Grafische Benutzeroberfläche mit fltk

Prof. Dr. Rethmann & Jochen Peters

7. Oktober 2015

Inhaltsverzeichnis

1 Einleitung **2**

2 fltk 1.3.x **2**
2.1 Installation 3
2.1.1 Linux 3
2.1.2 Os X 3
2.1.3 Windows 3
2.2 Die Make-Skripte 4

3 Mini-Protokoll (Applikations-Schicht) **4**

4 Die Dateien **5**
4.1 chatterServer.cpp - der Server 5
4.2 chatter.cpp - der Client 7

5 Verbesserungswürdig **9**

Warnung: Die hier gezeigte Software ist nur zu Lernzwecken erstellt worden und die Autoren übernehmen keine Verantwortung für Schäden, die durch die Verwendung der Software entstehen. Verbesserungsvorschläge und Bug-Reports sind willkommen.

1 Einleitung

Die Dateien *chat-gui.fl*, *chatter.cpp* und *chatterServer.cpp* dienen zusammen mit den drei Make-Skripten als weiteres Beispiel, wie Sie unter Linux, Os X und Windows zusammen mit UniSocket ein Chat-Programm realisieren können. Für die grafischen Komponenten ist in diesem Beispiel das fltk-Framework zuständig, da dieses auf allen drei Betriebssystemen nutzbar ist.

Dieses Beispiel erfüllt nicht alle Anforderungen, die für ein Projekt in der Lehrveranstaltung „Verteilte Systeme“ gefordert sind, wie z.B. Ausfallsicherheit, Nebenläufigkeit und Transparenz. Ein MVC-Muster sucht man auch vergebens - im Chat-Client fehlt also jede Trennung von Funktion und User-Interface. Aber es kann als Basis für ein verteiltes System dienen und ausgebaut werden.

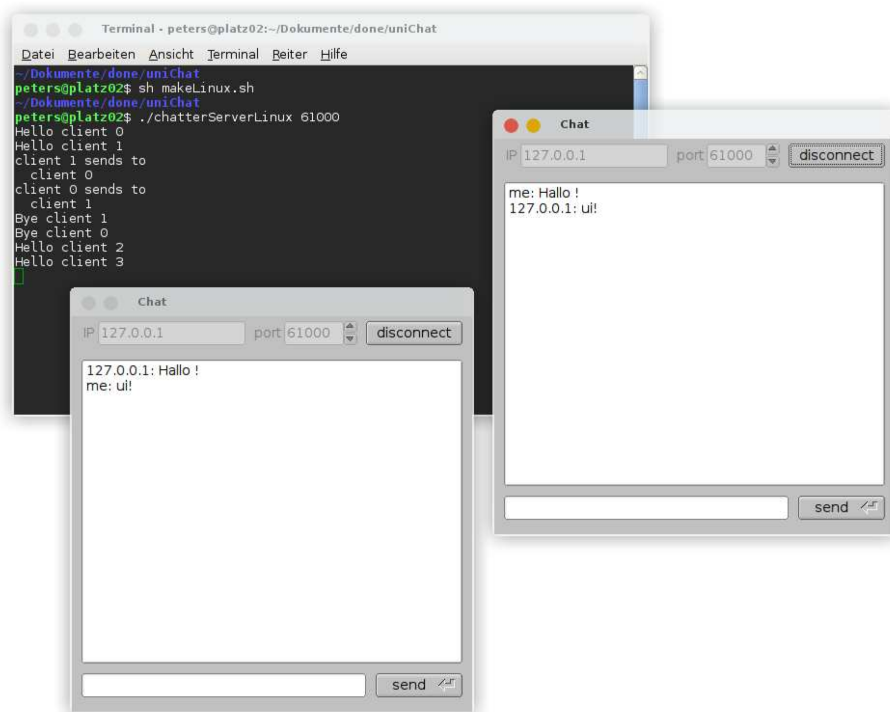

Abbildung 1: chatter - Beispiel eines GUI-Chat-Client

2 fltk 1.3.x

Leider sind keine GUI-Bibliotheksfunktionen in der Standardbibliothek von C++ enthalten. Daher verwenden wir das GUI-Framework fltk „Fast Light Tool Kit“ zur Erstellung von einfachen grafischen Oberflächen in C++. Das Programm *fluid* ist ein wichtiger Bestandteil von fltk. Damit können die GUI-Elemente von fltk für ein Fenster zusammengestellt werden. *fluid* als Editor speichert diese Zusammenstellung in einer *.fl* Datei ab. Ruft man `fluid` mit dem Parameter `-c` in der Kommandozeile auf, wird ein C++ Code und eine Header-Datei erzeugt, die die Klasse für das GUI enthält.

