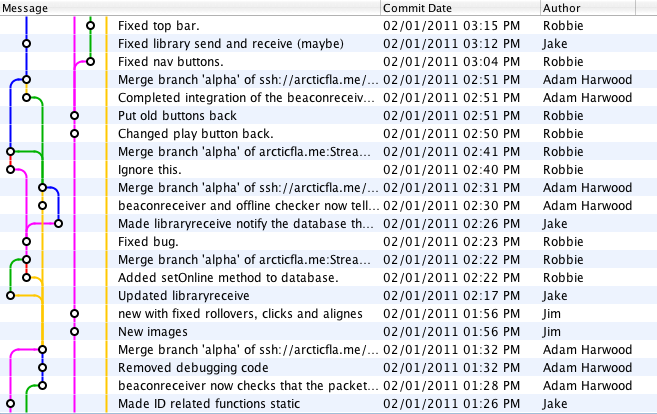
We used the Git version control system to track versions of the program as it has many useful features, especially when compared to Dropbox or email as a method of version control. We used the free program SmartGit as a graphical front-end to Git, which we feel made it a lot easier to grasp.

At the start of the project Git allowed all members of the team to work in separate branches. This was beneficial as we could all see each others’ progress without affecting each others’ code. When it was necessary to share branches, Git often handled merging of code automatically meaning several people could work on the same classes in parallel with ease. Should conflicts occur they are usually easily solved in SmartGit with the manual merge user interface.

As Git is distributed we could each work on our own local copy of the repository. By keeping the version on the server relatively free of bugs, we could each develop on a working version of the repository, without introducing bugs on other peoples’ copy of the program. After performing unit testing on our local copies, we could push the latest bug free version of the program to the server. This also gives the advantage of always being able to run the latest stable version; a feat almost impossible when sharing code via email.

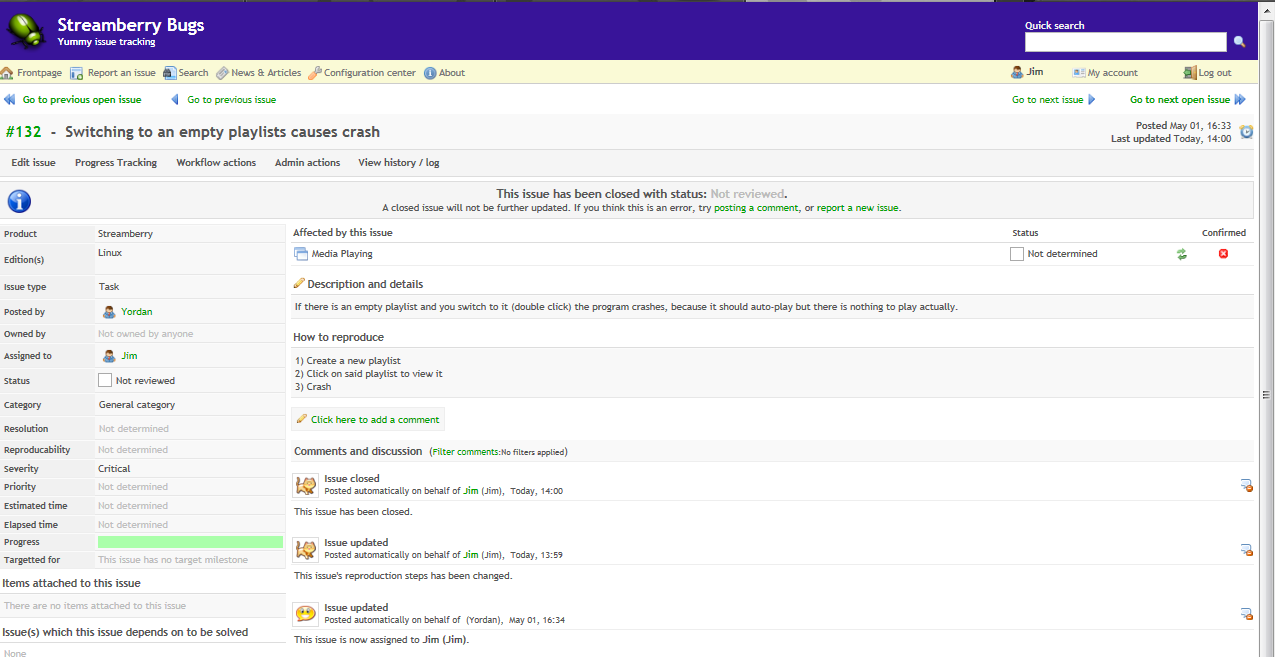
Should bugs happen be introduced, the repository has a complete history of every commit and so it is possible to see what has changed in the past and restore the program to a working version if necessary. The ‘bisect’ feature of Git helps with this, allowing you to temporarily revert to previous versions of the program, moving forward through each commit until you find the version that introduces the bug. Another favourite feature amongst the group is ‘blame’ which instantly specifies the user account that committed a specific line, allowing us to know who was responsible for writing a piece of code so we would know who to query if we had any issues.

No members of the group were previously familiar with version control systems or working on large projects as part of a group before this project, and so there was a period of adjustment as we learned the best practises of using Git and developing in teams. We found it is important to ‘pull’ the latest version of the repository often, you may write code that interfaces with an old version of the program resulting in you needing to rewrite the code. Also merge conflicts will often occur that, while they are fairly easily solved, they need a certain amount of interaction between the group members which is time consuming and frustrating.



*An example selection of commits to our Git repository*

At the early stages of the program we did not have a dedicated bug tracking system. At that stage it was unnecessary, as we were coding in separate Git branches and so rarely interacted with other group members’ code. Any issues we did have were discussed in meetings, on our Google Group or over Skype. As we began to merge our branches into a main branch for the program, we started to find other peoples’ bugs while using the program. We decided to use a web-based piece of software called  ‘The Bug Genie’ to track our bugs. This allowed us to post bugs and tasks on the system, with different priority levels, and assign them to users. Group members could then check for bugs assigned to them and fix them. Below is a screenshot of an example bug post in Bug Genie.



## User Testing

Nearer the end of the project we performed usability testing on two groups of users from our target audience, divided into two households. This type of testing proved to be invaluable as it showed how real users use the program, giving us feedback on features they liked and features we needed to improve on, and whether they feel it worked as expected.

The feedback from the testers was generally good, saying that the program looked ‘attractive’ and that the main feature of seamless library syncing was useful. The testers also gave beneficial feedback regarding several features of our program, for example commenting on how the file scanner gave no feedback and so there was no way to tell whether the file scanner was working or not. Following this feedback we implemented a progress bar to alert you when the file scanner is running.

The testers mentioned several features we had over looked too, such as the filtered playlists and a first run wizard. We decided to hold two rounds of testing for this very reason, so that we could get feedback and ideas for essential missed features from the first round of testing, implement these features and then get feedback from the second round, allowing us to tweak the features to our users’ needs. The second round therefore acted as our user acceptance testing.

As we had a fairly small number of testers, we were able to write down their feedback on paper and then submit their requests and findings onto our bug tracking system. We asked users to keep a note of issues they came across while using the program.

