UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Anonymisierbare Kommunikation innerhalb verteilter Anwendungen durch Kopplung des MAppLab-RPC-Framework mit dem mPart-API-Framework

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von

Arne Fritjof Schmeiser

Erstgutachter:	Prof. Dr. J. Felix Hampe Institut für Wirtschafts- und Verwaltungsinformatik
Zweitgutachter:	Dipl.-Inform. Nico Jahn Institut für Wirtschafts- und Verwaltungsinformatik

Koblenz, im Juli 2014

Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-Rom).

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Kurzbeschreibung

Durch die zunehmende Vernetzung von Geräten sind verteilte Anwendungen eine gängige Methode in der Software-Entwicklung. Obwohl ein Bedarf an Anonymität bei der Nutzung von verteilten Anwendungen besteht, mangelt es in der Entwicklung an der Unterstützung durch Software-Frameworks. Das Erstellen von anonymisierbar kommunizierenden Anwendungen benötigt daher meist eine aufwendige Individuallösung. Diese Arbeit integriert anonymisierbare Kommunikation mittels Remote Procedure Calls (RPCs) in ein Software-Framework für verteilte Anwendungen. Dazu wird ein Binding für das MAppLab Remote Procedure Call Framework auf der Basis des mPart-API-Frameworks konzeptuell entworfen, prototypisch implementiert und in einem Beispiel-Szenario angewendet.

Abstract

The increased networking of devices has established the usage of distributed applications as a common method in software development. Despite the demand of anonymity in using distributed applications, software frameworks still lack appropriate support in developing them. Building anonymous communicating applications therefore often results in an expensive individual approach. This work integrates an approach for anonymous communication using remote procedure calls into a software framework for building distributed applications. This also includes the design, development, and prototypical implementation of a binding for the MAppLab Remote Procedure Call Framework on the basis of the mPart API Framework. Furthermore the resulting prototype will be tested in an exemplary scenario.

Inhaltsverzeichnis

1 Anonymisierbare Kommunikation innerhalb verteilter Anwendungen **1**
1.1 Ziel der Arbeit 2
1.2 Beispiel-Szenario 3
1.3 Methode: Design Science Research 4
1.4 Aufbau der Arbeit 5

2 Verteilte Systeme, Frameworks und Anonymität **6**
2.1 Verteilte Systeme und Client-Server-Modell 6
2.1.1 Zerlegung in Komponenten 7
2.1.2 Kommunikation zwischen Client und Server 8
2.2 Realisierung verteilter Systeme mittels RPC 10
2.2.1 Vereinbarungen zwischen Client und Server 11
2.3 Zweck eines Software-Frameworks 12
2.4 Arten von Anonymität 13

3 Vorhandene Konzepte für RPC und anonymisierte Netzwerke **16**
3.1 Das MAppLab RPC Framework 16
3.1.1 Gemeinsames Service-Interface 16
3.1.2 Architektur von MappLab Remote Procedure Call Framework (MLRPCFW) 17
3.1.3 Nutzung von Bindings 18
3.2 Das mPart API Framework 19
3.2.1 Client-Modell 19
3.2.2 Architektur des mPart-API-Frameworks 21

3.2.3 Asynchrone Kommunikation mittels Handler 23
3.2.4 Privacy Modes 23

4 Das mPart-RPC-Binding für MLRPCFW 25
4.1 MpartPeer 26
4.2 Full-Transport-Client 26
4.3 No-Transport-Client und Trusted Server 28

5 Prototypische Implementierung des Bindings 31
5.1 Request und Reply 31
5.1.1 Serialisierung von Objekten 32
5.2 MpartPeer 33
5.3 MpartRpcClient 34
5.4 MpartRpcServer 36
5.5 MrmService 37
5.6 Beispiel-Szenario Student and Professor Chat 38
5.6.1 Service-Interface SpcService 38
5.6.2 Umsetzung mit dem mPart-RPC-Binding 39
5.6.3 Ergebnisse des Testlaufs 40

6 Evaluation 41

7 Fazit und Ausblick 43

Abbildungsverzeichnis

1.1 Design Science Research Prozessmodell 4

2.1 Komponentenverteilung Client-Server 7
2.2 Beispielhafter Ablauf eines RPCs 10

3.1 MLRPCFW Kommunikationsstruktur 17
3.2 mPart-API-Framework Client-Modell 20
3.3 mPart-API-Framework Architektur-Modell 22

4.1 Konzept Full-Transport-Client im mPart-RPC-Binding 27
4.2 Beispielhafter Ablauf eines RPCs im mPart-RPC-Binding . . 28
4.3 Konzept No-Transport-Client im mPart-RPC-Binding 30

5.1 Datentyp MpartRpcInvocationRequest 32
5.2 Klasse MpartPeer 33
5.3 Klasse MpartRpcClient 34
5.4 Invocation-Handler im MpartRpcClient 35
5.5 Klasse MpartRpcServer 37
5.6 Service-Interface MrmService 37
5.7 Komponenten MrmService 38
5.8 Service-Interface SpcService 39

Kapitel 1

Anonymisierbare Kommunikation innerhalb verteilter Anwendungen

Ein Jahr nach der NSA-Affäre geben viele Nutzer immer noch sorglos Informationen im Internet preis. Zu diesem Schluss kommt die Gesellschaft für Informatik (GI) Mitte 2014 in einem Aufruf [fIe] an ihre Mitglieder. Sie fordert darin zu „risikobewusstem Verhalten und Datensparsamkeit“ auf. Die GI bezieht sich in ihrem Szenario hauptsächlich auf die Verschlüsselung des E-Mail-Verkehrs. Diese Verschlüsselung schützt den Inhalt der Nachricht vor dem Ausspähen durch Dritte, garantiert aber keine Anonymität gegenüber dem Kommunikationspartner. Anonymität ist notwendige Voraussetzung für soziale Beratungsangebote oder anonyme Selbsthilfegruppen, ebenso wie für Wahlen. Sollen derartige Anwendungen über Informationssysteme angeboten werden, muss die Kommunikation mittels dieser System anonymisierbar sein. Aus Sicht der Gesetzgebung ist Anonymität in der Nutzung von Informationssystemen allerdings keine Neuheit; beispielsweise wird in Deutschland eine grundsätzliche Datensparsamkeit durch ein Bundesgesetz geregelt. Das Bundesdatenschutzgesetz verbietet bereits in seiner ersten Fassung von 1977 die Erhebung von Daten grundsätzlich [Bun77, S. 203]. Nur bei gegebener rechtlicher Grund-

lage oder mit Zustimmung des Betroffenen ist die Datenerhebung zulässig. Dies gilt für öffentliche wie auch nicht-öffentliche Stellen. Somit ist die anonyme Nutzung von Informationssystemen juristisch gesehen eigentlich der Normalfall.

Dennoch nutzen Anwendungen in der Regel Computernetzwerke, welche nicht die Option bieten, die Kommunikation zu anonymisieren. Die jeweiligen Kommunikationspartner sind stets aufgrund ihre IP-Adresse identifizierbar. Komponenten zur anonymisierbaren Kommunikation von Anwendungen fehlen im Portfolio von Software-Frameworks. Somit ist die Erstellung von anonymisierbar kommunizierenden Anwendungen eine aufwendige Individuallösung. Diese Arbeit soll einen Beitrag dazu leisten, die Entwicklung anonymer Informationssysteme zu vereinfachen. Dazu soll anonymisierbare Kommunikation in ein Software-Framework für verteilte Anwendungen integriert werden. Diese kommen zum Einsatz, wenn Ressourcen verwendet werden, welche sich nicht auf dem eigenen System befinden [And03b, S. 20]. Mobile Anwendungen werden deshalb häufig auf der Basis verteilter Systeme entwickelt. Beispielsweise stellt Google als Hersteller des Mobil-Betriebssystems Android mit *Protocol RPC*[1] eine Lösung für verteilte System auf der Basis von Remote Procedure Calls (RPCs) zur Verfügung.

1.1 Ziel der Arbeit

Im Rahmen dieser Arbeit wird das Konzept des *Generischen Frameworks für die mobile Kommunikation von anonymisierten Gruppen* [Pat13] von Patrice Matthias Brend'amour mit dem im Rahmen des Forschungspraktikums *Mobiles Studentenportal 6* [Hak14] entwickelten MappLab Remote Procedure Call Framework (MLRPCFW) verknüpft. Das bereits vorhandene RPC-Framework soll derart modifiziert werden, dass die Kommunikation innerhalb verteilter Anwendungen auch über anonymisierte Netzwerke erfolgen kann. Zur Abwicklung anonymisierter Kommunikation

[1] `https://code.google.com/p/google-protorpc/`

soll das mPart-API-Framework von Brend'amour verwendet werden. Um das mPart-API-Framework in MLRPCFW zu integrieren, soll ein sogenanntes *Binding* entwickelt werden, über welches verteilte Anwendungen anonymisiert kommunizieren können. Die Integration umfasst das Übernehmen von Paradigmen und Programmiermustern von MLRPCFW. Die entwickelten Komponenten sollen in ihrer Handhabung bereits vorhandenen Bindings entsprechen, so dass sie sich in die Architektur von MLRPCFW einfügen. Beispielsweise sollen bereits vorhandene Dienste mit geringem Aufwand auf das vorhandene Binding umgestellt werden können. Auf diese Weise können bereits existierende Anwendungen mit geringem Aufwand anonymisierte Kommunikation nutzen.

Beide Frameworks liegen in einer prototypischen Implementierung vor. Der Prototyp des mPart-API-Frameworks bietet derzeit selbst nur eine Unterstützung für Java-basierte Plattformen, da die verwendete Bibliothek zur Realisierung des I2P-Protokoll-Moduls nur in Java vorliegt. Eine eigene Implementierung von anonymisierten Netzwerken und diesbezügliche Erweiterung des mPart-API-Frameworks wird im Rahmen dieser Arbeit nicht angestrebt, da sie als zu speziell und aufwändig im Hinblick auf die Zielsetzung der Arbeit angesehen wird. Aus diesem Grund soll auch die Realisierung des Konzepts für alle von MLRPCFW unterstützten Plattformen vernachlässigt werden. Entsprechend wird die prototypische Implementierung nur in Java erfolgen, das Konzept wird jedoch plattformunabhängig entworfen und kann somit nachträglich auch auf weiteren Plattformen realisiert werden.

1.2 Beispiel-Szenario

Um die praktische Nutzung der entwickelten Framework-Modifikation zu erproben, wird ein Beispiel-Szenario mit der prototypischen Implementierung umgesetzt. In diesem Szenario, dem *Student and Professor Chat* kommunizieren Studierende über eine verteilte Anwendung mit Lehrkräften. Die Lehrkräfte registrieren ein Pseudonym, an welches ihnen die Stu-

dierenden anonym Nachrichten schicken können. Die Lehrkräfte können dem anonymen Teilnehmer antworten, ohne dabei zu erfahren, wer die ursprüngliche Nachricht verschickt hat.

1.3 Methode: Design Science Research

Die vorliegende Arbeit verwendet zur Durchführung die Methodik des *Design Science Research* von Vaishnavi und Kuechler [Vij04]. Das Prozessmodell der Methodik, welches in Abbildung 1.1 vereinfacht dargestellt ist, gliedert sich in fünf Schritte. Im ersten Schritt *Awareness of Problem* wird

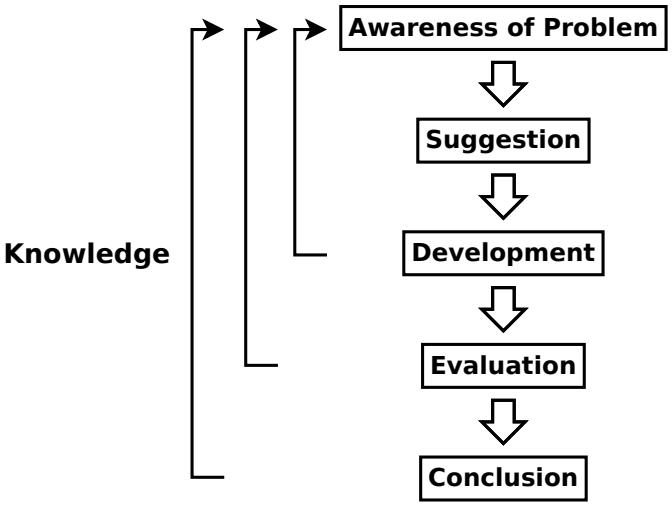

Abbildung 1.1: Prozessmodell des Design Science Research nach [Vij04]

aus dem Bewusstsein für ein Problem ein Forschungsvorhaben formuliert. Das Vorhaben dieser Arbeit ist die Entwicklung eines Frameworks für verteilte Anwendungen, welches für die interne Kommunikation anonymisierte Netzwerke verwendet. Im zweiten Schritt *Suggestion* wird ein vorläufiger Vorschlag entwickelt, wie das Problem zu lösen sei, das *Tentative Design*. In dieser Arbeit wird dazu ein Konzept für die Integration der verwendeten Frameworks entworfen. Während dem *Development*, dem dritten Schritt, wird ein Artefakt aus dem Tentative Design entwickelt. Die Entwicklung einer prototypischen Implementierung stellt ein

Artefakt dieser Arbeit dar. Im vierten Schritt wird das zuvor entwickelte Artefakt evaluiert. Besonders eingegangen wird dabei auf Abweichungen des Artefakts von Erwartungen, welche im Forschungsvorhaben formuliert wurden. In einer *Conclusion* werden im fünften Schritt die Fortschritte, die im Prozess gemacht wurden, dokumentiert und der Beitrag zum Stand der Forschung herausgearbeitet. Die fünf Schritte des Prozessmodells bilden einen fortwährenden Zyklus, da aus der Durchführung der einzelnen Schritten jeweils wieder Wissen gewonnen wird. Dies ist in der Abbildung durch die mit *Knowledge* beschrifteten Pfeile verdeutlicht. Die vorliegende Arbeit basiert auf der einmaligen Durchführung des Prozesszyklus und endet nach der ersten Conclusion.

