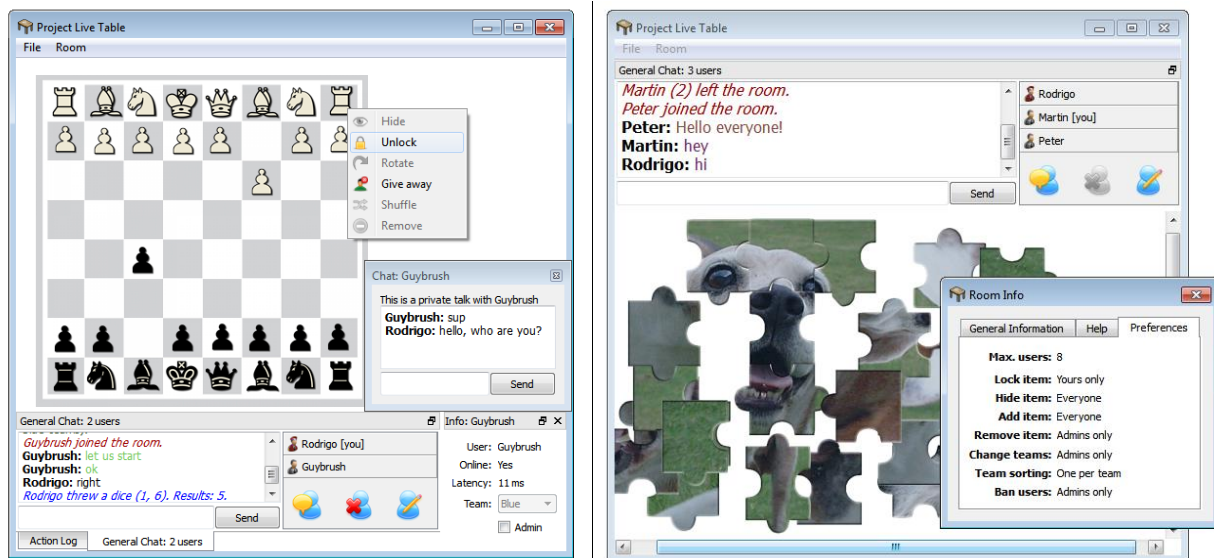


# Project Live Table

Project Live Table is an open-source application where users can interact in real time through a virtual table. You can use this table to share images, plan team strategies, play card or board games, etc. Although this is not a game, one of its main focuses is to allow any kind of board or card game to be easily playable through this table. You can create and host games inside the application and play with others in the web or in a local LAN network.



The application was developed using c++ and the Qt libraries. The web servers use PHP and MySQL for the database. Sound was edited with Sony Vegas and the images were manipulated in Photoshop. The purpose of this document is to help you understand the architecture behind the application.

## User-contribution:

One of the great features of the application is the ability to contribute. Users can create new tables and share these tables with others by hosting them. When a user hosts a table, a new folder with the type of table is created in the application folder. This table will save all the information about the table being hosted, such as images downloaded and uploaded. As an advantage, the table can be saved and reused in the future. Even if the user does not save the table, the application still saves it periodically into an auto-save file. Once the room is hosted, the application contacts the web server. The server will add the user IP and port into the database. When the room is closed, the IP is removed. Sometimes the room can be closed without sending a final message to the web server. In this case, the IP continues in the list for a fixed number of days (determined dynamically by administrators in the website).

So, how is this IP used? When users want to join a room, they will receive a list with the most recently added IPs. A specific number of threads will contact each room individually to fetch information, such as latency, location, and the number of users inside. All the IPs that responded will be added to the list of opened rooms. The search keeps fetching more and more IPs from the web server until the list is over or the number of IPs rejected in a row is too high. All of this is done by the Engine dynamic library. This library deals with all the communication and house-keeping. It hides all the complexity of the application without knowing what the application is all about. So, the same library can be used by any multiplayer application. I will talk about the Engine library later on. Let's get back on track.

In the loading screen, the user receives all the detailed information about the other users in the room and the table. But, it does not receive the heavy images. The room can hold thousands of items with images on it. The sharing of images is key to the user contribution. When the table tries to load the items, each item will search locally in the table folder for the image file based on the image size and an image hash received in the loading screen. If the file was not found, a request is sent to all others in the room. Then, the user downloads from the user who replied first (it assumes is the fastest internet connection). The file transference is crypted and happens in the Engine library. The priority of packages from this transference is quite low because there are higher priorities in the room, such as the chat and the moves. There are many types of exceptions handled by the Engine, such as if the user deletes the file being transferred or if the user sending the file leaves the room. In that case, the user will just download the file from a different user. If nobody has the image, an error image will be shown that can be refreshed at anytime if one user eventually decides to restore the image deleted. There are more exceptions that are not necessary to talk about. Let's trust it works and keep this simple.

Once users can share files with each other automatically, we open doors for many possible uses of the table. Users can pre-load in a table file a collection of images and then just access this collection inside somebody's room. All the images from this collection will be added to someone's table. This can have many uses, such as bringing to the table a whole deck of cards with permissions already prepared for a quick card game. And once the table is loaded in somebody's computer, all files can be accessed, copied or deleted from the application folder. The webserver does not keep track of any information. The communication is all crypted and confidential. If the room is password protected by the user, this password will be part of the secret key used in the cryptography of the chat messages and transference of files. The only information stored in the web servers are the host IPs, but they are stored only for a very short period of time. This confidentiality gives the users some privacy.

With the ability to contribute and open rooms to the public, strangers can play games together. They can not only play, but create their own games, host a room with a brand new idea and others can instantly join. I am myself unaware of all the possibilities that can come to light through this application. It is part of the application's main goal to allow easy and intuitive contribution and manipulation of content.

## Web services:



The application website was developed in PHP. I'm not really proud of the organization of the PHP files (except the fact that the whole website was developed in 2 days using only the windows notepad). The Facebook integration in the main page allows the user to comment and share the website with others. Comments on the website can be posted in the users Facebook profile and will warn them if someone replies. But, nobody cares about Facebook integration.



What is crucial about the web server is the support it gives to the application. When the user opens the application, the web server sends to the user the current version of the application, the user Geolocation, and the latest news. The version of the application is manipulated through an admin panel in the website. Admins can update the version, news, and version log. They can also upload and delete files. The server reads all files uploaded and prepares them in 2 ways. In the first way, the web server puts all files together in a zipped folder and the user can download this zipped folder through the download link in the website. In the second way, the web server prepares an index file with filenames and sizes for the automatic update. When the application receives the version, if this version does not match the current version, then the user receives a dialog alerting about a new version. Once accepted, the application will update itself replacing files that are not the same (while keeping the ones that are the same).