## Tests

Testing is an important part of the development process as it helps provide confidence that the program is working as intended. We did a lot of testing while implementing new features, but at the end of writing the program we devised a set of tests for software verification, and failure, load, system and functional testing. This allows us to prove that the system meets the original specification, whilst allowing us to detect bugs and eventually prove that the tasks performed in our tests are bug free.

The test set and the results of these tests is below.

|  |  |  |
| --- | --- | --- |
| **Test Description** | **Type** | **Result** |
| Discovering the maximum number of tracks the library can hold. | Load | On a moderately equipped laptop, the program reached breaking point at 250,000 tracks in the library. The load time for the program increased dramatically from about 6 seconds to 3 minutes. That said, despite these loading times, once the program had loaded it ran very smoothly with good responsiveness, and playback and other features worked well. The limitation therefore appeared to be the machine and not the program. The program was using around 1GB of RAM; with more I believe the program would be able to hold an endless number of tracks, although I would expect the load times to continue to increase dramatically, at some point causing the program to become unusable. |
| Running stability | System | Using a typical set up of the program simulated constant running of it for 1 hour. The result from this test was that after 1 hour of usage the program kept stable and responsive to commands. With about 1000 songs and videos the program used an average of 70 MB of RAM. |
| Playlists functionality | Functional | The program supports creation and management of custom playlists, which can be populated with media from the main user library either one at a time or as a group. |
| Mixed Playlists | Functional | The playlists support popular audio and video files. It is also possible to have them in a custom playlsit. However when the program finishes playing a video it doesn’t go back to the playlist. This issue can be solved by just selecting the playlist again. This bug was assigned as not critical to the whole program and because of that is not yet resolved. |
| Running on different OS | System | This test was in heart of our testing scheme. The set up for it includes running the software product on the three supported operating systems - Windows, Linux, Mac OS X on a test local network. Beacons were successfully sent and received by all users. The libraries were successfully synced. And finally music and video streaming was successful and fast enough to provide the user with pleasant media experience. |

## User Guide

Please see the user guide in the appendix.

# Project Management

## Original Schedule Recap

The schedule we set out back in December broke the implementation of Streamberry down into several medium-sized tasks, which would act both as a guide of what to do and a guide to our progress. There were thirty-four of these original tasks, which are listed below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Task No.** | **Task** | **Task Description** | **Time** | **Completion Date** | **Assigned to** |
| 1 | User guide | Write up a basic user guide to go with the alpha release | 5 | 04/12/2010 | Adam, Jake |
| 2 | Database | Implement the fields, database structure in SQLite | 1.5 | 29/11/2010 | Robert, Jim |
| 3 | Database | Implement the class for managing the SQL database: | 1 | 14/11/2010 | Robert |
| 4 | Database | SQL queries, DB connection | 5.5 | 14/11/2010 | Robert |
| 5 | Database | Creation of database | 1 | 14/11/2010 | Robert |
| 6 | Database | Storing, getting of settings | 1.5 | 14/11/2010 | Robert |
| 7 | Database | get + set folders | 1.5 | 14/11/2010 | Robert |
| 8 | Database | Delete record | 1 | 14/11/2010 | Robert |
| 9 | Database | Update Files | 1 | 14/11/2010 | Robert |
| 10 | Database | Get changes since a certain time | 5 | 21/11/2010 | Yordan |
| 11 | Database | Get last update time | 1 | 21/11/2010 | Yordan |
| 12 | Beacon (send, receive) | Implement the class that sends a heartbeat beacon, picks up other heartbeat beacons. Interfaces with the Database class to update the LibIndex to represent online clients. List of computers online | 15 | 14/11/2010 | Adam, Jake |
| 13 | Beacon Detection | Implement the thread that listens for beacons, responds | 5 | 14/11/2010 | Adam, Jake |
| 14 | Library send | Implement the class that listens for requests for it's library, transforms database into a series of SQL statements representing the changes made since last request date & time. Also can send a complete copy of library. | 3 | 14/11/2010 | Adam, Jake |
| 15 | Library receive | Implement the class that sends out requests for libraries, receives information, updates libraries using SQL statements | 6 | 14/11/2010 | Adam, Jake |
| 16 | Media Playing | Understand, how to play a media file in VLC media player, proof of concept it | 8 | 01/12/2010 | Vitaliy, Yordan |
| 17 | File transfer | Implement the class that streams a file of given id to a selected client, receives streams, monitors transfer rate | 6 | 20/01/2010 | Vitaliy |
| 18 | Tag reading | Implement a class that takes a file path, can read various tags from it in all the needed formats. | 8 | 01/05/2010 | Jim |
| 19 | File scanning | Implement the class that receives a folder path, scans it recursively adding media files to the database. Time to write Build Lib | 5 | 14/11/2010 | Jim |
| 20 | File scanning: | Scan Folder method | 4 | 14/11/2010 | Jim |
| 21 | File scanning: | Add Files method | 3 | 14/11/2010 | Jim |
| 22 | File scanning: | Verify files method | 2 | 14/11/2010 | Jim |
| 23 | File scanning: | Is Media method | 2 | 14/11/2010 | Jim |
| 24 | File scanning | Album art handling, cache | 5 | 05/01/2010 | Jim |
| 25 | Directory watch | Implement class that watches for changes to the folders scanned, updates the database with any changes made. | 6 | 20/01/2011 | Vitaliy |
| 26 | iTunes library import | Implement the class that can receive an xml file from iTunes, import the library into the local database | 5 | 20/01/2011 | Robert |
| 27 | GUI graphical elements | Design, draw the graphical elements in Photoshop | 15 | 25/01/2011 | Robert, Jim |
| 28 | GUI Coding | Code the GUI | 30 | 25/01/2011 | Robert, Jim, Yordan |
| 29 | Library display | Implement the class that displays the database libraries in an attractive, easy to view way | 15 | 25/01/2011 | Jake, Adam |
| 30 | Library searching | Implement the class that takes a string to search, displays the query results in the library pane | 15 | 25/01/2011 | Vitaliy, Yordan |
| 31 | Media player integration | Connect VLC media player to our GUI so it can receive a stream from another computer, play it | 8 | 25/01/2011 | Vitaliy |
| 32 | Internet tags | Implement the class that searches through the library, matches the files found to files found on last fm. Updates their tags with more correct/accurate information, provides album art. | 10 | 17/04/2011 | TBC |
| 33 | Web API | Design an implement the website that allows you to log in, stream music from your home network to your current computer. | 10 | 03/01/2011 | TBC |
| 34 | Report & User guide | Write up the user guide for the application | 30 | 20/02/2011 | All |

However, as will be commented on later, the schedule changed a lot during the implementation process as we not only discovered tasks we hadn’t accounted for at all, but also learnt a lot more about each task and found many of them needed to be broken down in smaller, sub tasks.

