Include "../include/UserManager.h" at the very top

-Change cmakefiles so you don't have path's like these. Don't use these relative paths because when you change the location of a file, then you will also have the change the include at the top of the file for every file that used it. You can change your CMakefile so you don't have to do this.

In UserManager::printAllUsers()
line 144 - The line "std::cout << user.second.getConnection().id,
there is a lot of method chaining, perhaps try rewriting some of your methods
so you don't have to chain so many methods together.

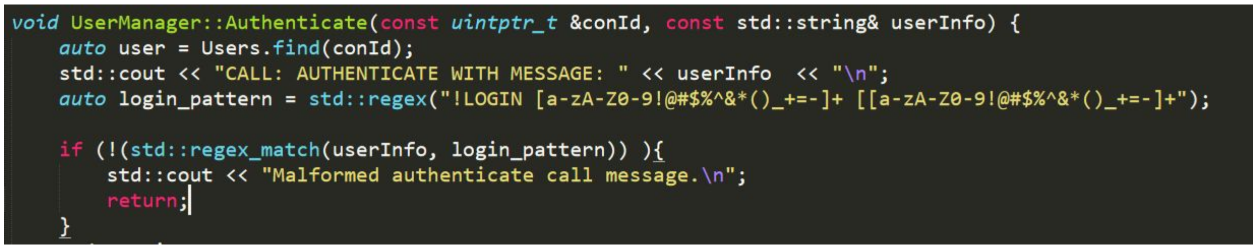
In line 104 - 108
the variable names can be a bit better instead of beg, try iterator or iter, same with tok, these names can be improved as it took me a couple of seconds of thinking to figure out what "beg" was. Another thing I would change is the name Users into listOfUsers because Users is quite close to User and it would be easy to get these to mixed up since they only differ by on character. By having listOfUsers or listUsers it becomes a lot more clear to what that variable is.

Line 81, No error handling for Users.find(). What if you pass in a
-Id for a user that doesn't exist, what will it return? Also If I'm right this code doesn't compile because there is no return outside the if statement, in the function findUser().

One thing I liked how you wrote a description for the UserManager Class at the top which tells you what the class is about, and what it does. Most code I’ve seen don’t write a nice short description of the class itself and what its responsibilities are. I should start telling my team to do this as well.

Code:

Are there performance or security concerns that should be addressed?



```cpp
void UserManager::Authenticate(const uintptr_t &conId, const std::string& userInfo) {
    auto user = Users.find(conId);
    std::cout << "CALL: AUTHENTICATE WITH MESSAGE: " << userInfo  << "\n";
    auto login_pattern = std::regex("!LOGIN [a-zA-Z0-9!@#$%^&*()_+=-]+ [[a-zA-Z0-9!@#$%^&*()_+=-]+");

    if (!(std::regex_match(userInfo, login_pattern)) ){
        std::cout << "Malformed authenticate call message.\n";
        return;
    }
```

I assume the login pattern is the user's password or login keyword that they have used to enter the chat room. I see that you guys are using Regular Expression, so I know that Regex is useful to compare patterns, but there also some disadvantages for it according to: https://www.ibm.com/support/knowledgecenter/en/SS2MBL_9.0.2/cxImpactUser/cxImpact/SrchinPtl/LimitationsOnUseOfRegularExpressions_42.html

Information such as numbers are not searchable by regular expression, so only letters can be searched. Also, multiple words cannot be used. Therefore, depending on your purpose and the way you would like to use Regex, it might be a good approach to use it, but in other cases it might not, so I thought it might be useful to know about it.

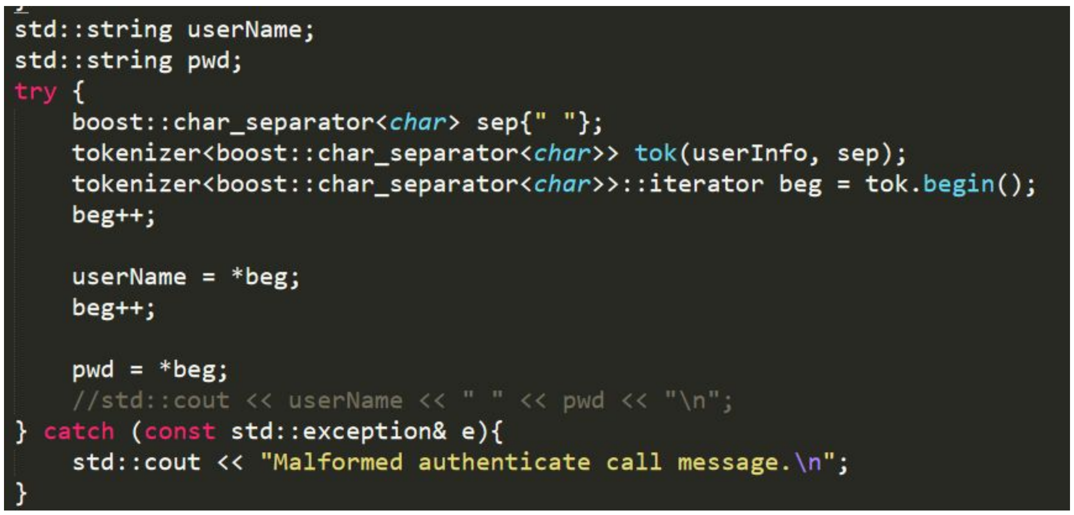
Design:

Does it have a clear, focused responsibility?

Yes, based on the function names and the initial class comment it is possible to understand the purpose of UserManager class which is to handle the incoming information and user authentication from the server.

Do comments clarify or just repeat what the code does? Is the intent of the code clear?

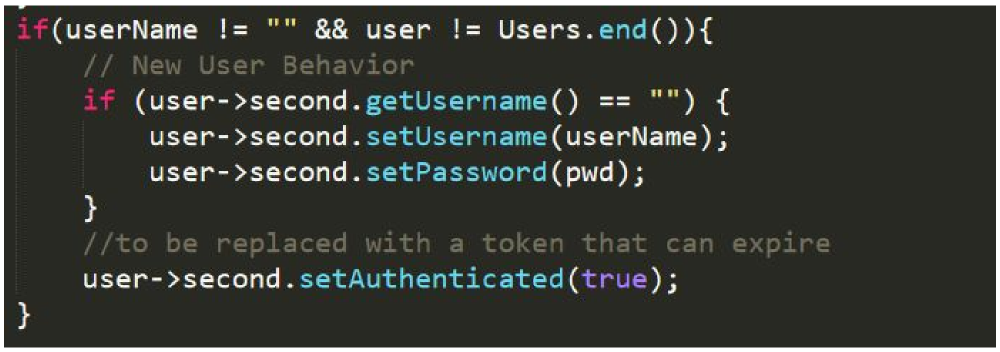
Most functions don't require comments, but the function Authenticate was the most confusing one, I had a hard time understanding what tokenizer<boost::char_separator<char>> and the rest of the code in the try block was trying to do. It is useful to add comments (explanation or just naming the process or concept that is being implemented) to improve readability and understanding.



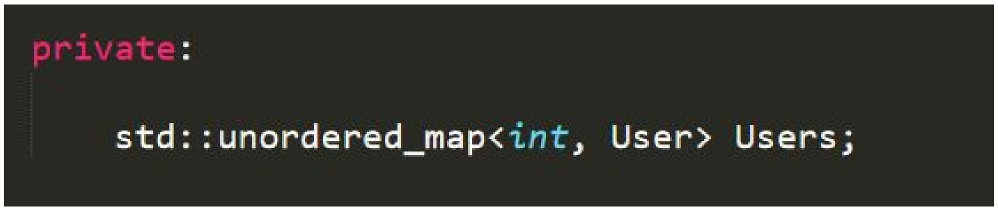
```
std::string userName;
std::string pwd;
try {
    boost::char_separator<char> sep{" "};
    tokenizer<boost::char_separator<char>> tok(userInfo, sep);
    tokenizer<boost::char_separator<char>>::iterator beg = tok.begin();
    beg++;

    userName = *beg;
    beg++;

    pwd = *beg;
    //std::cout << userName << " " << pwd << "\n";
} catch (const std::exception& e){
    std::cout << "Malformed authenticate call message.\n";
}
```

Is the intent of the code clear?

user->second… didn't make sense, what does second represent here and where was it first initialized?



```
if(userName != "" && user != Users.end()){
    // New User Behavior
    if (user->second.getUsername() == "") {
        user->second.setUsername(userName);
        user->second.setPassword(pwd);
    }
    //to be replaced with a token that can expire
    user->second.setAuthenticated(true);
}
```



```
private:

    std::unordered_map<int, User> Users;
```

I also like the fact that you guys used unordered_map<> which is more efficient in terms of performance than map<> because you don't need order in your case, so it smart to choose unordered_map to improve performance.
Reference:
https://www.geeksforgeeks.org/map-vs-unordered_map-c/

1) As discussed in class, it is not a good idea to pass id as an integer as it deeply couples this manager to the User data class. If Id is changed to be a string, most code would have to be refactored. Functions like removeUser/findUser ect are currently coupled. Maybe abstracting it to an ID class with something like a getValue() function might resolve this.