The web server Geolocation is based on a Geolocation library. The location is calculated based on the user IP. The server does not keep track of locations. It only checks the user IP, grabs the appropriate location and send to the application as a string. It is the user application that will share its location with others if the user allows that to happen.

Finally, there is the MySQL server that stores the tables. As mentioned above the tables are very simple and stores only what is essential. Take a look yourself:

hosts	updates
 ip: varchar(18)	 version: varchar
port: smallint(5)	description: text
time: timestamp	versionlog: text
	time: timestamp

That's it! The updates table is only edited when an admin sends information through the admin panel. "Description" is actually the news that can be edited at anytime. News and updates are in the same table because of the likelihood of releasing news together with some changes. Also, these tables can be created automatically through the admin config panel (double-password protected). Deletion also happens automatically. When a user hosts a room, the IP is added to the hosts table and IPs that are older than a certain date are deleted. Differently, the updates table has a fixed size. It deletes the last entry when a new version is released.

### Separation of concerns:

Every crucial and decisive computation is done on the server or host side. In the web server for example, the contact with the database and all the processing happens server side in PHP. The user will only receive the results. Now, it is good to remember that when users host a room, they are also creating their own server. So, they get responsible for the majority of decisions. Every random number in the application is requested to the host user that generates them and sends to others. This is done mostly to prevent cheating and improve synchronization. Key decisions such as to delete, lock, hide, add, delete, rotate, or possess an item are decided both on client and server side. The client checks for permission before sending the request to the host. The host checks if the client has permission and then, if the request was accepted, a message is sent to everyone in the room asking to carry on with the request. So, even if the users find a way to modify the application to ignore permissions, the host would still deny their request.

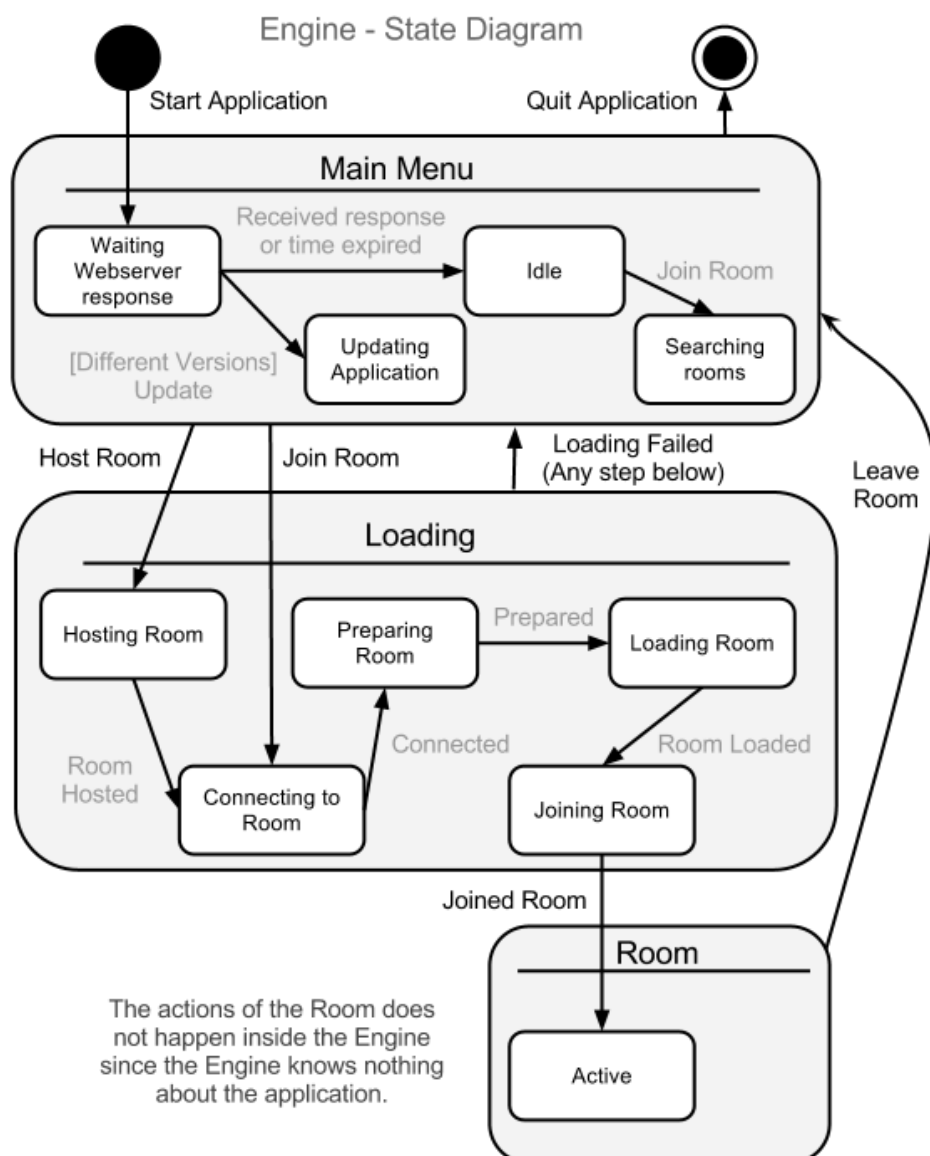
There are other separations of logic, such as controllers and views. But, it is the Engine library that does the biggest separation of concerns. I'll talk about it in the following topic.

### The Engine Library:

In the main function of the application, the Engine is initialized. The main purpose of the Engine is to decrease all the complexity of the application by hiding how messages are traded and views are switched. When the Engine starts, it might update the application (if the app received an argument to do so) or load the main application view. From this view, the user can find rooms and create new rooms. The only thing the application has to do in this case is to tell the Engine where each information will be displayed in the view (through the abstract factory). For example, the Engine asks the application where it will display the user nickname, and the only thing the application has to do is to point to the label where it will be displayed. Of course, the application can still extend some methods of the Engine to

implement more specific functionalities. But, with the Engine, to make a multiuser application becomes quite easy. It does all the communication with the web server, all the switching of views and the pre-processing of the loading screen. In the loading screen for example, the application only have to specify how to handle errors or what about the application that is going to be processed.