## Updated Retrospect Schedule

Below is a table showing an approximate breakdown of the tasks that resulted from the actual implementation. I have included the date on which these tasks were completed, though they were not necessarily running on schedule. The Task Category roughly divides the classes into groups based on the main classes they involved. The Task Description sets out what functionality is required by the end of each task. People Involved lists who contributed to each task; obviously in some cases the contributors worked for very different lengths of time, however, for completeness sake I have tried to include a name whenever someone worked for more than an hour on that task. Where possible I have tried to combine tasks for ease of reading, meaning there are twenty-two medium-large tasks shown below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Task Category** | **Task Description** | **Time Taken (hours)** | **Date Completed** | **People Involved** |
| Database | Implement the fields, database structure in SQLite and have it automatically created if it doesn't exist | 2.5 | 23/10/2010 | Robert, Jim |
| Database | Implement the class for managing the SQL database: Must ensure that no SQL is written outside of this class. Therefore, needs 'getters and setters' for pretty much everything in the database. Not to mention more complex functions relating to playlists and syncing. | 15.5 | 26/03/2011 | Robert, Jim |
| Database | Develop efficient and verstile function for searching the database based on Genre, Title, Artist or Album. Returns in any order or sorted | 17 | 17/03/2011 | Yordan, Robbie |
| Beacon (send, receive) | Implement the class that sends a heartbeat beacon, picks up other heartbeat beacons. Interfaces with the Database class to update the LibIndex to represent online clients. List of computers online | 21 | 24/11/2010 | Adam, Jake |
| Library send and Receive | Implement the class that listens for requests for it's library, transforms database into a series of SQL statements representing the changes made since last request date & time. Also can send a complete copy of library. Also implement the class that sends out requests for libraries, receives information, updates libraries using SQL statements | 20 | 15/01/2011 | Adam, Jake |
| Library Syncing | Merge two libraries together and display them as one in the GUI. | 23 | 16/01/2011 | Vitaliy, Adam, Jake, Robert |
| Streaming | Implement the class that streams a file of given id to a selected client, receives streams and directs them to the player | 36 | 05/05/2011 | Vitaliy, Adam |
| File scanning | Implement the class that receives a folder path, scans it recursively adding media files to the database. | 21 | 13/04/2011 | Jim, Robbie, Adam |
| File Scanning | Implement a class that takes a file path, can read various tags from it in all the needed formats. Should return at least: length, title, album, artist, genre | 3.5 | 16/12/2010 | Vitaliy |
| File scanning | Implement a method of drawing the album art for an Album from the internet and store the art in a database. | 25 | 04/05/2011 | Yordan, Jake |
| GUI Design and Graphical Elements | Create a full, high-res, digital mock-up of the GUI that can be split into elements and used as a programming guide. Also, design and draw the graphical elements in photoshop / vector graphics | 41 | 16/02/2011 | Yordan, Robbie, Jim |
| GUI Coding | Implement a sidebar which contains buttons allowing media navigation, a preview pane showing playlist information and album contents as well as album art. Also a list of the most recent playlists, controls for these playlists. | 49 | 05/04/2011 | Jim |
| GUI Coding | Implement a Settings menu for altering your media folders and a wizard for the first run that sets you up and runs a file-scan | 30 | 25/04/2011 | Adam, Jake |
| GUI Coding | Implement a class that displays all the media files in the library currently, maintains a list of what is playing at the moment. Maintains a history for back and forward purposes. Has context menus for each track allowing for more advanced options | 42 | 01/05/2011 | Jake, Robbie, Jim |
| GUI Coding | Basic GUI Structure/Layout. This includes overall layout, signals and slots and other miscellaneous tasks that do not fall into the other GUI related tasks | 44 | 05/05/2011 | Robert, Jim, Yordan |
| Media player Control | Implement buttons such as shuffle, repeat, play/pause, volume controls and such. Control next and previous, as well as the playing of playlists. | 45 | 27/04/2011 | Yordan, Robbie, Vitaliy |
| Playlists | Implement a class than allows the user to create playlists of tracks, either using a filter or manually. Associated controls such as modification, deletion… | 24 | 02/02/2011 | Jim |
| Media player integration | Set up vlc to player video in Streamberry. Either from a stream or a file, and across all platforms | 8 | 28/04/2011 | Vitaliy, Robbie |
| Media player integration | Connect VLC media player to our GUI so it can play local files | 7.5 | 13/01/2011 | Vitaliy |
| User guide | Write up the user guide for the application | 4 | 06/05/2011 | Adam, Jake |
| Thread Quitting | Figure out the order and method to quitting all these simultaenously running threads at the same time without seg faulting | 4 | 23/03/2011 | Adam, Robbie |
| Final Builds | Figuring out a way to build Streamberry on each OS, including statically linking all the required libraries. | 15.5 | 05/05/2011 | Vitaliy, Adam, Jake |

As you can see, some tasks we originally envisioned have been removed from this table, such as iTunes Library Import and Web API. These were both extra features we had considered including in the program if we finished early. However as we didn’t finish early, they were scrapped from the project. On the other hand, several new tasks have been added, specifically Playlist support, a better breakdown on the GUI elements and Library Syncing etc.

## Schedule Analysis, Actuality Compared With Intention:

Comparing the two tasks lists, it is obvious there is huge disparity between the predicted times and the actual times. For example, the schedule stated an estimated 15 hours for production of graphics for the GUI. However in reality a total of 41 hours was spent on the GUI images. This underestimation occurred because of two main things: Firstly we did redesign the whole GUI towards the end of January. This was a setback as the previous incarnation of the GUI was nearly complete, although from a design standpoint it had major flaws. With the second design of the GUI we aimed to achieve a very high quality finish, and as such we dedicated a considerable amount of time to drafting and redrafting the design until we were happy. Secondly, we had very little experience of the amount of time required to successfully ‘skin’ a GUI when we wrote the schedule, and as such, we underestimated the amount of work required to produce the high quality images we wanted.

In hindsight it was obviously a mistake to spend so much time creating graphics for the first incarnation of the GUI only to scrap it later on. More time finalising the design and creating quick digital sketches to ensure we were all happy with the look and feel would have saved many hours work.

Another area where we vastly under-estimated the time required was the coding of the GUI. The schedule states 30 hours allotted for the programming of the GUI, what seemed to us at the time as being a great deal of time. However, we spent around 190 hours in total working on GUI related tasks (excluding graphics production) which is 160 hours over estimate, a shocking mis-calculation. There are three reasons for this mistake: firstly none of us had ever programmed a GUI before, and thus we had to learn as we went. Although Qt is a very powerful and well documented library it still has a steep learning curve, especially when grasping the more complex GUI elements.

