HOCHSCHULE MITTWEIDA
UNIVERSITY OF APPLIED SCIENCES

Wissenschaftliche Arbeit zur Erlangung des akademischen Grades Bachelor of Science.

Generische Serveranwendung unter C++

Bachelorthesis

Hochschule Mittweida – University of Applied Sciences
Fachbereich Mathematik / Naturwissenschaften / Informatik
Studiengang Informatik

Vorgelegt von
Herrn Daniel Reichelt
Unterer Engen 18b
09627 Bobritzsch OT Naundorf
Matrikel-Nr.: 18545

Erstprüfer
Prof. Dr.-Ing. habil. Joachim Geiler

Zweitprüfer
Prof. Dr.-Ing. Mario Geißler

Abgabetermin
01.12.2011

Inhaltsverzeichnis

1. Motivation....4
2. Problemanalyse....5
3. Konzeption....6
   3.1 Plugin System....6
   3.2 Zonenkonzept....7
   3.3 Application Programming Interface....8
   3.4 Konfigurationsdateien....9
4. Arbeitsweise....10
   4.1 Logging-System....10
   4.2 Startvorgang....11
   4.3 Laden und Initialisieren von Plugins....13
   4.4 Netzwerkzugriff....14
5. Die Boost-Bibliotheken....15
   5.1 Verwendete Bibliotheken....15
   5.2 Die ASIO-Bibliothek....18
   5.3 Einschätzung....21
6. Einordnung....21
   6.1 Vergleich Superserver....21
   6.2 Vergleich SmartFox-Server....23
7. Anwendungsbereiche....24
   7.1 Fallbeispiel A....25
   7.2 Fallbeispiel B....27
8. Implementationsdetails....29
   8.1 Logging-System....29
   8.2 Server-Klassen....30
   8.3 Sessions und TransferTasks....33
   8.4 NetStreamBuffer....37
9. Beispiel-Plugins....39
   9.1 httpd....39
   9.2 simplechat....40
10. Verbesserungsmöglichkeiten....41

11. Weiterentwicklung der Software....42
12. Persönliche Weiterentwicklung....43
13. Fazit....44
14. Installationshinweise....45
15. Quellen und Referenzen....46
16. Abbildungsverzeichnis....47
17. Verwendete Hilfsmittel....47
18. Selbstständigkeitserklärung....48
Anhang....49
A. Verfügbare Events....49
B. Klassendiagramm HTTP-Addon....51

1. Motivation

Die Vermeidung von Sourcecodewiederholungen ist ein zentrales Problem in der Programmierung, mit der sich jeder einmal konfrontiert sieht. Speziell im Bereich der Netzwerkprogrammierung, wiederholen sich viele Routinen – von der Erstellung der Sockets bis hin zur Verwaltung der Verbindungen. Programmierer, welche viele kleine Netzwerkprogramme entwickeln, implementieren viele dieser Programmteile immer wieder neu oder verwenden den selben Programmcode in mehreren Anwendungen. Wird in diesen Teilen ein Fehler gefunden oder soll die Funktionalität erweitert werden, stehen sie vor dem Problem, dass sämtliche Programme manuell aktualisiert werden müssen.

Etablierte Techniken, zum Beispiel Remote Procedure Calls (RPC) oder Object Request Broker (ORB) vereinfachen zwar die Interprozesskommunikation, nehmen dem Entwickler aber gleichermaßen die Kontrolle über den eigentlichen Netzwerkverkehr, indem sie ein eigenes Protokoll definieren, über welches der Datenaustausch erfolgt. Eine direkte Kommunikation zu anderen Diensten wird damit erschwert und ist meistens davon abhängig, ob der Hersteller eine passende Schnittstelle anbietet oder Erweiterungen für seine Software zulässt. Ist dies nicht der Fall, ist eine Bibliothek oder eine eigene Implementation erforderlich, die die Mechanismen der Middleware umgeht.

Das Ziel dieser Arbeit soll es sein, eine Middleware zu entwerfen, die diese Nachteile nicht mit sich bringt und dem Programmierer bei netzwerkspezifischen Problemen eine Schnittstelle bietet, über welche er sein Vorhaben umsetzen kann, ohne dass ihm dabei die Kontrolle über den syntaktischen Aufbau des Netzwerkverkehrs entzogen wird. Gleichermaßen soll unter Beachtung dieser Vorgabe versucht werden, die Netzwerkprogrammierung so weit wie möglich zu vereinfachen und zu abstrahieren, so dass ein Programmierer sich auf seine eigentliche Arbeit konzentrieren kann, ohne sich auf verwaltungstechnische Probleme der Netzwerkprogrammierung einlassen zu müssen.

2. Problemanalyse

Die zu entwerfende Software soll unter der Programmiersprache C++90 entwickelt sowie lediglich unter linuxbasierenden Systemen einsetzbar sein und daher auf plattformabhängige Implementierungen verzichten (beispielsweise *Winsock* unter Windowssystemen). Jedoch soll die Möglichkeit bestehen, die Software in einer späteren Version plattformunabhängig zu gestalten. Unter diesem Aspekt sollten plattformabhängige Funktionsrufe gebündelt und in seperaten Klassen genutzt werden, um den Umstellungsaufwand über Precompiler-Statements gering zu halten. Im Falle einer Systemunabhängigkeit würden jedoch Probleme mit genutzten Bibliotheken auftreten. Speziell zu ladende Plugins in Form von dynamischen Bibliotheken wären problematisch, da diese sehr system- und compilerabhängig sind. Selbst das Kompilieren der Anwendung und des Plugins mit dem gleichen Compiler, aber unterschiedlichen Versionen, kann zu Kompatibiliätsproblemen führen, wenn sich die Symboltabellen unterscheiden.

Die Plugins selbst müssten eine Factory zur Verfügung stellen, mit der benötigte Objekte bzw. Instanzen erstellt werden können. Durch korrekt entworfene abstrakte Klassen können explizite Casts größtenteils vermieden werden, was auch die Fehleranfälligkeit verringert. Die Verwaltung der Plugins soll über eine einzelne Konfigurationsdatei erfolgen. Über eine geeignete Struktur soll jedes Plugin einen seperaten Bereich in dieser Datei zugewiesen bekommen, wobei die Parameterbezeichnung frei wählbar sowie dem Plugin bekannt sein muss. Dadurch wird vermieden, dass jedes Plugin ein eigenes System zur Einstellung seiner Optionen etabliert und das ganze System an zentraler Stelle vom Benutzer verwaltet werden kann.

Der Netzwerkzugriff der Plugins soll so weit wie möglich vereinfacht werden, ohne dass der Programmierer eingeschränkt wird. Es soll theoretisch möglich sein, jeden Standard, der auf TCP beruht, mit diesem System zu implementieren. Ein systemeigener Netzwerk-Layer zur Vereinfachung der Netzwerkprogrammierung kann demnach nicht zum Einsatz kommen.

3. Konzeption

3.1 Plugin System

Das Plugin-System ist eines der wichtigsten Teile der Server-Software, da erst mit diesem ein Verhalten spezifiziert wird. Die Plugins liegen als dynamische Bibliotheken vor und müssen von der Server-Software geladen werden. Dabei ist es erforderlich, einen Einstiegspunkt festzulegen, über den das Plugin initialisiert werden kann. Dies soll über eine C-Funktion realisiert werden, welche einen Zeiger auf ein Objekt einer Fabrikklasse zurückgibt, mit deren Hilfe sämtliche benötigte Objekte erstellt werden können.

Da die Struktur dieser Objekte bereits vorher bekannt sein muss, soll eine Schnittstelle entwickelt werden, welche gemeinsam, sowohl von der Server-Software als auch von den Plugins, verwendet wird. Die Plugins müssen dabei bestimmte abstrakte Klassen ableiten, mit denen sie den Programmablauf steuern können und damit ein Verhalten festlegen können. Dabei sollte ein weiterer Einstiegspunkt zum Einsatz kommen, an welchem die Plugins sich selbst initialisieren können. Denkbar wäre dabei das Erstellen von Objekten, welche das Plugin während seiner gesamten Laufzeit benötigt, oder das Nachladen weiterer Bibliotheken.

Die primäre Kommunkationsschnittstelle sollen Signale darstellen, welche bei wichtigen Ereignissen, wie zum Beispiel dem Annehmen einer Verbindung, ausgelöst werden. Das Plugin soll für jedes Signal eine beliebige Anzahl an Listenern registrieren können, und erhält so die Möglichkeit, flexibel auf spezifische Ereignisse zu reagieren. Wichtig ist, dass die betroffenen Objekte, die im direkten Zusammenhang mit dem Signal stehen, der Listenerfunktion beziehungsweise der entsprechenden Methode übergeben werden, damit das Plugin mit diesen arbeiten und entsprechende Änderungen vornehmen oder Aktionen durchführen kann. Das Funktionsprinzip und die Eigenschaften der Boost.Signals2 Bibliothek entsprechen diesen Anforderungen genau. Durch das Slot-System können einem Signal beliebig viele Callbacks zugewiesen werden, die prioritätsgesteuert und mit Parametern aufgerufen werden können. Da auch Rückgabewerte von Callback-Funktionen manuell ausgewertet werden können, ließe sich eine Vielzahl von Möglichkeiten der Interaktion integrieren. Recherchen[2] haben allerdings ergeben, dass die Boost.Signals2 Bibliothek eine deutlich schlechtere Performance aufweist als andere

Implementationen, wie zum Beispiel LibSigC++. Da man jedoch davon ausgehen kann, dass die Slotanzahl gering und die Zahl der Aufrufe höchstens moderat ausfallen dürfte, sollte sich der Performanceverlust in Grenzen halten. Da auch die Verwendung anderer Boost-Bibliotheken geplant ist, erscheint es sinnvoll, die Abhängigkeiten einheitlich zu halten anstatt auf mehrere unterschiedliche Produkte bzw. Bibliotheken zu setzen.

3.2 Zonenkonzept

Um die Plugins voneinander separieren zu können und eine logische Gliederung dieser innerhalb der Server-Software zu ermöglichen, sollen Zonen zum Einsatz kommen. Zonen können als Teilmenge der Software gesehen werden. Mit ihnen werden Plugins innerhalb der Hauptanwendung voneinander getrennt und sie stellen gleichzeitig den Handlungsspielraum der Plugins dar. Ein Plugin ist für die Verwaltung von genau einer Zone verantwortlich, in der es frei agieren kann. Es handelt in der Regel unabhängig von anderen Plugins und weiß nichts von der Existenz derer. Dieses Prinzip, welches im Allgemeinen unter dem Begriff Sandbox bekannt ist, soll jedoch nicht in vollem Maße umgesetzt werden. Es sollte die kategorische Möglichkeit bestehen, dass Plugins untereinander kommunizieren können, da nicht auszuschließen ist, dass Abhängigkeiten zwischen ihnen bestehen. So bestünde die Option der Interaktion zwischen den Plugins. Da die sich ergebenden Möglichkeiten wohl größer sind, als das Gefahrenpotential eines Missbrauchs, soll die Sandbox „lose“ umgesetzt werden. Im Normalfall werden die Plugins einander nicht bemerken, aber falls es vom Entwickler eines Plugins gewünscht ist, soll er die Option haben, dieses auch umzusetzen.

Des Weiteren sollen Räume zum Einsatz kommen, in welchen Verbindungen mit gleichen oder ähnlichen Eigenschaften gruppiert werden können. Räume, vergleichbar mit den Chaträumen des Internet Relay Chats, sind somit ein fakultatives Werkzeug zur logischen Trennung von Verbindungen, die ein Entwickler einsetzen kann aber nicht muss. Es soll zudem möglich sein, Räume in einer Baumstruktur anzuordnen, wobei der Wurzelknoten eine Zone sein müsste. Dies bedeutet, dass Räume einen oder keinen Elternknoten und keinen oder beliebig viele Kindknoten besitzen können.

Abstrakt betrachtet, stellen Zonen und Räume Mengen von Verbindungen dar. Eine Zone ist die Gesamtmenge aller Verbindungen auf einen Port, wobei die Räume als Teilmengen zu sehen sind.

Über die reine Funktion als Container hinaus, sollen Zonen und Räume Methoden zur Verwaltung der Verbindungen besitzen. Es muss möglich, sein Verbindungen einzugliedern, zu erfragen oder zu entfernen. Gleiches gilt für die angesprochene Baumstruktur, nur dass in in diesem Falle Räume statt Verbindungen verwaltet werden müssen.

Ferner werden Methoden nötig, über die man Nachrichten an alle Teilnehmer eines Raumes oder gar alle Verbindungen des Servers schicken kann. Auch das Versenden von Nachrichten an einzelne Verbindungen soll möglich sein.

3.3 Application Programming Interface

Das Application Programming Interface der Server-Software soll alle Headerdateien von Klassen enthalten, die nötig sind, um ein Plugin zu erstellen oder um mit der Hauptanwendung zu kommunizieren. Damit aber der Plugin-Entwickler nicht Zugriff auf die gesamte Anwendung erhält, sollten abstrakte Klassen mit rein virtuellen Methoden zum Einsatz kommen, welche von den Implementationsklassen erweitert werden. Somit ließe sich präzise festlegen, auf welche Methoden ein Programmierer Zugriff hat und könnte dadurch den unerlaubten Eingriff in das System zumindest etwas erschweren. Es ist jedoch darauf zu achten, dass innerhalb der API nur Referenzen auf API-Klassen gesetzt werden, da es sonst unmöglich wäre, ein Plugin ohne die originalen Header-Dateien zu kompilieren.

