Christoph-Scheiner-Gymnasium Ingolstadt

Seminararbeit

aus dem wissenschaftspropädeutischen Seminar

„Programmieren von Android-Apps“

im Fach Informatik

Ein Lernmittelbibliothekssystem mit Fokus auf klar strukturierten Kommunikationsprotokollen

angefertigt von	Dominik Okwieka
Reifeprüfungsjahrgang	2017
Kursleiter	StD Pabst

Ingolstadt, den 07. November 2016

Inhaltsverzeichnis

1. Einleitung

> **[Rule] 17. Omit needless words.**
>
> Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should not have unnecessary lines and a machine no unnecessary parts. This requires not that the writer make all sentences short, or avoid all detail and treat subjects only in outline, but that every word tell.[1]

libmangler ist ein Verwaltungssystem für Lernmittelbüchereien. Es besteht aus einem Server und einem Android-Client, die mithilfe eines einfachen Protokolls Daten austauschen. Dieses Protokoll ist jedoch trotz der Einfachheit generell und ist vergleichbar mit einer Datenbanksprache wie SQL,[2] jedoch zugeschnitten auf die Anwendung.

Jedes Buchexemplar hat eine einzigartige Identifikationsnummer, welche als QR-Code auf diesem befestigt wird. Die App kann diesen Code auslesen; der Nutzer kann dann das Medium entleihen, zurückgeben, mit Notizen versehen, kategorisieren, aus dem Verkehr ziehen oder ganz löschen. Gedacht ist die App für die Leiter der Lernmittelbücherei, für die Lehrer, die Beschädigungen notieren müssen, sowie für alle, die bei der Buchausgabe und der Rücknahme beschäftigt sind.

Es gibt eine klare Trennung der Begriffe (Buch-)Exemplar (*Copy*) und Buch: ersteres ist ein physikalisches Medium, letzteres bezieht sich auf eine Ansammlung von Medien mit derselben ISBN. Zum Beispiel gibt es ein Buch *Mathematik 8. Klasse*, aber 200 Exemplare, *Copies*, davon.

2. Überblick über das Projekt

Wie anfangs erwähnt, besteht *libmangler* aus einem Server und einem Client. Der Server ist in Go[3] geschrieben und speichert alle Bücher, Copies und User (Ausleiher, also die Schüler) in drei Textdateien ab, in einem Lisp-ähnlichen Format, *S-Expressions*, das intuitiv verständlich ist.

1 Strunk Jr. und White, *The Elements of Style*
2 Chamberlin und Boyce, „SEQUEL (Structured English Query Language)“
3 Griesemer, Pike, und Thompson, „The Go programming language“

Der Client ist vergleichbar mit einem Fenster in die Daten des Servers: er scannt einen QR-Code oder lässt den Nutzer eine Suchanfrage eintippen, fragt den Server nach dem Gesuchten, speichert nur dieses und erlaubt dann einige Aktionen bezüglich dieser Daten. Trotz einfacher Anforderungen stellte sich der Client als schwieriger heraus als der Server, da blockierende Netzwerkkommunikation in Android nicht erwünscht ist; am Ende wurde die Komplexität jedoch ersetzt mit einer blockierenden und funktionierenden Lösung, wenngleich das nicht zu den *best practices* gehört.

Bevor wir zu einer genaueren Beschreibung der Komponenten kommen können, muss das Protokoll verstanden sein. Seine Struktur ist die Struktur des Servers; seine Struktur prägt auch den Client.

3. Gesichtspunkte von Protokollen

Computer sind geprägt von formalen Sprachen: die meisten Programme erwarten ihre Eingabedaten und ihre Konfiguration in einem bestimmten Format und geben Daten mit einer bestimmten Struktur aus. Wenn man formale Sprachen mit menschlichen Sprachen ersetzt und Dateien mit Büchern vergleicht, dann sind Protokolle nichts anderes als Dialoge. Die einzelnen Sätze folgen einer Grammatik, doch das ist nicht alles: man muss verhindern, dass Missverständnisse entstehen, indem aneinander vorbeigeredet wird. Man muss darauf achten, dass die Botschaft unverändert ankommt, dass sie *sicher* ankommt, ohne überhört worden zu sein. Es kann viel schiefgehen.

In dieser Arbeit wird hauptsächlich von Anwendungsprotokollen die Rede sein, also Protokollen der siebten Schicht des OSI-Modells. IP, TCP, ARP, usw., sind natürlich auch Protokolle, lassen sich aber schwer mit Anwendungsprotokollen vergleichen.

3.1 Anforderungen

Nichts ist wichtiger als die Funktionsfähigkeit des Protokolls: es muss eine sinnvolle Kommunikation zwischen Hosts erlauben. Dazu gehört natürlich, dass nichts in falscher Reihenfolge, unvollständig, korrumpiert, oder am falschen Ziel ankommt. Deshalb verwenden die meisten Protokolle TCP[4] als Unterbau, dass all diese Dinge garantieren kann. Böswillige

4 Postel, „Transmission Control Protocol“

Betrachtung und Manipulation der Daten kann man z.B. durch SSL/TLS verhindern, das leicht integrierbar ist. Protokolle, die TCP verwenden, sind stream-basiert, das heißt, es scheint für sie eine bidirektionale Verbindung der Hosts zu bestehen. Solch ein Stream wird durch den sogenannten Three-Way-Handshake aufgebaut: der Client sendet ein SYN-Paket, der Server antwortet mit SYN-ACK, der Client antwortet darauf mit einem ACK.

Doch nicht immer laufen Protokolle über TCP. Prominentes Beispiel ist das Domain Name System (DNS), das primär der Auflösung von Hostnamen in IP-Adressen dient. Das DNS-Protokoll verwendet UDP (User Datagram Protocol), die verbindungslose Alternative zu TCP. Bei UDP werden einzelne Pakete übertragen, von denen man nicht weiß, ob und in welcher Reihenfolge sie ankommen. Die Pakete werden auch *Datagramme* genannt, daher der Protokollname. DNS verwendet UDP, um die Kosten des Three-Way-Handshake zu vermeiden; zudem muss der DNS-Server sich nicht um offene Verbindungen sorgen, weil es keine gibt.

Viele Protokolle haben also Performanceanforderungen. Es gibt hier zwei Größen: Bandbreite und Latenzzeit. Bandbreite ist die Datenmenge pro Zeiteinheit, die über das Protokoll übertragen wird; Latenzzeit ist die Zeit, bis die Antwort des entfernten Hosts beim Anfrager eintrifft (*Round Trip Time*, RTT, „Ping“). Je nachdem, was die Anwendung ist, ist das eine wichtiger als das andere. Wer Videos übertragen will, achtet auf Bandbreite. Wer ein Echtzeitmultiplayerspiel hat, den interessieren möglichst geringe Latenzzeiten.

Die folgenden Attribute sind jedoch am wichtigsten: Robustheit, Testbarkeit, Verständlichkeit und Portabilität. Ohne Robustheit und Portabilität kann ein Protokoll im heterogenen Internet nicht überleben: es gibt meist mehrere, immer leicht falsche Implementierungen, die auf einer Vielzahl unterschiedlicher Architekturen und einer Vielzahl unterschiedlicher Betriebssysteme laufen. Ohne Testbarkeit ist es unmöglich, genau diese Robustheit zu testen. Ohne Verständnis für das Protokoll tappt der Entwickler im Dunkeln herum. Dokumentation ist der Mörtel, der diese Tugenden zusammenhält.

