

# Ad-Hoc Chatsystem für mobile Netze Barracuda

Softwareentwicklungspraktikum  
Sommersemester 2007

## Grobentwurf



### Auftraggeber

Technische Universität Braunschweig  
Institut für Betriebssysteme und Rechnerverbund  
Prof. Dr.-Ing. Lars Wolf  
Mühlenpfordtstraße 23  
38106 Braunschweig

### Betreuer

Sven Lahde, Oliver Wellnitz

### Auftragnehmer

Gruppe 2

Name	E-Mail
Stephan Friedrichs	stephan.friedrichs@tu-bs.de
Henning Günther	h.guenther@tu-bs.de
Oliver Mielentz	o.mielentz@tu-bs.de
Martin Wegner	mw@mroot.net

Braunschweig, 2. Mai 2007

# Versionsübersicht

Phasenverantwortlicher: Martin Wegner

Version	Datum	Autor	Status	Kommentar
1	2. Mai 2007	Gruppe 2 (s. Auftragnehmer)		Erste Version

## Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>2</b>
1.1	Erfassen der Netzstruktur . . . . .	2
1.2	Nachrichten an Kanäle . . . . .	3
1.3	Broadcasts . . . . .	3
1.4	Nachrichten-Fehlerbehandlung . . . . .	4
1.5	Routing . . . . .	4
<b>2</b>	<b>Analyse der Produktfunktionen</b>	<b>5</b>
2.1	/F100/:Rendezvous . . . . .	5
2.1.1	Grobanalyse . . . . .	5
2.1.2	Feinanalyse . . . . .	5
2.2	/F200/: Nachricht an öffentlichen Kanal . . . . .	7
2.2.1	Grobanalyse . . . . .	8
2.2.2	Feinanalyse . . . . .	9
2.3	/F300/: Anonyme Kommunikation . . . . .	10
2.3.1	Grobanalyse . . . . .	10
2.3.2	Feinanalyse . . . . .	11
2.4	/F400/: Partitionierung und Neusynchronisation . . . . .	12
2.4.1	Grobanalyse . . . . .	12
2.4.2	Feinanalyse . . . . .	13
<b>3</b>	<b>Resultierende Softwarearchitektur</b>	<b>13</b>
3.1	Komponentenspezifikation . . . . .	13
3.1.1	UDP-Port . . . . .	13
3.1.2	MessageParser . . . . .	14
3.1.3	Distributor . . . . .	14
3.1.4	ServerInterface . . . . .	14
3.1.5	Benutzeroberfläche (GUI) . . . . .	14
3.1.6	Data . . . . .	14
3.1.7	Transmitter . . . . .	14
3.2	Schnittstellenspezifikation . . . . .	14
3.2.1	UDP-Port . . . . .	14
3.2.2	MessageParser . . . . .	15
3.2.3	Distributor . . . . .	15
3.2.4	ServerInterface . . . . .	15
3.2.5	Benutzeroberfläche (GUI) . . . . .	15
3.2.6	Data . . . . .	16
3.2.7	Transmitter . . . . .	16
3.3	Protokolle für die Benutzung der Komponenten . . . . .	17
3.3.1	MessageParser . . . . .	17
3.3.2	Transmitter . . . . .	18
3.3.3	Data . . . . .	18
3.3.4	ServerInterface . . . . .	19
3.3.5	Distributor . . . . .	19
<b>4</b>	<b>Verteilungsentwurf</b>	<b>20</b>

# 1 Einleitung

Zur Realisierung des gegebenen Ad-Hoc Chat-Protokolls ist es notwendig, dass die Netzstruktur und deren Änderungen permanent erfasst werden. Außerdem soll die Kommunikation in einem festen anonymen Kanal, sowie öffentlichen und privaten Kanälen möglich sein. Die Herkunft von anonymen Nachrichten wird verschleiert, Nachrichten in privaten Kanälen werden verschlüsselt. Allgemein werden Nachrichten signiert. Die gewonnenen Routinginformationen können überschrieben werden. Die Benutzer sind über fehlgeschlagene Zustellungen von Nachrichten zu informieren.

Das Programm wird intern in einen Server mit dem ServerInterface und in die an dieses andockbare Oberfläche aufgeteilt. Hierdurch kann ein Mehrbenutzerbetrieb auf einem Knoten erreicht werden.

## 1.1 Erfassen der Netzstruktur

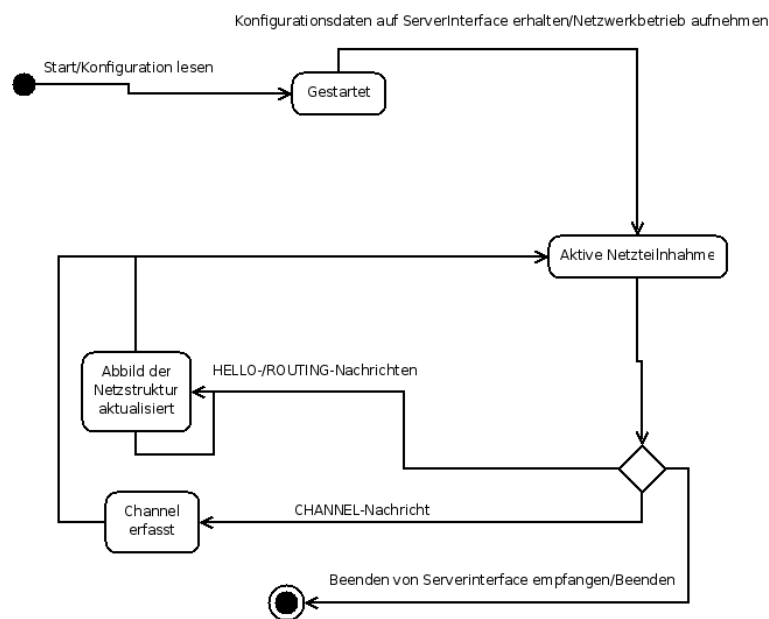


Abbildung 1: Erfassen der Netzstruktur

Beim Starten des Programms wird die Konfiguration, die möglicherweise zu setzende Routing-Informationen enthält, gelesen. Die Netzstruktur wird aus empfangenen HELLO- und ROUTING-Nachrichten gewonnen. Existente Kanäle werden den CHANNEL-Nachrichten entnommen.

## 1.2 Nachrichten an Kanäle

Es kann vom Benutzer ein neuer Kanal eröffnet werden. Die Herkunft der Nachrichten an den omnipräsenten anonymen Kanal wird verschleiert („obscuring“). Nachrichten an einen öffentlichen Kanal werden direkt versendet, Nachrichten an einen privaten werden zunächst mit dem dem Kanal zugehörigen symmetrischen Schlüssel verschlüsselt.