Die API soll somit eine Sammlung von Headerdateien sein, über die sich die generische Serveranwendung in ihrer Gesamtheit, als auch ihrer Teilprozeduren, steuern lässt und dabei die eigentliche Implementation für den Programmierer ausblendet.

Ein negativer Aspekt dieser Variante ist jedoch, dass sämtliche Instanzen einer abgeleiteten Klasse eine Virtual-Method-Table („*vtable*“) mit sich herumtragen müssen, was sowohl Einfluss auf den Speicherverbrauch als auch die Performance hat.

3.4 Konfigurationsdateien

Konfigurationsdateien sollen das von Boost entwickelte INFO-Dateiformat benutzen, welches speziell für den, in den Boost-Bibliotheken enthaltenen, PropertyTree konzipiert wurde. Die Syntax besticht durch ihre Einfachheit, welche dennoch ein hohes Maß an Flexibilität gewährleistet und eine Gliederung der Schlüssel und ihrer Werte zulässt. Durch dieses Format ist es möglich, sämtliche Einstellungen des Servers in nur einer Datei vorzunehmen, ohne dass die Übersichtlichkeit darunter leidet.

```
; A comment
key1 value1   ; Another comment
key2 "value with special characters in it {};#\n\t\"\0"
{
   subkey "value split "\
          "over three"\
          "lines"
   {
      a_key_without_value ""
      "a key with special characters in it {};#\n\t\"\0" ""
      "" value    ; Empty key with a value
      "" ""       ; Empty key with empty value!
   }
}
#include "file.info"    ; included file
```

Beispiel einer INFO-Datei [3]

Über die Konfigurationsdatei soll es möglich sein, beliebig viele Zonen zu erstellen und ihnen die, zum Starten benötigten, Parameter zuzuweisen. Unabdingbar ist die Nennung des Plugins, respektive den Ort der dynamischen Bibliothek, welches geladen werden soll. Zudem soll die Wahl der Portnummer flexibel gehalten werden und somit auch über die Konfigurationsdatei erfolgen. Ferner soll es dem Benutzer möglich sein, auch Räume über diese zu Erstellen.

Nicht festgelegte Namen von Unterschlüsseln, welche Zonen und Räumen zugewiesen werden können, sollen als pluginspezifische Einstellungen angesehen werden. Diese sollen von den Plugins erfragt werden können, um ihre eigenen Optionen durch den Benutzer festlegen zu können. Dabei liegt es in der Verantwortung des Plugin-Autors, diese zu prüfen und gegebenenfalls Standardwerte festzulegen oder den Benutzer des Servers auf sein Versäumnis hinzuweisen.

4. Arbeitsweise

4.1 Logging-System

Das Logging-System ist ein wichtiger Bestandteil einer jeden Software, um dem Benutzer über Fehler oder bestimmte Ereignisse in Kenntnis zu setzen. Das Logging-System des Tulpa-Servers ist ein Hybrid zwischen dem Proxy und dem Verteiler-Entwurfsmuster. Sämtliche Ausgaben werden an einen Proxy übergeben, welcher diese an entsprechende „Geräte“ (Devices) weitergibt. Das Device-Interface der API bildet den Grundstein dieses Systems. Es forciert die Implementation von drei Methoden, welche das Aktivieren, das Deaktivieren und das Annehmen von Log-Nachrichten umfassen.

Dabei ist nicht festgelegt, wie diese Nachrichten behandelt werden sollen. Es wäre denkbar, sie zu sammeln und täglich per E-Mail an den Systemadministrator zu verschicken, oder sie einfach auf der Konsole auszugeben. Letztere Variante wird duch die Klasse *ConsoleLogger* realisiert, welche derzeit das einzige Ausgabegerät des Servers ist und die Nachrichten direkt in die Konsole schreibt.

Der Tulpa-Server benutzt einen zentralen Proxy, der durch die Hauptklasse *TulpaServer* zu erfragen ist. In der Header-Datei *LevelBased.h* werden mittels Precompiler-Statements verschiedene Log-Level deklariert, mit denen man die Art der Nachricht beziehungsweise deren Wichtung festlegen kann. Zusätzlich wird ein einheitliches Format beibehalten, mit welchem ein Zeitstempel und das gewählte Level vor der eigentlichen Nachricht ausgegeben und überprüft wird, ob die Nachricht, ausgehend von ihrer Wichtung, gefiltert werden muss.

Die Proxyklasse benutzt den linksseitigen Bitshift-Operator des *ostringstreams*, um die Nachrichten entgegenzunehmen. Dies hat den Vorteil, dass Log-Nachrichten nicht über einen Formatierungs-String angegeben werden müssen und Umwandlungen von standardisierten Datentypen automatisch durch die in C++ eingebauten Streamklassen vorgenommen werden. Auf der Gegenseite steht die erschwerte Formatierung von numerischen Daten, wenn zum Beispiel die Zahl der Nachkommastellen festgelegt werden soll. Hierfür bietet die formatierte Ausgabefunktion der C-Bibliothek eine elegantere Lösung, welcher jedoch die zuvor genannten Nachteile entgegenstehen.

4.2 Startvorgang

Der Start des Servers verläuft klassisch. Zuerst werden interne Module, beispielsweise das Logging-System, initialisiert und überprüft, ob dieser Vorgang erfolgreich war. Ist dies der Fall, wird die Konfigurationsdatei gelesen und ausgewertet. Zonenknoten sind dabei von besonderer Bedeutung. Wird ein solcher Knoten gefunden, werden die benötigten Einstellungen geladen (Speicherort des Plugins, Port und Protokoll – wobei Letzteres keine Beachtung findet, da der Server auf TCP beschränkt ist) und anschließend nach eventuellen voreingestellten Räumen gesucht. Die Objektinstanzen werden direkt beim Fund des entsprechenden Konfigurationseintrages erstellt. Abhängig von deren Art, werden bei Unterschlüsseln Flags in einer temporären Variable gesetzt, um abschließend Prüfen zu können, ob obligatorische Einstellungswerte gesetzt wurden. Falls entsprechende, notwendige Einstellungen fehlen, wird der Start mit einer hinweisenden Fehlermeldung abgebrochen. Die dabei entstehende Struktur, bestehend aus den Zonen, den Räumen und Unterräumen, wird iterativ und „On-The-Fly“ erstellt – abhängig von der Reihenfolge der vom INFO-Parser gelieferten Werte.

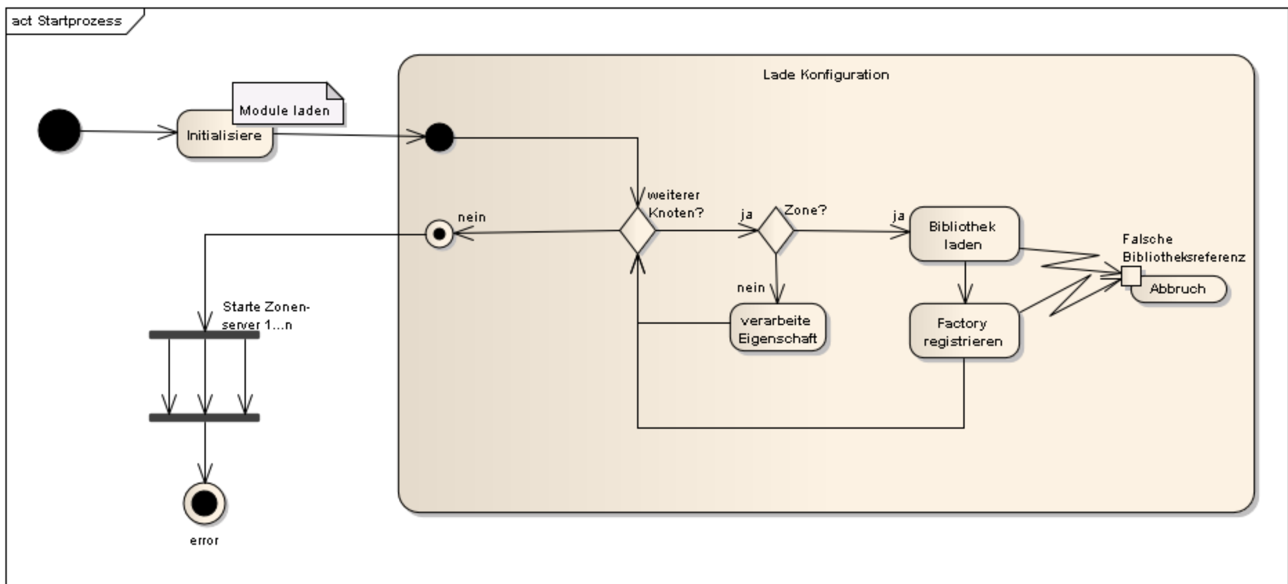

Abb. 1: Schematische Darstellung des Startprozesses

Traten während der Auswertung der Konfigurationsdatei keinerlei Fehler auf, wird für jede angegebene Zone das angegebene Plugin geladen und ein Server samt eigenem Thread erstellt und gestartet. Dabei kann es wiederum zu Fehlern kommen, wenn zum Beispiel der eingestellte Port bereits belegt ist. Unter Linux basierenden System existiert zudem die Einschränkung, dass das Binden von Ports im Bereich von 0 – 1023 nur von einem administrativen Benutzerkonto aus geschehen kann. Dementsprechend würde die, an ein Plugin zugewiesene, Portnummer 21 beim Start des Servers von einem normalen Benutzeraccount aus einen Fehler verursachen.

Nach dem Starten der Subserver und dem erfolgreichen Binden des Ports, beendet der Hauptprozess seine Arbeit. Sämtliche Zonen laufen in separaten Threads, auf deren Beendigung gewartet wird (dem „Joinen“ der Threads). Innerhalb der Threads selbst werden Verbindungen in einer Endlosschleife akzeptiert, und ein etwaiges Beenden dieses Vorgangs muss vom Plugin aus geschehen.

4.3 Laden und Initialisieren von Plugins

Im Anschluss an das Laden der Konfigurationsdatei, werden die angegebenen Plugins, respektive dynamische Bibliotheken, geladen. Dazu wird ein Objekt der Klasse *DynamicLibrary* instanziiert, welches über die Methode *load* versucht, die angegebene Bibliothek zu laden, oder im Fehlerfall eine Exception wirft. Anschließend wird über die Methode *get_symbol* nach dem Einstiegspunkt des Plugins gesucht. Dabei handelt es sich um eine C-Funktion, welche den Namen *get_factory* trägt und einen Zeiger auf eine von dem Interface *Factory* abgeleite Klasse zurückgeben muss, mit welcher benötigte, pluginspezifische Objekte erstellt werden können.

Die abgeleitete Klasse ist angehalten, zwei Methoden zu implementieren, wovon die Erste ein Addon-Objekt zurückgeben muss. Vom Addon-Interface abgeleitete Klassen können als die Hauptklassen von Plugins betrachtet werden. Von besonderer Bedeutung ist die Methode *on_enable*, die einmalig nach dem erfolgreichen Laden des Plugins aufgerufen wird. Damit soll sich das Plugin auf seinen Einsatz vorbereiten können, indem es Listener registriert, globale Objekte initiiert oder Dateien und dynamische Bibliotheken nachlädt. Ersteres ist besonders hervorzuheben, damit das Plugin auf Events (siehe Anhang A) reagieren kann, wenn beispielsweise eine neue Verbindung akzeptiert wird.

Des Weiteren muss die Methode *new_packet_serializer* von der Factory implementiert werden. Der *PacketSerializer* dient dem Zweck, serialisierte Objekte aus dem Netzwerkverkehr herauszulesen und in ihre ursprüngliche Form zu übersetzen, sodass mit den Daten gearbeitet werden kann. Innerhalb des TulpaServers werden diese serialisierten Objekte „Packets“ beziehungsweise „Pakete“ genannt.

4.4 Netzwerkzugriff

Das Lesen von Netzwerkdatenströmen geschieht über einen *PacketSerializer*. Ein solcher muss die Methode *recv_packet* implementieren, welche einen Zeiger auf ein Paket zurückgeben muss. Das Aussehen, genauer gesagt der Aufbau, muss vom Plugin selbst definiert und seinen Ansprüchen entsprechend gestaltet werden.

Ein Paket ist ein eher abstrakt zu betrachtendes Objekt. Der Aufbau kann je nach Anwendungsfall variieren und so kann es passieren, dass ein Plugin weit über 100 Pakete definiert, die oberflächlich nichts miteinander gemeinsam haben, aber dennoch einen spezifischen Zweck erfüllen. Ein Paket definiert demzufolge einen entsprechenden Teil des Netzwerkstromes – einen deserialisierbaren Abschnitt.

