Universaler Socket

Prof. Dr. Rethmann & Jochen Peters

13. Oktober 2015

Inhaltsverzeichnis

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

Wie in der Lehrveranstaltung *Verteilte Systeme* bereits besprochen, stellt nahezu jedes Betriebssystem Sockets zur Verfügung, eine Programmierschnittstelle (API: Application Programming Interface), damit Programme auf das Netzwerk zugreifen zu können. Leider sind die API-Funktionen nicht immer 100% identisch. Hinzu kommt, dass die wichtigen Socket-Funktionen und -Strukturen in C in ihrem Zusammenspiel sehr umständlich wirken. Ein C++-Objekt, das den Socket mit seinen Attributen wie z.B. die IP-Adresse und den Methoden wie *send()* und *recv()* darstellt, ist in seiner Verwendung deutlich angenehmer. Dieses Konzept, welches C-Funktionen in eine zusammengehörende Klasse abbildet, ist ein **Wrapper**. Da im Extremfall jedes Betriebssystem eigene Strukturen zum Binden an einen Port und eine IP-Adresse und andere Funktionen bspw. zum Senden und Empfangen anbietet, muss hier ein anderes, zweites **Wrapper**-Konzept greifen und eine passende Abstraktion bieten: Die Schnittstelle ist auf jedem Betriebssystem gleich, aber die Implementierung spezifisch für jedes Betriebssystem. Diesen Wrapper stellen wir in einer Klasse `UniSocket` zur Verfügung.

Warum haben wir UniSocket entwickelt? Um die Kenntnisse aus den Lehrveranstaltungen EPR (Einführung in die Programmierung), OOA (Objektorientierte Anwendungsentwicklung) sowie VSY (Verteilte Systeme) zu vertiefen und um das Wrapper-Konzept nicht nur theoretisch in der Vorlesung zu behandeln sondern um es praktisch umzusetzen. Es ist ein gutes Beispiel für eine plattformunabhängige Entwicklung, bei der betriebssystemspezifische Funktionen verborgen bleiben. Außerdem wird hier das Konzept des dynamischen Bindens von Programmen an Bibliotheken verdeutlicht und gezeigt, wie man den Visual-Studio-Compiler in einem Batch-Skript aufrufen kann.

Die ursprüngliche Idee war es, dass die Studierenden eine Header-Datei und (je Betriebssystem und Compiler) eine Shared-Library bekommen. So könnten die Studierenden in Ihrem Code `UniSocket` via einer Header-Datei nutzen, und der Linker sowie die *Executeable* greifen auf die Shared-Library zu. Aber:

1. Es ist zwar möglich alle Deklarationen in eine Header-Datei zu schreiben, aber das würde die übersichtliche Trennung von potenziell betriebssystemabhängigen Programmteilen erschweren. Aktuell sind pro Betriebssystem zwei Header-Dateien nötig, von der aber nur eine im eigenen Programm eingebunden werden muss.

2. Eine einzelne Shared-Library wie unter Linux oder Os X, die man sowohl zum Binden als auch zum Ausführen nutzen kann, gibt es unter Windows nicht. Dort wird für den Linker eine *.lib*-Datei und für die *Executeable* eine passende *.dll*-Datei benötigt: Sie kennen dies bereits aus den Vorlesungen OOA und GRA bei der Verwendung von OpenCV unter Windows.

2 Aufbau von UniSocket

Bevor wir auf den Aufbau der Klassen `UniSocket` und `UniServerSocket` genauer eingehen, wollen wir zunächst einige grundlegenden Eigenschaften der Klassen beschreiben.

- Um das Ende einer Nachricht zu erkennen, arbeitet die Klasse `UniSocket` mit einer End-Markierung anstelle einer Längenangabe in einem Header, so wie es auch in der Vorlesung VSY vorgestellt wurde.

