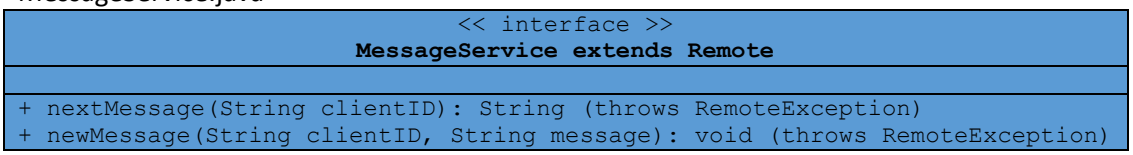
Verteilte Systeme Praktikum WS2014
Teilnehmer: Michael Schmidt, Eugen Winter

1. Praktikumsaufgabe: Client/Server-Anwendung „Verteilte Nachrichten-Queue“

Für die Implementierung einer RMI Server/Client-Anwendung soll Java RMI verwendet werden.
Es gilt das folgende Interface zu implementieren:

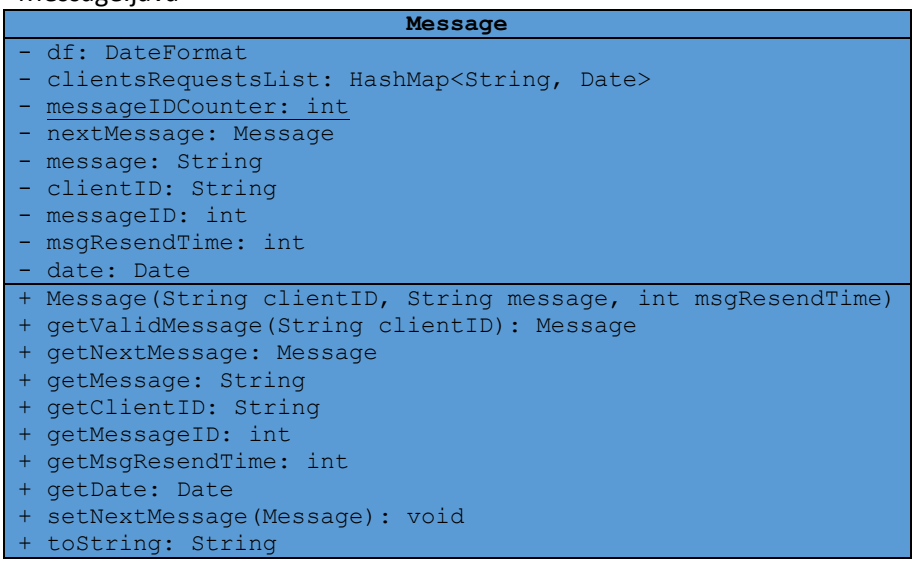
MessageService.java



Programmaufbau

Klassen:

Message.java



Die Message-Klasse kapselt die Informationen, die mit der newMessage- Funktion übermittelt werden. Sie speichert neben dem reinen Message-String und der Client-ID auch eine fortlaufend vergebene Message-ID, die Uhrzeit und die Gedächtniszeit *t* in der Variablen *msgResendTime*. Außerdem kennt sie für die spätere Verwendung in einer Delivery-Queue nach FIFO Prinzip eine Referenz auf ihr nächstes Message-Objekt.

Um die korrekte Funktionsweise der Gedächtniszeit *t* zu realisieren, besitzt jedes Message-Objekt zusätzlich eine HashMap in welcher als Key/Value Paare jeweils die Client-ID und die Zeit gespeichert werden, sobald ein Client die nextMessage-Funktion benutzt.

Beim Abruf einer Message heißt dies, dass unbekannte Clients die Nachricht erhalten und gleichzeitig mit Client-ID und Uhrzeit in der HashMap des Message-Objekts hinterlegt werden.
Ein erneutes Abrufen einer Message durch einen bekannten Client führt durch die Bildung der Differenz der aktuellen und der hinterlegten Zeit dazu, dass diese Nachricht nicht noch einmal an denselben Client gesendet wird, solange die Gedächtniszeit *t* seit dem letzten Abruf noch nicht überschritten ist.
Gleichzeitig frischt jeder erneute Abruf einer Message die hinterlegte Gedächtniszeit *t* für den bekannten Client auf, indem das alte Key/Value Paar aus Client-ID und Zeit gelöscht und ein neues mit derselben Client-ID und der aktuellen Zeit erstellt wird.