1.4 Aufbau der Arbeit

Die Arbeit ist wie folgt gegliedert: In Kapitel 2 erfolgt eine Einführung in die Grundlagen von verteilten Systemen und verteilten Anwendungen, mit Schwerpunkt auf der Realisierung solcher Systeme mittels entfernter Prozeduraufrufe. Des Weiteren wird in diesem Kapitel der Einsatz von Frameworks in der Software-Entwicklung kurz erläutert. Das Kapitel schließt mit einer Definition von Anonymitätsbegriffen. Kapitel 3 gibt eine Einführung in die in dieser Arbeit verwendeten Frameworks und stellt die zugrunde liegenden Konzepte vor. Im Anschluss daran wird in Kapitel 4 ein Entwurf für die Integration beider Frameworks entwickelt. Die praktische Erprobung in Form einer prototypischen Implementierung wird in Kapitel 5 beschrieben. Dabei wird auch auf die Umsetzung des Beispielszenarios eingegangen. Kapitel 6 bewertet abschließend die Ergebnisse der Arbeit. In Kapitel 7 werden diese Ergebnisse zusammengefasst und ein Ausblick auf künftige Entwicklungen gegeben.

Kapitel 2

Verteilte Systeme, Frameworks und Anonymität

2.1 Verteilte Systeme und Client-Server-Modell

Als *Verteiltes System* bezeichnet Günther Bengel ein System, in welchem „Funktionseinheiten, welche über ein Transportsystem verbunden sind, in Zusammenarbeit Anwendungen bewältigen“ [Gün02, S. 5]. Eine auf einem solchen System laufende Anwendung bezeichnet er entsprechend als *verteilte Anwendung*. Die Leistung einer verteilten Anwendung wird kooperativ erbracht, indem bestimmte Ressourcen einer Funktionseinheit für andere als Dienste, genannt *Services*, zur Verfügung gestellt werden. Bengel nennt zwei Rollen für Funktionseinheiten: *Server*, welche einen Service anbieten und *Clients*, welche ein solches Angebot in Anspruch nehmen können [Gün02, S. 4, S. 28]. Tanenbaum und van Steen skizzieren zwei Haupteigenschaften verteilter Systeme: Zum einen sind die Funktionseinheiten des Systems jeweils *autonom*, funktionieren also als Teilsysteme zunächst unabhängig vom Gesamtsystem. Zum anderen ist es für Benutzer und Anwendungen nicht ersichtlich, dass „ihre Prozesse und Ressourcen physisch über mehrere Computer verteilt sind“[And03b, S. 21]. Ein System, welches diese Abstraktion ermöglicht, nennt man *transparent* [And03b, S. 21].

2.1.1 Zerlegung in Komponenten

Um eine Anwendung auf einem verteilten System auszuführen, muss die Anwendung in Komponenten beziehungsweise Schichten zerlegt werden, welche dann von Servern als Service angeboten werden. Bengel teilt Anwendungen in die fünf Komponenten [Gün02, S. 18] grafische Präsentation, Benutzerinterface, Verarbeitung, Datenmanagement und Datenspeicherung auf. Die Komponenten oder Schichten einer verteilten Anwendung werden auf einem Client und einem oder mehreren Server verteilt ausgeführt. Zu verarbeitende Daten werden dabei zwischen den beteiligten Einheiten über ein Transportsystem, beispielsweise ein Computernetzwerk, ausgetauscht. Links in Abbildung 2.1 ist das Konzept des *Null Clients* zu sehen, welcher nur die grafische Präsentation erledigt [Gün02, S. 19]. Alle weiteren Komponenten sind auf einen Server ausgelagert. Bei einem sogenannten *Thin Client* wird zusätzlich die Logik des Benutzerinterfaces auf dem Client ausgeführt [Gün02, S. 19]. Ein *Fat Client* hingegen ist

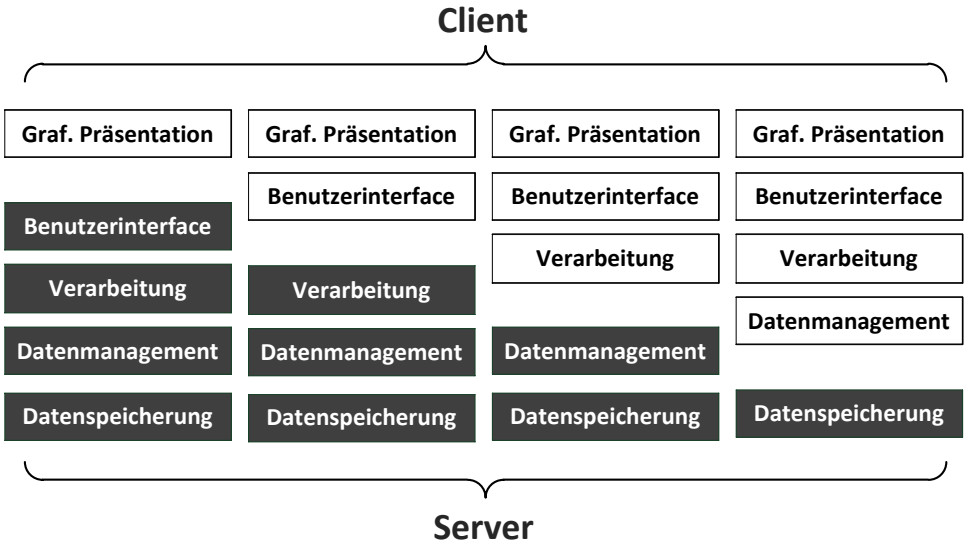

Abbildung 2.1: Komponentenverteilung auf einen Client und einen Server nach [Gün02]

in der Verarbeitungs-Komponente aufgetrennt [Gün02, S. 19]. Beispielsweise umfasst er alle Komponenten, bis auf das Datenmanagement und

die Datenspeicherung. Tanenbaum und van Steen nehmen eine ähnliche Aufteilung vor, fassen aber die Schichten für die Benutzeroberfläche und die Datenebene zusammen [And03b, S. 67 ff.].

Sowohl Bengel als auch Tanenbaum und van Steen unterscheiden ein zweistufiges Client-Server-Modell von einem mehrstufigen Modell [Gün02, S. 18f.] [And03b, S. 70ff.]. Im zweistufigen Client-Server-Modell wird die Anwendung lediglich an einer Stelle aufgetrennt. Beispielsweise liegen die Komponenten für die grafische Präsentation und das Benutzerinterface auf dem Client. Die restlichen Komponenten für die Verarbeitung und Datenmanagement und -speicherung befinden sich auf dem Server. Demgegenüber werden die Komponenten bei einer mehrstufigen Anwendung auf einen Client und mehrere Server aufgeteilt. Als Beispiel nennt Bengel eine dreiteilige Gliederung einer verteilten Anwendung. Auf dem Client befinden sich dabei die Komponenten für grafische Präsentation und das Benutzerinterface, auf einem Applikationsserver die Komponente für die Verarbeitung und auf einem Datenserver die Komponenten für Datenmanagement und -speicherung. Der Applikationsserver agiert in diesem Beispiel selbst als Client gegenüber dem Datenserver.

2.1.2 Kommunikation zwischen Client und Server

Client und Server stehen in einer Konsumenten-Produzenten-Beziehung. Der Client konsumiert die vom Server erbrachten Services, um seine Aufgaben zu erfüllen. Dabei folgt ihre Kommunikation einem festen Schema [Gün02, S. 29][And03b, S. 62]: Der Client fragt die Ausführung eines Service beim Server an. Diese Anfrage, genannt *Request*, wird vom Server bearbeitet. Nach Fertigstellung der Bearbeitung sendet der Server eine Antwort mit dem Ergebnis der Bearbeitung, genannt *Reply*, an den Client.

Die Kommunikation zwischen Client und Server kann sowohl blockierend, als auch nicht blockierend erfolgen [Gün02, S. 30f.]. Bei der blockierenden oder *synchronen* Kommunikation wartet der Client mit der Fortführung seiner Arbeit, bis die Antwort des Servers eingetroffen ist. Im Fall der nicht blockierenden, *asynchronen* Kommunikation führt der Client

seine Arbeit fort, nachdem die Anfrage an den Server geschickt wurde. Hierbei müssen jedoch Kontrollmechanismen eingeführt werden, welche Vorgänge im Client verzögern, die von der Antwort des Servers abhängen. Erst wenn die Antwort des Server eintrifft, können diese Vorgänge ausgeführt werden.

Unabhängig von ihrer Synchronität, kann die Kommunikation von Client und Server auch hinsichtlich ihrer Persistenz [And03b, S. 122ff.] betrachtet werden. Bleiben Anfragen des Clients beziehungsweise Antworten des Servers solange gespeichert, bis sie an die Gegenstelle ausgeliefert wurden, wird die Kommunikation als nicht flüchtig oder auch *persistent* bezeichnet. Werden Anfragen oder Antworten verworfen, falls die Gegenstelle nicht erreichbar ist oder gerade nicht ausgeführt wird, ist die Kommunikation flüchtig oder auch *transient*.

Zur Kommunikation von Clients und Servern in einem verteilten System können verschiedene Modelle genutzt werden, welche sich hinsichtlich ihrer Synchronität und Persistenz unterscheiden. RPCs erlauben einem Prozess, ein Programm auf einem anderen Teilsystem aufzurufen [And03b, S. 90ff.]. Die Kommunikation ist dabei in der Regel transient. Als Weiterentwicklung dieses Konzepts erlauben *verteilte Objekte* statt des Aufrufs von Prozeduren die Benutzung von Objekten, welche sich auf einem anderen Teilsystem befinden [And03b, S. 107ff.]. Sie werden daher in objekt-orientierten Programmiersprachen eingesetzt. Ein Beispiel ist Remote Method Invocation (RMI) in Java [Gün02, S. 144]. Komplexere Modelle ermöglichen persistente asynchrone Kommunikation, indem das Transportsystem die Anfragen und Antworten speichert, bis die Gegenstelle wieder erreichbar ist [And03b, S. 122ff.]. Da in dieser Arbeit die Realisierung von RPCs mittels anonymisierbarer Kommunikation betrachtet werden soll, wird dieses Methode im Folgenden näher erläutert.

2.2 Realisierung verteilter Systeme mittels RPC

Werden verteilter Systeme durch RPCs realisiert, kommunizieren die Komponenten der verteilten Anwendung durch Prozeduraufrufe. Der entfernte Prozeduraufruf soll dabei möglichst genauso aussehen, wie ein lokaler Prozeduraufruf. Der RPC ist transparent, wie in Abschnitt 2.1 beschrieben. Um dies zu erreichen muss eine lokale Version der entfernten Prozedur auf dem Client vorhanden sein, die stellvertretend für die entfernte Version aufgerufen wird. Da sie nicht die Funktionalität der entfernten Prozedur aufweist, wird sie *Client-Stub* genannt [And03b, S. 92]. In Abbildung 2.2 ist der Ablauf eines RPCs beispielhaft dargestellt. Der Client-Stub

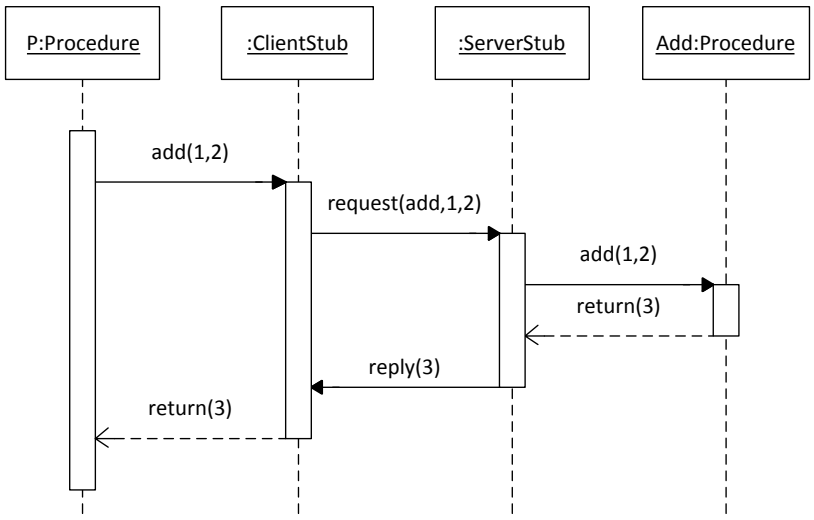

Abbildung 2.2: Beispielhafter Ablauf eines RPCs

nimmt die beim Aufruf übergebenen Parameter der Prozedur entgegen, verpackt sie in einen Request und sendet diese an den Server. Zusätzlich enthält dieser Request eine Kennung der Prozedur, welche auf dem Server ausgeführt werden soll. Analog zum Client-Stub existiert auf dem Server ein *Server-Stub* [And03b, S. 92]. Der Server-Stub erhält den Request und führt die Prozedur stellvertretend für den Client aus. Nach Beendigung der Prozedur verpackt der Server-Stub das Ergebnis in eine Reply und schickt sie an den Client zurück. Der Client-Stub empfängt die Antwort

und gibt das Ergebnis an die aufrufende Anwendung zurück, als wäre die Prozedur lokal ausgeführt worden.

