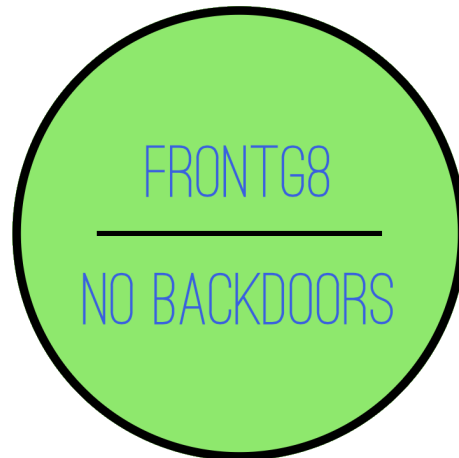




Projekt: frontg8

Software Architektur Dokument

Felix Morgner - fmorgner@hsr.ch
Tobias Stauber - tstauber@hsr.ch
Ueli Bosshard - ubosshar@hsr.ch



Änderungsgeschichte

Datum	Version	Änderung	Autor
06.04.2015	1.0		Ueli Bosshard
25.04.2015	1.1	Anpassung gemäss aktuellem Entwicklungsstand	Ueli Bosshard
29.04.2015	1.2	Überarbeitung der Clientarchitektur, Aktualisierung der Serverarchitektur	Ueli Bosshard
26.05.2015	1.3	Überarbeitung der Clientdokumentation	Ueli Bosshard



Inhalt

Änderungsgeschichte	2
Inhalt.....	3
1. Einführung	5
1.1 Zweck	5
1.2 Gültigkeitsbereich	5
1.3 Übersicht	5
2. Systemübersicht	6
2.1 Grundlegendes Konzept für den Austausch von Nachrichten	7
2.2 Verwendete Bibliotheken und Technologien.....	8
2.2.1 Protobuf.....	8
2.2.2 Asio	8
2.2.3 Posix AIO	8
2.2.4 Libconfig.....	8
2.2.5 pycrypto	8
2.2.6 pyelliptic.....	9
2.2.7 pyOpenSSL	9
2.2.8 asyncio	9
2.2.9 npyscreen.....	9
2.2.10 xmlrunner	9
3. Architektonische Ziele & Einschränkungen	10
3.1 Server	10
3.2 Client	10
4. Logische Architektur	11
4.1 Server: frontg8d	11
4.1.1 Klassenstruktur	12
4.1.2 Schnittstellen	13
4.1.3 Paket connectivity.....	14
4.1.4 Paket storage	17
4.1.5 Paket blacklisting	21
4.1.6 Paket messaging	22
4.1.7 Paket log	23
4.1.8 Wichtige interne Abläufe	24
4.2 Client: pyg8	25
4.2.1 Paket ui	26
4.2.2 Paket config	29
4.2.3 Paket crypto	30



4.2.4 Paket contact	31
4.2.5 Paket messaging	33
4.2.6 Paket connection	34
4.2.7 Paket protocol.....	36
4.2.8 Transitions	37
4.3 Wichtige Abläufe	38
4.3.1 Schlüsselaustausch	38
4.3.2 Nachricht versenden/empfangen	38
5. Prozesse und Threads	39
5.1 Server	39
5.2 Client	39
6. Deployment	40
6.1 Server	40
6.1.1 Toolchain zum Kompilieren des Sourcecodes.....	40
6.2 Client	40
6.2.1 Installation abhängiger Pakete	40
6.2.2 Patching protobuf für python	40
7. Datenspeicherung.....	41
7.1 Server	41
7.2 Client	41
8. Grössen und Leistung	42
9. Anhang.....	43
9.1 Links	43



1. Einführung

1.1 Zweck

Dieses Dokument zeigt die Software-Architektur für das Projekt frontg8

1.2 Gültigkeitsbereich

Das Dokument gilt für die Entwicklung der Software frontg8 während des SE2 Projekts im Frühlingsemester 2015.

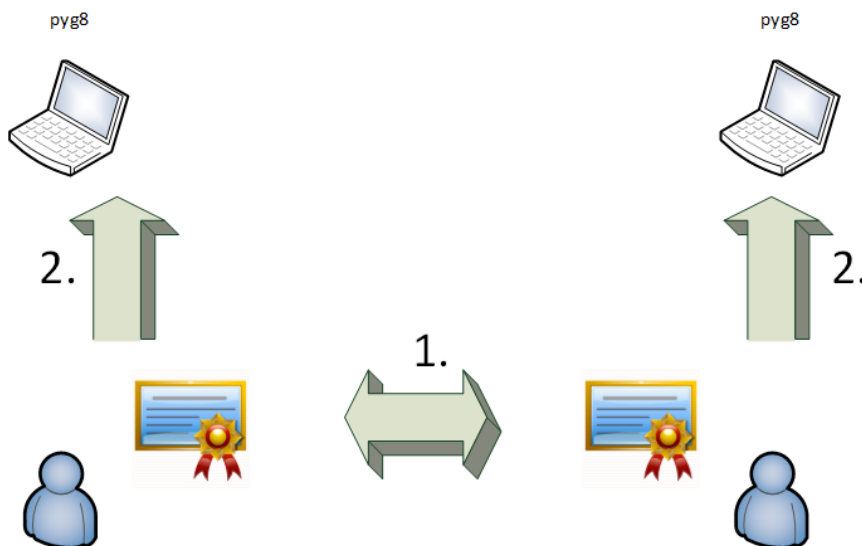
1.3 Übersicht

Dieses Dokument dokumentiert die Implementierung von Server und Clientkomponente. Details zur Kryptographie sind im „Crypto Design Dokument“ (Crypto_frontg8) festgehalten.

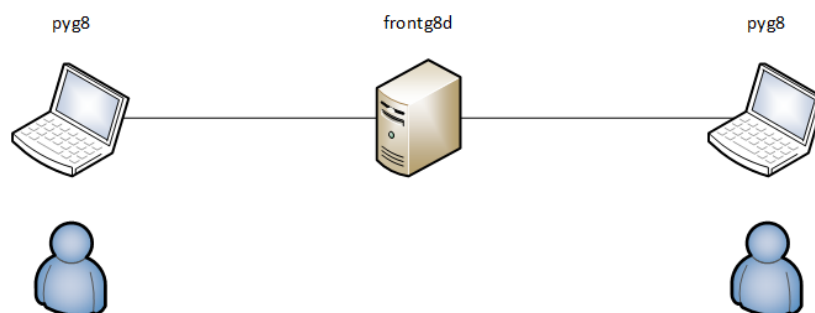
2. Systemübersicht

Im Lieferumfang des frontg8-Projekts befindet sich ein C++ Server „frontg8d“ und ein Python Client „pyg8“.

Frontg8 setzt starke Ende-zu-Ende-Verschlüsselung zwischen den Kommunikationspartnern ein. Der symmetrische Schlüssel zur Verschlüsselung der Nachrichten wird durch den Austausch von ECDH-Schlüsselteilen generiert. Zu diesem Zweck treffen sich die beiden Kommunikationspartner physisch und tauschen ihre öffentlichen Schlüssel aus.

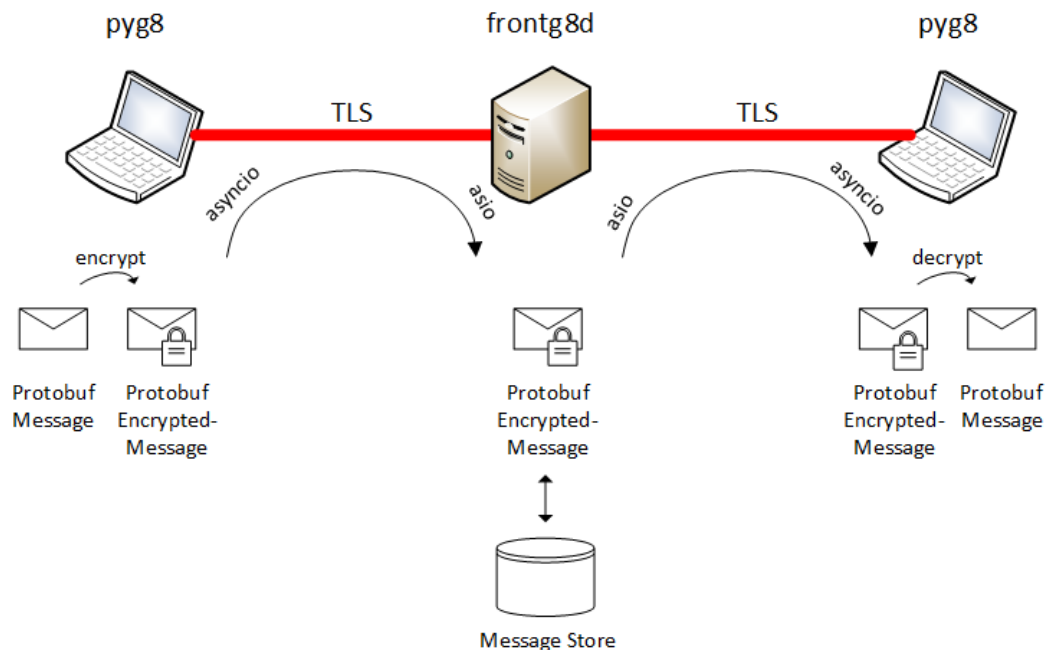


Nachdem beide Teilnehmer ein gemeinsames Geheimnis erarbeitet haben, sind sie in der Lage mit einander zu kommunizieren. Die gesamte Kommunikation wird asynchron über einen zentralen Server durchgeführt. Die Asynchronität ermöglicht es, dass beide Teilnehmer miteinander kommunizieren können, ohne jemals gleichzeitig mit dem Server verbunden sein zu müssen.



