Team Pretzels

Entwufsdokument Lauffeuer

Version 1.0

Vitali Kaiser, Philipp Serrer, Huyen Chau Nguyen und Tobias Sturm
Betreut durch Dipl. Inform. Korbinian Molitorisz und Dipl.-Inform. David Meder
05.01.2012

1. Einleitung

In diesem Dokument wird ein Softwaresystem vorgestellt, dessen Zweck die Nachrichtenzustellung in Krisengebieten und über deren Grenzen hinaus ist. Benutzer werden in die Lage versetzt, ohne herkömmliche zentralisierte Netzwerkinfrastruktur (wie z.B. GSM), Botschaften zu versenden, die trotzdem privat und sicher sind.

Das Softwaresystem verwendet dezentrale P2P-Übertragungen um ein spontanes und nicht beständiges Netzwerk zwischen Teilnehmern auf zu bauen. Botschaften werden dazu zwischen genau zwei Geräten synchronisiert, das bedeutet, dass jeder Netzwerkteilnehmer (auch Knoten genannt) nicht nur seine eigenen Botschaften speichert und übermittelt, sondern auch Botschaften von Fremden. Durch mehrfache Synchronisierung einer Nachricht zwischen vielen unterschiedlichen Knoten, soll der Empfänger der Nachricht erreicht werden.

2. Entwurfsentscheidungen

Aus den nichtfunktionalen Anforderungen des Pflichtenhefts werden die folgenden Entwurfsentscheidungen abgeleitet. Sie betonen Eigenschaften, die das System erfüllen muss.

2.1. Leistungskriterien

Im Folgenden wird auf Leistungen des Systems eingegangen, die zwar messbar sind, für die allerdings keine konkrete Schranke angegeben wird, denn sie sollen so gut wie nur möglich optimiert werden.

2.1.1. Kommunikation zwischen Geräten - schneller Verbindungsaufbau und möglichst schnelle Übertragung

Ein wichtiges Kriterium für die schnelle Übertragung einer Nachricht vom Absender bis zum Empfänger ist, dass die Nachricht, in Form eines Pakets (siehe 5.1. Pakete), an möglichst viele andere Benutzer weiter gegeben wird. Darum muss der Verbindungsaufbau zwischen Geräten möglichst kurz dauern und die zu übertragene Datenmenge möglichst gering gehalten werden, sodass schnellstmöglich Pakete ausgetauscht werden.

Pakete sollen einzeln übertragen werden, sodass bei einem Verbindungabbruch zumindest ein Teil der Pakete übertragen wurde.

2.1.2. Effiziente Speicherung von Paketen

Die Anwendung hat den höchsten Nutzen, wenn möglichst viele Pakete sowohl übertragen, als auch gespeichert werden. Deshalb muss darauf geachtet werden, dass der Speicherverbrauch pro Paket sehr gering ist, sodass möglichst wenige Benutzer in Speicher-Engpässe kommen und so gezwungen wären, fremde Pakete zu löschen. Zusätzlich zur kompakten Speicherung von Paketen, soll die Datenstruktur zur Verwaltung, auch wenn sie sehr viele Pakete verwaltet, schnell Arbeiten können.

2.2. Verlässlichkeit

Mit diesen Kriterien wird festgelegt, wie das System in Ausnahmebedingungen reagieren soll.

2.2.1. Fehlererkennung / -korrektur bei Datenübertragung

Bei direkter Datenübertragung zwischen zwei Geräten wird durch Protokolle der einzelnen Übertragungstechniken die Fehlerfreiheit auf Bit-Ebene garantiert. Dies ist sowohl bei Bluetooth- als auch bei WLAN-Verbindungen, die von den meisten *Smartphone*-Betriebssystemen bereitgestellt werden, der Fall. Die Übertragung zwischen einem Gerät und dem Server (via Internetverbindung) erfolgt durch HTTP. Auch hier wird die Fehlererkennung bzw. -korrektur der Übertragungstechnik genutzt.

2.2.2. Datenkonsistenz

Das Verarbeiten von Paketen, während und nach dem Synchronisieren mit einem anderen Gerät, soll keine Inkonsistenzen von Paketinhalten zur Folge haben.

Sollten defekte Pakete (nicht interpretierbare Pakete) empfangen werden, sollen diese erkannt und verworfen werden.

2.2.3. Priorisierung von Paketen

Es muss davon ausgegangen werden, dass bereits bei einer kleinen Anzahl von Benutzern die Anzahl der Pakete auf jedem Gerät nach wenigen Synchronisationen bereits stark angewachsen ist. Trotz möglichst kleiner Pakete (siehe 2.2.1. Effiziente Speicherung von Paketen) ist es wahrscheinlich, dass Speicher- oder Übertragungsgrenzen (Dauer der Verbindung) erreicht werden. Darum soll jedes Gerät Pakete priorisieren um die Reihenfolge ihrer Übertragung festzusetzen und um zu entscheiden, welche Pakete verworfen werden können.

2.2.4. Fehlertoleranz

Der Hauptanwendungsfall der Anwendung sind Krisensituationen. Man muss also davon ausgehen, dass der Benutzer in einem sehr emotionalen Zustand ist und daher Fehler während der Bedienung macht.

Solche Fehler sollen von der Anwendung weitestgehend toleriert werden, sodass es nicht zu Ausfällen seitens der *Smartphone*-Anwendung kommt. Sollte der Benutzer jedoch so schwere Bedienfehler machen, dass eine Überlastung des Systems bevorstünde (z.B. zu große Nachrichten), soll die Anwendung die vom Benutzer geforderte Aktion nicht durchführen und dem Benutzer eine, für Laien klar verständliche, Fehlermeldung anzeigen.

2.2.5. Schutz vor Angriffen auf Nachrichteninhalte

Für die Verschlüsselung von Paketen wird RSA (siehe 5.2.5. Kryptographie-Implementierung) verwendet. Jeder Benutzer der Anwendung besitzt zwei Schlüssel: Einen öffentlichen und einen privaten (siehe Glossar). Der Öffentliche wird auf dem zentralen Server abgelegt, sodass andere Benutzer diesen Schlüssel auf ihr Gerät kopieren können (siehe /F040/ Kontakte synchronisieren). Mit Hilfe eines öffentlichen Schlüssels können Nachrichten so verschlüsselt werden, dass nur der Erzeuger dieses Schlüssels die Nachricht, mit Hilfe seines privaten Schlüssels, entschlüsseln kann. Zusätzlich kann ein Verfasser einer Nachricht den Inhalt signieren. Empfänger, die im Besitz des öffentlichen Schlüssels des Verfassers sind, können überprüfen, ob der Inhalt der Nachricht unerlaubt verändert wurde (Integrität) und ob es sich beim Absender um den Inhaber des Schlüssels handelt (Nicht-Abstreitbarkeit). Siehe dazu 5.7. Kryptographie.

2.2.6. Schutz vor Angriffen auf die Paketübertragung

Einige Informationen, die für die Priorisierung von Paketen notwendig sind (siehe 2.2.3. Priorisierung von Paketen), werden unverschlüsselt übertragen und werden von verschiedensten übertragenen Geräten aktualisiert. Es ist durchaus denkbar, dass ein Angreifer diese Informationen fälscht um fremde Pakete zu benachteiligen, sodass eigene Pakete schneller ans Ziel gelangen. Von solch einem Angriffsszenario wird jedoch nicht ausgegangen, um die Komplexität des Systems gering zu halten. Es ist denkbar das System zu einem späteren Zeitpunkt dahingehend zu ändern, dass es solche Angriffe erkennt und unterbindet.

2.2.7. Schutz vor Angriffen auf den Server

Wir gehen davon aus, dass der Server vom Betreiber dahingehend gesichert ist, dass Angriffe auf den *Windows Server* (siehe Glossar) nicht möglich sind.

Um unautorisierten Zugriff zu vermeiden, wird jede Schlüsselaktualisierung mit einem Bestätigungs-Code vom Benutzer bestätigt werden muss (siehe 6.6. Lauffeuer-Server).

2.3. Wartung und Erweiterbarkeit

Im Folgenden wird auf die Erweiterbarkeit (neue Funktionen) und Modifizierbarkeit (bestehende Funktionen verändern) eingegangen.

2.3.1. Erweiterbarkeit

Das System soll so gestaltet werden, dass das Prinzip der dezentralen Nachrichtenübermittlung möglichst gut für andere Anwendungen, neben Katastrophennachrichten, einsetzbar ist. Dazu wird bei dem Systementwurf darauf geachtet, dass Programmteile, die für Verschlüsselung, Paketübertragung und -verwaltung zuständig sind, so allgemein wie möglich entworfen werden. Sie sollen zum einen eine möglich einfache Schnittstelle nach außen bieten, um die geforderten funktionalen Anforderungen zu implementieren. Zum anderen soll es aber möglich sein Verschlüsselungsalgorithmen, Paketverarbeitung- und Speicherung austauschbar zu machen. Um diesem Ziel näher zu kommen, werden wiederverwendbare Komponenten in einer API (siehe Glossar) zusammengefasst und des Öfteren das Entwurfsmuster *Abstract Fabric* und *Template Method* verwendet.

2.3.2. Modifizierbarkeit

Alle logisch voneinander getrennten Subsysteme (siehe 3. Trennung zwischen Lauffeuer und Merkur) kommunizieren über Interfaces, wodurch die Implementierungen hinter den Interfaces einfacher ausgetauscht werden können.

Außerdem wird der hardwarenahe Paketübertragungsweg abstrahiert, wobei ein besonderes Augenmerk auf der Bluetooth-Schnittstelle liegt, bei der die Bluetooth-Simulation siehe (4.1. Bluetooth-Simulation) durch eine echte Bluetooth-Schnittstelle ersetzt werden kann, falls die Firma *Microsoft* eine nachträgliche Implementierung von Bluetooth einführt.

2.4. Endnutzerkriterien

2.4.1. Nutzbarkeit

Das System soll nach Möglichkeit keine Konfiguration seitens des Benutzers erfordern. Nach der Installation sollen so wenige Benutzereingaben wie möglich nötig sein, um alle relevanten Daten zu initialisieren. Befindet sich der Benutzer nicht in einer Krisensituation, soll er keinen Aufwand betreiben müssen, um z.B. seine Kontakte auf dem neusten Stand zu halten. Die Anwendung soll alle

nötigen Änderungen an Kontakten des Benutzers automatisch vornehmen, wenn dieser sein Telefonbuch ändert (siehe /F040/ Kontakte synchronisieren). Darüber hinaus soll der Benutzer nach Möglichkeit nicht mit technischen Begriffen konfrontiert werden.

2.4.2. Kontrolle über Speicherverbrauch, Übertragungen

Der Benutzer hat die volle Kontrolle darüber, wann Daten zwischen zwei Geräten übertragen werden und wie viel Speicherplatz er für fremde Pakete zur Verfügung stellt (siehe /D220/ Paket-Speicher). Der Benutzer kann die Anwendung in einen Modus versetzen, in dem sie automatisch Verbindungen mit anderen Geräten eingehen kann - was für die Verbreitung von Nachrichten in Krisengebieten wünschenswert ist. Der Benutzer hat allerdings auch die Möglichkeit die Kommunikation der Anwendung zu unterbinden, z.B. um den Akkuverbrauch zu verringern (siehe /F130/ Benutzer-Einstellungen).

3. Trennung zwischen Lauffeuer und Merkur

Das im Pflichtenheft beschriebene System wird Komponenten enthalten, die für die dezentrale Nachrichtenübertragung zuständig sind. Diese Komponenten sollen (nach den Vorgaben des Pflichtenheftes) so flexibel sein, dass sie auch für andere Softwareprojekte eingesetzt werden können, die ebenfalls mit einer dezentralen Nachrichtenübertragung arbeiten.

Um alle Komponenten, die ausschließlich für die Übertragung, Speicherung und Verwaltung von Botschaften zuständig sind, von den übrigen zu trennen, wird die API Merkur[1] entwickelt. Alle Softwarekomponenten von Merkur sollen anpassbar und wiederverwendbar sein. Die restlichen Softwarekomponenten, die nötig sind um die Anforderungen des Pflichtenheftes umzusetzen, werden als Lauffeuer, Anwendungsebene, Anwendungsschicht oder einfach Anwendung bezeichnet.

Merkur kann als eine Zusammenfassung von vielen Mechanismen bezeichnet werden, die in einer Konfiguration zusammengestellt werden, sodass eine Aufgabe, wie die Paketübermittlung für Lauffeuer, realisiert werden kann. Die Konfiguration kann von der Anwendungsebene frei verändert werden. Die Standard-Konfiguration von Merkur ist jedoch genau die Konfiguration, die im Pflichtenheft für Lauffeuer beschrieben wurde (siehe 5.8. Standard-Konfiguration).

In beiden Subsystemen (Lauffeuer und Merkur) wird nicht explizit zwischen Paketen und Nachrichten unterschieden. Im Gegensatz zum Pflichtenheft ist diese Unterscheidung nicht nötig, da bei der Realisierung der Software Nachrichten immer in Pakete gepackt werden, um eine einheitlichere Schnittstelle zu gewährleisten.

[1] römischer Götterbote

3.1. Schichtenarchitektur

Die *Smartphone*-Anwendung Lauffeuer wird in verschiedene Schichten aufgeteilt werden. Die Basis und somit eine eigene Schicht bildet die API Merkur.

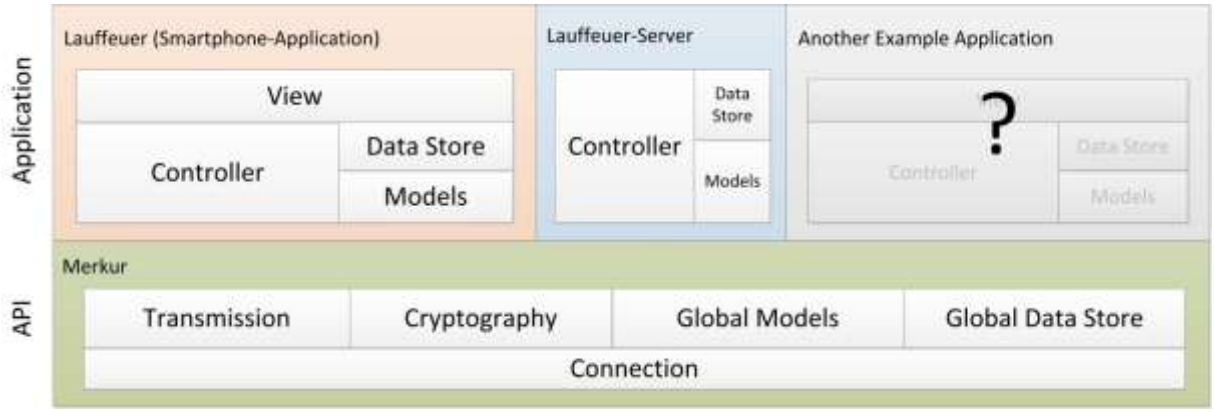

Abbildung 1 - Schichtenarchitektur

Diese Schicht benutzt die Logik, die in Merkur implementiert ist. Dort sind die Modelle (`Models`) und auch die Datenbank (`DataStore`) genauer spezifiziert (siehe 5.6. Models und DataStore). Außerdem gibt es einen eigenen *Controller*, der den genauen Programmablauf steuert (siehe 6.4. Controller). *Controller*, Modelle und die Datenbank interagieren gemeinsam mit der graphischen Benutzeroberfläche (`View`, siehe 6.3. GUI).

Nicht nur die *Smartphone*-Anwendung Lauffeuer, sondern auch der Lauffeuer-Server (siehe 6.6. Lauffeuer-Server), benutzt Merkur, da dieser ähnliche Komponenten wie Lauffeuer verwendet (z.B. auch Pakete empfangen und entschlüsseln).

An diesem Punkt ist die Notwendigkeit einer Unterteilung in API und Anwendung deutlich: Die API ist mehrfach einsetzbar und ist deswegen auch sehr allgemein gehalten. Dadurch wird es leicht, weitere Anwendung, die auch auf einem ähnlichen Prinzip wie Lauffeuer basieren. Zur Verdeutlichung wurde in Abb. 1 die Anwendung *Example* hinzugefügt, die auch, wie Lauffeuer oder der Lauffeuer-Server, auf die API zugreift und sie benutzt.