2.1 Installation

2.1.1 Linux

Unter Debian/Ubuntu z.B. mittels Muon-Paketmanager, oder:

```
sudo apt-get install libfltk libfltk-dev
```

Unter Archlinux:

```
sudo pacman -S fltk
```

2.1.2 Os X

Wir empfehlen Ihnen, vor der Installation von fltk zunächst das Programm Mac-Ports zu installieren (siehe `https://www.macports.org/`). Anschließend können Sie in einem Terminal fltk, ähnlich wie unter Linux, mit dem folgenden Befehl installieren:

```
sudo port install fltk
```

2.1.3 Windows

Eine gute Beschreibung, wie man fltk unter Windows installiert, ist in der README-Datei enthalten, die man zusammen mit dem fltk-Quellcode herunterlädt:

> `http://www.fltk.org/software.php`

Hier ein frei übersetzter Auszug aus der **README.MSWindows.txt**:

> Starte VisualStudio. Öffne die Projektdatei
>
> `...\fltk-1.3.xxxx\ide\VisualC2010\fltk.sln`
>
> Wähle in der „Projektmappenkonfiguration" den Modus „Debug" oder „Release" aus. [...]
>
> Legen Sie mit dem Kontext-Menü das **demo**-Projekt „als Startprojekt" fest. Als Nächstes wähle „Projektmappe erstellen" aus dem „ERSTELLEN" Menü oder drücke F7 um alle Bibliotheken zu erstellen. [...]
>
> Die Standard Installation der VisualC 2010 Bibliotheken und Header-Dateien befindet sich hier:
>
> `C:\Program Files\Microsoft Visual Studio 10.0\VC\`
>
> Es ist möglich alle FLTK Bibliotheken, Header-Dateien, und Fluid in entsprechende Unterordner zu kopieren, so dass diese bei zukünftigen Entwicklungen zur Verfügung stehen ohne im Projekt Linker und Include-Pfade hinzufügen zu müssen.
>
> - kopiere den gesamten **FL**-Ordner in den `VC\include\`-Ordner

- kopiere alle **.lib**-Dateien aus dem fltk `lib\`-Ordner in den `VC\lib\`-Ordner
- Kopiere `fluid.exe` vom fltk `fluid\`-Ordner in den `VC\bin\`-Ordner

Dieser Text bezieht sich zwar auf VisualStudio 2010, aber in 2012 und 2013 ließ sich die Projektdatei ebenfalls öffnen. Anstelle des obigen Pfades muss für Visual Studio 2012 der Pfad `\Microsoft Visual Studio 11.0\` und für 2013 der Pfad `\Microsoft Visual Studio 12.0\` verwendet werden.

2.2 Die Make-Skripte

Der Einfachheit halber wurde auf shared libraries verzichtet, wie es in der Beschreibung und den Compiler-Aufrufen des „UniSocket Wrappers" der Fall war. So lässt sich der Client unter Windows sehr leicht ausführen: man braucht nur die 32-bit *exe*-Datei und die *dll*-Datei des genutzten Compilers, und man kann Chatten. Wer sich darüber ärgert, warum man die *dll*-Datei von Microsoft noch zusätzlich braucht, dem sei gesagt, dass auch beim Firefox-Browser (April 2015) zur Sicherheit eine *msvcr120.dll* im Programm-Ordner beigelegt wurde. Näheres dazu finden Sie in der Anleitung zu *UniSocket*.