Da es zahllose Varianten gibt, wie ein solches Paket aufgebaut sein kann, muss der Entwickler den Deserialisierungsprozess selbst implementieren. Dazu wurde die Hilfsklasse *NetStreamBuffer* entworfen, über die sich der Netzwerkverkehr sequenziell verarbeiten lässt, indem sie Methoden zum Lesen und Schreiben von primitiven Datentypen anbietet und den Netzwerkverkehr auf diese Weise abstrahiert. Anstatt eine Zeichenkette auswerten zu müssen, kann der Entwickler einfach angeben, welchen Datentyp er zu diesem Zeitpunkt erwartet und ihn über Streamoperatoren in eine Variable schreiben lassen. Das Senden geschieht auf analoge Art.

Da der *PacketSerializer* ein fertiges Paket zurückgeben muss, kann auswertet werden, wann das Lesen eines logisch abgetrennten Teils des Netzwerkstroms abgeschlossen wurde. Ist dies geschehen, wird ein Signal ausgelöst, auf das der Entwickler in seinem Plugin reagieren kann. In der Regel wird dies die Auswertung des Paketes und die entsprechende Antwort an den Client sein. Dabei steht es dem Programmierer jedoch frei, wie er dies umsetzt, denn es gibt diesbezüglich keine Restriktionen oder Vorgaben.

5. Die Boost-Bibliotheken

5.1 Verwendete Bibliotheken

Bei den Vorarbeiten zu dieser Arbeit stellte sich die Frage nach der Netzwerkimplementation. Durch Recherchen ergab sich, dass die Boost ASIO Bibliothek (Asynchronious Input/Output) die Netzwerkroutinen erheblich vereinfachen kann. Das Konzept der Bibliothek schien logisch und effizient zu sein. Nachdem der Einsatz dieser beschlossen war, wurden mit der Zeit mehrere Boost-eigene Bibliotheken verwendet.

Die Boost-Signals2 Library offeriert ein threadsicheres Signalsystem, welches mit Slots beziehungsweise Callbacks arbeitet. Das Signal-Slot-Konzept ist eine Variante des Beobachter-Verhaltensmusters. Ein Signal wird dabei mit einer beliebigen Anzahl von Slots, oder auch Callbacks beziehungsweise Rückruffunktionen, verknüpft. Wird das Signal anschließend ausgelöst, werden alle Beobachter benachrichtigt, die ihrerseits entsprechend darauf reagieren können. Konkret bedeutet dies, dass ein bestimmtes Ereignis ausgelöst wird, welches sämtliche verknüpfte Funktionen aufruft. Dabei können Parameter an diese übergeben, oder auch die Rückgabewerte ausgewertet werden. Callback-Funktionen müssen den exakt gleichen Aufbau haben, wie es bei der Definition des Slots angegeben wurde. Andernfalls kommt es beim Kompilieren zu Fehlern.

Ein einfach gehaltenes Beispiel aus der Dokumentation verdeutlicht die Funktionsweise.

```
// [...]

// Deklaration der Callback-Funktion innerhalb einer Struktur
struct HelloWorld
{
	// Aufzurufende Funktion. Standardmäßig wird der
	// Klammeroperator verwendet; normale Funktionen
	// oder Methoden sind ebenfalls möglich.
	void operator()() const
	{
		// Beliebige Aktion
		std::cout << "Hello, World!" << std::endl;
	}
```

```
};

int main()
{
      // Deklaration des Signals und Aufbaus der Callback-
      // Funktionen. Im Folgenden eine parameterlose Funktion
      // ohne Rückgabewert.
      boost::signals2::signal<void ()> sig;

      // Instanziierung eines HelloWorld-Objektes und
      // Anbinden an das Signal "sig".
      HelloWorld hello;
      sig.connect(hello);

      // Auslösen des Signals bewirkt eine Hello-World Ausgabe
      sig();

      // Beenden des Programms
      return 0;
}
```

Beispiel der Signals2 Dokumentation [4] (Kommentare geändert)

In diesem Beispiel wird eine Struktur angelegt, in welcher ein Klammeroperator definiert wird. Diese werden standardmäßig beim Auslösen von Ereignissen aufgerufen, wenn es sich bei dem Parameter zum Verbinden eines Callbacks um ein derartiges Objekt handelt. In den meisten Fällen würde man wahrscheinlich eine bestimmte Funktion oder eine Klassenmethode anstatt eines gesamtes Objekt übergeben, um ein unerwartetes Verhalten zu verhindern. Es folgt das Anlegen eines Signales, bei der die Struktur der Callback-Funktion angegeben werden muss. In diesem Fall handelt es sich um eine parameterlose Funktion ohne Rückgabewert. Mit dem Auslösen des Signals, welches über den Klammeroperator des Signales geschieht, werden sämtliche Callbacks aufgerufen. Entsprechend dieses Beispieles, wird die Zeichenkette „Hello, World!“ auf die Konsole geschrieben, bevor sich das Programm beendet. Hätte man das Objekt *HelloWorld* zweimal mit dem Signal verknüpft, würde die Zeichenkette ebenfalls zweimalig auf dem Bildschirm ausgegeben.

Dieses Beispiel zeigt eine sehr einfache Verwendungsweise der Bibliothek. Die erweiterte Verwendung dieser bietet eine Vielzahl von Möglichkeiten, welche von Priorisierung über die Auswertung der Rückgabewerte der Callbacks reichen.

Performancetechnisch ist die Boost-Signals2 Bibliothek eher dem Mittelfeld zuzuordnen. Andere Implementationen erzielen bei annähernd gleichen Rahmenbedingungen deutlich bessere Ergebnisse. Trotz dieses Mankos, sollte ein einheitliches System im Vordergrund stehen und somit die Signals2 Bibliothek eingesetzt werden, was bei den externen Abhängigkeiten positiv gewertet werden kann.

Eine weitere verwendete Boost-Bilbiothek ist die Thread-Library. Eine wichtige Eigenschaft der Boost-Bibliotheken ist, dass sie plattformunabhängig sind. Zwar soll der Server vorerst nur unter Linux lauffähig sein, jedoch soll die Software zu einem späteren Zeitpunkt auch unter anderen Betriebssystemen funktionieren. Darauf bedacht, ist darauf zu achten, von Beginn an möglichst wenige Abhängigkeiten an ein bestimmtes Betriebsystem zu schaffen. Threads sind ein Aspekt, bei dem es betriebssystemspezifische Unterschiede gibt und eine Crossplattform-Lösung den nachträglichen Arbeitsaufwand verringern kann.

Die Thread Bibliothek führt eine Reihe von Techniken zur Bewältigung von Synchronisationsproblemen zwischen Threads ein. Diverse Lock-Konzepte können je nach Einsatzgebiet einen Geschwindigkeitszuwachs bedeuten, wenn sie richtig eingesetzt werden. So ist zum Beispiel eine Unterscheidung zwischen lesendem und schreibendem Zugriff möglich, wobei beliebig viele Threads von einer Ressource lesen können, ohne das ein exklusiver Mutex vorausgesetzt wird. Dieser wird lediglich beim Schreiben auf den Speicher notwendig. Durch Upgrade-Mechanismen kann dieser problemlos aufgewertet werden. Der Thread, der eine Aufwertung „beauftragt“, wird dabei solange eingefroren, bis kein anderer Thread mehr lesend auf die entsprechende Ressource zugreift. Anschließend erhält der wartende Thread exklusiven Zugriff auf die Ressource und kann die Daten ändern, ohne dass Synchronisationskonflikte auftreten. Diese automatisierten Vorgänge erleichtern die Programmierung von Programmen, die viele Threads einsetzen und helfen, Fehler zu vermeiden.

Das sogenannte Scoped-Lock Konzept vereinfacht die Programmierung kritischer Abschnitte noch weiter, indem es exklusive Zugriffe über Blöcke (dem Bereich zwischen zwei geschweiften Klammern) regelt. Dabei wird ein entsprechender Lock innerhalb eines Blockes erstellt und ein exklusiver Zugriff erlangt. Wird der Blockabschnitt nun verlassen, wird das Objekt zerstört und sein Destruktor aufgerufen, durch welchen der Mutex freigegeben wird. Somit ist beim Programmieren lediglich ein Scoped-Lock zu erstellen und darauf zu achten, an welchem Punkt dieses zerstört wird.

Die Boost-Thread-Bibliothek beinhaltet zudem noch viele weitere Funktionen, um Threads entsprechend ihrem Aufgabenspektrum zu verwalten. So ist es möglich Threads zu gruppieren, synchronisierte Bedingungsvariablen anzulegen, Threads außerplanmäßig, ohne das Entstehen von Speicherlecks, zu beenden oder threadspezifische Objekte zu erzeugen. Letztere stellen sicher, dass eine Objektinstanz genau einmal pro Thread erstellt wird, ohne dass eine weiterführende Überprüfung notwendig wird.

Die letzte verwendete Boost-Biliothek ist die System-Library. Diese stellt hauptsächlich plattform- und boostspezifische Exceptions zur Verfügung. Das Einbinden dieser Bibliothek ist notwendig, um entsprechend auf diese Ausnahmen zu reagieren.

5.2 Die ASIO-Bibliothek

Wie bereits erwähnt, wurde der Server ursprünglich und von Grund auf mit der Boost ASIO-Bibliothek entwickelt. Diese Bibliothek stellt eine einheitliche Schnittstelle zur Arbeit mit synchronen sowie asynchronen Geräten zur Verfügung, wozu auch Netzwerkgeräte gehören.

Der Aufbau der Schnittstelle wirkt durchdacht und die gezeigten Beispielanwendungen[5] erstaunen durch die Tatsache, dass sie in aller Kürze zum Teil komplexe Probleme lösen. Der Ansatz, Funktionsrufe asynchron abzuhandeln ist insofern interessant, als dass man auf eine eigene Threadverwaltung verzichten kann, da dies von dem unterliegendem System übernommen wird. Diese Technik ist zwar nicht revolutionär, da sie auch in anderen Programmiersprachen standardmäßig Anwendung findet (zum Beispiel bei ActionScript), weiß aber durchaus durch die gute Umsetzung und den damit verkürzten Programmieraufwand zu gefallen.

Asynchrone Funktionen oder Methoden beziehen sich dabei auf Sockets und die damit verbundenen Lese- und Schreibaufgaben. Über die Funktionsparameter gibt man dabei an, wie viele Daten in welchen Speicherbereich geschrieben werden sollen, welche Rückruffunktion aufgerufen werden soll und welche Parameter an diese zu übergeben sind. Letztere sind in der Regel die Anzahl der übertragenen / gelesenen Bytes und ein Fehlerobjekt, welches es zu überprüfen gilt. Zahlreiche weitere Einstellungen ermöglichen es dabei, die Netzwerkvorgänge den eigenen Anforderungen anzupassen.

```
socket_.async_read_some(boost::asio::buffer(data_, max_length),
      boost::bind(&session::handle_read, this,
            boost::asio::placeholders::error,
            boost::asio::placeholders::bytes_transferred));
```

Beispielaufruf von async_read_some

Dieses Beispiel zeigt einen Lesezugriff auf ein Socket. Die Methode *async_read_some* erfragt die lesebereiten Daten des Netzwerkstroms, wobei die Datenmenge *max_length* nicht überschritten werden darf. Als Puffer wird ein von ASIO eingeführtes Pufferobjekt verwendet, welches ein char-Array *data_* von der Größe *max_length* als internen Speicher benutzt. Der zweite Parameter beschreibt die Rückruffunktion. Also jene, die aufgerufen wird, wenn der Lesevorgang abgeschlossen oder unterbrochen wurde. Hierzu wird die Methode *handle_read* der Klasse S*ession* an ein davon instanziiertes Objekt *this* gebunden. Die letzten Platzhalter beschreiben die Parameter der Callback-Funktion, wobei es sich hierbei um ein Fehlerobjekt und einen numerischen Wert der gelesenen Bytes handelt. Durch den Aufruf dieser Funktion, wird ein von der ASIO-Bibliothek verwalteter Thread angewiesen, Daten in der vorgegebenen Weise aus dem Netzwerkverkehr herauszulesen. Hat dieser seine Arbeit beendet (Puffer voll, Fehlerfall, derzeit keine weiteren Daten lesbar), wird die Callback-Funktion aufgerufen, in der das weitere Vorgehen spezifiziert sein muss. Diese asynchronen Funktionen ermöglichen es, auf eigene separate Threads zu verzichten und ein Programm so zu entwickeln, als würde es keine blockierenden Funktionen aufrufen.

