BonfireChat Dokumentation

- Datenstrukturen
  - Identität
    - IPublicIdentity
    - Public Key Infrastruktur
    - Identity
    - Contact
  - Nachrichten
  - Unterhaltungen
- Nachrichtenübertragung
  - Senden und Empfangen
    - ConnectionManager
    - SocketProtocol
  - Datenstruktur Umschlag
  - verschlüsselt übertragene Daten
  - Traceroute
- Kryptographie: Bibliothek libkalium
- Protokolle
  - BluetoothProtocol
  - WiFiProtocol
  - GcmProtocol: Client-Server-Architektur
- Routing
  - Flooding
  - Geplante Mechanismen
- Kontaktaustausch
  - serverbasierte Suche
  - direkter Austausch (NFC oder QR-Code)
- lokale Datenhaltung: SQlite
- serverseitige Datenhaltung
  - MySQL-Datenbank
  - API
    - POST /register.php
    - GET /search.php
    - POST /traceroute.php
    - GET /traceroute.php
- User Interface Design
  - Menü und Activities
  - Nachrichten

- Kontakte bearbeiten
- eigene Identität teilen

Datenstrukturen

Im Package `bonfirechat.models` befinden sich alle Klassen, mit denen die verwendeten Datenstrukturen dargestellt werden. Diese werden im Folgenden vorgestellt.

Identität

Zunächst wird eine Person beschrieben. Dabei handelt es sich momentan auch um ein eindeutig indentifiziertes Gerät. Eine Person kann also die gleiche Identität nicht auf mehreren Geräten verwenden. Dies könnte in Zukunft noch geändert werden. Zur Darstellung einer Identität existie die folgenden Klassen:

IPublicIdentity

Das Interface `IPublicIdentity` fasst alle Daten zusammen, die zur Identitfikation einer Person unbedingt erforderlich sind. Es abstrahiert von Daten, die für die konkrete Speicherung der Perso auf einem Gerät notwendig sind, und enhält keinerlei private Informationen. Daher kann eine Inst von `IPublicIdentity` problemlos zusammen mit einer Nachricht verschickt werden, reicht aber um keine Doppeldeutigkeiten bei der Zuordnung der Person zu verursachen.

Folgende Funktionen sind vorhanden:

```
MyPublicKey getPublicKey();
String getNickname();
String getXmppId();
String getPhoneNumber();
```

Public Key Infrastruktur

Unbedingt notwendig zur Identitfikation einer Person ist der Public Key. Wir setzen Public-Key-Kryptographie ein, um Nachrichten verschlüsselt zu übertragen und die Authentizität des Absend zu verifizieren. Für jeden neuen Benutzer wird daher beim Erstellen der Identität ein Schlüsselpaa generiert. Genaue Informationen zur Erstellung finden sich im Abschnitt *Kryptographie*. Es hande sich um einen 32 Byte Schlüssel für Curve25519.

Die Klasse `MyPublicKey` erweitert die Basisklasse `PublicKey` um einige nützliche Funktionen. Dazu gehören:

```
public byte[] asByteArray();
public String asBase64();
public static MyPublicKey deserialize(byte[] publicKey);
```

```
public static MyPublicKey deserialize(String base64publicKey);
```

Identity

Um abzubilden, dass der Benutzer über seine eigene Identität mehr Informationen verwalten mus als über fremde Kontakte, wird `IPublicIdentity` von zwei Klassen implementiert, die diese beid Fälle abdecken. Die Klasse `Identity` stellt dabei die eigene Identität dar. Ein Objekt dieser Klass wird beim ersten Start von BonfireChat erzeugt und serverseitig registriert.

Folgende Felder sind vorhanden:

```
public String nickname, server, username, password, phone;
public MyPublicKey publicKey;
public PrivateKey privateKey;
public long rowid; // Zeilen-ID in der Datenbank
```

Da insbesondere der PrivateKey in dieser Klasse gespeichert wird, darf eine Instanz von `Identity` natürlich nicht versendet werden.

Contact

Für fremde Kontakte, mit denen Nachrichten ausgetauscht werden, gibt es die Klasse `Contact`. Auch Contact implementiert `IPublicIdentity`. In Contact-Instanzen können auch Daten darübe gespeichert werden, wie der Kontakt momentan erreichbar ist. Dazu gehören z.B. WiFi- und Bluetooth-Adressen.