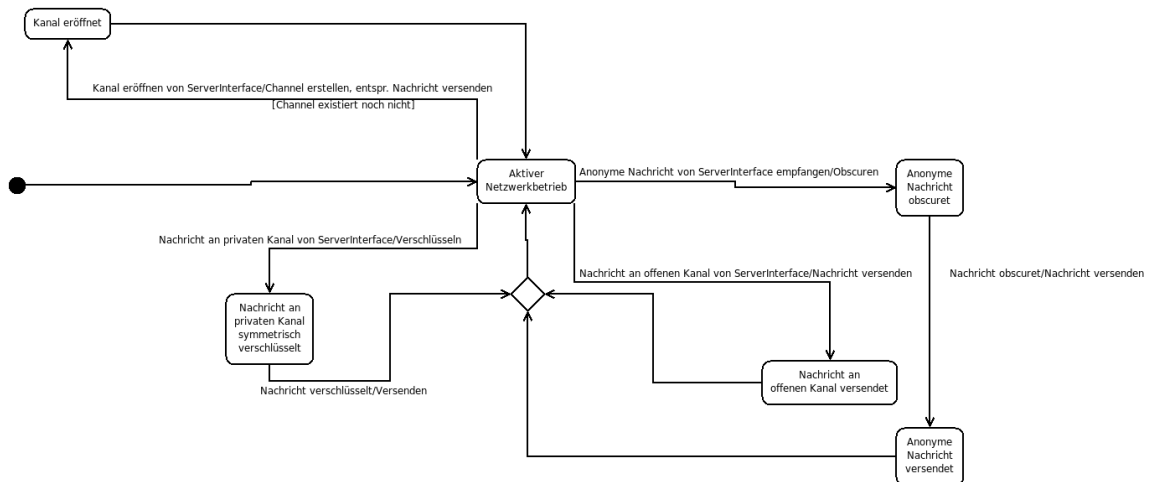


Abbildung 2: Nachrichten an Kanäle

## 1.3 Broadcasts

Um die Netzwerkstruktur zu erhalten, werden periodisch verschiedene Broadcasts an das Netz gesendet: Ein HELLO geht an alle benachbarten Knoten, genau so wie die bekannten Routinginformationen. Wird in einem bestimmten Zeitintervall für einen lokal beigetretenen Kanal keine Ankündigung empfangen, wird für diesen eine versendet.

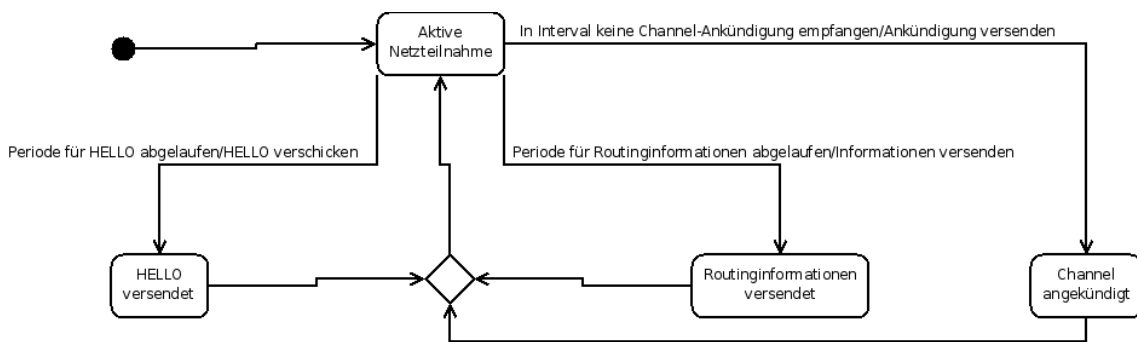


Abbildung 3: Broadcasts

### 1.4 Nachrichten-Fehlerbehandlung

Nachrichten, die nicht an lokale Benutzer gerichtet sind, werden durch das Netzwerk weiter versendet. Nachrichten an lokale Benutzer werden anhand der Signatur auf ihre Gültigkeit geprüft. Ist die Signatur gültig, wird die Nachricht über das ServerInterface an die Benutzeroberfläche weitergeleitet. Ist die Signatur ungültig, wird der lokale Benutzer hierüber ebenfalls in der Benutzeroberfläche informiert.

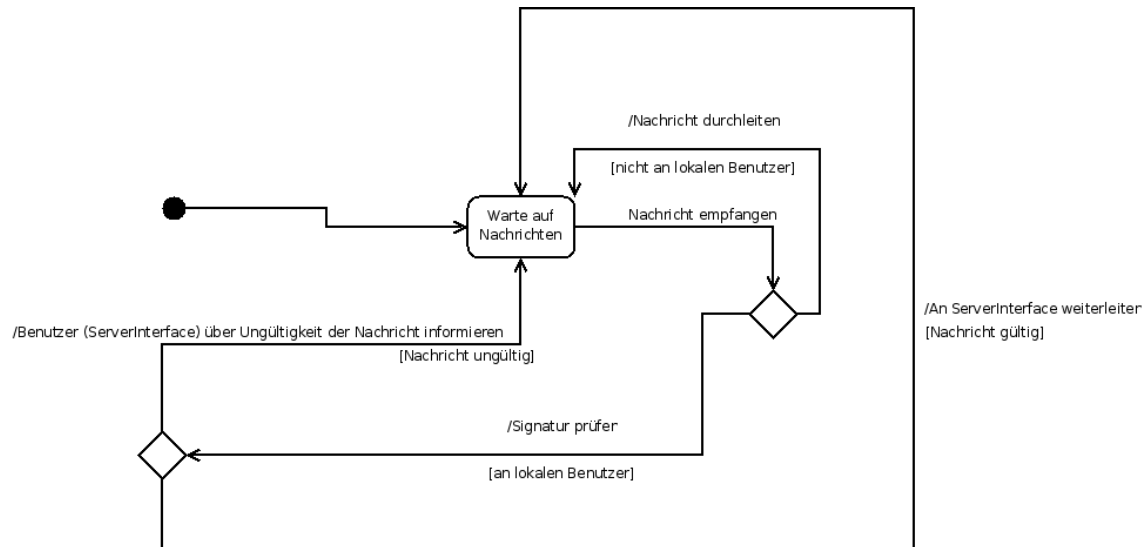


Abbildung 4: Nachrichten-Fehlerbehandlung

### 1.5 Routing

Wenn eine Nachricht empfangen wurde, wird an den vorherigen Hop eine Bestätigung übermittelt. Dann wird versucht, eine Route für die Nachricht zu ermitteln, sofern sie nicht an einen lokalen Benutzer adressiert ist. Schlägt dies fehl, wird dem Sender der Nachricht der Misserfolg mitgeteilt. Wurde eine Route gefunden, so wird die Nachricht über diese weitergeschickt.

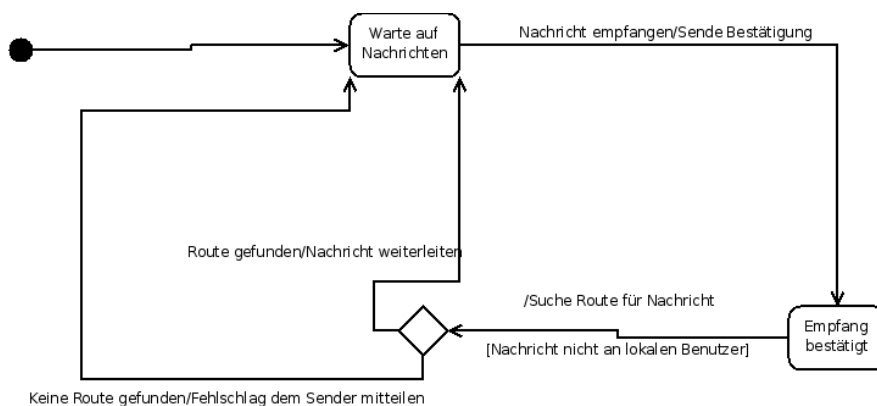


Abbildung 5: Routing

## 2 Analyse der Produktfunktionen

### 2.1 /F100/:Rendezvous

Die Rendezvous-Funktionalität wird benötigt, damit Nodes erfahren, dass sie sich in Übertragungsbereichweite befinden.