4. Einschränkungen durch Windows Phone 7

Einige wichtige, im Pflichtenheft geforderte, Funktionen des Systems sind im Rahmen der unter Windows Phone 7 (WP7) gegebenen Möglichkeiten nicht umsetzbar. Daher wird es in einigen Punkten der Anwendung Einschränkungen geben, die hier erwähnt sind.

4.1. Bluetooth

Es ist nicht möglich, dass eine Anwendung, wie Lauffeuer, auf einem WP7 die Bluetooth-Schnittstelle des *Smartphone* verwenden kann.

Um eine Bluetooth-Übertragung zu simulieren, wird von Merkur ein BTS Interface (siehe Glossar, BTSI) angeboten.

Das Interface bietet die gleiche Signatur an, wie es eine vergleichbare Schnittstelle (z.B. vom Betriebssystem *Android*) tut. Es wird die WLAN-Schnittstelle des WP7 verwendet, um die Kommunikation zwischen den Endgeräten zu gewährleisten.

Da WP7 auch kein Ad-hoc WLAN unterstützt, müssen alle Geräte, die sich in der simulierten Bluetooth Reichweite befinden sollen, auch mit demselben WLAN Access Point verbunden sein.

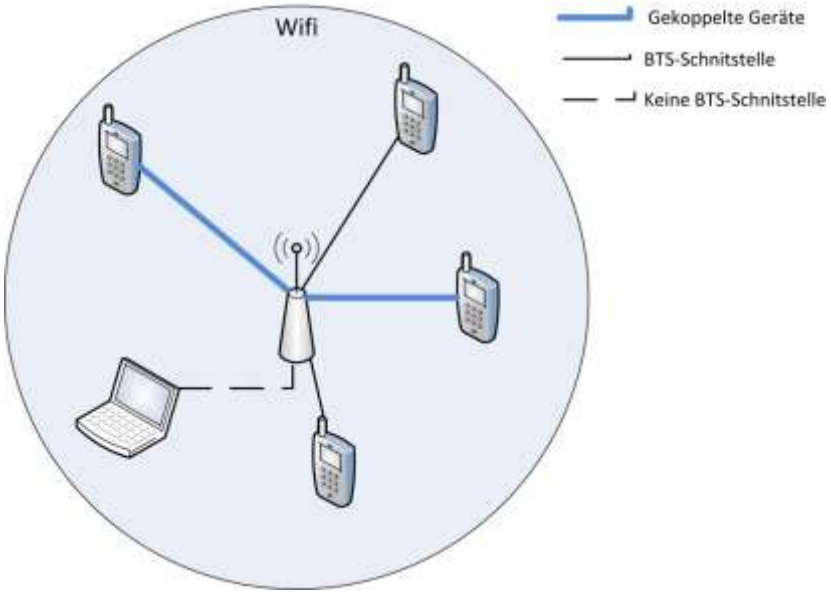

Abbildung 2 – Bluetooth-Simulation mittels WLAN

Durch einen Broadcast (siehe Glossar) wird ermittelt, welche Geräte ebenfalls die simulierte Schnittstelle implementieren. Sobald diese „gekoppelt“[2] werden, sollen diese durch einen direkten WLAN-Kommunikationskanal die Daten austauschen können.

4.2. Hintergrundprozesse

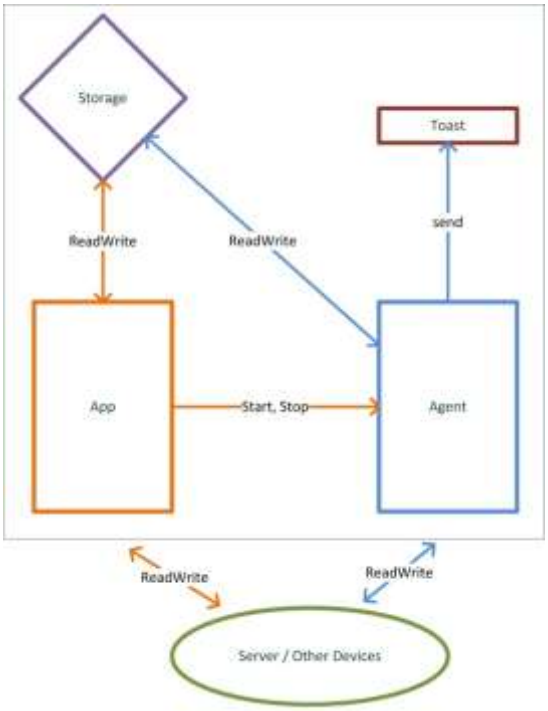

Abbildung 3 - Verschiedene Prozesse des Systems

Durch die Prozessverwaltung von WP7 ist es nicht möglich, Dienste dauerhaft laufen zu lassen, um so etwa eine spontane und automatische Übertragung zu realisieren, sobald sich zwei Geräte im gegenseitigen Empfangsradius befinden, da die Anwendung dafür im Vordergrund sein muss.

Es ist aber möglich mit einem sog. `Background-Agent` in gewissen Zeitintervallen Paket-Synchronisationen und Telefonbuchabgleiche (siehe /F040/ Kontakte synchronisieren) durchzuführen.

Für alle immer wieder anfallenden Aufgaben wird ein `PeriodicalService`, der von `BackgroundTask` erbt, implementiert, der z.B. sog. *ToastNotifications* (siehe Glossar) erzeugt oder die Kontaktsynchronisation (siehe /F040/) startet.

[2] gekoppelt: Bevor man mit Bluetooth eine Verbindung mit einem anderen Bluetooth Gerät aufbauen kann müssen diese sich für untereinander autorisieren

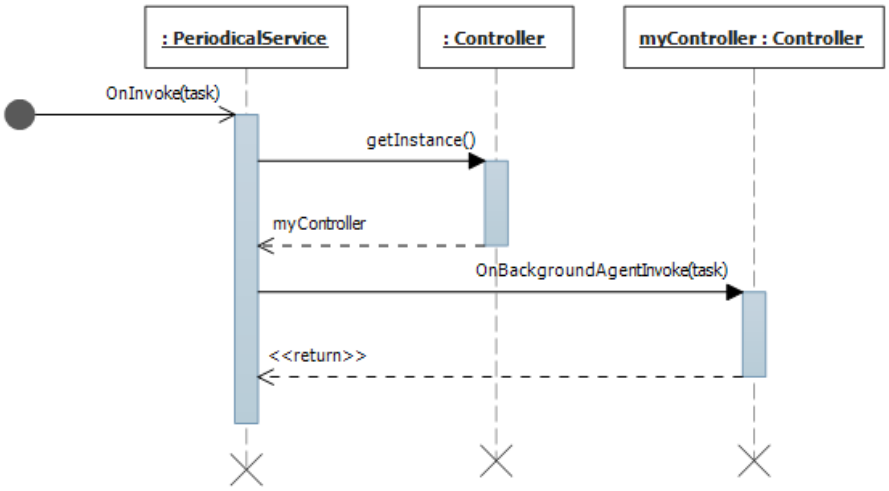

Abbildung 4 - Funktion der `PeriodicalService`-Klasse

5. Merkur

Merkur beinhaltet Mechanismen um Pakete zu übertragen und zu verwalten. Dazu bietet Merkur Möglichkeiten, die es erlaubt, je nach Anwendungszweck weitere Mechanismen hinzuzufügen, wie z.B. eine alternative Priorisierung von Paketen (siehe 5.5.2. Priorisierungsmechanismen).

5.1. Pakete

Pakete repräsentieren Dateneinheiten, die zwischen Knoten ausgetauscht werden. Es handelt sich um Botschaften zwischen zwei Benutzern mit zusätzlichen Informationen, die zur Übertragung notwendig sind (siehe 5.5. Routing). Pakete können für einen Knoten lesbar sein, oder nicht (siehe 5.7. Kryptographie). Knoten können selbständig Pakete erstellen und sie durch Synchronisationen mit anderen Teilnehmern in das Netzwerk einbringen. Der Inhalt der Pakete wird maßgeblich durch die Anwendungseben bestimmt. Durch Mechanismen wie *LayerStacks* (siehe 5.2.1. TransmissionLayer-Konzept) werden Inhalte so verändert, dass sie möglichst effizient und sicher übertragen werden.

5.1.1. Aufbau eines Paketes

Ein Paket wird durch eine Instanz der Klasse `Packet` repräsentiert. Diese besteht aus Routinginformationen (siehe 5.5. Routing) und einem weiteren *PackageContent* (siehe unten). Bei den Routinginformationen handelt es sich um einen spezieller *PackageContent* bei dem der Inhalt variabel ist und von den benutzen *Layern* (siehe 5.2.1. TransmissionLayer-Konzept) abhängt.

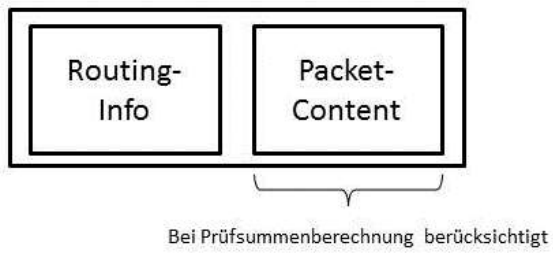

Abbildung 5 - Packet-Instanz

Jedes Paket besitzt eine eindeutige ID, die durch die Prüfsumme über den *PacketContent* bestimmt wird (siehe Abb. 5).

Die Klasse `Packet` enthält ein Array mit den Empfängern (siehe Abb. 10). Dieses Attribut wird nicht im Paket versendet, es dient nur der Informationsübertragung zwischen der Anwendungs-Ebene und Merkur.

5.1.2. PacketContents

So genannte *PacketContents* sind Inhalte eines Pakets, die mithilfe der Klasse `PacketContent` repräsentiert werden. Jede `PacketContent`-Instanz kann einen Datentyp bzw. eine Datenstruktur aufnehmen. Man kann unterscheiden zwischen Inhalten die unmittelbar durch die Anwendung, also durch Eingaben des Benutzers erstellt werden, und solchen Inhalten, die beim Durchlaufen eines *LayerStacks* (siehe 5.2.1. TransmissionLayer-Konzept) erzeugt bzw. geändert werden.

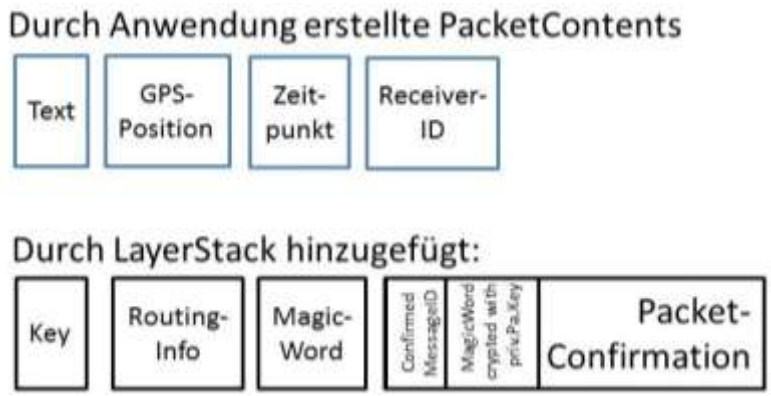

Abbildung 6 - Grundlegende `PacketContent`-Klassen, die von Merkur bereitgestellt werden

Die von Merkur unterstützten grundlegenden *PacketContents* sind:

- `TextContent` (z.B. eine Botschaft von einem Benutzer an einen anderen)
- `GPSContent` (Geo-Informationen)
- `TimeContent` (z.B. Absende Zeitpunkt)
- `KeyContent` (z.B. *public*- und *privatePacketKey* siehe 5.7.4. Pakete verschlüsselt versenden)
- `MagicWordContent` (siehe 5.7.1. Verwendung eines MagicWords)
- `ConfirmationContent` (für Empfangsbestätigungen)
- `RoutingInfoContent` (siehe 5.5. Routing)
- `ReceiverIDContent` (Angabe von Empfängern, sodass der Server Nachrichten via Email weiterleiten kann, siehe 5.4. Nachrichtenzustellung über den Server)

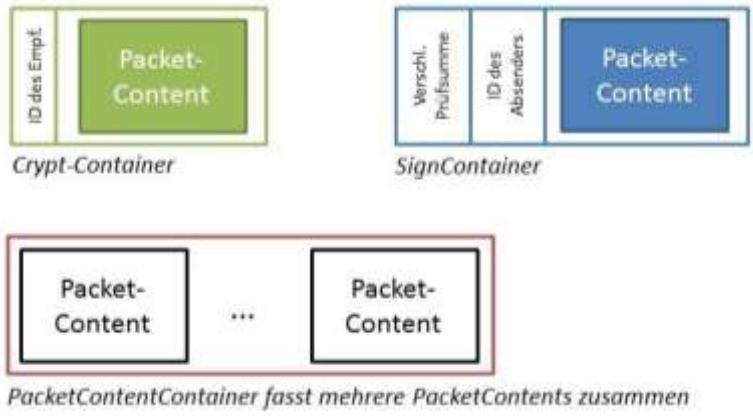

Abbildung 7 - *PacketContentContainer* enthalten andere `PacketContent`-Instanzen

Neben den acht grundlegenden `PacketContent`-Unterklassen gibt es drei weitere, bei denen es sich um sogenannte *PacketContentContainer* (oder *Container*) handelt, also Inhalte, die andere Inhalte beinhalten (siehe Abb. 7). Dazu zählen:

- `CryptContainer` (verschlüsseln einer `PacketContent`-Instanz)
- `SignContainer` (signieren einer `PacketConten`-Instanz)
- `PacketContentContainer` (vereint mehrere `PacketContent`-Instanzen zu einer einzelnen `PacketContent`-Instanz)

Alle *PacketContents* erben von der abstrakten Klasse `PacketContent`. Die Anwendungschicht kann so ggf. eigene *PacketContents* definieren, die dann von dieser einen, abstrakten Klasse, oder von einer existierenden `PacketContent`-Unterklasse erben.

Informationen einer Empfangsbestätigung sind ebenfalls ein *PacketContent.* Darum sind Empfangsbestätigungen auch gewöhnliche Pakete.

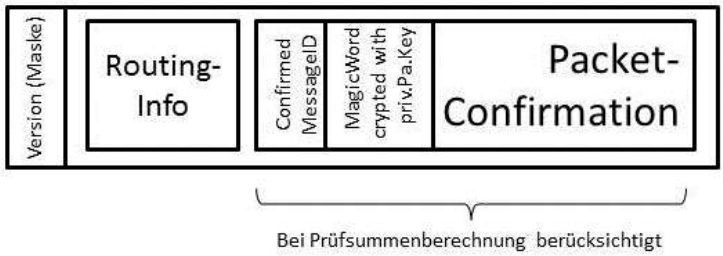

Abbildung 8 - Empfangsbestätigung als Paket

`PacketContent`-Instanzen müssen zu Byte-Arrays serialisiert und deserialisiert werden können, damit sie verschlüsselbar und signierbar sind (siehe 5.7. Kryptographie) bzw. durch einen Bytestream transportiert werden können. Darum stellt die abstrakte Klasse `PacketContent` die Methoden `Serialize` und `Deserialize` bereit, die von `Deserialize`-Methode von `Packet` aufgerufen werden (Entwurfsmuster *Template Method*).

