Softwareprojekt Mobilkommunikation (SS09)

Abschlussbericht

1 Einleitung

(von Alexander E.)

Im Rahmen der Veranstaltung „Softwareprojekt Mobilkommunikation“ haben wir uns mit der Software Architektur für mobile Geräte beschäftigt. Das Ziel des Praktikums war es für einen vorgegebenen Anwendungsfall zuerst eine geeignete Anwendungsarchitektur zu entwerfen und diese anschließend zu implementieren.

Bei Android handelt es sich um eine komplette Software für mobile Geräte, die ein Betriebssystem, Middleware und einige Schlüsselanwendungen beinhaltet. Android basiert auf dem Betriebssystem Linux und verwendet als Programmiersprache Java. Die Software unterliegt.

Vorgabe für unsere Anwendung ist folgender Anwendungsfall:

- Benutzer A speichert seine Profildaten ab und teilt Benutzer B die #ID seiner Profildaten mit
- Benutzer B fügt Benutzer A mittels dieser #ID in sein Adressbuch ein
- Benutzer A autorisiert Benutzer B seine Profildaten zu lesen
- Benutzer B hat die Profildaten von Benutzer A in seinem Adressbuch gespeichert
- Benutzer B wird über Aktualisierungen der Profildaten von Benutzer A informiert

Zu Beginn des Projektes war die Frage der Netzwerkarchitektur zu klären. Eine Server/Client-Lösung wäre leichter zu implementieren, hierbei läge die Kontrolle allerdings beim Provider. Bei einem Peer-to-Peer-Modell hingegen ist eine Verbindung zwischen den einzelnen Geräten erforderlich, statt einer zentralen Verwaltung. Dieses Modell wäre schwieriger umzusetzen, dafür eignet sich diese Lösung besser für unseren Anwendungsfall. Es wurde beschlossen, dass beide Lösungen umgesetzt werden, damit die restlichen Komponenten zeitnah in ein Netzwerk zum Testen integrieren können.

Zur Entwicklung der einzelnen Komponenten wurden kleine Arbeitsgruppen gebildet, die je nach Arbeitsaufwand dynamisch in ihrer Gruppenstärke angepasst wurden. Um den erfolgreichen Ablauf zu sichern, wurden in der Anfangsphase Meilensteine definiert und ein Arbeitsplan erarbeitet. Innerhalb des Praktikums wurde mit dem Android Emulator gearbeitet, als Versionsverwaltung wurde Subversion eingesetzt. Für das Projekt haben wir das Android SDK 1.5 verwendet und als Programmierumgebung wurde Eclipse eingesetzt.

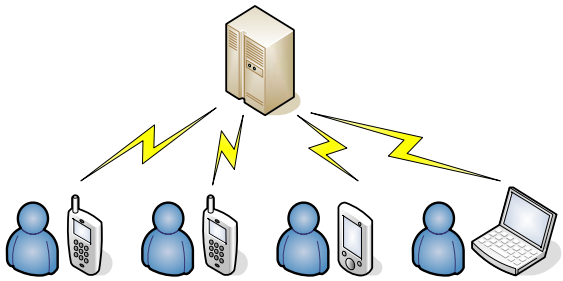
Im Folgenden werden die einzelnen Komponenten und ihr aktueller Status vorgestellt.

2 Architektur

(von Ronald S.)

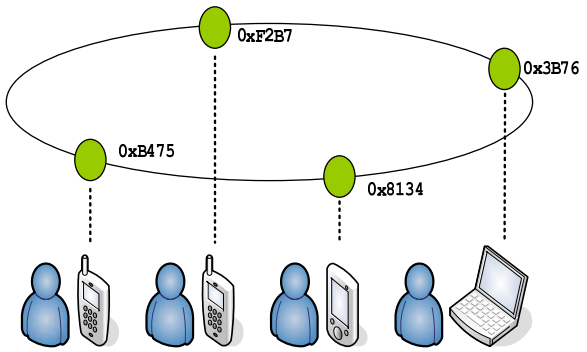
Für den „Profile-Exchange“ Anwendungsfall standen zwei mögliche Varianten für die Netzwerkstruktur zur Auswahl: Eine Implementierung als Server/Client-Struktur oder als Peer-to-Peer Netzwerk.

Vorteile der Client/Server-Architektur:
Relativ einfache Implementierung, keine redundanten Daten. Außerdem besaßen alle am Projekt Beteiligten schon Erfahrungen und Grundkenntnisse über diese Architektur aus anderen Veranstaltungen.



Nachteile:
Hohe Anforderungen an die Verfügbarkeit des Servers (single point of failure)

Vorteile von Peer-to-Peer:
Ständige Verfügbarkeit der Daten, keine zentrale Kontrolle



Nachteile:
Komplexer und komplizierter zu implementieren, benötigt Direktverbindung zwischen den Endgeräten. Außerdem mussten sich die Beteiligten erst das nötige Wissen über Peer-to-Peer aneignen.

Der Prototyp der Anwendung wurde daher als Server/Client-Struktur implementiert. Die fertige Anwendung wird über ein Peer-to-Peer Netzwerk arbeiten.

Bei der Prototyp-Implementierung werden alle Profildaten auf einem Server gespeichert. Dieser kommuniziert über das Client/Server-Protokoll via UDP mit dem Client. Eingegangene Daten werden an die GUI der Anwendung weitergeleitet. Diese holt nötigenfalls Daten aus der eigenen Datenbank oder speichert die erhaltenen Daten dort.

Als Schnittstelle zwischen Client und GUI dient das I_Network-Interface. Das Interface wird auch für die Peer-to-Peer-Lösung der Anwendung verwendet. Die bestehende GUI kann somit während der Entwicklung weiter zu Testzwecken und für die finale Version verwendet werden. Außerdem können über das Interface später weitere Anwendungen an das dann entwickelte Peer-to-Peer-Netzwerk angebunden werden.