Im oben beschriebenen Fall wartet der Client auf die Antwort des Server, bevor er mit der Ausführung der Anwendung fortfährt. Dies kann aus unterschiedlichen Gründen von Nachteil sein. Beispielsweise könnte die Benutzeroberfläche des Clients blockieren, so dass der Anwender keine weiteren Eingaben tätigen kann. Die Kommunikation von Client und Server kann alternativ auch asynchron ablaufen [And03b, S. 100]. Diese Variante ist dann geeignet, wenn die Antwort des Servers für das Fortführen der Anwendung auf dem Client nicht erforderlich ist oder der Client währenddessen andere Aufgaben erledigen kann. Tanenbaum und van Steen nennen als Beispiel das Einfügen von Einträgen in eine Datenbank [And03b, S. 100].

2.2.1 Vereinbarungen zwischen Client und Server

Um die zum Aufruf der Prozedur notwendigen Parameter an den Server zu verschicken, müssen diese zuvor in ein Format überführt werden, welches für den Versand geeignet ist. Dieser Vorgang wird als *Marshalling* bezeichnet [And03b, S. 94]. Werden statt einfacher Werte ganze Objekte für den Versand kodiert, wird dies als *Serialisierung* bezeichnet [Gün02, S. 152]. Die Objekte werden dazu vom Client in einen Byte-Strom umgewandelt und auf der Server-Seite wieder zu einem Objekt rekonstruiert. Sowohl beim Marshaling als auch bei der Serialisierung muss zuvor präzise zwischen Client und Server vereinbart worden sein, in welchem Format die Parameter in der Anfrage beziehungsweise Antwort zu verpacken sind. Aufgrund unterschiedlicher Hardware, Betriebssysteme oder auch Programmiersprachen auf dem Client und dem Server kann sich das Format unterscheiden, in dem die Daten kodiert sind.

Des Weiteren müssen Client und Server vorab vereinbart haben, welche Prozeduren aufgerufen werden können, welche Parameter diese zur Ausführung benötigen und welches Art Ergebnis zu erwarten ist. Diese Vereinbarung geschieht mittels einer *Schnittstelle*, auch *Interface* genannt

[And03b, S. 98]. Eine solche Schnittstelle definiert die Prozeduren, die vom Client aufgerufen werden können und vom Server zu implementieren sind. Während objekt-orientierte Programmiersprachen wie Java oder PHP Konzepte zur Definition von Interfaces bereitstellen, verwenden verteilte Systeme zum Teil auch spezielle Beschreibungssprachen für Interfaces. Zum Beispiel gibt CORBA eine eigene *Interface Description Language (IDL)* vor [Gün02, S. 159].

2.3 Zweck eines Software-Frameworks

Andreas Rüping definiert die Aufgabe eines *Frameworks* darin, einen Bauplan zur Konstruktion von Anwendungssystemen festzulegen [And97, S. 22]. Dazu werden Modelle für eine Menge von Anwendungssystemen beschrieben und eine gemeinsame Architektur festgelegt. Die Architektur besteht aus überwiegend abstrakten Komponenten, welche für die einzelne Anwendung vom Entwickler konkretisiert werden. Bei objekt-orientierter Entwicklung handelt es sich bei den abstrakten Komponenten des Frameworks um *abstrakte Klassen* [Wie95, S. 61]. Eine abstrakte Klasse fasst Eigenschaften von ähnlichen Klassen zusammen und dient damit als Oberklasse [Wol97, S. 26]. Mittels abstrakter Klassen lassen sich *abstrakte Methoden* definieren, das heißt Platzhalter für eine Methode oder eine vorläufige Implementierung einer Methode. Dadurch schafft die abstrakte Klasse eine gemeinsame Schnittstelle, da alle abgeleiteten, konkreten Klassen diese Methode ebenfalls zu Verfügung stellen müssen [Wol97, S. 26]. Zusätzlich kann ein Framework auch eine Bibliothek fertig implementierter, konkreter Komponenten zur Verfügung stellen, welche von den abstrakten Komponenten der Architektur abgeleitet sind [And97, S. 22]. Die Benutzung eines Frameworks unterscheidet sich hier von der reinen Wiederverwendung vorhandener Software-Bausteine, indem das Framework die Systemarchitektur der Anwendung vorgibt, wogegen der Entwickler diese sonst selbst definiert [And97, S. 11]. Außerdem beschreibt ein Framework, wie sich konkrete Anwendungssysteme aus der abstrakten Archi-

tektur herleiten lassen. Wird auf diese Weise ein konkretes System erstellt, bezeichnet Rüping dies als *Instantiierung* des Frameworks [And97, S. 11]. Scherp und Boll beschreiben die Umkehrung des Kontrollflusses [Ans09, S. 384] als weiteres Merkmal von Software-Frameworks. Der Hauptkontrollfluss der Anwendung wird durch das Framework gesteuert. Der Entwickler definiert die für seine Anwendung spezifischen Komponenten, die an geeigneter Stelle vom Framework aufgerufen werden.

Durch die Abstraktion von Komponenten in Frameworks können Software-Bausteine und Architekturentwürfe von Anwendungen wiederverwendet werden [Wol97, S. 8]. Diejenigen Komponenten und Konzepte, die in mehreren Anwendungen zum Einsatz kommen, werden durch das Framework vorgegeben und sind somit Bestandteil aller Anwendungen [And97, S. 22]. Wolfgang Pree nennt als Vorteile der Wiederverwendung neben geringeren Kosten durch Entwicklung und Wartung, auch höhere Effizienz und Softwarequalität [Wol97, S. 2]. Pree begründet dies dadurch, dass mehrfach verwendete Software besser getestet sei und auf effiziente Software-Bausteine zurückgegriffen werden könne.

2.4 Arten von Anonymität

Johann Bizer eröffnete einen Vortrag zum „Recht auf Anonymität“[Joh00] mit der Feststellung, dass es mindestens zwei intuitive Definitionen des Begriffs *Anonymität* gäbe: Zum einen könne man Anonymität verstehen als Anonymität gegenüber dem Kommunikationspartner. Zum anderen jedoch könne auch der Wunsch bestehen, die Kommunikationsbeziehung an sich gegenüber Dritten geheim zu halten. Andreas Pfitzmann und Marit Hansen definieren in ihrem Artikel [And10] eine Terminologie im Bereich Anonymität mit dem Zweck derartige Mehrdeutigkeiten zu vermeiden. Sie unterscheiden die fünf Begriffe Pseudonymität, Anonymität, Unverkettbarkeit, Unentdeckbarkeit und Unobservierbarkeit, welche auch im Laufe dieser Arbeit zur Beschreibung von Anonymitätskonzepten verwendet werden. Sie erläutern die Begriffe an einem Modell von kommunizie-

renden Einheiten. In diesem Modell tauschen sich handelnde Einheiten, genannt *Subjekte*, mittels Nachrichten, genannt *Objekte*, aus. Darüber hinaus beschreibt das Modell *Aktionen*, welche Subjekte mit Objekten durchführen. Ein Beispiel hierfür wäre das Absenden einer Nachricht. Die Beschreibung der einzelnen Szenarien des Modells erfolgt aus Sicht eines Angreifers. Dieser versucht von außerhalb die Kommunikation zu beobachten, Muster zu erkennen oder die Kommunikation sogar zu manipulieren. Der Angreifer nutzt alle im zur Verfügung stehenden Möglichkeiten, um für ihn interessante Elemente, sogenannte *Items of Interest*, aus der Beobachtung zu schlussfolgern. Ein solches Element könnte beispielsweise der Absender einer Nachricht sein.

Unter *Pseudonymität* verstehen Pfitzmann und Hansen den Einsatz eines Decknamens statt des realen Namens eines Subjekts. Sie verweisen darauf, dass eine solche Kennung für eine Kommunikation, die beidseitig erfolgen soll, unbedingt notwendig ist. Der Empfänger einer Nachricht kann ansonsten unmöglich wissen, an wen er seine Antwort adressieren soll.

Anonymität bedeutet im Sinne von Pfitzmann und Hansen, dass ein Subjekt in einer Gruppe von Subjekten nicht durch einen Angreifer identifiziert werden kann. Der Angreifer kann also feststellen, dass jemand innerhalb dieser *Anonymitätsgruppe* die Kommunikation durchgeführt hat, allerdings ist es ihm nicht ohne weitere Informationen möglich, das einzelne Subjekt zu bestimmen. Dies wird ermöglicht, da alle Subjekte sich eine gemeinsame Menge von Eigenschaften teilen, die es dem Angreifer erschweren, sie zu unterscheiden.

Mit dem Begriff der *Unverkettbarkeit* beschreiben Pfitzmann und Hansen, dass ein Angreifer nicht erkennen kann, ob eine Beziehung zwischen zwei oder mehr Items of Interest besteht. Ist Unverkettbarkeit gegeben, dann ist der Kenntnisstand über den Zusammenhang zweier Items of Interest vor und nach der Beobachtung des Angreifers der gleiche. Der Angreifer gewinnt also durch die Beobachtung keine zusätzlichen Informationen.

Die *Unentdeckbarkeit* einer Kommunikation macht es einem Angreifer

unmöglich herauszufinden, ob überhaupt kommuniziert wurde. Im Modell von Pfitzmann und Hansen ist es dem Angreifer nicht möglich, herauszufinden, ob ein Item of Interest existiert oder nicht.

Der Begriff der *Unobservierbarkeit* kombiniert die Anforderungen der Anonymität nach Pfitzmann und Hansen mit denen der Unentdeckbarkeit. Ein Angreifer kann nicht feststellen, ob überhaupt kommuniziert wurde und die an einer Kommunikation beteiligten Subjekte können sich auch gegenseitig nicht identifizieren.

Kapitel 3

Vorhandene Konzepte für RPC und anonymisierte Netzwerke

3.1 Das MAppLab RPC Framework

Innerhalb des Forschungspraktikums „Mobiles Studentenportal 6“ wurde mit dem MLRPCFW ein Framework für die Entwicklung verteilter Anwendungen entwickelt. Das Framework gibt eine Softwarearchitektur für verteilte Anwendungen auf der Basis von RPCs vor. Darüber hinaus stellt es eine Bibliothek abstrakter, sowie auch fertig implementierter Komponenten in den Programmiersprachen C++, Java und PHP zur Verfügung. Der in Kapitel 2.3 vorgestellte Framework-Begriff ist daher auf MLRPCFW anwendbar. Im Folgenden wird bis einschließlich Abschnitt 3.1.3 die Ausarbeitung [Hak14] des Forschungspraktikums als Quelle verwendet.

3.1.1 Gemeinsames Service-Interface

Zentrales Konzept des Frameworks ist das Anbieten und Nutzen von *Services* in verteilten Systemen. Zur Beschreibung eines Service wird ein *Service-Interface* definiert, welches alle Prozeduren auflistet, die genutzt werden können. Bei der Ableitung von konkreten Komponenten aus den abstrakten Vorlagen des Frameworks müssen die Prozeduren in jede Komponente implementiert werden. Dadurch erfolgt die Instantiierung des

Frameworks in einer konkreten verteilten Anwendung. Obwohl alle Komponenten das Service-Interface implementieren, stellt nur eine Komponente die tatsächliche Funktionalität zur Verfügung. Diese wird als *Provider* bezeichnet. Eine *Consumer* genannte Komponente stellt die im Service-Interface definierten Prozeduren nach außen zur Verfügung und reicht dazu die Anfragen an den Provider weiter. Die Kommunikation zwischen Consumer und Provider wird mittels RPC abgewickelt.