- Die Methode *accept()* von `UniServerSocket` gibt ein Objekt der Klasse `UniSocket` zurück, welches die Verbindung zum Client repräsentiert.
- Die Methode *getIp()* von `UniSocket` liefert die IP-Adresse des Verbindungspartners.
- `UniSocket` bietet die Methoden *send()* zum Senden und *recv()* zum Empfangen einer Nachricht sowie *close()* zum Schließen des Sockets und dem Beenden der Kommunikation an.

Der prinzipielle Aufbau der Klassen `UniSocket` und `UniServerSocket` ist in Abbildung 1 dargestellt. Ein `#ifdef` in der Datei *UniSocket.hpp* sorgt dafür, dass je Betriebssystem eine

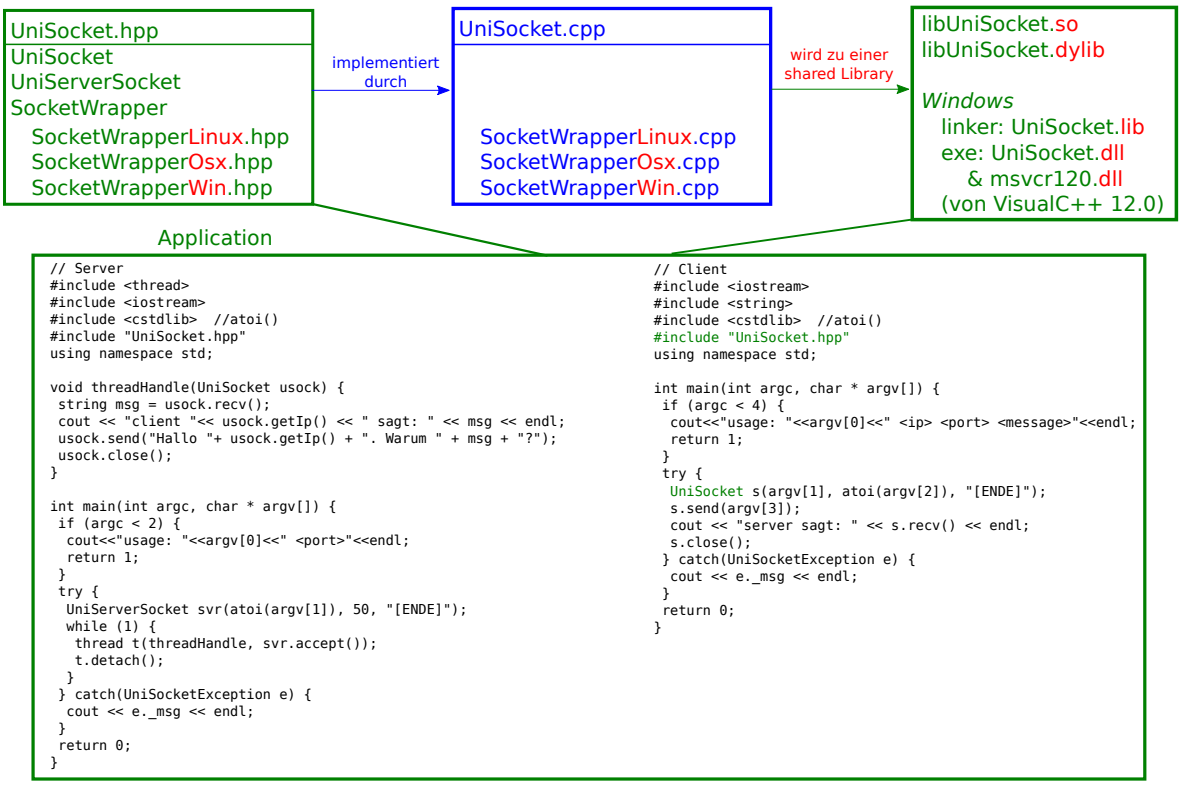

Abbildung 1: Aufbau von UniSocket

passende Datei *SocketWrapperXXXX.hpp* eingebunden wird. Da Sie in Ihrem Projekt nur mit der `UniSocket`- und `UniServerSocket`-Klasse arbeiten sollen, brauchen Sie nur die Datei *UniSocket.hpp* einbinden. Die zum Betriebssystem passende SocketWrapper-Header-Datei muss aber auch im Projekt vorhanden sein – sonst schlägt dieses betriebssystemabhängige Einbinden fehl.