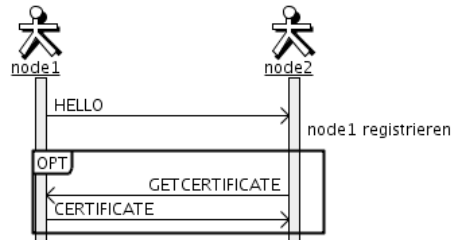


Abbildung 6: Ablauf des Rendezvous

#### 2.1.1 Grobanalyse

Wird eine *HELLO*-Nachricht empfangen, so muss die IP des Absenders in der Routing Tabelle als direkter Nachbar gespeichert werden. Desweiteren kann von dem Benutzer, der mithilfe der *HELLO*-Nachricht bekannt geworden ist, sofern noch keins vorhanden ist, ein Zertifikat angefordert werden.

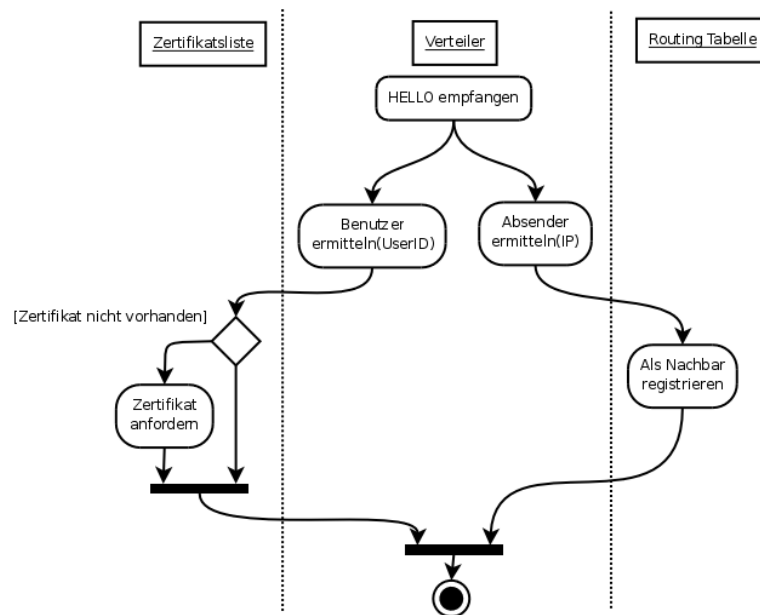


Abbildung 7: Verarbeitung einer *HELLO*-Nachricht

#### 2.1.2 Feinanalyse

Um eine neue Nachricht zu empfangen, muss zunächst am UDP-Port ein Datagramm empfangen werden. Der Textinhalt muss dann vom Nachrichten-Parser analysiert werden. Man beachte, dass diese Pull-Architektur in Haskell eher implizit durch unendliche Listen realisiert werden wird. Der UDP-Port hat dann eine Funktion „getDatagrams“ die eine unendliche Liste der Datagramme zurückgeben würde, die dann durch die entsprechenden Komponenten geleitet wird.

Nachdem die Nachricht geparkt ist, muss im Falle der *HELLO*-Nachricht IP-Adresse sowie Benutzer-ID der Nachricht ermittelt werden. Auch dieser Vorgang wird etwas leichter realisiert

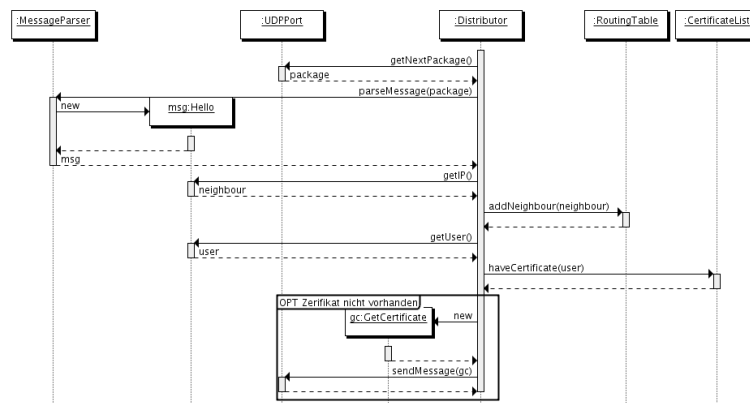


Abbildung 8: Interner Ablauf

werden als im Sequenzdiagramm angegeben, statt Funktionsaufrufen wird ein Pattern-Matching auf den *HELLO*-Datentyp durchgeführt.

Die IP-Adresse wird jetzt in die Routing-Tabelle als Nachbar vermerkt. Desweiteren könnte der Benutzer dort mit der Information, dass man ihn mit einem Hop erreicht gespeichert werden.

## 2.2 /F200/: Nachricht an öffentlichen Kanal

Diese Funktionalität ist notwendig, um Basis-Chatmöglichkeiten zu bieten.

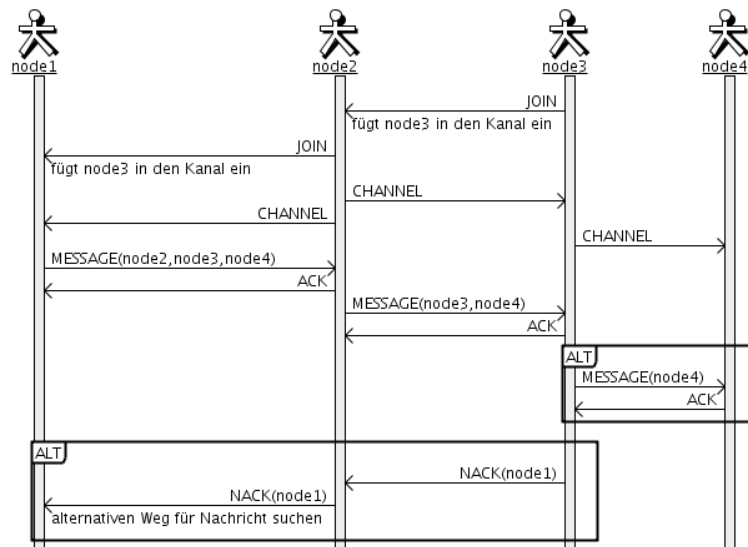


Abbildung 9: Ablauf einer öffentlichen Kanal-Nachricht

### 2.2.1 Grobanalyse

Zum Verarbeiten einer *MESSAGE*-Nachricht sind folgende Schritte notwendig:

- Prüfen der Signatur
- Ausliefern an lokale Benutzer
- Routen für entfernte Benutzer ermitteln
- Weiterleiten an entfernte Benutzer