Furthermore, the Engine is responsible for dealing with users. A user object can only be instantiated from inside the Engine. When a user joins a room, a pointer to this user object is sent to the application. All the basic information about the user is stored there, such as a unique ID, nickname, IP address, etc. The application extends the User class to add more application-specific variables, such as the user team. In the abstract factory, the application can decide what Engine classes are going to be extended. Not all classes can be extended, only those superficial ones, such as the user class or the room. The low level of the Engine that deals with the network and cryptography cannot even be accessed. Here's a state diagram that shows exactly the states of the Engine:



The chat room is all done inside the Engine. It assumes that every application would have the same chat. But, the “user” button in the chat opens a new dialog with user information, and this dialog can be extended outside the Engine to implement what is going to be displayed. In this case, the application is extending it to display the user team.

While the Engine has its own protocols and does all the core network computations in the background, the application does what I call “the Actions.” The Actions have a list of all users in the room and can send a message to one. This message is sent to the Engine that will handle everything. Once you received a message, the Engine alerts the Actions. It is the Actions that will interpret the messages and turn them into... actions!

### **Security:**

We all know that there is no such thing as a perfectly secure application. I do not want to be responsible if someone hacks the website and releases a version update that is actually a different application. Although I still don’t understand why anyone would do that, there were precautions made. First, the admin folder in the website is password protected. There’s also a second password for the admin config panel that handles the database. Every input method in the website was sanitized.

I believe that the form I sanitized the input will protect against SQL Injection and Cross site scripting. I tried to attack myself and could not do it. Also, user input that goes to the database is all done through POST methods (never GET methods). In addition, the website has a second URL where it can be accessed as https (using a SSL certificate).

Buffer Overflow is well prevented in the application, especially in the Engine. I’m using Qt classes to control all the arrays, and Qt provides some protection against overflow. Also, every network package handled in the Engine has a fixed size (TCP packages are bigger than UDP ones). The application won’t check after that size.

To prevent errors in numeric representations, the messages try to keep the same format and primitive data types are 100% avoided in the trading of network messages. Instead of using “int” for example, “qint32” is used. This second one is defined by Qt and deals with cases like integers having different sizes in different operational systems. Numeric representations are also sent through a QDataStream that does the Endianness conversion, if necessary.

Since authorizations are all processed on server side, it is hard to take advantage of racing conditions (forcing a redirection between the time that the authorization was checked and when the access occurs). When a user wants to join the room, the request is sent to the host of the room. If the host denies this request, every subsequent message from the user will just be ignored. Even if the users hack the application to find a way to join the room, they won’t be able to send or receive messages because they will all be ignored in the server side. When a message is received, the host checks if the socket descriptor is in the “allowed” list. To be on that list, the user has to go through all the authorization steps in the loading screen. Otherwise, the message will be ignored. Moreover, when a room is created, a secret key is randomly generated in the low level of the Engine. Every user that

receives authorization to join the room will receive a unique md5 hash that combines the application secret, the room password, and this key generated when the room was opened. This hash is the key to be able to read messages sent from the server. If the user was not authorized to join a room, he/she won't receive this key.

The contact with the database also requires a hash key. It is a different key than the one used for the contact between users, but it also makes use of the application secret key. The web server also has this application secret (that can be set in the admin config panel). So, both the application and the PHP file have a copy of the same algorithm that produces the hash key. When the user sends a request to the web server, it also sends the hash key. The web server will only proceed with the request if the key matched its internal key. So, even if a user downloads the source code of the application and modifies a few details, it will not be able to join a room or see the list of rooms because the application secret won't match. In case somebody uses reverse engineering to steal the application secret from the executable file, administrators can change it in a matter of seconds by releasing a quick update through the website.

Despite all keys, users can still create their own. Every room can be password protected and this password becomes part of the key. Users that try to access this room will never receive this password. They only submit the password to the host user and receive a reply saying whether or not the password is valid. If invalid, they might have to wait for a while before trying again. If valid, they will receive the new key that combines this password. Also, the host user can ban users from the room. When a user is banned, the user IP is blocked. Any message coming from the user IP will be ignored.

Directory traversal could also be a problem because the version number is part of the filename to download the application zip file. To prevent it, a JavaScript limits the characters allowed in the version number to avoid versions like "../" Also, other characters are added to the filename before the version (assuming the user somehow managed to get into the admin panel and ignored the JavaScript function).

Now, suppose that there was a security flaw in the Engine library that was patched in a new version. The user could still use the old Engine library instead. To prevent that, the Engine has also an internal version number, and this version must match with the Engine version number in the application. Otherwise, the application won't load. Versions between users must also match. If a user using a different version tries to join a room, this user will receive an error message saying that the version is different.

Finally, the web server tries to prevent Denial-of-Service attacks by blocking the user's IP for hours if the user is sending too many requests in a short period of time.

## Design:

The Engine and the application have different design focuses. The main focus of the Engine is to be easy to use, reliable and fast. It chooses performance over adaptability, and it was planned to not be so constantly updated. Differently, the application focuses on being easy to change. It is planned to be constantly evolving and receiving updates. For items in the table for example, even though the application was released with one type of item, it has a whole architecture designed to easily add more. The item decorator wraps up any type of item. The table has an item factory that can easily add new types, and the “TableItem” class can be easily extended. There’s also a folder for table items (such a sad folder with just one class).

The design of the GUI focuses on simplicity. This application has enough potential to be feature creep, which should be avoided. Instead of having 2 rotate buttons in the context menu for example, there is only one. So, to rotate to the other side you have to press it 3 times (or just double click the image once).

Simplicity does not mean lack of options. The application does its best to keep it easy to use, but impossible to master. There was an attempt to add shortcuts to most of the options. More shortcuts might even be added in the future. One great concern about this application was the wrong aesthetics. Each sound effect of the application was listen dozens of times to make sure it does not get annoying after a long exposure. A feature that is constantly used and does not flow or is not intuitive enough can make the user leave and never come back. To make the application more intuitive, I let others use the application without saying anything about it just to see where they were having most trouble with. So far, it sounds sufficiently dumb-proof, but more design changes are definitely expected in the future.

When designing the application, one of the focuses was in its target audience. It is an application that might be mostly used for young adults playing games. The initial target audience are those creative people who want to spend some time discovering ways to use the table and socializing with others; people who were willing to download an unknown application in a new website. Based on stereotypes, the design decisions focused more on the “classic” of gaming. That includes, of course, LAN parties. For that reason, the application can still work fine without internet connection. Logins sounded like it would be too demotivational for those who just want to have a sneak peek of what the app is all about. Facebook integration would be hell, especially when one of the main concerns of the application is privacy.