Die ASIO-Bibliothek unterstützt zahlreiche Protokolle wie zum Beispiel UDP, ICMP und TCP sowie SSL über TCP. Im Rahmen der Vorbereitung zu dieser Arbeit, war dies ein wichtiger Faktor für eine Entscheidung zur Verwendung der Bibliothek und sollte die Implementation weiterer Protokolle vereinfachen. Speziell die Möglichkeit SSL schnell und unkompliziert implementieren zu können, überzeugte. Geplant war, die Server-Software vorerst nur mit TCP-Funktionalitäten auszurüsten und die betreffenden Klassen später in Template-Klassen umzuwandeln, welche als Template-Parameter das zu verwendende Protokoll, respektive Socket, annehmen sollten.

```
tulpa::networking::SessionImpl<boost::asio::ip::tcp::socket> *tcp_session = new
tulpa::networking::SessionImpl<boost::asio::ip::tcp::socket>;

tulpa::networking::SessionImpl<boost::asio::ip::udp::socket> *udp_session = new
tulpa::networking::SessionImpl<boost::asio::ip::udp::socket>;
```

Schematische Darstellung der ursprünglich geplanten Syntax

Da sämtliche Socket-Klassen eine gemeinsame Basis besitzen, war davon auszugehen, dass dieses Vorhaben problemlos möglich sei. Beim Versuch, dieses Konzept zu realisieren, zeigte sich jedoch, dass sämtliche Netzwerkfunktionen von ASIO ein bestimmtes Protokoll voraussetzen. Weitere Recherchen ergaben, dass die Netzwerkroutinen innerhalb der Bibliothek ebenfalls separat implementiert wurden. Der Versuch dieser Problematik mit dynamischen Casts entgegenzuwirken, beantwortete der Compiler jedoch mit zahlreichen Fehlermeldungen, welche sich nicht beseitigen ließen. Erneute Nachforschungen zu diesem Thema brachten zum Vorschein, dass bereits andere Entwickler versucht haben, dies umzusetzen und zu dem Schluss gekommen sind, dass es so nicht möglich und eine Verwendung der protokolleigenen Funktionen obligatorisch ist. Da auch die Boost-ASIO-Dokumentation im Kapitel „A combined TCP/UDP asynchronous server“ [6] keine wirkliche Lösung aufzeigt wie dies zu erreichen ist und stattdessen jeweils Server- und Sessionroutinen für jedes Protokoll einzeln implementiert, muss davon ausgegangen werden, dass es sich dabei um den einzigen, sofern man ihn so nennen kann, Lösungsansatz handelt. Da man, von der Konzeption ausgehend, jeweils zwei Klassen für die drei Protokolle entwickeln müsste, welche dazu noch fast komplett identisch wären, wurde beschlossen auf die Boost-ASIO Bibliothek zu verzichten und stattdessen die betriebsystemeigenen Netzwerkroutinen einzusetzen.

5.3 Einschätzung

Die Boost-Bibliotheken verfolgen im Allgemeinen das Ziel, die Produktivität bei der Entwicklung unter C++ zu steigern und auf allen Plattformen verfügbar zu sein.

Die verwendeten Bibliotheken offenbaren viele Wege, mit Problemen umzugehen, indem sie entweder bekannte Algorithmen implementieren oder gar neue Lösungsansätze einsetzen. Auf Grund des Umfangs und durchdachter Routinen kann es die Entwicklung von Software beschleunigen, indem es dem Programmierer viele Teile seiner Arbeit erleichtert oder Lösungen bereitstellt, die er hätte selbst entwickeln müssen.

Dagegen steht der enorm hohe Einarbeitungsaufwand. Es fällt anfangs schwer, Zusammenhänge und Abhängigkeiten in diesem komplexen System zu erkennen, welches zum Großteil auf Template- und Metaprogrammierung aufbaut. Zudem ist es erforderlich, wie bei jeder Technologie, sich zuerst in diese einzuarbeiten und zu versuchen sich einen Überblick über die gebotenen Funktionen zu verschaffen und abzuschätzen, welche sinnvoll im spezifischen Kontext zu verwenden sind und welche nicht.

6. Einordnung

6.1 Vergleich Superserver

Als Superserver bezeichnet man einen Prozess, welcher auf mehreren Ports lauscht und auf eine Verbindungsanfrage wartet. Wird eine solche angenommen, wird anhand des Ports und der dazugehörigen Konfiguration bestimmt, welches Programm mit diesem assoziiert ist und anschließend gestartet, wobei das Socket an den neuen Prozess übergeben wird. Mit dem Schließen der Verbindung wird auch der Prozess beendet. Hervorzuheben ist, dass für jede Verbindung ein Prozess gestartet wird und die Relation zwischen der Menge der Verbindungen und der Menge der Prozesse demnach bijektiv ist.

Der Einsatz von Superservern kann ressourcenschonend sein, wenn sich die Anzahl der gleichzeitigen Verbindungen nicht sehr hoch ist und die dazugehörigen Prozesse nicht zu komplex sind. Im klassischen Modell überwacht ein Serverprozess einen Port und wartet auf Verbindungen. Selbst wenn sämtliche Verbindungen geschlossen sind, läuft die Anwendung weiter und belegt somit durchweg Ressourcen, auch wenn diese zu dieser Zeit gar nicht benötigt werden. Hier zeigt sich der Vorteil des Superserver-Prinzips, der in diesem Fall die Ressourcen durch das Beenden der Prozesse wieder frei gibt und sie so anderen Anwendungen, die zu diesem Zeitpunkt möglicherweise aktiver arbeiten, zur Verfügung stellt.

Entsprechend der Natur eines solchen Servers, sind die zugewiesenen Dienste normalerweise nicht darauf erpicht, mit anderen aktiven Verbindungen zu kommunizieren, sondern arbeiten stets mit der ihnen zugewiesenen Verbindung zu einem Client. Dagegen stellt die Kommunikation zwischen den Verbindungen beim TulpaServer einen zentralen Bestandteil dar, indem über die API Funktionen angeboten werden, über die sich genau festlegen lässt, welche Nachricht an welche Gruppe von Clients versendet wird.

Vergleichend gibt es Parallelen zwischen Superservern und dem TulpaServer, welche sich aber im wesentlichen auf das zwingende Vorhandensein von weiterverarbeitenden Diensten beschränkt. Zudem ist die Verwaltung der Verbindungen innerhalb des TulpaServers eine essentielle Funktion des Software-Systems, die den Serverdiensten die Arbeit erleichtern soll, wobei diese nicht nur weitergereicht, sondern auch im vollen Umfang durch die Plugins manipuliert und kategorisiert werden können. Die Interaktion zwischen unterschiedlichen Clients spielt dabei eine wichtige Rolle und es werden entsprechende Werkzeuge geliefert, um dies einfach und schnell realisieren zu können. Dazu ist es unter Anderem erforderlich, dass Eigenschaften zwischen ihnen geteilt werden und zu jedem Zeitpunkt änderbar sein sollen. Dies wäre zwar auch zwischen den Diensten des Superservers möglich, indem man einen Shared-Memory-Bereich mit Semaphoren verwaltet, jedoch wäre dies bei Weitem nicht so komfortabel, als mit direkten Objektinstanzen zu arbeiten.

Ein weiterer Unterschied ist, dass die Serverdienste, beziehungsweise die Plugins nicht gestoppt, sondern stets am Laufen gehalten werden. Dies hat zwar negative Auswirkungen auf den Ressourcenverbrauch, jedoch sollte sich die unnötige Belegung des Speichers durch die Verwendung von Threads in Grenzen halten, da nicht stets der gesamte Programmcode für jede Verbindung in den Speicher geladen werden muss, sondern sich dieser zwischen den simultan laufenden Threads geteilt wird.

6.2 Vergleich SmartFox-Server

Der SmartFox-Server ist eine unter Java programmierte Software, welche das Ziel verfolgt, die Kommunikation mehrer Flash-Clients untereinander so einfach wie möglich zu gestalten. Dazu bietet sie dem ActionScript Entwickler eine clientseitige API, welche den Netzwerkverkehr zwischen Client und Server übernimmt und eine Schnittstelle für Kommunikation mit dem SmartFox-Server bietet.

Diese Serversoftware war die Inspiration für die Entwicklung des TulpaServers und so gibt es viele Parallelen zwischen ihnen. Diese lassen sich beispielsweise in der Strukturierung der Verbindungen finden. So benutzt der TulpaServer ebenfalls Zonen, um Anwendungsbereiche voneinander zu trennen. Gleichzeitig dienen sie als Container für Räume, welche wiederum Container für Verbindungen darstellen. Jedoch fand dieses Konzept nur minimalistisch Einzug in den TulpaServer. Während der SmartFox-Server sehr zahlreiche Funktionen, wie zum Beispiel das Ernennen von Moderatoren, Verwaltung von Freundeslisten, dem Setzen von Limitierungen und vielen mehr, zur Verwaltung des Servers innerhalb der Zonen bietet, wurden diese Funktionalitäten bewusst reduziert.

Zwar gibt es einige Gemeinsamkeiten, jedoch verfolgt der TulpaServer ein anderes Ziel. Es wird versucht, die Netzwerkentwicklung für Programmierer so einfach wie möglich zu gestalten, ohne dabei selbst Einfluss auf den Netzwerkverkehr zu nehmen. Der SmartFox-Server dagegen benutzt, zur Umsetzung seiner Komfort-Funktionen, ein eigenes Netzwerkprotokoll und zwingt die Entwickler, die diese Software benutzen, zur Verwendung seiner API, welche nur unter der Programmiersprache ActionScript verfügbar ist. Somit beschränkt sich dieser Server auf die Zusammenarbeit mit nur einer Sprache. Der TulpaServer dagegen versucht eine Lösung für alle Programmiersprachen anzubieten, indem er allein dem Programmierer von Plugins die Entscheidung überlässt, welche Daten gesendet und empfangen werden sollen. Das Ziel bei der Entwicklung war es, einen generischen Server zu entwickeln, der in der Zahl der Verwendungsmöglichkeiten keine Restriktionen beinhalten sollte. Server für HTTP, IRC, POP3 und viele weitere sind so mit dem SmartFox-Server nicht umzusetzen, da er seine Prioritäten anders festlegt.

Der SmartFox-Server war der Impuls zur Entwicklung des TulpaServers: die Vereinfachung der Netzwerkprogrammierung sowie fertige Lösungen für Anwendungsfälle, auf die ein Entwickler treffen könnte. Jedoch ohne ihn auf eine Programmiersprache beschränken zu wollen oder anderweitig in den Netzwerkverkehr einzugreifen und diesem somit die Möglichkeit zu geben, eine Vielzahl von unterschiedlichen Anwendungen mit diesem System entwicklen zu können.

7. Anwendungsbereiche

Obwohl es eine Vielzahl von vorstellbaren Plugins gibt, die für den TulpaServer entwickelt werden könnten, muss dieses System nicht unbedingt die beste Wahl darstellen. Es gilt abzuschätzen, was eine Anwendung können soll, wie sie gegebenenfalls mit anderer Software interagiert, wie hoch die Performanceanforderungen sind und mit welcher Auslastung, auch bezüglich die simultanen Verbindungen, gerechnet werden muss. Zudem spielt auch die Erfahrung des Programmierers eine Rolle. Hat er nur wenig Erfahrung in C++, ist aber Spezialist in einer anderen Programmiersprache, so wird er diese mit hoher Wahrscheinlichkeit auch einsetzen, anstatt auf ein System zu setzen, in das er sich erst einarbeiten und sich zudem noch mit einer ihm weniger vertrauten Programmiersprache beschäftigen muss.

Der TulpaServer wurde dahingehend entwickelt, dass er mehrere kleinere bis mittelgroße Plugins beziehungsweise Anwendungen am Laufen hält. Für große Projekte ist dieser zwar nicht gänzlich ungeeignet, aber wenn die Performance einen sehr hohen Stellenwert einnimmt, sollte auf die Verwendung des Servers möglicherweise verzichtet werden. So liegen dem Funktionsprinzip der API virtuelle Klassen zu Grunde, was dazu führt, dass für jede abgeleitete Klasse der API eine sogenannte *vtable* generiert wird. Zudem erhält jedes erstellte Objekt einen Zeiger auf die entsprechende virtuelle Tabelle, in der die Einstiegspunkte für abgeleitete Methoden der entsprechenden Klasse aufgeführt sind. Wird eine virtuelle Methode aufgerufen, muss zuerst in der *vtable* nach der entsprechenden Einstiegsadresse nachgeschlagen werden, was CPU-Zeit (*Zyklen*) kostet und sich entsprechend aufsummieren kann. Zudem dauert die Dereferenzierung von Routinen innerhalb einer dynamischen Bibliothek länger (nach Angaben von IBM etwa acht Zyklen[1]), als es bei direkt einkompiliertem Quellcode der Fall ist. Auf Grund dieser Geschwindigkeitsnachteile, die durch das Plugin-System entstehen, ist der TulpaServer für hochperformante Anwendungen die

schlechtere Wahl. Wenn die Geschwindigkeit einer Anwendung oberste Priorität hat, sollte daher ein alleinstehendes Programm für das entsprechende Problem entwickelt werden, oder aber auf Hilfsmittel zurückgegriffen werden, die diese Performencenachteile nicht besitzen.

Seine Vorteile kann der TulpaServer vor Allem bei Entwicklergruppen oder Firmen ausspielen, die viele kleinere Anwendungen im Netzwerkbereich programmieren, die einen zentralen Hauptserver voraussetzen. Anstatt viele kleine Dienste zu entwickeln, bei denen der Netzwerkverkehr jedes Mal neu implementiert werden muss, könnten diese als Plugins umgesetzt werden, wobei die Netzwerkroutinen an zentraler Stelle implementiert wären. Stehen die Dienste untereinander in Abhängigkeit, ließe sich zudem die Kommunikation zwischen diesen über den TulpaServer realisieren. So könnte zum Beispiel *Plugin1* eine Callback-Funktion auf einen Raum innerhalb des Verwaltungsbereiches von *Plugin2* registrieren, und so auf Ereignisse innerhalb eines anderen Dienstes reagieren.