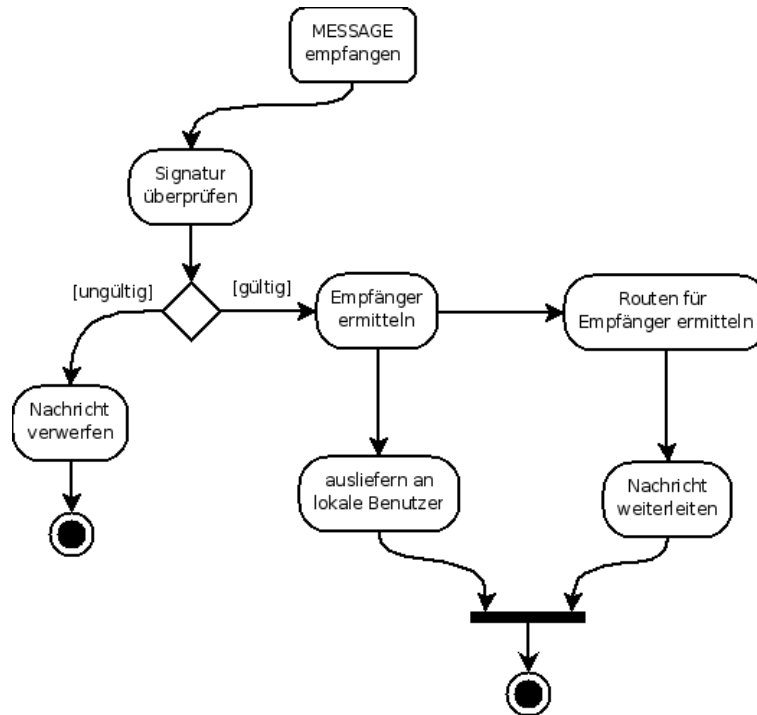


Abbildung 10: Verarbeitung einer öffentlichen Nachricht

### 2.2.2 Feinanalyse

Nachdem die Nachricht geparkt und als *MESSAGE*-Nachricht erkannt wurde, muss die Signatur geprüft werden. Ist die Signatur ungültig, wird die Nachricht verworfen. Danach müssen die Empfänger der Nachricht ermittelt werden. Aus den Empfängern werden die lokalen Benutzer entfernt und die Nachricht an sie zugestellt. Für die verbliebenen Benutzer werden mit Hilfe der Routing-Tabelle die Nachbar-Nodes ermittelt, an die die Nachricht weiter verschickt werden muss. Als letztes wird die Nachricht entsprechend an die Nodes verschickt.

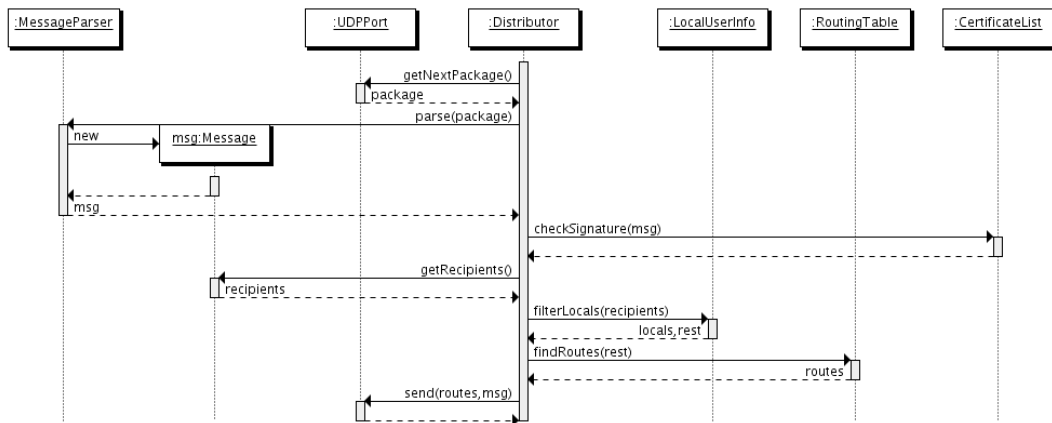


Abbildung 11: Interner Ablauf

### 2.3 /F300/: Anonyme Kommunikation

Anonyme Kommunikation wird ermöglicht durch das Zwiebschalenprinzip: 3 bis 5 Benutzer, mit deren Schlüssel die Nachricht nacheinander verschlüsselt wird, werden zufällig gewählt. Danach wird die Nachricht auf dem umgekehrten Verschlüsselungsweg verschickt.

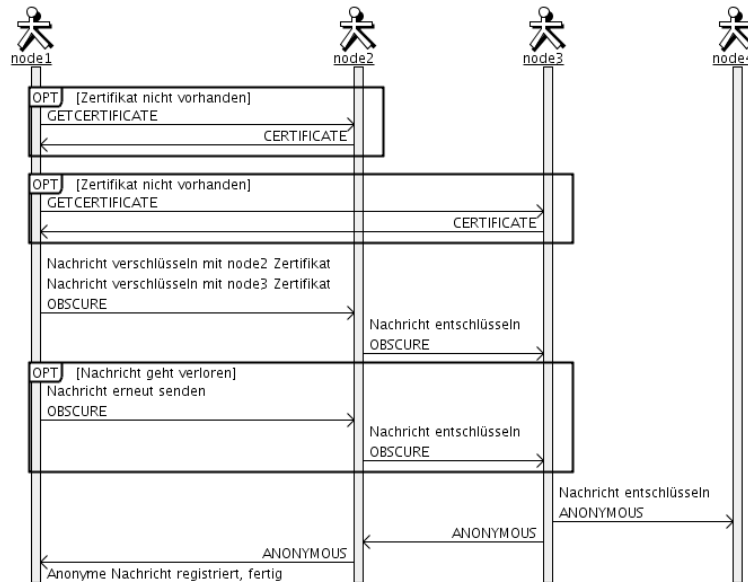


Abbildung 12: Ablauf einer anonymen Kommunikation

#### 2.3.1 Grobanalyse

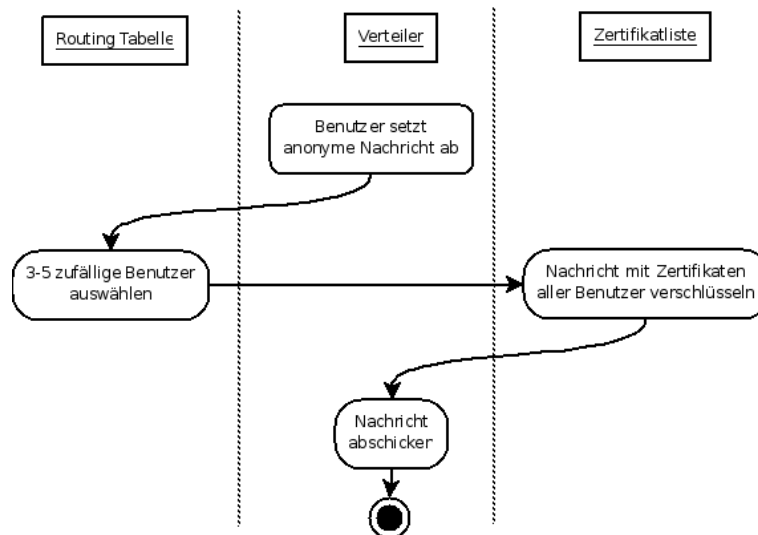


Abbildung 13: Erzeugung einer anonymen Nachricht

### 2.3.2 Feinanalyse

Als erstes erzeugt der Distributor eine Nachricht. Aus der Routingtabelle können dann Benutzer gewählt werden, mit deren Zertifikat (das evtl. angefordert werden muss) nacheinander verschlüsselt wird. Danach wird das verschlüsselte Paket an den letzten Verschlüsseler verschickt.

