Meetify

Meeting Assistance Application

Ingenieurpraxis an der Fakultät der Elektro- und Informationstechnik der Technischen Universität München.

Betreut von Stefan Röhrl, M.Sc.
Lehrstuhl für Datenverarbeitung

Eingereicht von Thomas Leyk
St. Nikolaus Platz 3
80939 München
thomas.leyk@tum.de

Michael Geier
Rheinstraße 33
80803 München
michael.geier@tum.de

Eingereicht am München, den 17.05.2018

Ziel der Ingenieurpraxis

Aufgabe ist es eine Android App zur Unterstützung von Meetings an Unternehmen zu entwickeln. Geplant sind mehrere Funktionen, unter anderem eine Live Agenda, ein Text Chat, Live Umfragen und ein Questions & Answers.
Entwicklungsumgebung für das Projekt ist Android Studio, eine von Google speziell für Android Apps zur Verfügung gestellte IDE. Genutzt wird die aktuellste Version 3.0.1. Installiert sind die AndroidSDK Versionen 5.0 (Lollipop) bis 8.0 (Oreo) und die Android API 27. Für den Datenaustausch zwischen den Geräten steht ein Linux-Debian Rechner zur Verfügung, auf diesem ist in C ein Server implementiert.

Aufbau der App

Wird die App gestartet so gelangt man auf eine der zwei Activities, die *SetupActivity*. Auf ihr wird zu Beginn ein Fragment angezeigt, welches eine Liste alter Meetings enthält. Über einen Button in der Actionbar oder einer Wischgeste am linken Rand gelangt man in den NavigationDrawer. Im Kopf des Menüs ist ein Platzhalter für ein Profilbild, ein Textfeld für den eigenen Namen und die E-Mail Adresse. In dem NavigationDrawer ist das aktuell angezeigte Fragment hervorgehoben und es sind weitere Fragmente aufgelistet, zum einen Kontakte und zum anderen ein Fragment für die Settings. In dem *ContactsFragment* sind eine Liste mit den Kontakten und ein Button zum Hinzufügen weiterer Kontakte dargestellt. Das *SettingsFragment* enthält Preferences zum Ändern der Einstellungen, bisher ist es hier nur möglich den Nutzernamen zu setzen bzw. zu bearbeiten. Im *MeetingListFragment* ist es über den FloatingActionButton (FAB) in der rechten unteren Ecke möglich einem bereits erstellten Meeting beizutreten. Es gibt zwei Möglichkeiten dies zu tun, einerseits kann man die zugeteilte MeetingID direkt eingeben, andererseits den dazu erstellten QR-Code scannen. Über den *new Meeting* Button werden neue Meetings erstellt. Zuerst werden ein Thema und das voraussichtliche Datum benötigt, anschließend gelangt man nach Abschluss in die *MeetingActivity*.

Die *MeetingActivity* wird für das eigentliche Meeting genutzt, auf ihr werden beim Start das *AgendaFragment* und ein Dialog mit der vom Server zugeteilten MeetingID und dem erstellten QR-Code angezeigt. Der Dialog ist weiterhin über die Actionbar aufrufbar, um die ID mit weiteren Nutzern zu teilen. Auf dieser Activity ist es möglich vier Fragmente gleichzeitig anzuzeigen. Über den FAB können weitere Fragmente, Text Chat und Live Umfragen, geöffnet und wieder geschlossen werden. Über das Textfeld im *AgendaFragment* ist es möglich neue

Gliederungspunkte hinzuzufügen. Mit einer Wischgeste nach rechts wird der geklickte Agendapunkt gelöscht. Nach einem langen Klick auf einen Agendapunkt ist es möglich die Reihenfolge zu bearbeiten oder diesen einzurücken. All diese Aktionen werden über den Server mit den anderen verbundenen Clients geteilt, um eine einheitliche und aktuelle Agenda für alle Benutzer bereitzustellen. Das *ChatFragment* enthält einen Chat wie man ihn aus anderen Apps auch kennt. In dem *LivePollFragment* werden laufende und beendete Umfragen angezeigt, welche jeweils über ein ausklappbares Textfeld zugänglich sind. Erstellen kann man diese in dem *CreateLivePollFragment*, welches in der Actionbar geöffnet werden kann. Hier werden eine Frage und mindestens zwei Antworten benötigt, um einen LivePoll zu erstellen. Die Umfrage erscheint dann bei allen Nutzern in dem *LivePollFragment* und es kann abgestimmt werden. Ein weiteres Feature, das *QandAFragment*, ist über die Actionbar aufrufbar. In diesem können Fragen gestellt werden, welche andere Nutzer beantworten. Als letztes gibt es in der Actionbar noch mal das *SettingsFragment* und einen *finishMeetingButton*. Mit dem Button wird das Meeting beendet, der Connector wird geschlossen und die *MeetingActivity* beendet.