3.1.2 Architektur von MLRPCFW

Zur Realisierung von Services gibt die Softwarearchitektur von MLRPCFW ein Kommunikationsstruktur-Modell für verteilte Anwendungen vor, welches in Abbildung 3.1 dargestellt ist. Das Modell entspricht einem zweistufigen Client-Server-Modell, wie es in Kapitel 2.1 beschrieben ist. Auf

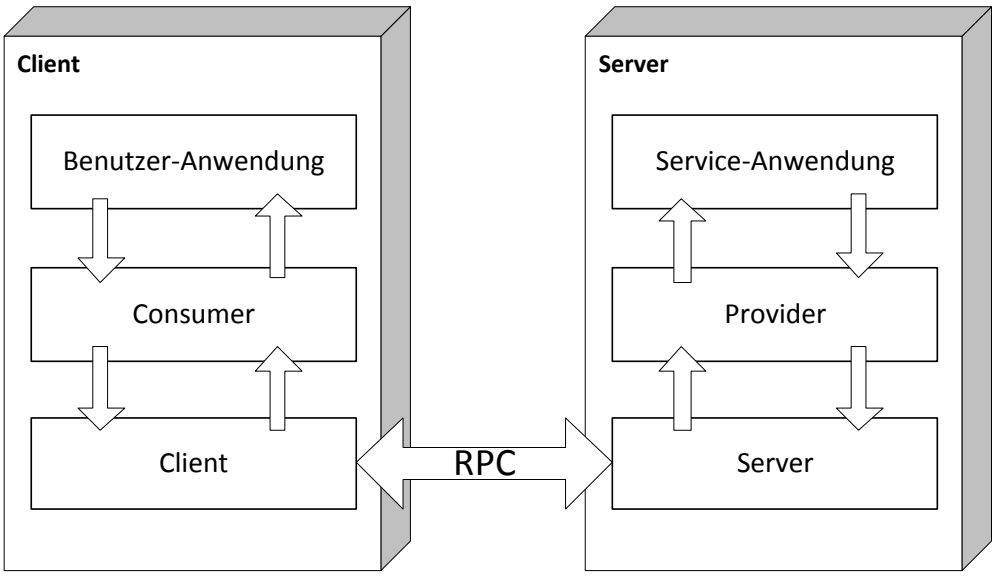

Abbildung 3.1: Kommunikationsstruktur von MLRPCFW nach [Hak14]

oberster Ebene ist jede Anwendung in eine Benutzer-Anwendung und eine Service-Anwendung aufgeteilt. Die Benutzer-Anwendung stellt die Schnittstelle zum Benutzer des Systems dar, beispielsweise über eine Benutzeroberfläche. Die Service-Anwendung kann eine Datenbasis sein oder

eine beliebige andere Anwendung oder Ressource, welche als Service angeboten werden soll.

Wie in Abschnitt 2.1.2 beschrieben, stehen Client und Server in einer Konsumenten-Produzenten-Beziehung zueinander. In der Architektur von MLRPCFW wird zwischen Produzenten und Konsumenten der Daten und den mittels RPC kommunizierenden Stubs unterschieden. Die Benutzer-Anwendung greift auf einen Consumer zurück, um einen Service in Anspruch zu nehmen. Dieser Consumer konsumiert die Daten eines Providers. Da sich der Provider jedoch auf einem anderen Teilsystem befindet, nutzt der Consumer einen sogenannten *Client* zur Kommunikation. Dabei handelt es sich um einen Client-Stub im Sinne von Kapitel 2.2, welcher mittels RPC eine Anfrage an einen Server-Stub sendet. Diesen Server-Stub bezeichnet MLRPCFW als *Server*. Der Server-Stub ruft die angefragte Prozedur nun in einem Provider auf, welcher zur Durchführung der Prozedur auf eine Service-Anwendung zurückgreifen kann. Die Instantiierung dieser Architektur in einer konkreten verteilten Anwendung erfolgt durch das Ableiten konkreter Komponenten aus den vier abstrakten Komponenten Consumer, Provider, Client und Server. Die Architektur der Benutzer- und der Service-Anwendung werden durch das Framework nicht vorgegeben.

3.1.3 Nutzung von Bindings

Während Consumer und Provider die Verteilung eines Service logisch umsetzen, dienen die abstrakten Komponenten Client und Server als Stellvertreter für eine technische Umsetzung der Kommunikation im verteilten System. MLRPCFW erlaubt es, konkrete RPC-Implementierungen in dieses allgemeine Konzept einzubinden. Dazu muss je eine spezifische Client- und Server-Komponente von den abstrakten Komponenten abgeleitet werden. Diese neuen Komponenten bilden nun ein *Binding*, da ein Consumer oder Provider mit Verwendung eines spezifischen Clients oder Servers an eine bestimmte Art der Kommunikation gebunden wird. Die Bibliothek von MLRPCFW enthält eine Client- und Server-Umsetzung

der RPC-Implementierung Perfect High Performance Remote Procedure Call (PHPRPC)[1]. Die Client-Umsetzung von PHPRPC liegt für mehrere Plattformen vor, unter anderem auch Java. Die Server-Umsetzung liegt derzeit nur in PHP vor.

3.2 Das mPart API Framework

Damit Client und Server in einem verteilten System anonym kommunizieren können, müssen entsprechende Anonymisierungstechniken eingesetzt werden. Eine Möglichkeit ist die Nutzung eines Netzwerks, welches die Datenübertragung anonymisiert. Patrice Brend'amour hat ein generisches Framework für die Nutzung solcher *anonymisierter Netzwerke* geschaffen, das *mPart-API-Framework* [Pat13]. Dieses stellt eine generische Schnittstelle für verschiedene anonymisierte Netzwerke zur Verfügung, über welche Anwendungsprogramme Nachrichten austauschen können. Sie ist insofern generisch, als sie trotz Einbindung unterschiedlicher Netzwerke in das Framework gegenüber der Anwendung stets die gleichen Methoden bereitstellt. Des Weiteren ist sie unabhängig von der verwendeten Plattform oder dem Einsatzszenario [Pat13, S. 21f]. Aufgaben des Frameworks umfassen die Verwaltung von Verbindungen zu anonymisierten Netzwerken, das Versenden und Empfangen von Nachrichten über diese Verbindungen und das Authentifizieren und Autorisieren von Teilnehmern eines Netzwerks [Pat13, S. 26].

3.2.1 Client-Modell

Das mPart-API-Framework unterscheidet drei verschiedene Arten von Clients [Pat13, S. 22ff.], welche im Modell in Abbildung 3.2 veranschaulicht sind. Ein *Client* im Sinne des Modells ist ein Teilnehmer an einem anonymen Netzwerk, welcher in diesem Netzwerk Nachrichten an andere Teilnehmer versenden kann. Ebenso kann der Client Nachrichten anderer

[1] `http://phprpc.org/en/`

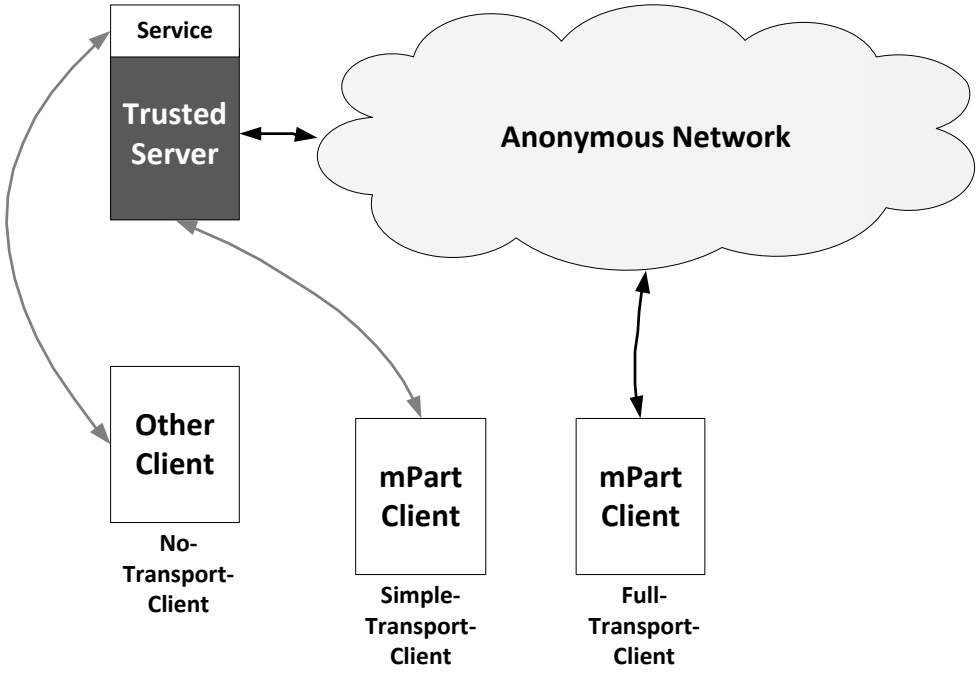

Abbildung 3.2: Client-Modell des mPart-API-Frameworks nach [Pat13]

Teilnehmer empfangen. Im Normalfall verwendet der Client eine vollständige Implementierung des mPart-API-Frameworks und mindestens eines anonymisierten Netzwerks. Der sogenannte *Full-Transport-Client* kommuniziert dann mit Hilfe des Frameworks direkt über ein anonymisiertes Netzwerk.

Alternativ zur vollständigen Implementierung auf dem Client können Teile des Frameworks zu einem vertrauenswürdigen Dritten ausgelagert werden. Sie befinden sich dann auf einem sogenannten *Trusted Server*. Beim zweiten Client-Typ, dem *Simple-Transport-Client* wird die Teilnahme am anonymisierten Netzwerk auf den Trusted Server ausgelagert. Der Client verwendet weiterhin das Framework, kommuniziert jedoch über eine nicht-anonymisierte Verbindung mit dem Trusted Server. Dieser nimmt stellvertretend für den Client am Netzwerk teil.

Bei einem *No-Transport-Client* ist das gesamte Framework auf den Trusted Server ausgelagert. Die Anwendung auf dem Client verwendet einen

Service, den der Trusted Server anbietet, um am Netzwerk teilzunehmen. Dabei wird statt der Schnittstelle des Frameworks beispielsweise eine Web-Schnittstelle genutzt. Da die Kommunikation zwischen Simple- beziehungsweise No-Transport-Client und Trusted Server nicht über ein anonymisiertes Netzwerk erfolgt, muss die genutzte Netzwerkverbindung, ebenso wie der Trusted Server, vertrauenswürdig sein. Ein Beispiel hierfür ist die Nutzung eines Servers in einem Unternehmensnetzwerk, welchem die Mitarbeiter vertrauen, um anonym mit Teilnehmern außerhalb des Unternehmens zu kommunizieren. Dieses Modell entstand aus der Problematik, dass Implementierungen von anonymisierten Netzwerken nicht auf allen Plattformen vorliegen oder der Einsatz zu aufwendig wäre [Pat13, S.24]. Ein Beispiel für letzten Fall stellen mobile Geräte dar, da sie in Bezug auf Rechenleistung oder Netzwerkanbindung gegebenenfalls eingeschränkt sind.

3.2.2 Architektur des mPart-API-Frameworks

Die Architektur des mPart-API-Frameworks besteht aus einer zentralen Kommunikationskomponente und weiteren Modulen. Dies ist in Abbildung 3.3 dargestellt. Die zentrale Komponente, der sogenannte *mPart Core*, stellt Anwendungen die gesamte Funktionalität des Frameworks als Methoden der Schnittstelle *mPart Core Interface* zur Verfügung. Der mPart Core delegiert Teile des Funktionsumfangs an Module. Dabei unterscheidet das Framework zwei Arten von Modulen: Auth-Module und Protokoll-Module. Ein Auth-Modul implementiert ein Verfahren, das der Authentifizierung beziehungsweise Autorisierung von Netzwerkteilnehmern dient [Pat13, S. 26]. Ein Protokoll-Modul wickelt die gesamte Kommunikation über ein anonymisiertes Netzwerk ab. Für jedes Verfahren beziehungsweise Netzwerk, das im mPart-API-Framework verwendet werden soll, muss ein eigenes Modul implementiert werden. Der Prototyp des Frameworks beinhaltet ein Protokoll-Modul für das anonymisierte Netzwerk *I2P*.