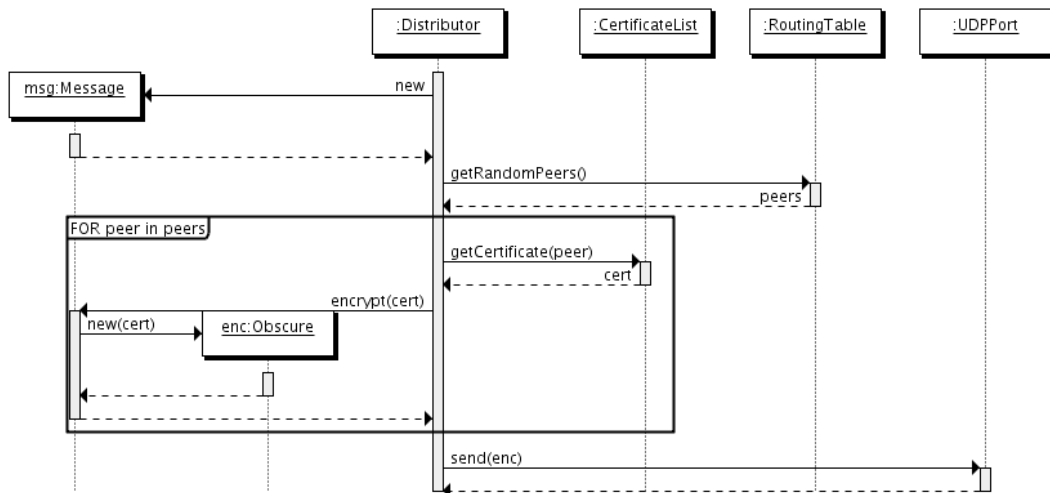


Abbildung 14: Detaillierte Erzeugung einer anonymen Nachricht

## 2.4 /F400/: Partitionierung und Neusynchronisation

Falls das Netzwerk durch Knotenausfälle in mehrere unabhängige Teile zerlegt wird, müssen Kanäle wieder zusammengeführt werden.

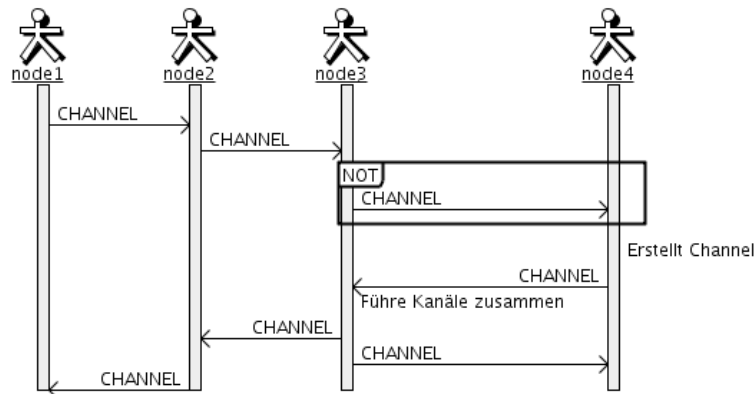


Abbildung 15: Ablauf einer Kanalsynchronisation

### 2.4.1 Grobanalyse

Zunächst muss zwischen privaten und öffentlichen Kanälen unterschieden werden. Öffentliche Kanäle werden immer dann zusammen geführt, wenn der Name gleich ist, private jedoch nur, wenn ihre Kanal-ID gleich ist. Desweiteren muss bei privaten Kanälen überprüft werden, ob der Absender einer **CHANNEL**-Nachricht schon vorher im Kanal war.

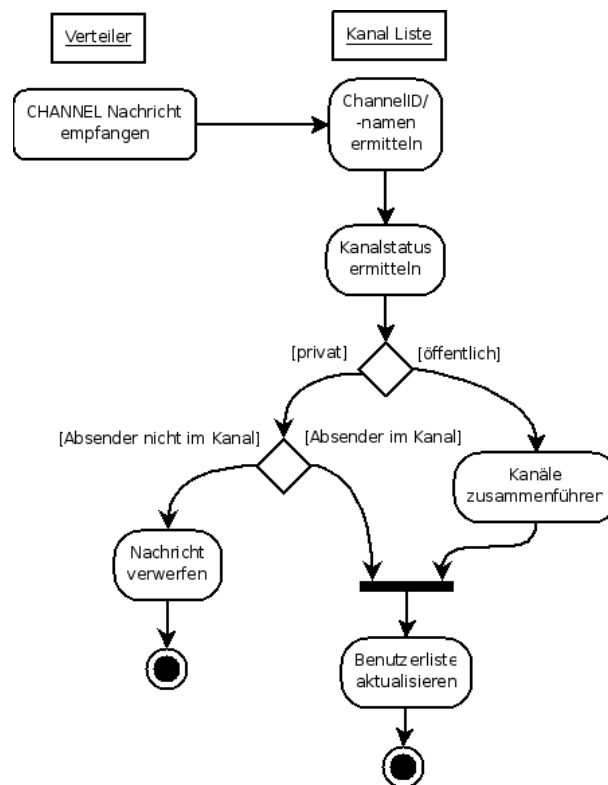


Abbildung 16: Bearbeitung einer **CHANNEL**-Nachricht

### 2.4.2 Feinanalyse

Nachdem die Nachricht wie in den anderen Funktionalitäten besprochen geparkt wurde, müssen Kanal-ID, sowie Benutzer im Kanal ermittelt werden und in der Kanal-Liste entsprechend angepasst werden.