Planung der App

Zu Beginn des Projekts wurde ein grober Zeitplan mit einer Einarbeitungsphase und den ersten Features geplant. In dem Plan sind auch aufgrund von Fehleinschätzungen Features eingeplant die nicht umgesetzt werden, z.B. Voice over IP und Videochat. Diese haben sich nach erster Recherche als zu aufwendig herausgestellt und hätten den zeitlichen Rahmen des Projekts gesprengt.

Der App-Aufbau besteht aus zwei *Activities* und einigen Fragmenten, welche auf Diesen angezeigt werden. Fragmente bieten ein sehr flexibles UI Design und können beliebig auf der *Activity* angezeigt werden, existieren jedoch nicht eigenständig. Die *SetupActivity* ist für das Geschehen rund um die App verantwortlich, um beispielsweise alte Meetings anzuzeigen. In der *MeetingActivity* soll es sich nur um das Meetings selbst drehen, d.h. sie kann nur geöffnet werden, wenn ein Meeting erstellt oder einem bereits existierenden beigetreten wird. Nach Beendigung eines Meetings wird es auf der SD-Karte gespeichert und in der *MeetingList* auf der *SetupActivity* angezeigt. Es ist möglich diese alten Meetings zu betreten und alle Daten noch mal durchzusehen, jedoch kann man diese nicht mehr bearbeiten. Über den FAB in der *MeetingActivity* sind Polls, Agenda und Chat aufrufbar, welche auch gleichzeitig auf dem Bildschirm angeordnet werden können, nebeneinander bzw. untereinander. Die anderen Features, InviteUsers, CreateLivePoll und Q&A, zugänglich über die Actionbar, hingegen können nur alleine auf dem Bildschirm stehen. Über diese Features wird nicht direkt mit

anderen Nutzern interagiert, bzw. werden diese nur selten benutzt, daher wurde die Designentscheidung getroffen für diese nur den ganzen Bildschirm zu nutzen.

GUI Design

Jede Activity oder Fragment benötigt ein Layout, welches die UI definiert. In den Layouts existiert eine Hierarchie zwischen Views und ViewGroups. Während ein View Elemente auf dem Bildschirm zeichnet, ist eine ViewGroup nur ein unsichtbarer Container, welcher die Layout Struktur definiert und Views, bzw. auch andere ViewGroups, enthält.
Als ViewGroup wird in den meisten Layouts das Linear-, Relative- oder ConstraintLayout verwendet. Jedes dieser Layouts hat Vor- und Nachteile, bei dem LinearLayout zum Beispiel werden Views entsprechend dem XML Code auf einer Linie angeordnet, wohingegen beim RelativeLayout die Positionen relativ zu den anderen Views angeordnet werden müssen. Die ViewGroups können jede Menge verschiedener Views enthalten, beispielsweise Text-, EditTextViews, Buttons und ListViews etc.
Für jede Liste wird zusätzlich noch eine XML Datei mit einem List Item benötigt. In diesem wird das Layout der einzelnen Zeilen definiert. Für dieses Projekt wird für die Implementierung der Listen ein RecyclerView benutzt.
RecyclerView ist der Nachfolger zum ListView und bietet eine flexiblere und effizientere Umsetzung. Über individuell anpassbare ViewHolder ist es möglich die Liste nach eigenem Bedarf zu modifizieren.
Benutzte Farben und Strings sind in dem values Ordner enthalten und können ohne großen Aufwand bearbeitet werden.