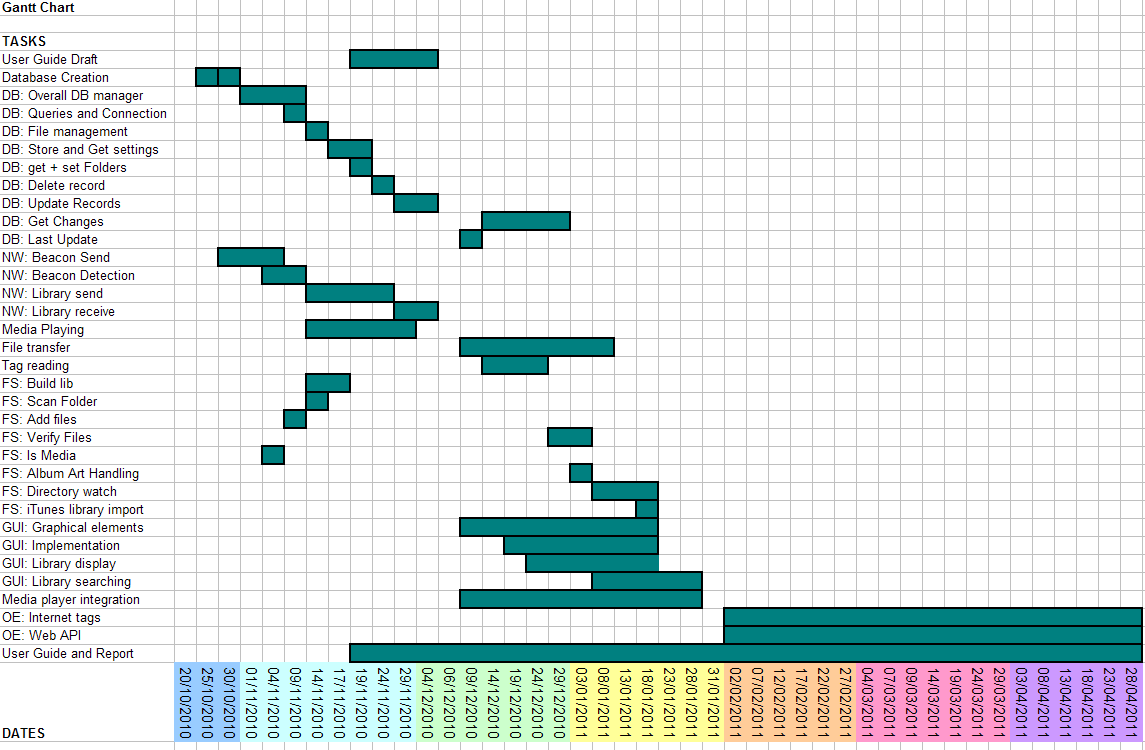
Secondly, we did not know exactly what the GUI would consist of before we began. As is evident in the original schedule, the entire development of the user interface was described as “Code the GUI”. With each stage of the GUI implementation we noticed a several new things we would have to implement in order to complete the GUI functionality. For example, although the Playlist class offered support for deletion of songs from playlists is was not until we actually got playlists displaying and playing in the GUI that we realised a context menu would be required in the playlist view, to remove songs from the current playlist. Upon completion of this context menu, we realised it was also necessary to have a similar right-click menu in the main library view, to make it easy to add files to a playlist. This constant expansion of the task before us meant it was difficult for us to predict how much work was still needed to complete the GUI, even as implementation drew to a close.

Thirdly, as before, we had no idea of what to expect when programming a GUI and our seemingly high estimate was pitifully low. Obviously, this has taught us all a great deal about how much work is actually required to create a fully functional GUI and if we embark upon another, similar, project we will be able to plan with much better estimates across the board.

In all fairness, I think we actually set our sights really high for this project. We naively underestimated the sheer scale of what we were attempting and I think, given the time constraints, we have done very well to finish Streamberry to the level that we have.

The Gantt charts below compare the original schedule to the actual achieved timings. Although the tasks from each version are not completely comparable, many of them are similar enough for the chart to show the clear difference between the schedule and reality:

### Original Gannt Chart:



### New Gannt Chart:

### 

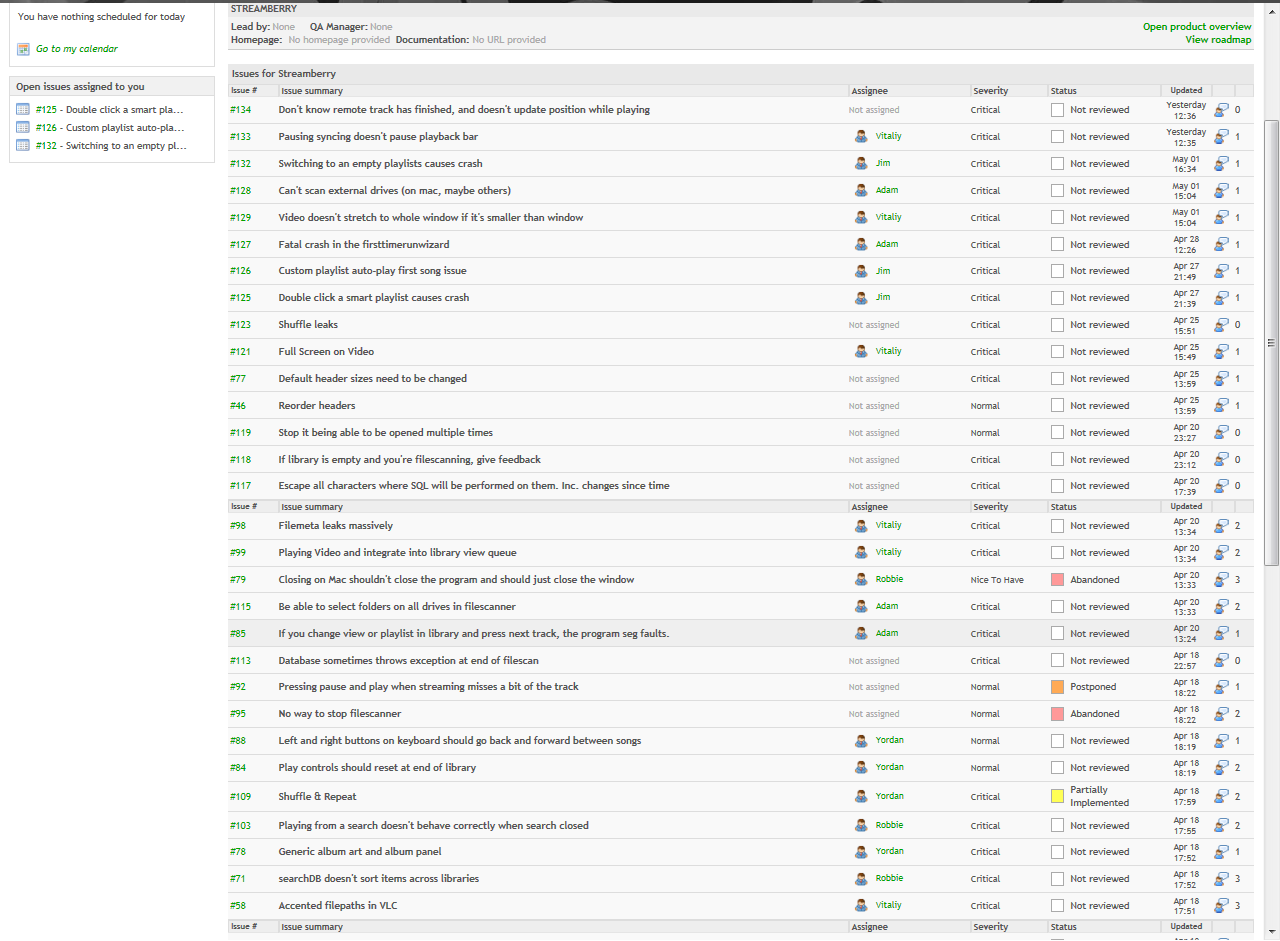
As you can see, the predicted Gannt chart and the actual Gannt chart are hugely different. This is due to two things, our underestimation of many ‘hours needed’, as I’ve said before, and also due to the way in which we worked on the project. Rather than working on a task, completing it utterly and then moving on and never working on it again, as the first chart suggests, we took a more ‘constant improvement’ approach. A class would be implemented, with its functionality drawn from the design. Then, as other classes were built and more functionality was required, we’d alter old classes to accommodate the new requirements.