Eine Contact-Instanz bietet folgende Felder:

```
private String nickname, firstName, lastName, phoneNumber;
public MyPublicKey publicKey;
public String xmppId; // Daten über die Erreichbarkeit des Kontakts. Diese werden momentan nic
public String wifiMacAddress;
public String bluetoothMacAddress;
public long rowid; // Zeilen-ID in der Datenbank
```

Nachrichten

Eine einzelne Nachricht wird als Instanz der Klasse `Message` dargestellt. Dabei stellt das Message Objekt die Nachricht dar, nachdem sie empfangen oder bevor sie gesendet wurde. Es handelt sic also um die Repräsentation auf einem Gerät, nicht um die Daten, die versendet werden. Für dies Zweck würde eine Instanz der Klasse `Envelope` verwendet werden, welche genauer im Kapitel*Nachrichtenübertragung* beschrieben wird.

Eine Nachricht besteht aus den folgenden Feldern:

```
public UUID uuid;
```

```
public Date sentTime;
public IPublicIdentity sender;
public List<Contact> recipients;
public String body;
public String transferProtocol;
public String error;
public int flags;
```

Die `uuid` identifiziert eine einzelne Nachricht eindeutig. Diese wird insbesondere verwendet, um überprüfen, ob eine Nachricht schon mehrfach empfangen und/oder gesendet wurde, und als eindeutige Kennzeichnung der Nachricht in der Datenbank. Die UUID wird beim Erstellen einer neuen Nachricht mit `UUID.randomUUID()` zufällig generiert.

Der `sender` ist als Instanz von `IPublicIdentity` implementiert, da hier je nachdem, ob die Nachricht versendet oder empfangen wurde, unterschiedliche Dinge eingetragen werden: ist ma selbst der Absender der Nachricht, steht das eigene `Identity` Objekt im Absender. Empfängt m die Nachricht, wird aus dem `sender` Feld des empfangenen `Envelope` Objekts eine Instanz von `Contact` erzeugt.

Mit diesem Mechanismus wird auch festgestellt, ob eine Nachricht empfangen oder gesendet wurde. Das ist unter Anderem für die visuelle Darstellung in den UI Klassen notwendig. Für dieser Zweck existiert die folgende Funktion:

```
public MessageDirection direction() {
    return (sender instanceof Identity) ? MessageDirection.Sent : MessageDirection.Received;
}
```

Folgende Werte sind für `MessageDirection` möglich:

```
public enum MessageDirection {
    Unknown, // wird momentan nicht verwendet
    Sent,
    Received
}
```

Im Feld `transferProtocol` wird das verwendete Protokoll in Textform gespeichert, um den Benutzer zu informieren, wie seine Nachricht angekommen ist. Tritt beim Versenden eine Except auf, wird die Fehlermeldung im Feld `error` abgelegt und als Nachrichtentext angezeigt. Das ist v Allem während der Entwicklung zur Fehlersuche sehr hilfreich, wenn keine Zugriff auf die ADB-Lo vorhanden ist.

Im Integer `flags` werden verschiedene Informationen zur Nachricht als Flags kodiert. Folgende Flags können dabei gesetzt werden:

```
public static final int FLAG_ENCRYPTED = 4;
public static final int FLAG_PROTO_BT = 16;
public static final int FLAG_PROTO_WIFI = 32;
public static final int FLAG_PROTO_CLOUD = 64;
```

Das Flag `FLAG_ENCRYPTED` wird gesetzt, wenn die Übertragung mit Ende-zu-Ende-Verschlüsselu stattfindet. Die anderen Flags markieren, über welche Protokolle die Nachricht gesendet wurde. D können auch mehrere sein. Damit werden unter Anderem in den UI Klassen kleine Icons angezei

Unterhaltungen

Nachrichten werden in BonfireChat, ähnlich wie in anderen Instant Messengern, in Unterhaltunge mit Kontakten gruppiert. Momentan werden nur Einzelchats unterstützt, es ist also nicht möglich mehrere Kontakte als Unterhaltungsteilnehmer einzutragen. Falls in der Zukunft Mehrbenutzerch unterstützt werden sollen, müsste dies hier erweitert werden. Die anderen Datenstrukturen, insbesondere die Repräsentation einer Nachricht beim Verschicken, sind teilweise bereits darauf vorbereitet, Mehrbenutzerchats darzustellen.

Eine Unterhaltung wird als Instanz von `Conversation` dargestellt. Diese besteht aus folgenden Feldern:

```
private Contact peer;
private ArrayList<Message> messages;
public String title;
public ConversationType conversationType = ConversationType.Single;
public long rowid; // Zeilen-ID in der Datenbank
```

Die Nachrichten, die zu der Unterhaltung gehören, werden als `Message` Objekte in einer `ArrayList` gespeichert. Diese Liste enthält dabei sowohl die gesendeten als auch die empfangenen Nachrichten.