Serveraufbau und Funktionen

Vom Lehrstuhl wurde ein Server zur Verfügung gestellt der folgende Spezifikationen besaß
ajax-der-kleine 4.9.0-6-amd64 #1 SMP Debian 4.9.88-1 x86_64 GNU/Linux
und unter dem Hostnamen *ajax-der-kleine.clients.ldv.ei.tum.de* auch von außerhalb des

Lehrstuhlnetzes erreichbar war. Allerdings waren zum Zeitpunkt der Entwicklung alle Ports, mit Ausnahme des ssh-Port (22) blockiert. Ein Testgerät muss sich also im LDV-CPP WLAN-Netzwerk befinden um eine Verbindung aufbauen zu können. Es wurden Putty und PuttySCP genutzt um auf den Server zuzugreifen und den Quellcode von den Windows Maschinen zum Server zu kopieren.

Der Server ist ausschließlich in C programmiert, der verwendete Compiler ist gcc. Der Source Code ist auf mehrere Dateien verteilt, wobei jede Datei einem Modul entspricht und eine Gruppe von Funktionen zur Verfügung stellt (Network, MemBuf, Support, etc.). Die Komponenten werden über ein Makefile miteinander verbunden und zu der finalen Binärdatei „Server“ kompiliert.
Sobald ein Endgerät eine TCP-Verbindung mit dem Server aufgebaut hat, wird ein Challenge-Response Protokoll abgewickelt um sicherzustellen, dass kein unbefugter Zugriff auf den Server stattfindet. Wenn der Client die Challenge nicht richtig beantwortet, bricht der Server sofort die Verbindung ab. Ansonsten hat der Client die Möglichkeit eine der folgenden Anfragen (Request) an den Server zu stellen.

CREATE	Ein neues Meeting wird auf dem Server erstellt, dazu wird ein neuer *MeetingThread* gestartet, welcher alle weitere synchrone Kommunikation übernimmt.
JOIN	Ein neuer Client tritt einem bereits existierendem Meeting bei, dazu wird das entsprechende Socket in der *LinkedList* der Clients aufgenommen
REJOIN	Sollte die Verbindung zum Server irgendwie unterbrochen werden, z.B. der User beendet die App, gibt es die Möglichkeit später wieder unter demselben Namen (ID) dem Meeting beizutreten.

Alle laufenden Meetings werden in einer *HashMap* verwaltet, die Größe ist zum Kompilierzeitpunkt durch ein *#define* festgelegt. Dadurch ist es möglich, auch bei theoretisch unendlich vielen Meetings, eine konstante Zugriffszeit für Join/Rejoin zu garantieren, solange die *HashMap* eine ausreichende Größe hat. Wenn die Anfrage bearbeitet ist wird der Client in ein Meeting eingefügt, entweder wird ein Element an das Ende der *LinkedList* gehängt (Create/Join), oder ein Element der Liste wird aktualisiert (Rejoin). So wird garantiert das der Hauptthread, welcher nur dafür zuständig ist Anfragen zu verarbeiten, niemals Speicher modifiziert, welcher auch von einem *MeetingThread* modifiziert wird. Es gibt also keine Notwendigkeit die vielen verschiedenen Threads zu synchronisieren.

MeetingThread

Der *MeetingThread* verwaltet jeweils ein bestimmtes Meeting indem er aus der Liste aller Clients die sich im Meeting befinden, diejenigen die aktuell eine Verbindung haben herausnimmt und zusammen mit dem Systemaufruf *poll()* auf eingehende Pakete wartet. Die Clients können so Befehle (*Commands*) an den Server schicken welcher diese der Reihe nach abarbeitet und alle Änderungen, die an den Komponenten des Meetings (Agenda, Chat, etc..) vorgenommen wurden, an alle aktuell verbundenen Clients weitergibt. Die Clients können folgende Befehle an den Server schicken.