2) In UserManager::Authenticate, I see you are using tokenizer from boost to parse a string which is good code design. Code would look complex if you would parse it yourself with many loops. You many also miss edge cases if you implemented it from scratch. Use the library solves these issues and has good documentation which is helpful for those who do not know the library specifically.

3) In UserManager::Authenticate, your catch statement catches a generic exception which seems suspicious. It might be better to catch the proper exceptions that are created so debugging this function is made easier. At a glance, there is incidental complexity created as it’s unknown where the exception will be created and what caused it.

4) For UserManager::Logout, it seems suspicious that nothing happens if the user fails to logout. An argument can be made that nothing should be done on a failed logout because it should never happen, but it can. By logging error cases, like printing to console that a logout was called with an unknown id, it can help catch bugs in code if a programmer mistakenly triggers this.

Individual Code Review 2

1. Don’t use using directive (line 71)

Avoid the using directive as this may complicate things in the future. A big disadvantage is when you have the same function name in different libraries there can be conflicts and then you must go back and refactor your code unnecessarily. Also, by not using the using directives and using the prefixes objectively add clarity to the code.

2. Unnecessary comments (lines 7-9 or 62-65)

It is good to omit unnecessary comments as they can proliferate as you work on the project and they will make the code harder to read in the future.

3. Include statements (line 66)

Remove include statements with specific paths as they become hard to write and modify code in the future. For example, if a file was moved in the project, the relative path will change. You can fix this by making a proper CMakeLists.txt.

4. Good header comment (lines 20 – 27)

It is very useful to have helpful comments in header files explaining what the class does. Your header comment for UserManager is very good and helps developers quickly identify what the class does without having to read any code and thus saves time.

5. Stylistic issues

It is good to be consistent with styling issues such as naming variables or spacing as it makes code easier to read. One place where I noticed a small issue is on lines 81-83 and lines 90-91 where there is a spacing inconsistency. You need to decide whether you want a small space after the function header or not. Another small issue has to do with inconsistent use of braces. For example, in the for each loop on lines 135-137 you decided to use braces and then the for each on line 143, you decided not to use them.

Code Review for Babka

lines 34-40
- the parameter name 'conId' is not so clear
    - conId looks to be the user ID, however I don't quite see what 'con' is or where it came from
        - My guess is that it stands for connection, but I'm not too sure if that makes sense
    - a better parameter name might make it more readable (ie userId?)

line 48
- member variable 'Users' - why is it a capital 'U'?
- I'm not sure if that's a standard, but it kind of brings confusion
    - when used in line 74, for example, I thought it was a class and insert()) was a static function of that class
- naming it 'users' should probably be fine

line 93
- regex is generally confusing and hard to understand
    - might be a good idea to add a comment on what you're trying to do on that line

lines 32-34
- the naming on the public methods of the class is a little inconsistent
    - how come Authenticate() and Logout() starts with capital letters and the rest not?
    - maybe come up with a standard within the group on how you want to name your functions (and variables in that case)

1.

```
void UserManager::Authenticate(const uintptr_t &conId, const std::string& userInfo) {
    auto user = Users.find(conId);
    std::cout << "CALL: AUTHENTICATE WITH MESSAGE: " << userInfo  << "\n";
    auto login_pattern = std::regex("!LOGIN [a-zA-Z0-9!@#$%^&*()_+=-]+
[[a-zA-Z0-9!@#$%^&*()_+=-]+");

    if (!(std::regex_match(userInfo, login_pattern)) ){
        std::cout << "Malformed authenticate call message.\n";
        return;
    }
    std::string userName;
    std::string pwd;
    try {
        boost::char_separator<char> sep{" "};
        tokenizer<boost::char_separator<char>> tok(userInfo, sep);
        tokenizer<boost::char_separator<char>>::iterator beg = tok.begin();
        beg++;

        userName = *beg;
        beg++;

        pwd = *beg;
        //std::cout << userName << " " << pwd << "\n";
    } catch (const std::exception& e){
        std::cout << "Malformed authenticate call message.\n";
    }
    if(userName != "" && user != Users.end()){
        // New User Behavior
        if (user->second.getUsername() == "") {
            user->second.setUsername(userName);
            user->second.setPassword(pwd);
        }
        //to be replaced with a token that can expire
        user->second.setAuthenticated(true);
    }
}
```

While this implementation is defintely seems functional, design-wise I would argue this design complicates further iterations/improvements to it, most noteably the authentication. I would suggest breaking up this function further.

