

ei-push

Datentransfer via UDP

Objektorientiertes Programmieren, TFH Berlin
15. Februar 2007

Torben Zech		738845
Martin Henning		736150
Abdurrahman Namdar		739068

Inhaltsverzeichnis

1	Vorraussetzungen	3
2	Das Protokoll	4
2.1	Der Header	4
2.2	Die Paketkennung	4
2.3	Kommunikationsablauf: Ein Beispiel	4
3	Dokumentation	6
3.1	Grundlagen	6
3.2	Paketbehandlung	7
3.3	Datei-/Datenoperationen	7
3.4	Protokollierung	8
4	Fazit	8
A	Quellen	9

1 Voraussetzungen

Aufgrund persönlicher Präferenzen ergab sich für unsere Gruppe als Grundvoraussetzung die cross-platform-Kompatibilität der zu erstellenden Software. Da es sich *nur* um ein Konsolenprogramm handeln sollte, wurde zum Anfang lediglich das bereits vorgegebene `winsoc`-Interface nach Apple OS X (BSD) und damit im Prinzip nach *NIX portiert.

Weiterhin fällt bei näherer Betrachtung des UDP-Protokolls auf, dass in Sachen Fehlervermeidung und Kommunikationsorganisation im Prinzip alles auf Softwarebasis selber zu implementieren ist. UDP kann lediglich durch Verwendung von Prüfsummen eine Unterscheidung zwischen korrekt und zwischen fehlerhaft übertragenen Paketen vornehmen. Zum Zwecke der Kommunikation zwischen Server und Client wurde infolgedessen ein Protokoll mit einer Anzahl an Steuer- und Datenpaketen entworfen, welches uns erlaubt, selbst im Fehlerfall (hoher Datenverlust auf der Übertragungsstrecke, Verbindungsabbruch) für eine reibungslose Funktion zu sorgen.

Die zu übertragenen Daten sollten unabhängig von ihrer Art mit unserem Programm kompatibel sein und in ihrer Dateigröße nicht übermäßig durch das verwendete Protokoll beschränkt werden. Letztendlich wird unsere Lösung heutzutage nach oben hin durch das zugrunde liegende Dateisystem beschränkt.

Zur Anpassung an verschiedene Situationen sollten die Programme in variablen Modi mit unterschiedlichen Timeout-Konfigurationen laufen können. Angedacht war die Wahl eines Modus beim Start des Servers und/oder Clients, worauf im Nachhinein aus Zeitgründen jedoch verzichtet wurde. Streckenweise wurde eine automatische Timeout-Anpassung aufgrund der laufenden Transaktionen implementiert, diese ist jedoch ausbaufähig.

2 Das Protokoll

2.1 Der Header

Die Kommunikationsgrundlagen bilden 256 Byte lange Datenpakete, welche einen 8 Byte breiten sog. *Header* besitzen. Der Header wiederum besteht aus 3 Byte Paketnummer, der Paketlänge in ebenfalls 3 Byte sowie eine Paketkennung von 2 Byte, welche später zur Erkennung unterschiedlicher Pakettypen benutzt werden kann. Zur vollen Ausnutzung des Headers wird dieser in unserer Implementierung hexadezimal (bzw. binär) belegt und sieht dann wie folgt aus:

- 0xFFFFFFFF - Paketnummer
- 0xFFFFFFFF - Paketlänge
- 0xFF - Paketkennung
- 0 bis 247 Byte - Nutzdaten

Wie im Weiteren noch erläutert, können wir mit dieser Umsetzung nach folgender Rechnung

$$\frac{0xFFFFFFFF \cdot 247\text{Byte}}{1024^3} = 3\text{GB} \quad (1)$$

Dateien von bis zu genau 3GB übertragen, was u.U. sogar die Grenze des Dateisystems sprengt. Die Paketlänge ist mit 3 Byte sicherlich völlig überdimensioniert, wurde aber in der Aufgabenstellung so vorgegeben. Hier hätten wir eher noch 3 Byte mehr an Nutzdaten untergebracht.

2.2 Die Paketkennung

Auch die Paketkennung braucht nicht unbedingt 2 Byte, jedoch haben wir uns diese Vorgabe zunutze gemacht: Da die Belegung hexadezimal erfolgte, sind die einzelnen Pakete ihrer Bestimmung nach in Gruppen geordnet worden, was sich bei einer hexadezimalen Darstellung recht gut machen läßt und noch dazu die Übersichtlichkeit im Programmcode stark erhöht (siehe Abb. 1):

2.3 Kommunikationsablauf: Ein Beispiel

Zur Erläuterung der Funktionsweise des Protokolls soll im weiteren Verlauf auf die sog. *Init Sequence* näher eingegangen werden. Hierbei handelt es sich um einen grundlegenden Paketaustausch zwischen Client und Server, welcher schließlich ein Zustandekommen einer Verbindung zur Folge hat.