AGENDA_ADD	Hinzufügen eines neuen Agendapunktes
AGENDA_REMOVE	Entfernen eines Agendapunktes
AGENDA_REORDER	Vertauschen von zwei Agendapunkten
AGENDA_INDENT	Einrücken eines Agendapunktes
CHAT_SEND	Senden einer Nachricht im Gruppenchat
POLL_CREATE	Erstellen eines neuen LivePolls
POLL_VOTE	Abgeben der Stimme zu einem bestimmten LivePoll
POLL_FINISH	Beenden eines LivePoll
UPDATE	Aktualisieren einer bestimmten Komponente
UPDATE_ALL	Aktualisieren von allen Teilen des Meetings (Misc, Clients, Agenda, Chat, Polls, QanA)
ADD_QUESTION	Hinzufügen einer Frage in dem QandA
DELETE_QUESTION	Entfernen einer bestimmten Frage aus dem QandA
ADD_ANSWER	Hinzufügen einer Antwort zu einer bestimmten Frage
DELETE_ANSWER	Entfernen einer bestimmten Antwort von einer bestimmten Frage

Module

- Network.c:

Das Netzwerkmodul stellt 2 wichtige Funktionalitäten bereit:

1. Das Initialisieren eines neuen Server-Socket welches auf eingehende Verbindungsanfragen von Endgeräten wartet.
2. Ein passendes *Recv*/*Send* Gegenstück zu den *Send*/*Recv* Methoden des *ConnectorSocket*

Die im C-Standard verfügbaren Funktionen arbeiten nur mit Byte-Arrays und garantieren nicht, dass alle Bytes aus einem gegebenen Buffer verschickt werden. Um diesen beiden Probleme einzukapseln stellt das Netzwerkmodul neue *Send*/*Recv* Funktionen bereit welche

einen *MemBuf* komplett verschicken, und empfangene Daten korrekt in einem gegebenen *MemBuf* speichern. Es wird eine Symmetrie zwischen einem *Send* und einem *Recv*-Aufruf, insofern dies möglich ist, garantiert.

- MemBuf.c:

Da häufig das Problem auftritt komplexere Daten wie Integer und Strings als Bytes zu speichern werden diese Funktionen von dem *MemBuf* bereitgestellt. Dieser bietet die Möglichkeit Strings beliebiger Länge und Integer beliebiger Wortbreite zu speichern, ohne über Bufferüberläufe oder Endianness nachdenken zu müssen. Der Buffer wächst bei Bedarf automatisch, alle Integer werden als Big-Endian gespeichert und alle Strings im UTF-8 Format.

- ArrayList.c:

Das Speichern und Verwalten von einer großen Menge Daten variabler Größe und Zahl wird über eine ArrayList realisiert. Diese ist im Prinzip nichts weiter als ein Array, stellt jedoch die für eine Liste typischen Funktionen (add, insert, remove, etc.) zusätzlich zu Verfügung. So kann die Effizienz eines Arrays mit der Bedienbarkeit einer Liste verbunden werden. Darüber hinaus werden oft sogenannte *flexible array members* genutzt um in einer C-struct zusätzlich einen String zu speichern, so z.B. bei einem *AgendaItem* oder einer *ChatMessage*, dadurch verbessert sich die Speicherlokalität des Servers.

- Security.c:

Alle sicherheitstechnischen Funktionen werden in diesem Modul implementiert. Zurzeit ist das nur die *ChallengeResponse*, es könnte jedoch noch um eine eigene One-Way-Hashfunktion sowie ein Diffie-Hellman-Schlüsselaustausch-Protokoll erweitert werden.

Netzwerkimplementierung der Android Seite

Das Gegenstück zu dem Server, auf der Android-Seite, ist der *NetworkConnector*. Dieser stellt Methoden zur Verfügung um die einzelnen REQUEST und CMD Pakete an den Server zu schicken. Dazu wird über das *ConnectorSocket* eine Verbindung zum Server aufgebaut, dann werden mithilfe der *Send*/*Recv* Methoden Pakete zwischen Client und Server ausgetauscht.

Eines der größten Probleme unter Android ist die strikte *ThreadPolicy* für den UI-Thread, welche es nicht erlaubt Netzwerkoperationen auf diesem durchzuführen. Andererseits ist es

nicht erlaubt UI-Elemente aus einem Hintergrund-Thread heraus zu verändern. Aus diesem Grund verwaltet der *NetworkConnector* einen *HandlerThread*. Netzwerkoperationen werden sequentiell auf einem neuen Thread ausgeführt, dazu verwaltet dieser intern eine *MessageQueue* welche über einen *Handler* ansprechbar ist. Zusätzlich wird ein weiterer *Handler* verwendet welcher an den UI-Thread gebunden ist, um Updates des UI vorzunehmen, wenn eine Änderung über das Netzwerk eingetroffen ist.