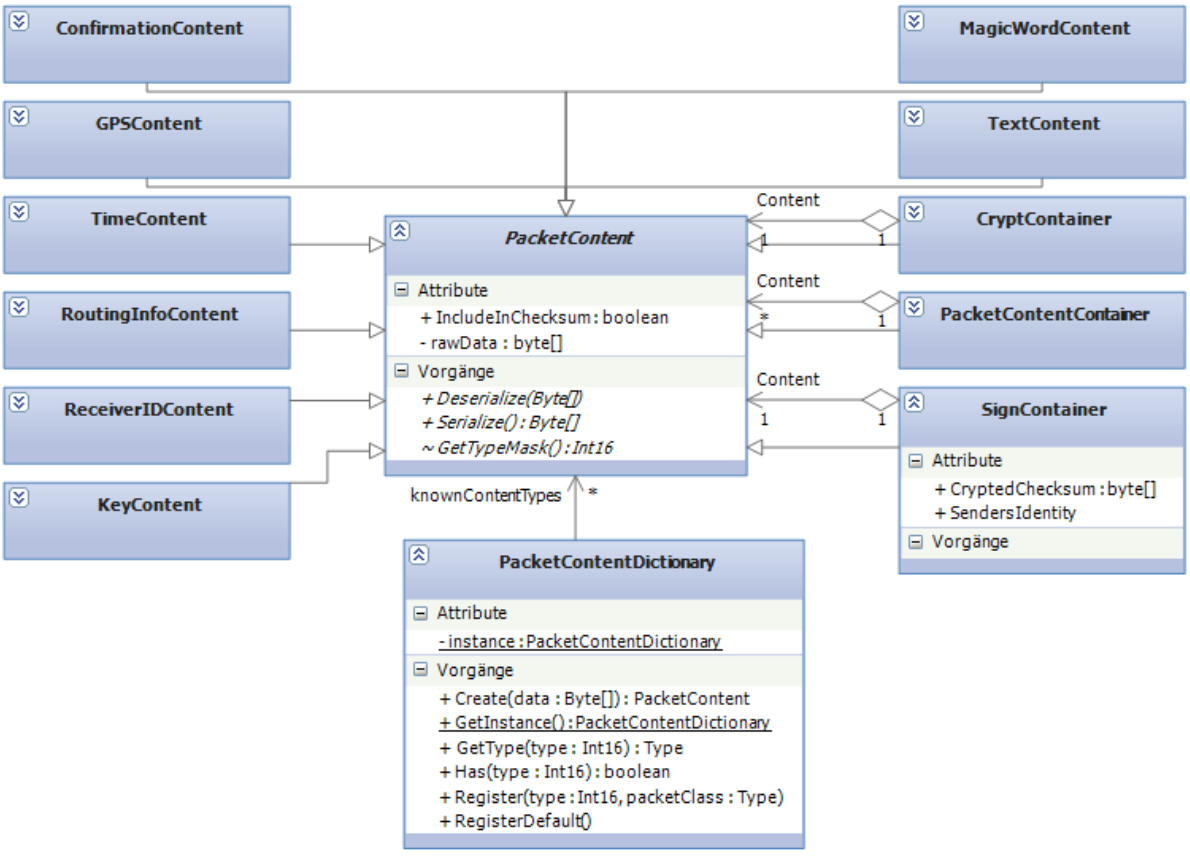

Abbildung 9 - *PacketContents* von Merkur

5.1.3. Deserialisieren von Paketen

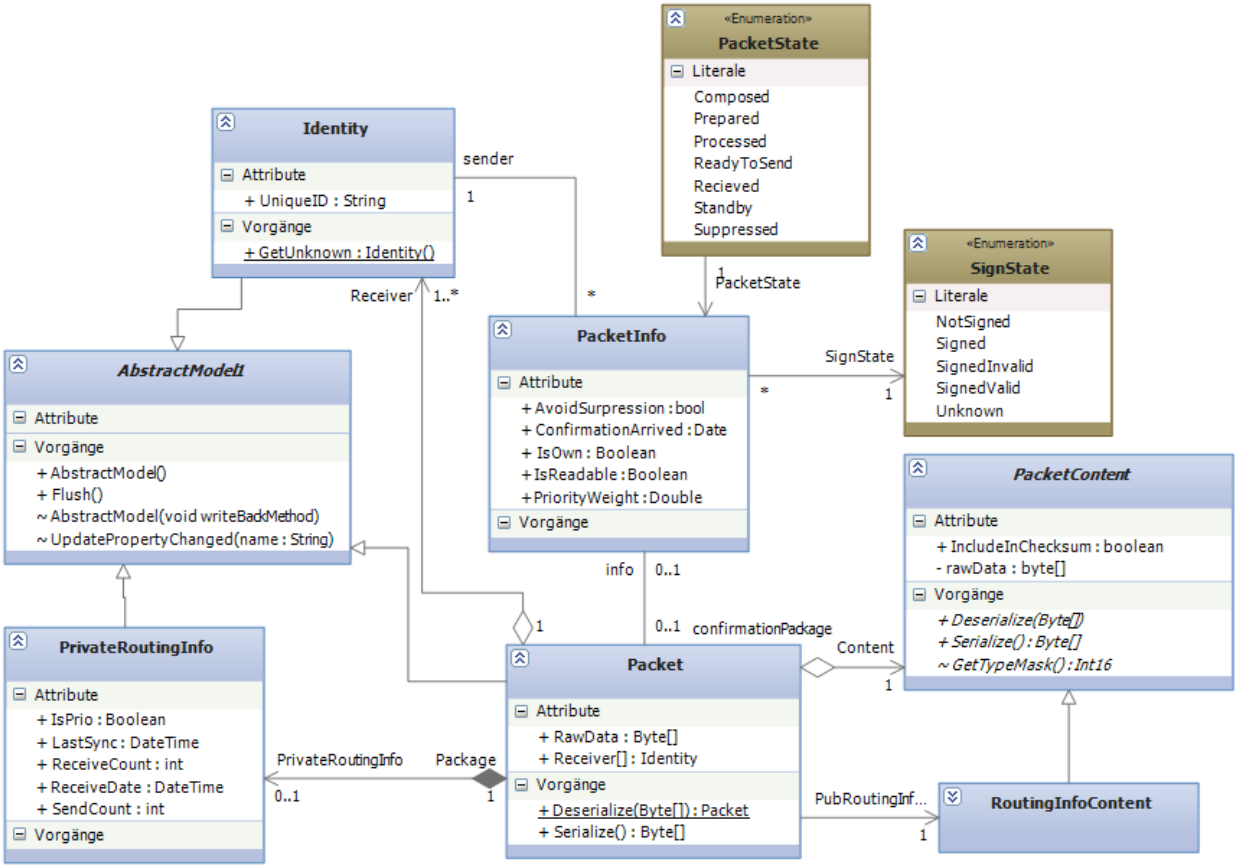

Abbildung 10 - Klassen für die Repräsentation von Paketen

`RawData` (siehe Abb. 10) sind die Daten, die beim Empfangen eines Paketes gegeben sind. Die Klasse `Packet` unterteilt bei Aufruf der Methode `Deserialize` die Daten in `RawData` in zwei Teile. Der erste Teil wird als `RoutingInfoPacketContent` interpretiert, der zweite wird als beliebiger `PacketContent` interpretiert.

Mit Hilfe der Methode `Create` der Klasse `PacketContentDictionary` (siehe Abb. 9) kann dann aus einem Byte-Array die passende, konkrete Kind-Klasse von `PacketContent` erzeugt werden.Hierzu wird zuerst aus den Daten der `PacketContent`-Typ ermittelt, der in den ersten 2 Byte des Bytestroms (interpretiert als `Int16`) steht (siehe Abb. 9). Über `GetType` wird hierzu dann die dazugehörige Klasse ermittelt und erstellt. Danach wird die `Deserialize` Methode dieser neu erstellten Klasse aufgerufen um den Inhalt aus dem Bytestrom wieder herzustellen.

Bei `PacketContentDictionary` handelt sich um ein *Singleton* (gleichnamiges Entwurfsmuster), bei dem alle verfügbaren `PacketContent`-Typen beim Start von Merkur registriert werden (siehe 5.8. Standard-Konfiguration).

Die `Deserialize`-Methode ruft sich so lange rekursiv auf, bis ein *PacketContainer* nicht deserialisiert werden kann (weil z.B. der dafür nötige Schlüssel nicht verfügbar ist) oder bis ein *PacketContent* ein grundlegender *PacketContent* ist (also keine anderen enthalten kann).

Wie ein *PacketContent* deserialisiert werden kann, ist in seiner `PacketContent`-Klasse definiert.

5.1.4. Serialisieren von Paketen

Dieser Vorgang ist nur dann nötig, wenn eine Empfangsbestätigung erstellt oder ein Paket von Benutzer verfasst wird. Alle Pakete die von anderen Knoten empfangen wurden, werden (auch wenn sie lesbar sind) in serialisierter Form gespeichert. Dieser *Tradeoff* zwischen Rechenzeit und Speicherverbrauch soll bewirken, dass Pakete möglichst schnell versendet werden können und nicht erst serialisiert werden müssen.

In welcher Reihenfolge und Kombination welche *PacketContents* ineinander geschachtelt oder verschlüsselt werden, wird durch einen *LayerStack* definiert (siehe 5.2.1. TransmissionLayer-Konzept).

Beim Serialisieren von *PacketContents* wird die `PacketContentDictionary`-Klasse nicht benötigt, da `PacketContent`-Instanzen als Parameter übergeben werden.

Die `Serialize`-Methode wird, wie die `Deserialize`-Methode, rekursiv aufgerufen.

5.1.5. Paket-Zustandsübergänge

Um den Umgang von Paketen in Merkur oder auch der Anwendungsschicht zu ermöglichen, müssen Paketübergänge definiert werden.

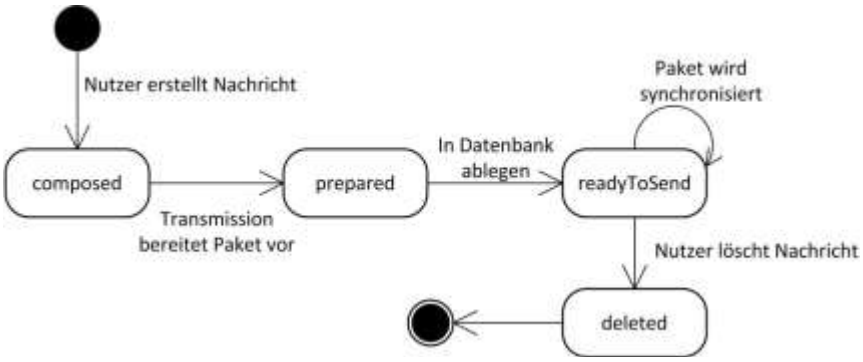

Abbildung 11 - Zustände eigener Pakete

Pakete können verschiedene Zustände haben. Zustände werden explizit angegeben, obwohl sich einige ihnen implizit durch die Daten eines Pakets bereits gegeben sind.

composed: Eine Botschaft wurde von einem Benutzer verfasst. Sie wird in Form eines *Arrays* von `PacketContent`-Instanzen zur Transmission-Instanz weitergereicht. Sie wird zum Versenden vorbereitet (siehe *PrepareLayerStack* in 5.2.1.7. DefaultLayerFactories). Sobald sie im Zustand *prepared* ist, wird sie dann in die Datenbank geschrieben und befindet sich im Zustand *readyToSync*.

readyToSync: Alle Pakete in diesem Zustand werden bei Synchronisationsvorgängen berücksichtigt. Ein Paket verlässt diesen Zustand, wenn entweder eine passende Empfangsbestätigung eintrifft (→ *deleted*) oder es bei Speicherknappheit als unwichtig deklariert wird (→ *supressed*).

deleted: Ein Paket das gelöscht wurde, wird nicht mehr gespeichert. Allerdings wird Paket-ID weiterhin gespeichert, um das erneute Empfangen und Aufnehmen zu verhindern. Ein Benutzer kann selbst verfasste Pakete löschen. Fremde Pakete können nicht explizit durch den Benutzer gelöscht werden.

standby: Pakete im Zustand *standby* werden nicht weiter verbreitet, aber auch nicht gelöscht. Trifft eine Empfangsbestätigung für eine vom Benutzer selbst versendete Nachricht ein, so wird die weitere Verbreitung der Nachricht eingestellt. Dieser Zustand macht nur für eigene Nachrichten Sinn.

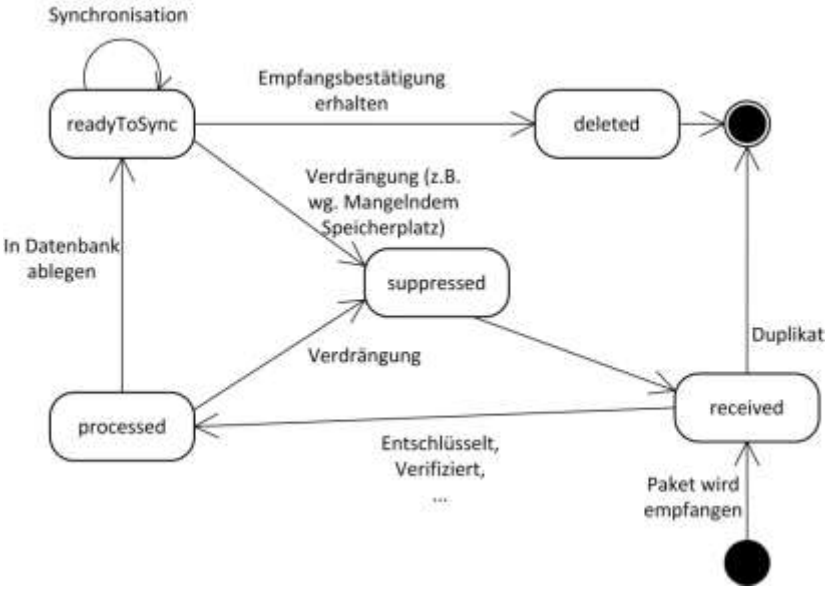

Abbildung 12 - Zustände für Pakete von anderen Knoten

received Ein Paket befindet sich in diesem Zustand, sobald es von Transfer an eine `Transmission`-Instanz übergeben wird. Während das Paket den *ReceivingLayerStack* (siehe 5.2.1.7. DefaultLayerFactories) durchläuft, bleibt es in diesem Zustand. Sollte während des Durchlaufens des *LayerStacks* erkannt werden, dass es sich um ein Duplikat handelt, so wird es sofort verworfen. Nach dem Durchlauf befindet sich das Packet im Zustand *processed*.

processed Alle Informationen die extrahiert werden konnten, sind in der `Packet`-Instanz gespeichert. Instanzen, die das `IPacketObserver`-Interface implementieren (siehe 5.2.2. Schnittstellen von Transmission), werden benachrichtigt. Sollte kein Speicherplatz für das Paket verfügbar sein, so wird es unterdrückt und befindet sich im Zustand *suppressed*. Steht ausreichend Speicherplatz zur Verfügung, wird das Paket in der Datenbank abgelegt und in den Zustand *readyToSync* versetzt.

suppressed Pakete die als "unwichtig" (niedrig priorisiert) definiert wurden siehe 5.5.2. Priorisierungsmechanismen) und darum nicht gespeichert werden konnten, befinden sich in diesem Zustand. Der Unterschied zu *deleted* ist der, dass eine Wiederaufnahme des Pakets bei erneutem Empfang nicht ausgeschlossen wird.

5.2. Transmission

Der *Namespace* `Transmission` (auch Transmission-System oder Transmission genannt) vereint alle Klassen, die zur Organisation von Paketen dienen und während bzw. nach Synchronisierungsvorgängen notwendig sind. Er stellt Mechanismen für folgende Aufgaben bereit:

- empfangene Pakete verarbeiten (Verschlüsselung, Deserialisieren, ...)
- Pakete zum Senden anhand der Priorisierung auswählen
- Pakete priorisieren
- Routing von Paketen (Anzahl der Pakete im gesamten System optimieren)

Transmission beinhaltet wiederum den *Namespace* `Crypto` (siehe 5.2.5. Kryptographie-Implementierung). Außerdem verwendet das Transmission-System die Schnittstellen aus dem *Namespace* `Transfer` (siehe 5.3. Transfer).

5.2.1. TransmissionLayer-Konzept

Die Schnittstelle die Merkur einer Anwendung zur Verfügung stellt, soll möglichst variabel verwendbar sein. Um diese Vorgabe zu erfüllen, wird das im Folgenden beschriebene Konzept verwendet.

5.2.1.1. Zweck

Während einer Paket-Synchronisation müssen für jedes Paket, das gesendet oder empfangen wird, bestimmte Operationen in einer festgelegten Reihenfolge ausgeführt werden. Am Beispiel Lauffeuer sind das beim Empfangen von Paketen: Dekompression, Doppelte Pakete erkennen, evtl. Entschlüsselung, Validierung der evtl. vorhandenen Signatur, Deserialisieren, Anfügen von eigenen Routing- und Meta-Informationen, Generieren von Empfangsbestätigungen usw.