Es ist möglich, eine Unterhaltung zu benennen. Der Name wird dann im Feld `title` gespeichert. Standardmäßig wird als Titel der Name des Kontaktes, mit dem die Unterhaltung geführt wird, verwendet.

Falls Mehrbenutzerchats unterstützt werden, würde der das Feld `conversationType` auf einen anderen Wert als `Single` gesetzt werden. Die folgende möglichen Werte existieren:

```
public enum ConversationType {
    Single,
    GroupChat
}
```

Nachrichtenübertragung

Die bis jetzt behandelten Datenstrukturen haben sich alle mit der Speicherung der Daten auf ein lokalen Gerät befasst. Im Folgenden soll es nun um die tatsächliche Übertragung von Daten gehe

Senden und Empfangen

Da BonfireChat verschiedenste Protokolle unterstützt, gibt es einige Abstraktionsebenen, die die Teile der Nachrichtenübermittlung bündeln, die bei allen verwendeten Technologien gleich bleibe

ConnectionManager

Es gibt eine Instanz der Klasse `bonfirechat.network.ConnectionManager` . Diese kümmert sich darum, die einzelnen Protokolle zu verwalten und stellt die Schnittstelle zwischen den Protokolle also der Netzwerkschicht, und der Darstellung, also den UI Klassen, dar.

Der `ConnectionManager` ist ein `NonStopIntentService` . Um eine Nachricht zu versenden, könn neue `Intent` -Objekte erstellt und an den ConnectionManager gesendet werden. Dafür gibt es d folgenden statischen Wrapper-Methoden:

```
public static void sendEnvelope(Context ctx, Envelope envelope);
public static void sendMessage(Context ctx, Message message);
```

Diese erstellen den Intent und starten ihn. In der `onHandleIntent` Methode bearbeitet der ConnectionManager ihn anschließend. Dies stellt eine FIFO-Queue dar. Neue zu sendende Nachrichten werden hinten eingereiht und von einem separaten Thread abgeholt und bearbeitet

Der ConnectionManager sucht sich ein verfügbares Protokoll aus und überprüft in den App-Einstellungen, ob dieses verwendet werden soll. Anschließend wird die Nachricht an dieses Proto zum Senden übergeben. Genauere Informationen zu den Protokollen folgen im Kapitel *Protokolle*

Neben dem Senden kümmert sich der `ConnectionManager` auch um das Empfangen von Nachrichten. Dafür implementiert er eine Instanz des Interfaces `OnMessageReceivedListener` ur übergibt eine Referenz darauf den Protokollklassen. Wird von einem Protokoll nun eine Nachricht empfangen, kann es diese Instanz als Callback verwenden.

In der Methode `onMessageReceived` wird anschließend vom `ConnectionManager` geprüft, ob die Nachricht bereits bearbeitet wurde. Dies ist notwendig, da es beim Mesh-Netz-Routing dazu kommen kann, dass die gleiche Nachricht mehrmals ankommt, oder eine bereits gesendete Nachricht wieder zurückkommt. Dafür wird eine Implementierung der Datenstruktur Ringpuffer, nämlich `bonfirechat.helper.RingBuffer` verwendet. Die UUIDs aller Nachrichten, die bereits bearbeitet (also entweder empfangen oder gesendet) wurden, werden an den Ringpuffer angehängt. Beim Empfangen wird nun überprüft, ob die UUID der eingehenden Nachricht im Puff existiert, und in diesem Fall nichts getan. Der Puffer hat eine Größe von 250 Einträgen. Wird er vo werden alte Nachrichten einfach überschrieben und damit vergessen. Dies ist kein Problem, da diese Nachrichten mit sehr großer Wahrscheinlichkeit bereits angekommen sind oder wegen großem `hopCount` verworfen wurden.

Bei neuen empfangene Nachrichten wird überprüft, ob der eigene Public Key in der Liste der Empfänger vorhanden ist. Wenn das der Fall ist, wird die Nachricht in der Datenbank gespeichert, wenn nötig ein neues `Conversation` oder sogar `Contact` Objekt erstellt und die Nachricht zur Darstellung mit einem neuen Intent an die UI-Klassen übergeben.