Eine genauere Darstellung der Funktionsweise des Gesamtsystems findet sich im Dokument "Domainanalyse_frontg8" im Kapitel "Systemübersicht". Eine detaillierte Beschreibung der eingesetzten Verschlüsselungstechnik findet sich im „Crypto Design Dokument“ (Crypto_frontg8).

2.1 Grundlegendes Konzept für den Austausch von Nachrichten



1. Die Nachricht wird in eine Protobuf¹ Nachricht verpackt
2. Die Protobuf Nachricht wird serialisiert und anschliessend verschlüsselt.
3. Die verschlüsselte Nachricht wird wiederum in eine Protobuf-Nachricht des Typs Encrypted verpackt und für den Versand serialisiert.
4. asyncio² hält eine TLS-Verbindung über TCP/IP zum Server offen und übergibt die Nachricht an den Server.
5. Auf dem Server kümmert sich Asio³ um die TLS-Verbindung und nimmt die verschlüsselte Nachricht entgegen. Die eingegangenen Nachrichten werden im Message Store vorgehalten welcher die nach wie vor verschlüsselten Nachrichten in einer RAM-Disk speichert.
6. Verbundene Clients werden über die neue Nachricht informiert. Clients die keine aktive Verbindung offen haben, bekommen die neue Nachricht, sobald sie beim Server nach neuen Nachrichten fragen.
7. Alle Clients holen die Nachricht ab, aber nur falls die Nachricht für sie bestimmt ist, können sie die Nachricht entschlüsseln.

¹ Protobuf wird in Kapitel 2.2.1 erklärt

² asyncio wird in Kapitel 2.2.8 erklärt

³ Asio wird in Kapitel 2.2.2 erklärt

2.2 Verwendete Bibliotheken und Technologien

2.2.1 Protobuf

Protobuf ist eine von Google entwickelte Bibliothek zur Serialisierung strukturierter Daten. Google bietet Implementationen für Java, C++ und Python an. Da die Bibliothek quelloffen ist, gibt es jedoch auch für diverse andere Sprachen, wie unter anderem C, Ruby und Haskell, Implementationen. Protobuf ist unter der 3-Klausel-BSD-Lizenz veröffentlicht.

Protobuf legt den Fokus auf Einfachheit und Leistung. Aus diesem Grund verwendet Protobuf ein Binärformat zur Repräsentation von Nachrichten. Die Kern-API ist über Sprachgrenzen hinweg konsistent gehalten und erleichtert so das Wechseln zwischen verschiedenen Programmiersprachen.

Wir setzen Protobuf ein, da es eine weit verbreitete und Plattformübergreifende Bibliothek ist. Zudem ist das Projekt lebendig und entwickelt sich kontinuierlich weiter.

2.2.2 Asio

Asio ist ein plattformübergreifendes C++ Toolkit für asynchrone Programmierung. Es wird meistens in der Netzwerkprogrammierung mit C++ eingesetzt, unterstützt aber auch asynchrone Signalbehandlung und andere I/O-Operationen. Eine grosse Stärke von Asio ist es, dass es eine konsistente API bietet und die jeweiligen Operationen mit den vom Betriebssystem bereitgestellten Mechanismen umsetzt.

Asio ist FLOSS (Free-Libre-Open-Source-Software) unter der Boost Software License.

Wir setzen Asio zur Kommunikation mit den Clients und den Aufbau einer TLS Verbindung ein. Asio selbst verwendet hierzu OpenSSL.

2.2.3 Posix AIO

AIO ist eine in „POSIX.1-2008 Base“ spezifizierte API, welche asynchrones schreiben von Dateien erlaubt. Aufgrund des Umstands, dass Asio keine asynchronen Dateioperationen erlaubt, setzen wir zur Speicherung von Dateien AIO ein.

AIO ist seit 2008 in der „POSIX.1 Base Specification“ enthalten, und viele weitverbreitete Betriebssysteme implementieren es. Diese breite Unterstützung erlaubt es uns diese Technologie zu verwenden ohne die angestrebte Portabilität zu gefährden.

2.2.4 Libconfig

Libconfig ist eine Bibliothek, welche die Verarbeitung und Erstellung von Konfigurationsdateien erheblich vereinfacht. Die Bibliothek ist auf Kompaktheit und Effizienz ausgelegt und ist dadurch sehr gut für Systeme mit beschränkten Systemen geeignet. Libconfig ist unter der GNU Lesser General Public License (LGPL) veröffentlicht.

2.2.5 pycrypto

PyCrypto ist ein Wrapper für kryptographische Funktionen, welcher im Hintergrund OpenSSL verwendet.

2.2.6 pyelliptic

PyElliptic ist ein highlevel Wrapper für kryptografische Funktionen im Bereich elliptische Kurven. Im Hintergrund werden Funktionen von OpenSSL verwendet.

2.2.7 pyOpenSSL

pyOpenSSL ist ein Wrapper für den Zugriff auf Funktionen von OpenSSL.

Wir verwenden pyOpenSSL für die Generierung von X.509-Zertifikaten im Zusammenhang mit der TLS-Verbindung zwischen Server und Client.

2.2.8 asyncio

asyncio ist ein Python-Modul, welches eine Infrastruktur für das Multiplexing von Input-/Output-Zugriffen ermöglicht.

Wir verwenden asyncio um die TCP/IP-Verbindung im Hintergrund offen zu halten.

2.2.9 npyscreen

npyscreen ist eine Widget-Bibliothek für Python, welche im Hintergrund auf Ncurses aufsetzt.

Wir verwenden npyscreen für das interaktive Benutzerinterface im Client.

2.2.10 xmlrunner

xmlrunner wird benötigt, um die Unit-Tests von Python in einem von Jenkins lesbaren XML-Format zu generieren.

3. Architektonische Ziele & Einschränkungen

3.1 Server

Die Architektur des Servers wird vor allem durch drei Faktoren beeinflusst: Geheimhaltung, Schlankheit und Portabilität.

An vorderster Stelle steht hier die **Geheimhaltung**. Der Server erfasst nur ein Minimum von Daten. Beispielsweise ist es nicht vorgesehen, dass der Server ein Logfile mit Zuordnungen von IP-Adressen zu Hashes der eingelieferten Nachrichten führt. Des Weiteren speichert der Server die empfangenen Nachrichten nur für einen begrenzten Zeitraum (Standard 3 Tage). Eine weitere Schutzmassnahme ist, dass die empfangenen Nachrichten in regelmässigen Abständen gespeichert werden. Dies soll eine Korrelation von Nachrichten über ihren Zeitstempel erschweren. Ein Angriffspunkt ist hier jedoch das vom Server geführte und verschlüsselt abgelegte Journal, welches er benötigt, um auf Message-Request Nachrichten reagieren zu können.

Ein zweiter wichtiger Faktor ist '**Schlankheit**'. Um dieses Ziel zu erreichen, wird beispielsweise die non-Boost Version von Asio eingesetzt. Dies erlaubt es uns, die Abhängigkeiten so gering wie möglich zu halten und trotzdem nicht auf moderne Toolkits zu verzichten. Zur Verwaltung der gespeicherten Daten setzen wir eine eigens entwickelte Infrastruktur ein und verzichten hier auf den Einsatz von Boost::Filesystem.

Der dritte grosse Faktor ist die **Portabilität** des Quellcodes. Zu diesem Zweck halten wir uns an Standards wie POSIX und ISO C++. Wir verwenden keine GNU-Erweiterungen der Programmiersprache C++, um so für die Zukunft in der Wahl von Zielplattformen flexibel zu bleiben.

3.2 Client

Bei dem Python-Client handelt es sich um einen Prototyp, der als Referenzimplementierung für mobile Clients dient (zum Beispiel Android, iOS oder Windows Phone). Dabei wird ausschliesslich Wert auf die **grundlegende Funktionalität** gelegt um zu zeigen, wie das System funktioniert.

Einschränkungen ergeben sich durch die Nutzung einer interpretierten Programmiersprache. Python unterstützt prinzipiell Multithreading, die von uns eingesetzte Implementation (CPython) verwendet jedoch das sogenannte „Global Interpreter Lock (GIL)“, was bedeutet, dass immer nur ein Thread gleichzeitig laufen kann.

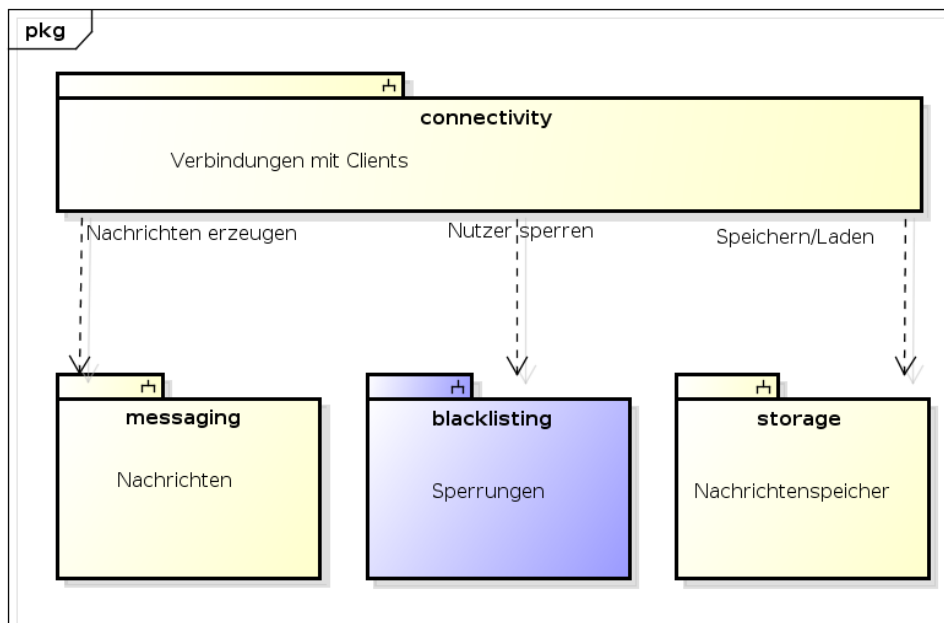
Bei dem Userinterface **verzichten wir bewusst auf graphisches System**, da es für die Machbarkeit des Projekts nebensächlich ist. Zu Beginn konzentrieren wir uns auf ein Command-Line-Tool um das Versenden und Empfangen von Nachrichten zu zeigen. Anschliessend implementieren wir ein einfaches Userinterface auf Basis von Ncurses, damit wir interaktiv mit dem Programm agieren können, beispielsweise wenn im Hintergrund eine neue Nachricht eintrifft.