Um das Protokoll korrekt implementieren zu können, muss es einfach sein, denn einfache Protokolle führen zu einfachen Implementierungen. Einfacher Code lässt sich vollständiger testen, ist daher wartbar und zumeist portierbar.

3.2 Ansätze

Es ist Zeit, mehrere Ansätze zu benennen; jeder hat bestimmte Vor- und Nachteile. Die folgende Liste enthält verschiedenartige Protokolle, manche mehr und manche weniger bekannt. Die ersten zwei Beispiele werden binär codiert, der Rest ist textbasiert.

3.2.1 9P

9P[5] ist das Dateisystemprotokoll des Betriebssystems *Plan 9 from Bell Labs*.[6] Plan 9 wurde entwickelt, um das Unix-Prinzip „*Everything is a file*“ weiterzutreiben: alles – Geräte, Mailboxen, das Netzwerksystem, das Grafiksystem und viel mehr – wird durch *Dateisysteme* repräsentiert, deren Daten zumeist on-the-fly generiert werden (vgl. `/proc`[7]). Jeder Prozess hat einen eigenen sogenannten *Namespace*, der die Ansammlung aller von diesem Prozess gemounteten Dateisysteme ist. Der Zugriff auf diese findet über 9P statt; zur Implementierung eines eigenen Dateisystems muss man nur einen 9P-Server schreiben, was durch Hilfsfunktionen sehr einfach ist.[8] Die 9P-Verbindung wird zumeist über TCP getunnelt, wenn der Server nicht lokal ist. Man kann das `/proc`-Verzeichnis eines anderen Systems mounten und dann die dortigen Prozesse debuggen. Man kann den Bildschirm, die Maus und die Tastatur eines anderen Systems mounten und diesen dann als Terminal verwenden. Man kann die Zwischenablage eines anderen Systems mounten und so auslesen oder modifizieren.

Das Protokoll kümmert sich um das Navigieren im Verzeichnisbaum sowie dem Erstellen, Öffnen, Lesen, Schreiben und Löschen von Dateien. Die Belastung des Protokolls ist vielseitig: manchmal werden wenige, große Pakete versendet, so z.B. beim Lesen großer Dateien von einer Festplatte. Meist jedoch werden viele kleine Pakete versendet, da kurze Strings in die Kontrolldateien von Geräten geschrieben werden. In diesem Fall kann die Größe der Paket-Header Überhand nehmen, jedoch tragen diese nur das Nötigste an Information.

9P verwendet binär kodierte Header. Clients vergeben an ihre offenen Dateien in der 9P-Session eindeutige Ganzzahlen, die *Fids* genannt werden; 9P ist also zustandsbasiert. Der Client beginnt jede Transaktion mit einer T-Message (*T* steht für *transmit*), der Server antwortet mit einer R-Message (*R* steht für *reply*). Jede T-Message erhält vom Client einen

5 „intro(5) of the Plan 9 man pages“
6 Pike u. a., „Plan 9 from Bell Labs“
7 Killian, „Processes as files“
8 „9p(2) of the Plan 9 man pages“

eindeutigen *Tag*; die Antwort des Servers hat denselben. *Tags* finden sich auch in IMAP und in der früheren Version 5 des *libmangler*-Protokolls. Fehler werden gemeldet, indem ein spezielles Paket, `Rerror`, gesendet wird; dieses enthält einen String, der den Fehler beschreibt.

3.2.2 NTP – Network Time Protocol

Wenngleich moderne Computer zumeist eine batteriebetriebene Echtzeituhr besitzen, muss diese mit genaueren Uhren synchronisiert werden, damit sie korrekt bleibt.[9,10] Schon 1985 hatte das Network Time Protocol eine Referenzimplementierung und wurde in RFC 958[11] dokumentiert. In weiterentwickelter Form wird das Protokoll in fast allen internetfähigen Systemen verwendet.

Die NTP-Hierarchie ist in sogenannte Strata eingeteilt: Stratum 1 bezeichnet die an genauen Zeitgebern angeschlossenen Computer (primäre Zeitserver). Generell greifen Stratum *n*-Rechner jeweils auf Stratum (*n-1*)-Rechner zu und gleichen sich zudem untereinander ab. Das System versucht, einen möglichst minimalen Baum an Verbindungen aufzubauen, um die Latenzzeiten zu Stratum 1 gering zu halten. Der restliche Fehler wird durch eine auf Statistiken basierenden Formel entfernt.

Je nach Anwendung kommt eine der drei Betriebsmodi zum Einsatz: Client/Server, bei dem der Client vom Server pullt; der symmetrische Modus, bei dem sich zwei *Peers* gegenseitig synchronisieren; Broadcast, bei dem der Server an mehrere Clients Pakete sendet. Mit jedem Paket wird ein *Packet Mode*-Wert übertragen, der den Modus identifiziert. Es gibt drei Zeitformate: *Short*, *Timestamp* und *Date*. Wenn möglich, wird das Datumsformat verwendet,[12] das aus einer *Era Number*, einem in Sekunden gemessenen *Era Offset* und einem Bruch besteht. Die *Era Number* bezeichnet den Bereich, in dem der 32-bit Offset nicht überläuft. Momentan sind wir in Era 0; ab dem 08. Februar 2036 werden wir in Era 1 sein. Das im Protokoll verwendete *Timestamp*-Format hat einen 32-bit Sekundenzähler und einen Bruch; das *Short*-Format ist ähnlich, hat aber nur 16 Bit Präzision. Diese Brüche sind wie folgt zu verstehen: im *Short*-Format, das insgesamt 32 Bit groß ist, bilden die ersten 16 Bit den

9 „How accurate is the CMOS clock?“
10 Daviel, „Web time“
11 Mills, „Network Time Protocol (NTP)“
12 Mills u. a., „Network Time Protocol Version 4: Protocol and Algorithms Specification“, Sektion 6

ganzzahligen Teil, die anderen 16 Bit den Bruchteil einer Festkommazahl, dessen Komma nach dem sechzehnten Bit zu finden ist.

TCP kann hier nicht verwendet werden, weil es verlorene Pakete wieder überträgt und dadurch die Zeitstempel in diesen verfälscht,[13] deswegen wird UDP auf Port 123 verwendet. NTP verwendet konventionelle binäre Pakete mit einem Header und einem aus vier Timestamps bestehenden Payload. Ein invalider Wert im Header, Stratum 0, initiiert ein sogenanntes *Kiss-o'-Death*-Paket, mit welchem Kontrollcodes übertragen werden; diese sind Vier-Zeichen-ASCII-Strings an der Stelle, an der sonst die Referenz-ID des Zeitgebers steht (z.B. `GPS`).

3.2.3 HTTP – Hypertext Transfer Protocol

Das Web ist die *Killer Application* des Internets, so wie die Glühbirne die *Killer Application* der Elektrizität war. Viele Laien können heute die Begriffe „Web“ und „Internet“ nicht mehr auseinanderhalten. Das Hypertext Transfer Protocol ist so erfolgreich, dass es als Transportprotokoll für alles gebraucht wird, obwohl es nicht auf generelle Kommunikation ausgerichtet ist.