This practice of constant modification continued through the whole development process. On one hand this method was absolutely crucial to the program’s success, because we did not have a crystal clear idea of what each class should entail and hence changes had to be made as implementation made it apparent what needed doing. However, had we planned better, perhaps we could have saved time by trying to complete each class entirely before moving onto another. Nevertheless, in my view, it would have been very difficult to approach the development in this way, as we were too inexperienced at the start of the project to understand exactly what each class would have to contain.

Overall, I do not think our original schedule was accurate enough to be helpful past the first few months. Due to lack of understanding of the task before us we failed to appreciate the time required for many tasks, and as such our original estimates were off. This had a knock-on effect resulting in the schedule becoming quickly out-dated; so I conclude the project did not run to schedule.

## Progress Monitoring and Communication:

Nonetheless, Robert did an extremely good job of keeping the group’s effort up, and as such we managed to maintain a good pace of work despite having no formal schedule to meet. He would set short-term deadlines at the end of each meeting, which gave us focus with our work. He was also responsible for setting up online bug tracking software, which, before we reached the testing phase, doubled as an online tasks list. As mentioned earlier, this enabled us to keep a log of what needed doing as well as giving us a sense of accomplishment when we could ‘close’ a task or cleared our assigned task list. Below is a screenshot showing how the tasks-list/bug tracking software appeared in the browser window:

****

The software we used, The Bug Genie, provided a great deal of functionality that allowed us to monitor our progress as we went; making sure that the most critical tasks were completed quickly.

We did not use only Bug Genie to facilitate group communication; we used Skype’s group chat and conference call functionality extensively, as well as setting a Google Group. The Google Group was extremely central to the project in the early stages, acting as a place for us to post questions or information as well as discuss the design of the project when we couldn’t meet in person. We used our group as a central repository of files before we got our version control system up and running. However, due to major changes that took place with Google Groups during February, we lost the ability to store files on the group as since then the group became much less important. This change in focus is also probably due to us setting up Bug Genie in February also, meaning we now had a new location to discuss the project and monitor progress.

As I mentioned, throughout the project Skype was the main form of communication between group members, allowing for quick questions to be asked, files to be transferred and also for group discussions. Setting up a group conversation in Skype was easy, and allowed us to hold conversations with all six group members simultaneously, either through text or voice. During the holidays we held group-meetings via webcam and this allowed us to simulate the usual meeting atmosphere, and encourage everyone to talk freely about how they thought the project was going.

We also held regular meetings or group programming sessions, where we gathered in labs to discuss program design or issues as well as to use our combined programming knowledge to solve difficult problems. Given that Streamberry is a network program and requires two computers at least to be network tested, it was in these group programming sessions that much of our bug-fixing and alpha-testing occurred. Below I have listed the minutes for some of our key meetings:

|  |  |  |
| --- | --- | --- |
| **Meeting Date** | **Attendees** | **Points Covered** |
| 14/10/2010 | Adam, Jake, Jim, Robbie, Vitaliy, Yordan | * First meeting where we brainstormed ideas e.g. * Phone apps, OCR food packet reader, web browser, wifi remote for media player, vnc, gambling bot, tournament organising, spotify powered media player, instant messenger with live media streaming, zero-conf media streamer… |
| 21/10/2010 | Adam, Jake, Jim, Robbie, Vitaliy | * Zero-conf media player decided as project * Streamberry & Sons decided as name * Streamberry is the program name * Drew diagram of possible features |
| 01/11/2010 | Adam, Jake, Robbie, Vitaliy, Yordan | * Meeting to finalise product description and outline pitch * Also worked out some GIT stuff and discussed how we would start implementation. |
| 11/11/2010 | Adam, Jake, Jim, Robbie, Vitaliy, Yordan | * Discussed tasks and times * Talked through some key processes in detail |
| 26/11/2010 | Adam, Jake, Jim, Robbie, Yordan | * Discussed plan for Alpha * Drew up a list of features that needed to be complete |
| 15/12/2010 | Adam, Jake, Jim, Robbie, Yordan | * Discussed how we would approach the GUI * How will we communicate over Christmas? * Prelim GUI designs drawn up |
| 28/01/2011 | Adam, Jim, Yordan, Robbie | * Meeting about GUI progress * Need to step up the work to make Beta come-together |
| 08/02/2011 | Adam, Jake, Jim, Robbie, Vitaliy, Yordan | * Meeting to see how everyone is doing and the general direction of the program |
| 24/02/2011 | Robbie, Yordan, Jim | * Meeting to discuss Gamma progress * After exceptionally tough Beta deadline we are behind again. Need to step up workload |
| 25/02/2011 | Adam, Jake, Jim, Robbie, Vitaliy, Yordan | * Everything is coming along, * Really nice progress made with the GUI * Streamberry is coming together * Going to do a mock weighting later in the day * Mock Weighting delayed because Jim had to go to Japanese * Networking is nearly there. Beacons seem to be working fine, except on Windows where offline beacons don’t send GUI coming along, hopefully in soon, but will be done by Gamma/ * Local playback is working, and parts of the next and previous are working. * Streaming soon. * Playlists backend is done and hopefully the GUI side will be in for Gamma. * Planning to have Gamma at a usable state so that we can do testing soon after Gamma release. |
| 04/03/2011 | Adam, Jake, Jim, Robbie, Yordan | * GUI almost finished * Only the playlists and currently playing song areas left. * Some bugs with Git were fixed. |
| 08/03/2011 | Jake, Jim, Robbie, Yordan, Vitaliy | * Went over presentation * Tested what we were going to do for the demo, and then found it didn’t work so went to labs to finish it. * Next thing to do is polish the app so it’s good enough to give to testers |
| 11/03/2011 | Adam, Jake, Jim, Robbie, Yordan, Vitaliy | * Went over the new presentation. * Tested what we were going to do for the demo, practised syncing and streaming. The GUI is almost completely working. |
| 14/03/2011 | Adam, Jake, Jim, Robbie, Yordan | * Went over the updated presentation * Tested what we were going to do for the demo, practised syncing and streaming, again. |
| 23/03/2011 | Adam, Jim, Robbie, Vitaliy | * Discussed dates of when things need to be done by. * Decided that we should work as hard as possible next week to get the app to a high standard, near completion. * Agreed on set of features we must have implemented before second testing. * Adam will finish the settings dialog by Friday so we can do testing over the weekend. * Still need to work out how to statically link and distribute on Windows. |
| 03/05/2011 | Jim, Jake, Robbie, Adam, Vitaliy | * First meeting after Easter * Panic at how little time we have left! * Aim to start and finish report in three days * Report plan completed * Testing for demo begun * Weighting completed |
| 06/05/2011 | Jim, Jake, Robbie, Adam, Vitaliy, Yordan | * Second meeting after Easter * Check on any last minute changes * General report discussion |

## Risks We Took and Risk Assessment:

Even as we begun this project we knew there would be many obstacles along the road, any of which could cause serious delays to the project. The risks can be split broadly into two categories, risks we took ourselves, and risks to the success of the project from outside sources. During the course of development there were several points where we made a major decision that could have set us back a long way.