5.2.1.2. Konzept

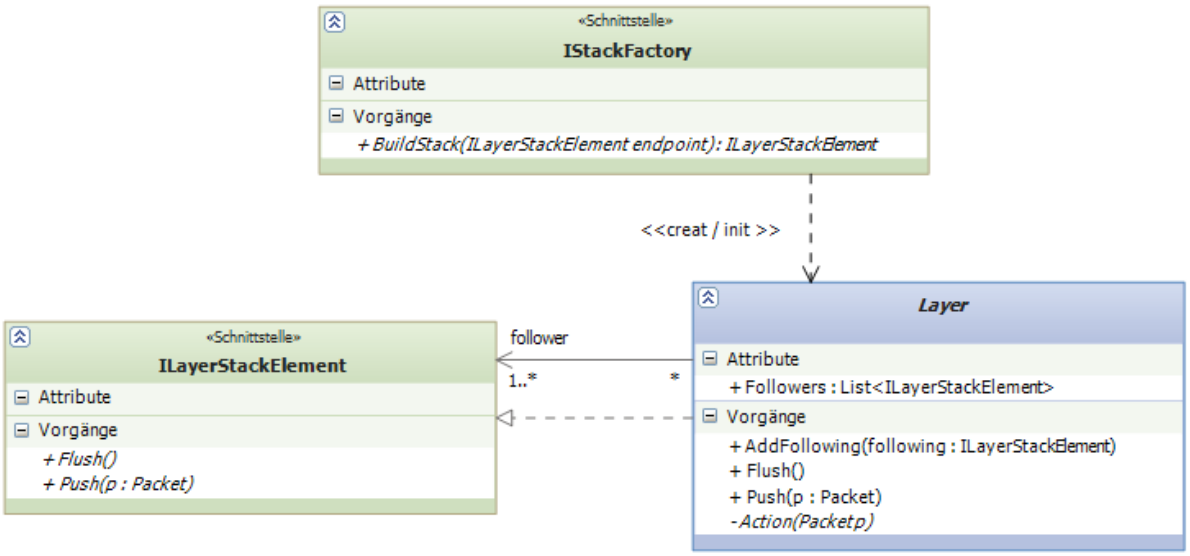

Abbildung 13 - Konzept von *Layern* die zu einem *LayerStack* verkettet werden können

Jede Aufgabe, die an einem Paket durchgeführt werden kann, wird durch eine Klasse repräsentiert, die von der abstrakten Klasse `Layer` erbt. Diese wiederum erbt von `ILayerStackElement`. Im Folgenden werden Instanzen von Klassen die von `Layer` erben einfach *Layer* genannt.

`ILayerStackElement` bietet zwei Methoden:

- `Push` dient dazu eine `Packet`-Instanz in die Instanz zu übergeben..
- `Flush` markiert das Ende einer Folge von `Packet`-Instanzen.

Die Klasse `Layer` erweitert das Interface, sodass sie **immer** mindestens einen Nachfolger besitzt, bei dem es sich wiederum um eine Implementierung von `ILayerStackElement` handelt.

`Layer`-Instanzen werden zu einer Pipeline (sog. *LayerStack*, siehe 5.2.1.3 Pipeline / blockierende Layer) zusammengeschlossen, das bedeutet, jede `Layer`-Instanz hat (mindestens) einen Nachfolger, der wiederum eine `Layer`-Instanz darstellt. Die abstrakte Klasse `Layer` stellt die Methode `Action` bereit, die von der Methode `Push` aufgerufen wird (*Template Method* Entwurfsmuster). Erbende Klassen implementieren in `Action` die Aufgaben, die an jedem Paket ausgeführt werden soll.

Es gibt einen dedizierten Startpunkt und einen dedizierten Endpunkt eines jeden *LayerStacks*. Beim Empfangen von Paketen handelt es sich beim Start um eine Implementierung von `ITransmissionCallback` (siehe 5.2.3. Verbindung mit Transfer) und beim Ende um eine `Controller`-Instanz aus `Transmission`. Beim Versenden von Paketen muss dieser `Controller` den Startpunkt verwenden und eine Implementierung von `ITransferServicePoint` das Ende (siehe Abb. 17). Wie *LayerStacks* initialisiert werden, wird unter 5.2.1.5. Anwendung von LayerStacks beschrieben.

5.2.1.3. Pipeline / blockierende Layer

Um die beste Performance mit einem *LayerStack* zu erzielen, sollten alle *Layer* das Pipeline-Prinzip (sog. *Pipelining*) realisieren (ein Paket bearbeiten, weiterreichen und sofort das nächste bearbeiten). Manche *Layer* erfordern allerdings einen Satz von Paketen. Diese Art von *Layer* wird von nun an *blockierende Layer* genannt. Beide Konzepte (blockierend, nicht blockierend) sollen mit `Layer`-Unterklassen umsetzbar sein. Dazu wird die Methode `Flush` eingeführt, die das Ende eines Paket-Stroms markiert. Mit `Flush` kann ein *Layer* seinem Nachfolger signalisieren, dass er vorerst keine weiteren Pakete erhalten wird.

Abbildung 14 - *Layer 2* ist ein blockierender *Layer*. Erst Aufruf von Methode `Flush`, werden die Pakete an *Layer* 3 weiter gegeben

Blockierende *Layer* werden so implementiert, dass sie beim Aufruf der Methode `Action` das Paket lediglich intern ablegen. Die eigentliche Operation und die Weitergabe der Pakete zu dem/den Nachfolger/n wird erst beim Aufruf von `Flush` ausgeführt.

Layer hingegen, die das Pipeline-Prinzip umsetzen, führen ihre Operationen direkt beim Aufruf von `Action` auf und reichen das Paket unmittelbar an die Nachfolger weiter. Ein Aufruf von `Flush` wird verwendet, um nicht mehr benötigte Verwaltungsdaten zu entfernen und um den `Flush`-Aufruf an alle Nachfolger weiter zu geben.

Es ist offensichtlich, dass *Layer* unterschiedlicher Art (blockierend, nicht blockierend) kombiniert werden können. Allerdings sollten blockierende *Layer* nicht parallelisiert werden (siehe 5.2.1.4 Nebenläufigkeit) und möglichst am Anfang eines *LayerStacks* eingesetzt werden, da sonst der Vorteil des *Pipelinings* gehemmt wird. Es sind auch Szenarien denkbar, in denen eine Parallelisierung von

blockierenden *Layern* durchaus Sinn macht, darum wird diese Entscheidung dem Entwickler einer `IStackFactory`-Implementierung überlassen (Entwurfsmuster *Abstract Fabric*, siehe 5.2.1.7. DefaultLayerFactories).

5.2.1.4. Nebenläufigkeit

Manche Operationen können, relativ zu anderen, rechenintensiv sein, was eine optionale Parallelisierung wünschenswert macht.

Darum kann ein *Layer* mehrere Pipeline-Nachfolger haben. Pakete werden reihum unter den Nachfolgern verteilt, sodass jeder Nachfolger dieselbe Auslastung erfährt. Um parallele Pipeline-Abschnitte zusammenzuführen, verweisen die letzten *Layer* der parallelen Abschnitte auf denselben Nachfolger.

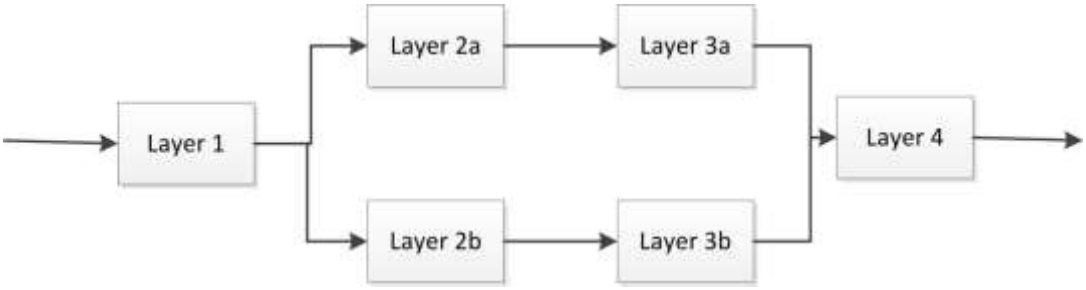

Abbildung 15 - Operation die von *Layer 2* und *Layer 3* durchgeführt, wird je mit zwei Instanzen von *Layer* bearbeitet. Anschließend werden alle Ergebnisse im Pipeline-Prinzip an *Layer 4* weitergereicht, von dem nur eine Instanz existiert.

5.2.1.5. Anwendung von LayerStacks

Es gibt insgesamt fünf Anwendungen für *LayerStacks*. Je ein *LayerStack* wird benötigt um Nachrichten zu versenden und zu empfangen (sog. *SendLayerStack* und *ReceiveLayerStack*). Zwei weitere *LayerStacks* werden benötigt um Pakete, die der Benutzer neu erstellt, zu präparieren (siehe 6.2.3. Präparieren von Paketen, *PrepareLayerStack*). Diese Pakete können entweder verschlüsselt werden (*SecureLayerStack*) oder nicht. Daher werden zwei *LayerStacks* benötigt. Der fünfte *LayerStack* wird verwendet um die Priorität von Paketen zu aktualisieren (*PriorizingLayerStack*).

LayerStacks werden durch *LayerStackFactories* erzeugt (Implementierungen des Intrerfaces `IStackFactory`, Entwurfsmuster *Abstract Fabric*). Für jeden oben aufegührten *LayerStack*-Typ gibt es eine Klasse, die den *LayerStack* erzeugt. So kann auch außerhalb von Transmission eine Reihenfolge von Operationen definiert werden, die beim Senden, Empfangen, Priorisieren oder Vorbereiten von Paketen notwendig ist. Dabei ist es auch möglich nachträglich weitere Klassen zu erstellen, die von der abstrakten `Layer`-Klasse erben.

5.2.1.6. Implementierte Layer

Es folgt eine Auflistung aller `Layer`-Unterklassen, die von Merkur bereitgestellt werden. *Layer* die für die Priorisierung zuständig sind, werden in 5.5.2 Priorisierungsmechanismen aufgeführt.

`DecryptLayer`

versucht Inhalte einer `Packet`-Instanz zu entschlüsseln. Falls nicht möglich, wird die Instanz unverändert weitergereicht.

`CheckSignLayer`

versucht `SignContainer` einer `Packet`-Instanz zu validieren (siehe 5.1.2. PacketContents). Falls nicht möglich, wird die Instanz unverändert weitergereicht.

SignLayer

signiert mit Hilfe des `Crypto`-*Namespaces* alle `SignContainer` in einer `Packet`-Instanz (siehe 5.1.2. PacketContents).

EncryptLayer

verschlüsselt mit Hilfe des `Crypto`-*Namespaces* alle `CryptContainer` in einer `Packet`-Instanz (siehe 5.1.2. PacketContents).

ExcludeLayer

verwirft alle `Packet`-Instanzen, deren IDs auf der *Layer*-internen Ausschlussliste sind. Ist keine Ausschlussliste gegeben, werden alle Instanzen weitergegeben.

SaveLayer

speichert eine `Packet`-Instanz in der Datenbank. Existiert bereits ein Eintrag mit der entsprechenden ID, werden alle Änderungen in die Datenbank übernommen.

ValidationCheckLayer

prüft, ob `RawData` in einer `Packet`-Instanz lesbar ist und ob die angegebene Version unterstützt wird.

CheckDuplicateLayer

verwirft alle `Packet`-Instanzen, deren IDs bereits in der Datenbank hinterlegt sind.

RoutingInfoUpdateLayer

aktualisiert die Routinginformationen in jeder `Packet`-Instanz.

PacketInfoAttachLayer

fügt die Klasse `PaketInfo` zu einer empfangenen `Packet`-Instanz hinzu.

SecureLayer

erstellt das Paket-Schlüsselpaar (siehe 5.7.4. Pakete verschlüsselt versenden). Der *privatePacketKey* wird in den im Paket bereits vorhandenen *Container* hinzugefügt. Der *Container* wird in eine `SignContainer`– und anschließend in eine `CryptContainer`-Instanz hinzugefügt. Diese Instanz wird dann gemeinsam mit dem *publicPacketKey* und einer `MagicWordContent`-Instanz, die sich in einer weiteren `CryptContainer`-Instanz befindet, in einer `ContentContainer`-Instanz abgelegt.

PrepareLayer

Fügt Routinginformationen (`RoutingInfoContent`) für ein Paket hinzu, dass durch den Benutzer erstellt wurde.

5.2.1.7. DefaultLayerFactories

In Merkur werden die folgenden `IStackFactory`-Implementierungen bereitgestellt. Sie erzeugen *LayerStacks*, die unterschiedliche Aktionen mit Paketen durchführen. Die so erzeugten *LayerStacks* können wiederum verbunden werden, da die Anfänge und Enden `ILayerStackElement`-Implementierungen sind.

In der folgenden Aufzählung muss unterschieden werden zwischen den *DefaultFactory*-Klassen und den Attributen in einer Transmission-`Controller`-Instanz. Wird die Transmission wie in 5.8. Standard-Konfiguration verwendet, dann beinhalten die Attribute von Controller genau die hier aufgeführten *DefaultFactories*.

DefaultSecureFactory

bereitet eine `Packet`-Instanz so vor, dass es nur vom Empfänger gelesen werden kann. Dazu werden folgende `Layer`-Instanzen in dieser Reihenfolge aneinandergereiht und initialisiert:

- `SecureLayer`
- `SignLayer`
- `EncryptLayer`

Die `SecureFactory` (Attribut in `Controller`) wird in der Methode `ComposeAndSecure` zusammen mit der `PrepareFactory` und `PriorizeLayerFactory` verwendet.

Die *LayerStacks* werden in der Reihenfolge `SecureFactory`, `PrepareFactory`, `PriorizeLayerFactory` aneinandergereiht.

`DefaultPrepareFactory`

fügt initiale Routinginformationen zu `Packet`-Instanzen hinzu. Dazu wird der `RoutingInfoUpdateLayer` verwendet. Diese `PrepareFactory` (Attribut in der Klasse `Controller`) wird beim Aufruf der Methode `Compose` verwendet. Die *LayerStacks* werden in der Reihenfolge `PrepareFactory` und dann `PriorizeLayerFactory` aneinandergereiht.

`DefaultPriorizeLayerFactory`

priorisiert `Packet`-Instanzen mit Hilfe von *Layern* (siehe 5.5 Priorisierungsmechanismen). Die konkrete Aneinanderreihung von *Layern* wird erst festgelegt, nachdem eine optimale Zusammenstellung und Parametrisierung mit Hilfe einer Simulation gefunden wurde. Am Ende des *LayerStacks* der `DefaultPriorizeLayerFactory` befindet sich eine `SaveLayer`-Instanz.

`DefaultReceiveLayerFactory`

bearbeitet empfangene `Packet`-Instanzen. Dazu werden folgende *Layer* aneinandergereiht:

- `ServiceAccessForTransfer`
- `ValidationLayer`
- `PacketInfoAttachLayer`
- `RoutingInfoUpdateLayer`
- `CheckDuplicateLayer`
- `DecyptLayer`
- `CheckSignLayer`
- `Controller` (Transmission)

`DefaultSendLayerFactory`

sendet `Packet`-Instanzen. Dazu werden folgende *Layer* aneinandergereiht:

- `Controller` (aus `Transmission`)
- `ExcludeLayer`
- `UpdateRoutingInfoLayer`
- `SaveLayer`
- `ITransferServicePoint`

5.2.1.8. Zusammenfassung

Es gibt verschiedene Arten von *LayerStacks* (Senden, Empfangen, Vorbereiten, Gewichten). Ein *LayerStack* ist eine Pipeline, also eine Aneinanderreihung von Aufgaben, wobei jede Aufgabe von einem *Layer* abgedeckt wird. *Layer* können parallel verwendet werden. Jeder *LayerStack* wird durch eine erzeugende Klasse definiert (Entwurfsmuster A*bstract Factory*). Es können nachträglich weitere *Layer* erstellt und eingebunden werden. Das Transmission-System ist also sehr flexibel einsetzbar.