Zusätzlich wird überprüft, ob weitere Empfänger vorhanden sind oder die Nachricht nicht für das Gerät bestimmt war. In diesem Fall wird die Nachricht weiter verbreitet, falls sie (angezeigt durch ihren `hopCount` noch nicht mehr als 20 Hops durchlaufen hat. Mehr zu dieser Technik wird im Kapitel *Routing* behandelt. Zum erneuten Senden erstellt der ConnectionManager einfach einen neuen Intent, sodass die Nachricht in die Sendewarteschlange eingereiht wird.

SocketProtocol

Auch die Kodierung von Nachrichten von Nachrichten in eine Form, die zum Senden geeignet ist, für alle Protokolle gleich. Daher existiert die abstrakte Basisklasse `SocketProtocol`, welche dies Operationen kapselt. Die notwendigen Funktionen, um einem Protokoll die eigene Identität und d Callback-Objekt für eingehende Nachrichten zuzuweisen, werden hier bereits ausformuliert.

Daneben sind die beiden Funktionen `sendEnvelope` und `recieveEnvelope` implementiert. Dami wird der Transport von `Envelope` Objekten über Streams abstrahiert. Die konkreten Protokolle müssen also nur noch ein Socket zum Empfänger öffnen und einen `OutputStream` bzw. `InputStream` erzeugen.

Das eigentliche `Envelope` Objekt wird mit `writeObject` bzw. `readObject` geschrieben und gelesen. Es wird also die Standard-Java-Serialisierung verwendet.

Datenstruktur Umschlag

Um Nachrichten auf dem Transportweg zu repräsentieren, werden sie in einen Umschlag eingepackt. Dieser wird durch ein Objekt der Klasse `Envelope` dargestellt. Damit das Envelope-Objekt im Zusammenhang mit Streams verwendet werden kann, implementiert die Klasse das Interface `Serializable`.

Die folgenden Felder sind vorhanden:

```
public UUID uuid;
public int hopCount;
public Date sentTime;
public ArrayList<byte[]> recipientsPublicKeys;
public String senderNickname; // eventuell entfernen, für Entwicklungszwecke aber sehr praktis
public byte[] senderPublicKey;
public byte[] encryptedBody; // verschlüsselt
public byte[] nonce;
public int flags;
```

Zusätzlich zur `uuid`, welche die Nachricht und damit den Umschlag eindeutig kennzeichnet, wird das Feld `hopCount` geführt. Dieses wird bei jedem Empfangen des Umschlags hochgezählt, sodass damit festgestellt werden kann, über wie viele Geräte die Nachricht an ihr Ziel gelangt ist.

Von den Empfängern werden nur die Public Keys übertragen, da diese zur Identifikation ausreich Dabei wird auch nicht das vollständige `MyPublicKey` Objekt übertragen, sondern nur der Schlüss

selbst als Byte-Array.

Der Absender wird ebenfalls durch seinen Public Key gekennzeichnet. Zusätzlich wird vom Abser der Nickname übertragen. Das ist zwar nicht unbedingt nötig, aber sehr praktisch, um in Logs od beim Empfänger ohne Anfragen an einen Server anzuzeigen, von wem die Nachricht ist.

Im Integer `flags` werden, ähnlich wie bei `Message` Objekten, verschiedene Daten zum Umschla abgelegt. Folgende Flags sind möglich:

```
public static final int FLAG_ENCRYPTED = 4;
public static final int FLAG_TRACEROUTE = 8;
```

`FLAG_ENCRYPTED` zeigt an, dass der Nachrichteninhalt verschlüsselt im Umschlag steht. `FLAG_TRACEROUTE` markiert eine spezielle Traceroute-Nachricht. Mehr dazu wird im Abschnitt *Traceroute*beschrieben.

verschlüsselt übertragene Daten

Das einzige Feld, das im `Envelope` Objekt verschlüsselt übertragen wird, ist `encryptedBody`. Dieses ist daher auch direkt ein Byte-Array und kein String. In BonfireChat werden alle Strings mit `UTF-8` Kodierung zu Byte-Arrays und zurück transformiert.

Für die Verschlüsselung notwendig ist außerdem das Feld `nonce`. Genaue Informationen, wie die Verschlüsselung funktioniert, folgen im Kapitel *Kryptographie*.

Die Felder `uuid`, `hopCount` und `recipientsPublicKeys` zu verschlüsseln, macht keinen Sinn, d dann Routing nicht mehr möglich wäre und die Nachrichten nicht ankommen könnten. `nonce` und `flags` sind für die Entschlüsselung notwendig und müssen daher ebenfalls Klartext übertragen werden.

Die Felder `sentTime`, `senderPublicKey` und `senderNickname` könnten verschlüsselt werden, da nur der Empfänger wissen muss, von wann und wem die Nachricht stammt. Dies ist momentan a noch nicht umgesetzt.