Beim generieren von kryptographischen Schlüsseln und den Ver- und Entschlüsselungsoperationen setzen wir auf vorhandene Bibliotheken wie pycrypto, pyelliptic und pyOpenSSL, die schlussendlich alle auf OpenSSL aufsetzen. Durch den Einsatz dieser Bibliotheken setzen wir uns dem Risiko aus, dass sich fremde Fehler in unsere Software einschleichen. Trotzdem setzen wir in unserem Projekt auf bewährte Bibliotheken, damit wir uns nicht auf eine eigene, ungeprüfte Implementation verlassen müssen.

4. Logische Architektur

4.1 Server: frontg8d

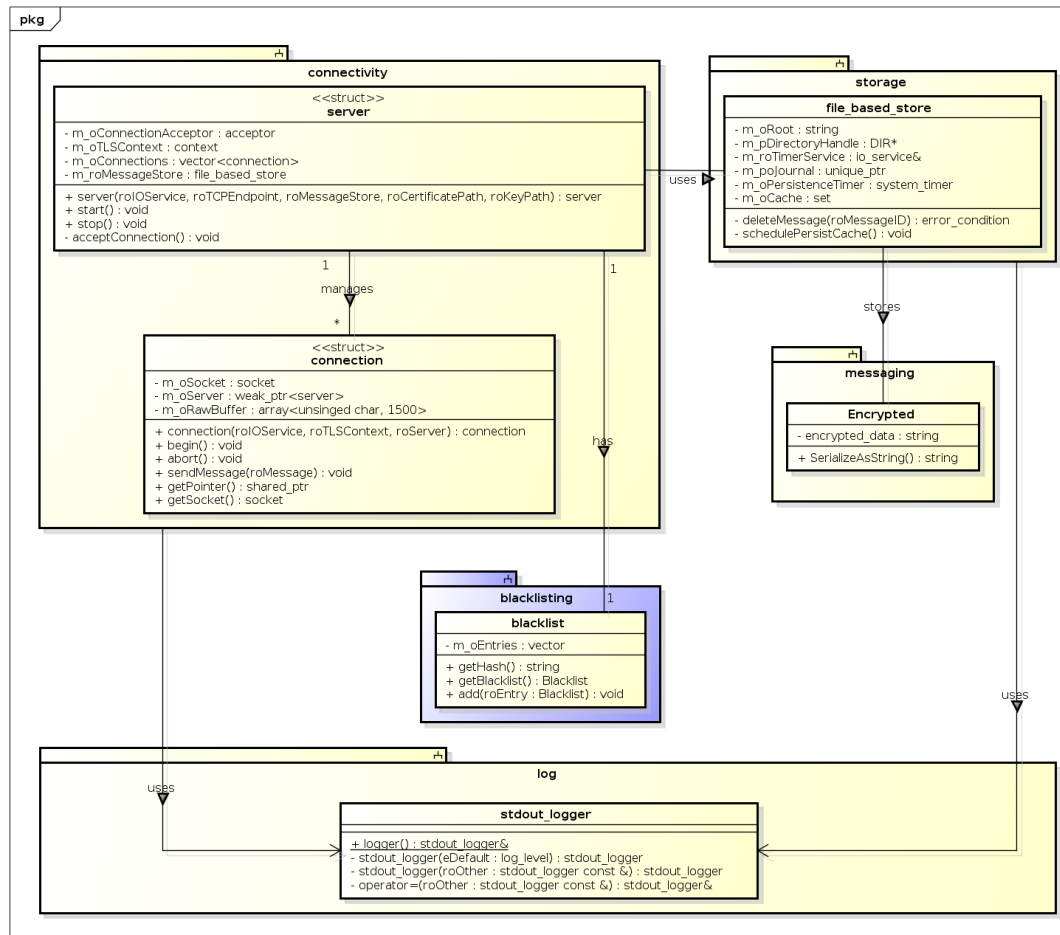
Der C++ Server (frontg8d) kann in folgende logische Subsysteme unterteilt werden:



1. Das "connectivity" Subsystem. Dieses Subsystem ist für die Verwaltung von Verbindungen zu entfernten Clients zuständig. Es besteht weitestgehend aus der 'server' sowie der 'connection' Klasse. Die 'server' Klasse ist für das Entgegennehmen von Verbindungen und das Erzeugen von Objekten des Typs 'connection' zuständig.
2. Das "messaging" Subsystem. Dieses Subsystem ist zuständig für die Erzeugung und Verarbeitung von Nachrichten. Es besteht aus den durch Google Protobuf erzeugten Nachrichten-Klassen ('Encrypted', 'Notification', 'Blacklist', 'Revocation' und 'MessageRequest') sowie einigen 'free-functions'. Diese 'free-functions' sind für das Packen und Entpacken von Nachrichten, sowie das Erzeugen von Antwortnachrichten zuständig.
3. Das "storage" Subsystem. Dieser Layer ist verantwortlich für das Speichern und Laden von Nachrichten aus dem externen Speicher. Er ist auch dafür zuständig, das Journal zu führen und zu schreiben. (Für eine Erklärung zum Journal siehe Kapitel „Paket storage“)
4. Das "blacklisting" Subsystem. In diesem Subsystem wird die Verwaltung der Blacklist realisiert. Es speichert Blacklist Einträge und berechnet den aktuellen Blacklist-Hash.

4.1.1 Klassenstruktur

Nachfolgend sind die wichtigsten Interaktionen und Schnittstellen zwischen den verschiedenen Subsystemen dargestellt. Vollständige Klassendiagramme der Subsysteme befinden sich in den jeweiligen Kapiteln.