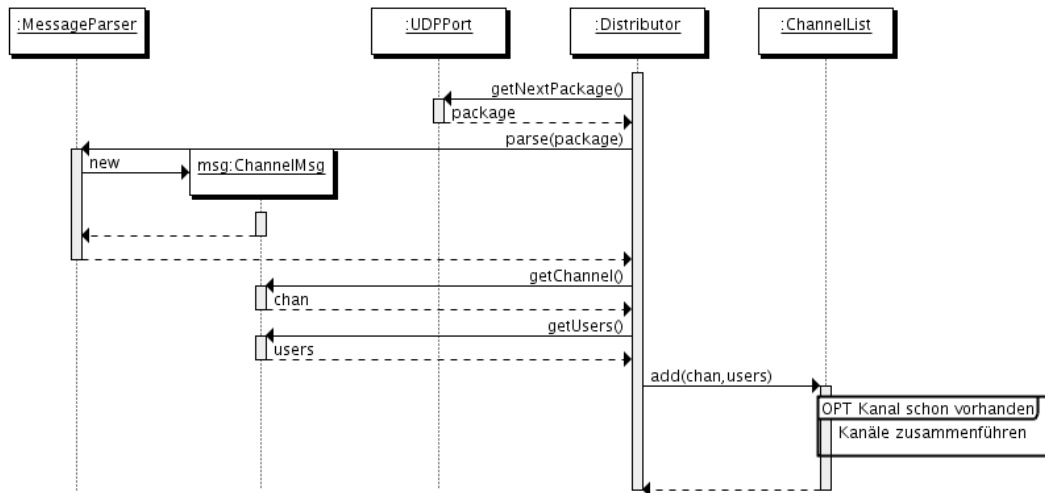


Abbildung 17: Interne Bearbeitung einer Kanalzusammenführung

## 3 Resultierende Softwarearchitektur

### 3.1 Komponentenspezifikation

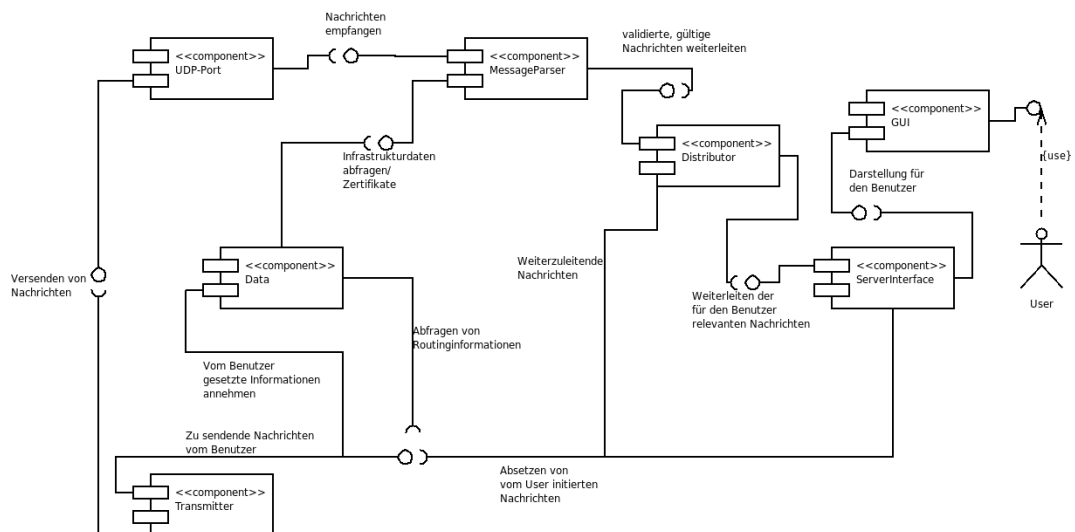


Abbildung 18: Komponentenentwurf

Das Programm gliedert sich in die großen Komponenten UDP-Port, MessageParser, Distributor, ServerInterface, GUI, Data und Transmitter.

#### 3.1.1 UDP-Port

Auf dem UDP-Port werden Nachrichten empfangen und gesendet.

### 3.1.2 MessageParser

Der MessageParser parst die eingehenden XML-Nachrichten. Hierzu werden zunächst die Infrastrukturdaten und die Zertifikate benutzt, um Nachrichten von bestimmten Knoten zu ignorieren (Infrastrukturmodus) und invalide herauszufiltern.

### 3.1.3 Distributor

Der Distributor agiert als eine Art Weiche für die verschiedenen Nachrichtentypen und leitet diese jeweils an die relevanten Komponenten weiter. Dazu generiert er aus den eingehenden Nachrichten Informationsströme für

- ausgehende Nachrichten (Transmitter)
- die Netzwerktopologie (Data)
- die GUI (ServerInterface)

### 3.1.4 ServerInterface

Das Serverinterface stellt die Kommunikationsschnittstelle für die Benutzeroberfläche bereit. Hierzu werden zum einen Events generiert, die die Benutzeroberfläche über bestimmte Aktivitäten (empfangene Nachrichten, Änderung der Netzstruktur, etc.) benachrichtigen. Zum anderen nimmt das ServerInterface über die Benutzeroberfläche durch den Benutzer getätigte Aufträge (Kanal beitreten, Nachricht senden, etc.) entgegen.

### 3.1.5 Benutzeroberfläche (GUI)

Die Benutzeroberfläche visualisiert über das ServerInterface empfangene Events geeignet für den Benutzer und bietet die Möglichkeit Aufträge für das ServerInterface zu erzeugen.

### 3.1.6 Data

Data ist die Datenhaltungskomponente und speichert die Netzwerktopologie. Hierzu werden verwaltet:

- Eine Liste von bekannten Zertifikaten (CertificateList)
- Die Nutzerkennungen und die Mitgliedschaften in in Kanälen der lokalen Benutzer (LocalUserInfo)
- Die Routinginformationen, u. a. die erreichbaren Nutzer im Netzwerk (RoutingTable)
- Die Liste ovn allen bekannten Kanälen und deren Mitgliedern (ChannelList)

### 3.1.7 Transmitter

Der Transmitter nimmt alle vom Programm ausgehenden Nachrichten (Broadcasts, direkte P2P-Nachrichten, zu routende Nachrichten) an, erzeugt die dazu benötigten XML-Dokumente aus diesen Nachrichten, versendet diese und verwaltet diese für eine eventuelle weitere Bearbeitung (z. B. zur Erkennung des Misserfolgs der Zustellung).

## 3.2 Schnittstellenspezifikation

### 3.2.1 UDP-Port

Die Implementierung des UDP-Ports ist gegeben, siehe GHC-Dokumentation.

### 3.2.2 MessageParser

Operation	Beschreibung
isInfrastructure(): Bool	Gibt zurück, ob der Infrastrukturmodus aktiviert ist
setInfrastructure(Bool): void	Aktiviert/deaktiviert den Infrastrukturmodus
getInfraPeers(): [HostAddress]	Gibt die Liste von Peers zurück, die bei aktiviertem Infrastrukturmodus als bekannt angenommen wird
setInfraPeers(): [HostAddress]	Setzt die Liste von Peers, die bei aktiviertem Infrastrukturmodus als bekannt angenommen wird ohne den Infrastrukturmodus zu aktivieren
parse(XML): ProtocolMessage	Parst XML zu einer Nachricht

### 3.2.3 Distributor

Operation	Beschreibung
receive(ProtocolMessage): void	Nimmt eine protokollkonforme Nachricht an und leitet sie an die entsprechenden Komponenten weiter

### 3.2.4 ServerInterface

Operation	Beschreibung
readMessages(): [ServerUIMessage]	Liest Nachrichten für die Nutzeroberfläche
writeMessage(UIServerMessage): Either String ()	Sendet einen Befehl an die Netzwerkmodule und gibt zurück, ob der Befehl ausgeführt wurde und falls nein, eine Fehlermeldung

### 3.2.5 Benutzeroberfläche (GUI)

Die GUI kommuniziert über das ServerInterface. Es müssen keine Schnittstellen oder Internas dieses Moduls bekannt sein, es ist komplett austauschbar.