Die Anwendung kommuniziert jetzt über das Interface mit einem P2P-Adapter, der die Daten für den eigentlichen P2P-Service verarbeitet. Der Adapter erstellt Dateien, die die Informationen enthalten, die zwischen zwei Anwendungen ausgetauscht werden müssen, z.B. Benachrichtigungen über Profilupdates, Autorisierungsanfragen, usw. Die Dateien werden vom P2P-Service zur Kommunikation mit anderen Geräten benutzt und wie normale Nutzdaten im Peer-to-Peer-Netz verteilt.

2.1 Client/Server

2.1.1 Komponenten und Interfaces

(von Karsten G.)

Das Verbinden mehrerer Android-Emulatoren über ein Netzwerk bedurfte einiger Kniffe. Jeder Emulator auf einem Rechner läuft hinter einem virtuellen Router. Sollte ein Programm auf dem Emulator versuchen die Adresse `127.0.0.1` aufzurufen, so wird der Aufruf nicht an den Hostrechner weitergeleitet. Soll der Aufruf an den Hostrechner selbst gehen, so muss dieser stattdessen an `10.0.2.2` adressiert werden.

Eingehende uns ausgehende Anfragen werden standardmäßig geblockt und müssen explizit freigegeben werden. Dies geschieht über die *Android Debug Bridge (ADB)*. Diese ist ein Programm, welches im Hintergrund läuft und es erlaubt Ereignisse vom Emulator zu empfangen und Ereignisse / Kommandos an diesen zu senden. Um eine Weiterleitung vom Android (Port X) auf den Host PC (Port Y) einzurichten, muss man sich über die ADB mit dem Emulator verbinden. Die ADB lässt sich für den ersten Server via TCP auf Port 5554 ansprechen. (Für jeden weiteren Server wird die Portnummer um 2 inkrementiert.) Ein Weiterleitungs-Befehl für eine UDP-Weiterleitung sieht dann wie folgt aus:

```
nc localhost 5554
redir add udp:X:Y
```

Diese Weiterleitung erlaubt eingehende Verbindungen nur dann, wenn diese von dem Hostrechner kommen. Dies lies sich leider nicht umkonfigurieren. Ausgehende Verbindungen jedoch, funktionieren ohne Probleme. Um eingehende Verbindungen von anderen Rechnern zu erlauben, beispielsweise, um ein Szenario mit mehreren Teilnehmern zu testen, bedurfte es einer eigens entwickelten Anwendung: der **AndroidBridge**.

Die Bridge nimmt eingehende UDP Pakete entgegen und leitet diese an den Emulator weiter. Damit der Emulator die Absender der Pakete auseinander halten kann, müssen deren Absenderadressen geändert werden. (Beim Weiterleiten wäre sonst die Absenderadresse stets `127.0.0.1`. Da Java kein UDP-Spoofing erlaubt, haben wir uns entschieden, den Quellport zu ändern. Dieses wurde über eine HashMap (HashMap<SocketAddress, Integer>) realisiert. Der Fall für Ausgehende Verbindungen wurde nicht implementiert, da der Client (auf dem Emulator) stets mit einer festen Adresse (dem Server) kommuniziert.

Im Rahmen des **ContactServer** Sub-Projektes wurde ein Server entwickelt, welcher Benutzer-, Kontakt- und Nachrichtenverwaltung unterstützt. (Genaueres dazu im Kapitel „Protokolle“) Der Server diente in erster Linie zum Testen der Applikation und sollte später von einer P2P Implementierung abgelöst werden. Der Austausch des Servers durch eine P2P Implementierung kann problemlos geschehen, wenn diese das `net.I_Network` Interface implementiert.

2.1.2 Client/Server-Protokoll

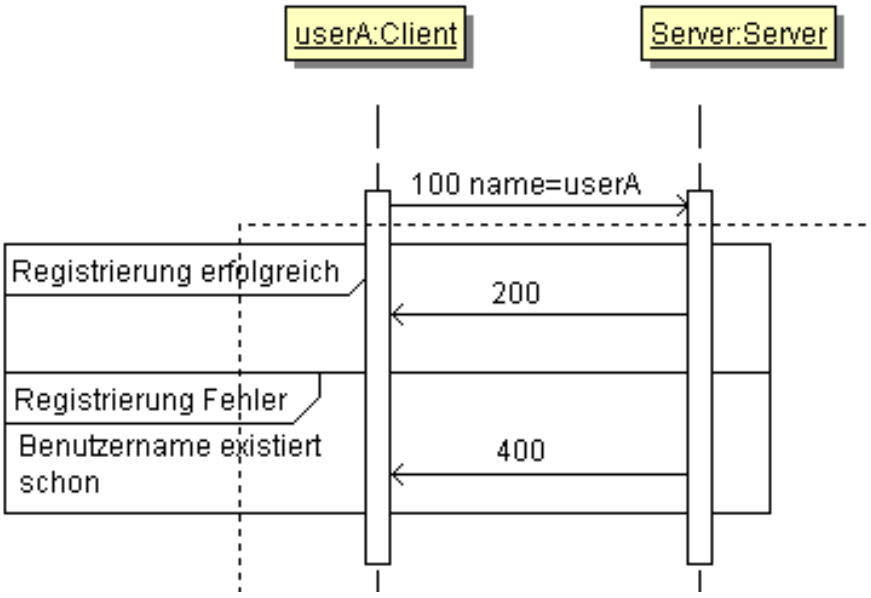
(von W. Yu)

Unser Client/Server-Protokoll unterstützt folgende Funktionalitäten:

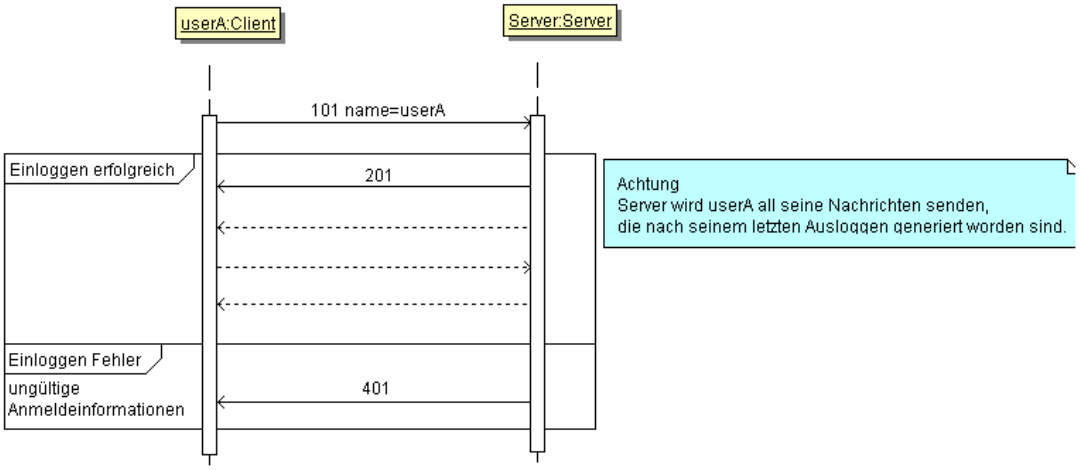
1) Registrierung eines neuen Benutzers
2) Anmeldung von Benutzern
3) Abmeldung von Benutzern
4) Aktualisierung von Benutzerprofilen
5) Einfügen eines neuen Kontakts
6) Anfrage der Benutzerprofile von all seinen Kontakten
7) Anfrage von Anzahl der Kontakte

Der Nachrichtenaustauch zwischen Client und Server ist im Folgenden graphisch dargestellt. Für Details siehe Protokoll Release 0.4.

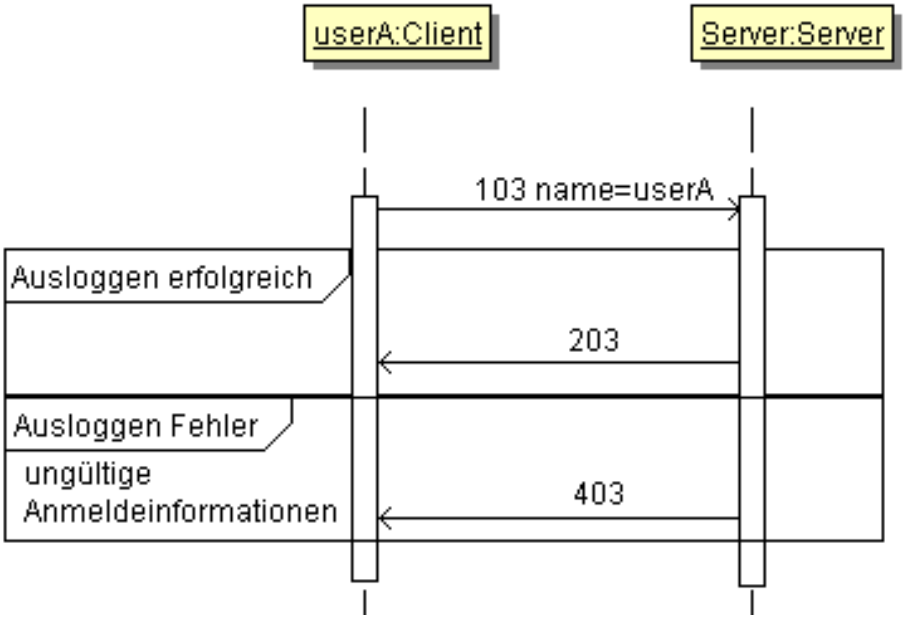
Registrierung von userA:



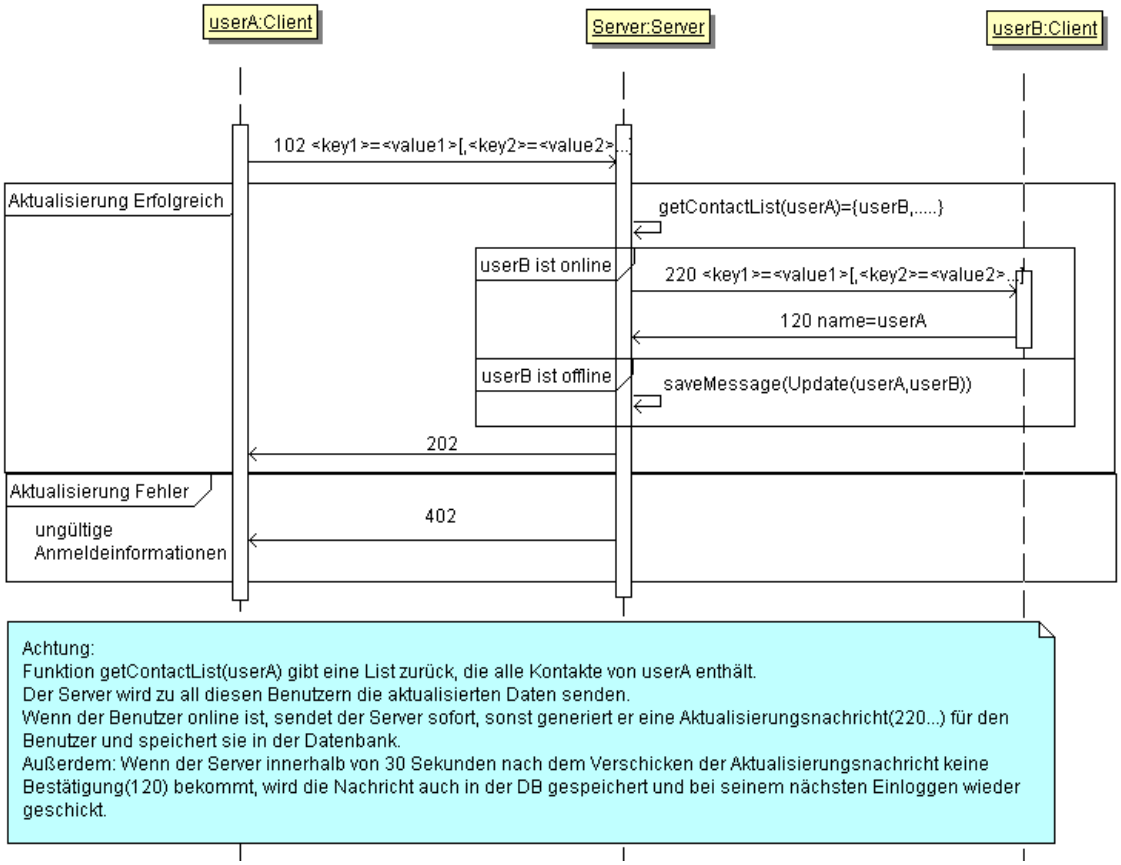
Anmeldung von userA:



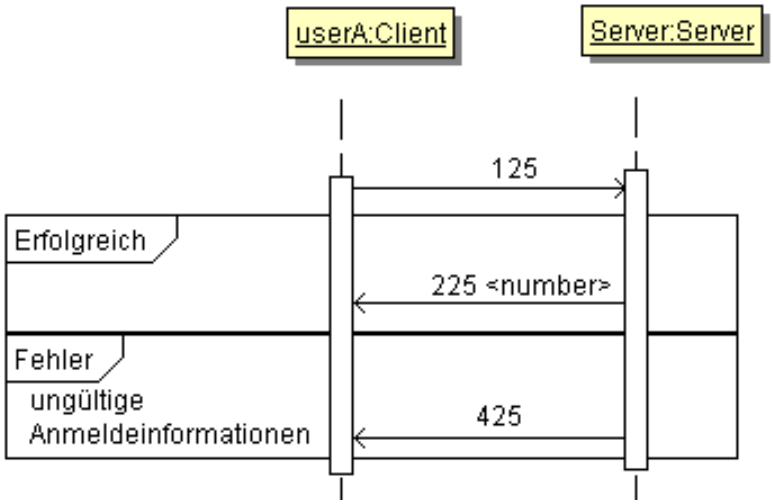
Abmeldung von userA:



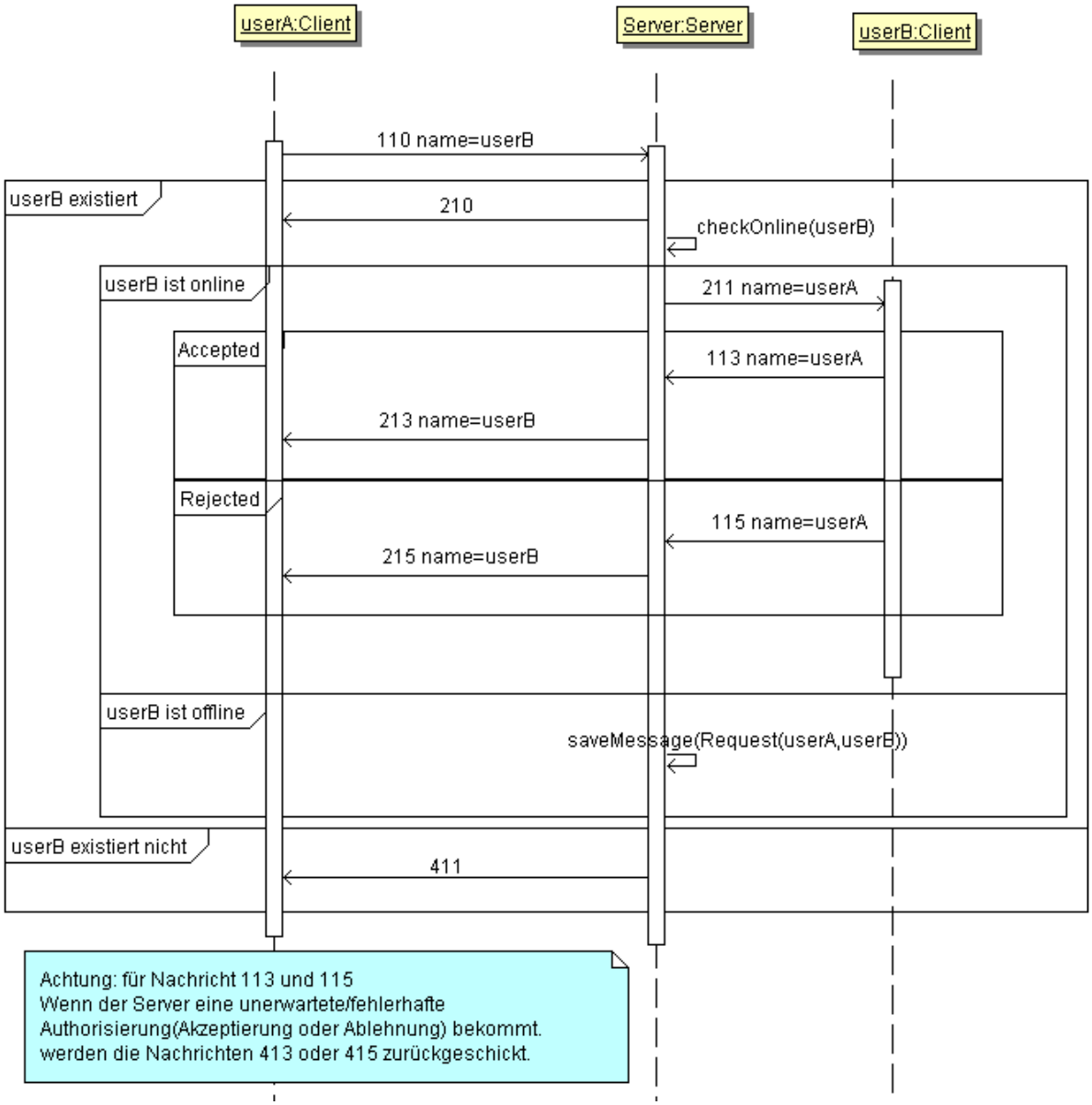
Aktualisierung des Profils von userA:



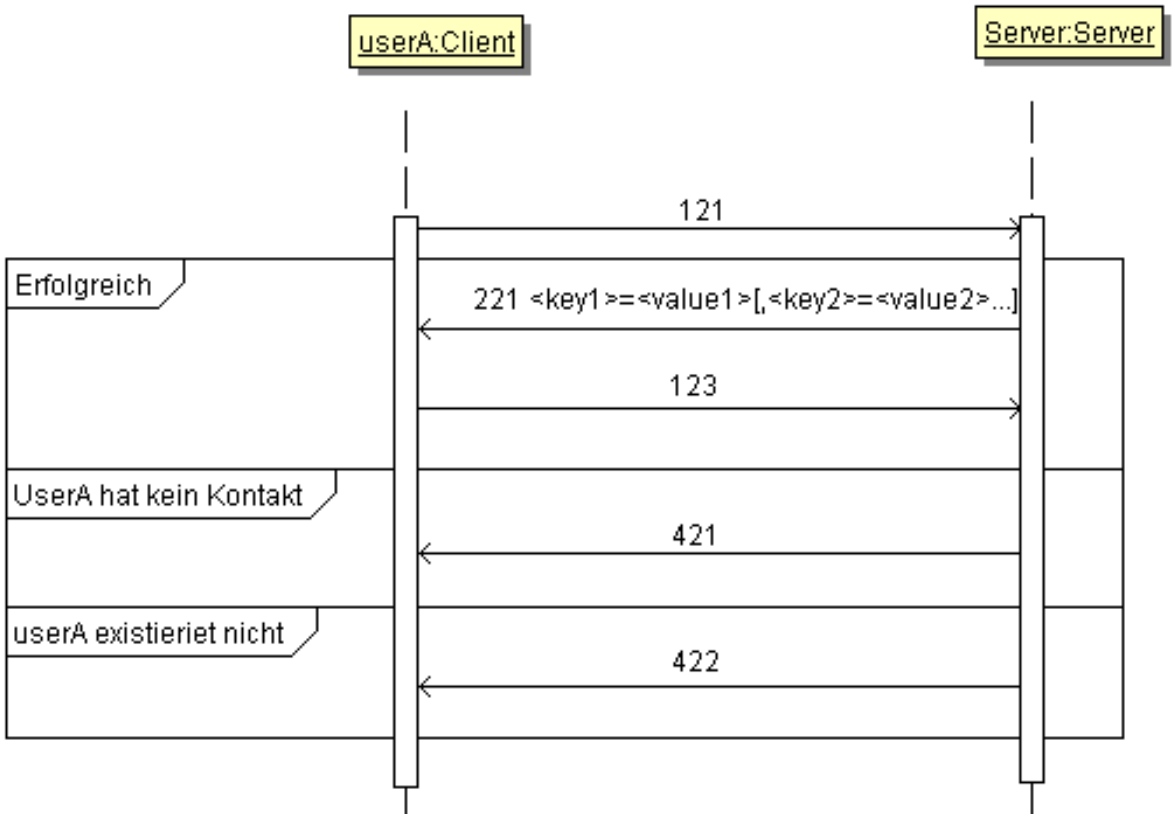
Anfrage von Anzahl der Kontakte:



Einfügen eines neuen Kontakts von userA:



Anfrage der Benutzerprofile von all seinen Kontakten:



In der Zukunft kann man noch mehr fehlerbehandelnde Nachrichten implementieren um das Protokoll zu verfeinern.

2.2 Peer-to-Peer

2.2.1 Komponenten und Interfaces

(von Sebastian W.)

Pastry als Anfang

Basis und Ausgangspunkt der Entwicklung der einzelnen Interfaces und Komponenten unserer Peer2Peer Implementierung waren die im Pastry Paper[1] beschriebenen Datenstrukturen und die dort empfohlene API. Auf diesen Kernstrukturen bauten wir den P2PService auf.

Pastry Schnittstellen

Das Paper[1] empfahl uns fünf Schnittstellen, die wir auch so übernahmen.

[1] A. Rowstron and P. Druschel, „Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems“ in Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), November 2001

Auf der Pastry Seite:

`nodeId = pastryInit(Credentials, Application)`
Wird aufgerufen von der Application und löst das Joinen des neu erstellten Knotens im Pastry Netz aus, wobei die Statetable generiert und die ID des Knotens (bei Erfolg) zurückgegeben wird. Im Folgenden wird dann die Referenz auf **`Application`** genutzt um über gewisse Geschehnisse zu informieren.

`route(msg,key)`
Löst das Routen einer **`msg`** an den Knoten aus, dessen ID numerisch am Nächsten zu ***`key`*** ist.

Auf der Application Seite (Bei uns P2P Service genannt):

`deliver(msg,key)`
Wird aufgerufen von Pastry, wenn eine Nachricht eingetroffen ist, die für den aktuellen Knoten bestimmt war (also die eigene Knoten-ID numerisch am Nächsten zum **`key`** ist).

`forward(msg,key)`
Wird aufgerufen, wenn eine Nachricht eingetroffen ist, die nicht für den aktuellen Knoten bestimmt war, und folglich als nächstes weitergeleitet werden soll. Wird jedoch von der **`Application`** der key auf NULL gesetzt, so wird das weiterleiten unterbunden.

`newLeafs(leafSet)`
Wird aufgerufen, wenn es eine Änderung im Leafset gibt. Gibt der **`Application`** die Möglichkeit gewisse Applikationsspezifische Invarianten abzugleichen.

`Credentials`
Müssen die Funktion toString() realisieren und werden von Pastry verwendet, um den Key zu generieren. In unserem Fall wird die IP verwendet. Später wäre es die IP plus Port.

Komponenten Koppelung

Eine weitere Komponente ist die den P2P Service nutzende (GUI-)Applikation. Damit diese den P2P Service nutzen kann, wird jener (zuvor) gestartet, welcher wiederum Pastry (DHT) startet, welcher wiederum seine UDP und TCP Server startet. Sollen nun Nachrichten verschickt werden, oder empfange Nachrichten verarbeitet werden, so wird diese Kette nacheinander aufgerufen. Ein Datenobjekt wird vom UDP oder TCP Server empfangen, dieser informiert Pastry, Pastry schaut abhängig vom Nachrichten Typen nach dem richtigen Handler und informiert (falls die KnotenID dafür zuständig) per deliver(msg,key) den P2PService über die eintreffende Nachricht. Der P2PService kann dann entsprechend dem Nachrichtentyp eine Benachrichtigung an die GUI geben, oder eine weitere Komponente, nämlich den FileTransferService, informieren, falls eine Nachricht um Dateiaustausch bittet. Diese Komponente ist dann in der Lage lokal gespeicherte Datensätze per TCP an anfragende Knoten zu verschicken.