HTTP hat für Internetstandards typische Merkmale: es verwendet Textbefehle, hat dreistellige Statuscodes mit einem angefügten, menschenverständlichen Text (z.B. `404 File Not Found`) und hat ein Headerformat, bei dem jedes Feld die Form 'Feldname: Wert' hat.

Gemeinhin haben Clients die Initiative und senden hauptsächlich `GET`- und `POST`-Requests. Eine HTTP-Verbindung hat keinen Zustand. Der Server beantwortet den Request und vergisst dann den Client. In HTTP/1.0 wurde für jeden Request einen neue Verbindung geöffnet. Persistente Verbindungen, die der Normalzustand seit HTTP/1.1 sind, erlauben einen Request nach dem anderen in der Verbindung; die Latenzzeit durch das Verbindungsöffnen entfällt. *Pipelining*, d.h. das Senden mehrerer Requests auf einmal und das Empfangen der Antworten auf einmal, optimiert den Prozess weiter.

Im Kern ist HTTP ideal für die Aufgabe, nicht interaktive Webseiten und andere Dateien zu übertragen. Ein Client gibt den Dateipfad auf dem Server an, der Server sendet die Datei. Kürzer kann man diese Interaktion nicht gestalten; problematisch wird es, wenn eine Seite

13 ebd, Sektion 1

viele Assets von vielen Servern, z.B. von Tracking- und Ad-Servern einbindet, denn persistente Verbindungen helfen auch hier nicht. Newsseiten sind häufige Übeltäter.

HTTP ist nicht für bidirektionale Kommunikation gedacht, da die Zustandslosigkeit im Weg steht. Cookies sind ein Hack, um diese zu umgehen, und niemand mag Cookies. Ein anderer Weg sind Parameter in der URL, die den ganzen Zustand übertragen und leicht zu manipulieren sind; mitunter wird grob fahrlässig ein verschlüsseltes Passwort übertragen (ich denke an eine bestimmte Webseite, die ich jedoch nicht nennen werde).

Ein neuer binärer Standard, HTTP/2,[14] wurde inzwischen veröffentlicht und wird von allen weit verwendeten Browsern unterstützt. Neben mehreren anderen Änderungen kann der Server nun Dateien pushen, für die der Client wahrscheinlich sowieso eine Anfrage gestellt hätte; z.B. würden beim Aufruf einer Seite gleich die CSS-Dateien und etwaiger Javascript-Code neben dem HTML-Text gesendet. Anfang November 2016 verwendeten 10.4% der 10 Millionen meistbesuchten Websites HTTP/2.[15]

3.2.4 IMAP – Internet Message Access Protocol

Mailboxen lassen sich mit dem *Post Office Protocol* (POP),[16] dem *Internet Message Access Protocol* (IMAP)[17] oder via einem Webmail-Interface im Browser verwalten, falls man nicht selbst Admin eines Mailservers ist. Bei POP ist es Konvention, die Nachrichten auf dem Server nach dem Abrufen zu löschen; die Mails residieren auf dem Client, wie auch der Verzeichnisbaum mit dem Posteingang, dem Postausgang und nutzererzeugten Ordnern.

IMAP ist eine neuere Entwicklung, um seine Nachrichtenordner auf dem Server zu verwalten; dadurch kann man von mehreren Geräten auf denselben Baum zugreifen. Der Client cacht die Mails nur; der Server hat die relevante Kopie. Verständlicherweise ist IMAP komplexer als POP3.

Ich will IMAP deswegen ansprechen, weil es *Tags* verwendet, wie es auch das *libmangler*-Protokoll zeitweise getan hat, und weil der Server von sich aus senden kann. Das folgende

14 Belshe, Peon, und Thomson, „Hypertext Transfer Protocol Version 2 (HTTP/2)“
15 Surveys, „Usage of HTTP/2 for websites“
16 Myers und Rose, „Post Office Protocol - Version 3“
17 Crispin, „INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1“

Exzerpt aus RFC 3501[18] soll das nun verdeutlichen. Zeilen mit einem `*` werden vom Server in Eigeninitiative gesendet (Zeile 1), oder deuten die Kontinuation des Outputs an. Die vom Client generierten alphanumerischen Tags, hier `a001` und `a002`, müssen eindeutig sein; Anfrage und Antwort haben denselben Tag. Bei Antworten folgt dann `OK` (Erfolg), `NO` (Fehlschlag), oder `BAD` (formaler Fehler).

Zudem verwendet IMAP wie *libmangler* S-Expressions, eine aus der Lisp-Welt stammende Notation für beliebig verschachtelte Listen, aus denen Lisp-Code und -Daten bestehen. Ein Beispiel aus einer Lisp-Sprache: `(define square (lambda (x) (* x x)))`. Das folgende Listing enthält auch eine S-Expression. Wo ist sie?

```
S:   * OK IMAP4rev1 Service Ready
C:   a001 login mrc secret
S:   a001 OK LOGIN completed
C:   a002 select inbox
S:   * 18 EXISTS
S:   * FLAGS (\Answered \Flagged \Deleted \Seen \Draft)
S:   * 2 RECENT
S:   * OK [UNSEEN 17] Message 17 is the first unseen message
S:   * OK [UIDVALIDITY 3857529045] UIDs valid
S:   a002 OK [READ-WRITE] SELECT completed

[...]
```

3.2.5 mpmp

mpmp ist kein Internetstandard. Es ist ein noch unfertiger Monopoly-Klon im Stil der Weimarer Republik, bei dem man über ein Netzwerk spielen kann. Entstanden als Informatikprojekt der elften Klasse, das einige aus dem Seminar erstellt haben, wird es nun von mir instandgehalten. Das Protokoll ist meine Schöpfung, weswegen ich über die speziellen Entscheidungen schreiben will, die in das Protokoll einflossen.

Der Server enthält den Spielzustand; die Clients cachen diesen, stellen ihn dar und senden Befehle an den Server, der Änderungen des Spielzustands allen Clients mitteilt. Client und Server werden aus demselben Code kompiliert und verwenden ein völlig symmetrisches Protokoll. Sowohl Clients als auch Server senden Befehle mit Argumenten aus und quittieren diese jeweils mit `+JAWOHL` oder `-NEIN`, gefolgt von einem Fehlerstring. Die Befehle, die

18 ebd, Sektion 8

allen Clients übermittelt werden, enden per Konvention in `-update`. Das Beispiel zeigt auch, dass einige der `+JAWOHL`s noch fehlen. Außerdem sieht man den einzigartigen `clientlist-update`-Befehl, der mehrere Zeilen Payload hat, deren Anzahl das erste Argument nennt, hier `1`. Durch einen Mitschnitt des Protokolls ab dem Beginn kann man den gesamten Verlauf des Spiels nachvollziehen.

```
S: +JAWOHL Willkommen, Genosse! Subscriben Sie!
C: subscribe player #0f0f0f oki
S: playerlist-update 1
S: #0F0F0F: Player: oki
C: +JAWOHL
C: chat Dies ist Chat!
S: chat-update (oki) Dies ist Chat!
C: +JAWOHL
C: start-game
S: pos-update 12 oki
S: turn-update 12 1 oki
S: start-update
C: +JAWOHL
C: end-turn
S: pos-update 21 oki
S: turn-update 9 0 oki
C: +JAWOHL
C: buy-plot 21 oki
S: money-update 25600 oki
S: show-transaction -4400 derp
S: plot-update 21 0 nohypothec oki
C: ragequit
S: clientlist-update 1
S: #0F0F0F: Spectator: oki
C: +JAWOHL
```

4. Das Protokoll

Das *libmangler*-Protokoll dient dem Zugriff auf Ansammlungen von Büchern, Copies, und Nutzern, also einer spezialisierten Datenbank. Insofern lässt es sich mit SQL vergleichen, ist jedoch weit simpler und nicht relational. Die Datenmengen, die verwaltet werden, sind gering, also ist die Bandbreitennutzung nie der Fokus gewesen. Vielmehr sollte das Protokoll auf möglichst simple und verständliche Weise möglichst generelle Mengen selektieren und auf diesen agieren können.