Speziell wenn die Verbindungen untereinander Interagieren sollen, kann der TulpaServer die Entwicklung beschleunigen, da er die dafür notwendigen Routinen bereits zur Verfügung stellt. Dies könnte für Programmierer interessant sein, die sich lieber auf das eigentliche Programm konzentrieren wollen, als auf derartige Nebensächlichkeiten.

7.1 Fallbeispiel A

Eine Firma plant die Umsetzung eines Web-Portales, auf dem die Benutzer in verschiedenen Kartenspielen gegeneinander antreten können. Dies soll auf Basis der in HTML5 verfügbaren WebSockets realisiert werden. Nach der Anmeldung des Benutzers soll dieser auf einen Lobbybereich weitergeleitet werden, in dem er sich mit anderen Spielern unterhalten und die Auswahl des gewünschten Kartenspieles treffen kann. Hat er diese Auswahl getroffen, wird der Spieler in einen Unterbereich geleitet, indem er das gewünschte Spiel einleiten oder bereits vorhanden Spielen beitreten kann. Nach diesem Arrangement soll das eigentliche Spiel beginnen.

Dies wäre ein Einsatzbeispiel, in welchem der TulpaServer die Entwicklung des Vorhabens beschleunigen könnte. Nach dem Login wird eine Verbindung zu diesem hergestellt und der Spieler in einen speziellen Lobbyraum eingegliedert, in welchem sich auch andere Spieler aufhalten. Diesem Raum wird ein PacketSerializer zugewiesen, welcher den Befehlssatz für diesen Bereich spezifiziert. Dies wäre unter anderem ein Paket für die Liste der anwesenden Spieler, ein Paket, welches neu beigetretene Spieler oder jene, die die Verbindung beendet haben nennt, ein Paket für Textnachrichten und eines für die Wahl des Spieles. Der neue Spieler erhält zu Beginn eine Liste der anderen Teilnehmer des Raumes, während alle Anderen lediglich den Namen des neuen Spielers zugesendet bekommen. Verlässt ein Spieler diesen Bereich, werden die verbleibenden Spieler ebenfalls benachrichtigt, so dass die Liste der Benutzer aktualisiert werden kann. Nebenbei werden Textnachrichten über das entsprechende Paket an alle Teilnehmer des Raumes gesendet, welche, zusammen mit dem Namen des Senders, in einem Textfeld angezeigt werden. Entschließt sich der Spieler nun für ein Kartenspiel, sendet er das entsprechende Paket an den Server, dass diesen dazu veranlasst, den Spieler aus dem Lobbyraum zu entfernen und ihn in den Eingangsbereich des entsprechenden Spieles verschiebt.

Da sich jedes Spiel, und somit auch das Einrichten der Spielesitzung unterscheidet, müssen für jedes Spiel unterschiedliche Befehlssätze (*PacketSerializer*) eingerichtet werden. Für ein Skatspiel wären beispielsweise genau drei Spieler nötig, damit es gespielt werden kann. Haben sich so viele Teilnehmer zusammengefunden, wird ein neuer Raum angelegt, in welchen die entsprechenden Verbindungen verschoben werden. Diesem wird erneut ein spezieller *PacketSerializer* zugewiesen, der den Befehlssatz für ein Skatspiel versteht.

Für jedes Spiel existiert nun eine Logik, die das Spielgeschehen überwacht und leitet. Es sorgt dafür, dass nur der Spieler eine Karte legen kann, der auch an der Reihe ist, und auch nur eine, die er auch „auf der Hand“ hat. Um dies zu realisieren, muss die Spiellogik die Nachrichten, respektive Pakete, der drei Spieler empfangen können. Dazu registriert diese für jeden Spieler eine Callback-Funktion für empfangene Pakete, wobei diese selbst eine Methode der entsprechenden Instanz der Logikklasse ist. Die Spiellogik empfängt nun die deserialisierten Pakete der Spielteilnehmer und kann ihnen, über die Session- bzw. Raumreferenz gezielt antworten und dabei unterscheiden, ob eine Antwort nur an eine bestimmte Verbindung geschickt werden muss (Senden des Pakets an die Session), oder an alle Spieler (Senden des Paketes an den Raum und somit an alle seine Teilnehmer).

Dieses Beispiel zeigt eine Möglichkeit auf, wie man Räume und *PacketSerializer* gezielt einsetzen kann, um verschiedene Verhaltensweisen zu implementieren. Der Lobbyraum, der Raum zum Beitritt in ein Spiel, sowohl der Spieleraum selbst setzen unterschiedliche Anforderungen an das System und den Netzwerkverkehr. Damit diese Anforderungen erfüllt werden können, kommen verschiedene *PacketSerializer* zum Einsatz, die genau die Teile des Netzwerkverkehrs abdecken, die in dem entsprechendem Bereich benötigt werden. Dies ermöglicht eine logische Trennung und vermeidet Verwaltungsaufwand (zum Beispiel das Filtern von Paketen, die in einem Bereich unzulässig sind). Die Räume werden genutzt, um die Spieler voneinander abzukapseln und so zu Gliedern, dass sie nicht unzulässig miteinander interferieren. Benutzer, die ein gemeinsames Spiel durchführen, können so mit den mitgelieferten Werkzeugen von denen getrennt werden, die beispielsweise noch auf ein Spiel warten. Indem man den Räumen spezifischen PacketSerializer zuweist, kann man die Reaktionen auf den Netzwerkverkehr einfach und direkt abändern, ohne dass ein womöglicher, erneuter Verbindungsaufbau nötig wird.

7.2 Fallbeispiel B

Ein Informatikstudent plant zur Aufbesserung seines Finanzhaushaltes die Entwicklung eines Einkaufsführers für technische Artikel. Diese Anwendung soll speziell für mobile Endgeräte konzipiert werden, so dass sich ein interessierter Käufer vor Ort über ein Produkt informieren kann. Zudem sollen sich die Betrachter eines Produktes untereinander in Echtzeit und in Textform unterhalten können, um so Erfahrungen darüber auszutauschen.

Der Student plant die Anzeige der Produktinformationen über HTML in Verbindung mit PHP auf der Serverseite zu lösen. Für den Echtzeitkommunikation zieht er die Entwicklung einer kleinen Serveranwendung unter C++ in Betracht, mit welcher sich sein Programm separat verbinden soll.

Bei der Entwicklung eines Prototypen fällt ihm jedoch auf, dass sein angemieteter, virtueller Server schon bei 20 gleichzeitigen HTTP-Verbindungen an seine Belastungsgrenze stößt. Dies ist viel zu wenig für sein Vorhaben und leider fehlt es ihm an den Mitteln, sich einen besseren Server zu mieten.

Anstatt diese Anwendung mittels Web-Techniken zu lösen, ließe sich ein Plugin für den TulpaServer in Betracht ziehen. Um Performanceengpässen entgegenzutreten, müsste der Rechenaufwand reduziert werden, wofür sich ein hardwarenahes Programm anbietet. Dafür könnte man die grafische Oberfläche fest in der mobilen Anwendung implementieren und nur die eigentlichen Produktinformationen, wie zum Beispiel die Beschreibung, Spezifikation und Bilder, übermitteln, was eine Verringerung des Netzwerkverkehrs mit sich bringen würde.

Beim Programmstart würde eine Verbindung zum Server aufgebaut werden, über welche nur die nötigsten Informationen gesendet werden. Über einen PacketSerializer ließe sich so ein entsprechendes Protokoll realisieren lassen, welches die Suche nach, sowie die Anzeige von Produkten unterstützt. Ruft ein Benutzer die Informationen für ein bestimmtes Produkt ab, wird ein Raum für dieses erstellt, sofern er noch nicht vorhanden ist, und die Benutzerverbindung diesem zugeordnet. Dieser Raum bekommt einen neuen PacketSerializer zugewiesen, welcher zusätzlich einen Befehlssatz für die Textkommunikation zwischen mehreren Benutzern unterstützt. Sämtliche Benutzer in diesem Raum hätten somit die Möglichkeit, sich über das entsprechende Produkt auszutauschen. Beim Wechsel in eine andere Perspektive der grafischen Oberfläche, wird die Verbindung aus dem Raum entfernt und die verbleibenden Benutzer über diesen Vorgang benachrichtigt.

8. Implementationsdetails

8.1 Logging-System

Das Logging-System wurde nach dem Verteiler-Entwurfsmuster entwickelt. Die Klasse Proxy agiert dabei als Verteiler der Nachrichten. Diese werden solange in einen *ostringstream* geschrieben, bis ein Flush durch eines der beiden *flush_symbols* ausgelöst wird. Diese Symbole sind Teil einer Enumeration der Proxy-Klasse innerhalb der API. Durch diesen Mechanismus ist es möglich, die Nachricht in mehreren Teilen zu konstruieren und die Handhabung mit unterschiedlichen Datentypen, welche in den Stream geschrieben werden sollen, zu vereinfachen. Erkennt die Proxy-Klasse, dass ein Flush-Symbol per Shift-Operator übergeben wurde, wird ein String aus dem Inhalt des Puffers gebildet, der anschließend an alle registrierten Geräte weitergereicht wird. Dabei gibt es keinerlei Vorschriften, wie sie die Daten auszuwerten und zu benutzen haben. Die Übergabe erfolgt durch die Methode *void log(const std::string& message),* welche von allen abgeleiteten Klassen von *Device* implementiert werden muss.

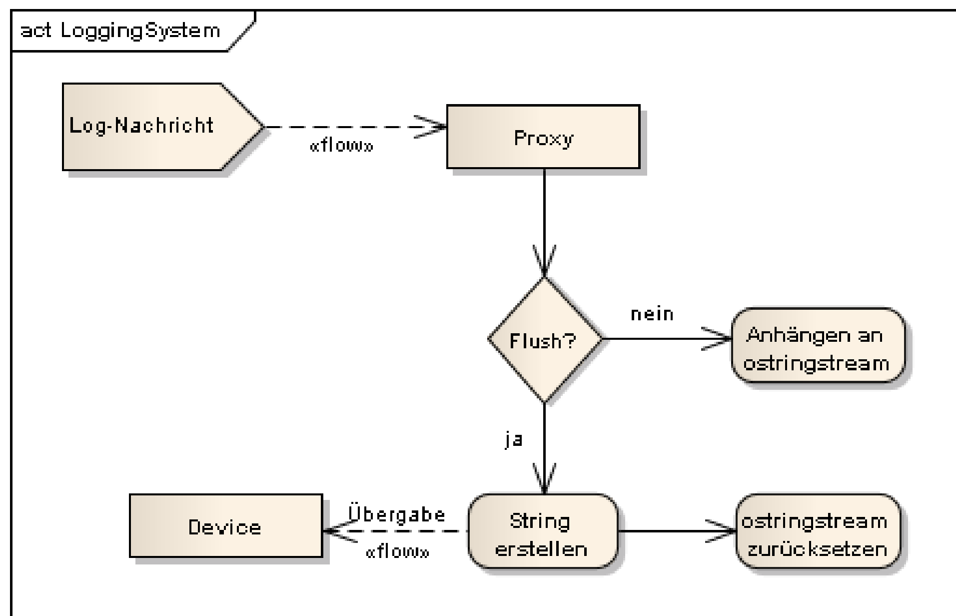

Abb. 2: Funktionsweise / Informationsfluss des Logging-Systems

Die Headerdatei *LevelBased.h* deklariert diverse Precompiler-Konstanten, durch welche eine Filterung der Nachrichten nach Priorität erreicht wird. Dabei wird überprüft, ob das Level der Log-Nachricht größer oder gleich der Filtervorgabe ist. Ist dies der Fall, wird eine globale Proxy-Instanz

erfragt, welcher die Nachricht übergeben wird, wobei dieser ein aktueller Zeitstempel und der Name des Log-Levels vorangestellt wird.