2.2.2 DHT-Protokoll

(von Hans-Christoph S.)

Ein Peer-to-Peer-Netz dient dazu, Objekte innerhalb eines Netzwerkes zu verteilen und für andere Teilnehmer zugänglich zu machen. Dabei sind alle Rechner in diesem

Rechnerverbund/Overlay-Netzwerk gleichberechtigt, es existiert also keine zentrale Instanz, welche für die Verwaltung und Koordination von Daten sowie Transaktionen zuständig ist.

Daraus ergeben sich die zwei wichtigsten Aufgaben eines P2P-Netzwerkes. Zum Einen gilt es, die vorhandene Netzwerktopologie richtig abzubilden, dies bedeutet, dass neu hinzukommende sowie austretende Rechner/Knoten erkannt und die Informationen darüber im Netz propagiert werden müssen. Dazu gehört auch, dass einige, der auf den Rechnern gespeicherten Objekte, eventuell neu im Netz verteilt werden müssen. Zum Anderen sollen die verteilten Objekte bzw. die Knoten, die sie speichern, schnell und effektiv von suchenden Rechnern gefunden werden.

Für Peer-to-Peer-Netzwerke existiert eine Reihe von unterschiedlichen Implementierungsmöglichkeiten, im Mobilkommunikationsprojekt wurde das Pastry-Protokoll implementiert. Dieses Protokoll basiert auf dem Prinzip einer „Distributed Hash Table“ (DHT). Dies bedeutet, dass allen Knoten/Rechnern des Overlay-Netzwerkes eine Identifikationsnummer (ID) zugewiesen wird und die IDs der anderen Knoten lokal in den StateTables des Knoten gespeichert werden. Diese Knoten-ID wird mit Hilfe eine Hash-Funktion ermittelt und hat eine feste Anzahl von Stellen/Digits. Dabei bilden alle Knoten-IDs zusammen einen Zahlenkreis, in dem sich jeder Knoten mit seiner ID einordnet. Bei Pastry werden aber nicht nur den Knoten IDs zugewiesen, sondern auch den verteilten Objekten bzw. Dateien.

In den StateTables eines Knotens werden IDs und IP-Adressen von bestimmten Knoten gespeichert, dabei bestehen die Tabellen aus dem LeafSet, der RoutingTable und dem NeighborhoodSet. Im LeafSet werden die Knoten-IDs gespeichert, welche sich im Zahlenkreis in naher Umgebung zum eigenen Knoten befinden – also IDs bei denen die Differenz zur eigenen ID möglichst gering ist. Die RoutingTable wird dahingegen so aufgebaut, dass sich in der i-ten Zeile IDs befinden, die mit dem Besitzer-Knoten i Digits bzw. einen Präfix der Läge i gemeinsam haben. Im NeighborhoodSet werden Knoten gespeichert, die sich lokal nahe zum eigenen Knoten befinden – als Metrik kann hierbei die Anzahl der Hops zwischen den Rechnern genommen werden.

Verteilung von Objekten: In einem P2P-Netzwerk muss eindeutig definiert sein, wie die Objekte im Netz verteilt und welchen Knoten sie zugewiesen werden. Hierbei sind die erzeugten Knoten- und Objekt-IDs von Bedeutung. Ein Objekt wird dem existierenden Knoten zugewiesen, dessen Knoten-ID am dichtesten zur eigenen Objekt-ID liegt.

Suchen von Objekten: Sucht ein Knoten ein Objekt, so berechnet er lokal die ID des gesuchten Objektes, indem die vorgegebene Hashfunktion benutzt. Dann erstellt er eine Request-Nachricht (mit der eigene IP-Adresse und der gesuchten Objekt-ID). Anschließend überprüft der Knoten sein LeafSet, ob die darin enthaltenen Knoten den Wert der Objekt-ID abdecken. Ist dies der Fall, dann wurde ein passender Knoten für das Objekt gefunden, und die Request-Nachricht wird an den Knoten gesendet, dessen ID am dichtesten zur Objekt-ID ist. Wenn im LeafSet kein Knoten gefunden wurde, dann wird die RoutingTable durchsucht, und zwar nach einem Knoten, dessen Präfix zwischen der Knoten-ID und der Objekt-ID um mindestens 1 Stelle/Digit größer ist, als zwischen der ID des aktuellen Knotens und der Objekt-ID. Falls ein Knoten gefunden wurde, dann wird an diesen Knoten die Request-Nachricht gesendet. Sollte auch in der RoutingTable kein passender Knoten gefunden werden, so werden die StateTables nach einem Knoten durchsucht, dessen ID dichter an der gesuchten Objekt-ID liegt, als der aktuelle Knoten. Wenn kein Knoten gefunden werden konnte, dann ist der aktuelle Knoten der Zielknoten und somit zuständig für die Datei.

Wenn ein Knoten nun eine Request-Nachricht erhält, so überprüft er auf die oben beschriebene Weise seine StateTables und leitet die Anfrage an einen anderen Knoten weiter oder, falls er der zuständige Knoten ist, sendet er eine Antwort-Nachricht an den

Request-Absender, der dann wiederum einen Objekt-Austausch/Transfer mit dem Knoten initialisiert.

Wartung der Netz-Topologie: Für das Protokoll sind in Bezug auf die Dynamik des Netzes zwei Aspekte von Bedeutung – was soll bei neu hinzukommenden und bei verlassenden Knoten passieren.

Wenn ein neuer Knoten dem Netzwerk beitreten möchte, dann muss er zunächst seine Knoten-ID gemäß der Hash-Funktion erstellen. Anschließend sendet er an einen Knoten, der vorher definiert worden ist (z.B. durch einen Administrator oder die Anwendung in „properties.xml“) und bereits Teil des P2P-Netzes ist, eine spezielle Join-Nachricht mit seiner ID und IP-Adresse. Diese Nachricht wird wie eine normale Request-Nachricht durch das Netz geroutet, und jeder Knoten, der eine solche Nachricht erhält, sendet seine StateTables an den neuen Knoten. Der Zielknoten der Nachricht – der Knoten mit der dichtesten ID – sendet eine spezielle Welcome/Hello-Nachricht, damit weiß der neue Knoten, dass seine Nachricht korrekt verteilt wurde und erstellt seine StateTables aus den erhaltenen Tabellen. Im Anschluss sendet der neue Knoten an alle Knoten in seinen Tabellen seine StateTables, die wiederum ihre StateTables aktualisieren und den neuen Knoten einfügen. Damit ist der Knoten nun Teil des Netzwerkes.