Traceroute

Um nachvollziehen zu können, über welche Geräte und Protokolle eine Nachricht versendet wurd gibt es den speziellen Nachrichtentyp Traceroute. Ein Umschlag wird mit `FLAG_TRACEROUTE` als solche markiert. Beim Senden und Empfangen eines Umschlags übergibt der `ConnectionManager` diesen an die statische Klasse `TracerouteHandler`. Dieser überprüft, ob das Flag gesetzt ist, und hängt in diesem Fall eine Zeile an den Nachrichteninhalt an. Traceroute Nachrichten werden stets unverschlüsselt übertragen, damit dies möglich ist.

In der angehängten Zeile ist vermerkt, ob der Umschlag empfangen oder gesendet wurde, über welches Protokoll dies geschah und auf welchem Gerät die Aktion stattgefunden hat.

Schließlich wird beim Empfangen einer Traceroute-Nachricht die Funktion `TracerouteHandler.publishTraceroute` aufgerufen. Diese übermittelt den Nachrichteninhalt an einen Server und ersetzt den lokalen Inhalt durch einen Link auf eine Seite, der man die Route anschauen kann. Auf diese Weise werden Traceroute-Nachrichten beim Empfänger sinnvoll dargestellt und auch der Absender kann sich unter dem Link die Route anschauen.

Kryptographie: Bibliothek libkalium

Alle kryptographischen Operationen werden durch die Bibliothek `Kalium` von AbstractJ (https://github.com/abstractj/kalium) unterstützt. Dabei handelt es sich um Java-Bindings und das Objekt `libkaliumjni.so` .

Kalium ist eine Portierung der `NaCl` Bibliothek von Daniel J. Bernstein auf Android. Diese stellt Funktionen zur Generierung von Schlüsselpaaren und zufälligen Nonces sowie zur sicheren Ver- und Entschlüsselung der Nachrichteninhalte bereit. In der `Envelope` Klasse wird eine Nachricht a die folgende Weise verschlüsselt:

```
if (encrypt) {
    Identity sender = (Identity) message.sender;
    Box crypto = new Box(new PublicKey(publicKeys.get(0)), sender.privateKey);
    envelope.nonce = CryptoHelper.generateNonce();
    envelope.encryptedBody = crypto.encrypt(envelope.nonce, envelope.encryptedBody);
    envelope.flags |= FLAG_ENCRYPTED;
}
```

Die CryptoBox ist ein Konzept von `NaCl` , welche die Verschlüsselung und Signierung einer Nachricht kapselt. Dabei wird die elliptische Kurve `Curve25519` , die sehr gute Streamverschlüsselung `xSalsa20` und `Poly1305` für Einmal-Authentifizierung verwendet. Mehr d unterhttp://cr.yp.to/highspeed/naclcrypto-20090310.pdf

Protokolle

BonfireChat unterstützt diverse Protokolle zur Nachrichtenübertragung. Um diese abstrakt zu verwenden, gibt es das Interface `IProtocol` . Diese wird von der abstrakten Basisklasse `SocketProtocol` implementiert und bietet die folgenden minimalen Anforderungen a ein Protokoll:

```
void sendMessage(Envelope envelope);
void setIdentity(Identity identity);
void setOnMessageReceivedListener(OnMessageReceivedListener listener);
boolean canSend();
```

Mit `setIdentity` wird dem Protokoll die eigene Identität bekannt gemacht, falls diese für den

Sendevorgang benötigt wird. `setOnMessageRecievedListener` erhält das Callback-Objekt, um eingehende Nachrichten an den `ConnectionManager` zu melden. Beide Funktionen werden bere in `SocketProtocol` implementiert.

`canSend` gibt zurück, ob mit dem Protokoll momentan Daten versendet werden können. Das kar zum Beispiel nicht der Fall sein, wenn keine Geräte in der Nähe sind, zu denen eine Verbindung aufgebaut werden kann.

BluetoothProtocol

Im `BluetoothProtocol` wird zunächst die Module ID definiert, welche Bonfire als Bluetooth-Dien kennzeichnet. Diese wurde zufällig erstellt und sollte vermutlich noch geändert werden.

```
private static final UUID BTMODULEUUID = UUID.fromString("D5AD0434-34AA-4B5C-B100-4964BFE3E739
```

Nach Erstellung einer Instanz startet das BluetoothProtocol zunächst einen Intent, das Handy unendlich lang sichtbar zu stellen. Dieser muss vom Benutzer akzeptiert werden. Anschließend werden mit einiger Zeitverzögerung die Bluetooth Threads initialisiert.