Das Protokoll bestht aus einem Low-Level-Teil, der sich mit dem Übertragen der eigentlichen Informationen beschäftigt, sowie der *kleinen Sprache*, in der die Anfragen gestellt werden. Um diese soll es vordergründig gehen. Dafür ist jedoch ein kleiner Exkurs vonnöten.

4.1 Die Anfragensprache

Die Anfragensprache ist von den Kommandosprache des Unix-Editors `sam`[19] inspiriert, der eine Weiterentwicklung von `ed` ist. Befehle sind einzelne Buchstaben. Die aktuelle Selektion, welche in `ed` zeilenweise und in `sam` zeichenweise Granularität hat, wird in einem Zwischenspeicher namens *Dot* gespeichert, der mithilfe eines Punktes (`.`) dargestellt wird. Befehle arbeiten entweder mit dem Inhalt von Dot oder setzen Dot zu einer neuen Selektion. In `sam` kann man auch mit der Maus Text selektieren und so *Dot* setzen.

```
x/^ /d
```

Diese `sam`-Schleife führt den `d` (*delete*)-Befehl für jedes Vorkommen des regulären Ausdrucks „^ “ in der Selektion aus; dieser Befehl entfernt also ein Einrückungslevel. Dot ist zu Beginn der Operation die gesamte bisherige Selektion; dann wird Dot zu den jeweiligen Vorkommnissen des Ausdrucks gesetzt. Hier ist Dot am Ende leer, weil der Löschbefehl Dot löscht.

Kommen wir nun zu libmanglers Befehlssprache und beginnen mit drei Beispielen.

```
B/978-0-201-07981-4/p

C/Hans, Max Mustermann/r
d

U/0, 405, 3050, /p
```

libmangler verwendet folgendes Schema zur Selektion: Großbuchstaben selektieren ganze Mengen, welche durch die Kriterien zwischen den Schrägstrichen eingeschränkt werden. Alles zwischen den Slashes wird als *Selektionsargument* bezeichnet. Es können mehrere Teilargumente mit Komma getrennt angegeben werden; ein Element gilt als selektiert, wenn es eines der Teilargumente erfüllt. Zur einfacheren automatischen Generation kann ein Komma nach dem letzten Argument stehen (siehe Beispiel 3).

19 Pike, „The text editor sam“

Eine Menge besteht aus Elementen vom selben Typ: Bücher, Copies oder User. Argumente sind ISBNs, Usernamen, IDs von Copies sowie Tags. Das erste Beispiel selektiert das eine Buch mit dieser ISBN und gibt alle Informationen darüber aus. Im zweiten Beispiel werden alle Copies selektiert, die die User `Hans` und `Max Mustermann` ausgeliehen haben; diese werden zurückgegeben (`r`) und dann ganz aus dem System gelöscht (`d`), weil Hans und Max eine Bücherverbrennung veranstaltet haben. Das dritte Beispiel selektiert die Ausleiher der Copies mit den IDs `0`, `405` und `3050` und gibt alle Informationen zu ihnen aus. Diese zwei Beispiele zeigen, dass die Selektionsargumente kontextgemäß interpretiert werden. Es wird immer das selektiert, was man erwartet. Aufgrund der internen Implementierung kommt es in Listen momentan zu Duplikaten.

Elementtyp	**ISBN**	**Copy-ID**	**Username**
Bücher	Buch mit dieser ISBN	Buch der Copy mit dieser ID	Bücher, von denen der User eine Copy entliehen hat
Copies	alle Copies dieses Buchs	Copy mit dieser ID	Copies, die der User entliehen hat
User	alle Entleiher einer Copy dieses Buchs	Entleiher dieser Copy	User mit diesem Namen

Dokumentiert ist die Sprache in der Spezifikation (`SPEC.md`); zum Testen kann man einfach einen Server starten und eine Verbindung mit `netcat`[20] aufbauen. So konnte ich schnell die Funktionalität testen; automatisiertes Testen kann über unkomplizierte Skripte und Testdateien von außen angebaut werden.

4.2 Outputformat des print-Befehls

Wie bereits in Sektion 3.2.4 erwähnt, verwendet *libmangler* zur Speicherung der Daten eines Elements sowie zur Serialisierung und Übertragung über das Netzwerk eine S-Expression. Hier eine vollständige Protokolltransaktion, die das illustrieren soll.

Anfrage des Clients:

```
C/594/p
```

20 *hobbit*, „Netcat“

Antwort des Servers:

```
(copy 594
    (user "Dominik Okwieka")
    (book 978-0-201-07981-4
        (authors "Alfred V. Aho" "Brian W. Kernighan" "Peter J.
Weinberger")
        (title "The AWK Programming Language")
    )
    (notes
        "2016-03-24T11:01+01:00 <- ISO 8601-Date"
    )
    (tags #derp #foo)
)
---
```

Man beachte die drei Bindestriche in der letzten Zeile; darauf komme ich später zurück. Fokussieren wir uns jedoch kurz auf den Ausdruck davor. Diese S-Expression besteht aus einem Atom `copy`, einer Zahl `594` sowie mehreren Unter-S-Expressions. Der Vorteil dieses alten Formats[21] ist der Verzicht auf kompliziertere Syntaxelemente. Dadurch ist das Parsen einfach (*libmanglers* Parser in `sexps/sexps.go` besteht aus zwei Funktionen plus dem Lexer). Der Mensch kann das Format zudem sofort begreifen, wenngleich in komplexeren Ausdrücken viele öffnende und schließende Klammern hintereinander auftreten können, was Lisp, das aus S-Expressions besteht, den Titel *Lots of Irritating Stupid Parentheses* einbrachte.

4.3 Low-level-Teil des Protokolls

Viel hat sich im „niedrigen“ Teil des Protokolls verändert, bis es zu einer adäquaten Lösung kam. Es gibt zwei Probleme: die Antworten müssen den Anfragen zugeordnet werden und die Größen mehrzeiliger Antworten müssen bekanntgemacht werden.