A virtual table can’t be as a real table, can it? Definitely not, but if the essence is captured, perhaps it can be even more fun. The ways to interact with the table items try to capture the essence of interactions. To allow interactions to be ruled, the table limits user freedom based on 3 main dimensions: LOCK, HIDE, and POSSESS. When you lock, you prevent others from interacting. When you hide, you prevent others from seeing. When you possess, you are the owner and have the privileges over the item. Of course the owner of the table might also have its privileges (the host user). Based on those three dimensions, it is expected that, with some creativity, most interactions should work. For example, you could hide a card to simulate a card in your hand or lock your chess pieces to prevent the



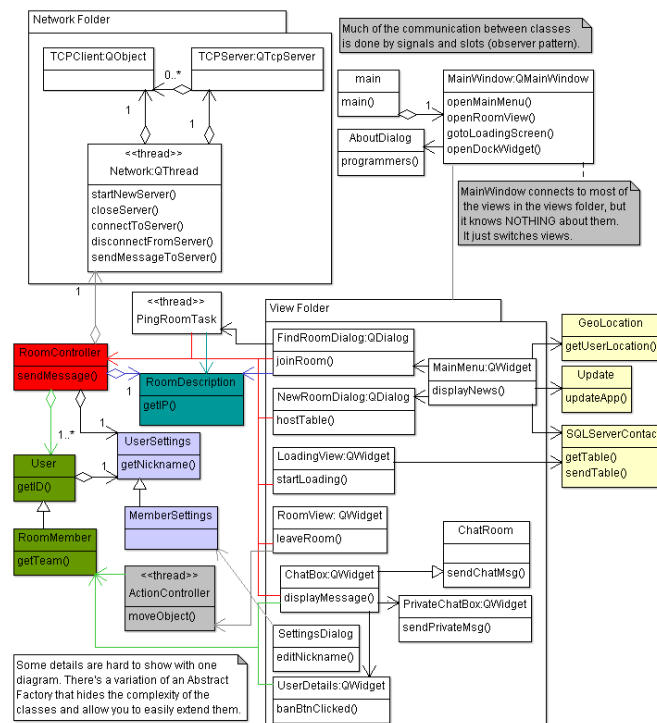
other player from moving them. As in most games, you can still cheat. But, this freedom to cheat might be one key component in the fun factor of the application. In the end, this is just an experiment. It is hard to be sure about how much enjoyable this application can be because to develop it means to look to the application with technical and critical eyes. Those eyes are good to analyse the code, but partially oblivious to the enjoyment. Only future feedback can determine whether or not this virtual table is capturing the essence of a real table interaction.

Also, features that are least used are the hardest to be found. By following the principle that the bigger the button, the most people will click on it, buttons that are expected to be clicked most of the time are actually bigger!

Find Table  
New Table  
Exit

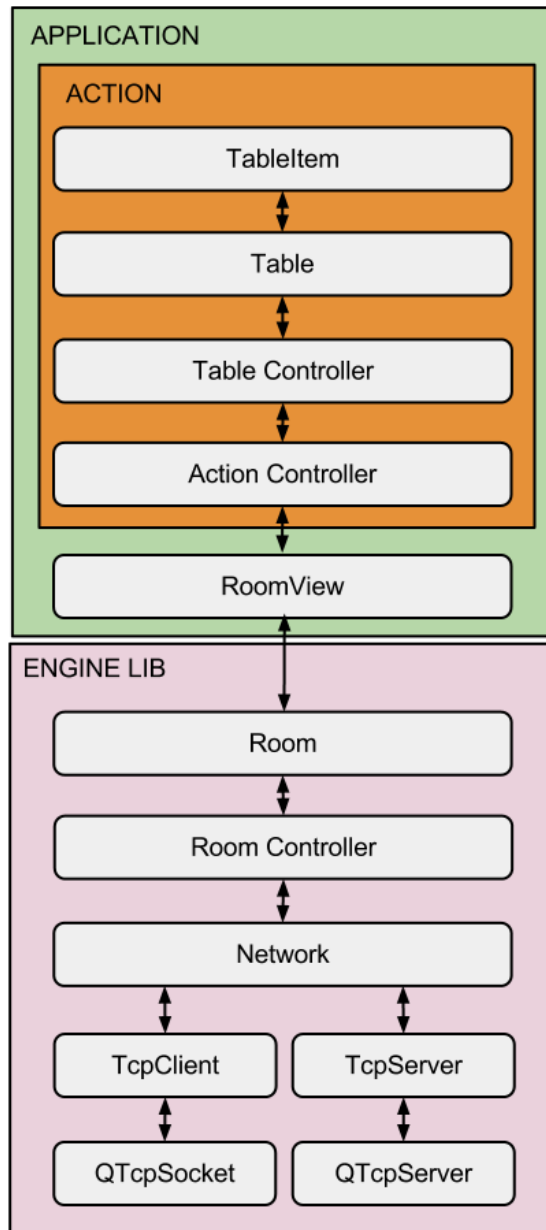
## Evolution:

The application has evolved since it started. The biggest evolution of the application is in the source-code. The architecture changed from a mess of classes to a more layered style. The Engine was before integrated with the application. Now, it is a dynamic library that is so oblivious of the purpose of the application that it can easily be used in any multiuser app. Here is the first planned architecture:



The architecture changed to this:

Layered Class Diagram  
(How the table items interacts with the network).