Der `searchDevicesThread` startet alle zwei Minuten die Suche nach Geräten in der Umgebung n Dabei wird eine Liste aller gefundenen Geräte zunächst geleert, und anschließend mit dem `BroadcastReceiver` `onDeviceFoundReceiver` wieder nach und nach befüllt.

Der `listeningThread` lauscht mit `listenUsingInsecureRfcommWithServiceRecord` nach eingehenden Verbindungen ohne vorheriges Pairing. Dafür wird in einer unendlichen while-Schleife `accept()` aufgerufen und das daraus resultierende Socket an einen neuen Thread übergeben, welcher für die Dauer der Verbindung läuft.

Dieser `IncomingConnectionHandler` erstellt aus dem Socket einen `InputStream`, empfängt mit `receiveEnvelope` den Umschlag und übergibt ihn an das `listener` Callback-Objekt.

Wenn eine Nachricht gesendet werden soll, wird in der Methode `connect` zunächst der evtl. laufende Discovery-Mode gestoppt. Anschließend wird zu jedem Gerät in der Liste der gefundene benachbarten Geräte ein Socket aufgebaut. Auftretende `IOExceptions` werden dabei ignoriert, d sie sehr oft auftreten. Zum Beispiel wird eine solche geworfen, wenn die App versucht, sich mit einem Gerät zu verbinden, auf dem BonfireChat nicht ausgeführt wird, da dann der passende Por nicht geöffnet ist.

Nachdem die Verbindungen aufgebaut wurden, wird mit `sendEnvelope` der Umschlag in jeden de erzeugten OutputStreams geschrieben. Daraufhin wird noch 50 Millisekunden gewartet, damit da Socket nicht direkt nach Ende der Daten geschlossen wird und die Daten sicher ankommen können. Anschließend werden die Verbindungen wieder getrennt und der DiscoveryMode wiede gestartet.

WiFiProtocol

TODO: nach erfolgreicher Implementierung dokumentieren

GcmProcotol: Client-Server-Architektur

Das `GcmProtocol` implementiert Kommunikation über ein Client-Server-Modell. Dabei wird Googl
Cloud Messaging verwendet, um gezielt Verbindungen zum Zielgerät über das Internet aufzubau
Über ein solches Socket wird dann der Umschlag mit den vorgesehenen Methoden übertragen.

TODO: Dokumentation erweitern.

Routing

Falls Nachrichten zu Empfängern verschickt werden sollen, die nicht direkt benachbart sind und r
einer Verbindung erreicht werden können, muss Routing eingesetzt werden.

Flooding

Momentan werden Nachrichten über ein Protokoll an alle verfügbaren Empfänger geschickt. Das
BluetoothProtocol zum Beispiel baut also zu allen Geräten in der Nähe eine Verbindung auf und
verschickt den Umschlag. Geplant ist weiterhin, nicht nur ein Protokoll zu verwenden, sondern di
Nachricht über alle verfügbaren Protokolle zu schicken.

Wenn eine Nachricht ankommt, die nicht für das eigene Gerät bestimmt ist oder noch weitere
Empfänger hat, wird sie erneut versendet. Dabei wird sie mit der gleichen Technik an die
Wartschlange der zu sendenden Nachrichten angehängt und somit also wieder an alle verfügbar
Empfänger geschickt. Außerdem wird der `hopCount` hochgezählt, und sichergestellt, dass eine
Nachricht nur 20 mal weiterversendet wird. Damit wird vermieden, dass eine Nachricht ewig im Kr
gesendet wird.

Bei diesem Flooding ist dann zu hoffen, dass die Nachricht früher oder später mindestens einmal
beim korrekten Empfänger ankommt.

geplante Mechanismen

Stumpfes Broadcasting der Nachrichten führt zu einem unnötig hohen Datenaufkommen.
Insbesondere über Bluetooth dauert das Versenden einer Nachricht recht lange, und es können
auch nicht mehrere Nachrichten gleichzeitig versendet werden. Daher ist es wünschenswert, ein
klügeres Routingverfahren einzusetzen.

Probleme bereitet dabei, dass kein Knoten Kenntnis über den momentanen Aufbau des Netzes h
und sich das Netz außerdem stetig verändert. Das liegt daran, dass die Personen sich umher

bewegen und die Signalqualität der eingesetzten Funktechnologien schwankt.