5.2.2. Schnittstellen von Transmission

Die Anwendungsschicht verwendet mindestens zwei Schnittstellen um mit dem Transmission-System zu kommunizieren: `ITransmission` und `IPacketObserver`).

Mit `ITransmission` kann

- geprüft werden, ob ein Gerät (`Partner`) zur Synchronisierung bereit ist (`SearchPartner`).
- eine Synchronisation mit einem Partner begonnen werden (`Sync`).
- ein vom Benutzer neu verfasstes Paket aufgenommen werden, sodass es bei der nächsten Synchronisierung versendet werden kann (`ComposePacket` bzw. `SecureAndComposePacket`).

Mit den übrigen Methoden und dem Interface `IPacketObserver` wird das *Observer*-Entwurfsmuster umgesetzt. Ein oder mehrere Instanzen der Anwendung können sich als Observer beim Transmission-System registrieren und werden benachrichtigt, wenn neue Pakete empfangen wurden. Dabei wird für jedes empfangene Paket eine Benachrichtigung erstellt, auch wenn diese nicht gelesen werden können.

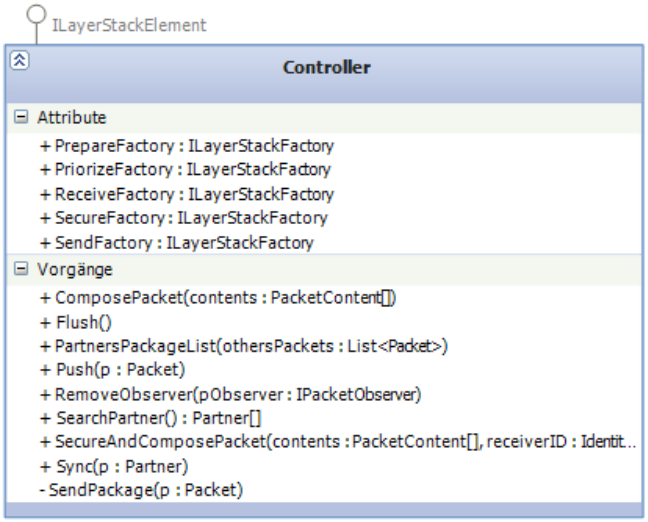

Abbildung 16 - Controller-Klasse des Namespace Transmission

Die Klasse `Controller` ist das Herzstück des Transmission-Systems. In ihr sind die zu verwendenden `IStackFactory`-Instanzen hinterlegt, die alle nötigen Mechanismen zum Vorbereiten, Senden, Empfangen und Gewichten von Paketen auswählen und initialisieren. Werden die *LayerStackFactories,* wie in 5.8. Standard-Konfiguration beschrieben, verwendet, wird die `Controller`-Klasse die *DefaultFactories* verwenden (siehe 5.2.1.7. DefaultLayerStackFactorie).

5.2.3. Verbindung mit Transfer

Einige Ereignisse (z.B. wie eine Synchronisationsanfrage eines anderen Gerätes) werden im Transfer-System ausgelöst und müssen dann durch das Transmission-System behandelt oder zur Anwendung weitergereicht werden. Solche Ereignisse werden durch Methoden des Interfaces `ITransmissionCallback` signalisiert:

- `PacketlistOfPartner` wird aufgerufen, wenn ein anderes Gerät eine Synchronisationsanfrage stellt. Eine Anfrage beinhaltet auch die Liste aller Pakete die zur Synchronisation angeboten werden.
- `ConnectionClosed` wird aufgerufen wenn die Verbindung beendet wurde.
- `RequestPacketList` wird von Transfer aufgerufen, wenn die Liste der eigenen, zur Synchronisierung Verfügung stehender Pakete, gefordert wird.

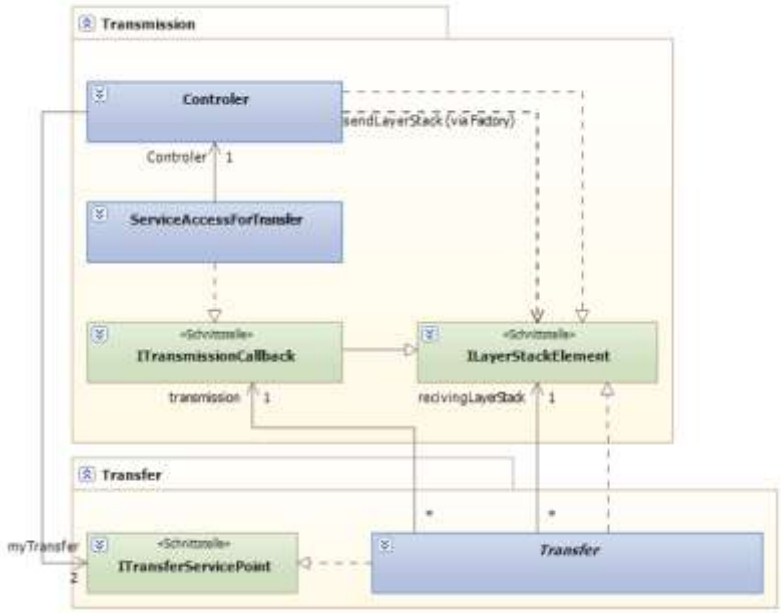

Abbildung 17 - Verwendete Schnittstellen zwischen Transmission und Transfer

`ITransmissionCallback` erweitert das Interface `ILayerStackElement`, sodass das Transfer-System mittels der Methode `Push` empfangene Pakete in den *ReceivingLayerStack* hinzufügen kann.

Die Klasse `ServiceAccess-ForTransfer` implementiert `ITransmissionCallback`. Sie stellt also zum einen den Anfang des *ReceivingLayerStack* dar, zum anderen eine Klasse, die Steuersignale des Transfer-Systems an die `Controller`-Instanz weiterreicht.

5.2.4. Priorisieren von Paketen

Für die Priorisierung von Paketen wird ebenfalls ein *LayerStack* verwendet (5.2.1 TransmissionLayer-Konzept). Die Priorisierung ist notwendig um Pakete sortierbar zu machen: Wichtige Pakete werden bei einer Synchronisierung als erstes versendet, unwichtige werden bei Speicherplatzmangel als erstes verworfen.

Die Priorisierung von Paketen erfolgt nicht relativ zueinander, sondern absolut.

Pakete werden, nachdem sie empfangen oder neu erstellt wurden, neu priorisiert. Mehr zu Priorisierung von Paketen, siehe 5.5.2. Priorisierungsmechanismen.

5.2.5. Kryptographie-Implementierung

Der *Namespace* `Crypto` stellt Kryptographiemechanismen und eine dazugehörige *Facade* (Entwurfsmuster) bereit.

Das Interface `ICryptMechanism` stellt zwei Grundlegende Funktionen bereit: Ver- und Entschlüsseln (`Encrypt` und `Decrypt`). Zusätzlich je zwei Methoden, die überprüfen, ob gegebene Daten mit einem gegebenen Schlüssel ver- bzw. entschlüsselt werden können (`CanEncrypt` und `CanDecrypt`). Alle Methoden arbeiten auf Byte-Arrays und dem `Key`-Model (siehe 5.6.3. GlobalModels). Die Methode `getChunkSize` gibt die maximale Anzahl der Bytes aus, die in einem Block verschlüsselt werden können. In Merkur wird lediglich ein RSA[3]-Mechanismus bereit gestellt, der wiederum durch das *.NET-Framework* zur Verfügung gestellt wird. Bei der Klasse `RSA` handelt es sich also nur um einen *Adapter* (Entwurfsmuster).

Abbildung 18 - Der `Crypt`-*Namespace*

`ICryptProvider` ist die *Facade* (Entwurfsmuster) des `Crypto`-*Namespaces*. Sie bietet Methoden, die `PacketContent`-Instanzen als Parameter benötigen:

- `CheckSign` prüft, ob die Signatur gültig ist. Der dafür notwendige Schlüssel kann mit Hilfe von `GetIdenity` bestimmt werden.
- `GetIdentity` versucht anhand der bekannten Schlüssel die Identität des Absenders des `PacketContents` zu bestimmen. Dafür wird das *MagicWord* verwendet (siehe 5.7.1. Verwendung eines MagicWord).
- `Decrypt` entschlüsselt ein `PacketContent` mit dem eigenen *privateKey*. Wenn das nicht möglich ist, bleibt der `PacketContent` unverändert und unlesbar - das Paket ist für einen anderen Empfänger bestimmt.

[3] Siehe http://de.wikipedia.org/wiki/RSA-Kryptosystem

- `Encrypt` verschlüsselt alle `CryptContainer` (siehe 5.1.2. PacketContents) mit dem eigenen *privateKey*.
- `Sign` signiert alle `SignContainer` (siehe 5.1.2. PacketContents) mit dem eigenen *privateKey*.
- `MatchesKey` stellt eine Methode bereit, mit der getestet werden kann, ob ein Byte-Array mit einem vorhandenen Schlüssel entschlüsselt werden kann, indem es das *MagicWord* überprüft. Das wird z.B. für die Validierung von Empfangsbestätigungen zu Nachrichten benötigt (siehe 5.7.6. Empfang, Zuordnung und Prüfung einer Empfangsbestätigung).
- Mit Hilfe von `Init` wird der `CryptProvider` mit dem zu verwendenden *privateKey* und einer Liste mit allen bekannten Identities ausgestattet.

5.3. Transfer

Das Transfer-System abstrahiert Netzwerkschnittstellen und bietet einheitliche Methoden für die Übertragung von Paketen zwischen zwei Knoten bzw. zwischen Knoten und Server.

5.3.1. Transfer

Im *Namespace* `Transfer` wird die Übermittlung von `Packet`-Instanzen zwischen den Endgeräten bzw. Server realisiert. Dabei wird eine abstrakte Schnittstelle für die einzelnen Übertragungstechnologien (Bluetooth, WLAN, ...) geschaffen.

Es wird zwischen der Implementierung für die Übertragung zu einem Server und der Implementierung der Ad-hoc Übertragung unterschieden.

Für die Anwendungsschicht Lauffeuer wird nur eine einzige Ad-Hoc Übertragungsmethode realisiert - *BTSI* (siehe 4.1. Bluetooth). Durch die Abstraktion ist es aber mit wenig Aufwand möglich, nachträglich weitere Übertragungstechnologien wie z.B. Ad-Hoc WLAN, Bluetooth oder NFC zu implementieren (siehe Abb. 19, grau dargestellt) sobald diese für die Plattform WP7 verfügbar sind.

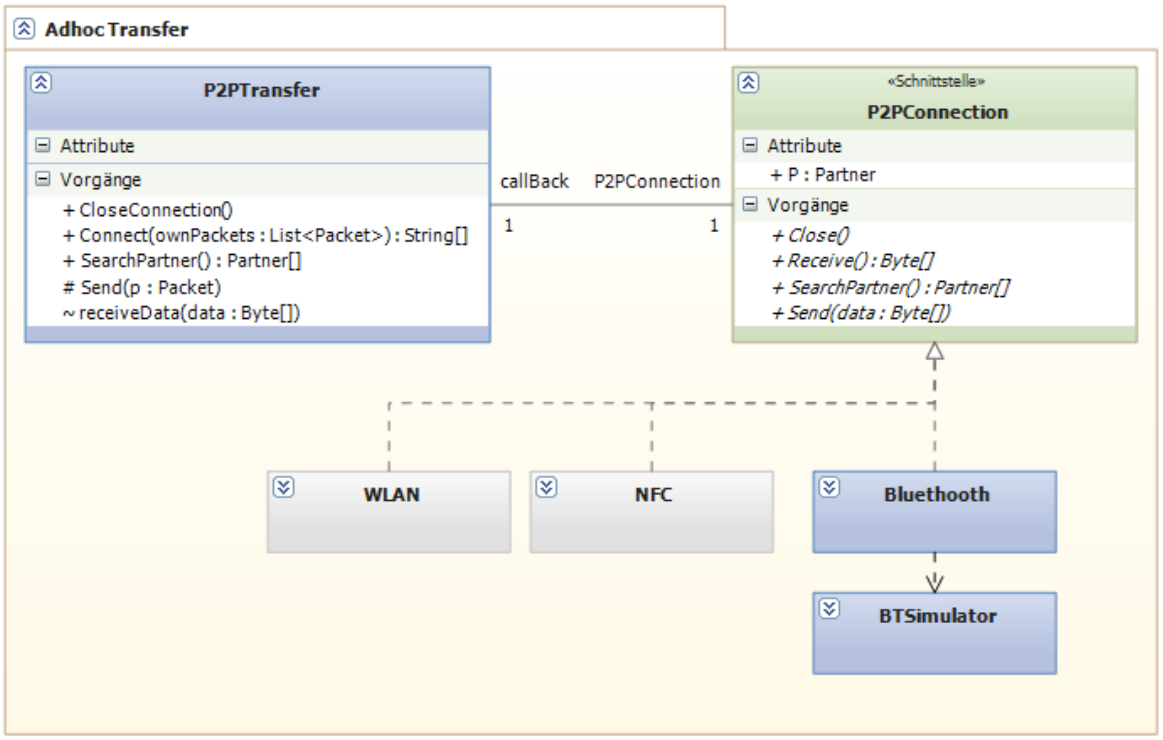

Abbildung 19 - Unterschiedliche Übertragungstechniken abstrahiert

Nach außen wird dem Transmission-System eine einheitliche Schnittstelle (`ITransmissionServicePoint`) geboten, über die andere Endgeräte gesucht, und Pakete ausgetauscht werden können.

5.3.2. Server

Der Server realisiert im System Lauffeuer eine Weiterleitung von Paketen via E-Mail. Wie für die Lauffeuer-Anwendung werden auch für die Lauffeuer-Server-Anwendung (siehe 5.13. Lauffeuer-Server) Methoden zur Verfügung gestellt.

Pakete werden (wie in /F110/ beschrieben) an den Server gesendet, sobald eine Internetverbindung besteht. Die Aufgabe des Servers ist es festzustellen, ob der Absender einer Nachricht gültig ist (siehe 5.7. Kryptographie) und stellt ggf. die Nachricht an die im Paket selbst angegebene Empfänger zu.

Zusätzlich zu den durch Merkur bereitgestellten Funktionen, sollen der Austausch öffentlicher Schlüssel und Profilinformationen möglich sein (siehe 5.13. Lauffeuer-Server).

5.4. Nachrichtenzustellung über den Server

Die API Merkur bietet im *Namespace* `Transfer` eine Schnittstelle mit dem Namen `HTTPPacketTransfer`. Über die dort definierten Methoden kann ein *Smartphone* mit der Anwendung Lauffeuer mit dem Server aufnehmen und ihm `Packet`-Instanzen zusenden. Dabei durchlaufen die Instanzen im *Smartphone* dieselben *LayerStacks* wie bei einer Synchronisation zwischen zwei Geräten (siehe 5.2.1. TransmissionLayer-Konzept). Der Server stellt eine Implementierung der Schnittstelle als Webservice[4] bereit. Alle empfangenen Pakete werden mit Hilfe eines *ServerReceiveLayerStacks*, der innerhalb der Klasse `ServerController` definiert ist, verarbeitet:

Als erstes wird anhand der ID der Pakete geprüft, ob es sich um ein bereits bearbeitetes Paket handelt, oder nicht. Handelt es sich um ein unbekanntes Paket, wird es mit dem privaten Schlüssel des Servers entschlüsselt. Der Inhalt des Pakets wird von den, in dem Paket angegebenen, E-Mail-Adressen (siehe `ReceiverIDContent`, 5.1.2. PacketContents) getrennt. Für jeden E-Mail-Empfänger wird eine Nachricht erzeugt, die zum einen den Inhalt des Pakets enthält, sowie einige Informationen wie z.B. Sendezeitpunkt beim Absender. Es werden noch die Profilinformationen des Absenders hinzugefügt, damit der Leser der Email weiß, von wem die Nachricht ursprünglich stammte.

5.5. Routing

Um die Übertragungen von Paketen zu regulieren, und Pakete festzulegen, die Netzwerkteilnehmer weitergereicht werden, muss ein Routingverfahren festgelegt werden. Mit den in 5.1.2. TransmissionLayer-Konzept beschriebenen Methoden, können unterschiedliche Routingverfahren implementiert werden. Auf diese soll nun kurz eingegangen werden.