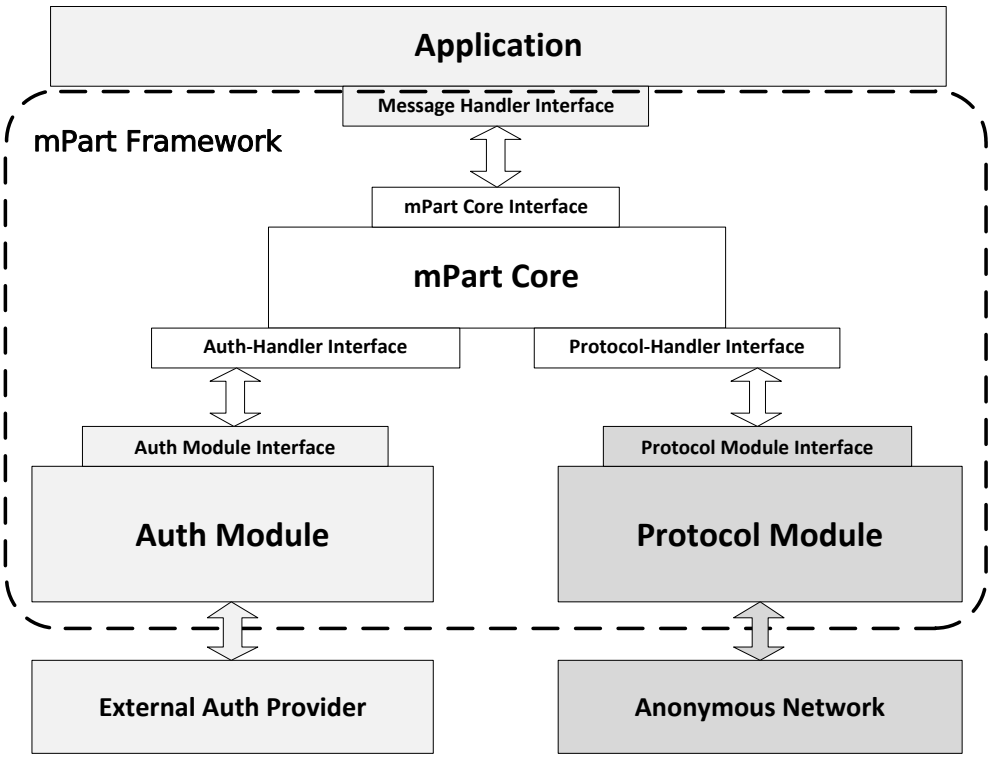

Abbildung 3.3: Architektur des mPart-API-Frameworks, nach [Pat13]

3.2.3 Asynchrone Kommunikation mittels Handler

Die Kommunikation verläuft sowohl zwischen dem mPart Core und den Modulen, als auch zwischen mPart Core und der Anwendung asynchron und nicht-blockierend. Deshalb existiert an den Schnittstellen jeweils ein Interface auf beiden Seiten und es findet bei der Initialisierung eine Registrierung statt, welche der mPart Core entgegen nimmt. Anwendungen, welche das Framework nutzen möchten, müssen beim mPart Core registriert werden. Ebenso muss jedes AuthModul beim mPart Core registriert werden. Möchte eine Anwendung eine Nachricht versenden, ruft sie die entsprechende Methode im *mPart Core Interface* auf. Der mPart Core verarbeitet diesen Aufruf und delegiert das Versenden der Nachricht an eines der bei ihm registrierten Protokoll-Module über dessen *Protocol Module Interface*. Weder die Anwendung, noch der mPart Core unterbrechen ihre Ausführung, um auf eine Antwort zu warten. Wird eine Nachricht über ein Protokoll-Modul empfangen, ruft das Modul das *Protocol Handler Interface* des mPart Core auf. Dieser verarbeitet die eingehende Nachricht und ruft seinerseits das *Message Handler Interface* auf, welches die Anwendung zuvor bei ihm registriert hat. Neben den beschriebenen Komponenten stellt das Framework Datentypen für Nachrichten, Empfänger-Adressen und Antwort-Adressen bereit. Diese Datentypen werden sowohl im Framework als auch an der Schnittstelle zur Anwendung genutzt. Die zur Umsetzung von Autorisierungen und Authentifizierungen notwendigen Auth-Module kommunizieren auf gleiche Weise mit dem mPart Core wie die Protokoll-Module. Da sie für die Problemstellung dieser Arbeit keine Rolle spielen, werden sie an dieser Stelle nicht näher betrachtet.

3.2.4 Privacy Modes

Beim Versenden von Nachrichten über das mPart-API-Framework kann zwischen zwei Varianten [Pat13, S. 30] gewählt werden. Beide Varianten des Versands von Nachrichten sind als Methoden im mPart Core Interface vorhanden und können durch die Anwendung genutzt werden. Zum

einen kann eine Verbindung zu einem bestimmten anonymisierten Netzwerk über ein Protokoll-Modul direkt gewählt werden. Der mPart Core delegiert dann das Versenden der Nachricht an das gewählte Protokoll-Modul. Zum anderen kann ein Modul vom mPart Core automatisch gewählt werden. Die Anwendung gibt dann keine Verbindung explizit an, sondern wählt eine von fünf Anonymitätsstufen. Diese Stufen werden als *Privacy Mode* bezeichnet. Sie orientieren sich an der in Kapitel 2.4 vorgestellten fünfstufigen Anonymitäts-Terminologie von Pfitzmann und Hansen. Der Privacy Mode, den die Anwendung angibt stellt die Mindestanforderung für den Versand dar. Der mPart Core wählt für den Versand aus den bei ihm registrierten Netzwerken eines aus, dass mindestens diesen Privacy Mode gewährleistet.

Kapitel 4

Das mPart-RPC-Binding für MLRPCFW

Das zu entwickelnde mPart-RPC-Binding stellt das Bindeglied zwischen den anonymisierten Nachrichten des mPart-API-Frameworks und der service-basierten Infrastruktur von MLRCPFW dar. Mit Hilfe des mPart-API-Frameworks können Nachrichten in anonymisierten Netzwerken versandt und empfangen werden. Die in Kapitel 1.1 entwickelte Anforderung an ein Binding für MLRPCFW ist demgegenüber allerdings die vollständige Abwicklung von RPCs über ein anonymisiertes Netzwerk. Betrachtet man die Kommunikation mittels RPCs beim MLRPCFW und die Kommunikation mittels einfacher Nachrichten beim mPart-API-Framework aus Sicht von Referenzmodellen für Computernetzwerke [And03a, S. 43ff.], so befindet sich die RPC-Kommunikation auf einer höheren Hierarchie-Ebene: Zur Abwicklung von RPCs wird auf die Kommunikation mittels einfacher Nachrichten zurückgegriffen. Das mPart-API-Framework wird folglich in einem Binding für MLRPCFW gekapselt, da die Vorgänge zum Initialisieren des Frameworks und zur Übermittlung der Nachrichten ebenso wie spezielle Datentypen für die darüber liegende Schicht nicht relevant sind. Dies entspricht beispielsweise der Vorgehensweise des OSI-Referenzmodells, eine neue Schicht einzuführen, wenn „ein neuer Abstraktionsgrad benötigt wird“ [And03a, S. 54]. Die nachfolgend vorgestell-

te Architektur des mPart-RPC-Bindings zeigt auf, wie die genannten Ziele konzeptuell erreicht werden sollen.

4.1 MpartPeer

Da Client-Stub und Server-Stub beide mittels des mPart-API-Frameworks kommunizieren, teilen sie sich die Funktionalität des Nachrichtenversands und Nachrichtenempfangs. Diese Funktionalität wird in der Komponente *MparPeer* gekapselt, so dass sie wiederverwendet werden kann. Die Komponente initialisiert das mPart-API-Framework und vermittelt zwischen diesem und dem Client- beziehungsweise Server-Stub. Sie initialisiert den mPart Core, registriert Protokoll-Module bei ihm und öffnet Verbindungen zu anonymisierten Netzwerken. Außerdem führt sie den Versand von Nachrichten über das mPart Core Interface mittels der Datentypen des mPart-API-Frameworks durch. Als Folge der starken Interaktion mit dem mPart-API-Framework ist die Komponente nur auf solchen Systemen verfügbar, auf denen auch dieses Framework verfügbar ist. Die sich daraus ergebende Problematik wird in Abschnitt 4.3 näher betrachtet.

4.2 Full-Transport-Client

Ist das mPart-API-Framework und somit auch der MpartPeer auf dem System verfügbar, können der Client- und der Server-Stub den Versand und Empfang von anonymisierten Nachrichten direkt an diese Komponente delegieren. In Anlehnung an die in Kapitel 3.2 vorgestellte Terminologie des mPart-API-Frameworks wird dieses Konzept als *Full-Transport-Client* bezeichnet. In Abbildung 4.1 ist die Verteilung der Komponenten dargestellt. Die Benutzer-Anwendung greift auf einen Dienst zu, welcher vom Consumer angeboten wird. Dieser nutzt zur Bereitstellung des Dienstes den *MpartRpcClient*, um einen entfernten Prozeduraufruf auf einem Server durchzuführen. Der MpartRpcClient verpackt den Prozeduraufruf in eine Anfrage, den *Request*, und delegiert den Versand an seinen Mpart-

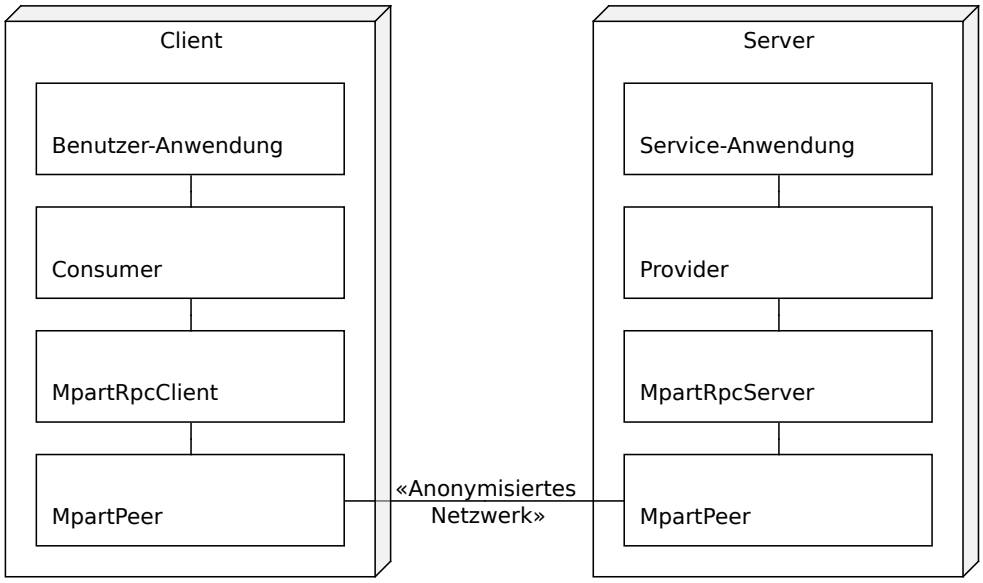

Abbildung 4.1: Konzept „Full-Transport-Client“ im mPart-RPC-Binding

Peer, welcher die Anfrage in eine Nachricht verpackt und über ein anonymisiertes Netzwerk versendet. Auf der Gegenseite nimmt ein MpartPeer die Nachricht entgegen und meldet seinem Server-Stub, dem *MpartRpcServer*, mittels eines Handlers, dass eine neue Nachricht eingegangen ist. Der MpartRpcServer nimmt die Nachricht entgegen und entpackt die darin enthaltene Anfrage. Mit den angegebenen Informationen ist es ihm möglich, die passende Prozedur im Provider aufzurufen und das Ergebnis auf umgekehrtem Weg zurück zum Consumer zu kommunizieren. Der Provider greift für die Ausführung der Prozedur gegebenenfalls auf eine Service-Anwendung, wie beispielsweise eine Datenbank, zurück. Dieses Vorgehen entspricht der von MLRPCFW vorgegebenen Kommunikationsstruktur, wie sie in Kapitel 3.1 beschrieben ist.

MLRPCFW benutzt innerhalb des Frameworks synchrone Aufrufe zum Austausch zwischen den Komponenten. Das mPart-API-Framework hingegen verwendet asynchrone Aufrufe mittels sogenannter *Handler*, wie in Kapitel 3.2.3 beschrieben. Zur Umsetzung eines synchron arbeitenden RPC-Bindings über den asynchronen Kommunikationskanal des mPart-

API-Frameworks muss der MpartRpcClient nach Absenden des Requests blockieren und warten, bis die Antwort des MpartRpcServer eingetroffen ist. Ein solcher Ablauf ist beispielhaft in Abbildung 4.2 abgebildet. Der

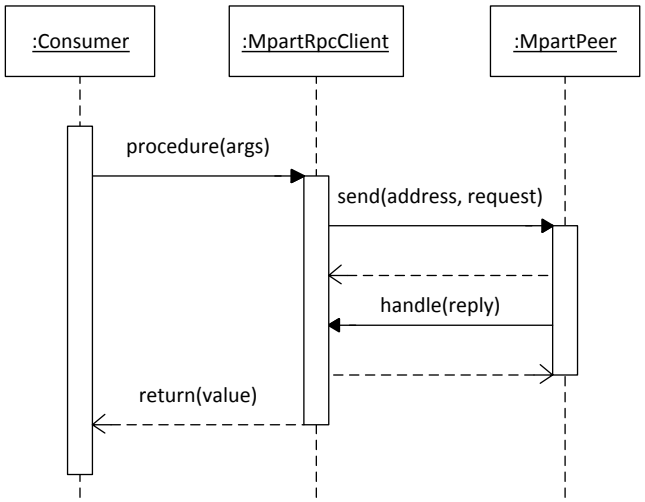

Abbildung 4.2: Beispielhafter Ablauf eines RPCs im mPart-RPC-Binding

Consumer ruft eine Prozedur im MpartRpcClient auf. Die Prozedur wurde in der Abbildung beispielhaft `procedure()` genannt. Anschließend wartet er, da er das Ergebnis benötigt, um es direkt oder nach Verarbeitung an die Benutzer-Anwendung zurückzugeben. Der MpartRpcClient delegiert den Versand der Nachricht durch die Prozedur `send()` an seinen MpartPeer, welcher diesen synchronen Aufruf bestätigt. Nun wartet der MpartRpcClient ebenfalls auf die Antwort des Servers. Sobald die Reply mittels der Prozedur `handle()` eingeht, kann er die Antwort des Servers an den Consumer zurückgeben.