Kontaktaustausch

Um eine Nachricht zu verschicken, muss dem Sender der Empfänger bekannt sein. Das ist nötig, den korrekten Public Key in die Nachricht einzutragen und die Nachricht zu verschlüsseln. Zum Austausch von Kontaktdaten stehen mehrere Methoden zur Verfügung:

serverbasierte Suche

Neu angelegte Kontakte werden auf einem Server gespeichert. Dieser bietet auch eine Suche na Kontakten. Der Benutzer kann mit der `SearchUserActivity` nach Nicknames anderer Kontakte suchen.

In einem `AsyncTask` wird anschließend eine HTTP-Anfrage an den Server gestellt. Der folgende Aufruf wird abgesetzt:

```
BonfireData.API_ENDPOINT + "/search.php?nickname=" + URLEncoder.encode(keyword, "UTF-8")
```

Der Server antwortet daraufhin mit einem `JSON` -Array. Dieses enthält für jeden gefundenen Kont ein Objekt, welches die folgenden Eigenschaften bietet:

```
{
  "nickname". "",
  "phone": "",
  "publickey": "",
  "xmppid": ""  // wird nicht mehr verwendet
}
```

Wählt der Benutzer daraus einen Kontakt aus, wird ein entsprechendes `Contact` Objekt erstellt in der Datenbank gespeichert.

direkter Austausch (NFC oder QR-Code)

Falls kein Internet vorhanden ist, oder die beiden Benutzer sich ohnehin gegenüber stehen, kön die Kontaktdaten auch direkt untereinander ausgetauscht werden. Dafür können zum Einen die eigenen Kontaktdaten als QR-Code angezeigt werden. Mittels `startActivityForResult` und ein Integration der `BarcodeScanner` App kann dieser QR-Code auf einem anderen Handy eingescanr werden.

Zusätzlich dazu können die Kontaktdaten auch über NFC übertragen werden. Dafür wird `Android Beam` verwendet.

In beiden Fällen wird ebenfalls ein `Contact` Objekt erstellt und in die Datenbank geschrieben.

lokale Datenhaltung: SQlite

BonfireChat speichert einige Dinge lokal auf dem Gerät. Dazu gehören die eigene Identität, Kontakte, Unterhaltungen und Nachrichten. Diese Daten werden in einer `SQlite` Datenbank verwaltet. Diese verwendet das folgende Schema:

```
CREATE TABLE if not exists contacts (
  nickname TEXT,
  firstName TEXT,
  lastName TEXT,
  phoneNumber TEXT,
  publicKey TEXT,
  xmppId TEXT,
  wifiMacAddress TEXT,
  bluetoothMacAddress TEXT
  );
CREATE TABLE if not exists conversations (
  peer INT,
  conversationType INT,
  title TEXT
  );
CREATE TABLE if not exists messages (
  uuid TEXT NOT NULL PRIMARY KEY,
  conversation INT NOT NULL,
  sender INT NOT NULL,
  flags INTEGER NOT NULL,
  protocol TEXT,
  body TEXT,
  sentDate TEXT,
  insertDate INT
  );
CREATE TABLE if not exists identities (
  nickname TEXT,
  privatekey TEXT,
  publickey TEXT,
  server TEXT,
  username TEXT,
  password TEXT,
  phone TEXT
  );
```

In der Klasse `bonfirechat.data.BonfireData` befinden sich Methoden, um Objekte der Klassen `Identity`, `Contact`, `Conversation` oder `Message` zu speichern und zu laden.

serverseitige Datenhaltung

Einige Daten, insbesondere die registrierten Benutzer, werden auf einem Server gespeichert, au
den die App zugreift. Die Adresse ist momentan `https://bonfire.projects.teamwiki.net/`.

MySQL-Datenbank

Auf dem Server werden die Kontakte in der `MySQL` Datenbank gespeichert. Diese hat das folgen
Schema:

```
CREATE TABLE `users` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `nickname` varchar(100) DEFAULT NULL,
  `xmppid` varchar(100) DEFAULT NULL,
  `publickey` varchar(200) DEFAULT NULL,
  `phone` varchar(200) DEFAULT NULL,
  `gcmid` varchar(200) DEFAULT NULL,
  `last_updated` datetime DEFAULT NULL,
  `created` datetime DEFAULT NULL,
  `ip` varchar(40) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `nickname` (`nickname`)
) ENGINE=InnoDB AUTO_INCREMENT=69 DEFAULT CHARSET=utf8;
CREATE TABLE `traceroutes` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `uuid` varchar(200) DEFAULT NULL,
  `traceroute` text DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

API

TODO: Genauer dokumentieren, insbesondere erwartete Encodings.

Der Server bietet mehrere API-Endpunkte, die Antworten im `JSON` Format zurückgeben:

POST /register.php

Mit diesem Endpunkt lassen sich neue Benutzer registrieren. Folgende Daten werden
als `multipart/formdata` erwartet:

```
postdata: {
  "nonce": "",
  "publickey": "",
  "data": {
    "xmppid": "",
    "nickname": "",
    "phone": "",
    "gcmid": ""
  }
}
```

Gibt bei Erfolg ein `200 OK` zurück, ansonsten den Fehlercode `400` mit Fehlerbeschreibung.

Die Daten im Feld `data` müssen dabei mit dem korrekten Schlüssel signiert werden, was
serverseitig mit dem übermittelten `publickey` geprüft wird.

GET /search.php

Sucht nach passenden Kontakten und gibt gefundene als `JSON` -Array zurück.

Dieser Endpunkt erwartet `nickname` als `GET` -Paramter.

Die Ausgabe sieht folgendermaßen aus:

```
[
  {
    "nickname": "",
    "xmppid": "",
    "publickey": "",
    "phone": "",
    "last_updated": ""
  },
  ...
]
```

POST /traceroute.php

Dieser Endpunkt dient zum Veröffentlichen einer Route, welche mit einer Traceroute-Nachricht bestimmt wurde. Genauere Informationen hierzu stehen im Abschnitt *traceroute*.

Der Endpunkt erwartet folgende Daten als `multipart/formdata` :

```
postdata: {
  "uuid": "",
  "traceroute": ""
}
```

Anschließend wird der entsprechende Tracroute in die Datenbank eingetragen.

GET /traceroute.php

Mit `GET` kann eine eingetragene Route abgerufen und angezeigt werden. Der Endpunkt erwartet `uuid` als `GET` -Parameter und zeigt die Route einfach im Response Body an.

User Interface Design

Abschließend folgt nun noch eine Dokumentation der Designentscheidungen hinsichtlich der Benutzererfahrung. Alle UI-Klassen befinden sich im Package `bonfirechat.ui` .

Menü und Activities

Die oberste Ebene in der Hierarchie bildet die `MainActivity` . Diese wird beim Start standardmäß
ausgewählt. Sie bietet links ein Menü, welches sich durch einen Button oder durch Streichen ein

und ausblenden lässt. Mit diesem Menü können die drei Ansichten für Unterhaltungen (`ConversationsFragment`), Kontakte (`ContactsFragment`) und Einstellungen (`SettingsFragmer` angezeigt werden.

`ConversationsFragment` und `ContactsFragment` zeigen jeweils Unterhaltungen bzw. Kontakte einer Liste an. Dies geschieht über den jeweiligen `ConversationsAdapter` bzw. `ContactsAdapte` Durch längeres Tippen auf einen Listeneintrag lassen sich ein oder mehrere Einträge auswählen. der Menüleiste werden dann angepasste Aktionen angeboten. Zu diesem Zweck implementiere die Klassen einen `MultiChoiceModeListener` .

Nachrichten

Durch einfaches Tippen auf eine Unterhaltung oder das Kontextmenü der Kontakte kann man die `MessagesActivity` öffnen, welche die Nachrichten der Unterhaltung anzeigt. Es handelt sich dabei auch um eine Liste, wofür die Klasse den `MessagesAdapter` nutzt.

Nachrichten werden, je nachdem ob sie empfangen oder gesendet wurden, auf der linken oder rechten Bildschirmseite angezeigt. Unter dem Nachrichtentext befindet sich die Sendezeit sowie kleine Icons. Diese zeigen an, ob die Nachricht verschlüsselt übertragen wurde und welches Protokoll genutzt wurde.

Kontakte bearbeiten

Im Kontextmenü der Kontakte lassen sich diese bearbeiten. Dafür wird die `ContactDetailsActivity` gestartet. Damit lassen sich Namen und Telefonnummer ändern. Public Key eines Kontakts kann nicht verändert werden, da dann die Nachrichten nicht mehr entschlüsselt werden könnten.

Auch die eigene Identität lässt sich (in den Einstellungen) bearbeiten, hierzu wird die (etwas abgewandelte) `IdentityActivity` genutzt.

eigene Identität teilen

Um die eigene Identität zu teilen, klickt man im `NavigationDrawer` der `MainActivity` ganz unte auf einen entsprechenden Button, oder nutzt den Menüeintrag in der `ContactsActivity` . Es wi daraufhin die `ShareMyIdentityActivity` gestartet, welche den eigenen QR-Code anzeigt und a NFC-Kontakte lauscht.