Die Zuordnung ist in einem zustandsbasierten synchronen Protokoll ein Nonproblem. Ein solches Protokoll war ursprünglich vorgesehen und ist am optimalsten für die Anwendung geeignet, da es insbesondere auf Serverseite sehr einfach umzusetzen ist und logisch auch mehr Sinn ergibt. Während man auf die Informationen wartet, die vom Server geholt werden, kann der App-Nutzer nichts tun. Da Android verständlicherweise Netzwerkverbindungen im UI-Thread verhindern will, weil diese potentiell lange dauern, ist es schwer, ein synchrones Protokoll zu implementieren. Es gab in Protokollversion 5 folgenden Ansatz: Vom Client frei

21 McCarthy, „Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I“

wählbare *Tags* wie in 9P und IMAP werden vor jedem Request angefügt. Die Serverantwort enthält denselben Tag. Der Client sollte beim Empfang einer Antwort die im Voraus für diesen Tag bestimmte Handlerfunktion ausführen. Da sich dies massiv auf die Komplexität der App auswirkte und obendrein nie funktionsfähig war, ignorierte der Autor Androids Warnung, nicht im UI-Thread zu netzwerken, und vereinfachte den Client wieder. Jetzt funktioniert er, und da der Socket einen Timeout von drei Sekunden bekommen hat, gibt es keine zu großen Wartezeiten.

Protokolltransaktionen arbeiten auf Zeilenbasis, wobei eine Zeile durch ein Newline (`\n`) begrenzt wird. Die Requests des Clients sind immer einzeilig; die Antworten des Servers mitunter auch mehrzeilig. Das kann man mit jeder beliebigen Shell vergleichen. Es stellt sich die Frage, wie die Größe einer Nachricht kommuniziert werden soll; dieses Problem nennt sich *Framing*. Es gibt bei der Konstruktion von Anwendungsprotokollen mehrere Denkweisen, um eine Nachricht „einzuboxen“:[22]

1. Alle Pakete gleich groß machen.
2. *octet-stuffing*: eine Zeichensequenz auf einer eigenen Zeile am Ende der Nachricht, zumeist ein Punkt (SMTP). Diese Zeichensequenz darf nicht in der Nachricht vorkommen und wird durch Escapen oder Duplikation verhindert (z.B. ist `.\n` unterscheidbar von `..\n`).
3. *octet-counting*: man zählt die Gesamtgröße in Byte und sendet sie zu Beginn (HTTP).
4. *line-counting*: man zählt zeilenweise statt byteweise (mpmp, libmangler v5). Diese Variante scheint nicht sehr weit verbreitet zu sein, aber ich halte es für sinnvoll, sie zu erwähnen.
5. *connection-blasting*: man öffnet eine neue Verbindung, sendet die Nachricht und schließt die Verbindung wieder (FTP[23]). Heutzutage nicht weit verbreitet, weil es *sehr* ineffizient ist, viele neue TCP-Streams zu öffnen und zu schließen.

22 vgl. Rose, „On the Design of Application Protocols“
23 Postel und Reynolds, „File Transfer Protocol“

Das aktuelle libmangler-Protokoll verwendet eine Variante des *octet-stuffing*: drei Bindestriche auf der letzten Zeile signalisieren das Ende der Antwort. Da diese Sequenz im Payload nicht vorkommen kann, ist es nie nötig, diese zu escapen. In Protokollversion 5 wurden Tags in Kombination mit Zeilenzählung implementiert; der Code auf Serverseite zählte die Newlines der ausgehenden Strings. Der Client war zu dem Zeitpunkt unfähig, überhaupt etwas zu empfangen. Diese Zeilenzählung wurde mit den Tags auch wieder entfernt und mit der „`---`“-Sequenz ersetzt, was auf Clientseite sehr einfach und robust funktioniert (`Connection.java:transact`):

```
while((line = in.readLine()) != null && !line.equals(ENDMARKER)) {
    Log.e("libmangler-proto", "[->proto] " + answer);
    answer.append(line);
}
```

Auf Serverseite war es viel einfacher als die vorige Lösung (`main.go:handle`):

```
fmt.Fprint(rw, ret)
fmt.Fprint(rw, protoEndMarker)
```

4.4 Geschichte und Ausblick

Das Protokoll durchlief elf Versionen, von denen die ersten nie implementiert wurden und andere wieder rückgängig gemacht wurden. Zu Beginn war die `sam`-Kommandosprache Hauptinspiration und in Version 1 sollte die Selektion funktionieren, indem über der ganzen Datenmenge mit regulären Ausdrücken gesucht wird. Da sich dies als schwer implementierbar erwies – der erste Server war in C und hatte keine Reflexionsmöglichkeiten – gab es bereits in Version 2 die Möglichkeit, nach ISBNs, Copy-IDs und Nutzernamen zu suchen, der `p`-Befehl lieferte aber noch JSON statt S-Expressions; mehrzeilige Antworten des Servers wurden mit einem einfachen Punkt begrenzt (vgl. SMTP[24]) und hatten eine Statuszeile zu Beginn.

Version 3 bringt S-Expressions. Version 4 bringt #tags, die Büchern, Copies und Nutzern hinzugefügt werden können. Version 5 nennt #tags in *Labels* um und fügt allen Requests und Responses Message-Tags wie in IMAP hinzu. Version 6 macht diese Änderungen, die große Komplexität im Client hervorriefen, wieder rückgängig und macht am Ende einer Antwort eine Zeile aus drei Strichen (`---`). Version 7 bringt die Kommandos, um #tags aufzulisten, zu erstellen und zu löschen. Version 8 erlaubt Suche nach Metadaten, Version 9 fügt einen Befehl

24 Klensin, „Simple Mail Transfer Protocol“

zum Auflisten von Selektionen hinzu. Version 10 implementiert *endlich* den Befehl zum Hinzufügen von Büchern vollständig; davor hat der nur die ISBN angenommen, weil es schwer ist, Titel und Autoren auf der „Kommandozeile“ des Befehls abzugrenzen. Jetzt verwendet der Befehl einfach eine S-Expression. Version 11 benannte die Kommandos `A` (Buch erstellen) und `a` (Copy erstellen) in `b` und `c` um, um, sich `u` (User erstellen) anzupassen.

Wie im letzten Absatz kurz abgerissen, ist das Protokoll einem stetigen Wandel unterworfen, um der Entwicklung der App und des Servers entgegenzukommen; gleichzeitig hat sich zentral seit Version 3 nichts geändert. Zukünftige Änderungen an der Sprache werden wohl einen ähnlich kleinen Maßstab haben. Die einzige größere Änderung wäre die Einführung eines Authentifizierungssystems und Verschlüsselung mit TLS.

5. Detailbetrachtung des Servers und des Clients

5.1 Server

Der Server basiert maßgeblich auf dem Protokoll. Zentral ist das Interface `Elem`, welches ein selektierbares Element repräsentiert und die auf alle anwendbaren Methoden enthält (`elem/sel.go`).

```
type Elem interface {
    fmt.Stringer              // returns the id (copies), ISBN
(books) or name (users)
    Print() string            // cmd p (all info)
    List() string             // cmd λ (single-line important info)
    Note(note string)         // cmd n
    Delete()                  // cmd d
    Tag(add bool, tag string) // cmd t
}
```

Die erste Zeile, `fmt.Stringer`, bettet das Interface `fmt.Stringer` in `Elem` ein, wodurch alle Methoden, die in `fmt.Stringer` sind, nun auch durch `Elem` gefordert werden. Das `fmt` bezeichnet das Package, in dem `Stringer` definiert ist. Wie viele Go-Interfaces, enthält `fmt.Stringer` nur eine einzige Methode `String() string`, welche also einen String zurückgibt; es ist das Equivalent zu Javas `toString`. Der Name solcher Ein-Methoden-Interfaces ist der Methodenname plus ein `er`-Suffix (vgl. `io.Reader`,

`io.Writer`). Die Implementierungen von `Elem` liefern als String nur die identifizierenden Informationen zurück, so die ID, die ISBN oder der Nutzername.