### 3.2.6 Data

Operation	Beschreibung
addCertificate(UserID, Certificate): void	Speichert das Zertifikat eines Teilnehmers im Ad-hoc Netzwerk
getCertificate(UserID): Maybe Certificate	Gibt das Zertifikat eines Nutzers zurück, falls es bekannt ist
saveCertificates(File): void	Speichert die Zertifikate permanent in einer Datei
getLocalUsername(): UserID	Ermittelt den lokal aktiven Nutzer
setLocalUsername(UserID): void	Setzt den lokal aktiven Nutzer
getLocalUserChannels(): [Channel]	Gibt die Kanäle zurück, in denen der lokale Nutzer aktiv ist
setLocalUserChannels([Channel]): void	Setzt die Kanäle, denen der lokale Nutzer beigetreten ist
lookupRoute(UserID): Maybe HostAddress	Gibt den Anfangspunkt der kürzesten Route zum Nutzer zurück, falls eine bekannt ist
updateRoutingTable(HostAddress, [(UserID, HostAddress)]): void	Speichert die Routingtabelle eines Peers
deleteRouts(TimeOfDay): void	Entfernt alle Routinginformationen, die älter als der gegebene Timestamp sind
getChannels(): [(Channel, [UserID])]	Gibt eine Liste von aktuell bekannten Kanälen und deren Mitgliedern zurück
updateChannel(Channel, [UserID]): void	Aktualisiert die bekannten Informationen über einen Channel
deleteChannels(TimeOfDay): void	Löscht alle Kanäle, über die seit dem gegebenen Timestamp keine Informationen empfangen wurden
getKey(Channel): Maybe Key	Gibt den symmetrischen Schlüssel für den privaten Channel zurück, falls bekannt
setKey(Channel, Key): void	Setzt den symmetrischen Schlüssel für den privaten Channel

### 3.2.7 Transmitter

Operation	Beschreibung
broadcast(ProtocolMessage): void	Führt einen Broadcast der Nachricht durch
send(ProtocolMessage, HostAddress, bool p2p): void	Sendet eine Nachricht gezielt an einen Empfänger. Ist dieser nicht direkt erreichbar und ist p2p nicht gesetzt, wird die Nachricht geroutet.

### 3.3 Protokolle für die Benutzung der Komponenten

#### 3.3.1 MessageParser

Der MessageParser bekommt einen UDP-Stream welcher nach dem bekannten Draft-Strauss-Protokoll aufgebaut sein muss. Jedes Programm, welches Nachrichten dieses Protokolls empfangen soll, kann dieses Modul nutzen.

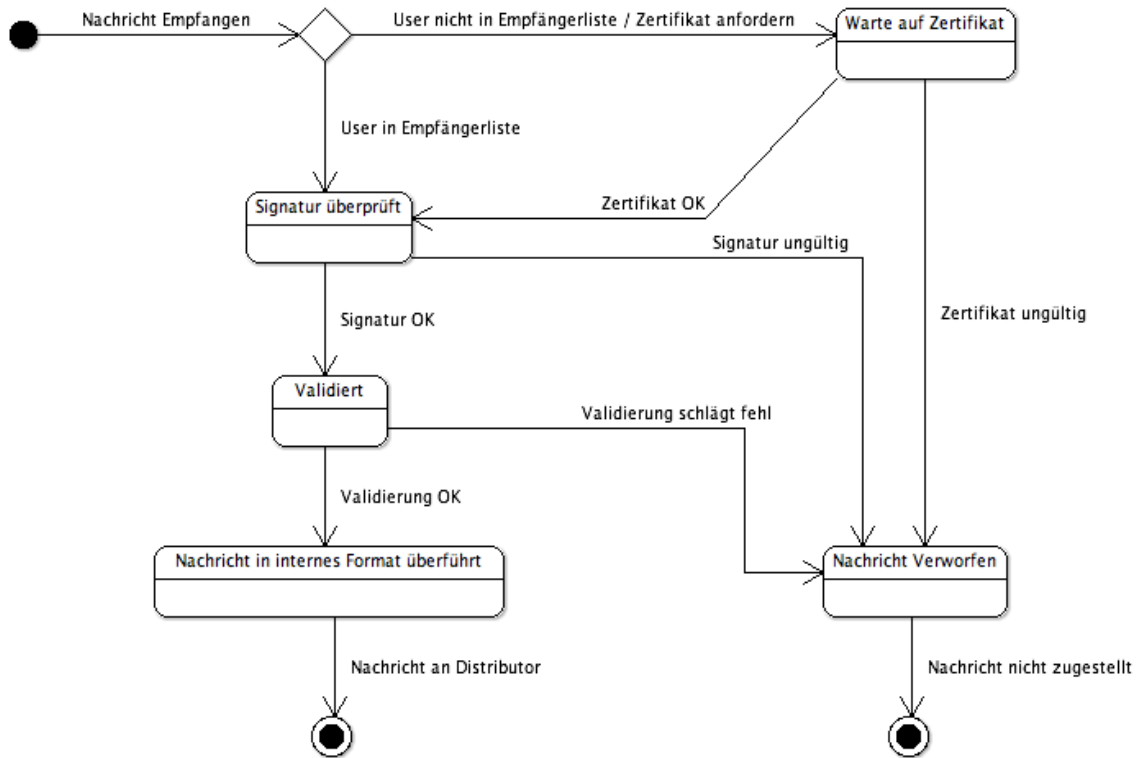


Abbildung 19: Komponentenprotokoll: MessageParser

### 3.3.2 Transmitter

Der Transmitter bekommt Daten (neue Message, ACK, NACK, ...) und Empfänger. Jedes Programm, welches Nachrichten über das Draft-Strauss-Protokoll versenden will, kann dieses Modul nutzen.

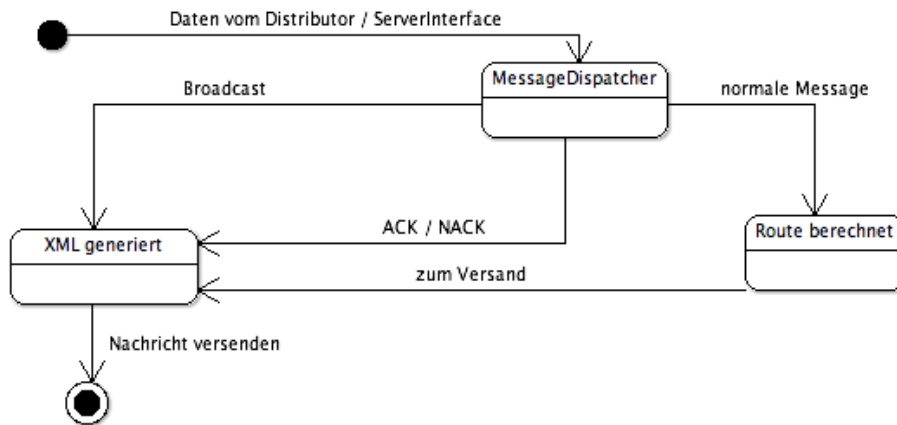


Abbildung 20: Komponentenprotokoll: Transmitter

### 3.3.3 Data

Dieses Modul ist speziell auf die in der Software gegenutzten Daten zugeschnitten und daher nicht für eine Wiederverwendung geeignet.

### 3.3.4 ServerInterface

Das ServerInterface bekommt Daten von der GUI oder dem Programm selbst. Im Fall der GUI können folgende Situationen auftreten:

**Neuen Channel erstellen:** Der Benutzer möchte einen neuen Channel eröffnen. Das ServerInterface leitet das Kommando an die entsprechenden Module der Software weiter, damit diese sich darum kümmern, einen neuen Channel zu erstellen und in der Channelliste zu publizieren.