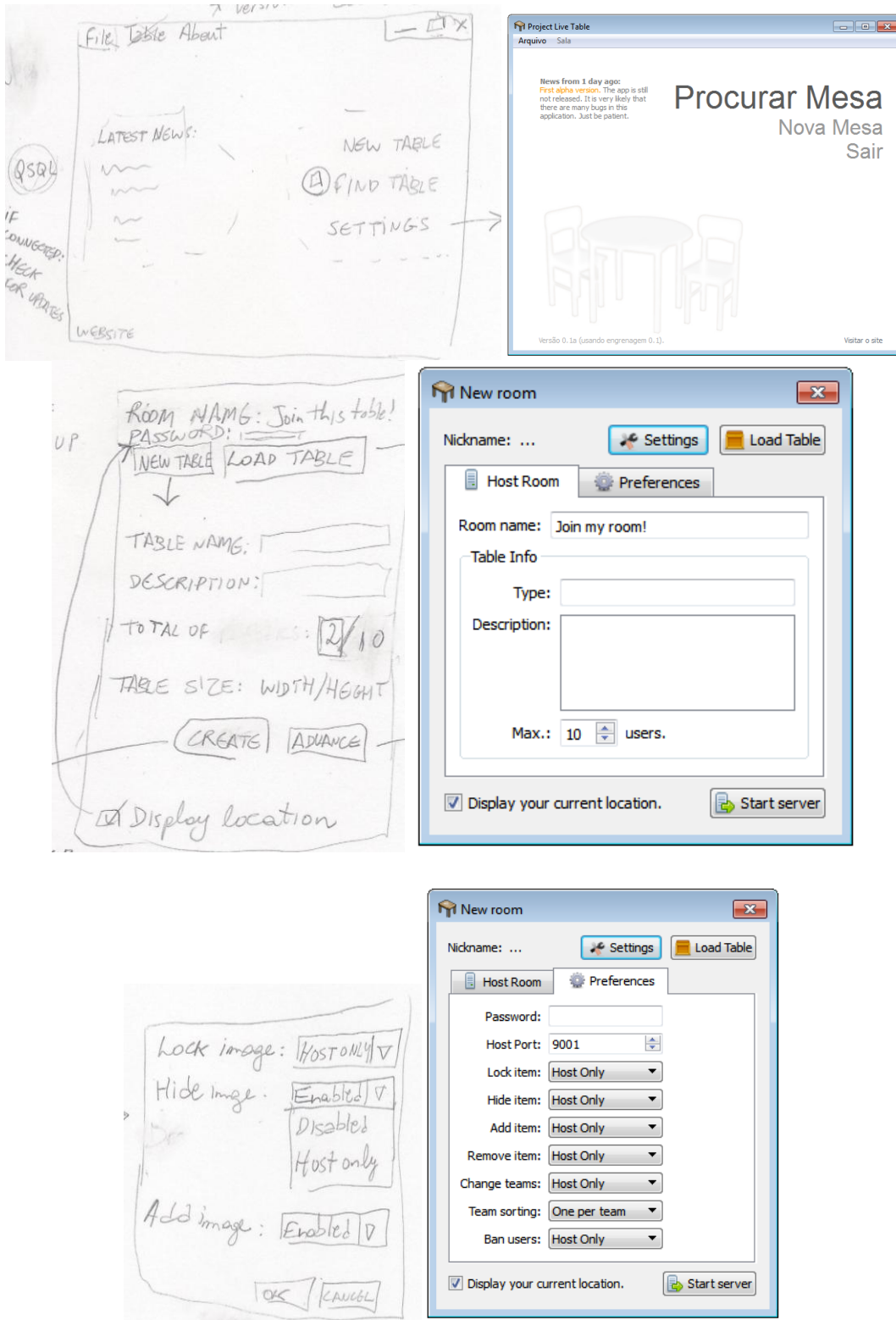


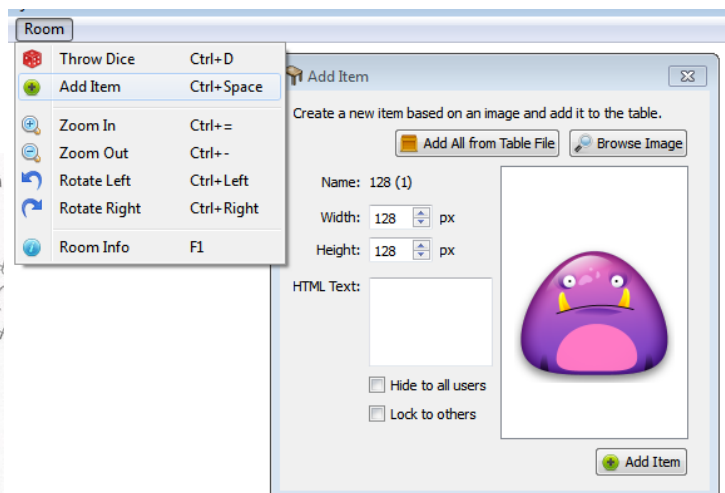
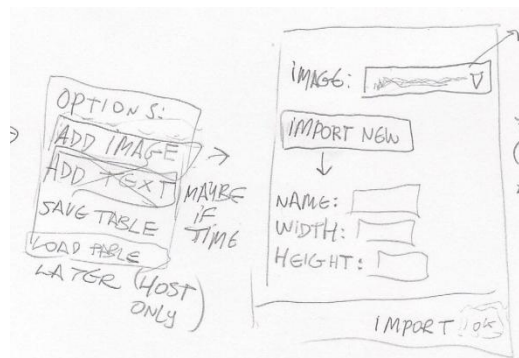
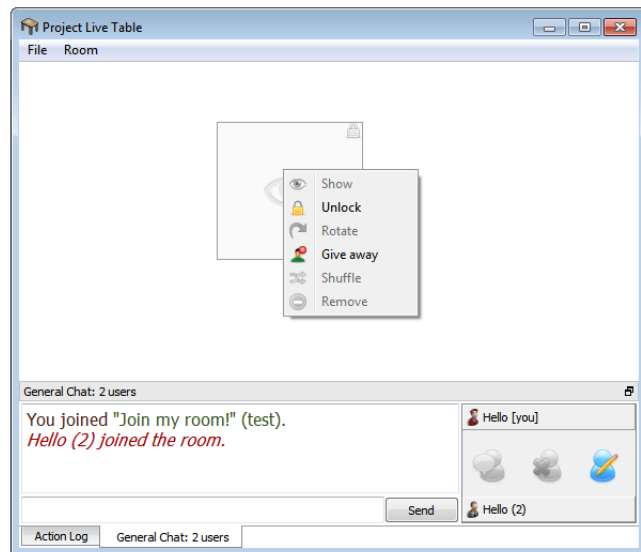
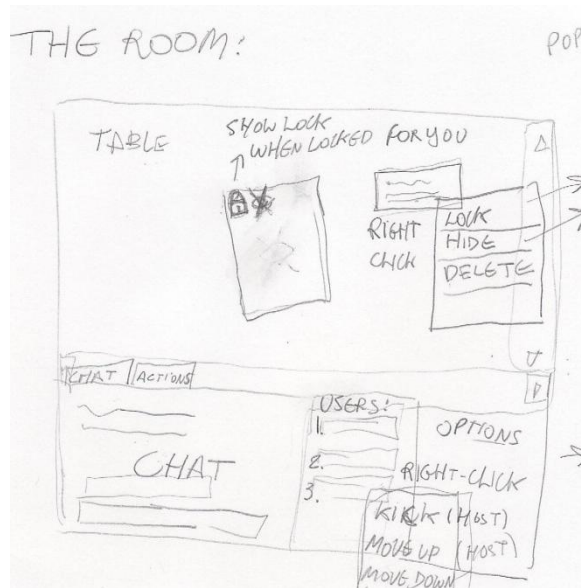
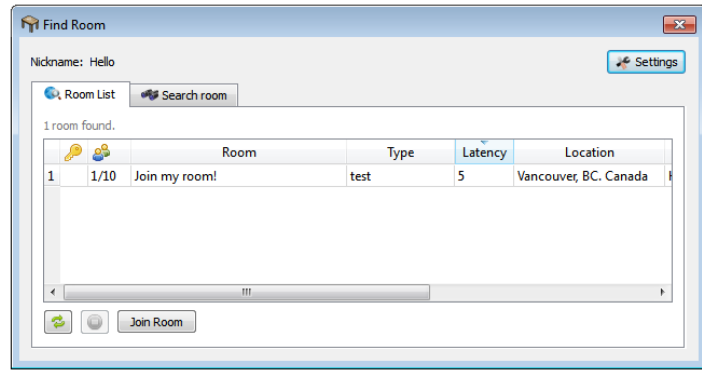
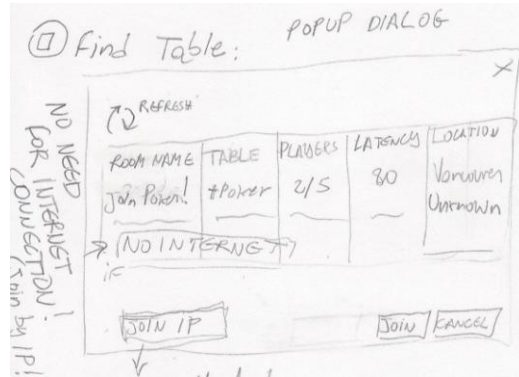
Despite the architectural changes, the purpose of the application is still the same. All the initial user stories were implemented as planned. However, the first 2 features of an item were “lock” and “hide” only. The “possess” feature was added later to address different problems of permission, such as who could lock and hide an item.