Streng genommen sind die Dateien *SocketWrapperXXXX.cpp* und *UniSocket.cpp* für Ihr Projekt nicht nötig, da dieser Code bereits kompiliert in den Dateien *libUniSocket.dylib* (für Os X), *libUniSocket.so* (für Linux) sowie *UniSocket.lib* und *UniSocket.dll* (für Windows) vorliegt. Aber je nachdem wie und wo Sie Ihr Projekt kompilieren, ob z.B. als 32-bit oder 64-bit oder statt mit dem VisualC++ 2012 mit dem VisualC++ 2011 Compiler, sind diese Shared-Libraries nicht zu gebrauchen und Sie müssen eigene erstellen. Unter Windows wird bei *.lib*- und *.dll*-Dateien eigentlich noch zwischen Release und Debug unterschieden sowie zwischen Desktop, App und WindowsPhone.

Die Dateien *client.cpp* und *server.cpp* sind zusammen mit den *makeXXXX*-Dateien nur Beispiele, wie Sie UniSocket verwenden können und wie Sie unter Windows, Linux und Os X selber Shared-Libraries erzeugen und via Kommandozeile, Batch oder Shell nutzen können.

3 Nutzung von UniSocket und UniServerSocket

Die Abbildung 2 zeigt ein Beispiel für eine Client- und eine Server-Anwendung, dessen Code in den Dateien *client.cpp* und *server.cpp* zu finden ist. Die make-Skripte erstellen entsprechende ausführbare Programme daraus (z.B. *clientLinux* und *serverWin.exe*).

Abbildung 2: Screenshot des Beispiels

Dieses Beispiel ist sehr minimalistisch, aber Sie können es als Grundlage für ein eigenes Projekt nutzen. Der Ablauf innerhalb vom Client und vom Server sowie die Kommunikation zwischen beiden ist im Folgenden beschrieben:

- Auf der Seite des Servers wird der Konstruktor des Socket-Wrappers `UniServerSocket` aufgerufen. Dieser reserviert sich mit der Methode *listen()* den Port, der über die Kommandozeile angeben wurde.
- Danach kann sich ein Client mit dem Server verbinden.
- Damit gleichzeitig mehrere Anfragen bearbeitet werden können, wird die Bearbeitung einer Anfrage vom Server in einen Thread ausgelagert:
  - Der Server läuft in einer Schleife und nimmt innerhalb eines Schleifendurchlaufs mit *accept()* die Verbindung entgegen und lagert die Bearbeitung der Verbindungsanfrage in einen Thread aus.
  - Der Thread bekommt als Parameter den UniSocket übergeben, der von *accept()* geliefert wird.
  - Mit *recv()* wartet der Thread auf ein Datenpaket.
- Der Client sendet mit *send()* eine Nachricht an den Server und wartet mit *recv()* auf eine Antwort.
- Der Thread des Servers empfängt die Daten, sendet eine Antwort und schließt die Verbindung mit *close()*.

- Der Client empfängt die Antwort und schließt auf seiner Seite die Verbindung mit einem *close()* und beendet sich.
- Der Thread beim Server beendet sich.
- Das *accept()* in der while-Schleife des Servers wartet auf weitere Verbindungen.

Vorsicht: *recv()* nutzt blockierendes Lesen. Sollten im Netzwerk die TCP-Pakete vom *send()* (Client an Server) verloren gehen, dann würde auf beiden Seiten auf eine Antwort des Verbindungspartners gewartet. Denken Sie sich in Ihrem eigenen Projekt eine Möglichkeit aus, wie Sie solchen Problemen mit einem Timeout, einem regelmäßigem close/reconnect und einer Session-ID begegnen können!