So erzeugt die Befehlskette

```
#include <tulpa/api/logging/LevelBased.h>
// (...)
LOG_INFO << "starting server..." << LOG_ENDL;
// (...)
```

die folgende Nachricht als String, welcher anschließend an die Geräte weitergereicht und beispielsweise auf einer Konsole ausgegeben wird.

```
[12.11.2011 15:52:08] INFO: starting server...\n
```

8.2 Server-Klassen

Als Serverklassen werden jene Klassen bezeichnet, die zum Starten eines Server-Sockets und zum Annehmen von Verbindungen benutzt werden. Das Wurzelelement in der Klassenhierarchie stellt dabei die abstrakte Klasse *Server* innerhalb der API dar. Diese definiert die drei Hauptmethoden *run*, *stop* und *running*. Die *running* Methode dient der Feststellung, ob der Server gestartet oder gestoppt ist, während *run* zum Starten und *stop* zum Beenden des Serverprozesses genutzt werden kann. Drei weitere Methoden dienen dem Erfragen des zu benutzenden Ports, der *Factory*-Instanz sowie der zugewiesenen Zone.

```
namespace tulpa{ namespace api{ namespace networking
{
     class Server
     {
           public:
            typedef boost::shared_ptr<Server> pointer;
                 virtual ~Server() {};

                 virtual void run() = 0;
                 virtual void stop() = 0;
                 virtual bool running() const = 0;
                 virtual port_t port() const = 0;
                 virtual Factory::pointer factory() const = 0;
                 virtual Zone* zone() const = 0;
     };
} } }
```

Die Headerdatei tulpa/api/networking/Server.h (gekürzt)

Da die Verwendung mehrerer Protokolle vorgesehen ist und somit eine Serverklasse für jedes Protokoll implementiert werden soll, wurde die Klasse *ServerBase* eingeführt, welche sich direkt von *Server* ableitet, jedoch kein Teil der API ist. Diese stellt die Basis für die detaillierten Protokollimplementation dar und hilft, Quellcodewiederholungen zu vermeiden. Dazu implementiert sie die Methoden *running*, *port*, *factory* und *zone* und definiert die nötigen Membervariablen, die zur Funktionsweise dieser benötigt werden. Die Methoden *run* und *stop* bleiben jedoch abstrakt, da deren Implementation abhängig vom verwendeten Protokoll ist. Zwar ließe sich das Protokoll auch über einen Funktionsparameter der *run* Methode selektieren, jedoch soll darauf, zu Gunsten einer besseren logischen Trennung, im Hinblick auf eine Plattformunabhängigkeit verzichtet werden.

```
namespace tulpa{ namespace networking
{
	class ServerBase
		: public Server,
		private boost::noncopyable
	{
		public:
			ServerBase(Factory::pointer p_factory, port_t p_port, Zone*
p_zone)
					: m_running(false),
					m_factory(p_factory),
					m_port(p_port),
					m_zone(p_zone) {}

			virtual ~ServerBase() {}

			virtual void run() = 0;
			virtual void stop() = 0;

			virtual bool running() const { return m_running; }
			virtual port_t port() const { return m_port; }
			virtual Factory::pointer factory() const { return m_factory; }
			virtual Zone* zone() const { return m_zone; }

		protected:
			virtual void running(bool p_running) { m_running =
p_running; };

		private:
			volatile bool m_running;
			Factory::pointer m_factory;
			port_t m_port;
			Zone* m_zone;
	};
} }
```

Die Klasse ServerBase (gekürzt)

Am unteren Ende der Hierarchie stehen die eigentlichen Implementation der Server abhängig von ihrem Protokoll, wobei der TulpaServer derzeit nur TCP unterstützt. Die Klasse *TCPServer* setzt nun die Implementation, der von *Server* vorausgesetzten Methoden, fort und definiert *run* und *stop*. Innerhalb der *run* Methode setzt die Klasse die Socketoptionen, bindet dieses an einen Port und akzeptiert in einer Endlosschleife Verbindungen. Dazu werden die zusätzlichen Methoden *accept* und *handle_accept* benutzt, die die Verbindungen annehmen, konfigurieren und sie der entsprechenden Zone zuweisen.

```
void TCPServer::accept()
{
	socklen_t client_addr_len = sizeof(m_client_addr);
	m_accepted_socket = ::accept(m_acceptor_socket, reinterpret_cast<struct
sockaddr*>(&m_client_addr), &client_addr_len);

	if(m_accepted_socket < 0)
	{
		LOG_ERROR << "error accepting connection" << LOG_ENDL;
		return;
	}

	PacketSerializer::pointer new_packet_serializer(factory()-
>new_packet_serializer());
	Socket* sock = new Socket(m_accepted_socket);
	SessionImpl* new_session = new SessionImpl(sock, new_packet_serializer);
	handle_accept(new_session);
}

void TCPServer::handle_accept(SessionImpl* p_new_session)
{
	m_session_threads.create_thread(boost::bind(&SessionImpl::recv,
p_new_session));
	Session::pointer session_sptr(p_new_session);
	zone()->add_session(session_sptr);
}
```

Implementierung von TCPServer::accept und TCPServer::handle_accept

Die Annahme der Verbindung erfolgt auf normalem Wege. Ist dies fehlerlos geschehen, wird der Verbindung ein eigener *PacketSerializer* zugewiesen, dessen Erstellung durch die vom Plugin festgelegte *Factory* übernommen wird. Die Zuweisung geschieht über den Konstruktor der Klasse *SessionImpl*, welche fortan für die Verwaltung der Verbindung verwendet wird. Anschließend wird dieses Objekt über die Methode *handle_accept* dem System bekannt gemacht und weiterführend initialisiert. Dem Thread-Pool wird dazu ein neuer Thread hinzugefügt, der das Empfangen von Daten aus dem Socket der neu erstellten Verbindung übernimmt. Schlussendlich wird die Session der entsprechenden Zone zugewiesen, welche diese einordnet und eingetragene Listener über die Annahme einer neuen Verbindung über ein Signal informiert.

8.3 Sessions und TransferTasks

Die *SessionImpl* Klasse, welche sich von der abstrakten Klasse *Session* der API ableitet, dient in erster Linie dazu, den Datenfluss zwischen Server und einer Verbindung zu steuern, indem sie Methoden zum Senden und Empfangen von Daten über das Client-Socket implementiert. Wie bei anderen Klassen auch, lassen sich Event-Listener anbinden, welche bei bestimmten Signalen benachrichtigt werden und somit auf diese reagieren können. Die Signale, welche von *SessionImpl* implementiert sind, umfassen das Senden und Empfangen von Paketen sowie einen Verbindungsabbruch zum Client, wobei dieser gewollt oder das Resultat eines Fehlers sein kann.

Das Senden von Daten wird über sogenannte *Transfer Tasks* (Übermittlungsaufgaben) realisiert. Die abstrakte Klasse *TransferTask* ist eine Schnittstelle der API, die es ermöglicht, die komplexe Übermittlung von Daten aus beliebigen Quellen ressourcenschonend durchzuführen. Dazu werden die zwei abstrakten Methoden *prepare* und *write* deklariert, welche einen boolschen Wert zurückgeben müssen, der den Erfolg (true) oder den Misserfolg (false) der Aktion darstellt, oder angibt, ob die Aktion fertiggestellt wurde.

```
namespace tulpa{ namespace api{ namespace networking{ namespace task
{
	class TransferTask
	{
		public:
			virtual ~TransferTask() {};
			virtual bool prepare() = 0;
			virtual bool write(NetStreamBuffer& p_nsb) = 0;
	};
} } } }
```

TransferTask Schnittstelle (gekürzt)

Die *prepare* Methode dient dem eventuellen Vorbereiten der Transaktion und wird vor Beginn dieser einmalig aufgerufen. Sie kann beispielsweise dazu benutzt werden, um eine Datei zu öffnen oder den Schreibprozess anderweitig vorzubereiten. Die Methode *write* kann dagegen mehrmals aufgerufen werden, was sich durch den Rückgabewert beeinflussen lässt. Dieser lässt sich mit „wurde die Aktion abgeschlossen?“ beschreiben. Wird *false* zurückgegeben, signalisiert dies, dass der Vorgang noch nicht abgeschlossen wurde und ein weiterer Aufruf notwendig ist.

Der TulpaServer liefert zwei fertige Übermittlungsaufgaben , die von Plugins verwendet werden können, ohne dass sie diese selbst implementieren müssen. Dies ist zum einen der *PacketTransferTask*, der zum Versenden von Paketen benutzt wird und der *StringTransferTask* zum einfachen Übermitteln von Zeichenketten.

```
PacketTransferTask::PacketTransferTask(Packet* p_packet)
      : m_packet(p_packet)
{
}

PacketTransferTask::~PacketTransferTask()
{
      delete m_packet;
}

bool PacketTransferTask::prepare()
{
      return true;
}

bool PacketTransferTask::write(NetStreamBuffer& p_nsb)
{
      m_packet->write(p_nsb);
      return true;
}
```

Implementationsdatei PacketTransferTask.cpp (gekürzt)

```
StringTransferTask::StringTransferTask(const std::string& p_str)
      : m_str(p_str)
{
}

StringTransferTask::~StringTransferTask()
{
}

bool StringTransferTask::prepare()
{
      return true;
}

bool StringTransferTask::write(NetStreamBuffer& p_nsb)
{
      p_nsb << m_str;
      return true;
}
```

Implementationsdatei StringTransferTask.cpp (gekürzt)

Die Beispielanwendung *httpd* führt einen weiteren TransferTask zum Versenden von Dateien ein. In der *prepare* Methode wird die Datei geöffnet, die Größe ermittelt und der entsprechende Erfolgswert zurückgegeben. Innerhalb von *write* wird danach der Dateiinhalt nach und nach in das Socket geschrieben. Die Beendigung des Übermittlungsvorganges wird mittels *std::ios::eof* überprüft.

```
///////////////////////////
// Datei httpd_filetrtask.h

#ifndef HTTPD_FILETRTASK_H
#define HTTPD_FILETRTASK_H

#include <string>
#include <fstream>
#include <tulpa/api/networking/NetStreamBuffer.h>
#include <tulpa/api/networking/task/TransferTask.h>

#include <iostream>

using tulpa::api::networking::task::TransferTask;
using tulpa::api::networking::NetStreamBuffer;

class httpd_filetrtask : public TransferTask
{
	public:
		httpd_filetrtask(const std::string& p_file_path);
		virtual ~httpd_filetrtask();
		virtual bool prepare();
		virtual bool write(NetStreamBuffer& p_nsb);
		size_t get_file_size() const;

	private:
		static const int BUFFER_SIZE = 8193;
		std::ifstream m_file;
		std::string m_file_path;
		size_t m_file_size;
		char m_fbuf[BUFFER_SIZE];
		size_t m_fbuf_avail;
};

#endif // HTTPD_FILETRTASK_H

/////////////////////////////
// Datei httpd_filetrtask.cpp

#include "httpd_filetrtask.h"

httpd_filetrtask::httpd_filetrtask(const std::string& p_file_path)
	: m_file(),
	m_file_path(p_file_path),
	m_file_size(0),
	m_fbuf(),
	m_fbuf_avail(0)
{
}
```

```
httpd_filetrtask::~httpd_filetrtask()
{
	if(m_file.is_open())
		m_file.close();
}

bool httpd_filetrtask::prepare()
{
	m_file.open(m_file_path.c_str(), std::ios::in | std::ios::binary |
std::ios::ate);
	if(m_file.good())
	{
		m_file_size = m_file.tellg();
		m_file.seekg(0);
		m_fbuf[BUFFER_SIZE-1] = '\0';
		return true;
	}
	else return false;
}

bool httpd_filetrtask::write(NetStreamBuffer& p_nsb)
{
	m_file.read(m_fbuf, BUFFER_SIZE-1);
	m_fbuf_avail = m_file.gcount();
	p_nsb.write_string(m_fbuf, m_fbuf_avail);
	return m_file.eof();
}

size_t httpd_filetrtask::get_file_size() const
{
	return m_file_size;
}
```

Jede Sessioninstanz besitzt ihre eigene Sendewarteschlange, die der Reihe nach abgearbeitet wird. Soll ein Paket an einen Client verschickt werden, wird der dazugehörige *TransferTask* in die Warteschlange eingereiht und ausgeführt, sobald alle vorherigen Übermittlungsaufgaben abgeschlossen wurden. Nach Beendigung des Tasks wird das Objekt gelöscht und somit der Destruktor der Klasse aufgerufen, in welchem die Aufräumarbeiten (z.B. das Freigeben von allokiertem Speicher) vorgenommen werden können. Falls eine Beendigung der Verbindung von der Serverseite gewünscht wird, wird zudem sichergestellt, dass die momentane Übermittlungsaufgabe abgeschlossen wird, bevor die Verbindung geschlossen wird.

```
void do_send()
{
	boost::recursive_mutex::scoped_lock scoped_lock(m_mutex_send);
	m_is_sending = true;
	TransferTask* ttp = NULL;
	bool job_done;
	while(!m_close_requested && !m_send_queue.empty())
	{
		ttp = m_send_queue.front();
		m_send_queue.pop();
```

```
            if(ttp->prepare())
            {
                  do
                  {
                        job_done = ttp->write(m_nsb);
                  } while(!job_done);
            }
            delete ttp;
      }
      m_is_sending = false;

      if(m_close_requested)
      {
            close_immediately();
      }
}
```

Die Methode SessionImpl::do_send zum Abarbeiten der Warteschlange

8.4 NetStreamBuffer

Die Klasse *NetStreamBuffer* vereinfacht die Lese- und Schreibzugriffe auf ein Socket für den Entwickler, wozu primär Streamoperatoren eingesetzt werden. Während Schreibzugriffe direkt an das Socket weitergeleitet werden, werden Lesezugriffe über einen Puffer behandelt. Dabei spielt die Größe des zu lesenden Datentyps eine entscheidende Rolle. Um zu ermitteln, ob sich die angefragten Daten im Puffer befinden, wird dessen aktuelle Größe mit der des angegebenen Datentyps verglichen und somit festgestellt, ob die Daten direkt aus dem Speicher gelesen, oder ob zuerst weitere Bytes aus dem Socket empfangen werden müssen.