0xFF	ACK	allgemeines OK
0x00	NACK	allgemeines Nicht-OK
0xA0	INIT_REQ	Anfrage zum Verbindungsaufbau
0xA1	FAIL	Ablehnung
0xA2	INIT_ACK	Bestätigung des Verbindungsaufbaus
0xB0	FILE_REQ	Anfrage nach einer Datei
0xB1	FILE_INFO	Infopaket über Vorhandensein und Größe
0xB2	FILE_ERR	Datei nicht vorhanden oder nicht lesbar
0xB3	FILE_INFO_OK	Dateiinfo erfolgreich empfangen
0xB4	SEG_ERR	Datensegment fehlt, kam nicht an, etc.
0xB5	SEG_ACK	Datensegment erfolgreich empfangen
0xB6	SEG_DATA	Datensegment

Abbildung 1: Pakettypenübersicht

Ist der Server gestartet, so erwartet er Anfragen von Client-Programmen. Ist ein Client verbunden, so werden andere Clients abgelehnt bis die Schnittstelle wieder frei ist. Eine solche Verbindungsanfrage wäre ein INIT_REQ-Paket, welches der Server wiederum mit INIT_ACK bestätigen würde. Nehmen wir an, dass der Client diese Bestätigung durch Probleme auf der Übertragungsstrecke nicht erhält: In diesem Falle würde er beispielsweise eine weitere Anfrage (INIT_REQ) an den Server schicken, welcher wiederholt eine Bestätigung aussenden würde. Erhält der Client diese schließlich, sendet er das Bestätigungspaket unverändert zurück. Ab diesem Zeitpunkt betrachten Server und Client eine Verbindung als zustande gekommen. Der Kommunikationsablauf wird jeweils in ein Logfile geschrieben. Auf dem Server (Abb. 2) werden die empfangenen Pakete auf dem Bildschirm ausgegeben:

```

tigabook — bash — bash — 84x17 — 983
-bash-2.05b$ ./ei-pushd
-----
nummer: 16777215 0xffffffff laenge: 19 0x000013 kennung: 160 0x00a0
Init REQ von Client 192.168.1.163:49550 erhalten.
Connection status: 0
Resend counter: 1
-----
nummer: 0 0x000000 laenge: 35 0x000023 kennung: 162 0x00a2
Init ACK von Client 192.168.1.163:49550 erhalten.
Connection status: 1
Resend counter: 0
-----
nummer: 16777215 0xffffffff laenge: 9 0x000009 kennung: 176 0x00b0
Connection status: 1
Resend counter: 0
^C
-bash-2.05b$

```

Abbildung 2: Initialisierungssequenz aus Serversicht

3 Dokumentation

Im Folgenden soll nicht auf alle Funktionen eingegangen werden. Vorgestellt werden lediglich die wichtigsten Funktionsgruppen, welche in funktionalen Einheiten zusammengefaßt werden konnten. Anderes bitten wir im Quelltext (siehe Anhang) nachzusehen.

3.1 Grundlagen

- `class STrans`
- `struct Packet`
- `class MyTimer`

Für jede Transaktion wird ein Objekt vom Typ `STrans` angelegt, welches zur Laufzeit einer Transaktion alle wichtigen Funktionen zur Verfügung stellt. Hier werden Informationen über Verbindungsstatus, Dateisegmente und Sicherheitsrelevante Zähler zur Ablaufsteuerung hinterlegt. Ist eine Transaktion zu Ende oder tritt ein eklatanter Fehler auf, wird das gesamte Objekt gelöscht und mit einer Standardkonfiguration neu angelegt. Dies hat zur Folge, dass für den Clienten wieder eine vollständige Initialisierungssequenz nötig wird, um eine neue Transaktion durchführen zu können. Muß der Server über ein einstellbares Limit hinaus Paketsendungen wiederholen (`int resend_counter;`), so führt auch dies zu einem Verbindungsabbruch.

Paketspezifische Informationen werden jeweils in einem dafür angelegten Objekt vom Datentyp `Packet` abgelegt. Hier finden sich alle relevanten Informationen, welche von den Funktionen zur Erstellung und Zerlegung von Paketen gelesen bzw. geschrieben werden.

Mit der `MyTimer`-Klasse haben wir ein Werkzeug zur Erzeugung beliebiger Timer für unterschiedliche Anwendungen. Wird ein Timer-Objekt erzeugt, speichert es automatisch den Zeitpunkt seiner Erzeugung. Wird später die Funktion `reset();` aufgerufen, so wird die seit der Erzeugung vergangene Zeit zurückgegeben und der Timer automatisch zurückgesetzt. Mithilfe eines solchen Timer-Objektes werden beispielsweise keep-alive-Funktionen gesteuert.