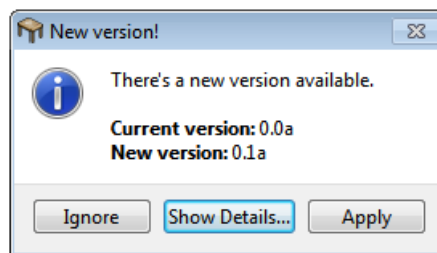
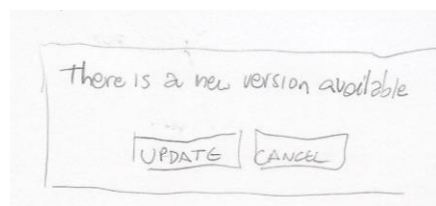
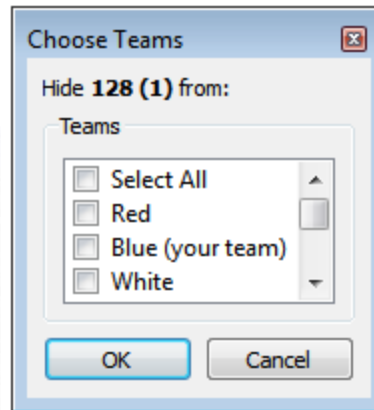
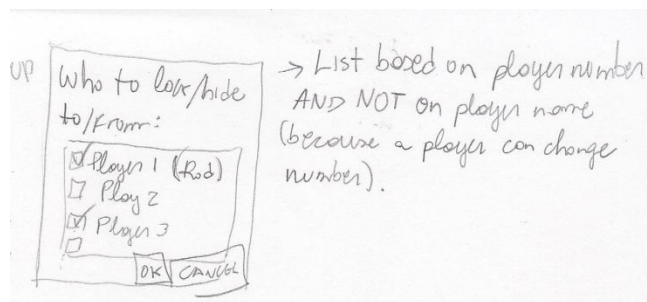
The project might continue to evolve along time. The Engine can continue to be refactored to become a library easy to use and fast enough for any time of application.

## Paper prototype:

I am now comparing the first ever paper prototype with the first working version:







## Testing:

Test cases were implemented using the QTestLib library. This is a library from Qt that provides special methods for testing. Meta classes, signal spies, and some injections were used in order to properly test the application. They were in a subproject that is not included in the source code.

## Verification:

For static analysis, a tool called Cppcheck was used. I'm not really sure about the quality of such tool, but it was the best I could find for free.

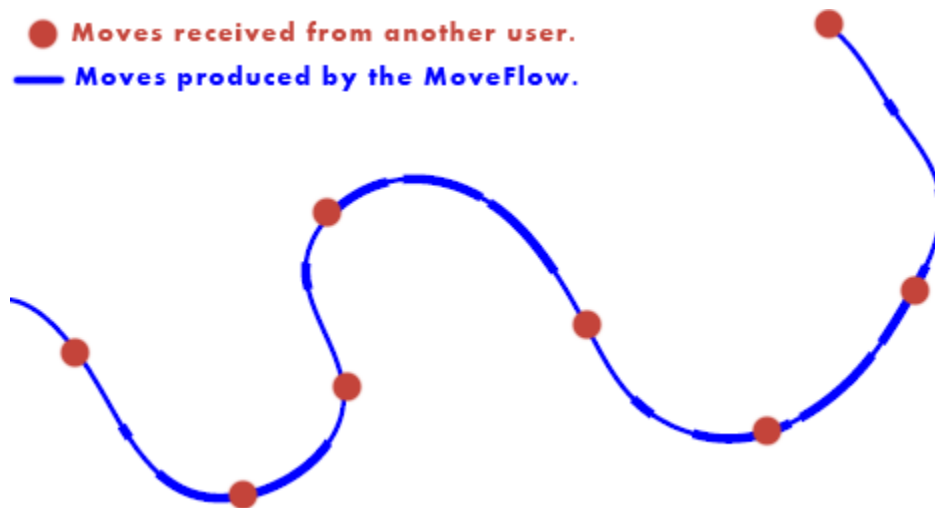
## Cool features:

There are many features in this project that deserves some attention. I will just talk about a few of them.

1. The most complex feature of all is the network and how it is integrated with the Engine library. The network is a thread that sends both TCP and UDP messages, and uses its own protocols. There is a complex procedure of messages being passed from class to class until it reaches the final destination (more about the network is explained in the graphs below). I have also implemented a file-transference system that allows you to send a signal to the File Transfer thread with the filename. The thread will take care of everything. It reads the file, cryptographs it based on randomly generated keys shared only with users in the room, breaks it into pieces, and sends to another user. The user receives the file and can see the loading progress of how

much was downloaded. So, users can send files to each other with all exceptions being handled (such as if a user leaves).