Der interne Puffer ist eine Eigenentwicklung, die bei drohendem Pufferüberlauf unverarbeitete Daten an den Anfang des Puffer verschiebt und damit bereits gelesene Daten überschreibt. Dies geschieht über die C-Funktion *memmove*, welche einen angegebenen Speicherbereich zuerst dupliziert und anschließend an die Zielposition schreibt. Dies kann besonders bei klein gewählten Puffern zu Performanceproblemen führen, da die Speicherverschiebung hier häufiger durchgeführt werden muss. Generell wird versucht, ein Drittel des Pufferbereiches für Leseoperationen aus dem Socket freizuhalten. Ist der Schreibbereich des Puffers kleiner als dieser Wert, erfolgt die Verschiebung des Lesebereichs an die Startadresse des Puffers.

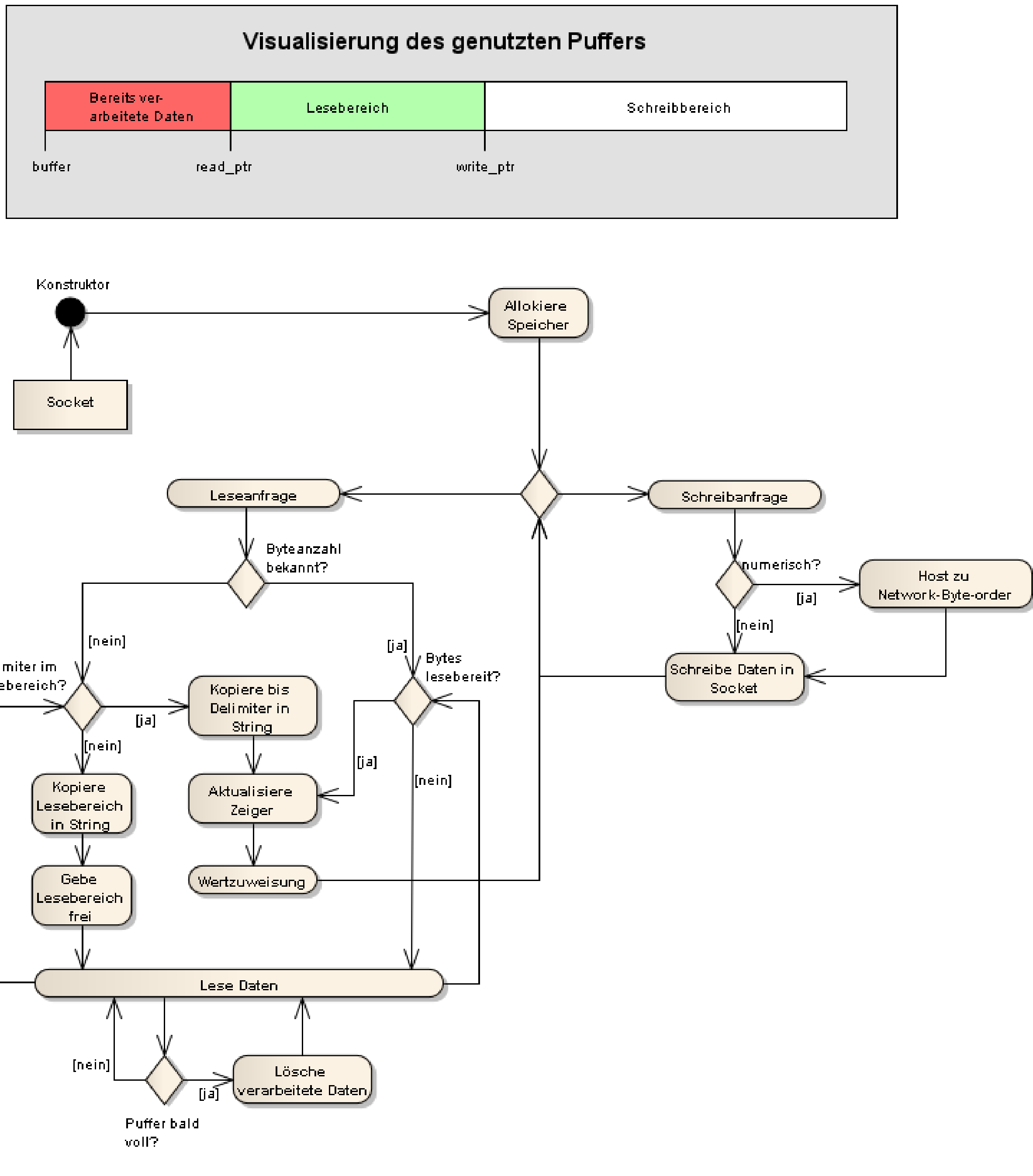

Abb. 3: Funktionsweise NetStreamBuffer

An dieser Stelle wäre die Nutzung eines Ringpuffers die bessere Wahl gewesen, da dabei keine Speicherverschiebung erforderlich wäre. Jedoch traten beim Implementationsversuch erhebliche Probleme bei der Nutzung dieses Puffers mit den Funktionen der C-API auf, da diese einen kontinuierlichen Speicherbereich benötigen, was beim einem Ringpuffer nicht der Fall ist. Dadurch erhöht sich der Verwaltungsaufwand, da stets geprüft werden muss, ob ein errechneter Speicherbereich auch durchgehend ist, wenn ein Zeiger auf den internen Puffer zurückgeliefert

werden soll. Trotz dieser Probleme soll die Implementation des NetStreamBuffers über einen Ringpuffer aber in der Zukunft getestet und realisiert werden.

9. Beispiel-Plugins

9.1 httpd

Das Plugin *httpd* ist ein äußerst simpler HTTP-Server für den TulpaServer. Es zeigt, wie pluginspezifische Werte aus der Konfigurationsdatei gelesen werden können, wie sich Event-Handler registrieren lassen und implementiert einen eigenen *TransferTask* sowie zwei Pakete.

In der Methode *on_enable* der Klasse *httpd_addon* wird zunächst versucht, den Wert *html_root* aus der Konfigurationsdatei zu lesen. Dieser bezeichnet das Verzeichnis, in dem sich die HTML-Dateien befinden und wird als Wurzelverzeichnis verwendet. Wurde der Wert nicht in der Konfigurationsdatei angegeben, wird */tmp* als Standardwert benutzt. Anschließend wird eine Callback-Funktion für das Signal *sig_session sig_s_add* (siehe Anhang A.) der Klasse *Zone* registriert, wozu die Methode *on_connect* benutzt wird. Verbindet sich daraufhin ein Client mit dem System, wird diese Methode aufgerufen und erhält als Parameter die Zone des Plugins sowie einen Pointer auf die Session. In dieser wird für das Signal *sig_packet sig_pr* der Klasse *Session* eine weitere Callback-Funktion registriert, die beim Empfangen eines Paketes vom Client aufgerufen wird.

Für den Datenaustausch zwischen Client und Server benutzt dieses Plugin zwei eigene Pakete. Dabei handelt es sich um *httpd_packet_request* (jene Daten, die ein Browser zum Server sendet, um eine Datei herunterzuladen) und *httpd_packet_reply* (die Antwort des Servers auf die Anfrage), welche die deserialisierte Form des HTTP-Headers repräsentieren. Es wird hierbei davon ausgegangen, dass nur Anfragepakete (*httpd_packet_request*) vom Client empfangen werden, die anschließend ausgewertet werden, um diesem eine Anwort auf die Anfrage zu senden. Diese werden von der Klasse *httpd_marshaller* erstellt, indem der Netzwerkverkehr bis zu der, durch das von HTTP spezifizierte, Trennzeichenfolge *\r\n\r\n* gelesen und die Daten Zeile für Zeile ausgewertet werden.

Wurde die Deserialisierung des Paketes abgeschlossen, erhält die Methode *on_packet* der Klasse *httpd_addon* einen Zeiger auf dieses und bereitet anschließend ein Antwortpaket zum Zurücksenden vor. Da HTTP-Antworten zusätzlich den Dateiinhalt der angeforderten Datei enthalten müssen, wird ein Objekt der Klasse *httpd_filetrtask* erstellt, welches die Übermittlung von Dateien realisiert. Die finale Antwort besteht demzufolge aus zwei Teilen. Dem HTTP-Header und dem eigentlichen Dateiinhalt, welche nacheinander gesendet werden. Nach der Übermittlung wird die Verbindung durch den Server geschlossen und der Client kann eine erneute Verbindung aufbauen, falls dies notwendig ist.

9.2 simplechat

Das zweite Plugin zeigt eine mögliche Vorangehensweise zur Realisierung einer serverseitigen Chatanwendung auf. Der Initiierungsvorgang verläuft dabei anolog zum *httpd*-Plugin. So werden auch hier Callback-Funktionen für neue Verbindungen und für das Empfangen von Paketen registriert.

Die definierten Pakete umfassen das Festlegen eines Benutzernamens für eine Verbindung, dem Senden einer Nachricht sowie dem Beitritt zu einem Raum. Zur Unterscheidung der Pakete deklariert das Basispaket *scp* eine ID mit einer Länge von einem Byte. Beim Deserialiseren des Netzwerkstroms ist es somit ausreichend, ein einziges Byte zu lesen, um herauszufinden, um welche Art von Paket es sich handelt und es dementsprechend zu erstellen.

Dieses Plugin ist als ein Lehrbeispiel zu verstehen, da kein entsprechender Client dazu existiert. Es dient dem Aufzeigen weiterer Möglichkeiten, wie man die Netzwerkroutinen des TulpaServers benutzt und mit diesem, respektive seinen Teilen, interagiert.

10. Verbesserungsmöglichkeiten

Auch wenn die Einbindung eines Plugin-Systems von vornherein feststand, so ist dieses während der Entwicklung zu sehr in den Fokus gerückt. Anstatt potentiellen Benutzern vorzuschreiben, dass sie ein spezielles Programm starten müssen, damit sie ihre Software benutzen können, wäre es wünschenswert gewesen, dies nur optional anzubieten. Viele der Hilfstechniken zur Vereinfachung der Netzwerkprogrammierung hätten auch innerhalb einer Bibliothek angeboten werden können. Die bereits vorhandene API hätte, mit kleineren Änderungen, auch in diesem Fall eingesetzt werden können. Programmierer hätten so die Wahl bekommen, sich zwischen einer der beiden Funktionsweisen, basierend auf ihren eigenen Bedürfnissen, zu entscheiden. Der jetzige Stand der Entwicklung erlaubt es beispielsweise nicht, dass der TulpaServer in Serverprogrammen eingesetzt wird, die veröffentlicht werden sollen. Denn kaum ein Entwickler wird seinen Benutzern vorschreiben wollen, ein derartiges Softwaresystem auf ihren Servern einzusetzen. Speziell deswegen, weil es anschließend noch konfiguriert werden muss. Hier würde sich das Einbinden des TulpaServers als statische Bibliothek anbieten. Es würde dem Programmierer die gleichen Vorteile bieten und dabei die Installation einer „Drittsoftware“ für den Endbenutzer vermeiden.

Des Weiteren sollte der Server nicht nur das TC-Protokoll unterstützen. Zumindest UDP sollte als zusätzliches Protokoll implementiert werden, wobei sich die Bedienung der API nach Möglichkeit nicht unterscheiden sollte. Auch die Einbindung eines verschlüsselten Datentransfers über TLS/SSL wäre wünschenswert. Gerade wenn es um den Transfer sensibler Daten über ein Netzwerk geht, ist die Verschlüsselung dieser äußerst wichtig und sollte daher standardmäßig implementiert sein.

Auch das Logging-System sollte verbessert werden. Zwar erfüllt es seinen Zweck und ist weitestgehend konfigurierbar, jedoch nutzt es globale Variablen und C-Funktionen, was in einer vollständig objektorientierten Software einen unausgereiften, beziehungsweise uneinheitlichen, Eindruck erweckt. Um die Logging-Funktionalität einheitlicher zu gestalten, sollte die Zeitstempelfunktion zu einer Methode einer Klasse umgewandelt werden, die globale Proxy-Instanz entfernt und die Precompilerstatements zum Erfragen dieser entfernt werden. Zudem wäre es notwendig, das entsprechende Log-Level an die Geräte zu übergeben.

11. Weiterentwicklung der Software

Aus persönlichem Interesse heraus, soll dieses Projekt auch nach der Abgabe weiterentwickelt werden. Das langfristige Ziel ist der Voranschritt zu einem abgerundeten Softwareprodukt und dessen anschließende Veröffentlichung. Der derzeitige Entwicklungsstand bietet eine solide Basis für diese Planung. Jedoch müssen bis dahin noch einige Dinge verfeinert und ausgiebig getestet werden. Die Serveranwendung soll auf mehreren Betriebsystemen und als Bibliothek in Anwendungen einsetzbar sein, um den Interessentenkreis zu erweitern. Letzteres ist speziell bei Serverprogrammen notwendig, die veröffentlicht werden sollen, da hier zumeist gewünscht wird, die unterliegende Funktionsweise verdeckt zu halten. Dagegen spricht der Zwang, den TulpaServer als Anwendung bei einem Endbenutzer zu installieren. Stattdessen sollten dessen Funktionalitäten in ein Programm eingebunden werden und von diesem gestartet werden.