3.2 Paketbehandlung

- `dissect_buffer()`
- `make_packet()`

Diese Funktionen bilden die Basis der Behandlung jeglicher Kommunikations- und Datenpakete. Ihr Ein- und Ausgaben sind mit dem o.g. Packet-struct verknüpft, sodaß hier eine einheitliche Schnittstelle geschaffen wurde. `dissect_buffer()`; kann ein beliebiges Paket in seine Bestandteile zerlegen und gleichzeitig die Struktur auf Fehler überprüfen. Falsch zusammengesetzte Pakete, zu lange oder zu kurze Header sowie falsch berechnete Paketlängen werden hier erkannt und können aussortiert werden. Der Header wird hierbei byteweise Hexadezimal umgesetzt.

Die Funktion `make_packet()`; stellt die entgegengesetzte Funktion zur Verfügung. Sie erstellt aus Paketnummer, Paketkennung und Nutzdaten ein strukturell gültiges Paket. Hierbei wird er Header wiederum aus hexadezimalen Werten zusammengesetzt, die Paketlänge dabei automatisch berechnet.

Wichtigste Eigenschaft beider Funktionen: Es werden keinerlei **String**-Funktionen benutzt. Sämtliche Operationen werden mit `unsigned char` abgearbeitet, was zu einem erheblichen Programmieraufwand führte. Der Grund: Das Ergebnis ist nicht anfällig für NullBytes und kann somit für Pakete jeglichen Inhalts verwendet werden. Dies ist wichtig wenn man Binärdaten austauschen will.

3.3 Datei-/Datenoperationen

- `write_to_file()`
- `get_file_segment()`
- `get_file_segment_count()`
- `get_file_size()`

Ein wichtiger Bestandteil, sind die Funktionen zur Behandlung von Datei- und Datenoperationen. Es existieren Funktionen zur Überprüfung auf Vorhandensein bzw. Lesbarkeit einer Datei, sowie zum Abrufen von Dateiinformationen wie Größe in Byte und die Anzahl der vorhandenen 247-Byte-Segmente. Es können beliebige Segmente von beliebigen Stellen einer Datei gelesen und geschrieben werden, wobei die Größe eventuell nicht voller 247 Byte-Pakete automatisch berechnet wird. Auch hier wird komplett binär eingelesen, sodaß der Inhalt einer Datei von der jeweilig angewendeten Funktion nicht beeinträchtigt oder gar verändert wird. In Testläufen wurde die Kopiergeschwindigkeit der Funktionen nicht durch sie selbst, sondern durch äußere Umstände wie zum Beispiel die Netzwerkgeschwindigkeit beschränkt.

3.4 Protokollierung

- `log_this()`

Sämtliche Kommunikations- und Datenpakete werden von Server sowie Client in eine Logdatei geschrieben. Hier kann die Funktionsweise des Protokolls nachvollzogen werden:

```
1 14.2.2007 - 0:21 on 127.0.0.1:6881 - SRV: Socket created. Waiting
   for connections.
2 14.2.2007 - 0:21 on 127.0.0.1:0 - Starting Client.
3 14.2.2007 - 0:21 on 127.0.0.1:0 - CLT: Connection request by user.
4 14.2.2007 - 0:21 on 127.0.0.1:0 - CLT: Socket created.
5 14.2.2007 - 0:21 on 127.0.0.1:6881 - CLT: INIT REQ sent to server.
6 14.2.2007 - 0:21 on 127.0.0.1:52440 - SRV: INIT REQ from client
   received.
7 14.2.2007 - 0:21 on 127.0.0.1:6881 - CLT: INIT ACK from server.
8 14.2.2007 - 0:21 on 127.0.0.1:0 - CLT: INIT ACK resent to server.
9 14.2.2007 - 0:21 on 127.0.0.1:0 - CLT: CONNECT SUCCESSFUL.
10 14.2.2007 - 0:21 on 127.0.0.1:52440 - SRV: INIT ACK from client
    received.
11 14.2.2007 - 0:21 on 127.0.0.1:0 - CLT: Received FILE REQ from user
    .
```

4 Fazit

Das Projekt erschien vor dem Hintergrund der Vorlesung relativ simpel, weshalb versucht wurde - durch die Implementierung verschiedener Funktionen - selbiges wenigstens in die Nähe der Realität zu rücken. Angesichts der erdrückenden Anforderungen der Realität, haben wir sehr schnell gemerkt, wie schwer es ist den eigenen Anspruch zu erfüllen. Sicherheitsrelevante Funktionen wie Timeouts, Flood-Protection und ein wasserdichtes Protokoll machen einfach viel zu viel Arbeit. Das Ganze auf zwei Plattformen parallel zu entwickeln hat da vergleichsweise gut geklappt. Rückschläge, meist zeitlicher Natur, mußten wir hier eigentlich nur auf dem Gebiet der Oberflächenprogrammierung hinnehmen, da die Einarbeit in zwei grundlegend unterschiedliche Systeme (`ncurses.h/conio.h`) schlichtweg nicht möglich war. Letztendlich verbrauchten kompliziertere Funktionen aus den Weiten des C++-Universums zusammen mit wilden *type casts* doch zu viel Zeit für Erstsemester-Zeh-plus-plus'ler. Alles in allem: Viel gelernt, wenig geschlafen, nichts gegessen.

A Quellen