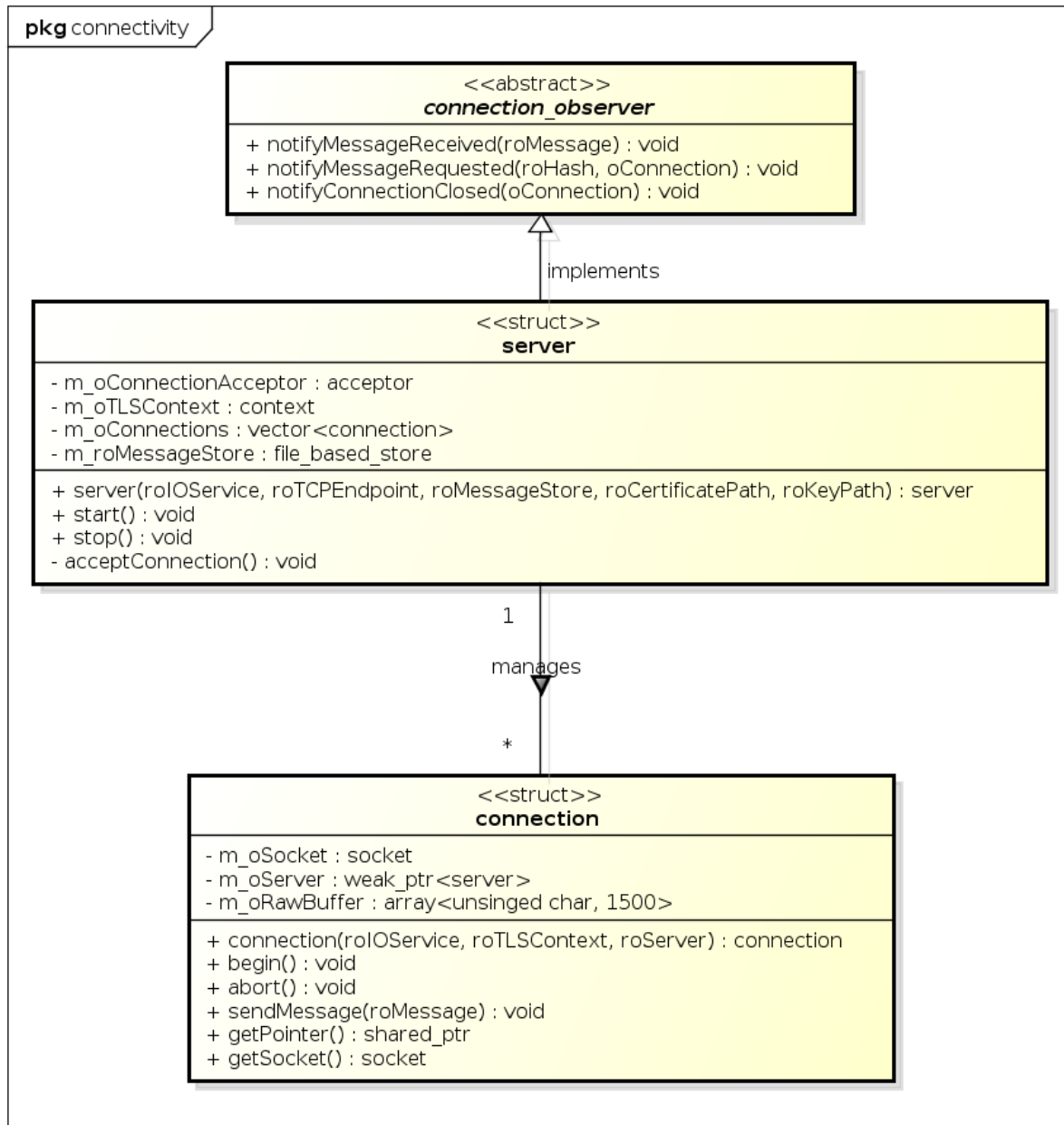


4.1.2 Schnittstellen

Schnittstelle	Beschreibung
server - file_based_store	Eine Instanz der Klasse server besitzt eine Referenz auf den globalen file_based_store. Sie verwendet die vom Interface 'basic_message_store' angebotenen Funktionen um mit dem Storage-Backend zu interagieren.
server - connection	Die 'server'-Instanz erzeugt Instanzen der Klasse 'connection'. Die 'server'-Instanz besitzt eine Liste mit Zeigern auf die von ihr erstellten 'connection's. Der Server ist des Weiteren in der Lage Verbindungen abubrechen und Benachrichtigungen an verbundene Clients zu schicken.
server - blacklist	Der Server besitzt eine Instanz der Klasse 'blacklist'. Diese verwendet er, um Nachfragen von Clients bezüglich der Blacklist zu verarbeiten. Diese Nachfragen beinhalten das Abfragen der aktuellen Blacklist, sowie das Hinzufügen von Einträgen.
connection - server	Eine 'connection' verwendet die durch die Schnittstelle 'connection_observer' angebotenen Funktionen, um den Server über Ereignisse zu informieren.

4.1.3 Paket connectivity

Dieses Subsystem ist für den Aufbau und die Verwaltung von Verbindungen zwischen dem Server und den Clients zuständig.



4.1.3.1 Desginentscheidungen

Die abstrakte Basisklasse **connection_observer** wurde eingeführt um folgende Ziele zu erreichen:

Geringe Kopplung: Die Verwendung einer abstrakten Basisklasse erlaubt es der Klasse 'connection' der umgebenden Infratraktur Mitteilungen über Ereignisse zukommen zu lassen, ohne von der konkreten Implementation des Observers abhängig zu sein.

Erweiterbarkeit: Die Extraktion der Schnittstelle ermöglicht eine einfache Erweiterbarkeit der Funktionalität. Es wäre zum Beispiel denkbar, eine Implementation von 'connection_observer' zu entwickeln, welche über Netlink-Sockets mit der Firewall kommuniziert. Dadurch wäre es möglich, auf Protokollverletzungen oder einen vermeintlichen Angriff, direkt zu reagieren.

4.1.3.2 Klassenbeschreibungen

Klasse	Beschreibung
connection_observer	Diese abstrakte Klasse dient als Schnittstellenbeschreibung für Klassen welche Verbindungen beobachten können. Im Moment ist die einzige Klasse welche sie implementiert die Klasse 'server'. Verbindungen können die angebotenen Funktionen nutzen um einen Beobachter über Ereignisse zu informieren. notifyMessageReceived(roMessage): Verbindungen nutzen diese Funktion um Beobachter über den Eingang einer Encrypted-Nachricht zu informieren. notifyMessageRequested(roHash, oConnection): Verbindungen nutzen diese Funktion um Beobachter über den Eingang einer MessageRequest-Nachricht zu informieren. notifyConnectionClosed(oConnection): Verbindungen nutzen diese Funktion um Beobachter darüber zu informieren, dass die Verbindung zum Client geschlossen wurde.
server	Diese Klasse erstellt ein Socket, auf welchem sie auf eingehende Verbindungen wartet. Sie initialisiert auch den für TLS benötigten Kontext. Die 'server' Klasse stellt das Interface zum 'storage' Subsystem dar, und verwendet den Message Store für das Speichern und Laden von Nachrichten. Sie implementiert die 'connection_observer' Schnittstelle. start(): Mit dieser Funktion wird der Server gestartet. Sie ist für das Starten des Run-Loops zuständig und initialisiert den "hörenden" Socket. stop(): Diese Funktion wird für das Beenden des Server verwendet. Sie schliesst den Message Store und beendet alle Client-Verbindungen.

connection

Diese Klasse repräsentiert eine Verbindung zu einem Client. Sie transportiert Daten zwischen Client und Server. Sie verwendet die beiden "free-functions" `pack(...)` und `unpack(...)` um Nachrichten für den Transport zu packen bzw. beim Empfang zu entpacken.

begin():

Führt den TLS-Handshake aus und baut damit einen sicheren Kanal zu einem Client auf.

abort():

Bricht eine Verbindung zu einem Client ab.

sendMessage(roMessage):

Sendet eine Nachricht eines beliebigen Nachrichten-Typs an einen Client.

getPointer():

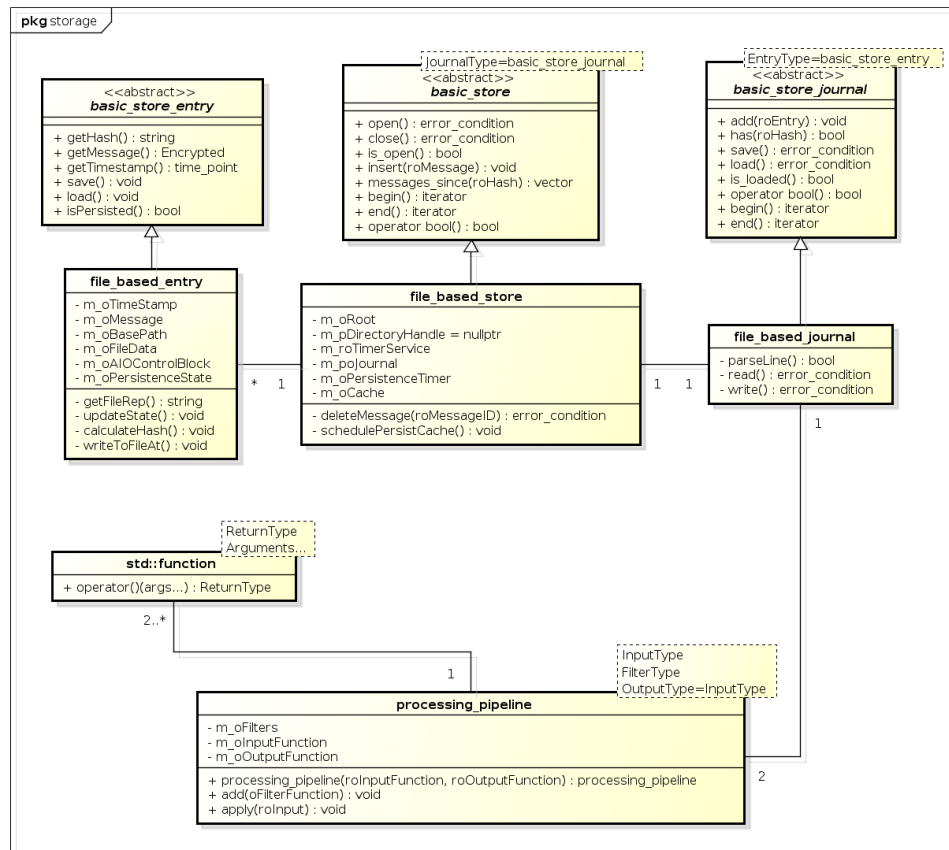
Gibt einen 'shared_ptr' auf die Instanz zurück. Diese Funktion wird vom Server verwendet um eine Verbindung in seine Verbindungssammlung aufzunehmen.

getSocket():

Gibt einen rohen TCP Socket zurück. Der Server verwendet diese Funktion für die Etablierung einer TCP-Verbindung zu einem Client.

4.1.4 Paket storage

Das 'storage' Subsystem ist für das Verwalten von Nachrichten und deren Sicherung zuständig. Das Storage-Subsystem führt ein Journal, in welchem es den SHA-256 Hash einer Nachricht ihrem Eingangszeitpunkt zuordnet. Dieses Journal ist nötig, da der Server nicht am Timestamp der abgelegten Dateien erkennen kann, wann eine Nachricht eingeliefert wurde.



4.1.4.1 Designentscheidungen

Das 'storage' Paket verwendet diverse abstrakte Basisklassen um folgende Ziele zu erreichen:

Geringe Kopplung: Die Verwendung von abstrakten Basisklassen erlaubt es dem Server, von der konkreten Implementierung des Message Stores unabhängig zu sein.

Bessere Testbarkeit: Ein- und Ausgabe ist schwer testbar, vor allem wenn Dateien als Quellen oder Ziele verwendet werden. Die Klasse 'stream_based_journal' beispielsweise mindert dieses Problem, indem sie es erlaubt, den verwendeten Stream zum Testen auszutauschen. 'file_based_journal' ist eine Template-Instanziierung der Klasse 'stream_based_journal' welche den Template-Parameter 'StreamType' auf 'std::fstream' bindet.

Erweiterbarkeit: Da die Verwendung unabhängig von der Implementierung möglich ist, lässt sich die Storage-Infrastruktur leicht auf alternative Backends (z.B. Redis, Socket, etc.) erweitern. Die Extraktion der Verarbeitung der Journal-Einträge ermöglicht es zum Beispiel einfach andere Verschlüsselungsverfahren einzusetzen.