`Print` liefert die S-Expression zurück, die alle Informationen zu dem Element enthält; der `p`-Befehl im Protokoll sendet die S-Expressions jedes Elements in *Dot*. Für die anderen Methoden in `Elem` gibt es ebenso Protokollbefehle, wie man in den Kommentaren nach den Methoden lesen kann.

Die drei Implementationen von `Elem` sind `*Book`, `*Copy` und `*User`. Auf die Sterne (`*`) kommen wir noch zurück. Betrachten wir Bücher als Beispiel. Das ist die Definition eines `Books` (`elem/book.go`):

```
type Book struct {
    ISBN    ISBN
    Title   string
    Authors []string
    Notes   []string
    Tags    []string
    Copies  []*Copy
}
```

Die Felder `Authors`, `Notes` und `Tags` sind *Slices* vom Typ `string`. Slices sind Arrays ähnlich, lassen sich jedoch vergrößern und werden als Referenzen übergeben, im Gegensatz zu Go-Arrays, welche eine fixe, im Typ enthaltene Größe haben (`[3]int` und `[4]int` sind grundlegend verschiedene Typen) und direkt übergebene Werte sind. `copies` ist eine Slice aus Pointer zu Copies.

Eine Methode sieht in Go folgendermaßen aus:

```
func (b *Book) String() string {
    return string(b.ISBN)
}
```

Der `(b *Book)`-Teil nennt sich *Receiver* und gibt an, auf welchen Typ eine Methode definiert ist (hier `*Book`) und wie die Instanz benannt wird, auf der die Methode ausgeführt wird (hier `b`). Es ist einfach ein spezieller Parameter. Man kann Methoden auf den Grundtyp definieren (`Book`), dann bekommt man eine Kopie der Instanz, weil Go *Pass-by-Value* bei Parametern nutzt. Wenn man also die Instanz *modifizieren* will, muss man die Methode auf einen Pointer definieren (`*Book`). Das nennt man dann einen *Pointer Receiver*.

Man sollte einfache Receiver verwenden, keine *Pointer Receiver*, sofern es nicht nötig ist, weil man von Pointern durch Indirektion einfach auf den Grundtyp schließen kann und das meist hilfreicher ist. Man könnte also auf den Gedanken kommen, die `String`-, `Print` und `List`-Methoden, die nichts modifizieren, auf `Book` zu definieren, die anderen Methoden von `Elem` auf `*Book`. Das ist jedoch nicht zielführend: `Book` und `*Book` sind verschiedene Typen und keiner von beiden würde in dem Fall das Interface `Elem` implementieren. Deswegen sind die drei Implementierungen von `Elem` Pointer: `*Book`, `*Copy`, `*User`.

Der Server speichert alle Bücher, Copies und User in drei Maps ab, die den jeweiligen Identifikatoren Pointer auf die Structs zuordnen (`elem/sel.go`).

```
var Books map[ISBN]*Book
var Copies map[int64]*Copy
var Users map[string]*User
```

Wegen dieser Maps ist der selektierende Teil des Protokolls recht einfach. Die `Select`-Funktion in `elem/sel.go` ist öffentlich und ist nur eine Zwischenstufe, die die eigentlichen Selektierroutinen in `seltab` aufruft. `seltab` ist eine statische Map von einzelnen Unicode-Zeichen (auch *Runen* genannt; hier `B`, `C`, `U`) zu Funktionen vom Typ `selFn` mit folgender Signatur:

```
type selFn func(sel []Elem, args []string) ([]Elem, error)
```

Eine `selFn` nimmt eine bestehende Selektion sowie die Selektionsargumente an und gibt eine Selektion und einen Fehlerwert zurück. Der Großteil des Codes in `seltab` bestimmt recht mechanisch den Typ des Arguments und selektiert das, was man erwarten würde (siehe Tabelle in Sektion 4.1).

Viel mehr lässt sich zum Server nicht sagen: in `manglersrv/main.go` wird für jede Verbindung eine Goroutine (eine Art leichter Thread[25]) gestartet, die dann `handle` ausführt, welches wiederum für alle Requests `interpret` aufruft und den sonstigen Zustand der Verbindung hält – inklusive *Dot*. Das Speichern der Elemente auf der Festplatte wird in `manglersrv/store.go` bewertstelligt, indem der Server für Bücher, Copies und User jeweils eine Datei erstellt und das Ergebnis von `interpret(`*X*`, &dot)` in die entsprechende Datei schreibt, wobei *X* bei Büchern `Bp`, bei Copies `Cp` und bei Usern `Cp` ist.

25 Griesemer, Pike, und Thompson, „Go FAQ – goroutines“

Das `dot` ist in dem Fall ein Dummy. Beim Laden werden die S-Expressions der Elemente durch einen simplen *recursive-descent* S-Expression-Parser gehetzt. Das Resultat ist ein Baum, der pre-order durchlaufen wird. Bei jedem Atom, d.h. bei jedem Blatt des Baums, wird eine Funktion aufgerufen, die eine Zustandsmachine weiterschaltet, die alle Informationen aus dem Baum extrahiert und so das Element erzeugt.

5.2 Client

Die Android-App ist der Client, mit welchem dem Protokoll eine grafische und mobile Nutzerschnittstelle verliehen wird. Aus der Nutzerperspektive ist die App aus verschiedenen Teilen aufgebaut: dem Hauptscreen, der ein Menü bildet, von dem aus man alle Teile des Clients erreichen kann; den drei Infoscreens für Bücher, Copies und User; und ein Auflistungsscreen, dessen Bücher-, Copy- oder User-Einträge man anklicken kann, um auf den jeweiligen Infoscreen zu schalten. Die Infoscreens bestehen aus einem Textfeld, welches das Element anzeigt, und einer zweiseitigen Liste an Befehlen wie „Ausleihen ...“ oder „Beiseitestellen“. Die Ellipse zeigt an, dass weitere Eingaben des Nutzers gefordert sind oder dass der aktuelle Screen verlassen wird, um einen anderen anzuzeigen.

Das Feature, weswegen es sinnvoll ist, den Client auf einem Mobilgerät zu implementieren, ist das Scannen von QR-Codes. Man bringt an jeder Exemplar seine ID in Stringform als QR-Code an und kann diesen Code mit der App scannen, um Informationen zu diesem Exemplar einzuholen und Befehle wie das Verleihen auszuführen. Die gebräuchiche Variante ist es in Android, auf ZXing[26] zurückzugreifen, einer App, die eine Vielzahl von ein- und zweidimensionalen Barcode-Formaten lesen kann und einen Intent dafür bereitstellt (`com.google.zxing.client.android.SCAN`).

Bei der Programmierung der App wurde auf möglichst geringe Code-Komplexität Wert gelegt, in Übereinstimmung mit der *New Jersey*-Denkweise:[27]

> [...] Simplicity – the design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design. [...]