Die Skripte sind nicht viel anders, als beim Beispiel zu *UniSocket*. Hinzugekommen ist natürlich `-lfltk`, um die Bibliotheken von fltk nutzen zu können. Unter Os X ist ein Include-Pfad und ein Library-Pfad nötig, da Mac-Ports die Header- und die Library-Dateien nicht an einer üblichen Stelle ablegt:

```
1  #!/bin/bash
2  ############# Beispiel makeOsx.sh ################
3  fluid -c chat-gui.fl
4  g++ -Wall -c chat-gui.cxx -I/opt/local/include
5
6  g++ -Wall -c UniSocket.cpp
7  g++ -Wall -c SocketWrapperOsx.cpp
8
9  g++ -Wall -std=c++11 -c chatter.cpp -I/opt/local/include
10 g++ -Wall -std=c++11 -o chatterOsx chatter.o chat-gui.o \
11         UniSocket.o SocketWrapperOsx.o -lfltk -L/opt/local/lib
12
13 g++ -Wall -std=c++11 -c chatterServer.cpp
14 g++ -Wall -std=c++11 -o chatterServerOsx chatterServer.o \
15         UniSocket.o SocketWrapperOsx.o
```

Am auffälligsten ist der Aufruf von `fluid` in Zeile 3, was zusammen mit dem Parameter `-c` die *.fl* Datei übersetzt. Dieses Übersetzen erzeugt eine *.cxx* und eine *.h* Datei, welche mit dem nächsten Befehl in Zeile 4 zu einer Objekt-Datei kompiliert werden. Diese muss dann beim Linker in Zeile 10 und 11 mit angegeben werden, wenn man die *Executable* der GUI erzeugt.

Unter Windows ist besonders das Einbinden/Linken mit unzähligen *.lib*-Dateien zu beachten.

3 Mini-Protokoll (Applikations-Schicht)

Da *UniSocket* ohne einen Timeout für das Empfangen arbeitet, muss zwischen Client und Server ein Dialog vor dem `UniSocket::close()` stattfinden. Wenn Sie einen Timeout einbauen

wollen, müssen Sie sich überlegen, ob Sie mit einem regelmäßigen „Ping-Pong“ in Ihrem Anwendungsprotokoll die Verbindung aufrecht erhalten wollen, oder lieber mit einem *close()* die Verbindung beenden. Im letzteren Fall müssen Sie sich Gedanken darüber machen, wie ihr Client an seine Nachrichten kommt. Bedenken Sie, dass die Lösung mit dem „Ping-Pong“ in der Lage wäre, die Verfügbarkeit des Servers auf Ihrer implementierten Anwendungsschicht prüfen zu können – ein *accept()* des Servers bedeutet nicht automatisch, dass dieser auch sofort Ressourcen zur Verarbeitung bereit stellt.

In unserer Beispiel-Anwendung sieht das Protokoll zum Beenden des Clients wie folgt aus:

- Man klickt beim Client auf den disconnect-Button.
- Der Client merkt sich in der Variable `tryClose`, dass er ein close() machen will.
- Der Client sendet den Text `[close]`.
- Der Server empfängt vom Client `[close]`.
- Der Server sendet ein `[bye]` an den Client und macht ein `UniSocket::close()`.
- Der Client empfängt ein `[bye]` und schaut in die Variable `tryClose`, ob er sich wirklich beenden wollte. Sollte dies nicht der Fall sein, wird das `[bye]` ignoriert.[1]
- Der Client springt aus der Endlos-Schleife, in der eingehende Nachrichten vom Server verarbeitet wurden.
- Der Client macht ebenfalls ein `UniSocket::close()`.

4 Die Dateien

4.1 chatterServer.cpp - der Server

Der Server hat keine GUI und sein Code unterscheidet sich nur wenig vom Code in der Datei *server.cpp*, welche als Beispiel bei *UniSocket* mitgeliefert wird. Nur drei Dinge sind verändert worden:

- Die UniSockets werden je Client in einer Map abgelegt.
- Eine Funktion durchläuft die Map, um die Nachricht jedem Client zu senden (außer an den, der die Nachricht gesendet hat).
- Die Thread-Funktion, die je Client auf Nachrichten horcht, wurde für ein Mini-Protokoll erweitert.