4.3 No-Transport-Client und Trusted Server

Ist das mPart-API-Framework und somit der MpartPeer auf dem System der Benutzer-Anwendung nicht verfügbar, muss der Client-Stub einen Stellvertreter nutzen, um an einem anonymisierten Netzwerk teilnehmen zu können. In Anlehnung an die in Kapitel 3.2 vorgestellte Terminologie des

mPart-API-Frameworks wird dieses als *No-Transport-Client* bezeichnet. Der Stellvertreter wurde analog zum Konzept des mPart-API-Frameworks *Trusted Server* benannt. Auf dem Trusted Server ist das mPart-API-Framework verfügbar und wird dem No-Transport-Client als Service angeboten. Dieses Konzept ist in Abbildung 4.3 veranschaulicht und wird nachfolgend beschrieben.

Der *MpartRpcNoTransportClient* hat keine MpartPeer-Komponente für den Versand und Empfang von anonymisierten Nachrichten zur Verfügung. Auf dem Trusted Server ist dagegen ein MpartPeer verfügbar. Der Versand von Requests und Replys über den Trusted Server wird daher als Service umgesetzt, dem *mPart-RPC-Message-Service (MrmService)*. MrmService ist ein nicht-anonymisierter Service auf Basis des PHPRPC-Bindings. Der Service folgt der Kommunikationsstruktur in MLRPCFW, welche in Kapitel 3.1.2 beschrieben ist.

Auf dem Client-System nimmt ein Consumer, der *MrmServiceConsumer*, die Requests des No-Transport-Clients entgegen. Er versendet sie mit Hilfe eines Client-Stubs, des *MrmServicePhpRpcClients*, an den Trusted Server. Dort nimmt ein Server-Stub, genannt *MrmServicePhpRpcServer*, den Request entgegen. Der Server-Stub ruft die Prozedur zum Versand des Request im *MrmServiceProvider* auf, welcher sie an seinen MpartPeer delegiert. Die Komponenten MpartRpcNoTransportClient auf dem Client und MrmServiceProvider auf dem Trusted Server erbringen gemeinsam die Funktionalität eines Full-Transport-Clients. Der MrmService verbindet beide Komponenten. Beim Versand eines Requests erfolgen synchrone Prozeduraufrufe, welche alle Komponenten bis hin zum MrmServiceProvider auf dem Trusted Server blockieren. Hat dieser eine Antwort-Nachricht vom Server empfangen, kommuniziert er die Reply auf umgekehrtem Weg zum Consumer und deblockiert damit Schritt für Schritt wieder alle wartenden Komponenten.

Mit der Einführung des MrmServices wird das zweistufige Client-Server-Modell von MLRPCFW zu einem dreistufigen Modell, bei ein Service einen weiteren Service verwendet. Der Trusted Server agiert wie in Kapitel 2.1.1 beschrieben selbst wieder als Client gegenüber einem Server.

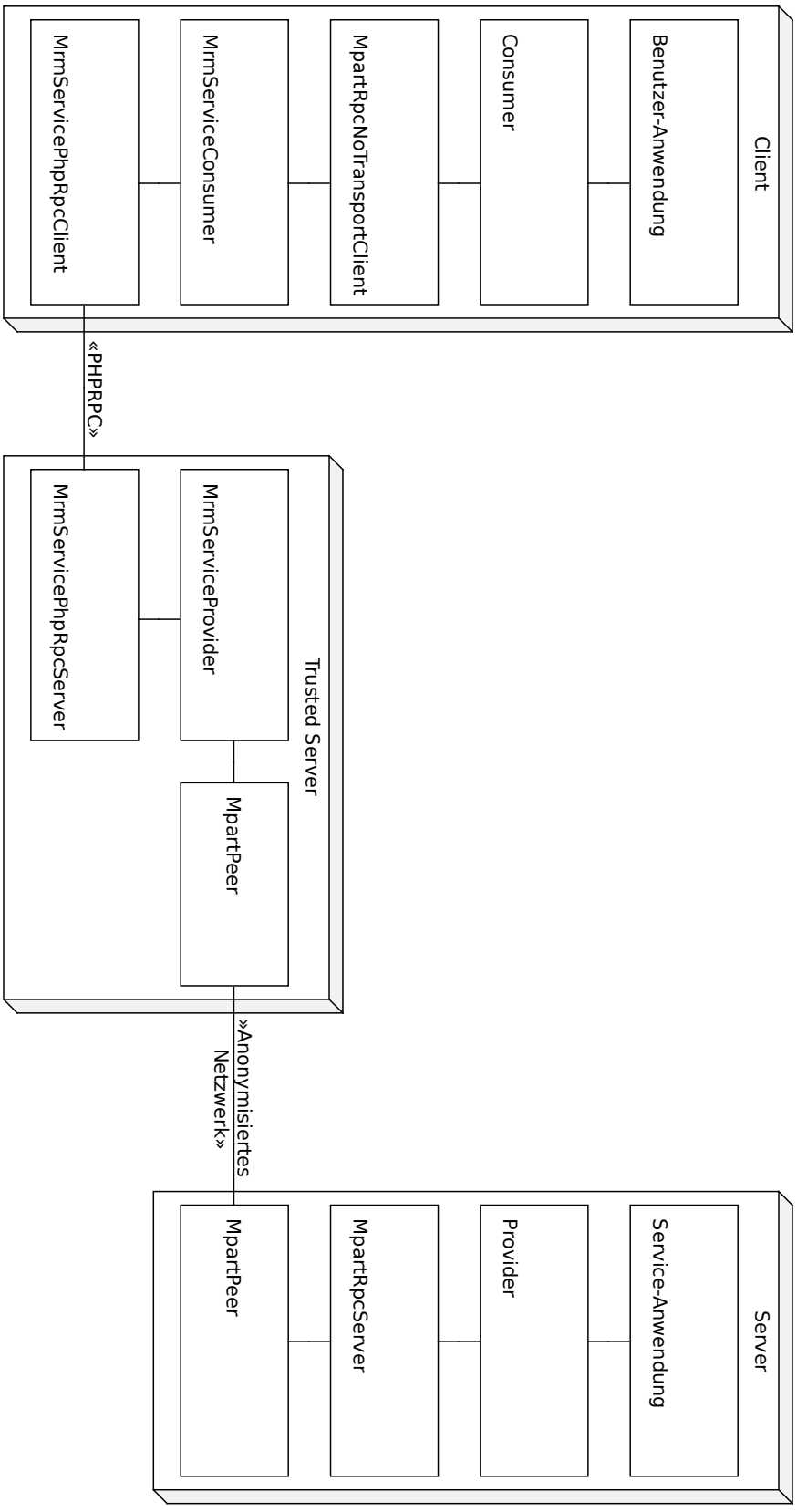

Abbildung 4.3: Konzept „No-Transport-Client“ im mPart-RPC-Binding

Kapitel 5

Prototypische Implementierung des Bindings

Im folgenden Kapitel wird die Umsetzung des zuvor entworfenen Konzepts für ein mPart-RPC-Binding als prototypische Implementierung beschrieben. Die im Konzept beschriebenen Komponenten werden in einem Java-Klassensystem umgesetzt. Das Kapitel schließt mit der Implementierung des Beispiel-Szenarios unter Verwendung des mPart-RPC-Bindings.

5.1 Request und Reply

Wie in Kapitel 2.2 beschrieben, sendet der Client-Stub zur Ausführung eines RPC eine Anfrage an den Server-Stub. In diesem *Request* muss eine Kennung für die gewünschte Prozedur enthalten sein, damit der Server-Stub diese zuordnen kann. Weiterhin muss der Request von der Prozedur benötigte Parameter beinhalten. Der MpartRpcClient und MpartRpcServer verwendet einen eigenen Datentyp für die Requests, welcher in Abbildung 5.1 dargestellt ist. Es handelt sich um ein einfaches Objekt, welches als Kennung den Methodennamen enthält und ein Array von Objekten, welche die Argumente für die entfernte Methode darstellen. Die Reply des MpartRpcServers wird in dieser prototypischen Implementierung ohne umschließenden Datentyp direkt verschickt, da stets nur eine Antwort

je Methodenaufruf erwartet wird.

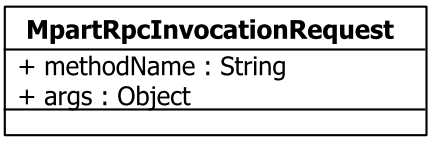

Abbildung 5.1: Der Datentyp MpartRpcInvocationRequest

5.1.1 Serialisierung von Objekten

Wie in Kapitel 2.2.1 beschrieben, müssen Objekte zunächst in einen Byte-Strom umgewandelt werden, damit sie versandt werden können. Diese sogenannte Serialisierung muss zuvor zwischen Client und Server vereinbart worden sein, damit die Objekte auf der Seite des Empfängers wieder deserialisiert werden können. Da Java die Zielplattform der prototypischen Implementierung ist, wurde der in der Java-Standardbibliothek vorhandene *Serializer* zur Serialisierung verwendet. So ist sichergestellt, dass alle Java-Objekte übertragen werden können. Da der Java-Serializer nicht unbedingt in anderen Programmiersprachen zur Verfügung steht, wurde er in einer eigenen Klasse gekapselt, damit er bei Bedarf leicht gegen eine andere Serialisierungslösung getauscht werden kann. MpartRpcClient und MpartRpcServer setzen die Serialisierung also nicht selbst um, sondern nutzen dafür die Klasse *MpartSerializer*. Dies ermöglicht die Wiederverwendung der prototypischen Implementierung in zukünftigen Iterationen des Konzepts.

In einer ersten Umsetzung des No-Transport-Client-Konzepts wurde der serialisierte Request einfach als Byte-Strom bis zum MpartPeer durchgereicht. Auf dem gesamten Trusted Server bestand also keine Möglichkeit auf die kommunizierten Objekte zuzugreifen, da diese nur serialisiert vorlagen. Dieses Vorgehen reduziert einerseits den Rechenaufwand, da für jede Anfrage oder Antwort nur jeweils einmal serialisiert und deserialisiert wird. Andererseits verhindert die Kapselung der Daten in der serialisierten Form eine potentielle Wiederverwendung von Komponenten des

Trusted Servers. Im Sinne des Prinzips der Mehrfachverwendung [Hel89, S. 59], sollte auf die Kapselung der Daten verzichtet werden.

In einer zweiten Umsetzung wurden Objekte daher nur unmittelbar vor dem Transport über eine Netzwerkverbindung serialisiert und danach auch unmittelbar wieder deserialisiert. Diese Lösung weist den Vorteil auf, jederzeit auf die kommunizierten Objekte zugreifen zu können und ermöglicht künftige Modifikationen der Objekte auf dem Trusted Server. Zwischen dem Client- und dem Server-Stub des MrmService wurde dabei die Serialisierung durch das PHPRPC-Binding genutzt. Zwischen dem MrmServiceProvider auf dem TrustedServer und dem MpartRpcServer wiederum der MpartSerializer.

5.2 MpartPeer

Die Klasse MpartPeer übernimmt das Initialisieren des mPart-API-Frameworks und den Versand von Nachrichten über das Framework. Sie ist in Abbildung 5.2 als Klassendiagramm dargestellt. Die Klasse stellt zwei Me-

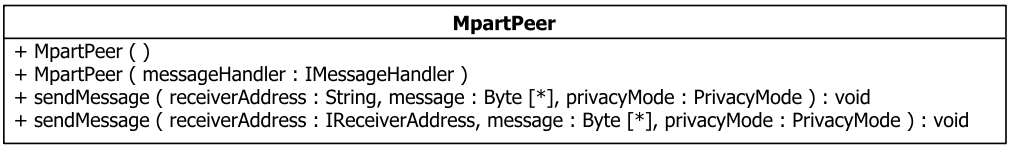

Abbildung 5.2: Die Klasse MpartPeer

thode für den Versand von Nachrichten über das mPart-API-Framework zur Verfügung. Diese unterscheiden sich lediglich in der Form, in der die Adresse des Empfängers beim Methodenaufruf anzugeben ist. Wird die Adresse beispielsweise aus der Benutzeroberfläche ausgelesen, übernimmt der MpartPeer sie als String und überführt sie selbst in den passenden Datentyp. Liegt die Adresse jedoch bereits im Datentyp des mPart-API-Frameworks vor, kann dieser Datentyp auch direkt als Argument genutzt werden. Letzteres ist beim Antworten auf eine Nachricht der Fall, da hier die Antwort-Adresse im Datentyp *IReceiverAddress* aus der ur-

sprünglichen Nachricht genutzt wird. Bereits mit Instantiieren der Klasse wird eine Verbindung zum anonymisierten Netzwerk I2P aufgebaut. Dieses ist derzeit als einziges Protokoll-Modul im Prototyp des mPart-API-Frameworks enthalten. Aus Sicherheitsgründen wird in I2P nicht eine Adresse pro System genutzt, vielmehr erhält jede Anwendung ihre eigene Adresse, die sogenannte *Destination* [Jua12]. Dabei handelt es sich um eine 517 Zeichen lange, zufällig generierte Adresse im Base64-Format. Um die Wartezeit beim ersten Nachrichtenversand zu reduzieren, erfolgt das Öffnen der Verbindung und damit Zuweisen der Adresse bereits im Konstruktor des MpartPeer. Die Klasse MpartPeer kann auf zwei Arten instantiiert werden: mit oder ohne Angabe eines Message-Handlers. Dieser Handler wird beim mPart Core registriert und dient der Benachrichtigung der Anwendung, hier des MpartRpcClients, bei Eingang einer Nachricht über ein anonymisiertes Netzwerk. Wird kein Handler angegeben, registriert der MpartPeer einen leeren Message-Handler, der eingehende Nachrichten verwirft. Dieses Vorgehen kann verwendet werden, falls keine Nachrichten empfangen, sondern nur gesendet werden sollen.