26 „ZXing“
27 Gabriel, „The rise of Worse Is Better“

Schlussendlich wurde dieses Ziel erreicht, aber die App hat einen steinigen Weg hinter sich – Entwicklungsschwierigkeiten, überkomplexe Teillösungen und Vernachlässigung wegen eines anderen Projekts. Die Entwicklungsschwierigkeiten rühren daher, dass der Autor zu Beginn kein funktionierendes Smartphone zum Testen hatte, und dass der Server in der Schule, wo zu Beginn entwickelt wurde, nicht funktionierte, der Go-Compiler als Virus eingestuft wurde und wird und die Rechnerleistung zu wünschen übrig lies. Nachdem die Entwicklung auf Linux fortgesetzt wurde, ohne IDE und mit einem Texteditor ohne Syntax Highlighting, waren diese Nebenprobleme beseitigt und das Hauptproblem trat in den Vordergrund: das Protokoll war synchron und zustandsbehaftet, Android erlaubt jedoch blockierendes Netzwerken im EDT im Standardfall nicht. Der Versuch, Message-Tags wie in IMAP einzuführen und damit die Zustandshaftigkeit zu begrenzen, blähte den Code immens auf und wurde wieder verworfen. Der Durchbruch passierte mit dem Erlauben synchronen Netzwerkens auf dem EDT[28] entgegen der Android-Prinzipien.

Diesen Regelbruch will ich kurz rechtfertigen. Man sollte nicht im zeichnenden Thread blockierende Operationen durchführen, weil das Aktualisierung der GUI dann nicht geschehen kann. Stattdessen soll man Netzwerkoperationen asynchron starten und dann in einem Callback, wenn das Ergebnis angekommen ist, dieses auswerten. Nun ist meine App so geschrieben, dass man vor der Antwort des Servers nichts, *nichts*, Sinnvolles tun kann. Die Zeitpunkte, an denen die App eine Anfrage stellt, sind vorhersehbar: nach dem Scannen einer ID, nachdem eine Suche gestartet wurde, nachdem nach einer Auflistung verlangt wurde und wenn Detailinformationen zu einem Element angezeigt werden sollen. Es gibt kein Pollen und keine Initiative vom Server; der Client initiiert jeglichen Protokollaustausch. Zudem ist auf dem Socket ein Timeout von drei Sekunden gesetzt: wenn die Verbindung schlecht ist, wird der Versuch abgebrochen und die UI kann wieder aktualisiert werden.

Die `transact`-Funktion in `Connection.java` bildet die Basis für die Netzwerkoperationen. Sie nimmt einen zu sendenden String an und liefert die Antwort als String zurück. Auf dieser Basis existieren viele kleine Routinen in `Connection.java`, wie zum Beispiel `printCopy`:

28 user1169115, „How to fix android.os.NetworkOnMainThreadException?“

```
public String printCopy(long... id) {
    return transact("C/" + mksel(id) + "/p");
}
```

In der App wurde die Möglichkeit nicht genutzt, mehrere IDs hier anzugeben, aber in den Signaturen ist immer noch `long...` zu finden. `mksel` ist eine kleine private Methode, die dieses Array an IDs zu einem String vereint, der dann an den Server gesendet werden kann.

Ein weiteres Merkmal der Simplizität des Clients sind die verschiedenen Ansichten beziehungsweise Layouts. In *libmangler* gibt es nur eine Activity, die `MainActivity`. Zwischen den Layouts wird mit einem `ViewFlipper` umgeschaltet, dem man mit dessen Methode `setDisplayedChild` mitteilen kann, die wievielte Ansicht angezeigt werden soll. Da das häufig vorkommt, gibt es eine kleine Helferprozedur.

```
/**
 * Flip to a linear layout. See indexes at the top of MainActivity.
 */
private void flipView(int layout) {
    ViewFlipper vf = (ViewFlipper) findViewById(R.id.flipper);
    vf.setDisplayedChild(layout);
}
```

Die Informationsscreens für Bücher, Copies und Nutzer haben jeweils zwei Seiten voller Befehle; auch hier wurde ein `ViewFlipper` verwendet.

Der Großteil des Codes von `MainActivity` ist in den `initLayout`-Prozeduren zu finden, in denen den Buttons ihre Aktion zugewiesen wird. Da es in Java 5 noch keine Lambda-Ausdrücke gibt, ist die Notation recht umständlich und mechanisch. Zuletzt ist noch `StringDialog` nennenswert, welches einen Dialog erstellt, in dem ein String eingegeben werden kann. Der zu viel Speicher verbrauchende Code in `Model.java` ist eine Übersetzung des Go-Codes für das Einlesen der S-Expressions von Büchern, Copies und Usern und hätte erlaubt, auf den Infoscreens eine schönere Darstellung anstatt der rohen S-Expression zu zeigen; das Testhandy konnte aufgrund der Rekursion im Parser nicht mithalten. Die Datei ist inzwischen veraltet und nur im System, um zu demonstrieren, wie es hätte gehen sollen. Ohne `Model.java` hat der Client überhaupt kein Verständnis von den Daten, welche er handhabt. Das stellte sich aber als ein Non-Problem heraus, weil das Protokoll alles

abdeckt, was der Client können soll. Screenshots dreier selbsterklärender Ansichten sind inm Ordner `img` zu finden.

6. Ausblick

Abschließend lässt sich sagen, dass die Seminararbeit doch noch eine gute Wendung genommen hat, als die App endlich vervollständigt wurde. Dennoch gibt es einige Punkte, die man noch verbessern kann kann. So fehlt ein Authentifizierungssystem oder die Verschlüsselung des Protokolls mittels TLS noch komplett. Zudem ist das Einpflegen von Daten momentan nur mithilfe der App möglich; will man am Desktopcomputer Daten einpflegen, muss man eine `netcat`-Session öffnen und Protokollanfragen per Hand stellen; ein grafischen Desktop-Client, wie anfangs geplant, wurde nicht realisiert. Es war eine gute Entscheidung, den Server in Go zu realisieren; es war das erste Go-Projekt des Autors und funktionierte dennoch schon früh sehr gut. Auch die Entscheidung, auf eine SQL-Datenbank zu verzichten und stattdessen auf Dateien zu setzen, war sinnvoll: der Server läuft ohne Installation oder Konfiguration auf jedem von Go unterstützten Betriebssystem, also Unix-likes, Windows, Mac OS und Plan 9.

7. Danksagungen

Ich danke im Besonderen Leander Dreier für seine Kommentare zum Text, die verhindert haben, dass der nicht vertiefte Leser sich verläuft, und für sein Hilfe als Betatester der App. Des Weiteren danke ich den Erfindern von Go für ihre tolle Sprache, ohne die der Server weit schwieriger zu implementieren gewesen wäre, sowie Rob Pike, dem Autoren meines Texteditors `sam`, der die Inspiration für das Protokoll gegeben hat und in dem die App, der Server und die Seminararbeit geschrieben wurden. Zuletzt danke ich den Autoren von *The Elements of Style*[29] für die hilfreichen Ratschläge in ihrem Buch, sowie den zahlreichen Fragestellern und -beantwortern von StackOverflow.