3.1 Beispiel-Server

Für die jeweiligen Betriebssysteme wird der Beispiel-Server wie folgt in einer Kommandozeile aufgerufen:

- unter Microsoft Windows: `.\serverWin.exe <Port>`
- unter Mac Os X: `./serverOsx <Port>`
- unter Linux: `./serverLinux <Port>`

Dabei bezeichnet `<Port>` die Port-Nummer, unter der der jeweilige Server auf Anfragen horchen soll. In den folgenden Abschnitten beschreiben wir das Client- und das Server-Programm. Beginnen wollen wir mit dem Server.

3.1.1 server.cpp

In den Zeilen 25 und 26 erzeugt der folgende Multithreaded-Server für jede eingehende Verbindung einen Thread. Dies geschieht so: Das erzeugte UniServerSocket-Objekt `svr` bekommt im Konstruktor den Port, eine Puffer-Größe, die die maximal zulässige Anzahl an gleichzeitig eingehenden Verbindungsanfragen repräsentiert, und die End-Markierung übergeben. In einer Endlos-Schleife wird bei jeder eingehenden Verbindungsanfrage ein Verarbeitungs-Thread *threadHandle()* gestartet. Dieser bekommt als Parameter das UniSocket-Objekt übergeben, das von *svr.accept()* erzeugt wird. Der Aufruf *svr.accept()* blockiert solange, bis von einem Client eine Anfrage vorliegt.

```
1  #include <thread>
2  #include <iostream>
3  #include <cstdlib>                // for function atoi()
4  #include "UniSocket.hpp"
5  using namespace std;
6
7  // Achtung: Blockierendes lesen!
8  // thread beendet sich nicht, wenn keine Daten empfangen werden.
9  void threadHandle(UniSocket usock) {
10     string msg = usock.recv();
11     cout << "client " << usock.getIp() << " sagt: " << msg << endl;
12     usock.send("Hallo "+ usock.getIp() + ". Warum " + msg + "?");
13     usock.close();
14 }
15
```

```
16  // Achtung: keine Fehlerbehandlung!
17  int main(int argc, char * argv[]) {
18      if (argc < 2) {
19          cout << "usage: " << argv[0] << " <port>" << endl;
20          return 1;
21      }
22      try {
23          UniServerSocket svr(atoi(argv[1]), 50, "[ENDE]");
24          while (1) {
25              thread t(threadHandle, svr.accept());
26              t.detach();
27          }
28      } catch(UniSocketException e) {
29          cout << e._msg << endl;
30      }
31      return 0;
32  }
```

Wir haben in diesem Beispiel aus didaktischen Gründen auf eine Fehlerbehandlung verzichtet, um den Blick auf das Wesentliche nicht zu verstellen. Sie sollten in Ihren eigenen Projekten aber unbedingt eine Fehlerbehandlung implementieren.

3.2 Beispiel-Client

Für die jeweiligen Betriebssysteme wird der Beispiel-Client wie folgt in einer Kommandozeile aufgerufen:

- Microsoft Windows: `.\clientWin.exe <IP> <Port> <Nachricht>`
- Mac Os X: `./clientOsx <IP> <Port> <Nachricht>`
- Linux: `./clientLinux <IP> <Port> <Nachricht>`

Damit sich der Client mit dem Server verbinden kann, benötigt der Client die IP-Adresse des Servers und außerdem den Port, auf dem der Server auf Anfragen reagiert. Diese Angaben sind im obigen Aufruf mit `<IP>` und `<Port>` bezeichnet. Die Nachricht, die der Client an den Server schickt, ist mit `<Nachricht>` gekennzeichnet und darf in diesem einfachen Beispiel keine Leerzeichen enthalten, es sei denn, die gesamte Nachricht wird mit doppelten Hochkommata eingeschlossen.