2. Another interesting feature is the MoveFlow (it's an Engine feature). Anything that walks in a 2D plane can subscribe to the MoveFlow. What the MoveFlow does is improve the movements of this object. It does a calculation based on a cubic spline algorithm where the acceleration takes the user latency into consideration. An object that is moving horribly bad (due to the network latency) flows through the screen.

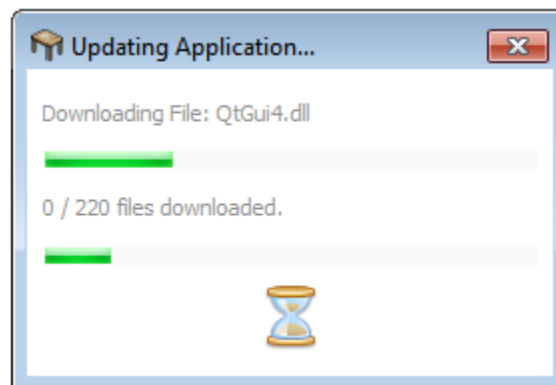


Without the MoveFlow, the items in the table would be moving in the plane at around 1 move per second. The MoveFlow transforms this into X moves per second ( $X = 45$  right now). In other words, with this algorithm any player movement across the world will look like they are playing in the same local network. And the best thing is that any object can subscribe to the Move Flow class in any type of application (that shows how portable the Engine is).

3. The features of the table also deserve some attention. You can rotate the table, the items, zoom in, zoom out etc. But, what is really impressive is that the table size is dynamic. It expands and contracts based on the position of the items. There was some work involved in order to make the window so flexible. If you want more space in the table, the only thing you have to do is to drag the items further away from each other, and the table will expand. If you bring them close again, the table will contract until the scrollbars disappear! And this happens in perfect synchronization through the network. Every user sees exactly the same table size and the positions of the items are in perfect synchrony. Even users who had their table zoomed and rotated still see things in proportion.
4. The image manipulation is also interesting. Images are downloaded from others and saved with the same filename as the original. You can upload different images with the same filename and they will all be in the same folder. When the user sends another image with the same name, it checks all the images you already have with the same name. If the image was found in your computer, you don't download again. In other words, you only download from each other what you don't have already in your table folder. There's no download being wasted. Even if the user

has an image with name “duck (3).png” and this image is exactly the “duck (7).png” you have in your folder, then the item will point correctly to your duck image (otherwise it might create a “duck (8).png”). What happens is that when a user adds a new item to the table, this item comes with a hash of the original image. This hash is compared with the images you already have (perhaps from other rooms with the same room type). If you have the image, you are going to use it. Otherwise, you send a request to everyone and download from the user with the fastest internet. You will not see the same file duplicated in the same table folder (unless this file has a different size). There’s a function that calculates how much percentage your images matched with an image added by the user. Also, there are some restrictions with the image search because it should not take too long. The application’s scalability is good.

5. Finally, it’s important to remember that the application updates itself. Administrators can in seconds publish a new version on the website, and instantly all users will receive a message warning about a new version (with the version notes). They can download this version automatically and the application will replace all the files that were updated and reinitiate it. The magic behind it requires some copy of files back and forth inside one temporary folder in the application: the original application downloads the new one and calls the temporary copy in a folder by passing an argument asking to update the application. The copy of the app replaces files and calls the application again in the main folder. Finally, the application deletes the temporary copy.