Was passiert beim Verlassen von Knoten: Es muss angenommen werden, dass ein Knoten im P2P-Netzwerk jederzeit ausfallen kann. Um dies festzustellen, senden Knoten regelmäßige „Pings“ (DetectExistence-Nachrichten) an Knoten im LeafSet. Ein Knoten muss innerhalb einer bestimmten Zeit auf solch eine Nachricht antworten. Antwortet der Knoten nicht, löscht der Absender des Pings ihn aus seinen Tabellen und fragt einen anderen Knoten aus seinem LeafSet nach dessen LeafSet. Mit diesem erhaltenen Set aktualisiert dann der Knoten sein LeafSet.

Wird dagegen während des normalen Routings festgestellt, dass ein Knoten nicht erreichbar ist, dann stellt der Knoten eine Anfrage nach zusätzlichen Informationen an einen anderen Knoten aus seiner RoutingTable und aktualisiert damit seine eigene Tabelle.

3 Anwendungen

3.1 Datenhaltung

(von Lars R.)

Die Datenhaltung in unserem Projekt nutzt nahezu alle Möglichkeiten die Android bietet. Für die Verwaltung großer Datenmengen sind Datenbanken die beste Alternative und so gibt es in Android eine SQLite Schnittstelle. Diese Schnittstelle wird in unserem Projekt benutzt um 2 Datenbanktabellen zu erstellen. Einmal zum Speichern der verschiedenen Profile. Verschiedene Nutzer können hier ihr Profil anlegen und beliebig viele eigene Daten speichern wie auch im Android Telefonbuch. Die zweite Tabelle ist für die einzelnen Kontakte der Nutzer. In dieser werden zu den im Android Telefonbuch gespeicherten Namen der Kontakte die passenden Benutzernamen für das Netzwerk gespeichert und der Status der Kontakte.

Android hat auch eine einfache Möglichkeit geringe Datenmengen zu speichern mit den sogenannten „Shared Preferences“. Damit kann man sehr leicht XML Dateien erstellen, welche verschiedene Datentypen speichern können. Eine Speicherung erfolgt immer in der

„field-value“ Struktur und so können die Werte über einen eindeutigen Bezeichner wieder ausgelesen werden. Diese Struktur ist in unserem Projekt für die Datentypen String, int und Boolean implementiert.

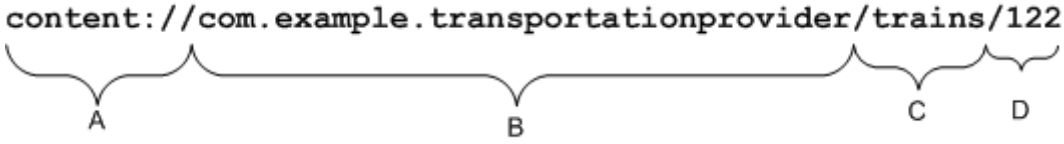
Ein weiterer großer Teil der Datenhaltung beschäftigt sich mit dem Android Telefonbuch. Hier bietet die Datenklasse unseres Projekts nahezu alle Möglichkeiten die auch das Telefonbuch selber bietet. Den Kontakten können hier beliebig viele Telefon-, Fax- und Mobilfunknummern zugewiesen werden mit allen möglichen Eigenschaften die das Telefonbuch bietet. Des Weiteren werden alle Arten von Instant Messenger Kontakten unterstützt. Zusätzlich ist es noch Möglich Nutzer in verschiedene Gruppen zu verschieben und ihnen Kontaktbilder zuzuweisen.

Die Datenhaltung in Android egal in welchem Bereich wird über sogenannte Provider geregelt. Diese Unterstützen die gängigen Operationen wie insert, delete und update. So sind sie abhängig von der zugrundeliegenden Speicherstruktur. Also egal ob SQLite Datenbanken, das Telefonbuch oder Textdateien. Zugriff auf die Provider bietet eine eindeutige URI welche frei wählbar ist und für unsere SQLite Datenbank folgendermaßen aussieht:

content://de.fu-berlin.cst.mobkomm09/mobkomm

Um nun auf die Datensätze zuzugreifen hängt man einfach den Tabellennamen an die URI an. In Unserem Fall also „/Profile“ für die Profile oder „/ContactStatus“ für die Kontakte und erhält über diese ein Cursor mit allen Datensätzen. Will man nur einen Datensatz verändern, löschen oder abfragen dann gibt es die Möglichkeit an die zusammengesetzte URI noch die eindeutige ID des Datensatzes anzuhängen z.B. „/1“.

Hier ist dazu nochmal ein Beispiel aus der Android Reference:



Wobei (A) das Standardprefix für die ContentProvider darstellt. (B) ist der Authority Part der URI hier sollte der Klassenname des Providers genutzt werden, da so die Einzigartigkeit sichergestellt wird. (C) ist der Datentyp auf dem Zugegriffen wird. Dies können SQLite Tabellen, das Telefonbuch oder die Bildersammlung sein. (D) ist die eindeutige ID eines Datensatzes.

3.2 Profilverwaltung

(von Michael D.)

Die Profilverwaltung ist eine Schnittstelle, die sowohl eine grafische Oberfläche für die Benutzerinteraktion als auch ein Verbindungspunkt alle Module (Server, Client, Datenverwaltung) ist.
Dem Benutzer stehen folgende Funktionen zur Verfügung:

1. **Veränderung eigenen Profils** – an dieser Stelle kann der Benutzer sein eigenes Profil (Telefonnummer, Adresse, Email-Adresse etc.) verändern und speichern.