3.2.1 client.cpp

Der Konstruktor der Klasse `UniSocket` benötigt als Parameter die IP-Adresse des Servers sowie den Port, auf dem der Server horchen soll. Außerdem müssen wir eine End-Markierung angeben, die das Ende einer Nachricht kennzeichnet. In der Zeile 13 erzeugt der Client mit diesen Parametern ein `UniSocket`-Objekt `s`, das die Verbindung zum Server repräsentiert. Der `try-catch`-Bereich soll Exceptions beim Verbindungsaufbau abfangen.

```
1  #include <iostream>
2  #include <cstdlib>                // for function atoi()
3  #include "UniSocket.hpp"
4  using namespace std;
5
```

```
6  int main(int argc, char * argv[]) {
7      if (argc < 4) {
8          cout << "usage: " << argv[0]
9               << " <ip> <port> <message>" << endl;
10         return 1;
11     }
12     try {
13         UniSocket s(argv[1], atoi(argv[2]), "[ENDE]");
14         s.send(argv[3]);
15
16         cout << "server sagt: " << s.recv() << endl;
17         s.close();
18     } catch(UniSocketException e) {
19         cout << e._msg << endl;
20     }
21     return 0;
22 }
```

In Zeile 14 sendet der Client die auf der Kommandozeile eingegebene Zeichenfolge an den Server und wartet dann in Zeile 16 auf die Antwort des Servers und gibt diese Antwort auf der Konsole aus.

4 make-Skripte, Shared-Library und Programmaufruf

Es handelt sich bei den make-Skripten nicht um ein Makefile, welches mit *make* ausgeführt wird, sondern um Shell- bzw. Batch-Skripte, die eine Reihe von Kommandozeilen-Befehlen enthalten. Unter Linux öffnen Sie die Kommandozeile und wechseln in den Ordner, in dem sich `makeLinux.sh` befindet. Führen Sie das Shell-Skript mit

```
./makeLinux.sh
```

aus. Wenn es nicht mit `chmod +x makeLinux.sh` ausführbar gemacht wurde, können Sie das Shell-Skript auch mit

```
sh makeLinux.sh
```

ausführen. Unter Windows sollten Sie die *PowerShell* als Kommandozeile verwenden. Benutzen Sie die Suche des Startmenüs, um das Programm *PowerShell* zu finden und auszuführen. Um schneller in den Ordner zu wechseln, in dem sich das Batch-Skript befindet, gibt es einen Trick, der zum Teil auch unter Linux und Os X klappt:

- Geben Sie in der Kommandozeile `cd` mit einem anschließendem Leerfeld ein.
- Öffnen Sie mit einem Datei-Browser (Datei-Explorer) den Ordner, in den Sie wechseln wollen.
- Ziehen Sie durch das Gedrückt-Halten der linken Maustaste den Pfad aus der Pfadangabe des Datei-Browsers in das Kommandozeilen-Fenster und lassen dann die linke Maustaste los.
- Der Pfad sollte dann als Text hinter `cd` erscheinen. Durch drücken der Eingabe-Taste führen Sie den Befehl aus und Sie gelangen mit der Kommandozeile in den Ordner.

Wenn Sie in den Ordner gewechselt sind, geben Sie `.\makeWin.bat` ein. Dann sollten alle Programme kompiliert werden.

4.1 Linux

Ein Shell-Skript ist unter Linux und Os X eher unüblich, da ein *Makefile* mehr Möglichkeiten bietet. Die sequenzielle Abarbeitung ist jedoch in einem Shell-Skript deutlicher und lässt sich leichter Schritt für Schritt verstehen.

```
1  #!/bin/bash
2  g++ -Wall -fPIC -c UniSocket.cpp
3  g++ -Wall -fPIC -c SocketWrapperLinux.cpp
4  g++ -Wall -o libUniSocket.so -shared SocketWrapperLinux.o UniSocket.o
5
6  g++ -Wall -std=c++11 -c server.cpp
7  g++ -Wall -std=c++11 -o serverLinux server.o -lUniSocket -pthread -L.
8
9  g++ -Wall -c client.cpp
10 g++ -Wall -o clientLinux client.o -lUniSocket -L.
```