Der letzte Punkt ist erforderlich gewesen, weil im ursprünglichen Client/Server-Beispiel von UniSocket nur je eine Nachricht gesendet und empfangen wurde, die Verbindung direkt geschlossen und der „horchende“ Thread beendet wurde.

```
1  #include <thread>
2  #include <mutex>
3  #include <iostream>
4  #include <map>
5  #include <string>
6  #include "UniSocket.hpp"
7  using namespace std;
8
9  mutex mtx;
```

[1] Im Kapitel „Verbesserungswürdig“ ist beschrieben, welche Probleme hierbei entstehen können.

```
10  map<int, UniSocket> clientPool;
11
12  void sendAllOthers(string msg, int id) {
13      cout << "client " << id << " sends to " << endl;
14      map<int, UniSocket>::iterator it;
15      mtx.lock();
16      for (it = clientPool.begin(); it != clientPool.end(); ++it) {
17          if (it->first == id)
18              continue;
19          cout << "  client " << it->first << endl;
20          (it->second).send("client " + to_string(id) +
21              " (" + (it->second).getIp() + "): " + msg
22          );
23      }
24      mtx.unlock();
25  }
26
27  // jeder client sein eigener Thread!
28  void sockHandle(UniSocket usock, int myId) {
29      cout << "Hello client " << myId << endl;
30      mtx.lock();
31      clientPool[myId] = usock;
32      mtx.unlock();
33      string msg;
34      while (true) {
35          msg = usock.recv();
36          if (msg == "[close]")
37              break;
38          sendAllOthers(msg, myId);
39      }
40      cout << "Bye client " << myId << endl;
41      usock.send("[bye]");
42      usock.close();
43      mtx.lock();
44      clientPool.erase(myId);
45      mtx.unlock();
46  }
47
48  int main(int argc, char * argv[]) {
49      int idCounter = 0;
50
51      if (argc < 2) {
52          cout << "usage: " << argv[0] << " <port>" << endl;
53          return 1;
54      }
55
56      try {
57          UniServerSocket svr(atoi(argv[1]), 5, "[ENDE]");
58          while (true) {
59              idCounter++;
60              thread t(sockHandle, svr.accept(), idCounter);
61              t.detach();
62          }
```

```
63      } catch(UniSocketException e) {
64          cout << e._msg << endl;
65      }
66      return 0;
67  }
```

Da der Container **map** nicht thread-safe ist und es daher ein Problem geben kann, wenn mehrere Threads auf die Map zugreifen, ist eine Mutex-Variable `mtx` zur Sicherung von kritischen Abschnitten genutzt worden. Mutex-Semaphore sind im 2011er-Standard von C++ aufgenommen worden.

4.2 chatter.cpp - der Client

Der Client besteht aus einer main-Funktion, einem „horchendem“ Thread und zwei Callback-Funktionen für die zwei Buttons der GUI.

```
1   #include "chat-gui.h"
2   #include <iostream>
3   #include <string>
4   #include <thread>
5   #include <mutex>
6   #include "UniSocket.hpp"
7   using namespace std;
8
9   UserInterface ui;
10  UniSocket * sock;
11  Fl_Text_Buffer * buf;
12
13  mutex mtx;
14  bool tryClose;
15
16  // Thread, der dauerhaft auf eingehende Daten wartet
17  void doReceive(void) {
18      while (true) {
19          string msg = sock->recv();
20          mtx.lock();
21          if (msg == "[bye]" && tryClose) {
22              mtx.unlock();
23              break;
24          }
25          mtx.unlock();
26          msg += "\n";
27          buf->append(msg.c_str());
28          // zeichnet GUI-Elemente neu, falls Inhalte neu sind
29          Fl::check();
30      }
31      ui.connectBtn->label("connect");
32      sock->close();
33  }
34
35  // wird beim Klick auf den connect- bzw. disconnect-Button aufgerufen
36  void doConnect(Fl_Widget * obj, void *) {
37      string lbl = obj->label();
```

```
38      if (lbl == "connect") {
39          try {
40              sock = new UniSocket(
41                  ui.ipInput->value(),
42                  ui.portInput->value(),
43                  "[ENDE]"
44              );
45              mtx.lock();
46              tryClose = false;
47              mtx.unlock();
48              thread t(doReceive);
49              t.detach();
50
51              // bei "connect" passende GUI-Elemente ein/ausschalten
52              ui.msgInput->activate();
53              ui.sendBtn->activate();
54              ui.ipInput->deactivate();
55              ui.portInput->deactivate();
56              // der Connect-Button wird umbenannt
57              obj->label("disconnect");
58          } catch(UniSocketException e) {
59              e._msg += "\n";
60              cout << e._msg;
61              buf->append(e._msg.c_str());
62          }
63      } else {
64          // bei "disconnect" passende GUI-Elemente ein/ausschalten
65          ui.msgInput->deactivate();
66          ui.sendBtn->deactivate();
67          ui.ipInput->activate();
68          ui.portInput->activate();
69          // Setzen der Markierung fuer Empfangs-Thread, welcher
70          // auf ein [bye] vom Server wartet ...
71          mtx.lock();
72          tryClose = true;
73          mtx.unlock();
74          // ... nachdem "ich" Server ein [close] gesendet habe:
75          sock->send("[close]");
76      }
77  }
78
79  // wird beim Klick auf den send-Button aufgerufen
80  void doMsg(Fl_Widget * obj, void *) {
81      sock->send(ui.msgInput->value());
82      string ret = "me: " + string(ui.msgInput->value()) + "\n";
83      buf->append(ret.c_str());
84      // Eingabefeld leeren
85      ui.msgInput->value("");
86  }
87
88  // Erzeugung notwendiger Objekte und Konfiguration
89  int main(void) {
90      Fl_Double_Window * win = ui.make_window();
```

```
 91        Fl::scheme("gtk+");
 92        // Die Chat-Log Ausgabe wird mit einem Textbuffer verbunden
 93        buf = new Fl_Text_Buffer();
 94        ui.chatLog->buffer(buf);
 95
 96        tryClose = false;
 97        // fuer die Events der Buttons werden
 98        // Callback-Funktionen hinterlegt
 99        ui.connectBtn->callback(doConnect);
100        ui.sendBtn->callback(doMsg);
101
102        win->show();
103        // Starten der GUI-Endlos-Schleife
104        return Fl::run();
105    }
```