2. **Kontaktliste** – in dieser werden sowohl alle autorisierte Kontakte angezeigt als auch die Möglichkeit einen neuen Kontakt, der der Service auch nutzt, hinzuzufügen zur Verfügung gestellt.
3. **Kontaktansicht** – für die Applikation relevante Daten des ausgewählten Kontakts.
4. **Autorisierungsliste** – hier werden die Kontakte angezeigt, die die Autorisierung angefordert haben. Von hier aus werden diese abgelehnt oder akzeptiert.

3.2.1 Veränderung eigenes Profils

Wird das Profil neu erstellt / verändert und anschließend gespeichert, so werden diese Daten lokal in der Datenbank abgelegt und an den Server versendet.

3.2.2 Kontaktliste

In der Kontaktliste werden alle Kontakte aufgelistet die bereits vom Benutzer autorisiert wurden oder auch diejenige bei denen der Benutzer eine Autorisierung angefordert hat.
Um einen neuen Kontakt zu der Liste hinzuzufügen ist es erforderlich seinen eindeutigen Benutzernamen zu kennen mit dem dieser sich an dem Service angemeldet hat.

3.2.3 Kontaktansicht

Zu dem Kontaktansicht gelingt man durch das anklicken einer der Namen aus der Kontaktliste. Hier werden neben den eindeutigen Benutzernamen auch der tatsächlicher Name des Kontakts, sowie auch sein Status („Authorized“, „Authorization requested“, „Uptodate“) und der Zeitpunkt letzter Aktivität angezeigt.

3.2.4 Autorisierungsliste

Die Autorisierungsliste beinhaltet die Kontakte, die an den Benutzer eine Anfrage mit der Autorisierung versendet haben. Diese befinden sich solange in der Liste bis sie entweder abgelehnt oder akzeptiert werden. Durch das anklicken eines Kontakt aus der Liste erscheinen in dem oberen Teil des Bildschirm die Tasten „accept“ und „decline“ sowie dazugehörige eindeutige Benutzername des Kontakts, der die Autorisierung angefordert hat. Akzeptiert der Benutzer die Anfrage, so wird der entsprechender Kontakt zu der Kontaktliste hinzugefügt, sein Status wird auf „authorized“ gestellt und seine Profildaten werden automatisch im Adressbuch von Android-Gerät gespeichert. Anschließend wird der Eintrag aus der Autorisierungsliste entfernt. Andernfalls wird der Kontakt nur aus der Liste entfernt.

3.2.5 Aktualisierung des Profils

Verändert einer der autorisierten Kontakte sein Profil, so werden seine Daten automatisch im Adressbuch des Android-Geräts überschrieben und in dem Kontaktansicht entsprechender Benutzer das Status auf „Uptodate“ gesetzt, sowie auch aktuelle Zeit eingetragen.

3.3 Karten und Positionierung

(von Thilo M.)

Karsten und Thilo haben nach dem Abschluss der Arbeiten an der Client-Server-Implementierung begonnen, eine zweite Anwendung für Android zu erstellen. Ihr Programm stellt einen weiteren Anwendungsfall dar und soll wie die auch erste Anwendung sowohl das Client-Server-Netzwerk als auch das P2P-Netz zum Datenaustausch verwenden können.

Anstelle des Austausches von Kontaktdaten übermittelt die Anwendung Standortdaten. Dazu wird in regelmäßigen Abständen von einem im Hintergrund laufenden Thread die Position

Abbildung 2: Kartenabbildung - Berlin, Alexanderplatz

Abbildung 2: Kartenabbildung - Berlin, Campus Freie Universität

mit Hilfe des GPS-Empfängers bestimmt. Bei genügend großer Abweichung zur vorhergehenden Messung wird die neue Position automatisch an den Server bzw. per P2P an alle autorisierten Kontakte übermittelt. Die Empfänger können sich eine Karte anzeigen lassen, auf dem die Position des Senders mit einer Markierung angezeigt wird. Auf diese Weise lässt sich verfolgen, wo sich ein Kontakt gerade aufhält.

Wie in Abbildung 2 und Abbildung 2 illustriert, wird die dafür notwendige Karte in Echtzeit auf dem Android-Gerät gezeichnet. Es werden keine Daten von einem Server heruntergeladen, sodass auch keine Verbindung zu einem Mobilfunknetz erforderlich ist.

Quelle der Kartendaten ist das OpenStreetMap-Projekt (www.openstreetmap.org), dessen Daten unter einer freien Lizenz stehen und kostenlos verwendet werden können. Die komprimierten Kartendaten sind lokal in einer Datei auf der SD-Karte oder im internen Speicher abgelegt. Das Auslesen der Kartendaten und das Rendern der Karte ist ein komplizierter und rechenintensiver Vorgang, der künftig noch beschleunigt werden muss. Die momentane Entwicklerversion ist noch langsam und fehlerhaft, zudem werden noch keine Straßennamen angezeigt.

4 Zusammenfassung und Ausblick

(von Silke R.)

Insgesamt verlief die Entwicklung an der Software sehr positiv. Die Einteilung in kleine dynamische Arbeitsgruppen hat sich als sehr effizient erwiesen. Wir haben damit erreicht, dass ein großer Teil des erarbeiteten Wissens aus speziellen Bereichen leicht untereinander

ausgetauscht werden konnte. Zum Anderen als positiv erwiesen hat sich, dass es eine gewisse Trennung zwischen Softwarearchitektur und Implementierung gegeben hat, denn es war so möglich, klare Strukturen in die Software zu bringen, da es eine zentrale Anlaufstelle für Designfragen gab.

Die Entscheidung sowohl ein Client/Server-Modell, als auch ein Peer-to-Peer-Modell zu implementieren richtig war, da so die Implementierungen anderer Komponenten frühzeitig getestet werden konnten.

Es hat sich weiterhin gezeigt, wie immens wichtig es ist, immer eine funktionierende Arbeitsversion der Software oder des gerade bearbeiteten Teilmoduls zur Verfügung zu haben, um entsprechende Funktionstest und Debuggingdurchläufe durchführen zu können. So war es in unserem Projekt sinnvoll, dass mehrere nebenläufige „Teilprojekte“ existierten, die letztendlich zu einem Ganzen zusammengeführt wurden.

Die Teilnehmer haben einstimmig beschlossen, dass das Projekt unter der Lizenz GPLv3 stehen soll.