Die ersten drei Aufrufe der GNU-Compiler-Suite `g++` erzeugen die Shared-Library *libUniSocket.so*, die für Ihr Projekt die Klassen, die in den Header-Dateien *UniSocket.hpp* und *SocketWrapperLinux.hpp* beschrieben sind, zur Verfügung stellt. Die Option `-c` gibt an, dass die Dateien nur zu kompilieren (compile) sind, ohne eine ausführbare Datei zu erstellen. Mit der Option `-o` gibt man den Namen an, den die übersetzte Datei haben soll. In den Zeilen, in denen `g++` ohne die Option `-c` aufgerufen wird, wird nicht der Compiler sondern der Linker aufgerufen und alle angegebenen Dateien werden gebunden. Diese Vorgehensweise, das Übersetzen und das Binden zu trennen, wird immer dann gewählt, wenn eine Datei in mehreren Programmen gebunden werden soll, aber nur einmal übersetzt werden muss.

Eine Shared-Library wird nur ein einziges Mal in den Hauptspeicher geladen, auch wenn mehrere Programme bzw. Prozesse darauf zugreifen. Damit das funktioniert, müssen die Daten vom Code getrennt werden. Daher muss die Library zur Laufzeit je Programm einen eigenen Platz für Variablen erzeugen können. Um dies zu ermöglichen, muss der Compiler mit dem Parameter `-fPIC` einen passenden Code bauen, der für Variablen Sprungpunkte in einen anderen Speicherbereich (Global Object Table) benutzt. Mehr dazu finden Sie im Kapitel „Position-independent code“ unter dem Link:

`http://www.iecc.com/linker/linker08.html`

Die *Global Object Table* einer Shared-Library ist vergleichbar mit der *Virtual Method Table*, die Sie aus der Veranstaltung OOA (Objektorientierte Anwendungsentwicklung) kennen. In der Virtual Method Table wird für jedes Objekt zur Laufzeit ein Sprungpunkt zur Implementierung der Methode ermöglicht.

Die weiteren Aufrufe der Compiler-Suite `g++` zeigen, wie man die Shared-Library in seinem Projekt nutzt. Da der beispielhafte Server die Threads aus dem 2011er `C++`-Standard verwendet, der in der Regel nicht beim `g++`-Compiler per default eingestellt ist, ist ein entsprechender Parameter gesetzt. Mit `-lxyz` sucht der Linker, der über `g++` aufgerufen wird, nach einer Library mit dem Datei-Namen *libxyz.so* in seinem Suchpfad. Da die *libUniSocket.so* im selben Ordner liegt, wird dieser Suchpfad mit dem Parameter `-L.` auf den aktuellen Ordner erweitert.

Grundsätzlich kann man die Shared-Library im Suchpfad des Linkers ablegen, der sich standardmäßig im Pfad `/usr/lib/` befindet - dann muss man aber als Administrator (Super-User) mit dem Befehl `ldconfig` den Linker-Cache neu aufbauen. Wenn Sie daran inter-

essiert sind, wo `ldconfig` nach neuen Libraries sucht, schauen Sie in die entsprechenden Konfigurations-Dateien (beginnen mit `/etc/ld.so.conf`). Dort sind die Pfade hinterlegt, in denen nach Shared-Libraries gesucht wird. Für die *Executable* muss die Shared-Library ebenfalls auffindbar sein.

4.2 Os X

Das Skript *makeOsx.sh* ist dem unter Linux sehr ähnlich. Auffällig ist nur, dass dem Compiler `g++` weitere Parameter für die *.dylib* Datei übergeben werden müssen. Die Option `-fPIC` ist unter Os X nicht nutzbar, da Os X für Shared-Libraries ein anderes Konzept benutzt. Außerdem muss beim Ausführen des Clients oder Servers unter Os X kein Suchpfad zur Shared-Library *libUniSocket.dylib* angegeben werden.