Frage: Tritt überhaupt eine Race-Condition bei `tryClose` auf? *doReceive()* sollte doch nur einmal laufen? Leider kann man nicht ausschließen, dass der Anwender mehrfach auf den Connect/Disconnect Button klickt, was mehrere Empfangs-Threads startet. Spielen Sie dies in Gedanken durch – Was kann noch passieren?

Die Aufrufe von `Fl::scheme()` und `Fl::run()` sowie `Fl::check()` sind eine Besonderheit, da fltk genauso wie eine ältere Version von *openGL* eine „State Machine“ ist, siehe dazu auch:

`http://www.glprogramming.com/red/chapter01.html#name4`

Falls Sie der Meinung sind, dass man das hier vorgestellte noch viel schöner hätte implementieren können, dann haben Sie völlig Recht. Im folgenden Abschnitt geben wir einige Vorschläge zur Verbesserung.

5 Verbesserungswürdig

Konzepte wie Publish-Subscriber, MVC oder ein Event-Manager wären angebracht. Bei einem größeren Projekt ist diese Modularisierung sinnvoll: Warum sollte man nicht in der Callback-Funktion eines Button-Clicks den Connect und Disconnect Code ablegen? Warum sollte man nicht den Text im Button als Indikator nehmen, ob man gerade verbunden oder nicht verbunden ist? Weil das zu Problemen führen würde, falls man z.B. auf Qt umsteigen möchte oder die Button-Beschriftungen internationalisiert werden sollen. Es schleichen sich schnell Fehler ein, wenn Sie Code hin und her kopieren und dazwischen GUI-Komponenten anpassen. Die Versuchung Code anzupassen, der mit der Darstellung nichts zu tun hat, ist ebenfalls hoch. Unter Linux gibt es bei einigen Programmen auch die Möglichkeit, auf ein Command-Line-Interface zu wechseln. Das wäre alles elegant lösbar, wenn man Darstellung und Logik trennt.

Ein einzelnes Objekt, welches als Manager zwischen *Chat* und *GUI* agiert, um z.B. der *GUI* einen Modus “disconnected” mitzuteilen, oder *Chat* die zu übertragene Nachricht zu übergeben, ist deutlich professioneller. Die globalen Variablen in *chatter.cpp* ließen sich auf diese Art ebenfalls vermeiden.

Es ist außerdem *fast* kein Protokoll definiert. Normalerweise ist auf dem Server für einen Client eine Session hinterlegt oder der Client fragt regelmäßig den Server, ob neue Nachrichten vorhanden sind. Außer bei Websockets ist es eher unüblich, dass eine Socket-Verbindung

beliebig lange offen bleibt. Eine Fehlerbehandlung fehlt, wenn z.B. das gesendete `[bye]` vom Client ignoriert wird. Sollte der Server ein *close()* machen, wird ein Aufruf zum Empfangen fehlschlagen.