4.1.4.2 Klassenbeschreibungen

Klasse	Beschreibung
basic_store	Diese Klasse definiert die Schnittstelle, welche ein Message Store unterstützen muss. Der Server verwendet eine Implementation dieser Schnittstelle um Nachrichten zu speichern und abzurufen. open(): Wird zum Öffnen eines Stores verwendet. Beispiele sind hier das Öffnen von Datenbankverbindungen oder Verzeichnissen. close(): Schliesst einen Message Store und seine möglicherweise assoziierten Handles. is_open(): Prüft ob ein Message Store offen ist. insert(roMessage): Fügt Nachrichten zum Message Store hinzu. messages_since(roHash): Holt alle Nachrichten aus dem Message Store, welche nach der Nachricht mit dem angegebenen Hash eingegangen sind. operator bool(): Siehe is_open() begin(): Gibt einen Iterator auf den ersten Eintrag des Stores zurück. end(): Gibt einen Iterator hinter den letzten Eintrag des Stores zurück. journal(): Gibt eine Referenz auf das Journal des Stores zurück.
basic_store_entry	Diese Klasse definiert die Schnittstelle, welche ein Eintrag für einen Message Store implementieren muss. getHash(): Liefert den SHA256 Hash eines Eintrags.

getMessage():

Liefert die assoziierte Nachricht zurück.

getTimestamp():

Liefert den Erstellungszeitstempel des Eintrags zurück.

save():

Speichert den Eintrag.

load():

Lädt einen Eintrag.

isPersisted():

Überprüft, ob ein Eintrag persistiert ist.

basic_store_journal

Diese Klasse definiert die Schnittstelle, welche ein Journal für einen Message Store implementieren muss.

add(roEntry):

Fügt dem Journal einen Eintrag hinzu.

has(roHash):

Prüft ob das Journal einen Eintrag für den angegebenen Hash besitzt.

save():

Speichert das Journal.

load():

Lädt das Journal.

is_loaded():

Prüft ob das Journal geladen wurde.

operator bool():

siehe is_loaded()

begin():

Gibt einen Iterator auf den ersten Eintrag des Journals zurück.

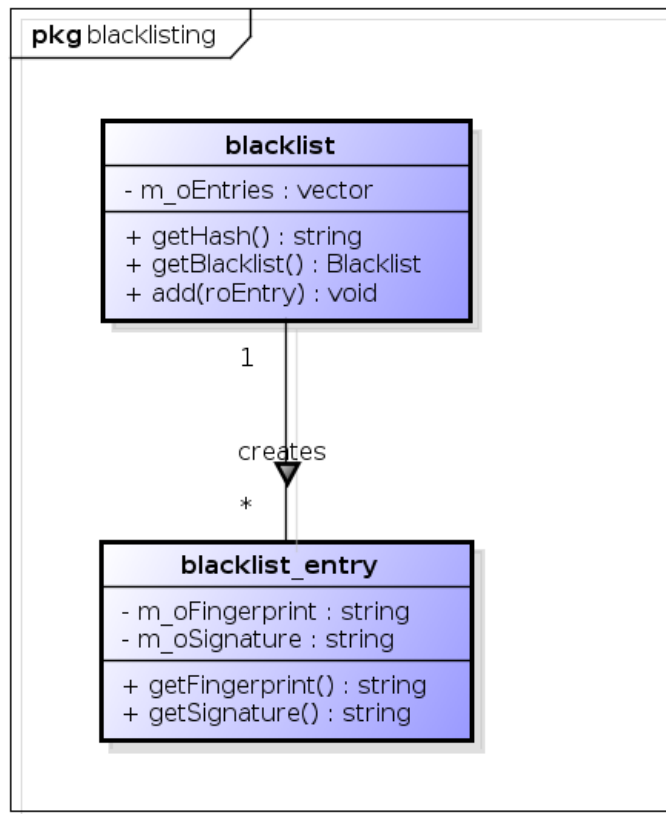
end():

Gibt einen Iterator hinter den letzten Eintrag des Journals zurück.

processing_pipeline	Diese Klasse stellt eine konfigurierbare Verarbeitungs- und Filterungs-Pipeline dar. Sie verwendet eine Output- sowie eine Input-Funktion um Output und Input zu transformieren. Sie hält eine Liste von Filter- bzw. Transformationsfunktionen, welche in der abgelegten Reihenfolge angewendet werden. apply(roInput): Startet die Verarbeitung. add(oFilterFunction): Fügt eine Filter- bzw. Transformationsfunktion hinzu.
file_based_entry	Spezialisierung der Schnittstelle ‚basic_store_entry‘ für das Speichern in Dateien. Verwendet intern die POSIX.1 AIO API.
file_based_store	Spezialisierung der Schnittstelle ‚basic_store‘ für das Speichern des Stores in Dateien. Verwendet intern ‚file_base_entry‘ und ‚file_based_journal‘ zur Speicherung.
stream_based_journal	Spezialisierung der Schnittstelle ‚basic_store_journal‘ für die Ausgabe auf beliebige Streams.
file_based_journal	Spezialisierung der Klasse ‚stream_based_journal‘ für die Ausgabe auf ‚std::fstream‘

4.1.5 Paket blacklisting

Dieses Paket ist für die Verwaltung der Schlüssel-Sperrliste verantwortlich. **Dieses Paket ist noch nicht implementiert.**



4.1.5.1 Klassenbeschreibungen

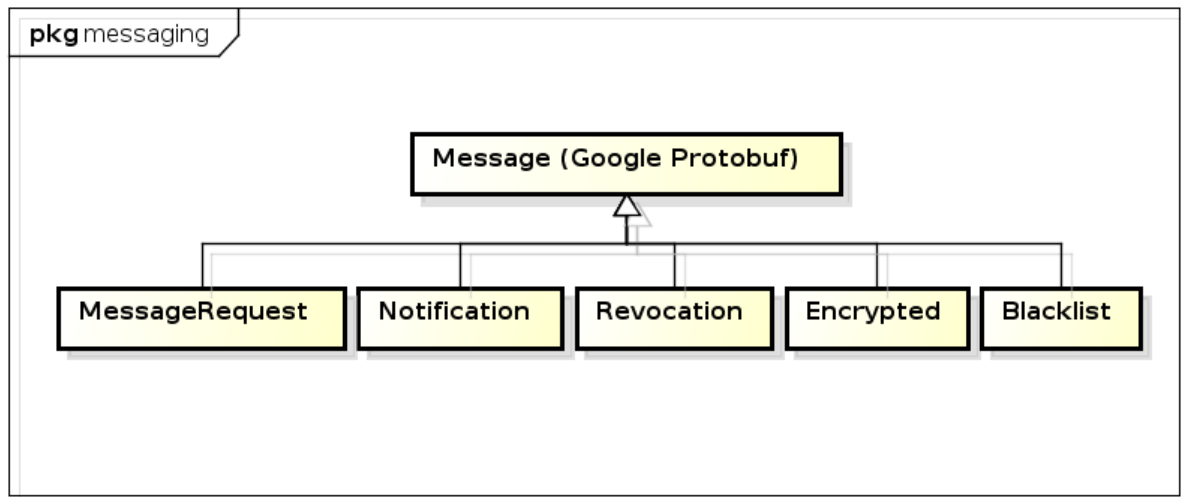
Klasse	Beschreibung
blacklist	Diese Klasse verwaltet Blacklist-Einträge und ist für die Bearbeitung von Blacklist-Nachrichten zuständig. getHash(): Liefert den aktuellen Blacklist Hash. getBlacklist(): Liefert die aktuelle Blacklist verpackt in einer Blacklist Nachricht. add(roEntry): Fügt der Blacklist einen Eintrag hinzu.
blacklist_entry	Diese Klasse repräsentiert einen Sperrlisten-Eintrag. getFingerprint(): Liefert den Fingerprint des gesperrten Schlüssels.

getSignature():

Liefert die Unterschrift über den gesperrten Schlüssel.

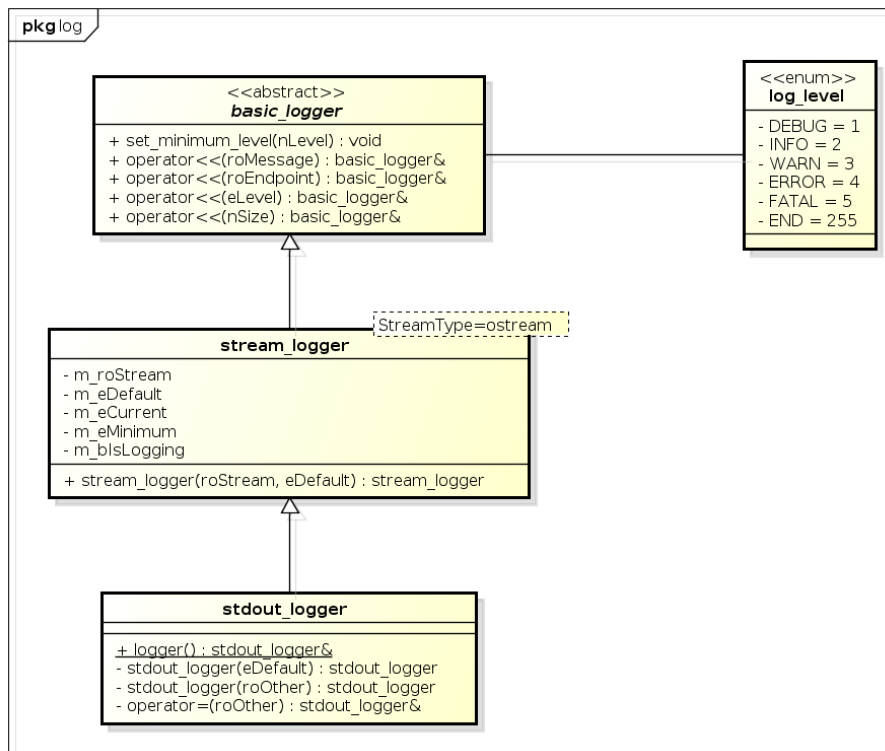
4.1.6 Paket messaging

Das 'messaging' Subsystem bietet die Infrastruktur für das Erzeugen und Serialisieren von Protokoll-Nachrichten an. Das Paket selber wird von Google Protobuf zur Verfügung gestellt. Aus diesem Grund wird auf den internen Aufbau des Pakets nicht genauer eingegangen.



4.1.7 Paket log

Das Subsystem log bietet grundlegende Loggingfunktionalität an.



4.1.7.1 Klassenbeschreibungen

Klasse	Beschreibung
loglevel	Enthält feste Werte, welche die verschiedenen Loglevel beschreiben.
basic_logger	Diese Klasse definiert die Schnittstelle, welche von konkreten Loggern implementiert werden muss. set_minimum_level(nLevel): Setzt den minimalen Loglevel. operator<<(...): Gibt das Argument auf dem Logger aus.
stream_logger	Spezialisierung der Schnittstelle basic_logger für die Ausgabe auf beliebige Steams.
stdout_logger	Spezialisierung der Klasse stream_logger für die Ausgabe auf std::out.

4.1.8 Wichtige interne Abläufe

Beim Start der Applikation wird eine Instanz der Klasse 'server' erzeugt. Nachdem auch der Message Store initialisiert wurde, wird der Main-Run-Loop mittels der Member-Funktion 'start()' der Klasse 'server' gestartet. Dieser Aufruf ist blockierend und kehrt erst zurück wenn der Server mittels 'stop()' angehalten wird.

Instanzen der Klasse 'connection' kommunizieren niemals direkt miteinander. Sie verwenden die Schnittstelle 'connection_observer' um den Server über Ereignisse zu Informieren. Dieser löst dann die verlangten Ereignisse auf den anderen Verbindungen aus.

'connection' Instanzen kommunizieren auch nie direkt mit dem Message Store, sondern verwenden auch hier die Schnittstelle 'connection_observer'.

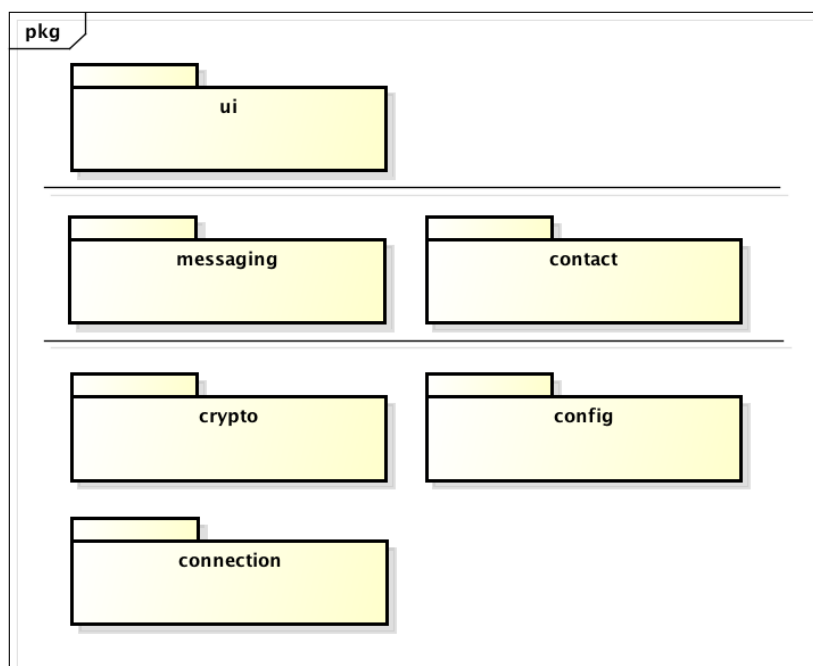
4.2 Client: pyg8

Für die Architektur des Clients haben wir uns überlegt, was für Concerns und Experts in unserem System vorhanden sind und welche Experts sich um welche Concerns kümmern.

Concerns	Experts
Kontakte ver-/entschlüsseln	contact/crypto
Nachricht ver-/entschlüsseln	contact/crypto
Server Kommunikation	connection
Nachricht aufbereiten für Versand/Verarbeitung	messaging
Datennachricht anzeigen	ui
Kontakt sperren	contact
Konfigurationswert liefern	config

Die ermittelten Experts haben wir jeweils in ein eigenes Paket gesteckt, damit wir den Code sauber strukturieren können. In den folgenden Kapiteln wird auf jedes Paket eingegangen.

Packages:

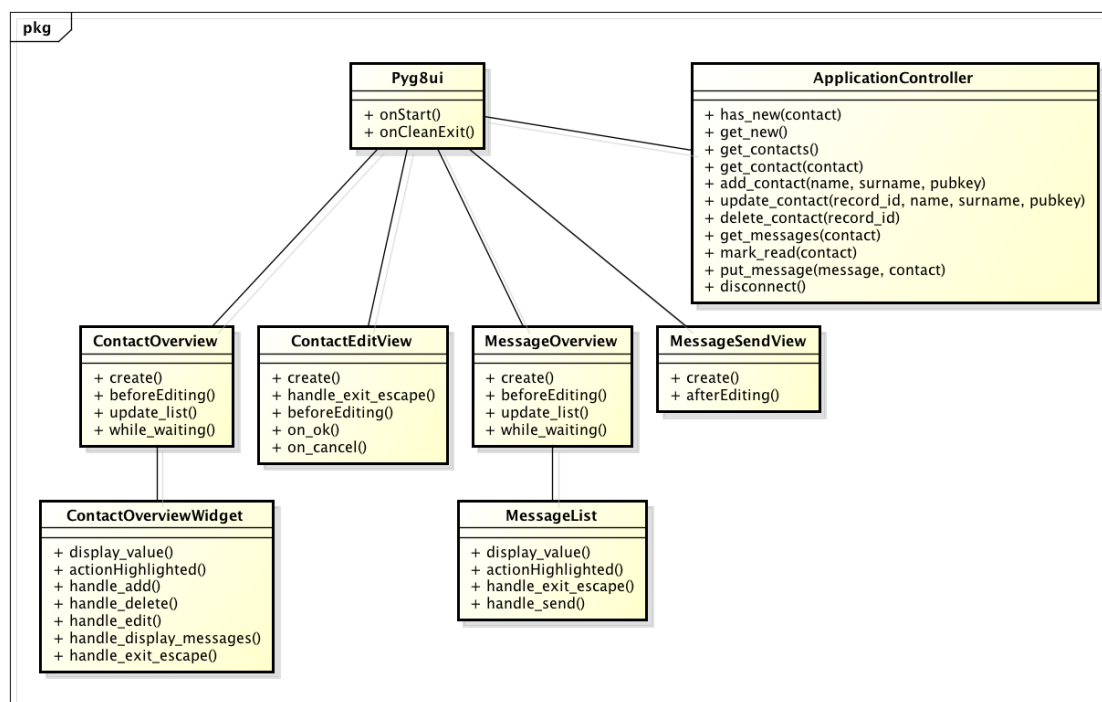


4.2.1 Paket ui

Das Paket ui beinhaltet das Terminal-Userinterface (TUI), welches mit Ncurses umgesetzt wird.

Wir verwenden npyscreen welches uns den Umgang mit Nurses stark vereinfacht. npyscreen zeichnet das Userinterface in dem Formulare definiert werden. Wir haben die Klassenbeschreibung nur knapp beschrieben, da die Funktionen in npyscreen sehr intuitiv zu benutzen sind.

4.2.1.1 Klassenstruktur



4.2.1.2 Wichtige interne Abläufe

Beim Start der Applikation wird mittels eines Komandozeilenarguments entschieden, ob im Commandline- oder im TUI-Modus gearbeitet wird. Wird pyg8 im TUI Modus gestartet, wird ein Pyg8ui instanziiert, welches dann die Interaktion mit dem Benutzer durchführt.

4.2.1.3 Klassenbeschreibungen

Klasse	Beschreibung
Pyg8ui	Startet das Ncurses-Userinterface. onStart(): Initialisiert alle Formulare. onCleanExit(): Delegiert das Beenden der Applikation an den Application-Controller.
ApplicationController	Der ApplicationController kümmert sich um den Datenaustausch zwischen User-Interface und den übrigen Programmkomponenten.

has_new(contact):

Gibt zurück, ob für ein angegebener Kontakt eine neue Nachricht vorliegt.

get_new():

Gibt alle noch nicht angezeigten Nachrichten zurück.

get_contacts():

Gibt alle Kontakte zurück.

get_contact(contact):

Gibt einen bestimmten Kontakte zurück.

add_contact(name, surname, pubkey):

Fügt einen neuen Kontakt hinzu.

update_contact(record_id, name, surname, pubkey):

Ändert einen Kontakt.

delete_contact(record_id):

Löscht den angegebenen Kontakt.

get_messages(contact):

Liefert alle Nachrichten des angegebenen Kontakts.

mark_read(contact):

Markiert die Nachrichten des angegebenen Kontakts als gelesen.

put_message (message, contact):

Markiert die Nachrichten des angegebenen Kontakts als gelesen.

disconnect():

Wird beim Beenden des TUI aufgerufen um die offenen Netzwerkverbindungen zu schliessen.

ContactOverview	Die bei den *View-Klassen handelt es sich um Formulare anhand derer npyscreen das TUI-Interface realisiert.
ContactEditView	
MessageOverview	
MessageSendView	
ContactOverviewWidget	
MessageList	create(): Platziert die Elemente auf dem Formular.

beforeEditing():

Befüllt die Textfelder mit sinnvollen Anfangswerten.

afterEditing():

Wird ausgeführt, wenn das Formular geschlossen wird.

on_ok():

Behandelt das Auswählen des OK-Buttons.

on_cancel():

Behandelt das Auswählen des Cancel-Buttons.

handle_*():

Behandelt den jeweiligen Shortcut.

update_list()

Aktualisiert die angezeigte Liste (refresh)

actionHighlighted():

Behandelt das Auswählen des aktuellen Listenelements.

while_waiting()

Ermöglicht das Ausführen von Checks die zu einem Refresh führen können, falls in diesem Moment auf dem TUI gerade keine Interaktion erfolgt.

4.2.2 Paket config

Das Paket liest die Einstellungen aus einer Konfigurationsdatei.

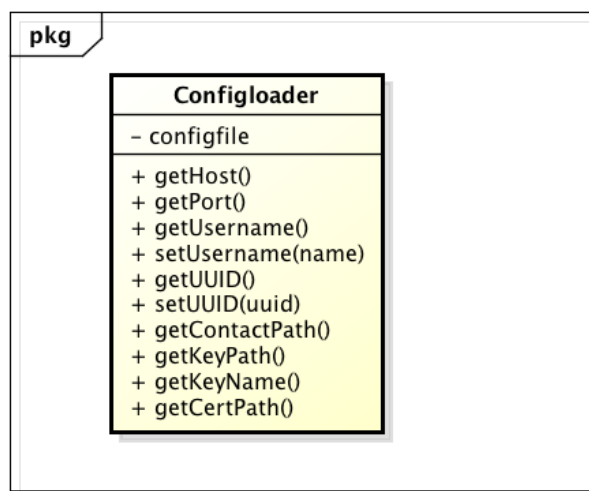
Die Konfigurationsdatei für den Client liegt im selben Verzeichnis wie pyg8, hat den Namen „config“ und folgenden Inhalt:

```
[general]
contact_path = data/contacts/
username = Username
my_uuid = {00000000-0000-0000-0000-000000000000}

[connection]
host = localhost
port = 10000

[crypto]
key_path = data/keys/
key_name = key1
cert_path = data/certs/
```

4.2.2.1 Klassenstruktur



4.2.2.2 Wichtige interne Abläufe

Der Konstruktor der Configloader-Klasse nimmt ein Konfigurationsfile entgegen. Für alle benötigten Werte aus dem Konfigurationsfile wird anschliessend ein Getter zur Verfügung gestellt.

4.2.3 Paket crypto

Das Paket Crypto enthält Funktionen zur Generierung von Schlüsseln und kümmert sich um die Ver- und Entschlüsselung von Daten. Crypto wurde nicht in einer Klasse umgesetzt, da keine Zustände notwendig sind.

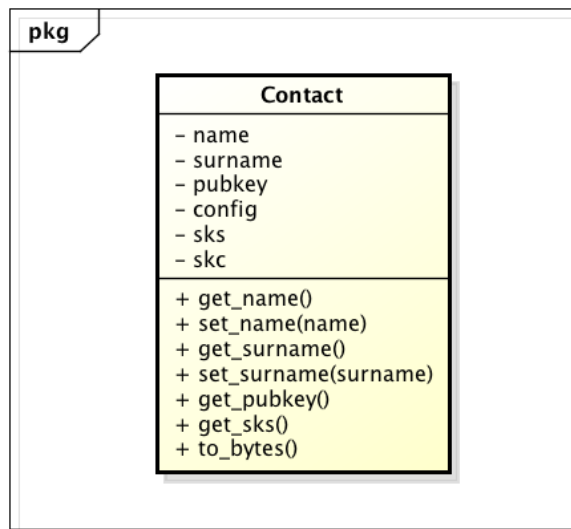
4.2.3.1 Funktionen

Funktion	Beschreibung
generate_ecdhKey()	Generiert ein ECDH-Key-Objekt.
get_sk(my_key_obj, other_pubkey)	Generiert den Session-Schlüssel aus dem eigenen ECDH-Key-Objekt und dem Public-Schlüssel des Partners.
_kdf3(sk)	Leitet einen symmetrischen Schlüssel aus einem anderen symmetrischen Schlüssel ab.
get_sks_skc(sk)	Nutzt kdf3() um Sessionkey.Signing und Sessionkey.Crypto zu generieren.
_get_cipher(symmetric_key, counter_init_val)	Gibt einen AES/CTR mit entsprechendem symmetrischen Schlüssel und dem Counter auf counter_init_val initialisiert zurück.
encrypt(bytes_in, symmetric_key)	Verschlüsselt bytes_in mit cipher aus symmetric_key.
decrypt(ciphertext, symmetric_key, counter_init_val)	Entschlüsselt ciphertext mit cipher der aus symmetric_key und counter_init_val generiert wird.
get_hmac(symmetric_key, msg)	Generiert HMAC/SHA256 über msg mit symmetric_key.
verify_hmac(symmetric_key, msg, received_hmac)	Verifiziert HMAC/SHA256 received_hmac über msg mit symmetric_key.
get_sh256(in_bytes)	Gibt Secure Hash SHA256 über in_bytes zurück.
write_privKey(ecdhKey, filename)	Schreibt ecdhKey.get_privkey() in filename.
write_pubKey(ecdhKey, filename)	Schreibt ecdhKey.get_pubkey() in filename.
load_privKey(filename)	Generiert ECDH-Key-Objekt aus „filename“ und dem zugehörigen publickey-file.
load_pubKey(filename)	Liest den Private Key aus filename.
get_unlockKey(user, password)	Generiert mittels PBKDF2/SHA256 aus user und password einen symmetrischen Schlüssel.

4.2.4 Paket contact

Der Contact hält alle nötigen Informationen über den Kontakt. Er beinhaltet unter anderem den Public-Key des Kontakts und die daraus generierten symmetrischen Sessionschlüssel skc und sks.

4.2.4.1 Klassenstruktur



4.2.4.2 Funktionen

Funktion	Beschreibung
@classmethod from_contact(cls, contact, unlockkey)	Verschlüsselten, serialisierten Kontakt einlesen
to_bytes(unlockkey)	Kontakt verschlüsseln und serialisieren
delete()	Kontakt von Festplatte löschen
get_uuid(), get_name(), get_surname(), get_pubkey(), get_sks(), get_skc()	Getter für Objektattribute
set_name(name), set_surname(surname), set_pubkey(pubkey)	Setter für Objektattribute
add_msg(msg)	Nachricht zu Kontakt hinzufügen und die entschlüsselte Message zurückgeben
get_msg(msg_hash)	Nachricht mit bestimmtem Hash zurückgeben
get_all_messages()	Alle Nachrichten des Kontakts zurückgegeben
_write_msg(msg)	Nachricht auf Festplatte schreiben
_read_msg(msg_hash)	Nachricht von Festplatte lesen
_encrypt_msg(dataMsg)	Nachricht verschlüsseln
_decrypt_msg(encDataMsg)	Nachricht entschlüsseln
write(unlockkey)	Kontakt auf Festplatte schreiben
@classmethod _read(cls, path, unlockkey)	Kontakt von Festplatte lesen
_get_my_path()	Pfad zu den Kontakten aus Konfiguration lesen

@classmethod _get_my_key(cls)	Liest den privaten Schlüssel von der Festplatte
@classmethod get_by_name(cls, name, surname, unlockkey)	Kontakt mit bestimmten Namen zurückgeben
@classmethod get_by_uuid(cls, uuid, unlockkey)	Kontakt mit bestimmter UUID zurückgeben
@classmethod _encrypt_contact(cls, contact, unlockkey)	Kontakt verschlüsseln
@classmethod _decrypt_contact(cls, encrypted_contact, unlockkey)	Kontakt entschlüsseln
@classmethod get_all_contacts(cls, unlockkey)	Alle Kontakte zurückgeben
@staticmethod _create_dirs(folder)	Kontakt-Verzeichnisse anlegen
@classmethod validate_unlockkey(cls, unlockkey)	Überprüft, ob der übergebene unlockkey gültig ist

4.2.4.3 Wichtige interne Abläufe

Neben der Verwaltung der Kontaktinformationen kümmert sich die Klasse Contact auch um die Verwaltung der Nachrichten. Bei unserer Architektur muss ein Kontakt entscheiden können, ob ihm eine verschlüsselte Nachricht gehört, indem er versucht eine Encrypted Message zu entschlüsseln.



4.2.5 Paket messaging

Das Paket messaging wurde nicht in einer Klasse umgesetzt, da keine Zustände notwendig sind.

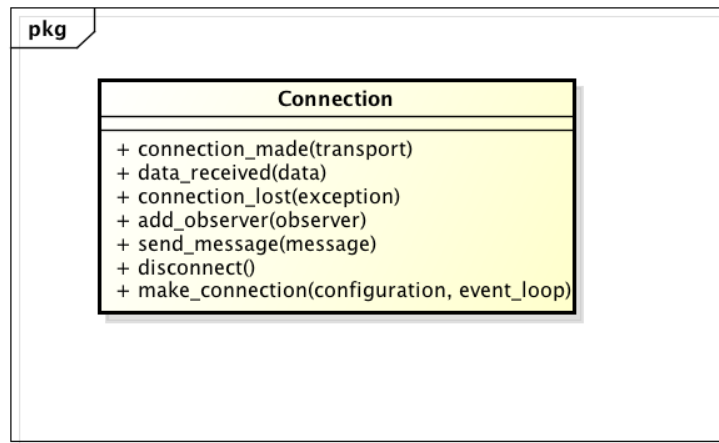
4.2.5.1 Funktionen

Funktion	Beschreibung
is_complete(potential_message)	Prüft, ob eine Nachricht komplett angekommen ist.
pack_data_msg(message_data, contact)	Generiert eine Protobuf-Data-Message und liefert den daraus generierten Bytestrom.
pack_request_msg(message_hash)	Generiert eine Protobuf-Request-Message und liefert den daraus generierten Bytestrom.
unpack_msg(message)	Liefert eine Protobuf-Notification-Message aus einem empfangenen Bytestrom.
data_msg(message)	Liefert eine Protobuf-Data-Message aus einem serialisierten Bytestrom.

4.2.6 Paket connection

Das Paket connection übernimmt sämtliche Aufgaben, die mit der Verbindung zum Server zu tun haben. Wir greifen dabei auf die asyncio-Bibliothek zurück, die den TCP-Verbindungsaufbau, die TLS-Verschlüsselung und den asynchronen Message-Austausch vereinfacht.

4.2.6.1 Klassenstruktur



4.2.6.2 Klassenbeschreibungen

Klasse	Beschreibung
Connection	Die Klasse kümmert sich um die TCP-Verbindung und die TLS-Verschlüsselung. connection_made(transport): Die Funktion ist ein Callback aus asyncio, der ein Objekt zurückliefert, das den Socket der Verbindung enthält. data_received(data): Die Funktion ist ein Callback aus asyncio, der aufgerufen wird, sobald Daten empfangen werden. Sobald ein komplettes Paket empfangen wurde informieren wir die Observer. connection_lost(exception): Die Funktion ist ein Callback aus asyncio, der aufgerufen wird, sobald die Verbindung unerwartet getrennt wurde. add_observer(observer): add_observer dient der Registrierung von Observern, die benachrichtigt werden, sobald neue Daten angekommen sind. @asyncio.coroutine send_message(message): send_message versendet eine Nachricht an den Server.



@asyncio.coroutine
disconnect():

Diese Funktion trennt die im Hintergrund laufende Verbindung zum Server.

@staticmethod
make_connection(configuration, event_loop):

make_connection ist eine statische Funktion, welche ein Connection-Objekt zurück gibt.

4.2.7 Paket protocol

Die Klassen für das Protokoll sind mit protobuf generiert und sind in Kapitel 4.1.6 abgebildet. Die Klassen dienen dazu, die verschiedenen Message-Typen abzubilden. Jedes Objekt muss mit den benötigten Attributen befüllt werden und kann anschliessend serialisiert werden.

4.2.8 Transitions

Die Transition beschreiben die Übergänge von einem Programmzustand in den Nächsten. Wir zeigen hier die vier Zustände von pyg8 und beschreiben was in einem Übergang zwischen den Zuständen passiert.

4.2.8.1 Init: Beendet -> Getrennt

Der Übergang Init beschreibt den Wechsel vom Zustand Beendet in den Zustand Getrennt.
Beteiligte Packages: UI, Crypto, Contact

Aktionen:

- Crypto starten
- Kontakte laden
- Kontakte entschlüsseln
- Kontakte auftauen
 - Key laden
 - Messages entschlüsseln

4.2.8.2 Connect: Getrennt -> Verbunden

Der Übergang Connect beschreibt den Wechsel vom Zustand Getrennt in den Zustand Verbunden.
Beteiligte Packages: Connection

Aktionen:

- TLS Context aufbauen
 - Eigenes Cert laden
 - Server Cert laden
- TLS Verbindung aufbauen

4.2.8.3 Update: Verbunden -> Bereit

Der Übergang Update beschreibt den Wechsel vom Zustand Verbunden in den Zustand Bereit.
Beteiligte Packages: Message, Connection, Contact, Crypto, UI

Aktionen:

- Blacklist holen
- Kontakt Sperrzustand aktualisieren
- Nachricht abholen
- Nachrichten entschlüsseln
- UI aktualisieren

4.2.8.4 Deinit: Bereit -> Beendet

Der Übergang Deinit beschreibt den Wechsel vom Zustand Bereit in den Zustand Beendet.
Beteiligte Packages: Connection, Contact, Crypto

Aktionen:

- Verbindung trennen
- Kontakte einfrieren
- Kontakte verschlüsseln
- Kontakte speichern
- Crypto herunterfahren

4.3 Wichtige Abläufe

4.3.1 Schlüsselaustausch

Der Schlüsselaustausch ist im „Crypto Design Dokument“ beschrieben.

4.3.2 Nachricht versenden/empfangen

Der Ablauf für das Versenden und Empfangen ist folgendermassen:

Auf dem Client1:

1. Textnachricht in Message verpacken
2. Message serialisieren
3. Serialisierte Message verschlüsseln
4. Verschlüsselte Zeichenkette in eine Encrypted Message verpacken
5. Senden der Encrypted Message an den Server

Auf dem Server:

1. Empfangen von Nachrichten
2. Ausliefern von Nachrichten auf Anfrage oder per Notifizierung

Auf dem Client2:

1. Nachricht anfragen
2. Notification empfangen und in einzelne Encrypted Messages unterteilen
3. Encrypted Messages auspacken und entschlüsseln

Details zum Protokoll finden sich im Dokument *Domainanalyse_frontg8* im Kapitel *Protokollspezifikation*.

4.3.2.1 Login

frontg8 gibt dem Benutzer maximal 3 Versuche zur erfolgreichen Anmeldung mittels Benutzername und Passwort.

5. Prozesse und Threads

5.1 Server

Momentan ist frontg8d eine Single-Threaded Applikation. Wir verwenden Asio für die Verarbeitung von Datentransfers über die eingesetzten TCP-Sockets. Für das Speichern und Laden von Nachrichten und des Journals setzen wir POSIX AIO ein. Performance Messungen haben jedoch ergeben, dass selbst bei der Angabe von 'O_NONBLOCK' der 'open()' Systemaufruf bis zu mehrere hundert Millisekunden blockiert. Deshalb benötigen wir für den Message Store wahrscheinlich einen eigenen Thread. Da der Testing- und Implementationsaufwand jedoch hoch ist, werden wir diesen nicht im Rahmen dieses Projekts einbauen können.

5.2 Client

Python bringt zwar in der Sprache gute Unterstützung für Threads mit, allerdings muss bei der Nutzung von Threads mit Einschränkungen gerechnet werden. Da Python eine interpretierte Sprache ist, läuft ein Script grundsätzlich in einem einzigen Thread. Das Ausführen von mehreren Threads wird also vorgetäuscht, denn in Wirklichkeit laufen mehrere Python Threads in einem einzigen Systemthread.

Falls im Client wirklich parallele Prozesse benötigt werden, so kann dies mit dem python multiprocessing package erreicht werden. Dessen Einsatz scheint aber zum jetzigen Zeitpunkt nicht nötig zu sein.

6. Deployment

Die Software ist in einen Server und in einen Client aufgeteilt. Beide Softwarekomponenten wurden auf der Linux-Distribution „Ubuntu 14.04“ getestet. Nachfolgend sind die Schritte beschrieben um die Komponenten einsetzen zu können. Mehr Informationen zur verwendeten Toolchain sind im Projektwiki (https://redmine.frontg8.ch/redmine/projects/frontg8/wiki/C++_Toolchain) zu finden.

6.1 Server

6.1.1 Toolchain zum Kompilieren des Sourcecodes

Folgende Software wird zum Kompilieren benötigt:

- llvm
- clang
- protobuf
- Asio
- Cute

6.2 Client

Der Client setzt python3 voraus. Da der Python-Code interpretiert wird, ist der Code aus dem Software-Repository nach Installation der benötigten Abhängigkeiten sofort einsatzfähig.

6.2.1 Installation abhängiger Pakete

Folgende Softwarekomponenten können mit Bordmitteln von Ubuntu nachinstalliert werden:

```
apt install libprotobuf-dev protobuf-compiler python-protobuf
pip3 install protobuf3
pip3 install xmlrunner
pip3 install pycrypto
pip3 install pyelliptic
pip3 install pyOpenSSL
```

6.2.2 Patching protobuf für python

In der Python-Version von protobuf existiert ein Fehler, der uns direkt betrifft. Folgende Datei muss angepasst werden, damit pyg8 funktioniert:

/usr/local/lib/python3.4/dist-packages/protobuf3/message.py

Vor Zeile 88 einfügen:

```
if number == 0:
    result.append(number)
```

7. Datenspeicherung

7.1 Server

Die Speicherung der Nachrichten und des Journals wird via "Flatfiles" realisiert. Das Backend für diese Speicherung muss vom Serverbetreiber bereitgestellt werden. Wir empfehlen hierzu den Einsatz des mitgelieferten Skripts. Dieses Skript erstellt eine RAM-Disk. Die Speicherung der Daten in einer RAM-Disk stellt sicher, dass die Daten zerstört werden, sollte das Gerät auf dem der Server läuft vom Strom getrennt werden.

Die Daten werden in regelmässigen Abständen (Standardeinstellung: 5 Minuten) auf dem oben erwähnten Storage-Backend gespeichert und nach einer konfigurierbaren Zeit (Standardeinstellung: 72 Stunden) wieder entfernt.

7.2 Client

Wie auf dem Server werden auch auf dem Client die verschlüsselten Nachrichten in „Flatfiles“ abgelegt. Weiter werden auch die Kontakte in verschlüsselten Dateien abgelegt. Details zur Verschlüsselung finden sich wiederum im „Crypto Design Dokument“.



8. Grössen und Leistung

Als Performanceziel nehmen wir uns 2000 Nachrichten pro Tag vor.

Sobald sich der Server und Client nicht mehr im Alpha Stadium befinden werden wir Performance-Messungen vornehmen um die Leistungsfähigkeit zu überprüfen.



9. Anhang

9.1 Links

- Protobuf - <https://github.com/google/protobuf>
- Asio - <https://think-async.com>
- OpenSSL - <https://www.openssl.org>
- LLVM - <http://llvm.org>
- CUTE - <http://cute-test.com>
- PyCrypto - <https://pypi.python.org/pypi/pycrypto>
- PyElliptic - <https://pypi.python.org/pypi/pyelliptic>
- PyOpenSSL - <https://pypi.python.org/pypi/pyOpenSSL>