Verbindungen zwischen Netzwerkknoten entstehen stochastisch und sind kurzlebig. Gleichzeitig sollen so viele Pakete wie möglich an sehr viele Geräte übertragen werden. Daher wird ein Protokoll benötigt, welches die Weitergabe und Kopie der Pakete (sog. *replication-based routing*) in einer Umgebung mit einem spärlichen und stochastischen Netzwerk (sog. *delay-tolerant networking*) effizient umsetzt.

[4] Siehe http://msdn.microsoft.com/en-us/library/ms972326.aspx

Die Anzahl der Pakete wird durch Senden von Empfangsbestätigungen verringert. So sollen Nachrichten entfernt und nicht weiter übermittelt werden, wenn eine Empfangsbestätigung für sie eingetroffen ist.

5.5.1. Replication-based routing

Mit Hilfe des Simulators sollen bereits existierende Protokolle auf Verwendbarkeit bzgl. Lauffeuer geprüft werden.

Getestet werden *Epidemic Routing*, *MaxProp* und *Spray and Wait*[5].

5.5.2. Priorisierungsmechanismen

Um die Reihenfolge der Pakete, die synchronisiert werden, festzulegen oder um zu entscheiden, welche Pakete im Falle von Speicherknappheit gelöscht werden sollen, müssen alle Pakete priorisiert werden.

Die Priorisierung erfolgt absolut und berücksichtigt verschiedene Kriterien. Für die Priorisierung wird ein *LayerStack* verwendet (siehe 5.2.1. TransmissionLayer-Konzept). Jedes Kriterium wird dabei durch ein *Layer* repräsentiert. Die Priorisierung von Paketen erfolgt nach jeder Synchronisation. Pakete die vom Benutzer erstellt wurden werden priorisiert, sobald sie zu Transmission hinzugefügt werden.

Die Kriterien werden im Folgenden erläutert. Welche davon final und mit welchen Gewichtungen bzw. Parametrisierungen übernommen werden, wird durch Tests im Simulator (siehe Pflichtenheft, Kapitel 1.2. Wunschkriterien) entschieden.

Alter der Nachricht (`TotalAgePriorizingLayer`)
: Je älter ein Paket ist, desto niedriger wird es priorisiert.

Spamflagcount-Verhältnis (`SpamRatioPriorizingLayer`; gilt nur für öffentliche Nachrichten)
: Je öfter ein Paket, im Verhältnis zu Anzahl der übertragenden Knoten, als Spam markiert wurde, desto niedriger wird es priorisiert (gilt nur für öffentliche Pakete).

Anzahl der Hops (`HopPriorizingLayer`)
: Je höher die Anzahl der Hops eines Pakets (siehe Glossar), desto niedriger wird es priorisiert.

Anzahl der Übertragungen (`TotalSentCountPriorizingLayer`)
: Je höher die Anzahl der gesamten Übertragungen eines Pakets, desto niedriger wird priorisiert.

Hopdichte bzgl. der zurückgelegten Entfernung (`HopDensityPriorizingLayer`)
: Je höher die Hopdichte (Anzahl Hops bzgl. der zurückgelegten Entfernung, also je öfter eine Nachricht innerhalb eines Gebiets mit kleiner Reichweite übertragen wurde) eines Pakets ist, desto niedriger wird es priorisiert. Dafür ist es nötig, dass die Anwendungsebene eine Implementierung von `IPositionProvider` bereitstellt.

Eigene Nachrichten (`OwnPacketPriorizing`)
: Handelt es sich bei dem Paket um ein eigenes Paket, wird es hoch priorisiert

[5] Siehe: http://en.wikipedia.org/wiki/Routing_in_delay-tolerant_networking

Nicht übertragene Nachrichten (`OwnSentCountPriorizingLayer`)

Wenn ein Paket in der letzten Übertragung nicht übertragen werden konnte, wird sie höher priorisiert.

Zufallswahl (`RussianRoulette`)

Es wird eine Zufallszahl ermittelt. Unterschreitet diese einen bestimmten Wert, wird das Paket nicht übertragen.

Priorisierung zurücksetzen (`PriorityResetLayer`)

Setzt die Priorität eines Pakets zurück. Das ist nötig, wenn ein Paket z.B. völlig neu priorisiert werden soll.

5.5.2.1. Karma-Priorisierung

Einige Priorisierungsmechanismen beruhen auf dem Prinzip der Belohnung. Verhält sich ein Benutzer "gut", so sollen seine eigenen Nachrichten auch bevorzugt werden. Dieses "gute" oder "schlechte" Verhalten wird *Karma* genannt und wird in jedem Gerät für den jeweiligen Benutzer berechnet. Basierend auf *Karma* eines Benutzers, zum Zeitpunkt des Verfassens einer Nachricht, wird diese höher oder niedriger priorisiert (z.B. *TTL*, siehe unten).

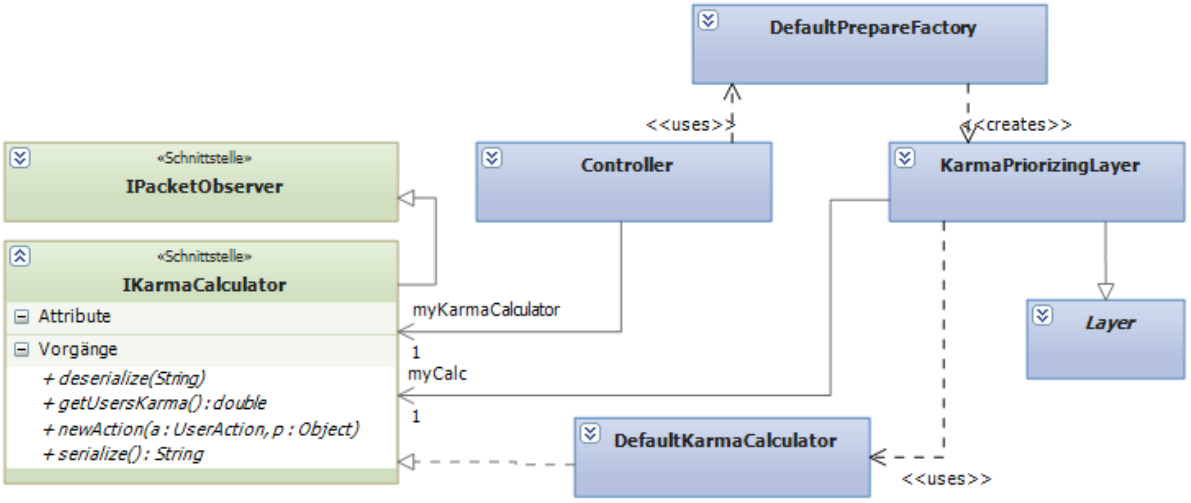

Abbildung 20 - Berechnung von *Karma*

Die Berechnung übernimmt eine Klasse, die das Interface `IKarmaCalculator` implementiert. Die Klasse `Controller` aus `Transmission` wird alle nötigen Ereignisse an eine `IKarmaCalculator`-Instanz weitergeben, sodass diese das *Karma* berechnen kann. Mit Hilfe des `KarmaPriorizingLayer` kann das *Karma* im *PrepareLayerStack* (siehe 5.2.1. TransmissionLayer-Konzept) verwendet werden.

Die Folgende Karma-Berechnung erfolgt durch die Klasse `DefaultKarmaCalculator`.

Anzahl der Paketübertragungen absolut

Hat ein Benutzer bereits viele Pakete übertragen, steigt sein *Karma*.

Anzahl fremde und unterschiedlichen übertragenen Pakete

Wurden viele fremde Pakete übertragen, steigt das *Karma*

Anzahl momentan gespeicherten, fremdem Pakete

Speicherung vieler fremder Pakete soll belohnt werden

Anzahl Synchronisierungen

Unabhängig von der Anzahl der übertragenen Pakete, ist es gut, wenn ein Benutzer oft eine Paketsynchronisation durchführt.

5.5.3. TTL

Alle Pakete erhalten eine *Time-To-Live* (kurz *TTL*). Sie ist in den Routinginformationen (`RoutingInfoContent`, siehe 5.1.2. PacketContents) enthalten und wird von jedem Knoten gepflegt. Der initiale Wert wird von *Karma* des Absenders beeinflusst, kann jedoch ein definiertes Maximum nicht übersteigen. Pakete, deren *TTL* abgelaufen ist, werden verworfen. Empfangsbestätigungen erhalten bei ihrer Erstellung den *TTL*-Wert des Pakets, das bestätigt werden soll.

5.6. Models und DataStore

Sogenannte *Models* haben den Zweck Daten im System zur Laufzeit zu speichern. Sie können mit Hilfe eines sog. *DataStore* auch dann gespeichert werden, wenn das Programm unterbrochen wurde. Es wird zwischen *Models* unterschieden, die von Merkur benötigt werden, also *Models*, die für generische P2P-Übertragungen notwendig sind (sog. *GlobalModels* aus dem gleichnamigen *Namespace*) und *Models*, die speziell für die Anwendung Lauffeuer benötigt werden (*ApplicationModels*, ebenfalls aus dem gleichnamigen *Namespace*).

5.6.1. DataStore

Der *DataStore* dient dazu, große Mengen von *Models* zu speichern, auch dann, wenn die Anwendung beendet wurde. Analog zu der Trennung zwischen *Global-* und *ApplicationModels*, existiert ein *DataStore* in Merkur (`GlobalDataStore`) und ein zweiter in der Anwendung Lauffeuer (`ApplicationDataStore`). Der `ApplicationDataStore` verwendet den `GlobalDataStore`, bietet aber für die Anwendung eine vereinfachte Schnittstelle. Ebenso gibt es für die Server-Anwendung (siehe 6.6. Lauffeuer-Server) einen eigenen `ServerDataStore`. Alle Objekte, die von *DataStores* zur Verfügung gestellt werden, erben von `AbstractModel` (siehe 5.6.3. GlobalModels).

5.6.2. Konsistenz

Alle Models, die in einem *DataStore* gespeichert werden können, erben (direkt oder indirekt) von der abstrakten Klasse `AbstractModel`. Diese dient dazu, die Konsistenz zwischen einem *DataStore* und unterschiedlichen *Models* wiederherzustellen, wenn sich ein *Model* verändert hat. Alle *Models* die aus einem *DataStore* geladen wurden, werden beim Aufruf der Methode `Flush` im *DataStore* erneut gespeichert.

Alle im Klassendiagramm verwendeten Attribute von *Models*, die als `public` deklariert sind, sind in der Implementierungsphase als Properties zu implementieren.

5.6.3. GlobalModels

Ein Paket wird durch ein *Model* dargestellt (sog. `Packet`). Es besteht aus verschiedenen Inhalten mit verschiedenen Typen (sog. `PacketContent`, siehe 5.1.2. PacketContents). Alle Inhaltstypen müssen sich zu einem Byte-Array serialisieren bzw. deserialisieren lassen. `ApplicationModels` können weitere Paketinhalte zur Verfügung stellen, indem sie von `PacketContent` erben. Zu

Pakten werden weitere Informationen zur Verfügung gestellt, die den Umgang mit Paketen vereinfachen sollen (z.B. `PaketInfo`). `PacketInfo`-Instanzen werden bei Übertragungen eines Paketes nicht übermittelt. Bei Routinginformationen wird zwischen öffentlichen, die übermittelt werden (`RoutingInfoContent`), und privaten (`PrivateRoutingInfo`), die nur lokal gepflegt und gespeichert werden, unterschieden.

Neben Paketen werden auch Identitäten (`Identity`) repräsentiert, die einen oder keinen Schlüssel (`Key`) besitzen. Alle Instanzen von Identitäten sind potentielle Empfänger einer Nachricht. Der in einer Identität abgelegte Schlüssel ist der öffentliche Schlüssel des zugehörigen Benutzers. Er wird benötigt, um Nachrichten zu verschlüsseln (siehe 5.7. Kryptographie).

5.7. Kryptographie

Damit Inhalte von Paketen nicht unerlaubt von Fremden gelesen werden können, verwendet Merkur kryptographische Methoden. Es gibt drei Anwendungen:

- Signieren von Inhalten, sodass der Empfänger den Absender des Pakets verifizieren kann (Authentizität und Integrität).
- Verschlüsselung von Inhalten, sodass nur der Empfänger in der Lage ist den Inhalt zu lesen.
- Erzeugen fälschungssicherer Empfangsbestätigungen. Nur der Empfänger eines Pakets ist in der Lage eine gültige Empfangsbestätigung zu erstellen.

Im Folgenden wird der öffentliche Schlüssel mit *publicKey* (siehe Glossar) und private Schlüssel mit *privateKey* (siehe Glossar) bezeichnet.

5.7.1. Verwendung eines MagicWord

Bei einem sog. *MagicWord* handelt es sich um eine konstante Bytefolge, die jedem Gerät im System bekannt ist. Sie dient dazu, dass geprüft werden kann, ob ein bekannter Schlüssel zu verschlüsselten Daten "passt", oder nicht. Das mit dem *publicKey* des Empfängers verschlüsselte *MagicWord* wird zu Beginn des Pakets übertragen. Der Empfänger prüft, ob er nach dem Entschlüsseln mit seinem *privateKey* das *MagicWord* erhält. Empfänger können so automatisiert testen, ob ein Paket für sie bestimmt ist oder nicht. Ebenso wird dieses Verfahren für die Verifikation von Empfangsbestätigungen verwendet (siehe 5.7.6. Empfang, Zuordnung und Prüfung von Empfangsbestätigungen).

5.7.2. Verschlüsselung von Paketen und Paketinhalten

Es können nur Pakete verschlüsselt werden, für die der Absender den *publicKey* des Empfängers besitzt. Die Voraussetzung dafür ist, dass der Empfänger auch einen Lauffeuer-Profil hat und als ein Kontakt beim Absender hinterlegt ist. Pakete bestehen aus mehreren Abschnitten (siehe 5.1.2. PacketContents). Manche werden verschlüsselt, signiert, beides oder unverschlüsselt und unsigniert übertragen (z.B. Routinginformationen).

Handelt es sich um eine öffentliche Nachricht, wird der Inhalt des Pakets (siehe Abb. 21, „content“) unverschlüsselt weitergereicht. Für diese Art von Paketen gibt es keine Empfangsbestätigung.

5.7.3. Signieren von Paketen und Paketinhalten

Beim Empfang eines signierten Pakets, soll der Empfänger feststellen können, ob der Inhalt unerlaubt verändert wurde bzw. der Absender des Pakets bekannt ist. Um dies zu realisieren, erstellt der Sender eines Pakets eine Prüfsumme (siehe Glossar) für den zu signierenden Paketinhalt (`PacketContent`) und verschlüsselt diese Prüfsummen mit seinem *pivateKey*. Der Empfänger entschlüsselt diese verschlüsselte Prüfsumme nach Empfang des Pakets, bildet zusätzlich selbst die Prüfsumme des zu signierenden Paketinhaltes und vergleicht seine selbst erzeugte Prüfsumme mit der entschlüsselten. Handelt es sich um dieselbe Prüfsumme, kann der Empfänger sicher sein, dass es sich um den bekannten Absender handelt und dass kein Dritter den Inhalt verfälscht hat. Um das möglich zu machen, muss der Sender eines signierten Pakets deshalb auch ein Kontakt des Empfängers sein.

Wird das Verfahren Signieren mit dem Verfahren Verschlüsseln (siehe oben) kombiniert, so werden Inhalte erst signiert und anschließend inkl. Signierung verschlüsselt.

5.7.4. Pakete verschlüsselt versenden

Möchte der Absender ein Paket versenden, für das später eine Empfangsbestätigung erstellt werden kann, erstellt er eigens für dieses Paket ein neues Schlüsselpaar bestehend aus einem privaten und einem öffentlichen Paket-Schlüssel (sog. *privatePacket*- und *publicPacketKey*). Der *privatePacketKey* wird dann genau so wie das *MagicWord* (siehe 5.7.1. Verwendung eines MagicWord), und der vom Nutzer erstellte Inhalt, mit dem *publicKey* des Empfängers verschlüsselt. Somit ist nur der Empfänger in der Lage den Inhalt und den privaten Paketschlüssel zu lesen. Die so entstandenen Daten und der unverschlüsselte *publicPacketKey* bilden gemeinsam mit den unverschlüsselten Routinginformationen[6] das neue versandfertige Paket.