**Neue Nachricht:** Der Benutzer hat eine neue Nachricht verfasst und möchte diese an den angegebenen Channel senden. Das ServerInterface gibt diese Nachricht weiter an den Transmitter damit die Nachricht zugestellt werden kann.

Im Fall der Software können folgenden Situationen auftreten:

**Channel-Liste aktualisiert:** Das ServerInterface leitet die aktuelle Channel Liste an die GUI weiter, damit sie dort angezeigt werden kann.

**Neue Nachricht empfangen:** Neu angekommene Nachrichten werden an die GUI weitergeleitet, damit sie angezeigt werden können.

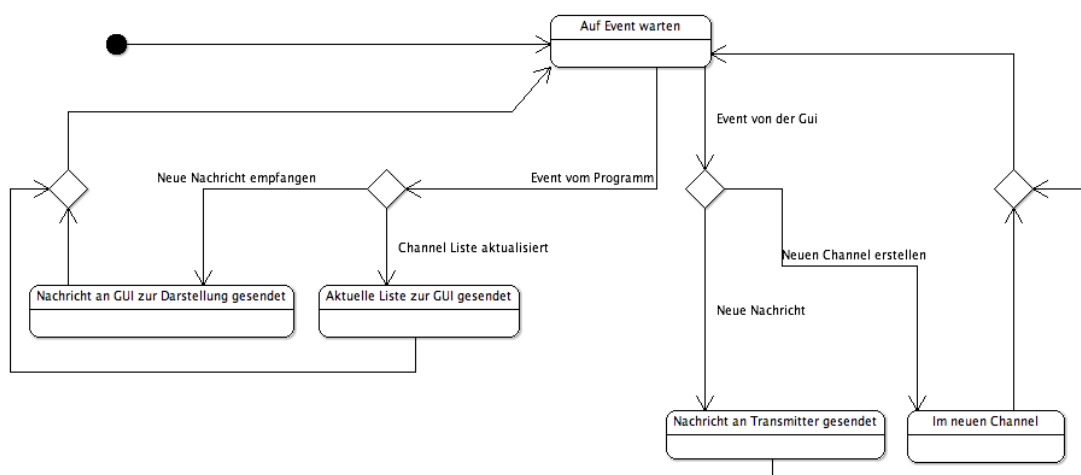


Abbildung 21: Komponentenprotokoll: ServerInterface

### 3.3.5 Distributor

Der Distributor verteilt eingehende Nachrichten an verschiedene Komponenten der Software und ist daher ebenfalls für eine Wiederverwendung nicht geeignet.

## 4 Verteilungsentwurf

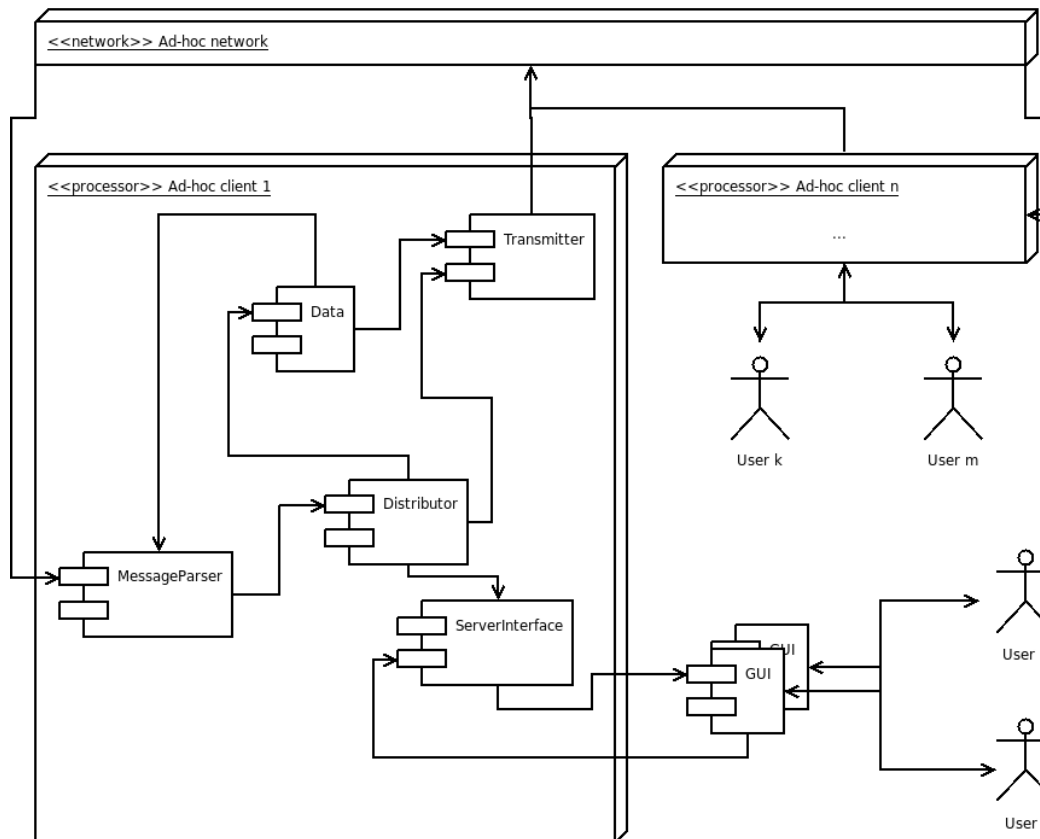


Abbildung 22: Verteilungsentwurf

Alle Knoten sind über ein Ad-Hoc Netzwerk direkt oder indirekt über andere Knoten miteinander verbunden. Auf jedem Knoten läuft ein sog. Server, der die Verwaltung der Protokollschicht übernimmt. Über das ServerInterface kann dieser von beliebig vielen Benutzern über deren Benutzeroberflächen auf dem Knoten angesprochen werden.

## Abbildungsverzeichnis

1	Erfassen der Netzstruktur . . . . .	2
2	Nachrichten an Kanäle . . . . .	3
3	Broadcasts . . . . .	3
4	Nachrichten-Fehlerbehandlung . . . . .	4
5	Routing . . . . .	4
6	Ablauf des Rendezvous . . . . .	5
7	Verarbeitung einer <i>HELLO</i> -Nachricht . . . . .	5
8	Interner Ablauf . . . . .	6
9	Ablauf einer öffentlichen Kanal-Nachricht . . . . .	7
10	Verarbeitung einer öffentlichen Nachricht . . . . .	8
11	Interner Ablauf . . . . .	9
12	Ablauf einer anonymen Kommunikation . . . . .	10
13	Erzeugung einer anonymen Nachricht . . . . .	10
14	Detaillierte Erzeugung einer anonymen Nachricht . . . . .	11
15	Ablauf einer Kanalsynchronisation . . . . .	12
16	Bearbeitung einer <i>CHANNEL</i> -Nachricht . . . . .	12
17	Interne Bearbeitung einer Kanalzusammenführung . . . . .	13
18	Komponentenentwurf . . . . .	13
19	Komponentenprotokoll: MessageParser . . . . .	17
20	Komponentenprotokoll: Transmitter . . . . .	18
21	Komponentenprotokoll: ServerInterface . . . . .	19
22	Verteilungsentwurf . . . . .	20