4.3 Windows

Das folgende Skript erzeugt 32-bit-Code (x86) und nutzt den VisualC++ 2013 Compiler. Die Zeile 2 mit dem `call` sollten Sie anpassen, wenn Sie z.B. keinen 32-bit-, sondern mit x64 ein 64-bit-Programm haben wollen. Den Pfad müssen Sie für Visual Studio 2012 auf `\Microsoft Visual Studio 11.0\` hin anpassen.

```
1  :: Pfad und Maschinentyp anpassen!
2  call "c:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\vcvarsall.bat" x86
3
4  :: Das macht .lib (zum linken) und .dll (fuer die exe)
5  cl -EHsc /LD UniSocket.cpp SocketWrapperWin.cpp ws2_32.lib
6
7  cl -c server.cpp -EHsc
8  link -out:serverWin.exe server.obj ws2_32.lib UniSocket.lib
9
10 cl -c client.cpp -EHsc
11 link -out:clientWin.exe client.obj ws2_32.lib UniSocket.lib
```

Der Call-Befehl ruft ein anderes Batch-Skript auf, welches in der *PowerShell* die Umgebungsvariablen so anpasst, dass der VisualC++ Compiler tadellos arbeitet. Allerdings kann es passieren (z.B. bei der `cmd.exe` statt der *PowerShell*), dass Pfade um einen weiteren Pfad erweitert werden und die Shell dies bei jeden Aufruf wiederholt (und nicht zusammenfasst). Irgendwann ist der Befehl zum Erweitern des Pfades so lang, dass eine Fehlermeldung erscheint. Da die Umgebungsvariablen aber nur innerhalb des Shell-Fensters gültig sind, kann man einfach ein neues Shell-Fenster aufmachen. Alternativ können Sie dieses Batch-Skript gerne um passende `IF`-Anweisungen erweitern, um den `call`-Aufruf nur einmalig auszuführen.

Ein Parameter für den `C++`-Standard aus dem Jahr 2011 ist nicht notwendig, da dieser ab Visual Studio 2012 als *Default* gesetzt ist. Ebenso ist eine *thread.lib* nicht mit aufzuführen, um Threads beim Beispiel Server-Code zu nutzen. Allerdings ist beim Linken unbedingt die Library *ws2_32.lib* anzugeben, die die Funktionen der Windows-Sockets (Version 2) bereit stellt.

Folgende Links können hilfreich sein, wenn Sie mehr über den Microsoft Compiler oder Visual C++ Redistributable wissen wollen:

- `https://msdn.microsoft.com/en-us/library/y0zzbyt4.aspx`
- `https://msdn.microsoft.com/en-us/library/vstudio/y0zzbyt4%28v=vs.110%29.aspx`

- `https://msdn.microsoft.com/en-us/library/vstudio/x4d2c09s%28v=vs.110%29.aspx`
- `http://www.microsoft.com/de-DE/download/details.aspx?id=40784`

5 Anhang

Im weiteren wollen wir noch das UML-Diagramm der von uns erstellten Klassen und einige Verbesserungsmöglichkeiten beschreiben. Die Klasse `SocketWrapper` liegt in drei verschiedenen Versionen vor: einmal pro unterstütztem Betriebssystem. Welcher dieser Klassen verwendet wird, entscheidet die `#ifdef`-Prä-Compiler-Anweisung in der Header-Datei `UniSocket.hpp`.

5.1 UML-Diagramm

Das Klassendiagramm besteht im Grunde aus den Klassen `UniSocket` und `UniServerSocket`, sowie aus den zusätzlichen Klassen `SocketWrapper` und `UniSocketException`. In Ihren eigenen Projekten arbeiten Sie nur mit den Klassen `UniSocket` und `UniServerSocket`. Die betriebssystemabhängige Klasse `SocketWrapper` wird nur im Hintergrund verwendet.