Das Netzwerk verwendet einen einzigen *MemBuf* um Daten zu serialisieren, dieser wird gelegentlich auch von anderen Threads modifiziert, z.B. wenn der UI-Thread Änderungen entgegen nimmt. Aus diesem Grund ist der Zugriff auf den Buffer über einen simplen boolean-Lock synchronisiert.

Typischerweise wartet der Netzwerkthread auf ein Update vom Server bis ein bestimmter Timeout abgelaufen ist, danach prüft er ob eine Nutzerinteraktion eine Serveranfrage benötigt und führt diese gegebenenfalls aus. Wenn neue Informationen vom Server ankommen, werden diese auf dem Netzwerkthread empfangen und gebuffert, danach werden sie an den UI-Thread weitergegeben, um dort an den entsprechenden Stellen angezeigt zu werden. Während der UI-Thread die Daten verarbeitet ist der Netzwerkthread inaktiv.

Fortführung des Projekts

Einige mögliche Erweiterungen wären z.B. ein Account-System welches es ermöglicht andere Nutzer über Firebase-Cloud-Messaging in ein Meeting einzuladen, oder das Hinzufügen weiterer Features, beispielsweise ein Whiteboard, VoIP oder sogar einen Videochat. Eventuell wäre es hilfreich zur Implementierung dieser Features den Server zumindest auf C++ umzustellen. Weiterhin bestände die Möglichkeit weitere Einstellungen hinzuzufügen, um die Oberfläche individuell anpassbar zu machen, z.B. Farb-Paletten, Schriftart und Größe, Einführung zu den Features beim ersten Start der App, etc.

Grobe Arbeitsaufteilung

Thomas	Michael
Activities und Fragmente	
Netzwerk und Protokoll	GUI Design
Server	Speicherung der Meetings auf der SD Karte
Support Klassen	Custom Views

Glossar:	
Activity:	Komponente einer App welche eine Funktion/Dienstleistung implementiert.
Android-SDK:	Von Google bereitgestelltes Software Development Kit für die App-Entwicklung in Java
ChallengeResponse	Verfahren zu Authentisierung von einem Kommunikationsteilnehmer bei einem Anderen.
Flexible array members	Eine in der Programmiersprache C genutzte Technik um Elemente variabler Länge in einer struct zu speichern
FloatingActionButton (FAB):	Android Design Widget, ein runder Button der über der restlichen Oberfläche zu schweben scheint, und die primäre Aktion darstellt.
Fragment:	Kleiner Teil einer (oder mehr) Activities, welcher nach Bedarf gezeigt/versteckt werden kann.
Handler	Ein Handler dient dazu neue Nachrichten in eine MessageQueue hinzuzufügen oder bestehende Nachrichten zu entfernen.
HandlerThread	Hilfsklasse von Android zum Umgang mit einem langlebigen Thread. Ein HandlerThread hat jeweils einen Looper mit einer zugehörigen MessageQueue
HashMap	Datenstruktur in der Elemente basierend auf einem künstlich berechnetem/festgelegtem Hash indiziert werden.
LinkedList	Datenstruktur in der jedes Element nur seine direkten Nachbarn kennt, also keine globale Struktur vorherrscht.
Looper	Ein Looper arbeitet sequentiell alle Aufgaben die in seiner MessageQueue liegen ab.
MessageQueue	Ein MessageQueue enthält Aufgaben für einen Looper, sie kann über einen Handler modifiziert werden.
NavigationDrawer:	Android Design Widget, einwischbares links- oder rechtsseitiges Navigationsmenü.
ThreadPolicy	Die Rechteverwaltung der Threads im Android-System
UI-Thread	Der Hauptthread eines Android-Prozesses, der erste von der JVM gestartete Thread.
View	Zeichnet ein Element auf den Bildschirm.
ViewGroup	Unsichtbarer Container, der die Layoutstruktur definiert und andere ViewGroups oder Views enthält.