The first of these was the decision to use VLC as our media player. VLC, despite being very popular as an open source media player, is a hugely complex program, not to mention poorly documented. However, due to its popularity, it was our go-to program when looking for a media library to build into Streamberry. At the time we didn’t really think too much about this decision, we simply assumed because it was popular it would be fit for our purpose. But, as development on Streamberry got well under way we began to realise that the VLC library was perhaps not all it was made up to be. We detected a considerable number of memory leaks coming from libvlc, and we had issues with playing .wma files, they were subjected to constant skipping. By this point in the project we didn’t have the time to revert our decision and begin looking for another media library; however I do feel that with more careful prior research we might have been more aware of VLCs weaknesses, and could have chosen another library to work with instead.

The second major risk was changing our programming language from Java to C++. Although this change occurred early on the project we still had to scrap our work on threading and on networking, and fact that no one in our group had ever used C++ before was an obvious concern. However, the benefits outweighed the risks, Java was slow, it didn’t offer a good platform on which to build a light-weight and quick program. Although java’s libraries are vast, there was unusually limited support for media player frameworks. By this point we had decided to use VLC as our media player and its java api was very restricted compared to the flexibility offered by C++. In addition, we heard about Qt as a C++ library and how powerful it was. With cross-platform issues being solved by Qt and more control over memory and processes the decision to change to C++ was quite easy. We accounted for the fact that none of us had C++ experience by deciding we’d all have to learn it eventually and it couldn’t be that different from Java crossed with C. Still, perhaps we were overly optimistic as unfamiliarity with the language was certainly an issue towards the beginning of the project.

As it turns out, this risk did pay for itself in the end. Using C++ and Qt made the project considerably easier, so much so that I would say we would not have been able to build Streamberry in the time given. Qt’s library gave us easy access to databases, an excellent framework to code a GUI from, and a set of highly useful data structures, such as QSqlRecords, that were essential to the development of Streamberry.

The three other risky decisions we made during development were: Rewriting the whole GUI even though we had already spent over 30 hours designing and programming it, rewriting the whole of networking long after it was deemed ‘complete’ and programming the whole GUI by hand.

Programming the GUI by hand and rewriting it entirely certainly gave us the opportunity to create a unique and aesthetically pleasing GUI, however it came at the cost of time. These decisions were made early in the project, before we realised how much time GUI programming would actually take, and I think these two decisions are the ones responsible for setting us behind schedule. Programming such a nice GUI was a joy, and everyone in the group is certainly proud of how the program looks; however it was perhaps a luxury, and Streamberry might have benefitted from more functionality and polish as opposed to a striking GUI. That said, the user interface is a very important feature to our user base.

The original networking class needed to be rewritten as its initial incarnation was poorly pieced together and prone to instability. Rewriting the class allowed us to have a much more solid base on which to build the core functionality of Streamberry, although it too came at the cost of time. In this case, the trade off was well worth it, as it greatly eased the testing process, allowing us to eliminate any bugs quickly.

Other risks to the success of the project came from outside the project itself. Holidays could hamper our communication, other coursework from other modules could severely limit the hours we could put in at crucial times and group members might not all contribute evenly.

In order to mitigate these risks we took a few steps. Firstly, we held a pre-holiday meeting before each break began. This enable us to get a handle on what tasks had to be accomplished over the break, and who would do them. During the holidays themselves we held meetings by conference call, and spoke regularly on Skype as noted above.

Secondly, we were very aware of what coursework needed doing and when the deadlines were. Although we tried to minimise the impact of other coursework assignments on Streamberry development by putting high hours in whenever we were fortunate enough to have a break from more short-term coursework; this was not always successful. When Streamberry deadlines fell very close to other coursework deadlines, work time was nearly always restricted. In addition, Streamberry was often shoved into second place for the simple fact that the final deadline was many months away, and work could always be put in later. Unfortunately, these work clashes clearly caused serious delays in the development of Streamberry, and I would say that other coursework assignments preventing us from working as much as we wanted too were one of the major reasons that we didn’t manage to implement all the features we wanted.

## Conflict Management

During the project it was also important for us to have be aware that a conflict could occur, should two or more group members disagree very strongly. Luckily, it was never an issue as there were was no point in the project where anyone had really strong contrasting views; we worked well together as a team and discussed important decisions in detail, especially when the decision could be controversial. Everyone in the group had a voice and no one was afraid to voice their opinion; this helped us avoid any nasty conflicts.

## Quality Assurance

Throughout the development of Streamberry it was important to ensure that the code we were writing was up to a good standard. We regularly tested the program as we went, ensuring that it worked as we expected it to. When we had meetings we would discuss any problems we had and any areas we thought needed improving. Robert’s strong vision of how Streamberry should look and work gave us guidance and allowed us to finish Streamberry to the highest level we could.

To ensure our code was reasonably laid out and organised, and identifiers were correctly named we drew up a coding standards document, which can be found in the appendix. We did stick to the constraints of the document as much as we could, which I feel has helped our code maintain a uniform look and ensures it’s easy to read. As with any GUI programming the necessity of setting properties for all the various GUI elements adds a huge amount of lines to the code and can really affect readability. Wherever possible we have striven to keep property-setting code together and other ‘mass’ code such as setting up connections between objects at the start of the class to make the code easier to read.

# Appendix

## beaconsender.h

#ifndef BEACONSENDER\_H  
#define BEACONSENDER\_H  
#include "database.h"  
class BeaconSender : public QObject  
{  
  Q\_OBJECT  
public:  
  //Constructs a beacon sender object, takes a reference to the database  
  BeaconSender(Database &datab);  
public slots:  
  //Sends a beacon to all machines on the LAN indicating that this machine is going offline  
  void sendOfflineBeacon();  
private slots:  
  //Sends a beacon to all machines on the LAN, assumes normal beacon unless passed 'false'  
  void send(bool online = true);  
private:  
  Database& db;  
  QTimer \*timer;  
  QString myip;  
};  
#endif // BEACONSENDER\_H

## beaconsender.cpp

#include "beaconsender.h"  
#include "networking.h"  
#include "database.h"  
#include <QtNetwork>  
#include <QtCore>  
#include <QDebug>  
BeaconSender::BeaconSender(Database &datab): db(datab)  
{  
  networking n;  
  myip = n.getmyip();  
  timer = new QTimer(this);  
  //connect the timer signal to the send slot  
  connect(timer, SIGNAL(timeout()), this, SLOT(send()));  
  //tell the timer to fire every 5 seconds  
  timer->start(5000);  
}  
// Sends a streambeacon to every machine on the local network  
void BeaconSender::send(bool online)  
{  
  QUdpSocket \*udpsocket = new QUdpSocket();  
  //bind the UDP socket to the port used for beacons  
  udpsocket->bind(QHostAddress::Broadcast, 45454, QUdpSocket::ShareAddress);  
  networking n;  
  QString sendme = "";  
  // Beacon structure is "STREAMBEACON|<unique ID>|<database timestamp>|<ip address>" or "STREAMOFFLINE..." for offline beacons  
  if (online)  
      sendme.append("STREAMBEACON|");  
  else  
      sendme.append("STREAMOFFLINE|");  
  sendme.append(n.getuniqid());  
  sendme.append("|");  
  //get the timestamp from the database and add it to the beacon if not going offline  
  if(online) sendme.append(db.lastUpdate("Local"));  
  sendme.append("|");  
  sendme.append(myip);  
  qDebug() << "sending " + sendme;  
  QByteArray datagram = sendme.toUtf8();  
  //send the beacon  
  udpsocket->writeDatagram(datagram.data(), datagram.size(), QHostAddress::Broadcast, 45454);  
  delete udpsocket;  
}  
// Sends offline beacons to all machines on local network  
void BeaconSender::sendOfflineBeacon()  
{  
  timer->stop();  
  send(false);  
  //kill the thread  
  exit(0);  
}