5.3 MpartRpcClient

Die Klasse MpartRpcClient führt die entfernten Methodenaufrufe mittels des MpartPeers durch. Sie ist in Abbildung 5.3 dargestellt. Durch ihren

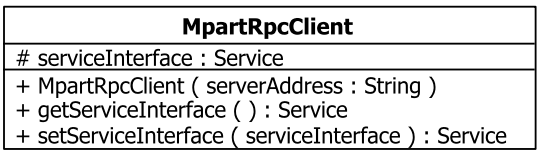

Abbildung 5.3: Die Klasse MpartRpcClient

Konstruktor erhält sie die Adresse des zu verwendenden MpartRpcServers, beziehungsweise dessen MpartPeers. Die Klasse enthält Methoden zum Setzen und Abfragen seines Service, auf welchen auch eine Referenz vorgehalten wird.

Wird im MpartRpcClient ein Service-Interface gesetzt, erzeugt er daraus mittels *Reflexion* eine Service-Klasse mit den im Interface enthaltenen Methoden. Die tatsächliche Funktionalität des Service steht auf dem Client auch weiterhin nicht zur Verfügung, sondern muss bei einem Server angefragt werden. Daher wird immer, wenn eine Methode in der erzeugten Service-Klasse aufgerufen wird, ein sogenannter *Invocation-Handler* ausgeführt, welcher den Methodenaufruf an den MpartRpcServer versendet und dann die Antwort des Servers zurückgibt. Dieser Ablauf ist in Abbildung 5.4 verdeutlicht. Die aufgerufene Methode ist dabei beispielhaft

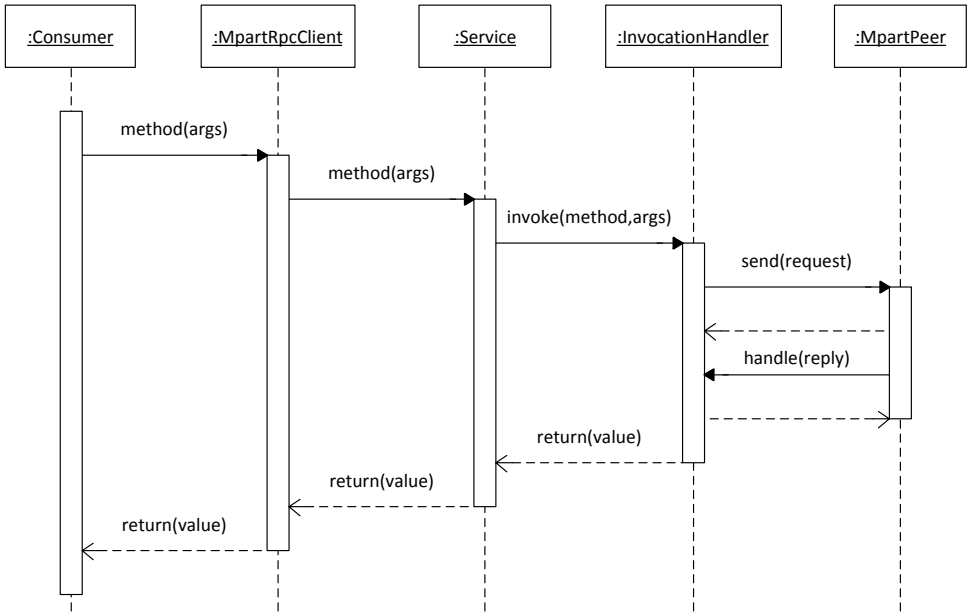

Abbildung 5.4: Invocation-Handler im MpartRpcClient

als `method()` benannt. Der Invocation-Handler stellt den eigentlichen Client-Stub dar, da er den Aufruf samt Argumenten verpackt und die Antwort-Nachricht des Servers entgegen nimmt. Auch hier wird wie in Kapitel 4.2 eine asynchrone Kommunikation über die beiden synchronen Aufrufe `sendMessage()` und `handleMessage()` abgewickelt. Nun blockiert der Client-Stub, bis die Antwort-Nachricht des Servers eingetroffen ist. Hierzu wird ein Wartemechanismus ähnlich eines *Semaphors* [Dou01, S. 155] eingesetzt, bei dem der Zugriff auf eine Ressource über ein Si-

gnal geregelt wird. Die Ressource ist hierbei der noch nicht eingegangene Rückgabewert, auf den deshalb noch nicht zugegriffen werden darf. Das Signal wird innerhalb des Invocation-Handlers durch die Methode `handleMessage()` gesetzt. Die dazwischen liegende Zeit verbringt der Invocation-Handler mit regelmäßigem Prüfen des Signals, dem sogenannten *Busy Waiting* [Dou01, S. 155]. Nach Eingang der Antwort erkennt die Methode `method()` am gesetzten Signal, dass die Antwort des Servers vorliegt und an den Consumer zurückgeben werden kann.

Nach dem Programmiermuster, welches von MLRPCFW vorgegeben ist, implementiert der konkrete MpartRpcClient einer verteilten Anwendung auch deren Service-Interface. Er stellt somit alle darin definierten Methoden für einen Consumer zur Verfügung und delegiert die Aufrufe dann an seinen Invocation-Handler. Beim MpartRpcNoTransportClient wird dieses Konzept analog umgesetzt. Der Invocation-Handler befindet sich dann allerdings beim MrmServiceProvider auf dem Trusted-Server, da dieser auch auf den MpartPeer zugreift und daher dessen asynchronen Aufruf mit der Antwort-Nachricht entgegen nimmt. Der genaue Ablauf innerhalb des MrmService wird in Kapitel 5.5 beschrieben.

5.4 MpartRpcServer

Die Klasse MpartRpcServer empfängt über ihren MpartPeer die Requests des MpartRpcClients und leitet diese an ihren Provider weiter. Sie ist in Abbildung 5.3 dargestellt. Durch ihren Konstruktor erhält sie eine Referenz auf einen Provider, an den sie Methodenaufrufe in ankommenden Requests delegiert.
Mit Aufruf der Methode `start()` wird der MpartPeer des Servers instantiiert. Dieser meldet mittels der Methode `receiveMessage()`, sobald eine neue Nachricht eingegangen ist. Der darin enthaltene Request wird vom Server entpackt und geprüft, ob eine passende Methode im Provider vorhanden ist. In diesem Fall wird die Methode des Providers mit den ebenfalls im Request enthaltenen Argumenten aufgerufen. Der MpartRpc-

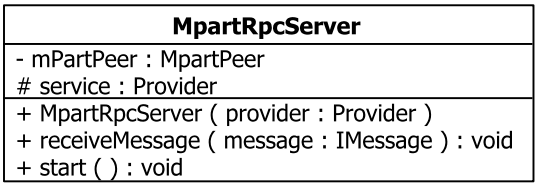

Abbildung 5.5: Die Klasse MpartRpcServer

Server wartet auf die Antwort des Providers und verschickt diese über den MpartPeer wieder an den anfragenden MpartRpcClient.

5.5 MrmService

Der *mPart-RPC-Message-Service (MrmService)* dient dem Versand von Requests und Erhalten von Replys über das mPart-API-Framework. Er wurde mittels des PHPRPC-Bindings umgesetzt und besteht aus den üblichen Klassen Consumer, Client, Server und Provider. Letzterer setzt die eigentliche Funktionalität des Service mittels eines MpartPeers um. Das Service-Interface, welches in Abbildung 5.6 dargestellt ist, besteht folglich nur aus einer Methode, welche die Adresse des anzufragenden Servers und den Request als Argumente übergeben bekommt. Die Methode gibt ein beliebiges Objekt zurück, welches die Reply des Servers auf den Request darstellt.

Abbildung 5.6: Das Service-Interface des MrmService

Wie in Abschnitt 3.1.3 erläutert, liegt der Server-Stub des PHPRPC-Bindings nur in PHP vor. Da jedoch der MpartPeer aufgrund der Abhängigkeit zum mPart-API-Framework nur in Java umgesetzt werden kann, musste der MrmServiceProvider geteilt in Java und PHP implementiert werden. Es existiert also eine gleichnamige Klasse auf der PHP- und der Java-

Seite. Dies ist in Abbildung 5.7 dargestellt. Der MrmServiceProvider auf Java-Seite wird samt MpartPeer durch den Provider auf PHP-Seite mittels Kommandozeilen-Aufrufs instantiiert. Er führt dann den gewünschten Versand des Requests durch, wartet auf die Antwort des Servers und gibt diese zurück. Anschließend terminiert er wieder.

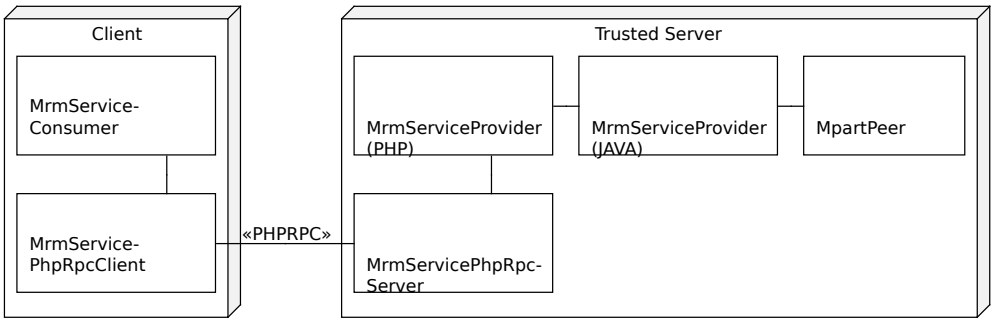

Abbildung 5.7: Übersicht über die Komponenten des MrmService

5.6 Beispiel-Szenario Student and Professor Chat

Um die praktische Nutzung des entwickelten Bindings zu erproben, wurde ein Beispiel-Szenario mit der prototypischen Implementierung umgesetzt. In diesem Szenario kommuniziert ein Student über eine verteilte Anwendung mit einem Dozenten. Der Student soll dabei anonym bleiben. Der Dozent soll hingegen über ein Pseudonym erreichbar sein, welches er vorab bekannt gegeben hat. Entsprechend dem Paradigma von ML-RPCFW wurde zunächst ein Service-Interface für diesen Service definiert, welches in Abbildung 5.8 dargestellt ist.

5.6.1 Service-Interface SpcService

Im Szenario soll zwischen den Parteien einseitig anonym kommuniziert werden. Es muss also eine Partei bekannt sein, um die Kommunikation zu beginnen. Daher definiert der Service eine Methode `registerUser()`

zur Registrierung eines Pseudonyms mit einem Passwort. Nur wer das Passwort kennt, kann das Pseudonym verwenden. Ein Dozent kann ein solches Pseudonym registrieren, um eine Kontaktaufnahme zu ermöglichen. Mittels der Methode `poseQuestion` kann eine Frage für ein be-

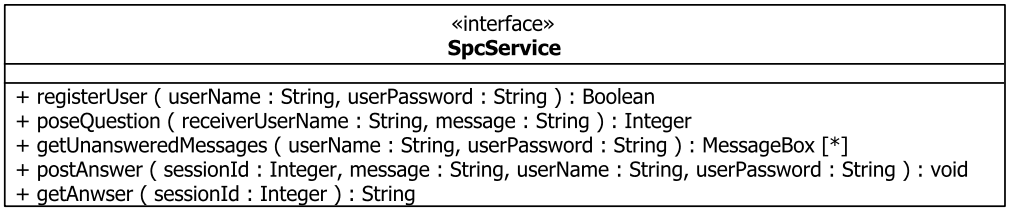

Abbildung 5.8: Das Service-Interface SpcService

kanntes Pseudonym hinterlegt werden. Für das Stellen von Fragen ist keine Registrierung erforderlich. Beim Stellen einer Frage wird eine Session erzeugt und ein Ganzzahl als Session-Nummer zurückgegeben. Mit dieser Session-Nummer kann der anonyme Nutzer die Antwort auf seine Frage abrufen. Der Dozent kann mit seinem registrierten Pseudonym die Methoden `getUnansweredMessages()` und `postAnswer()` nutzen. Sie dienen dem Abfragen von noch nicht beantworteten Fragen an das Pseudonym und dem Hinterlegen von Antworten.