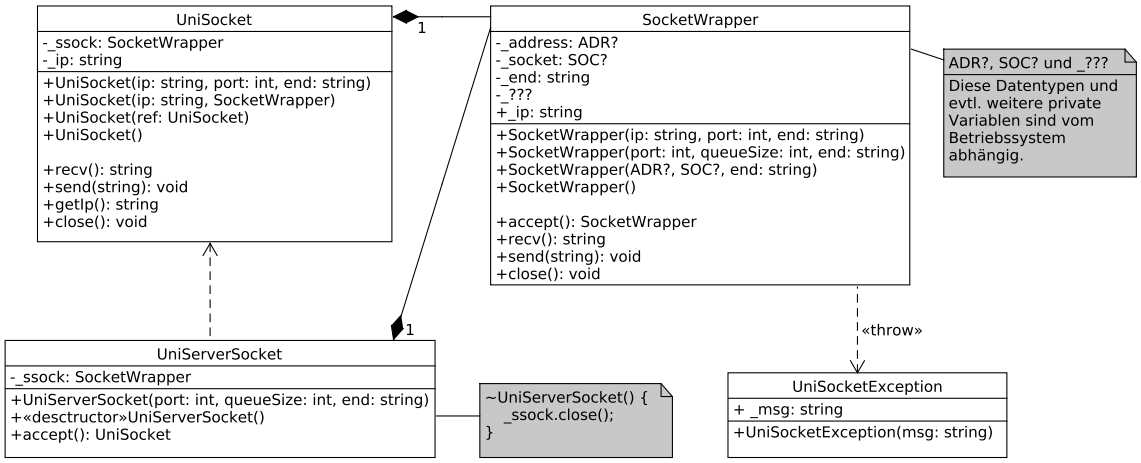

Abbildung 3: UniSocket und seine Partner-Klassen

5.2 Verbesserungen

Es gibt viele Stellen im Code von `UniSocket` und `UniServerSocket`, die Sie noch verbessern oder ausbauen können:

- Es fehlt die Option mit einem Header zu arbeiten, in dem die Anzahl der Bytes steht (statt einer End-Markierung, die potentiell auch Bestandteil der Nutzdaten sein kann).
- Eine Namensauflösung, damit man nicht die IP-Adresse angeben muss, ist nicht enthalten.

- Unter Linux ergab ein Test mit Wireshark, ein Programm zur Analyse von Netzwerk-Kommunikationsverbindungen (Sniffer), dass das *send()* von UniSocket ein Byte mit 0 (char '\0') im String senden kann, aber das *recv()* von UniSocket nur einen String bis zum ersten '\0' übergibt! Die Typ-Casts und benutzten String-Funktionen sollten daher noch einmal untersucht werden, wenn Sie Daten mit einem 0-Byte übertragen wollen.
- Es fehlt eine IPv6 Unterstützung.
- Im Beispiel-Server ist das blockierende Lesen von *recv()* in einen Thread ausgelagert, und man kann daher den Socket erst nach Erhalt einer Nachricht schließen. Da man in verteilten Systemen immer mit Verarbeitungsfehlern eines Verbindungspartners rechnen muss, sind blockierende Aufrufe ohne Timeout unvorteilhaft. Informieren Sie sich, wie Sie bei einem Socket, der über *accept()* erstellt wurde, mit der Option `SO_RCVTIMEO` und der Funktion `setsockopt` einen Timeout festlegen können.
- Da UniSocket Bytes nur mit Strings überträgt, müsste man Strukturen oder Binärcode z.B. im json-Format oder mit base64-Kodierung übertragen.

Die Verwendung eines Headers mit Informationen zur Länge der Nachricht anstelle einer End-Markierung sowie die Option des nicht-blockierenden Lesens sind in einer Weiterentwicklung enthalten, die Sie unter dem folgenden Link abrufen können:

`https://github.com/no-go/UniSocket`