Ebenso soll ein Weg gefunden werden, die Abhängigkeiten der Plugins vom Betriebssystem und dem Compiler zu lösen, um Konfliktpotential zu minimieren. Hierzu könnten die Plugins optional als LUA-Skripte eingebunden werden, wodurch die entstehenden Problematiken durch unterschiedliche Compiler gelöst werden könnten. Somit könnten Plugins über Betriebssystemgrenzen hinaus publiziert und eingesetzt werden. Dem steht jedoch der Performanceverlust durch die Interpretation entgegen, wobei der Entwickler jedoch selbst abschätzen muss, was für seine Bedürfnisse am Besten geeignet ist.

Der Programmierer soll dabei stets die Wahl haben, ob er den TulpaServer als eigene Anwendung, oder ihn aber als Bibliothek einbinden möchte. Da die API zur Umsetzung beider Varianten nur geringfügig abgeändert werden muss, sollte dies leicht zu realisieren sein.

Ferner sollte geprüft werden, ob sich einige der derzeit eingesetzten Konzepte noch weiter vereinfachen lassen. Dies ergibt sich aber in der Regel von selbst, wenn der Server für richtige Projekte (keine Testanwendungen) verwendet wird und die Benutzer ihr Feedback dazu abgeben. Auch muss getestet werden, inwiefern Performanceprobleme bei vielen Verbindungen auftreten und an dieser Stelle gegebenenfalls optimiert werden. Es ist zudem nicht auszuschließen, dass noch weitere Komponenten verbessert oder von Grund auf neu gestaltet werden.

12. Persönliche Weiterentwicklung

Mit dem Erstellen dieser Arbeit konnten meine Kenntnisse unter C/C++ enorm ausgeweitet werden. Bei Recherchen zu Problemen, generellen oder optimalen Lösungswegen konnte häufig ein tiefer gehender Einblick in die Funktionsweisen des Betriebsystems und des Compilers gewonnen werden, welches sich positiv auf die eigenen Programmiertechniken unter C++ ausgewirkt hat.

Auch die Benutzung der Boost-Bibliotheken hat mir verstehen geholfen, wie man C++ effektiv einsetzen kann. Da die Boost-Dokumentation zwar recht umfangreich ist, aber häufig die Erklärung der Funktionsweise der Komponenten außen vor lässt, mussten diese durch das Lesen des Quellcodes in Erfahrung gebracht werden. Die Boost-Bibliotheken setzen massiv auf die Verwendung von Templates und haben mir viele Verfahren gezeigt, wie man diese äußerst effektiv einsetzen kann und dadurch Arbeitszeit spart.

Bei dem Entwurf des Pluginsystems ließ sich ein tieferer Einblick zur Arbeitsweise von Bibliotheken, deren Symboltabelle, dem Linken und der Funktionsweise der Vererbung in C++, speziell die *vtables*, gewinnen. Letztere sind besonders interessant, wenn man Mehrfachvererbung einsetzt, bei dem sich Methoden der abgeleiteten Klassen gleichen, oder die irgendwo im Abhängigkeitsbaum eine gleiche Basis besitzen und es herauszufinden gilt, welche Methode welcher Klasse aufgerufen wird. Mit diesem Wissen kann man Vererbung gezielter einsetzen und genauer abschätzen, wann man eventuell darauf verzichten sollte.

Besonders hervorzuheben ist der Erfahrungsgewinn im Bereich der Netzwerkprogrammierung. Durch die intensive Auseinandersetzung mit dem Thema, konnten Lösungsansätze gefunden werden, mit der sich diese effizient umsetzen lässt. Zwar mögen solche für einen professionellen Programmierer zum Standardrepertoire gehören, jedoch wird dieser Bereich in Büchern und Artikeln gerne unterschlagen. Dies führte Anfangs zu diversen Schwierigkeiten bei der Umsetzung, welche aber dennoch gelöst werden konnten und somit zu einem tieferen Verständnis dieser Thematik führten.

13. Fazit

Das Ziel der vorliegenden Arbeit war es, einen generischen Server unter C++ zu entwerfen, der einem Entwickler seine Arbeit im Bereich der Netzwerkkommunikation vereinfacht, ohne ihn dabei einzuschränken. Dazu wurde eine Schnittstelle entwickelt, über die es möglich ist, primitive Datentypen auf einfache Art und Weise in ein Socket zu schreiben oder daraus herauszulesen. Diese wird über ein vorgefertigtes Zugriffsmuster (dem *PacketSerializer*) angesteuert, wobei es jedoch Aufgabe des Programmiers sein muss, dieses zu implementieren und es so seinen Bedürfnissen in Bezug auf den syntaktischen Aufbau der Netzwerkdaten anzupassen. Ferner sollte eine Verwaltung der Verbindungen über vorgefertigte Routinen möglich sein, um diese kategorisch zu Gliedern und ihrer Gruppierung nach anzusteuern, was durch die Einführung von Zonen und Räumen erreicht wurde.

Während die genannten Bereiche funktionstüchtig sind, sollten sie jedoch vor einer eventuellen Veröffentlichung optimiert werden. Dies betrifft vor allem den internen Puffer der Klasse *NetStreamBuffer*, der unnötige und aufwändige Speicherverschiebungen durchführt, die durch die Verwendung eines Ringpuffers vermieden werden könnten. Auch sollte sich der TulpaServer nicht auf die Verwendung von Plugins versteifen, da dies für die Funktion des zu Grunde liegenden Systems eigentlich nicht notwendig wäre und selbige auch durch eine dynamische Bibliothek angeboten werden könnten.

Trotz einiger Mängel, die durchaus behoben werden können, ohne das ganze System verwerfen zu müssen, wurden mit dieser Arbeit die angegebenen Ziele erreicht. Durch zahlreiche, ursprüngliche Lösungsansätze, von denen einige weiter optimiert und andere gar gänzlich entfernt wurden, entstand die generische Serveranwendung in ihrer jetzigen Form, die eine solide Basis für ein ausgereiftes Softwareprodukt darstellen sollte.

14. Installationshinweise

Zum Kompilieren und Ausführen der Anwendung benötigen Sie ein linuxbasierendes Betriebssystem sowie die Boost-Bibliotheken *libboost_system.so*, *libboost_signals2.so* und *libboost_thread.so*. Diese sollten Sie über den entsprechenden Paketmanager installieren können. Ferner werden die Bibliotheken *libm.so* und *libdl.so* benötigt sowie der Compiler *g++* der GNU Compiler Collection.

Kopieren Sie das Verzeichnis *tulpa-server* von der beigelegten CD auf die Festplatte in einen Ordner mit Lese- und Schreibberechtigung. Wechseln Sie anschließend in der Konsole auf das kopierte Verzeichnis und führen Sie den Befehl *make* aus, um die Anwendung samt der Beispielplugins zu kompilieren. Wurde dieser Vorgang erfolgreich abgeschlossen, wechseln Sie in das Unterverzeichnis *bin*, in welchem Sie die Hauptanwendung *tulpa-server*, zwei dynamische Bibliotheken (die Plugins) sowie eine Beispiel-Konfigurationsdatei mit der Dateiendung *info* vorfinden sollten, welche Sie nach Belieben modifizieren können.

Zum Testen des httpd-Plugins, suchen Sie in der Konfigurationsdatei nach dem Schlüssel *html_root* und weisen Sie diesem als Wert einen Verzeichnispfad zu, unter welchem sich vorzugsweise HTML-Dateien befinden sollten. Starten Sie nach dieser Wertzuweisung die Serveranwendung. Daraufhin sollten Sie in der Lage sein, sich diese Dateien über einen beliebigen Webbrowser anzeigen zu lassen. Die Eingabe in das Adressfeld könnte dabei beispielsweise folgendermaßen aussehen:

```
http://localhost:8080/index.html
```

15. Quellen und Referenzen

[1] IBM: When to use dynamic linking and static linking. http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.prftungd/doc/prftungd/when_dyn_linking_static_linking.htm (12.07.2011)

[2] Wang,Qi: A benchmark of three C++ open source callback/signal/slot libraries. http://www.kbasm.com/cpp-callback-benchmark.html (18.09.2011)

[3] Kalicinski, Marcin: How to Populate a Property Tree. http://www.boost.org/doc/libs/1_48_0/doc/html/boost_propertytree/parsers.html#boost_propertytree.parsers.info_parser (14.09.2011)

[4] Hess, Frank Mori: Tutorial – Boost 1.47.0. http://www.boost.org/doc/libs/1_47_0/doc/html/signals2/tutorial.html#id2863406 (20.09.2011)

[5] Kohlhoff, Christopher M.: Examples – Boost 1.47.0. http://www.boost.org/doc/libs/1_47_0/doc/html/boost_asio/examples.html (26.09.2011)

[6] Kohlhoff, Christopher M.: A combined TCP/UDP asynchronous server. http://www.systomath.com/include/Boost-1_36/doc/html/boost_asio/tutorial/tutdaytime7.html (05.08.2011)

[7] Diverse Autoren: Superserver. http://de.wikipedia.org/wiki/Superserver (15.07.2011)

[8] http://www.smartfoxserver.com (23.10.2011)

[9] Kaiser, Ulrich: *C/C++. Von den Grundlagen zur professionellen Programmierung.* 2. korrigierter Nachdruck. Bonn: Galileo Press, 2000

[10] Sayfan, Gigi: Building Your Own Plugin Framework. http://drdobbs.com/cpp/204202899 (30.07.2011)

[11] Diverse Autoren: Boost 1.48.0 Library Documentation. http://www.boost.org/doc/libs/1_48_0/ (25.11.2011)

16. Abbildungsverzeichnis

Abb. 1	Schematische Darstellung des Startprozesses	12
Abb. 2	Funktionsweise / Informationsfluss des Logging-Systems	29
Abb. 3	Funktionsweise NetStreamBuffer	38

17. Verwendete Hilfsmittel

- Linux Man-Pages
- Code::Blocks 10.05 IDE
- Enterprise Architect 7.5
- Angegebene Boost-Bibliotheken der Version 1.46.1

18. Selbstständigkeitserklärung

Hiermit versichere ich, Daniel Reichelt, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt wurden, sowie Zitate deutlich als solche kenntlich gemacht habe.

Naundorf, den 27.11.2011

Ort, Datum *Unterschrift*

Anhang

A. Verfügbare Events

	Bezeichnung	Erforderliche Syntax der Callback-Funktion	Auslöser und Funktionsparameter
Session	sig_packet **sig_ps**	void (**Session***, const **Packet***)	Ein Paket wird an eine Session gesendet. Referenziert die betroffene Session und das entsprechende Paket.
	sig_packet **sig_pr**		Ein Paket wurde von dieser Session vollständig empfangen. Referenziert die betroffene Session und das empfangene Paket.
	sig_session **sig_sc**	void (**Session***)	Die Verbindung zum Client wurde beendet. Referenziert die betroffene Session.
Room	sig_room **sig_sr_add**	void (**Room***, **Room***)	Einem Raum wurde ein untergeordneter Raum hinzugefügt (Kindraum). Referenziert den Elternraum und den Kindraum.
	sig_room **sig_sr_rem**		Einem Raum wurde ein untergeordneter Raum entfernt (Kindraum). Referenziert den Elternraum und den Kindraum.
	sig_session **sig_s_add**	void (**Room***, **Session***)	Einem Raum wurde eine Session hinzugefügt. Referenziert den betroffenen Raum und die hinzugefügte Session.
	sig_session **sig_s_rem**		Eine Session wurde aus einem Raum entfernt. Referenziert den betroffenen Raum und die entfernte Session.
	sig_packet **sig_p_send**	void (**Room***, const **Packet***)	Ein Paket wurde an alle Mitglieder des Raumes versandt. Referenziert den betroffenen Raum und das gesendete Paket.

	Bezeichnung	Erforderliche Syntax der Callback-Funktion	Auslöser und Funktionsparameter
Zone	sig_room **sig_sr_add**	void (**Zone***, **Room***)	Einer Zone wurde ein Raum untergeordnet. Referenziert die betroffene Zone und den hinzugefügten Raum.
	sig_room **sig_sr_rem**		Einer Zone wurde ein Raum entfernt. Referenziert die betroffene Zone und den entfernten Raum.
	sig_session **sig_s_add**	void (**Zone***, **Session***)	Einer Zone wird eine Session zugeordnet (neu angenommene Verbindung). Referenziert die betroffene Zone und die neue Session.
	sig_session **sig_s_rem**		Eine Session wurde aus einer Zone entfernt (manuell oder als Resultat eines Verbindungsabbruches). Referenziert die betroffene Zone und die entfernte Session.
	sig_packet **sig_p_send**	void (**Zone***, const **Packet***)	Ein Paket wurde an alle Mitglieder der Zone versendet. Referenziert die betroffene Zone und das versendete Paket.

B. Klassendiagramm HTTP-Addon