5.6.2 Umsetzung mit dem mPart-RPC-Binding

Zur Umsetzung des Szenarios wurde je ein *SpcConsumer* und *SpcProvider* von den abstrakten Klassen des MLRPCFW abgeleitet. Sie implementieren beide das oben vorgestellte Service-Interface. Der Consumer leitet die Methodenaufrufe lediglich an einen Client-Stub weiter, während der Provider die Funktionalität des Service zur Verfügung stellt. Für die Speicherung von Nachrichten auf der Server-Seite wurden einfache Datenstrukturen verwendet. Somit existiert keine Service-Anwendung im engeren Sinne, obgleich die Daten ebenso in einer Datenbank gespeichert sein könnten. Neben Consumer und Provider wurden Client- und Server-Stubs aus den Klassen MpartRpcClient, MpartRpcNoTransportClient und

MpartRpcServer des mPart-RPC-Bindings abgeleitet. Die so erzeugten Klassen implementieren alle die Methoden des SpcService-Interfaces und delegieren diese bei Aufruf an die allgemeinen Methoden ihrer jeweiligen Oberklasse. Beispielsweise nutzt der SpcMpartRpcClient den allgemeinen Invocation-Handler, wie er in Abschnitt 5.3 beschrieben ist.

5.6.3 Ergebnisse des Testlaufs

In einem Testlauf wurden alle Methoden des SpcConsumers jeweils in Kombination mit dem SpcMpartRpcClient und dem SpcMpartNoTransportClient getestet. Dazu wurde auf einem virtuellen Server ein SpcProvider samt SpcMpartRpcServer gestartet und auf einem weiteren virtuellen Server die Server-Seite des MrmService in Betrieb genommen. Das vom mPart-API-Framework verwendete I2P-Netzwerk-Modul nutzt zur Adressierung sogenannte *Destinations*, wie in Kapitel 5.2 erläutert. Da diese Destinations zufällig generiert werden, gibt der Prototyp des mPart-API-Frameworks sie beim Öffnen einer Verbindung auf der Kommandozeile aus. Diese Destination muss dem MpartRpcClient zum Herstellen der Verbindung zum Server mitgeteilt werden. Im Testlauf erfolgte dies durch händisches Kopieren der Adresse vom Server zum Client bei jeder Ausführung. Beim Testlauf des Full-Transport-Clients konnten alle entfernten Methoden erfolgreich ausgeführt werden. Der Testlauf des No-Transport-Clients verlief unter Verwendung der ersten Serialisierungsform aus Abschnitt 5.1.1 ebenfalls erfolgreich. Unter Verwendung der zweiten Serialisierungsform, bei der auf dem Trusted Server eine erneute Deserialisierung und Serialisierung stattfindet, kam es jedoch zu Laufzeitfehlern auf der Client-Seite. Diese konnten als Zeitüberschreitungs-Fehler im PHPRPC-Binding identifiziert werden: Die erneute Serialisierung und Deserialisierung auf dem Trusted Server dauerte in Kombination mit dem Kommandozeilenaufruf der Java-Klassen zu lange, um rechtzeitig eine Antwort vom angefragten Server zu liefern. Bevor die Antwort an den No-Transport-Client ausgeliefert werden konnte, brach dort das Binding aufgrund eines Timeouts die Verbindung ab.

Kapitel 6

Evaluation

Im Laufe dieser Arbeit wurde das MLRPCFW um ein Binding erweitert, welches anonymisierte RPCs zur Verfügung stellt. Dieses Binding wurde auf der Basis des mPart-API-Frameworks für anonymisierte Kommunikation entworfen. Um den vollen Funktionsumfang des MLRPCFW auch über anonymisierte Netzwerke nutzen zu können, wurde mit dem mPart-RPC-Binding eine eigene RPC-Implementierung unter Nutzung anonymisierter Netzwerke entwickelt. Dieses Binding ist ebenso generisch in Bezug auf die darüber abgewickelte Anwendung wie das bereits im MLRPCFW vorhandene. Mit dem mPart-RPC-Binding können also verteilte Anwendungen genauso umgesetzt werden, wie auch unter Verwendung des PHPRPC-Bindings. Durch die Integration des mPart-API-Frameworks ist es möglich, beliebige Netzwerke, welche als Protokoll-Modul in das mPart-API-Framework integriert wurden, zur Abwicklung von RPCs zu nutzen.

Mit der Entwicklung von eigenständig an anonymisierten Netzwerken teilnehmenden Client- und Server-Stubs konnte das Konzept des Full-Transport-Clients erfolgreich umgesetzt werden. Das Beispiel-Szenario *Student and Professor Chat* zeigt eine lauffähige Demonstration eines Praxis-Einsatzes. Die prototypische Implementierung wird nur durch das frühe Entwicklungsstadium des verwendeten Netzwerks I2P im Komfort beim Adressieren der beteiligten Systeme gemindert.

Das Konzept des No-Transport-Clients konnte unter Wiederverwendung des PHPRPC-Bindings ebenfalls umgesetzt werden. Die anonymisierte Kommunikation über einen vertrauenswürdigen Dritten wurde mit der Entwicklung eines entsprechenden Service auf einem Trusted Server realisiert. In der praktischen Erprobung im Beispiel-Szenario traten Probleme auf, welche auf Timeout-Probleme zur Laufzeit zurückgeführt werden konnten. Hierbei handelt es sich jedoch um ein Problem des spezifischen Prototyps und nicht um eine grundsätzliche Problematik des Konzepts. Entsprechend konnte die Umsetzbarkeit des Konzepts mit einem weniger rechenintensiven Ansatz dennoch gezeigt werden.

Eine konzeptuelle Schwäche des No-Transport-Clients ist die notwendige Vetrauenswürdigkeit des Trusted Servers und der Netzwerkverbindung zu diesem. Da die Verbindung zum Trusted Server nicht anonym ist, muss dem Netzwerk vertraut werden. Auch eine Verschlüsselung der Verbindung schafft keine Abhilfe. Remailer-Attacken, wie etwa von Lance Cottrell [Lan98] beschrieben, ermöglichen es durch Analyse der Nachrichten einen Zusammenhang zwischen den eingehenden und ausgehenden Nachrichten am Trusted Server herzustellen. Dabei wird etwa die Reihenfolge betrachtet, in der die Nachrichten am Trusted Server eingehen. Auf diese Weise können die in der selben Reihenfolge abgehenden, anonymisierten Nachrichten einen Schluss auf den Empfänger zulassen. Auch über die Größen der am Trusted Server eingehenden und abgehenden Nachrichten können Zusammenhänge hergestellt werden. Am Stärksten ist die Anonymität durch Manipulation des Trusted Servers selbst angreifbar, da hier Request und Zieladresse direkt vorliegen. Insgesamt stellt der Trusted Server eine Behelfslösung für noch nicht unterstützte Plattformen dar, die sobald möglich durch Full-Transport-Clients ersetzt werden sollte.

Kapitel 7

Fazit und Ausblick

Diese Arbeit begann mit dem Ziel, die Entwicklung anonymer Informationssysteme zu vereinfachen. Dazu wurde mit dem mPart-RPC-Binding für MLRPCFW anonymisierbare Kommunikation in ein Software-Framework für verteilte Anwendungen integriert. Die Evaluation von Konzept und Prototyp zeigt eine erfolgreiche, funktionsfähige Integration, welche durch den frühen Entwicklungsstand der verwendeten Prototypen in Komfort und Leistungsverhalten eingeschränkt ist. Das eingangs beschriebene Beispiel-Szenario konnte erfolgreich umgesetzt werden. Dies zeigt die Machbarkeit des Konzepts auf und weist den Weg für künftige Entwicklungen. Da eine Implementierung nur für die Zielplattform Java vorgenommen wurde, muss geprüft werden, ob das Konzept auch auf anderen Plattformen realisierbar ist. Auf dem Trusted Server könnten für die Übergabe der Requests statt eines Konsolenaufrufs alternative Methode erprobt werden. Der MpartPeer sollte dabei permanent ausgeführt werden, statt erst bei einem ankommenden Request zu starten. Da die zeitaufwendige Initialisierung der Netzwerkverbindung so vorab geschieht, würde diese Änderung die Laufzeit der Requests zum Server zu verringern. Ein Beispiel für diesen Ansatz wäre die Kommunikation über einen lokalen Port. Um den Wechsel zwischen den Programmiersprachen zu vermeiden, könnten alle Komponenten des Trusted Servers in Java implementiert werden. Dazu muss zunächst eine Java-Server-Implementierung

für das PHPRPC-Binding entwickelt werden. Umgekehrt könnte auf die Java-Komponenten auf dem Trusted Servers verzichtet werden, wenn das mPart-API-Framework für PHP verfügbar wäre. Die Verwendung eines Trusted Servers bietet zusätzliche Angriffsfläche in Bezug auf die Anonymität des Nutzers, wie im vorhergehenden Kapitel 6 beschrieben wurde. Idealerweise kann durch die Verfügbarkeit beider Frameworks auf weiteren Plattformen ganz auf dieses Konzept verzichtet werden.

Künftige Weiterentwicklungen der anonymisierten Netzwerke können den Komfort der Adressierung auf ein Niveau heben, wie es heute schon beim Internetprotokoll Standard ist. Für das verwendete I2P-Netzwerk werden Konzepte eines Adressbuch-Systems[1] erprobt, welche dem Domain Name System gleichen. Durch Angabe einer leicht zu merkenden Adresse kann dann komfortabel eine einseitig anonyme Kommunikation erfolgen. Wie die vorliegende Arbeit zeigt, muss sich diese Kommunikation nicht nur auf den Austausch von Nachrichten beschränken. Vielmehr könnte dann jegliche verteilte Anwendung anonymisiert genutzt werden. Die anonyme Nutzung von Informationssystemen im Alltag ist damit möglicherweise schon bald Standard.

[1] `https://geti2p.net/en/docs/naming`

Literaturverzeichnis

[And97] Andreas Rüping. *Software-Entwicklung mit objektorientierten Frameworks*. Shaker Verlag, 1997.

[And03a] Andrew S. Tanenbaum. *Computernetzwerke*. Pearson Studium, 2003.

[And03b] Andrew S. Tanenbaum, Maarten van Steen. *Verteilte Systeme - Grundlagen und Paradigmen*. Pearson Studium, 2003.

[And10] Andreas Pfitzmann, Marit Hansen. A terminology for talking about privacy by data minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management. 2010.

[Ans09] Ansgar Scherp, Susanne Boll. Framework-Entwurf. In Ralf Reussner, editor, *Handbuch der Software-Architektur*. dpunkt - Verlag für digitale Technologie GmbH, 2009.

[Bun77] Bundesministerium der Justiz. *Bundesgesetzblatt Teil 1 - Nr. 7 vom 1. Februar 1977*. Bundesanzeiger Verlag, 1977.

[Dou01] Douglas E. Comer, David L. Stevens. *Internetworking with TCP/IP - Volume 3: Client-Server Programming and Applications*. Prentice Hall, 2001.

[fIe] Gesellschaft für Informatik e.V. Privatsphäre schützen und sich gegen flächendeckende Ausspähung wehren. `http://www.gi.de/aktuelles/meldungen/detailansicht/article/`

privatsphaere%2Dschuetzen%2Dund%2Dsich%2Dgegen%2Dflaechendeckende%2Dausspaehung%2Dwehren.html, zuletzt abgerufen am 21. Juli 2014.

[Gün02] Günther Bengel. *Verteilte Systeme - Client-Server-Computing für Studenten und Praktiker*. Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, 2002.

[Hak14] Hakan Aksu, Tobias Bartsch, Ludwig Paulsen, Fabian Vickus, Freya Surberg, Falko Cremer, Martin Kastner. Mobiles Studentenportal 6. Technical report, Universität Koblenz-Landau, 2014.

[Hel89] Helmut Balzert. *Die Entwicklung von Software-Systemen : Prinzipien, Methoden, Sprachen, Werkzeuge*. B.I.-Wissenschaftsverlag, 1989.

[Joh00] Johann Bizer. Recht auf Anonymität. In Bettina Sokol, editor, *Datenschutz und Anonymität*, 2000.

[Jua12] Juan Pablo Timpanaro, Isabelle Chrisment, Olivier Festor. A Bird's Eye View on the I2P Anonymous File-Sharing Environment. In *Network and System Security - 6th International Conference*, pages 135–148, 2012.

[Lan98] Lance Cottrell. Mixmaster & Remailer Attacks, 1998. https://web.archive.org/web/20011021133038/http://obscura.com/~loki/remailer-essay.html, zuletzt abgerufen am 22. Juli 2014.

[Pat13] Patrice Matthias Brend'amour. Generisches Framework für die mobile Kommunikation von anonymisierten Gruppen. Master's thesis, Universität Koblenz-Landau, 2013.

[Vij04] Vijay Vaishnavi, Bill Kuechler. Design Science Research in Information Systems, 2004. http://desrist.org/desrist/content/

design-science-research-in-information-systems.pdf, zuletzt abgerufen am 18. Juli 2014.

[Wie95] Wieland Appelfeller. *Wiederverwendung im objektorientierten Softwareentwicklungsprozeß, dargestellt am Beispiel der Entwicklung eines Lagerlogistiksystems*. Peter Lang GmbH, 1995.

[Wol97] Wolfgang Pree. *Komponenten-basierte Softwareentwicklung mit Frameworks*. dpunkt - Verlag für digitale Technologie GmbH, 1997.