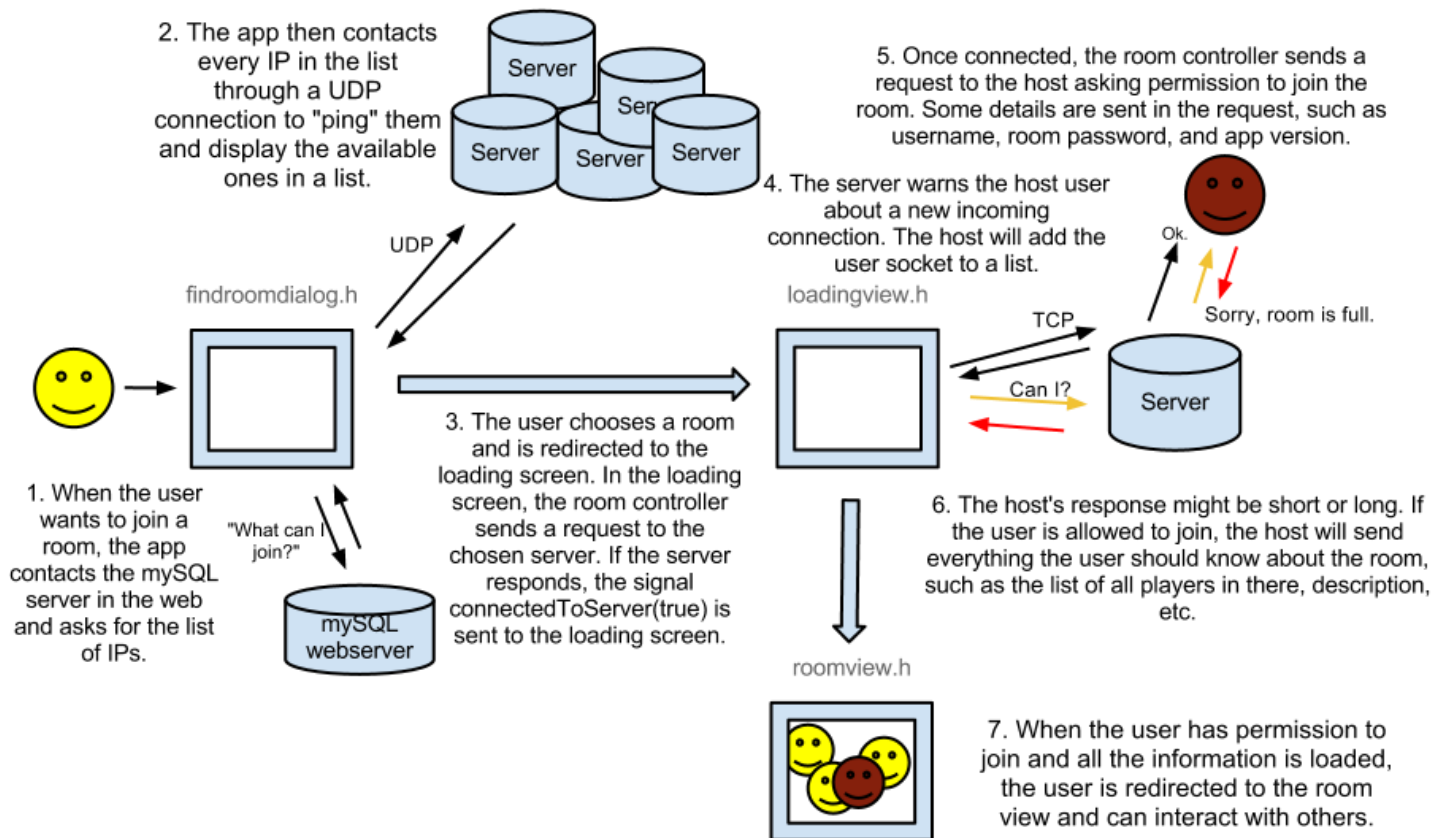


## Conclusion:

The Project Live Table application was successfully built. It can be compiled and executed in both Windows and Mac. To develop this application was initially a challenge because Qt was completely unknown. It took a while to learn how to use the Qt libraries. The project had to be done in less than 3 months, and that includes having to reinvent the wheel for network protocols and file transference. It is an achievement to have finished the application before the deadline. But, the time restraint prevented some more elaborated features, such as timers and drawing pads that could allow even more uses for the table.

Below are the graphs I drew as part of the initial documentation of the application. The app evolved, so they changed slightly, such as different class names. But, their core ideas are still the same.

## Understanding the Network





## Trading of network messages

1. The user writes a chat message to others. The chat message is converted to QByteArray. Then, the method `sendMessageToServer()` is called in the room controller.

"Hello!"

2. The room controller adds a type to the message (qint8) and send the message to the LocalNetwork class.

Type: MSG\_CHAT\_MESSAGE "Hello!"

3. The LocalNetwork class adds the socket descriptor of the user who should receive the message. It adds 0 when the message is for everyone.

To: 0 Type: MSG\_CHAT\_MESSAGE "Hello!"

5. The client object that represents that client in the server-side `qUncompresses` the message and takes the recipient out of the message.

Server ENTRANCE

4. The LocalClient class `qCompresses` the message and writes in the client socket.

0 10011 01

To: 0 Type: MSG\_CHAT\_MESSAGE "Hello!"

6. A signal `networkMessageReceived()` is sent to the LocalServer class.

From : 234 Type: MSG\_CHAT\_MESSAGE "Hello!"

Ps: Every message is written with `QDataStream` that provides serialization. So that, the message is read exactly the same for every host computer.

9. The client receives the message, `qUncompresses` it and sends to the LocalNetwork class. This class removes the sender and emits a signal to the RoomController that reads the message.

From : 234 Type: MSG\_CHAT\_MESSAGE "Hello!"

1 10011 01

8. The server-side client of the recipient will then `qCompress()` the file and write in the socket. If the message is for everyone, every server-side client will repeat this process.

Server EXIT

Ps 2: The network has its own thread.

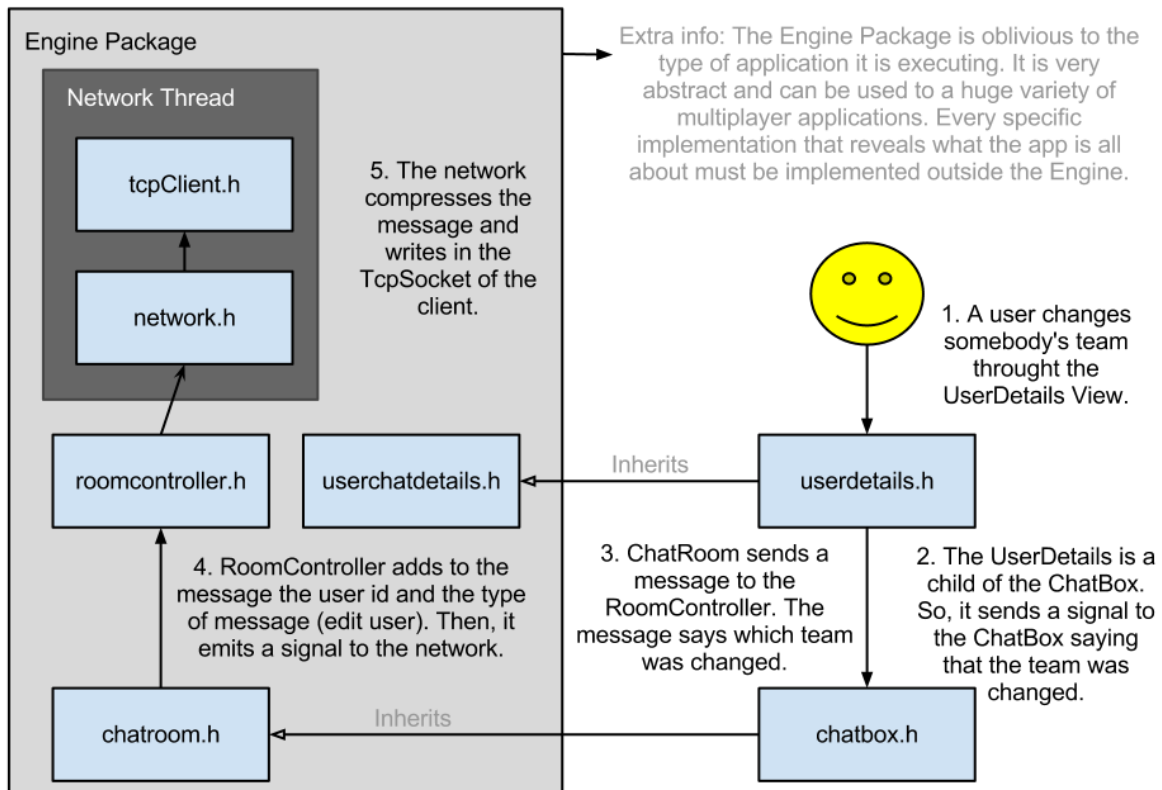
From: John Type: MSG\_CHAT\_MESSAGE "Hello!"

John says "Hello!"

10. The room controller first finds the RoomMember associated with the socket descriptor in the message. Then, it removes the message type to decide how to read the rest of the message.

11. A signal `chatMessageReceived()` is sent. The room can now read the message!

## USER CHANGING SOMEBODY'S TEAM:



## USER RECEIVING MESSAGE THAT HIS TEAM WAS CHANGED:

