A d - h o c C h a t s y s t e m f ü r m o b i l e N e t z e

G r u p p e 3 (A d B e e)

Softwareentwicklungspraktikum
Sommersemester 2007

F e i n e n t w u r f

Auftraggeber:
Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund
Prof. Dr.-Ing. Lars Wolf
Mühlenpfordtstraße 23, 1. OG
38106 Braunschweig
Deutschland

Betreuer: Sven Lahde, Oliver Wellnitz
Hiwi: Wolf-Bastian Pöttner

Auftragnehmer: Gruppe 3(AdBee)

Name	E - Mail
Ekrem Özmen	e.oezmen@gmail.com
Celal Özyalcin	c.oezyalcin@tu-bs.de
Thorben Schulze	thorben.schulze@tu-bs.de
Danny Melching	danny.melching@tu-bs.de

Phasenverantwortlicher: Celal Özyalcin

Braunschweig, 14.05.2007

Versionsübersicht

Version	Datum	Autor	Status	Kommentar
1.0	14.05.07	Ekrem, Celal, Danny, Thorben		Erste Version des Feinentwurfs

I n h a l t s v e r z e i c h n i s

A b b i l d u n g s v e r z e i c h n i s

1 Einleitung

Das Dokument gliedert sich in Erfüllung der Kriterien, Implementierungsentwurf und Datenmodell. Im Punkt Erfüllung der Kriterien, sind die, im Pflichtenheft, ausgearbeiteten Muss-, Wunsch- und Abgrenzungskriterien aufgelistet und beschrieben. Im Punkt Implementierungsentwurf findet sich das Komponentendiagramm aus dem Grobentwurf wieder. Hier werden die einzelnen Komponenten im Detail beschrieben, mithilfe eines Klassendiagramms und Auflistung und Beschreibung der Klassen und Methoden der einzelnen Komponenten. Im Punkt Datenmodell sind kurz die langfristig zu speichernden Daten beschrieben.

Ziel dieses Dokumentes ist, die Implementierung des Projekt AdBee hinreichend genau zu beschreiben, so dass es jedem Softwareentwickler möglich ist, das Produkt zu entwickeln.

2 Erfüllung der Kriterien

2.1 Musskriterien

Die folgenden Kriterien sind unabdingbar und müssen durch das Produkt erfüllt werden:

/M10/ Benutzer können Textnachrichten sowie Binärdaten miteinander austauschen

/M20/ Der Austausch von Nachrichten erfolgt in Kommunikationskanälen

/M30/ Kommunikationskanal befinden sich Benutzer, wobei jede gesendete Nachricht von allen Benutzern im jeweiligen Kanal empfangen wird

/M40/ Jeder Kommunikationskanal besitzt einen Namen und eine eindeutige Identifizierung

/M50/ Die Kanäle sind in ihrer Anzahl nicht begrenzt

/M60/ Man kann sich eine Liste aller existierenden Kanäle anzeigen lassen

/M70/ Die Kanäle öffentlich oder geschlossen

/M80/ Jeder kann den öffentlichen Kanälen beitreten

/M90/ In den geschlossenen Kanälen muss man von einem Benutzer, der sich schon in diesem Kanal befindet, eingeladen werden

/M100/ Es muss den Benutzern des Chatsystems möglich sein, einen Kommunikationskanal zu erstellen, in diese beitreten, sie wieder verlassen und jemand anderen in einen geschlossenen Kanal einladen zu können

/M110/ Jeder Benutzer befindet sich zum Start des Programms in einem anonymen Kanal

/M120/ Der anonyme Kanal besitzt den eindeutigen Namen „Anonymous“, welcher von den anderen Kanälen nicht verwendet werden darf

/M130/ Die Übertragung in dem anonymen und den geschlossenen Kanälen erfolgt verschlüsselt

/M140/ Jeder Benutzer besitzt ein RSA-Schlüsselpaar

/M150/ Jeder geschlossene Kanal besitzt einen gemeinsamen Schlüssel

/M160/ Das RSA-Schlüsselpaar ermöglicht den sicheren Austausch des gemeinsamen Schlüssels

/M170/ Das RSA-Schlüsselpaar ermöglich die Verschleierung der Herkunft der Nachricht in den anonymen Kanälen

/M180/ Die Benutzer des Chatsystems besitzen einen Namen und müssen eindeutig identifiziert werden können

/M190/ Jeder Benutzer kann sich in mehren Kommunikationskanälen befinden

/M200/ Das RSA-Schlüsselpaar besteht aus dem öffentlichen Zertifikat, welches vor Benutzung des Programms vergeben wird, und einem privaten Schlüssel, der zum entschlüsseln und zum signieren von Nachrichten vorgesehen ist.

/M210/ Die Nachrichten besitzen eine eindeutige Identifizierung

/M220/ Die Nachrichten müssen, außer in dem anonymen Kanal, einem Sender durch Authentifizierung eindeutig zugeordnet werden können

/M230/ Es muss möglich sein, das Zertifikat des RSA-Schlüsselpaares anzufordern und zu versenden

/M240/ Ein Nachrichtenaustausch des gemeinsamen Schlüssels für die geschlossenen Kanäle muss möglich sein

/M250/ Sobald eine Nachricht verschickt wird, findet sie automatisch ihren Weg durch das Netz

/M260/ Jeder Knoten sendet eine Empfangsbestätigung an den vorherigen Knoten

/M270/ Jede Nachricht besitzt eine TTL, die angibt, wie lange sie noch im Netz verbleiben darf, bevor sie aus dem Netz entfernt wird und dem Sender eine negative Empfangsbestätigung mit dem Inhalt der Nachricht gesendet wird

/M280/ Eine nicht angekommene Nachricht wird gespeichert und später erneut gesendet

/M290/ Falls eine Nachricht endgültig die Empfänger nicht erreicht, wird dies dem Sender mitgeteilt

/M300/ Da diese Nachrichten verzögert gesendet werden und die Übertragung verbindunglos mittels UDP-Datagrammen stattfindet, muss die richtige Reihenfolge am Empfänger gewährleistet werden

/M310/ Welche Kriterien beim versenden von Nachrichten genau beachtet werden müssen, findet man in der Protokollspezifikation[1].

/M320/ Das Chatprogramm muss interoperabel mit den Chatprogrammen der anderen Projektgruppen sein

/M330/ Aufgrund der Mobilität der Teilnehmer soll eine ständige Aktualisierung der Netzstruktur, durch periodenweises Schicken von Hello-Nachrichten und Routing-Nachrichten, erfolgen

/M340/ Es sollen nicht nur einzelne Benutzer mit in eine Netzstruktur aufgenommen werden können, sondern auch eine Verschmelzung von zwei oder mehr Netzstrukturen möglich sein.

/M350/ Öffentliche Kanäle mit gleichen Namen werden zu einem Gemeinsamen zusammengeschlossen

/M360/ Geschlossene Kanäle müssen weiterhin getrennt behandelt werden

/M370/ Zum Testen der Software gibt es eine Peerverwaltung die ein- bzw. ausgeschaltet werden kann(Infrastrukturmodus)

/M380/ Das Chatprogramm muss auf den Betriebssystemen Mac OS X und Linux lauffähig sein

/M390/ Die graphische Oberfläche sollte bedienerfreundlich gestaltet sein

2.2 Wunschkriterien

Die Erfüllung folgender Kriterien für das abzugebende Produkt wird angestrebt:

/W10/ Jeder Benutzer kann eine Liste aller gerade im Netz vorhandenen Benutzer anzeigen lassen

/W20/ Neue Funktionen, die die Handhabung erleichtern oder erweitern, sollten sich möglichst einfach einbinden lassen

/W30/ Eine Portierung auf andere Betriebssysteme sollte auch ohne größere Änderungen erfolgen können

/W40/ Die Benutzeroberfläche sollte vom jeweiligen Benutzer in sinnvollem maß veränderbar sein

2.3 Abgrenzungskriterien

Folgende Funktionalitäten werden nicht durch das Produkt, sondern wie folgt beschrieben anderweitig erfüllt:

/A10/ Das Programm wird nicht mit schon vorhandenen Chatsystemen Interoperabel sein.

/A20/ Die Größe der übertragbaren Dateien ist begrenzt.

3 Implementierungsentwurf

3.1 Gesamtsystem

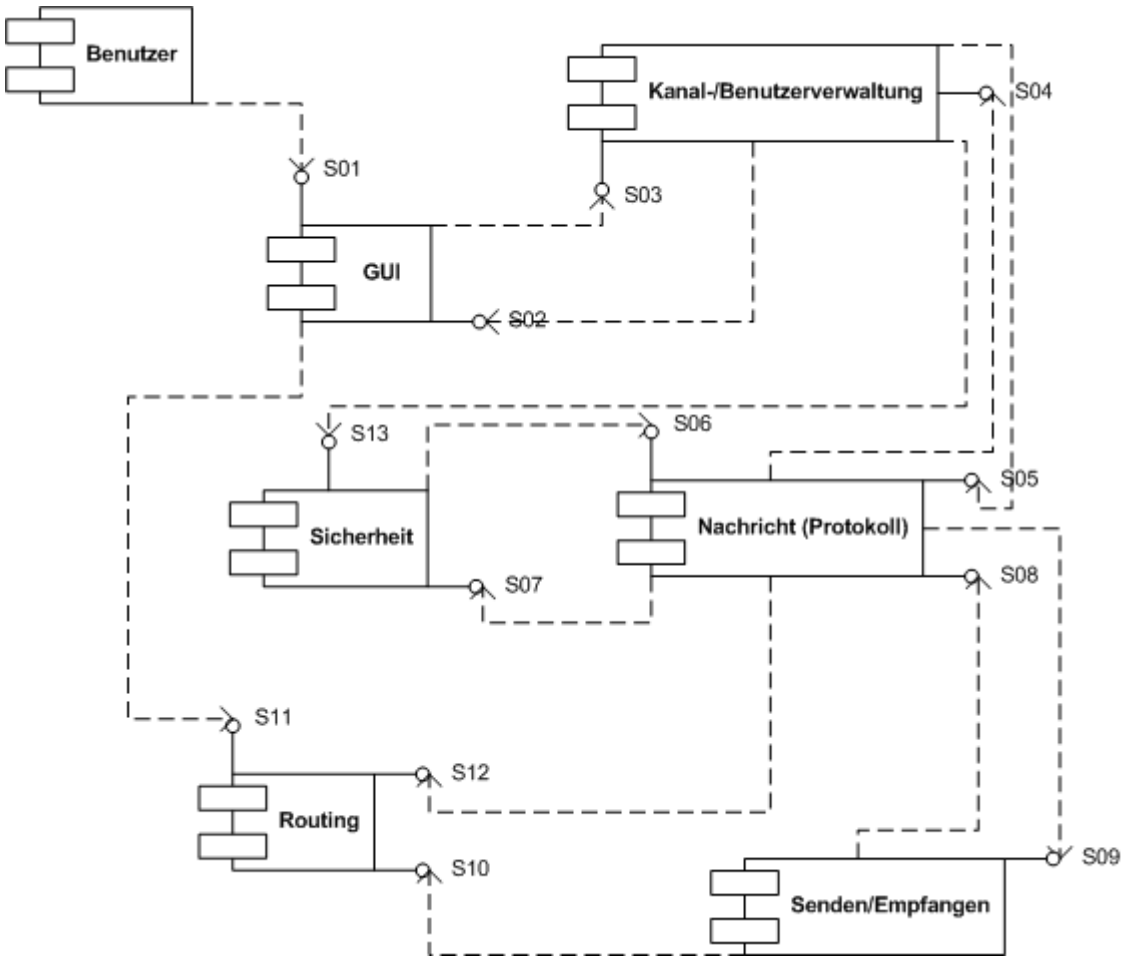

Abbildung 1: Komponentendiagramm

Das Diagramm zeigt die Komponenten die später im Programm miteinander kommunizieren. Die Benutzerkomponente steht für den menschlichen Teilnehmer, der mit dem Programm interagiert. Dieser verwendet das Programm nur über die GUI.

Die GUI leitet das weiter, was durch den Benutzer gefordert wird. Sie verwaltet die visuellen Objekte wie Kanalfenster, Eingabefeld, Teilnehmerfeld usw.

Die Kanal-/Benutzerverwaltung verwaltet die gesamten Informationen der Benutzer und Kanäle des Programms. Es interpretiert die Informationen, die von der Nachrichtenkomponente kommen und gibt sie in geeigneter Weise an die GUI weiter. Genauso nimmt sie Informationen von der GUI entgegen und gibt sie an die Nachrichtenkomponente weiter.

Die Nachrichtenkomponente kümmert sich um die Verwirklichung des Protokolls. Sie realisiert das Erstellen und parsen von XML-Dokumenten. Dementsprechend kann es auf Systemnachrichten reagieren, Nachrichten zum Senden erstellen und empfangene Nachrichten parsen. Ebenfalls benutzt es die Sicherheitskomponente zum verschlüsseln,

entschlüsseln, signieren und authentifizieren von Nachrichten. Ebenfalls übergibt es Informationen an die Routingkomponente bei eintreffenden Routingnachrichten, damit diese ihre Tabellen aktualisieren kann.
Die Sicherheitskomponente kümmert sich um die Sicherheit der Nachrichten und nutzt den Dienst der Nachrichtenkomponente, um Schlüssel und Zertifikate anzufordern.
Die Senden/Empfangen-Komponente ist die Schnittstelle zum Netz. Über sie kommen die UDP-Datagramme an und über sie werden die Nachrichten mittels UDP versenden. Es nutzt den Dienst der Routingkomponente, um an die richtigen Knoten zu senden.
Die Routingkomponente verwaltet die Routingtabellen und den Infrastrukturmodus. Die Nachrichten und Senden/Empfangen-Komponente wissen nicht, in welchem Modus sich die Komponente befindet, sondern nutzen nur deren Dienst, damit das Testen verwirklicht werden kann. Über die GUI kann der Infrastrukturmodus ein- und ausgeschaltet werden.
Allgemein verwenden die Komponenten auch Klassen der anderen Komponenten, die nicht in den Schnittstellen aufgelistet werden. Dazu gehören Hauptsächlich die User, die Message und die Channel Objekte, weil diese öfter benutzt werden, um Informationen abzurufen und weiterzugeben.

3.2 Implementierung von Komponente K01: GUI

Die GUI-Komponente besteht aus einem Hauptfenster(MainWindow), welches auch nach Start angezeigt wird. Andere Funktionen des Programmes, welche das Hinzufügen eines Anhangs, dem Erstellen eines Kanals, dem Einstellen der Fenster und der Behandlung des Infrastrukturmodus sind, werden in speziellen Dialogfenstern ausgeführt. Dabei werden alle Ereignisse, die durch die Fenster ausgelöst werden, durch die Eventhandling-Klasse behandelt und je nach Event bestimmte Funktionen ausgeführt.

Es werden Swing-Komponenten von Java für die GUI-Objekte benutzt.

3.2.1 Klassendiagramm

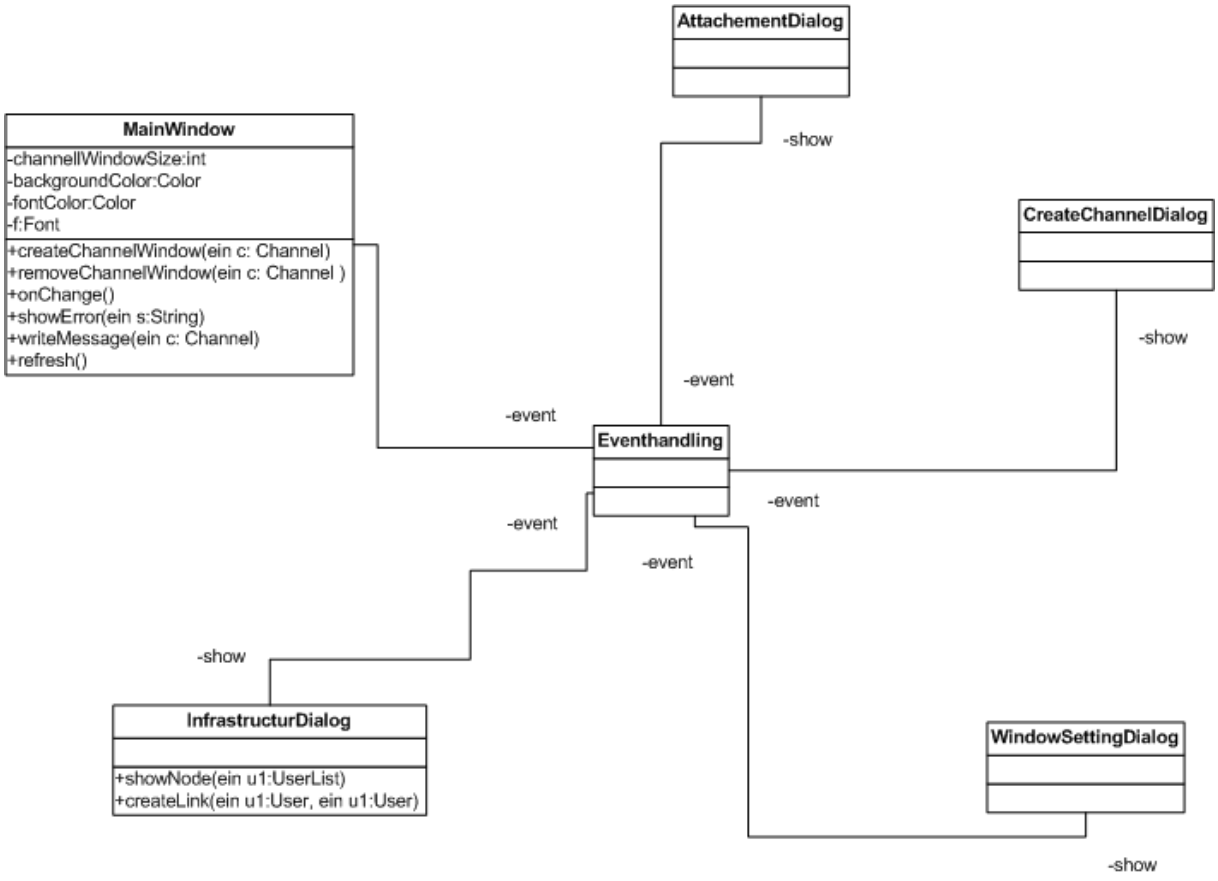

Abbildung 2: Klassendiagramm: GUI

3.2.2 Erläuterung

<table>
<tr><th>Klasse</th><th>Aufgabe</th><th>Attribute und Kommunikationspartner</th></tr>
<tr><td rowspan="8">MainWindow</td><td>Das Hauptfenster mit den Kanälen und Anzeigen für Kanäle und Benutzer.</td><td rowspan="8"><u>Attribute:</u>
ChannelWindowSize : int
backgroundColor: Color
fontColor: Color
font: Font

<u>Kommunikationspartner:</u>
Eventhandling: Events, die der Benutzer mit der GUI auslöst, werden durch die Eventhandling-Klasse behandelt.</td></tr>
<tr><td>ID: MW01
Name: createChannel(c:Channel)
Beschreibung:
Die Funktion erstellt ein neues Fenster und einen zugehörigen Reiter.</td></tr>
<tr><td>ID: MW02
Name: removeChannelWindow(c:Channel)
Beschreibung:
Die Funktion entfernt einen Kanal mit dem zugehörigen Reiter.</td></tr>
<tr><td>ID: MW03
Name: onChange()
Beschriebung:
Die Funktion aktualisiert alle angezeigten Listen. Dazu gehören: Kanalliste, Benutzerliste, Kanalteilnehmerliste.</td></tr>
<tr><td>ID: MW04
Name: +showError(s: String)
Beschreibung:
Öffnez einen Dialog mit einer Fehlermeldung.</td></tr>
<tr><td>ID: MW05
Name: writeMessage(text:String, c:Channel)
Beschreibung:
Fügt eine angekommene Nachricht in das Kanalfenster hinzu.</td></tr>
<tr><td>ID: MW06
Name: refresh()
Beschreibung:
Aktualisiert geänderte Kanalgrößen- und Farbeinstellungen.</td></tr>
<tr><td>Attachement-Dialog</td><td>Ein Dialog der angezeigt wird, wenn ein Anhang angefügt werden soll oder ein Anhang erhalten wird.</td><td><u>Attribute:</u>
Keine

<u>Kommunikationspartner:</u>
Eventhandling: Events, die der Benutzer mit der GUI auslöst, werden durch die Eventhandling-Klasse behandelt</td></tr>
</table>

Create-Channel-Dialog	Ein Dialog zum Erstellen von Kanälen	**Attribute:** Keine **Kommunikationspartner:** **Eventhandling:** Events, die der Benutzer mit der GUI auslöst, werden durch die Eventhandling-Klasse behandelt
Window-SettingDialog	Ein Dialog zum Verändern der Fenstereinstellungen	**Attribute:** Keine **Kommunikationspartner:** **Eventhandling:** Events, die der Benutzer mit der GUI auslöst, werden durch die Eventhandling-Klasse behandelt
Infrastrucutur-Dialog	Ein Dialog der zum Anschalten des Infrastrukturmodus und zum Verändern der virtuellen Netzwerkstruktur **ID:** ISD01 **Name:** showNodes(ul:UserList) **Beschreibung:** Zeigt auf dem Fenster die verfügbaren Knoten an. **ID:** ISD01 **Name:** createLink(u1:User, u2:User) **Beschreibung:** Zieht einen Link zwischen zwei Knoten.	**Attribute:** Keine **Kommunikationspartner:** **Eventhandling:** Events, die der Benutzer mit der GUI auslöst, werden durch die Eventhandling-Klasse behandelt.
Eventhandling	Alle Ereignisse, die der Benutzer durch Knopfdrücke oder Kommandos tätigt, werden hier behandelt und es wird dementsprechend reagiert. Methoden der Klasse sind Ereignismethoden der GUI-Elemente.	**Attribute:** Keine **Kommunikationspartner:** Erhält Events von den Fensterklassen der GUI-Komponente.

3.3 Implementierung von Komponente K02: Verwaltung

Die Verwaltung ist für die Verwaltung von Benutzern und Kanälen zuständig. Das UserManagement kümmert sich um die Benutzer und das ChannelManagement um die Kanäle. Sie bestehen jeweils aus Userobjekten bzw. Channelobjekten. Ebenfalls besitzt die Verwaltung eine Klasse zum erzeugen von IDs. User- und ChannelManagement werden als Singleton implementiert, da nur eine Verwaltung notwendig ist. Die Channel-Klasse wird in ClosedChannel, OpenChannel und AnonymousChannel spezialisiert, um diese zu trennen.

3.3.1 Klassendiagramm

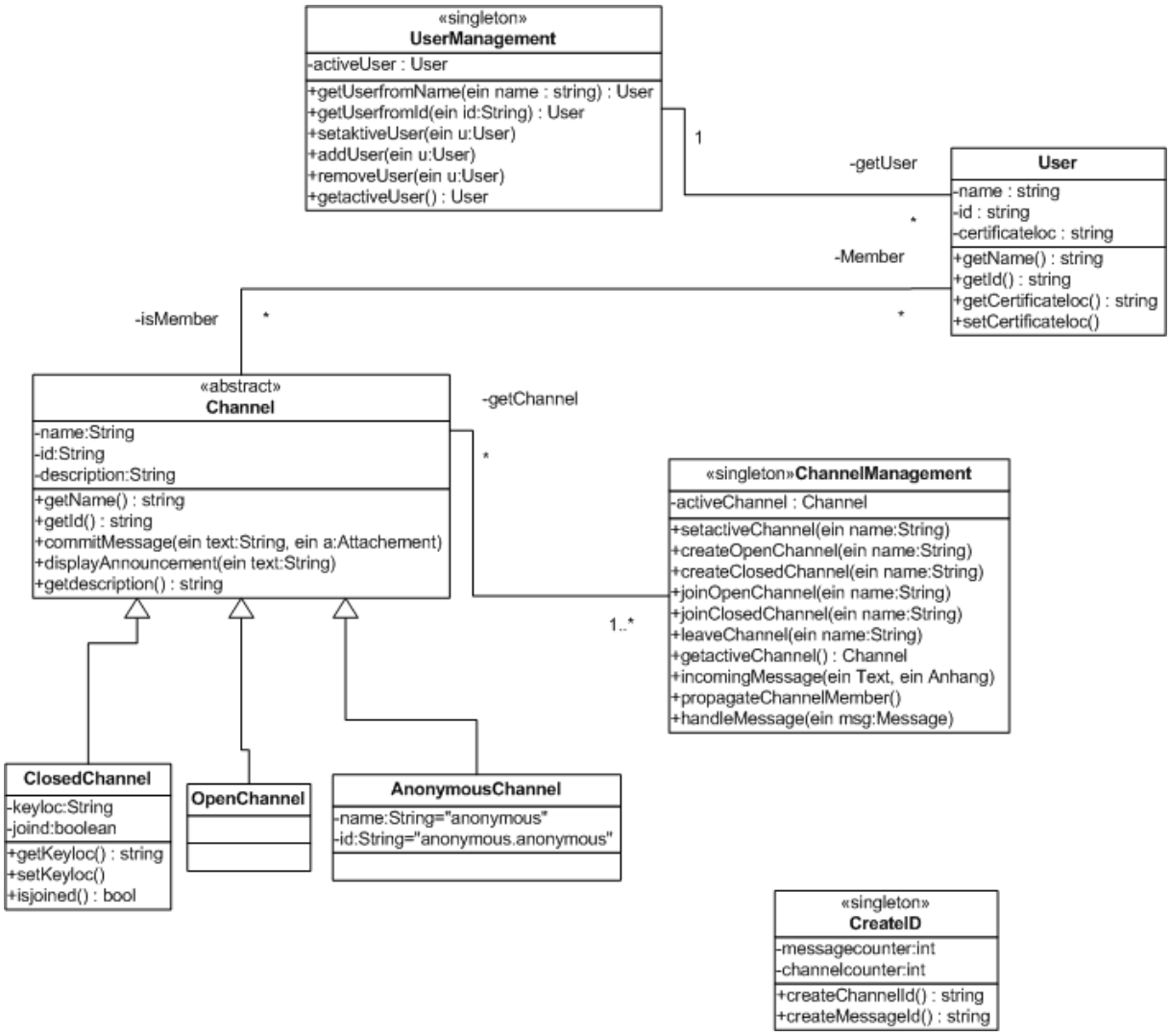

Abbildung 3: Klassendiagramm: Verwaltung

3.3.2 Erläuterung

<table>
<tr><th>Klasse</th><th>Aufgabe</th><th>Attribute und Kommunikationspartner</th></tr>
<tr><td rowspan="7"><<singleton>>
User-
Management</td><td>Das Management verwaltet die User, die sich alle im Netz befinden und den User, der das Programm benutzt.</td><td rowspan="7"><u>Attribute:</u>
aktiveUser : User

<u>Kommunikationspartner:</u>
User: Das UserManagement verwaltet die User die existieren.
Allgemein ist die Klasse Kommunikator zum Erhalt von UserInformationen</td></tr>
<tr><td>ID: UM01
Name: getUserfromName(name:String) : User
Beschreibung: Gibt einem das Userobjekt anhand eines Namens</td></tr>
<tr><td>ID: UM02
Name: getUserfromId(id:String): User
Beschreibung: Gibt einem ein Userobjekt anhand einer ID</td></tr>
<tr><td>ID: UM03
Name: setactiveUser(u:User)
Beschreibung: Setzt den aktiven User</td></tr>
<tr><td>ID: UM04
Name: addUser(u:User)
Beschreibung: Fügt einen User in die Liste aller User hinzu</td></tr>
<tr><td>ID: UM05
Name: removeUser(u:User)
Beschreibung: entfernt einen User aus der Liste</td></tr>
<tr><td>ID: UM06
Name: getactiveUser() : User
Beschreibung: Gibt den User zurück der das Programm benutzt.</td></tr>
<tr><td rowspan="3">User</td><td>Klasse zur Beschreibung eines Benutzers</td><td rowspan="3"><u>Attribute:</u>
name:String
id:String
certificateloc:String

<u>Kommunikationspartner:</u>
Userobjekte werden nur von anderen Klassen genutzt</td></tr>
<tr><td>ID: User01
Name: getName() : String
Beschreibung: Gibt den Namen zurück</td></tr>
<tr><td>ID: User02
Name: gedId() : String
Beschreibung: Gibt die Id zurück</td></tr>
</table>

	ID: User03 **Name:** getCertificatelco() : String **Beschreibung:** Gibt den Ort zurück wo das Zertifikat gespeichert ist.	
	ID: User04 **Name:** setCertificateloc() :String **Beschreibung:** Setzt den Ort wo das Zertifikat gespeichert ist	
<<singleton>> Channel- Management	Die Klasse verwaltet die Kanäle.	**Attribute:** activeChannel: Channel **Kommunikationspartner:** **Channel:** Das ChannelManagement verwaltet die Channel die existieren
	ID: CM01 **Name:** setactiveChannel(name:String) **Beschreibung:** Setzt den Kanal der gerade in der GUI aktiv ist.	
	ID: CM02 **Name:** createOpenChannel(name:String) **Beschreibung:** Erstellt einen offenen Kanal anhand eines Namens	
	ID: CM03 **Name:** createClosedChannel(name:String) **Beschreibung:** Erstellt einen geschlossenen Kanal anhand eines Namens.	
	ID: CM04 **Name:** joinOpenChannel(name:String) **Beschreibung:** Der aktive Benutzer tritt einen offenen Kanal anhand eines Namens bei.	
	ID: CM05 **Name:** joinClosedChannel(name:String) **Beschreibung:** Der aktive Benutzer tritt einen geschlossenen Kanal anhand eines Namens bei.	
	ID: CM06 **Name:** leaveChannel(name:String) **Beschreibung:** Der aktive Benutzer verlässt einen Kanal.	
	ID: CM07 **Name:** getactiveChannel() : Channel **Beschreibung:** Gibt den Kanal zurück der gerade in der GUI aktiv ist.	

	ID: CM08 **Name:** incomingMessage(text:String, a:Attachement) **Beschreibung:** Wird ausgeführt wenn eine Nachricht im aktiven Kanal gesendet worden ist.	
	ID: CM09 **Name:** propagateChannelMember(c:Channel) **Beschreibung:** Wird ausgeführt, wenn eine ChannelNachricht, unabhängig von den periodischen, verschickt werden soll.	
	ID: CM10 **Name:** handleMessage(msg:Message) **Beschreibung:** Behandelt ankommende Nachrichten die für Kanäle bestimmt sind.	
<<abstract>> Channel	Klasse die einen Kanal realisiert.	**Attribute:** name:String id:String description:String **Kommunikationspartner:** Channelobjekte werden nur von anderen Klassen benutzt
	ID: Channel01 **Name:** getName() : String **Beschreibung:** Gibt den Namen des Kanals zurück.	
	ID: Channel02 **Name:** getId() : String **Beschreibung:** Gibt die ID des Kanals zurück.	
	ID: Channel03 **Name:** commitMessage(text:String, a:Attachement) **Beschreibung:** Übergibt an die GUI eine Nachricht für den Kanal.	
	ID: Channel04 **Name:** displayAnnouncment(text:String) **Beschreibung:** Zeigt eine Systemmeldung in dem Kanal.	
	ID: Cannel05 **Name:** getDescription() : String **Beschreibung:** Gibt einem die Beschreibung des Kanals.	
Closed-Channel	Realisierung des geschlossenen Kanals	**Attribute:** keyloc:String joined:boolean **Kommunikationspartner:** Channelobjekte werden nur von anderen Klassen benutzt
	ID: CChannel01 **Name:** getKeyloc() : String **Beschreibung:** Gibt einem den Ort, wo der Schlüssel gespeichert ist zurück.	

	ID: CChannel02 **Name:** setKeyloc(loc:String) **Beschreibung:** Setzt den Ort, wo der Schlüssel gespeichert ist. **ID:** CChannel03 **Name:** isjoined() : boolean **Beschreibung:** Prüft, ob der Benutzer im Programm dem Channel beigetreten ist, weil im geschlossenen Kanal die Liste der Benutzer des Kanals nur darüber Auskunft gibt wer beitreten darf.	
OpenChannel	Benötigt keine zusätzlichen Funktionen	**Attribute:** Keine zusätzlichen **Kommunikationspartner:** Channelobjekte werden nur von anderen Klassen benutzt
Anonymous-Channel	Benötigt keine zusätzlichen Funktionen	**Attribute:** Name:String=“anonymous“ Id:String=<festgelegt> **Kommunikationspartner:** Channelobjekte werden nur von anderen Klassen benutzt
<<singleton>> CreateID	Die Klasse ist zum erstellen von eindeutigen IDs zuständig. **ID:** CID01 **Name:** createChannelId() : String **Beschreibung:** Übergibt eine eindeutige Kanal ID. **ID:** CID02 **Name:** createMessageId() : String **Beschreibung:** Übergibt eine eindeutige Nachrichten ID	**Attribute:** messageSequenz:int channelSequenz:int [Die Variablen dienen zum Erstellen einer eindeutigen ID] **Kommunikationspartner:** Message- und Channelobjekte lassen sich von diesem Objekt ID`s geben, damit diese eindeutig sind.

3.4 Implementierung von Komponente K03: Nachricht

Die Nachrichtenkomponente ist für die Verwaltung der Nachrichten zuständig. RecieveManagement und SendManagement sind die Klassen, die empfangene bzw. zu sendende Nachrichten speichern, damit sie vom MessageWorker verarbeiten und gesendet werden können. Die PeriodicMessageSender-Klasse ist dafür zuständig periodische Nachrichten zu versenden und bei gespeicherten Nachrichten, die durch ein Nack empfangen worden sind, zu prüfen, ob sie verschickt werden können. Die Message-Klasse wird in die einzelnen Unterklassen, je nach Messagetyp, spezialisiert. Dabei wird bei jeder Spezialisierung die getXML()-Funktion überlagert, um die richtige XML-Datei, wie sie im Protokoll definiert ist, zu erzeugen. Um überall das Parsen einer XML-Datei zu ermöglichen, besitzt die Messageklasse eine statische Pars-Funktion. Letztendlich sind die Managementklassen als Singleton zu implementieren.

3.4.1 Klassendiagramm

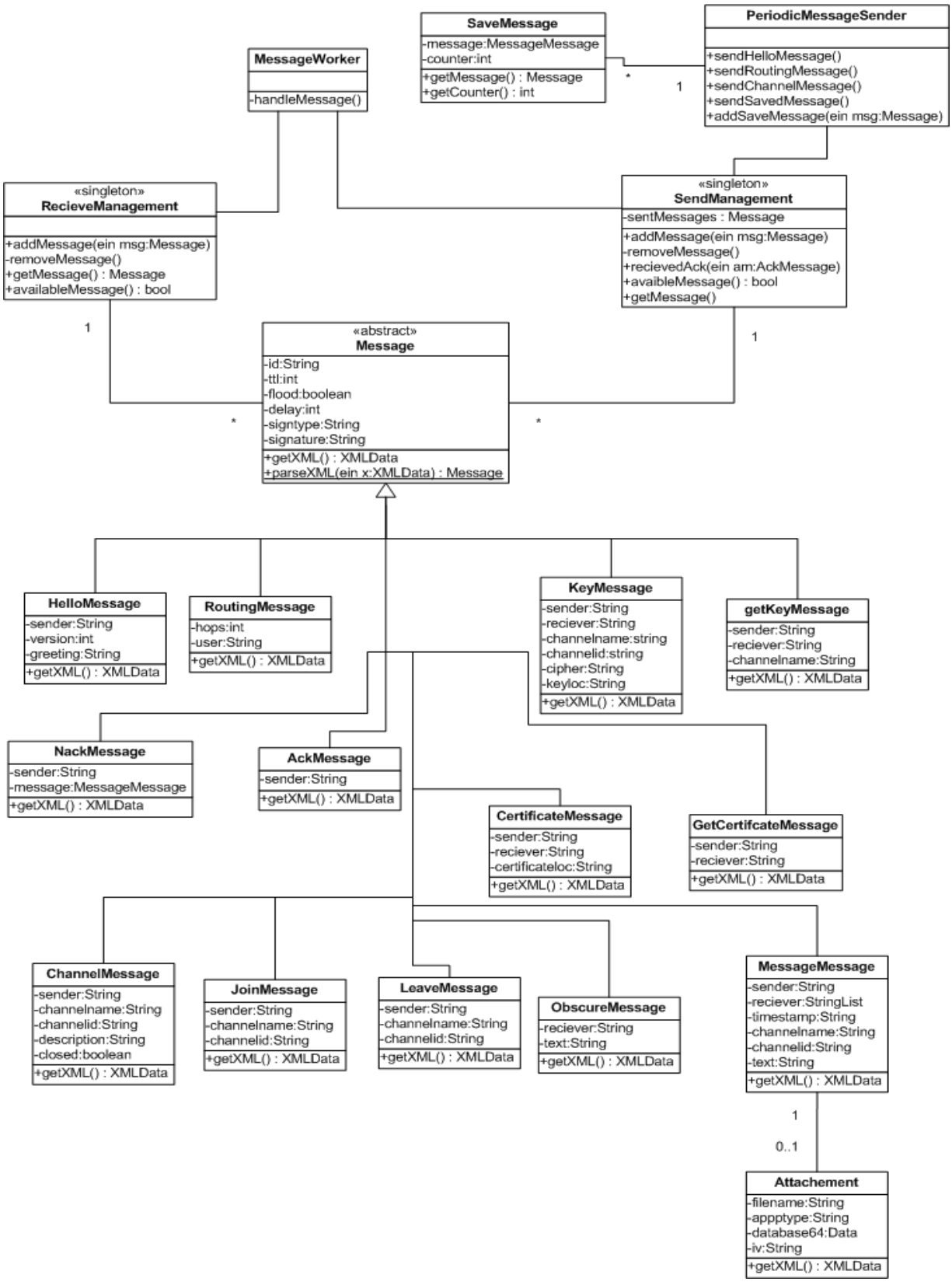

Abbildung 4: Klassendiagramm: Nachricht

3.4.2 Erläuterung

Klasse	Aufgabe	Attribute und Kommunikationspartner
<<singleton>> Recieve-Management	Puffert die Empfangenen Nachrichten	**Attribute:** Keine **Kommunikationspartner:** **Message:** Das Management verwaltet die Messages, die empfangen worden sind. **Allgemein** erhält das RecieveManagement-objekt nur Nachrichten, von der Senden/Empfangen-Komponente.
	ID: ReMa01 **Name:** addMessage(msg:Message) **Beschreibung:** Fügt eine Nachricht dem Puffer hinzu.	
	ID: ReMa02 **Name:** removeMessage() **Beschreibung:** Entfernt eine Nachricht aus dem Puffer.	
	ID: ReMa03 **Name:** getMessage() : Message **Beschreibung:** Holt eine Nachricht aus dem Puffer.	
	ID: ReMa04 **Name:** avaibleMessage() : boolean **Beschreibung:** Prüft, ob Nachrichten im Puffer sind.	
<<singleton>> Send-Management	Puffert zu sendende Nachrichten	**Attribute:** sentMessages:Message **Kommunikationspartner:** **Message:** Das Management verwaltet die Messages, die zum Senden bereit sind. **Allgemein** erhält das SendManagementobjekt nur Nachrichten von anderen Klassen die Nachrichten losschicken möchte.
	ID: SeMa01 **Name:** addMessage(msg:Message) **Beschreibung:** Fügt eine Nachricht dem Sendepuffer hinzu.	
	ID: SeMa02 **Name:** removeMessage() **Beschreibung:** Entfernt eine Nachricht aus dem Sendepuffer.	
	ID: SeMa03 **Name:** receivedAck(amsg:AckMessage) **Beschreibung:** Die Nachricht, zu der ein Ack erhalten wird, kann entfernt werden.	
	ID: SeMa04 **Name:** avaibleMessage() : boolean **Beschreibung:** Prüft, ob Nachrichten im Puffer sind.	
	ID: SeMa05 **Name:** getMessage() : Message **Beschreibung:** Holt eine Nachricht aus dem Sendepuffer.	

Message-Worker	Holt sich Nachrichten aus dem RecieveManagement und verarbeitet sie. **ID:** MeWo01 **Name:** handleMessage() **Beschreibung:** Bearbeitet die Nachrichten.	**Attribute:** Keine **Kommunikationspartner:** **RecieveManagement:** Dauerhaftes holen von Empfangenen Nachrichten, um diese zu verarbeiten. **SendManagement:** Je nach empfangener Nachricht müssen diese zum weiterleiten an dieses Objekt übergeben werden.
SaveMessage	Realisiert gespeicherte Nachrichten **ID:** **Name:** getMessage() : Message **Beschreibung:** Gibt die Nachricht zurück. **ID:** **Name:** getCounter() : int **Beschreibung:** Gibt den Zähler zurück, wie lange die Nachricht schon gespeichert ist.	**Attribute:** Message:MessageMessage Counter:int **Kommunikationspartner:** Wird von der **PeriodicMessageSender**-Klasse benutzt um Nachrichten zu speichern.
Periodic-Message-Sender	Packt Nachrichten periodisch in das SendManagement **ID:** PMS01 **Name:** sendHelloMessage() **Beschreibung:** Packt eine Hello-Nachricht in den Puffer. **ID:** PMS02 **Name:** sendRoutingMessage() **Beschreibung: :** Packt eine Routing-Nachricht in den Puffer. **ID:** PMS03 **Name:** sendChannelMessage() **Beschreibung: :** Packt eine Channel-Nachricht in den Puffer . **ID:** PMS04 **Name:** sendSavedMessage() **Beschreibung:** Packt eine gespeicherte Nachricht in den Puffer, falls der Benutzer nun in der Routingtabelle steht. **ID:** PMS05 **Name:** addSaveMessage(smsg:SaveMessage) **Beschreibung:** Fügt eine zu speichernde Nachricht hinzu.	**Attribute:** Keine **Kommunikationspartner:** **SendManagement:** Periodisch werden die notwendigen Nachrichten wieder an dieses Objekt gegeben

<<abstract>> Message	Realisiert die Nachrichten. Für alle Attribute gibt es get-Methoden und set-Methoden **ID:** **Name:** getXML() : XMLData **Beschreibung:** Wandelt die Nachrichten in XML-Daten um. **ID:** **Name:** static parseXML(x:XMLData) : Message **Beschreibung:** Wandelt XML-Daten in Message-Objekte um.	**Attribute:** id:String ttl:int flood:boolean delay:int signtype:String signatur:String **Kommunikationspartner:** Wird von den anderen Klassen benutzt um Nachrichten zu behandeln. Gleiches gilt für alle Spezialisierungen.
HelloMessage	Überlagerte getXML()-Funktion	**Attribute:** sender:String version:int greeting:String
RoutingMessage	Überlagerte getXML()-Funktion	**Attribute:** hops:int user:String
KeyMessage	Überlagerte getXML()-Funktion	**Attribute:** sender:String receiver:String channelname:String channeled:String cipher:String keyloc:String
GetKey-Message	Überlagerte getXML()-Funktion	**Attribute:** sender:String receiver:String channelname:String
NackMessage	Überlagerte getXML()-Funktion	**Attribute:** sender:String message:MessageMessage

AckMessage	Überlagerte getXML()-Funktion	**Attribute:** sender:String
Certificate-Message	Überlagerte getXML()-Funktion	**Attribute:** sender:String receiver:String certificatloc:String
GetCertificate-Message	Überlagerte getXML()-Funktion	**Attribute:** sender:String receiver:String
Channel-Message	Überlagerte getXML()-Funktion	**Attribute:** sender:String channelname:String channeled:String description:String closed:boolean
JoinMessage	Überlagerte getXML()-Funktion	**Attribute:** sender:String channelname:String channeled:String
Leave-Message	Überlagerte getXML()-Funktion	**Attribute:** sender:String channelname:String channeled:String
Obscure-Message	Überlagerte getXML()-Funktion	**Attribute:** receiver:String text:String
Message-Message	Überlagerte getXML()-Funktion	**Attribute:** sender:String receiver:StringList timepstamp:String channelname:String channeled:String text:String

		Kommunikationspartner: **Attachement:** Wird benutzt um Anhänge zu speichern.
Attachement	**ID:** **Name:** getXML() : XMLData **Beschreibung:** Wandelt den Anhang XML-tauglich um.	**Attribute:** filename:String apptype:String database64:Data iv:String **Kommunikationspartner:** Wird von **MessageMessage**-Objekten für Anhänge benutzt

3.5 Implementierung von Komponente K04: Sicherheit

Die Secure Komponente liegt als einzelne Klasse vor, genauer, als Singleton, die Klasse darf also nur einmal erzeugt werden und ist global verfügbar. Die Methoden der Klasse sollen die Sicherheitsvorgaben des Protokolls erfüllen. So wird in einem Geschlossenen Kanal ver- und entschlüsselt via Data Encryption Standard, im Öffentlichen und Geschlossenen Kanal wird via X.509 zertifiziert und authentifiziert um in Anonymen Kanal wird mit Hilfe des X.509 Zertifikats nach dem Zwiebelprinzip ver-/entschlüsselt

3.5.1 Klassendiagramm

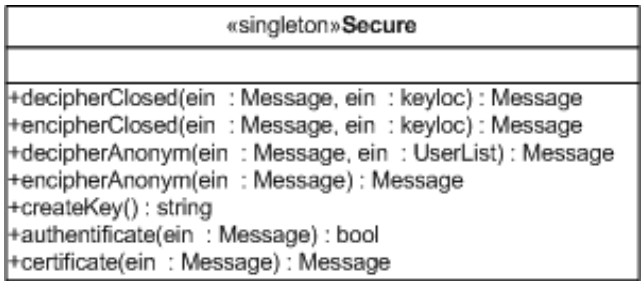

Abbildung 5: Klassendiagramm: Secure

3.5.2 Erläuterung

Klasse	Aufgabe	Attribute und Kommunikationspartner
Secure	**ID:** Secure01 **Name:** decipherClosed(m: Message, keyloc: String) : Message **Beschreibung:** Verschlüsselt eine Nachricht mit dem Schlüssel des jeweiligen Geschlossenen Kanals, falls vorhanden auch den Anhang.	**Attribute:** Keine **Kommunikationspartner:** **Allgemein** wird dies zum verschlüssel, entschlüsseln, zertifizieren und authentifizieren von Messageobjekten benutzt.
	ID: Secure02 **Name:** encipherClosed(m: Message, keyloc: String) : Message **Beschreibung:** Entschlüsselt den Inhalt einer Nachricht eines Geschlossenen Kanals und falls vorhanden auch den Anhang.	
	ID: Secure03 **Name:** decipherAnonym(m: Message, l: UserList) : Message **Beschreibung:** Erhält eine MESSAGE Nachricht, verschlüsselt diese und erstellt daraus eine OBSCURE Nachricht,	

	so oft, wie User in der UserList stehen. Verschlüsselt wird mit den Zertifikaten der User aus der Liste.	
	ID: Secure04 **Name:** encipherClosed(m: Message) : Message **Beschreibung:** Entschlüsselt den Inhalt einer Nachricht eines Anonymen Kanals.	
	ID: Secure05 **Name:** createKey() : String **Beschreibung:** Erstellt einen Schlüssel für einen Geschlossenen Kanal.	
	ID: Secure06 **Name:** authentificate(m: Message) : boolean **Beschreibung:** Liefert „true“, falls das Zertifikat, des Senders, der Nachricht, gültig ist.	
	ID: Secure07 **Name:** certificate(m: Message) : Message **Beschreibung:** Zertifiziert eine Nachricht.	

3.6 Implementierung von Komponente K05: Senden/Empfangen

Die Senden/Empfangen-Komponente übernimmt das Senden und empfangen der Nachrichten über UDP. Es holt sich Nachrichten aus der Nachrichtenkomponente um sie zu verschicken und übergibt empfangene Nachrichten. Der Port in dem die Nachrichten empfangen werden kann bei Benutzung des Infrastrukturmodus geändert werden.

3.6.1 Klassendiagramm

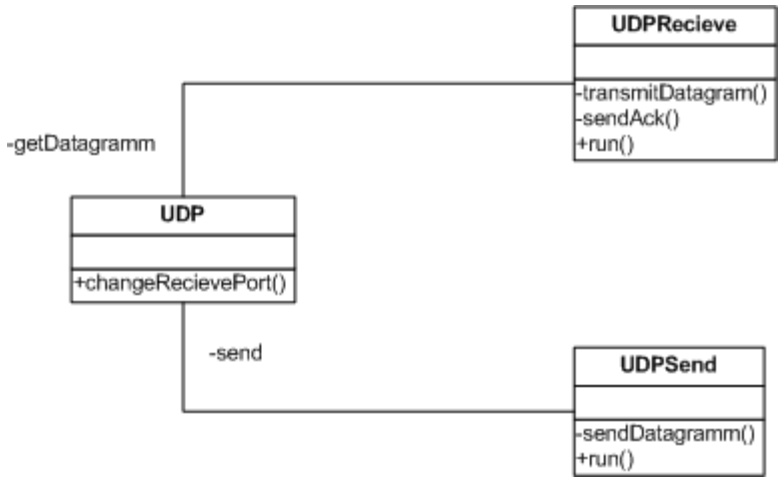

Abbildung 6: Klassendiagramm: Senden/Empfangen

3.6.2 Erläuterung

Klasse	Aufgabe	Attribute und Kommunikationspartner
UDP-Recieve	Übergibt empfangene Nachrichten an das RecieveManagement **ID:** UDPR01 **Name:** transmitDatagram() **Beschreibung:** Übergeben der Nachricht **ID:** UDPR02 **Name:** sendAck() **Beschreibung:** Sendet sofort die Bestätigung beim Eintreffen.	**Attribute:** Keine **Kommunikationspartner:** **RecieveManagement:** Nachrichten die ankommen müssen als **Messageobjekte** übergeben werden. **UDP:** Zum Nachrichten aus dem Netz holen
UDPSend	Holt sich Nachrichten aus dem SendManagement und versendet sie. **ID:** UDPS01 **Name:** sendDatagramm() **Beschreibung:** Holen der Nachricht und senden	**Attribute:** Keine **Kommunikationspartner:** **SendManagement:** Nachrichten, die losgeschickt werden sollen müssen von diesem Objekt geholt werden. **UDP:** Zum Nachrichten in das Netz geben

<table>
<tr><td rowspan="2">UDP</td><td>UDPKlasse die das Sendet/Empfangen realisiert.</td><td rowspan="2"><u>Attribute:</u>
Keine

<u>Kommunikationspartner:</u>
Wird von UDPSend und UDPRecieve genutzt um Nachrichten aus dem Netz empfangen und ins Netz senden zu können.</td></tr>
<tr><td>ID: UDP01
Name: changeRecievePort()
Beschreibung: Änderung des Ports über den die Datagramme gesendet und empfangen werden.</td></tr>
</table>

3.7 Implementierung von Komponente K06: Routing

Die Routingkomponente ist dafür zuständig die Netzstruktur durch Hello- und RoutingNachrichten aufzubauen. Ebenfalls kann man Routingnachrichten erstellen lassen und RoutingRichtungen abfragen.

3.7.1 Klassendiagramm

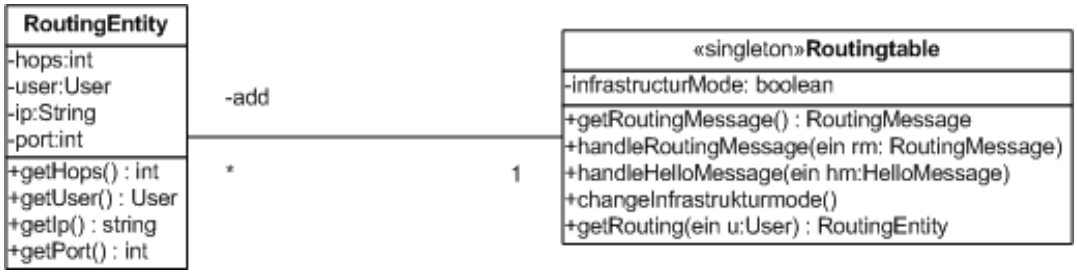

Abbildung 7: Klassendiagramm: Routing

3.7.2 Erläuterung

Klasse	Aufgabe	Attribute und Kommunikationspartner
<<singleton>> Routingtable	Speichert die Routingeinträge	**Attribute:** Infrastructurmode:boolean **Kommunikationspartner:** **MessageWorker:** Der MessageWorker übergibt die richtigen Nachrichten an das Routingtableobjekt. **PeriodicMessageSender:** Holt sich aktuelle RoutingMessages, falls diese verschickt werden müssen.
	ID: RT01 **Name:** getRoutingMessage() : RoutingMessage **Beschreibung:** Gibt eine aktuelle Routing-Nachricht zurück.	
	ID: RT02 **Name:** handleRoutingMessage (rmsg:RoutingMessage) **Beschreibung:** Aktualisiert die Tabelle.	
	ID: RT03 **Name:** handleHelloMessage(hmsg:HelloMessage) **Beschreibung:** Aktualisiert die Tabelle.	
	ID: RT04 **Name:** changeInfrastrukturmode() **Beschreibung:** Ändert den Modus in dem sich die Routingtabelle befindet.	
	ID: RT04 **Name:** getRouting(u:User) : RoutingEntity **Beschreibung:** Gibt einem einen Routingeintrag zurück.	

<table>
<tr><td rowspan="5">RoutingEntity</td><td>Realisiert die Routingeinträge</td><td rowspan="5"><u>Attribute:</u>
hops:int
user:User
ip:String
port:int

<u>Kommunikationspartner:</u>
Wird von der Routingtable-Klasse genutzt.</td></tr>
<tr><td>ID: RE01
Name: getHops() : int
Beschreibung: Gibt die Hops zurück.</td></tr>
<tr><td>ID: RE02
Name: getUser() : User
Beschreibung: Gibt den User zurück.</td></tr>
<tr><td>ID: RE03
Name: getIp() : String
Beschreibung: Gibt die Ip zurück.</td></tr>
<tr><td>ID: RE04
Name: getPort : int
Beschreibung: Gibt den Port zurück.</td></tr>
</table>

4 Datenmodell

Für das Programm wird kein Datenmodell benötigt. Pro Programm werden lediglich eingestellte Fenstereinstellungen gespeichert, die kein Beziehungsmodel benötigen. Ebenfalls benötigen die Zertifikate und Schlüssel keine Datenbeziehungsmodelle. Sie werden neben dem Programm in einem „Data“ Ordner als Dateien gespeichert.