2.

```
public

...

void UserManager::addUser(User &newUser) {
    Users.insert({newUser.getConnection().id, newUser});
}

void UserManager::removeUser(const uintptr_t &conId) {
    Users.erase(conId);
}

User& UserManager::findUser(const uintptr_t &conId) {

    auto user = Users.find(conId);
    if(user != Users.end()){
        return user->second;
    }

}

...

void UserManager::Logout(const uintptr_t &conId) {
    auto user = Users.find(conId);
    if(user != Users.end()){
        user->second.setAuthenticated(false);
    }
}
```

What is the benefit of exposing the add and remove methods for public API? When this is further developed I imagine your team would want to minimize the APIs to what is required, and as a personal suggestion, I believe the Logout(/Login?) APIs would highly suit the outward APIs better.

3.

```
std::deque<networking::Message> UserManager::buildOutgoing(const std::string& log) {
    std::deque<networking::Message> outgoing;
    for (auto user : Users) {
        outgoing.push_back({user.second.getConnection(), log});
    }
    return outgoing;
}
```

It's defintely helpful to reuse code (our profs in this case). However contextually why does the UserManager needs annouce it's actions to all it's users? Especially tying in to point #2, have a public method to do so. Perhaps repurpose this, or change it's name, or even move it to another class.

4.

Lastly, while the 'cout' method is for early testing purposes, you will have to eventually implement a way to write back to users, and I would highly not suggest using the UserManager to do that. This complicates the responsibilities of the UserManager. I would suggest now designing a UserManager being able to return some notifications, but in a way that makes it not responsible of directly putting it to the Client.

Code review for group: babka

1. addUser, removeUser and Logout methods don’t have return type. You would not know if the user gets added or remove or logout. The methods should return something to indicate the action performed.
2. findUser method cannot run because there is no return in case the if clause doesn’t execute. The alternative would be adding an else clause if cannot find the user or user is not in the table.
3. For the include `#include "../include/UserManager.h"` should be `#include “UserManager.h”`
4. Authenticate user should have return something so the caller knows that if the user is valid. Whereas in the code the caller needs to call user.getAuthenticated() I assume. What happen if user is not Authenticated? I would suggest to make this method return true or false. Return true if the user is authenticated and false otherwise.
5. Method naming issues. Should choose a better name for buildOutgoing method because the name is not clear about what does the method do. An example would be outGoingMessages.
6. Overall I think the price of code is well written. Passing by reference to protect the ownership of variables and objects. Method implementations are straight forward and should work as it is intended to.

1. Is the intent of the code clear?

I believe the intent of the code is clear. The UserManager class has a high-level comment describing what the class does and any responsibilities that will be handled within the class. This is a good starting point for a class design. Someone reading the code does not have to dive into the methods or their implementation to have a general understanding of what the class does. However, there are some issues that I also see with the intention of the methods. For example, in the implementation of findUser on line 81 to line 88, I am not sure what happens in the situation where a user is not found in the map of users. The function returns a User when a user exists in the map however it does not return anything when a user is not found. This begs the question, what is the return value in this context? And how is a client of the method supposed to handle such a use case? Definitely some points to consider for this method.

2. Does it have a clear, focused responsibility?

I believe that for the most part the class along with its methods and their implementations have a single, clear and focused responsibility. The method names match what the client would expect the methods to do when they are used and this is a good thing to have in the methods of a class. However, there is a particular method Authenticate on line 90 to line 124 that seems to have more than one responsibility. Given the method name Authenticate as a client I would expect the method to simply check that the user info provided matches the authentication standards set for the application and returns a response telling the client whether the authentication has succeeded or failed. From this method I can tell that this is being done in the first half of the method however in the second half there seems to be updating of the username and password which begs the question of whether or not this should be done by another method. It would make more sense if there was a login method that called within its implementation an authenticate method which would mean the client would use the login method with the idea that authentication of their credentials is a part of that process.

3. Specifically, does the interface hide design details to ease its use?

The UserManager interface does a pretty good job of hiding design details for its use. Someone looking at the header file can instantly get an idea of what the interface does and how to use its methods. A few things I would add is a const at the end of the methods printAllUsers, findUser and buildOutgoing to let the client know that this method will not change the calling object. This would be important for the client to be aware of and also to have the safety of knowing that these methods will not end up modifying the client in anyway. Such details I believe are important to show in an interface.

4. Do comments clarify or just repeat what the code does?

The commenting throughout the code snippet clears up any questions a reader may have. The code does well in highlighting potentially confusing areas of code. To add to the comments I would also advise refactoring by breaking up big methods like Authenticate into smaller methods with single responsibilities.