8. Glossar

Begriff	**Erläuterung**
Buch (Book).	Eine Veröffentlichung, identifiziert durch ISBN.
Dot	Aktuelle Selektion im Protokoll: Menge an Büchern, Copies oder Usern.

29 Strunk Jr. und White, *The Elements of Style*

Begriff	Erläuterung
Element	Kann Teil der Selektion sein: entweder ein Buch, eine Copy oder ein User.
Exemplar (Copy)	Physikalisches Medium, einen Buch zugeordnet. Das System vergibt eine ID.
Infoscreen	Screen, auf dem ein Element zu sehen ist sowie eine Befehlsliste.
Nutzer (User)	Potentieller Ausleiher von Copies, identifiziert durch den Namen.
Screen	Ansicht der App. Jedes Layout im zentralen ViewFlipper ist ein Screen.
Selektionsarg.	ISBN, Copy-ID, User, #tag, Metadaten-*Name:Wert*-Paar.
S-Expression	(das (ist eine (verschachtelte) (S-Expression)) (siehe Lisp))
synchron	nicht asynchron, blockierend

9. Bibliographie

Verwendete Hilfsmittel/Programme

- Eclipse (zu Beginn)
- sam, awk, mk, sowie weitere Unix-Befehle
- Android SDK 15
- Apache Ant
- pandoc, pandoc-citeproc, BibLaTeX

Diese ODT-Datei wurde aus `paper.md` um `Sun  6 Nov 19:57:50 CET 2016` durch den Befehl `mk note && cat paper.md note metadata.yaml | pandoc --filter pandoc-citeproc --biblatex -o paper.odt --reference-odt ref.odt` generiert.

Quellen

„9p(2) of the Plan 9 man pages“. Zugegriffen 6. November 2016. http://man.cat-v.org/plan_9/2/9p.

Belshe, M., R. Peon, und M. Thomson. „Hypertext Transfer Protocol Version 2 (HTTP/2)“. Request for comments. Internet Engineering Task Force; RFC 7540 (Proposed Standard); IETF, Mai 2015. http://www.ietf.org/rfc/rfc7540.txt.

Chamberlin, Donald D., und Raymond F. Boyce. „SEQUEL (Structured English Query Language)“. In *Proceedings of the 1974 ACM SIGFIDET workshop on data description,*

access and control, 249–64, 1974. http://www.almaden.ibm.com/cs/people/chamberlin/sequel-1974.pdf.

Crispin, M. „INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1“. Request for comments. Internet Engineering Task Force; RFC 3501 (Proposed Standard); IETF, März 2003. http://www.ietf.org/rfc/rfc3501.txtUpdated by RFCs 4466, 4469, 4551, 5032, 5182, 5738, 6186, 6858, 7817.

Daviel, Andrew. „Web time“. Zugegriffen 5. November 2016. http://vancouver-webpages.com/time/.

Gabriel, Richard P. „The rise of Worse Is Better“. Zugegriffen 5. November 2016. http://dreamsongs.com/RiseOfWorseIsBetter.html.

Griesemer, Robert, Rob Pike, und Ken Thompson. „Go FAQ – goroutines“. Zugegriffen 6. November 2016. https://golang.org/doc/faq#goroutines.

———. „The Go programming language“. Zugegriffen 6. November 2016. https://golang.org.

hobbit. „Netcat: The TCP/IP swiss army“. Zugegriffen 6. November 2016. http://nc110.sourceforge.net/.

„How accurate is the CMOS clock?“ Zugegriffen 5. November 2016. http://www.ntp.org/ntpfaq/NTP-s-trbl-spec.htm#AEN5674.

„intro(5) of the Plan 9 man pages“. Zugegriffen 5. November 2016. http://man.cat-v.org/plan_9/5/intro.

Killian, Tom J. „Processes as files“. In *Proceedings of the summer 1984 USENIX conference*, 203–7, 1984. http://lucasvr.gobolinux.org/etc/Killian84-Procfs-USENIX.pdf.

Klensin, J. „Simple Mail Transfer Protocol“. Request for comments. Internet Engineering Task Force; RFC 2821 (Proposed Standard); IETF, April 2001. http://www.ietf.org/rfc/rfc2821.txtObsoleted by RFC 5321, updated by RFC 5336.

McCarthy, John. „Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I“. *Communications of the ACM*, 1960. http://www-formal.stanford.edu/jmc/recursive.html.

Mills, D., J. Martin, J. Burbank, und W. Kasch. „Network Time Protocol Version 4: Protocol and Algorithms Specification“. Request for comments. Internet Engineering Task Force; RFC 5905 (Proposed Standard); IETF, Juni 2010. http://www.ietf.org/rfc/rfc5905.txtUpdated by RFC 7822.

Mills, D.L. „Network Time Protocol (NTP)“. Request for comments. Internet Engineering Task Force; RFC 958; IETF, September 1985. http://www.ietf.org/rfc/rfc958.txtObsoleted by RFCs 1059, 1119, 1305.

Myers, J., und M. Rose. „Post Office Protocol - Version 3“. Request for comments. Internet Engineering Task Force; RFC 1939 (INTERNET STANDARD); IETF, Mai 1996. http://www.ietf.org/rfc/rfc1939.txtUpdated by RFCs 1957, 2449, 6186.

Pike, Rob. „The text editor sam“. *SOFTWARE—PRACTICE AND EXPERIENCE* 17, Nr. 11 (1987): 813–45. http://doc.cat-v.org/plan_9/4th_edition/papers/sam/.

Pike, Rob, Dave Presotto, Ken Thompson, und Howard Trickey. „Plan 9 from Bell Labs“. In *Proceedings of the summer 1990 UKUUG conference*, 1–9, 1990. http://doc.cat-v.org/plan_9/4th_edition/papers/9.

Postel, J. „Transmission Control Protocol“. Request for comments. Internet Engineering Task Force; RFC 793 (INTERNET STANDARD); IETF, September 1981. http://www.ietf.org/rfc/rfc793.txtUpdated by RFCs 1122, 3168, 6093, 6528.

Postel, J., und J. Reynolds. „File Transfer Protocol“. Request for comments. Internet Engineering Task Force; RFC 959 (INTERNET STANDARD); IETF, Oktober 1985. http://www.ietf.org/rfc/rfc959.txtUpdated by RFCs 2228, 2640, 2773, 3659, 5797, 7151.

Rose, M. „On the Design of Application Protocols“. Request for comments. Internet Engineering Task Force; RFC 3117 (Informational); IETF, November 2001. http://www.ietf.org/rfc/rfc3117.txt.

Strunk Jr., William, und E.B. White. *The Elements of Style*. Fourth edition. Pearson, 1999.

Surveys, W3Techs Web Technology. „Usage of HTTP/2 for websites“. Zugegriffen 6. November 2016. https://w3techs.com/technologies/details/ce-http2/all/all.

user1169115. „How to fix android.os.NetworkOnMainThreadException?“, 2012. http://stackoverflow.com/a/9289190.

„ZXing“. Zugegriffen 6. November 2016. https://github.com/zxing/zxing

10. Eidesstattliche Erklärung

Ich erkläre, dass ich die Seminararbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis aufgeführten Quellen und Hilfsmittel verwendet habe.

Ingolstadt, den 07. November 2016

_____

(Dominik Okwieka)