## beaconreceiver.h

#ifndef BEACONRECEIVER\_H  
#define BEACONRECEIVER\_H  
#include <QHash>  
#include <QtNetwork>  
#include <QUdpSocket>  
#include "database.h"  
Q\_DECLARE\_METATYPE(QHostAddress)  
class BeaconReceiver : public QObject  
{  
  Q\_OBJECT  
public:  
  //Constructs a beacon receiver object, takes a reference to the database  
  BeaconReceiver(Database &datab);  
signals:  
  //Indicates that a library is out of sync and contains the information needed to update  
  void getLibrary(QHostAddress theirip, QString theirid, QString dblastupdate);  
private slots:  
  //Processes any incoming streambeacons  
  void processPendingDatagrams();  
  //Removes all machines that have timed out  
  void removeOfflineMachines();  
private:  
  //Creates a hash table that maps a string to an int  
  QHash<QString, int> onlinemachines;  
  Database &db;  
  QString myid;  
  //Checks that the BeaconReceiver knows the machine is online and that we have the most recent version of their library  
  void checkID(QString id, QString dbtimestamp, QHostAddress theirip);  
  QUdpSocket \*udpsocket;  
  QTimer \*timer;  
};  
#endif // BEACONRECEIVER\_H

## beaconreceiver.cpp

#include "beaconreceiver.h"  
#include "networking.h"  
#include "sbexception.h"  
#include "utilities.h"  
#include <QtNetwork>  
#include <QUdpSocket>  
#include <QHashIterator>  
#include "libraryrequester.h"  
// Creates a beaconReceiver that will process all streambeacons received over the LAN  
BeaconReceiver::BeaconReceiver(Database &datab) : db(datab)  
{  
  qRegisterMetaType<QHostAddress>();  
  networking n;  
  myid = n.getuniqid();  
  LibraryRequester \*lr = new LibraryRequester(db);  
  //QThread \*lrthread = new QThread(this);  
  //lr->moveToThread(lrthread);  
  //lrthread->start();  
  connect(this, SIGNAL(getLibrary(QHostAddress,QString,QString)), lr, SLOT(getLibrary(QHostAddress,QString,QString)));  
  //Creates a timer that tells the object when to check for timeouts  
  timer = new QTimer(this);  
  connect(timer, SIGNAL(timeout()), this, SLOT(removeOfflineMachines()));  
  timer->start(10000);  
  //Creates a UDP socket and sets it to listen for streambeacons  
  udpsocket = new QUdpSocket(this);  
  udpsocket->bind(45454, QUdpSocket::ShareAddress);  
  connect(udpsocket, SIGNAL(readyRead()), this, SLOT(processPendingDatagrams()));  
}  
//Gets a datagram from the UDPsocket, decodes it and checks that it belongs to streamberry, takes appropriate action if so  
void BeaconReceiver::processPendingDatagrams()  
{  
  while (udpsocket->hasPendingDatagrams())  
  {  
      try  
      {  
          QByteArray datagram;  
          datagram.resize(udpsocket->pendingDatagramSize());  
          udpsocket->readDatagram(datagram.data(), datagram.size());  
          QString datastring = (QString) datagram.data();  
          qDebug() << "received " + datastring;  
          networking n;  
          QString id = n.parsebeacon(datastring, networking::uid);  
          //If normal beacon then check sender is in the hashtable and check their library  
          if (n.parsebeacon(datastring, networking::beaconHeader) == "STREAMBEACON")  
          {  
              //Checks that beacon is not our own  
              if (id != myid)  
              {  
                  QString dbtimestamp = n.parsebeacon(datastring, networking::timestamp);  
                  checkID(id, dbtimestamp, QHostAddress(n.parsebeacon(datastring, networking::ip)));  
              }  
          }  
          //If offline beacon  
          else if (n.parsebeacon(datastring, networking::beaconHeader) == "STREAMOFFLINE")  
          {  
              if (id != myid)  
              {  
                  //Sets the machine offline in the database and removes it from the hashtable  
                  db.setOnline(id, "0");  
                  onlinemachines.remove(id);  
                  //qDebug() << id + " is offline";  
              }  
          }  
      }  
      catch (SBException e)  
      {  
          qDebug() << e.getException();  
      }  
  }  
}  
//Checks that the BeaconReceiver knows the machine is online and that we have the most recent version of their library  
void BeaconReceiver::checkID(QString id, QString dbtimestamp, QHostAddress theirip)  
{  
  try  
  {  
      QString lasttimestamp = db.lastUpdate(id);  
      //if never seen before, add them to the database  
      if(lasttimestamp=="")  
      {  
          QString username = "Gary Oak";  
          db.makeUser("0", QString::number(Utilities::getCurrentTimestamp()), id, username);  
          getLibrary(theirip, id, lasttimestamp);  
      }  
      //if their timestamp is newer, get new one  
      else if (lasttimestamp.toInt() < dbtimestamp.toInt())  
      {  
          getLibrary(theirip, id, lasttimestamp);  
      }  
      //If the machine has been seen before, but just come online then tell the database that it is online  
      else if(onlinemachines.value(id) == 0)  
      {  
          qDebug() << "setting machine online, lasttimestamp =" << lasttimestamp;  
          db.setOnline(id, "1");  
      }  
      db.setIPaddress(id, theirip.toString());  
  }  
  catch (SBException e)  
  {  
      qDebug() << e.getException();  
  }  
  //Add/update the online machines table  
  onlinemachines.insert(id, Utilities::getCurrentTimestamp());  
}  
//Iterates over the hashtable of online machines and checks for timeouts  
void BeaconReceiver::removeOfflineMachines()  
{  
  QHashIterator<QString, int> i(onlinemachines);  
  while (i.hasNext())  
  {  
      i.next();  
      //If we haven't received a beacon from a machine in over 12 seconds, assume timeout  
      if (Utilities::getCurrentTimestamp() - i.value() > 12)  
      {  
          //Set machine offline in the database  
          db.setOnline(i.key(), "0");  
          //qDebug() << i.key() + " has timed out";  
          //Remove the machine's ID from the hashtable of online machines  
          onlinemachines.remove(i.key());  
      }  
  }  
}