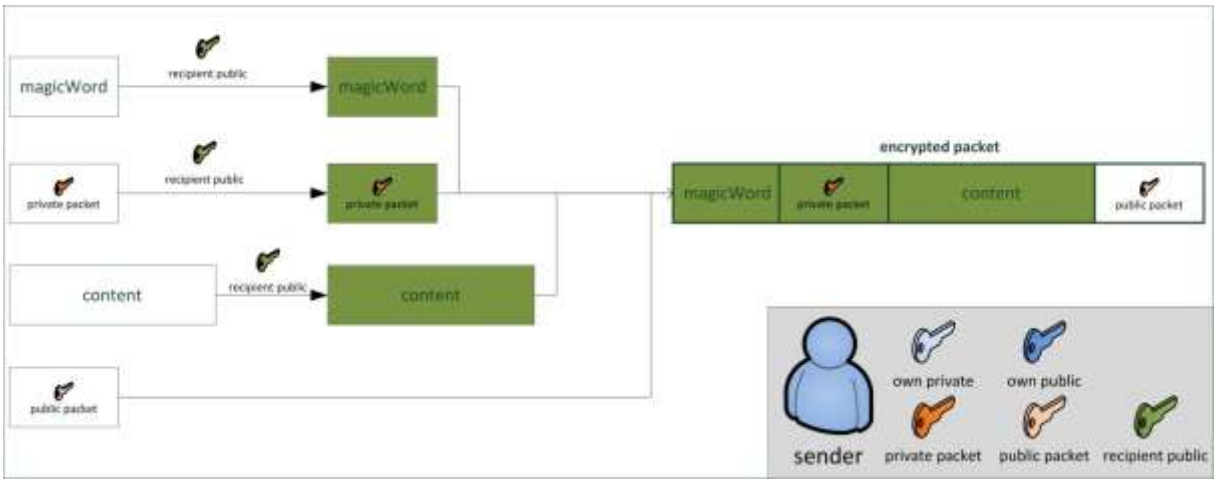

Abbildung 21 - Sender sichert ein privates Paket und erstellt dafür *PacketKeys*

[6] Für mehr Übersichtlichkeit wird im Folgenden nur auf die verschlüsselten Inhalte eines Pakets eingegangen. Auch die Grafiken abstrahieren mehrere `PacketContent`-Instanzen, die eigentlich unterschiedlich behandelt werden, als einen einzelnen Block, der verschlüsselt wird.

5.7.5. Empfang eines verschlüsselten Pakets und Erzeugen der Empfangsbestätigung

Erhält der Empfänger neue Pakete kann er mit Hilfe seines eigenen *privateKeys* und dem *MagicWord* herausfinden, welche Pakete an ihn gerichtet sind. An ihn gerichtete Pakete werden entschlüsselt und enthaltene Signaturen überprüft.

Um eine Empfangsbestätigung zu generieren, entschlüsselt der Empfänger auch den *privatePacketKey*, mit dem er wiederum das *MagicWord* verschlüsselt. Dieses und die ID des Pakets (siehe 5.1.2. PacketContents) bilden dann die Empfangsbestätigung.

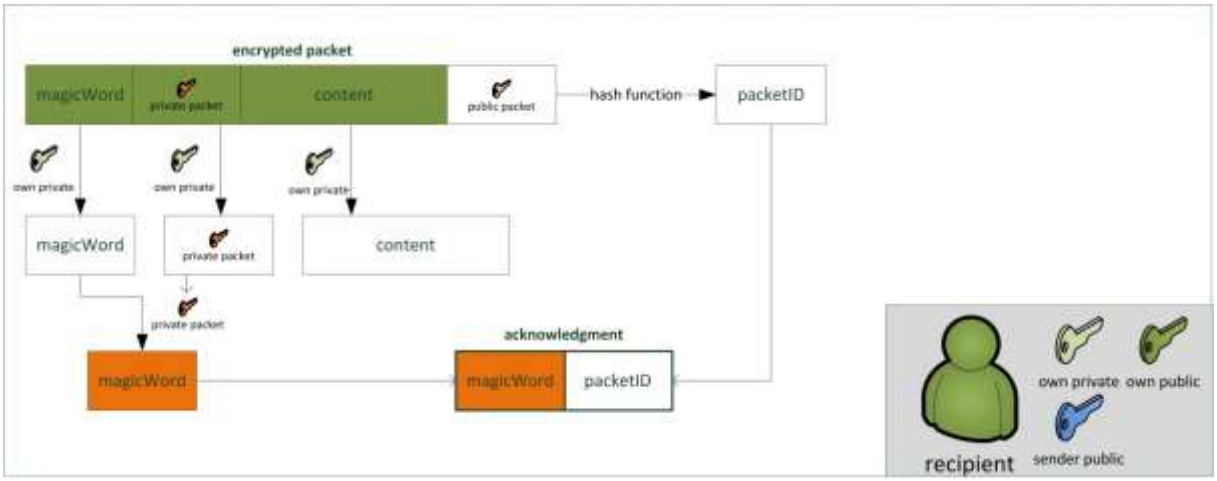

Abbildung 22 - Empfangen eines privaten Pakets; Erzeugen der Empfangsbestätigung

5.7.6. Empfang, Zuordnung und Prüfung einer Empfangsbestätigung

Wird eine Empfangsbestätigung (Abb. 22, „acknowledgement“) empfangen, wird anhand der enthaltenen ID (Abb. 22, „packetID“) das dazu passende Paket gesucht. Um zu überprüfen, ob eine Empfangsbestätigung gültig ist, wird das in der Empfangsbestätigung verschlüsselt enthaltene *MagicWord* mit dem im zu bestätigendem Paket enthaltenen *publicPacketKey* entschlüsselt. Handelt es sich bei den entschlüsselten Daten um das bekannte *MagicWord*, ist sichergestellt das die Empfangsbestätigung nur vom Empfänger erstellt werden konnte.

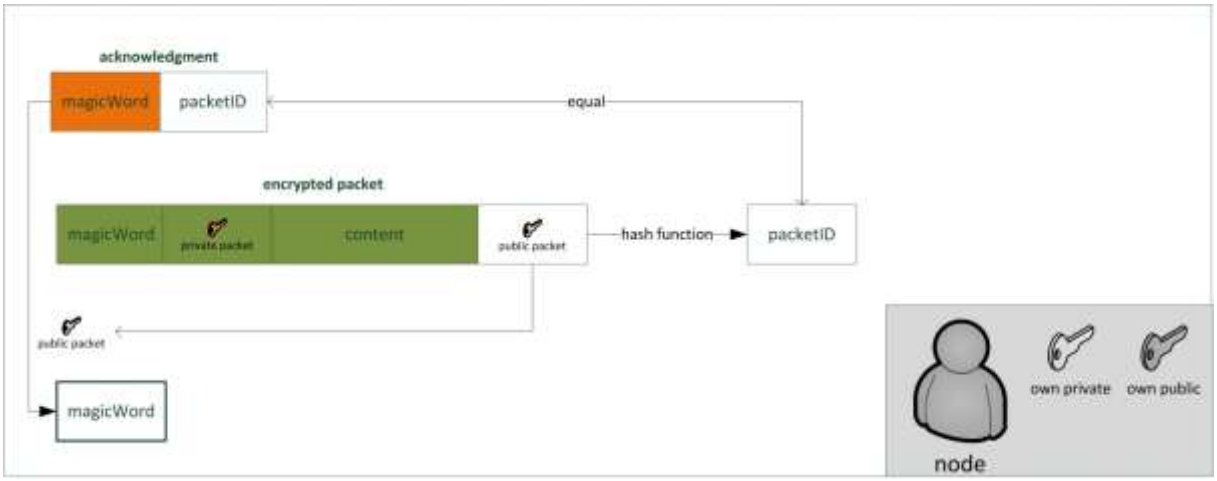

Abbildung 23 - Knoten prüft Empfangsbestätigung

5.8. Standard-Konfiguration

Wie in 3. Trennung zwischen Lauffeuer und Merkur beschrieben, bietet Merkur mindestens die Mechanismen, für alle funktionalen Anforderungen aus dem Pflichtenheft. Die Standard-Konfiguration soll die passenden Mechanismen kombinieren. Um die Verwendung von Merkur für Lauffeuer möglichst einfach zu gestalten, wird in Merkur die Klasse `ConfigurationHelper` erstellt. Sie besitzt mehrere *Convenience Methods* (Entwurfsmuster).

6. Lauffeuer

Bei Lauffeuer handelt es sich um ein Softwaresystem, das auf das Merkur aufbaut.

Dabei wird grundsätzlich zwischen der Anwendung für WP7 und der Anwendung für den Windows Server (siehe Glossar) unterschieden, der die Weiterleitung von Paketen via Email umsetzt (siehe /F110/).

Die WP7-Lauffeueranwendung bietet eine graphische Benutzeroberfläche für den Benutzer an. Dadurch können Profilinformationen vor Beginn einer Krise ausgetauscht werden, sodass er in einer Krisensituation in der Lage ist, verschlüsselte Nachrichten zu versenden.

6.1. Unterschiede zwischen Paketen

Ein Anwender hat, wie in /F060/ beschrieben, die Möglichkeit eine Nachricht durch ein Paket an einen Lauffeuer Benutzer zu schicken oder an eine gewöhnliche E-Mail Adresse. In beiden Fällen wird das Paket durch Synchronisation unter verschiedenen Endgeräten immer weiter propagiert. Ist der Empfänger ein Lauffeuer Benutzer der sich auch im Krisengebiet befindet, kann er auf diesem Wege das Paket empfangen. Befindet sich der Benutzer nicht im Krisengebiet, oder ist der Empfänger mit einer E-Mail Adresse hinterlegt, muss das Paket solange propagiert werden, bis ein Knoten über eine Internetverbindung verfügt (z.B. weil er das Krisengebiet verlassen konnte) und nun alle Pakete an den Server übertragen kann. Der Server benachrichtigt dann die Empfänger per E-Mail.

Die Anwendung wird eine Nachricht an einen Benutzer immer in zwei Paketen versenden: Einmal direkt an den Benutzer adressiert und einmal an den Server. Das an den Server adressierte Paket enthält die Emailadresse des Empfängers. Wird eine Nachricht an mehrere Benutzer adressiert, muss wg. der Verschlüsselung für jeden Benutzer ein eigenes Paket erzeugt werden. Zusätzlich wird ein weiteres Paket erstellt, das an den Server adressiert ist und alle Emailadressen der Empfänger enthält (siehe /F110/ Paketaustausch zwischen Server und Benutzer).

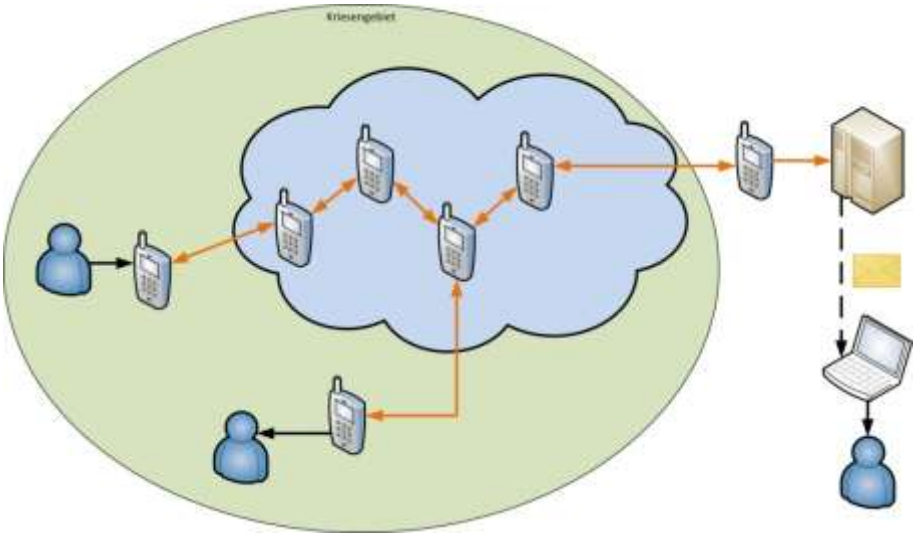

Abbildung 24 - Unterschiedliche Pakete

Eine Kommunikation von außerhalb in das Krisengebiet wird nicht bereitgestellt.

6.2 Lauffeuer-Pakete

Im Folgenden wird beschrieben, wie Pakete aufgebaut sind, die mit dem System Lauffeuer in Kombination mit Merkur erstellt wurden.

6.2.1. private Lauffeuer-Pakete

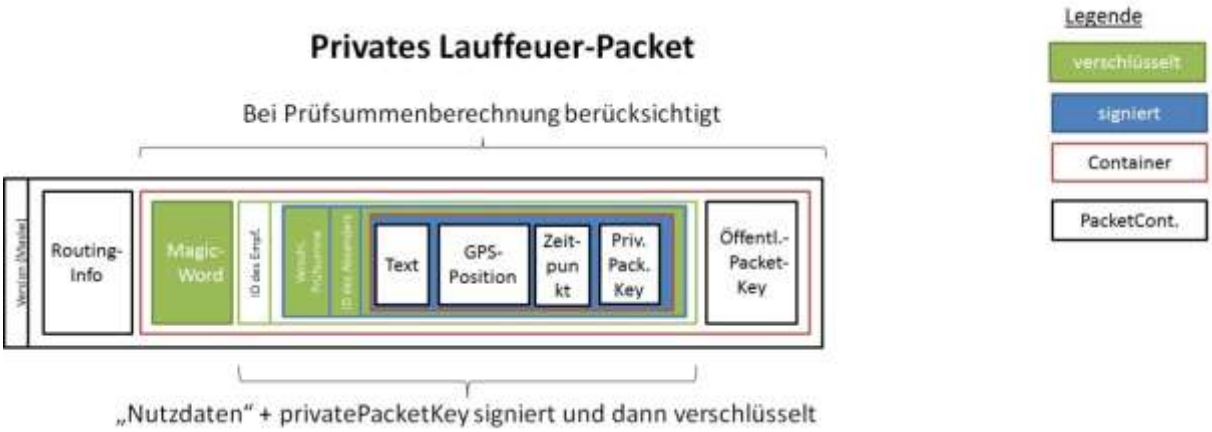

Abbildung 25 - privates Lauffeuer-Paket

Die zu sendende Botschaft, der Sendezeitpunkt und optional eine GPS-Position, sowie der neu erstellte private Paketschlüssel (siehe 5.7.4. Pakete verschlüsselt versenden) werden jeweils in einer `PacketContent`-Instanz abgelegt. Diese werden mit einer `PacketContentContainer`-Instanz zusammengefasst und mit Hilfe einer `SignContainers`-Instanz signiert (siehe 5.7.3. Signieren von Paketen und Paketinhalten). Der `SignContainer` wiederum wird mit Hilfe einer `CryptContainer`-Instanz verschlüsselt. In einer seperaten `CryptContainer`-Instanz wird ein `MagicWordPacketContent` ebenfalls verschlüsselt. Diese beiden `CryptContainer` werden nun gemeinsam mit dem *publicPacketKey*, der sich in einer `KeyContent`-Instanz befindet, in einem `PacketContentContainer` gepackt. Die daraus resultierenden Daten stellen die `PacketContent`-Instanz von `Packet` dar. Die Prüfsumme darüber ist die sog. *packetID*. Nun wird eine Instanz von `RoutingInfoContent` angefügt, die weder verschlüsselt, noch signiert wird.

6.2.2. öffentliche Lauffeuer-Pakete

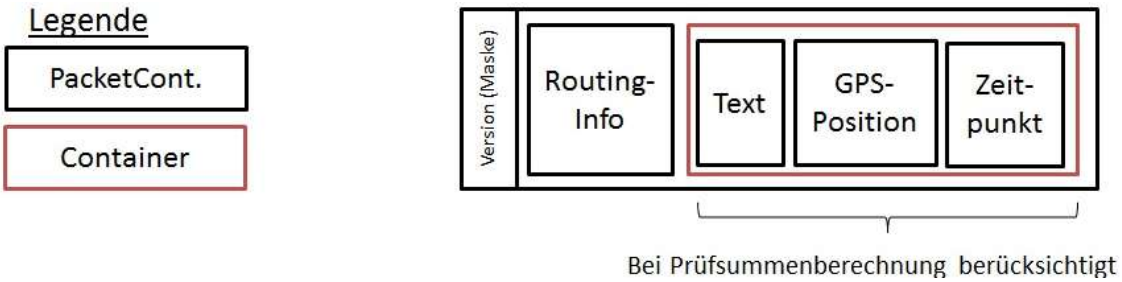

Abbildung 26 - öffentliches Lauffeuer-Paket

Die Sendezeit, der Text und die optionale GPS-Angabe werden je mit eine `PacketContent`-Instanz repräsentiert, die in einem `PacketContentContainer` zusammengefasst werden. Zusätzlich werden analog zu privaten Paketen Routinginformationen angefügt, die bei der Berechnung der Prüfsumme nicht berücksichtigt werden.

6.2.3. Präparieren von Paketen

Bevor Pakete an den Transmission-`Controller` gesendet werden, müssen noch folgende Schritte durchlaufen werden:

1) Trennung von Empfängern mit Lauffeuer-Profil und E-Mail-Empfängern:
   Alle E-Mail Empfänger werden je als `ReceiverIDContent-Instanz` in ein gemeinsames Paket hinzugefügt, welches an den Server adressiert wird. Für alle Empfänger, die ein Profil haben, wird je ein Paket erstellt.
2) Private Pakete in `Crypt-` bzw. `SignContainern` verpacken, sodass sie von Transmission verschlüsselt bzw. signiert werden.

6.3 GUI

Im *Namespace* GUI befinden sich alle nötigten Klassen für die Benutzeroberfläche der Anwendung, die auf dem *Smartphone* ausgeführt werden kann.

6.3.1. Konzept

Die zur Auswahl stehenden Funktionen ändern sich je nach Kontext, in dem sich der Benutzer gerade befindet (so steht z.B. die Funktion *Nachricht senden* nur im Kontext *Nachricht verfassen* zur Verfügung). Diese Kontexte werden in sogenannten Anzeigen (engl. *View*) zusammengefasst. Eine Anzeige kann mehrere Funktionen anbieten, ebenso kann eine Funktion unter mehreren Anzeigen verfügbar sein. Der Wechsel zwischen Anzeigen erfolgt ausschließlich durch Benutzereingaben.

Zur Verfügung stehen die Anzeigen:

- `ViewStartmenu`
- `ViewMessage` (Ansicht einer Nachricht nach /F120/ Nachricht lesen)
- `ViewMessageOverview` (nach /F050/ Nachrichtenübersicht)
- `ViewContactlist` (nach /F030/ Personen Sperren)
- `ViewSettings` (nach /F130/ Benutzereinstellungen)
- `ViewAboutUs`
- `ViewTutorial` (nach /F170/ Tutorial anzeigen)
- `ViewStatistics`

6.3.2. Benachrichtigungen auf der Benutzeroberfläche

Die Benutzeroberfläche bietet eine Schnittstelle, mit der eine Controller-Instanz die Benutzeroberfläche beeinflussen kann (durch sog. *Callbacks*). Dies ist dann nötig, wenn eine neue Nachricht empfangen, oder ein neuer Sychronisationspartner gefunden wurde. Befindet sich die Anwendung im Hintergrund, oder ist gerade eine andere Anzeige im Vordergrund, wird eine *ToastNotification* (siehe 4.2. Hintergrundprozesse) erzeugt. Durch Klicken auf diese *ToastNotification*, gelangt der Benutzer zu der entsprechenden Anzeige.

6.3.3. Mockups

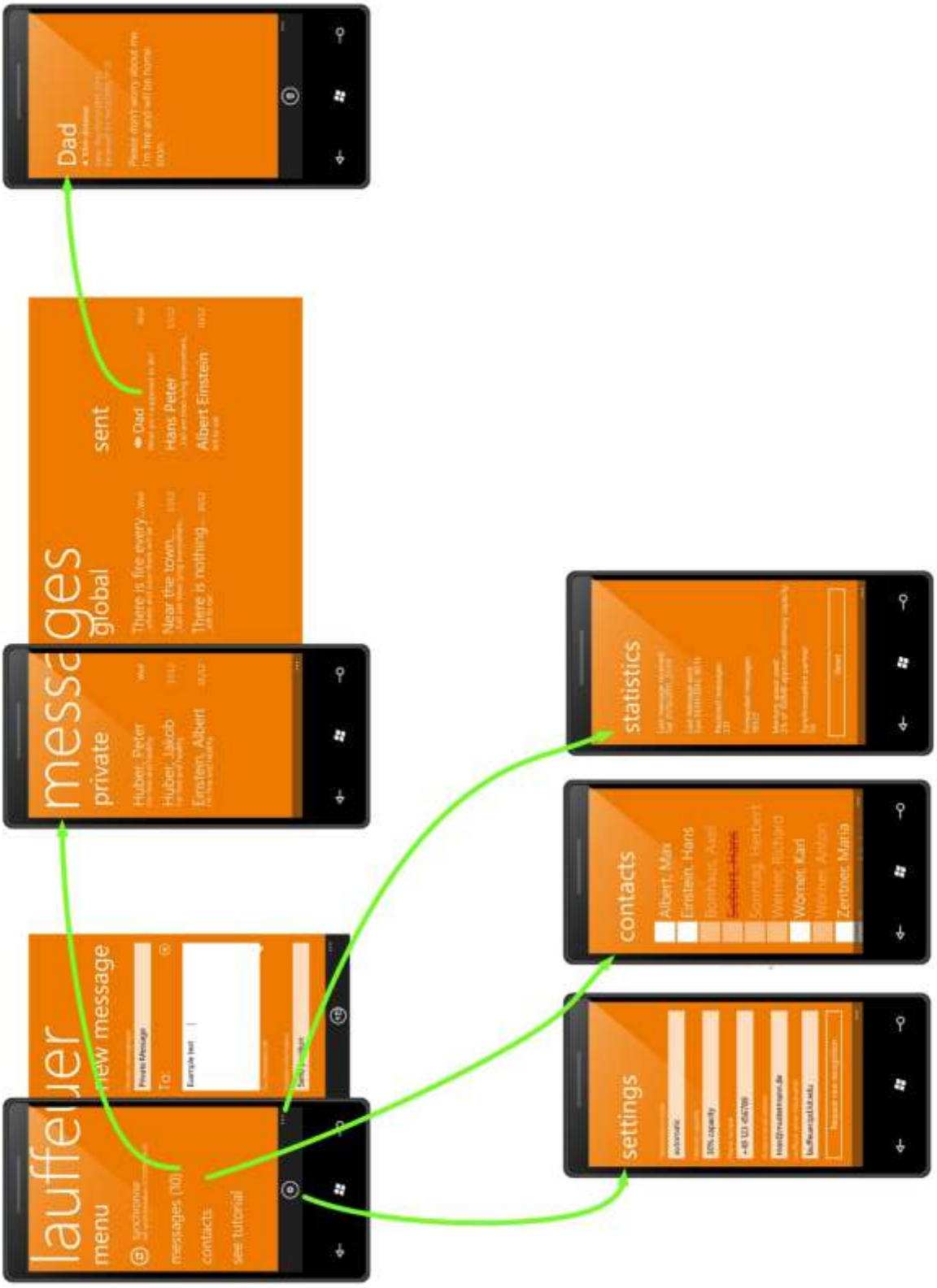

Abbildung 27 - Übergang zwischen Views

Abbildung 28 –Mockup `ViewSettings`

Abbildung 29 – Mockup `ViewStatistics`

Abbildung 30 - Mockup `ViewContactlist`.
Gesperrte Kontakte gestrichen, Ohne Profil grau

Abbildung 31 - `ViewMessage`

6.4. Controller

Die `Controller`-Klasse im *Namespace* `Lauffeuer` ist für die Koordination der Vorgänge der Anwendung zuständig und steuert den Kontrollfluss.

So verarbeitet der Controller unter anderem auch Anfragen, die durch Eingaben des Benutzers auf der GUI (siehe 6.3. GUI) ausgelöst wurden.

Die Aktionen, die die GUI ausführen kann sind:

- Personen sperren (`BlockPerson`)
- Neue Nachricht verfassen (`AddNewMessage`)
- Eigenes Profil löschen (`DeleteProfile`)
- Eigenes Profil ändern (`EditProfile`)
- Nachricht löschen (`RemoveMessage`)
- Synchronisationspartner suchen (`SearchPartner`)
- Pakete synchronisieren (`SyncPackets`)
- Personen entsperren (`UnblockPerson`)

Diese Aktionen können durch die Schnittstelle von `IController` aufgerufen werden.

Andersherum steuert eine `Controller`-Instanz zum einen auch die GUI, falls Ereignisse auftreten (z.B. die private Methode `EventNewPartner`) und kontrolliert zum anderen die `ApplicationDataStore`-Instanz. So kann `Controller` Instanzen von `AbstractModel` (z.B. `Person`, siehe 6.5.1. Lauffeuer-Models) in den `ApplicationDataStore` hinzufügen, ändern und löschen.

6.5. Lauffeuer-Models und Lauffeuer-Datastore

Die Anwendungsschicht (in diesem konkreten Fall: Lauffeuer) erweitert die durch Merkur bereitgestellten *Models*. Dazu wird ein weiterer *DataStore* (siehe 5.6. Models und DataStore) erstellt, der das Speichern und Laden der neu definierten *Models* ermöglicht.

6.5.1. Lauffeuer-Models

Alle Personen, die als Empfänger einer Botschaft in Frage kommen, werde mit der Klasse `Person` dargestellt, die wiederum von `Identity` erbt. Benutzer (also Personen die selbst die Anwendung verwenden) werden in einer Klasse repräsentiert, die von Person erbt (`User`). Der Besitzer der Anwendung wird in einer weiteren Klasse (`OwnersProfile`) gespeichert, die wiederum von `User` erbt. `OwnersProfile` besitzt neben dem Schlüssel (von der Klasse `User`) einen zusätzlichen privaten Schlüssel (vom Typ `Key`). Die Klasse `Server` ist eine Erweiterung der Klasse `Person`, da der Server ebenfalls adressiert werden kann (siehe 6.2. Lauffeuer-Pakete).

6.6. Lauffeuer-Server

Neben der Funktionalität die durch Merkur bereitgestellt wird (Pakete zwischen Server und *Smartphone* übertragen), müssen die Profile der Benutzer verwaltet werden.

6.6.1. Versenden von E-Mails

Abbildung 32 - Implementierung des Lauffeuer-Servers

Empfängt der Server ein neues Paket, wird der `ServerController` mit der Methode `ReceivedNewPacket` aufgerufen. Handelt es sich um ein entschlüsselbares Paket mit gültigem Absender, wird eine E-Mail mit Hilfe einer `IMailCreator`-Instanz (z.B. `PlainTextMailCreator`) erstellt. `IMailCreator`-Instanzen legen die Formatierung und den textuellen Rahmen einer E-Mail fest.

Zum Versenden wird die Klasse `SmtpClient` verwendet, bei der es sich um einen Adapter (gleichnamiges Entwurfsmuster) zum *Smtp-Client* handelt, der durch das *.NET-Framework* bereitgestellt wird.

6.6.2. Anlegen und Aktualisierung von Profilen auf dem Server

Jeder Benutzer besitzt ein Profil. Darin sind mindestens seine Handynummer und ein öffentlicher Schlüssel enthalten. Optional auch eine Emailadresse. Bei der Installation der Anwendung, wird ein Profil erstellt und auf dem Server gespeichert. Dieser Mechanismus wird nicht von Merkur bereitgestellt, sondern von der Anwendung Lauffeuer. Darum besitzt der Server einen weiteren Webservice (siehe 5.4. Nachrichtenzustellung über den Server), der zur Profilverwaltung dient.

Um Missbrauch vorzubeugen (siehe 2.2.7. Schutz vor Angriffen auf den Server) bekommt jeder Benutzer bei der Registrierung und jeder Profilaktualisierung eine SMS mit einem Registrierungscode an die angegebene Handynummer, mit dem er sein Profil freischalten muss. Solange ein Profil nach einer Aktualisierung nicht bestätigt wurde, ist die ältere Version gültig.

Aus Kostengründen werden im Rahmen des *PSE* keine echten SMS verschickt. Stattdessen werden SMS ähnliche Nachrichten auf der Konsole ausgegeben.

6.6.3. Emails auf Blacklist des Servers setzen

In jeder Email, die durch eine `IMailCreator`-Instanz (siehe 6.6.1. Versenden von E-Mails) erstellt wird, befindet sich einen URL auf eine ASPX-WebPage[7] (`Spamprotect`), die einen sog. *Keyphrase* enthält. Wird diese URL aufgerufen, prüft der `ServerController`, ob der *Keyphrase* bekannt ist und erstellt und speichert daraufhin ggf. eine neue `BlackListEntry`-Instanz.

Abbildungsverzeichnis

[7] http://msdn.microsoft.com/en-us/library/428509ah.aspx

Glossar

Wichtig: Dieser Glossar ergänzt den Glossar aus dem Pflichtenheft des Projektes Lauffeuer.

Botschaft

„Eine Nachricht oder Mitteilung von einem Sender an einen Empfänger, ursprünglich im Sinne einer durch einen Boten überbrachten Nachricht.“[8]

Broadcast

Bei einem Broadcast wird eine Nachricht innerhalb eines Netzes gleichzeitig an mehrere Empfänger übertragen.

Diese Nachricht geht von einem Punkt an alle anderen Teilnehmer des Netzes aus.

Hop

Wird ein Paket einmal übertragen, so hat das Paket einen Hop (Sprung) gemacht. Ein Paket wird für unterschiedliche Knoten unterschiedliche Hop-Anzahlen haben.

Prüfsumme

Eine Prüfsumme dient zur Gewährleistung der Datenintegrität, der Korrektheit der Daten.

Schlüssel

In der Kryptologie bezeichnet man eine Information, mit der man Klartext zu einem Geheimtext verschlüsseln kann, als Schlüssel.

Gleichzeitig kann man mit einem passendem Schlüssel Geheimtext in Klartext umwandeln. Im asymmetrischen Verschlüsselungsverfahren gibt es immer ein Schlüsselpaar, das aus einem öffentlichen und einem privaten Schlüssel besteht.

öffentlicher Schlüssel / publicKey

Mit einem öffentlichen Schlüssel können Klartexte zu einem Geheimtext verschlüsselt werden, die nur mit dem zugehörigen privaten Schlüssel wieder entschlüsselt werden können.

Wurde ein Klartext von einem privaten Schlüssel signiert, so kann diese Signierung durch den öffentlichen Schlüssel verifiziert werden.

privater Schlüssel / private Key

Mit einem privaten Schlüssel können Geheimtexte, die zuvor mit dem zugehörigen öffentlichen Schlüssel verschlüsselt wurden, entschlüsseln.

Außerdem können Klartexte mit diesem Schlüssel signiert werden.

Swipen

Eine Geste die der Benutzer auf einem *Touchscreen* vollführen kann. Es ähnelt einem seitlichem schubsen.

Knoten

Ein Netzwerkteilnehmer; *Smartphone* oder auch Server. Knoten sind in der Lage mittels P2P untereinander Pakete auszutauschen und zu speichern.

[8] Nach https://de.wikipedia.org/wiki/Botschaft, am 4.1.2012 0:40

ToastNotification

Eine Benachrichtigung, die auf dem Display des Benutzers angezeigt wird, das von einer Anwendung generiert wurde, die gerade nicht im Vordergrund ist.

Abbildung 33 - ToastNotification

Durch Tippen auf dieses Toast, gelangt der Benutzer zu der entsprechenden Anzeige der Anwendung.

Routing

Verfahren um Anzahl der Pakete im gesamten System aufgrund lokaler Entscheidungen zu optimieren bzw. minimieren.

TTL

TTL ist die Abkürzung für "Time to live" (Lebenszeit). Sie bestimmt die Gültigkeitsdauer von Daten.

Windows Server

Gleichnamiges Betriebssystem der Firma Microsoft für Server.