## Sections from database.cpp

QList<QSqlRecord>\* Database::searchDb(int type, QString playlist, QString searchtxt, QList<QString>& sortcols, QList<QString> order, int musicorvideo)  
{  
QString condition(" WHERE 1=1");  
QString ordering;  
QString sql;  
QSqlQuery result;  
QList<QSqlRecord> users;  
QList<QSqlRecord> \*files;  
searchtxt = searchtxt.trimmed();  
QStringList searches = searchtxt.split(" ", QString::SkipEmptyParts);  
switch(type)  
{  
case 1:  
  foreach(QString str, searches)  
  {  
    condition += " AND (Artist LIKE \"%";  
    condition += str;  
    condition += "%\")";  
  }  
  break;  
  case 2:  
  foreach(QString str, searches)  
  {  
    condition += " AND (Title LIKE \"%";  
    condition += str;  
    condition += "%\")";  
  }  
  break;  
  case 3:  
  foreach(QString str, searches)  
  {  
    condition += " AND (Genre LIKE \"%";  
    condition += str;  
    condition += "%\")";  
  }  
  case 4:  
  foreach(QString str, searches)  
  {  
    condition += " AND (Album LIKE \"%";  
    condition += str;  
    condition += "%\")";  
  }  
  case 5:  
  foreach(QString str, searches)  
  {  
    condition += " AND (Length LIKE \"%";  
    condition += str;  
    condition += "%\")";  
  }  
  default:  
  foreach(QString str, searches)  
  {  
    condition += " AND ((Artist LIKE \"%";  
    condition += str;  
    condition += "%\") OR (Title LIKE \"%";  
    condition += str;  
    condition += "%\") OR (Album LIKE \"%";  
    condition += str;  
    condition += "%\") OR (Genre LIKE \"%";  
    condition += str;  
    condition += "%\"))";  
  }  
  break;  
}  
condition += " AND (Deleted='0')";  
switch(musicorvideo)  
{  
case 0:  
case 1:  
  condition += " AND MusicOrVideo='";  
  condition += QString::number(musicorvideo);  
  condition += "'";  
default:  
  break;  
}  
int sortcount = sortcols.length();  
if(sortcount>0)  
{  
  ordering += " ORDER BY ";  
  ordering += sortcols.at(0);  
  ordering += " ";  
  ordering += order.at(0);  
  for(int i = 1; i < sortcount; i++)  
  {  
    ordering += ", ";  
    ordering += sortcols.at(i);  
    ordering += " ";  
    ordering += order.at(i);  
  }  
  ordering += ", Track ASC";  
}  
try  
{  
  sql = "SELECT UniqueID FROM LibIndex WHERE (Online=1)";  
  result = query(sql);  
  result.first();  
  while(result.isValid())  
  {  
    users.append(result.record());  
    result.next();  
  }  
  files = new QList<QSqlRecord>();  
  while(!users.isEmpty())  
  {  
    sql = "SELECT \* FROM Lib";  
    QString id = users.takeFirst().value(0).toString();  
    sql += id;  
    if(playlist!="")  
      sql += ", PlaylistTracks";  
    sql += condition;  
    if(playlist!="")  
    {  
      sql += " AND Playlist='";  
      sql += playlist;  
      sql += "' AND PlaylistTracks.ID=Filepath AND PlaylistTracks.UniqueID='";  
      sql += id;  
      sql += "'";  
    }  
    sql += ordering;  
    result = query(sql);  
    result.first();  
    while(result.isValid())  
    {  
      QSqlRecord copy(result.record());  
      QSqlField temp("Length", QVariant::String);  
      QString intval = result.record().field("Length").value().toString();  
      QString timeval = Utilities::intToTime(intval.toInt());  
      temp.setValue( timeval );  
      int pos = copy.indexOf("Length");  
      copy.remove(pos);  
      copy.insert(pos, temp);  
      if(result.record().field("Filepath").value().toString().indexOf("\\;") != -1)  
      {  
        const QString fieldname = "Filepath";  
        QSqlField temp(fieldname, QVariant::String);  
        temp.setValue(result.record().field(fieldname).value().toString().replace("\\;",";"));  
        int pos = copy.indexOf(fieldname);  
        copy.remove(pos);  
        copy.insert(pos, temp);  
      }  
      if(result.record().field("Artist").value().toString().indexOf("\\;") != -1)  
      {  
        const QString fieldname = "Artist";  
        QSqlField temp(fieldname, QVariant::String);  
        temp.setValue(result.record().field(fieldname).value().toString().replace("\\;",";"));  
        int pos = copy.indexOf(fieldname);  
        copy.remove(pos);  
        copy.insert(pos, temp);  
      }  
      if(result.record().field("Album").value().toString().indexOf("\\;") != -1)  
      {  
        const QString fieldname = "Album";  
        QSqlField temp(fieldname, QVariant::String);  
        temp.setValue(result.record().field(fieldname).value().toString().replace("\\;",";"));  
        int pos = copy.indexOf(fieldname);  
        copy.remove(pos);  
        copy.insert(pos, temp);  
      }  
      files->append(copy);  
      result.next();  
    }  
  }  
}  
catch(SBException e)  
{  
  throw e;  
}  
return files;  
}  
QString Database::getUniqueID()  
{  
QString sql = "SELECT UniqueID FROM LibIndex WHERE Local = 1";  
try  
{  
  QSqlQuery result1 = query(sql);  
  result1.first();  
  QString result = result1.value(0).toString();  
  return result;  
}  
catch(SBException e)  
{  
  throw e;  
}  
}  
void Database::setIPaddress(QString uniqueID, QString ipaddress)  
{  
QString sql;  
sql = "UPDATE LibIndex SET IPAddress='";  
sql += ipaddress;  
sql += "' WHERE UniqueID='";  
sql += uniqueID;  
sql += "';";  
try  
{  
  query(sql);  
}  
catch(SBException e)  
{  
  qDebug() << e.getException();  
  throw e;  
}  
}

## The Streamberry Coding Standards Document

### Brackets, Spacing & Indentation

Bracing and indentation should be done in a style similar to the Allman style. The measurement of indentation is four spaces. Indentation should increase on the line after an open curly bracket (if curly brackets are omitted in a statement, indentation should act as if the brackets were still present) and should decrease on a line including a right curly bracket.

Curly brackets should have their own line. There should be an empty line two lines above a left curly bracket, unless this is also a curly bracket. There should be two empty lines between function declarations. Empty lines can also be used to break up code into logical sections.

There should be a space to the left of left round brackets, with the exception of when they are used in function calls and declarations, or the symbol to the left is another left round bracket. Code surrounded by round, angular and square brackets should not be padded with white space.

Assignment and bitwise operators should have a space on both sides, with the exception of increment and decrement. Comparison operators should not have spaces either side. Commas should have no space to the left, and one space to the right.

### For example:

int main (int argc, char[] \*argv)  
{  
  for (int i = 0; i<2; i++)  
  {  
      std::cout << i << “\n”;  
  }  
  
  std::cout << “Program starting.\n”;  
  
  if (argc==1)  
  {  
      std::cout << “Hello World.\n”;  
  }  
  else  
  {  
      std::cout << helloWithArguments(argv);  
  }  
  
  std::cout << “Program finished\n”;  
}

### Naming

Class names should have a capital letter at the start of every work that makes up the class name.

Method names should use camel case, with a lowercase letter for the first word only.

Variable names should be entirely lowercase. If the variable name is a combination of two or less words, an underscore is not usually necessary, however if it becomes difficult to read, underscores may be used to split up words in variable names.

Global constants (i.e. hash-defines) should be all in upper case, and if necessary underscores may be used.

### For example:

#define WINDOW\_HEIGHT 500  
  
class ExampleClass  
{  
public:  
  void exampleMethod();  
  void anotherExampleMethod(int variablename);  
private:  
  int examplevariable;  
  SecondClass \*second\_example\_variable;  
};