Verteilte Systeme Praktikum WS2014
Teilnehmer: Michael Schmidt, Eugen Winter

DeliveryQueue.java

DeliveryQueue
- firstMessage: Message - currentSize: int - MAXSIZE: int
+ DeliveryQueue(int maxSize) + addMessage(Message message): void + getMessage(String clientID): Message + toString: String

Die Delivery-Queue wird mit einer maximalen Größe vorinitialisiert. Sie besitzt zwei Funktionen zum Ablegen und Herausholen der Message-Objekte aus der Delivery-Queue. Dabei ist der Speicher nach dem FIFO-Prinzip aufgebaut und die Message-Objekte liegen als verkettete Liste vor. Ist die maximale Kapazität erreicht, wird das älteste Message-Objekt aus der Liste entfernt.
Die getMessage-Funktion greift zurück auf die getValidMessage-Funktion eines Message-Objekts, um sicherzustellen, dass nur neue, nicht bereits abgerufene Nachrichten an Clients zurückgegebenen werden und dabei nicht innerhalb der Gedächtniszeit *t* des jeweiligen Clients liegen.

MessageServiceServer.java

MessageServiceServer extends UnicastRemoteObject implements MessageService
- SERIALVERSIONUID: long - LOGSEND: File - LOGRECV: File - stringBuffer: StringBuilder - messageService: MessageService - deliveryQueue: DeliveryQueue - registry: Registry - serverName: String - msgResendTime: int - deliveryQueueSize: int - textServerName: JTextField - textQueueSize: JTextField - textMsgResendTime: JTextField - textOutputArea: JTextArea - buttonStartRegistry: JButton - buttonStartServer: JButton - buttonShowQueue: JButton
+ MessageServiceServer() + nextMessage(String clientID): String + newMessage(String clientID, String message): void - createAndShowGUI(): void - showDeliveryQueue(): void - findIpAddr(): void - writeLog(String clientID, String message, File logfile): void - startRegistry(): void - startServer(): void

Der Server implementiert das MessageService-Interface und konkretisiert somit die beiden Funktionen nextMessage und newMessage. Da die Logik hierfür bereits in der DeliveryQueue-, sowie der Message-Klasse implementiert worden sind, werden die Remote-Aufrufe von Clients direkt an ein statisches DeliveryQueue-Objekt weitergeleitet, welches die Nachrichten der Clients als Message-Objekte verpackt entgegennimmt, bzw. sie als solche zurückliefert.
Die **At-most-once Fehlersemantik** des Servers wird mit Hilfe der Gedächtniszeit *t* bei der Implementierung der Message-Klasse realisiert. Auf diese Weise verschickt der Server die durch den Client mehrfach angeforderte Nachricht innerhalb der Gedächtniszeit *t* nur ein einziges Mal.

Weiterhin besitzt der Server eine GUI, eine Funktion zum Ermitteln der lokalen IP-Adressen, Logging, sowie die Möglichkeit die Java RMI Registry direkt aus der GUI zu starten und danach den Server an der RMI Registry anzumelden.

MessageServiceClient.java

MessageServiceClient
- LOGFILE: File - stringBuffer: StringBuilder - messageService: MessageService - registry: Registry - messageToSend: String - serverAddr: String - serverName: String - clientID: String - serverTimeout: int - isResending: boolean - frame: JFrame - textServerAddr: JTextField - textServerName: JTextField - textClientID: JTextField - textServerTimeout: JTextField - textOutputArea: JTextArea - textInputArea: JTextArea - buttonConnect: JButton - buttonExit: JButton - buttonSend: JButton - buttonRecv: JButton
- createAndShowGUI: void - writeLog(String message): void - connect: void - resetGUI: void - sendMessage: void - receiveMessage: void

Der Client besitzt eine GUI, über die man die Adresse und den Namen des Remote-Servers angeben kann. Ebenso seinen Namen, bzw. die Client-ID und den Wert für den Server-Timeout, falls es zu Problemen bei der Verbindung oder einem Verbindungsabbruch kommen sollte.
Nachrichten werden im unteren Teil des Fensters getippt und mit einem Klick auf den Send-Button oder via Enter-Taste an den Server geschickt.
Mit Hilfe des Receive-Buttons lassen sich alle neuen Nachrichten und Nachrichten außerhalb der Gedächtniszeit *t* abrufen. Diese erscheinen dann im Textfeld in der Mitte der GUI.

Die **Maybe Fehlersemantik** für das Empfangen von Nachrichten wird durch den Aufruf der receiveMessage-Funktion dahingehend umgesetzt, als dass sie versucht einmalig alle neuen Messages vom Server abzuholen. Messages die „null“ sind, also nicht existieren, bleiben dabei unberücksichtigt. Im Fehlerfall wird der Client zwar gewarnt, aber es wird nicht automatisch versucht einen neuen Request an den Server zu schicken.

Die **At-least-once Fehlersemantik** in der sendMessage-Funktion fängt eine beim fehlgeschlagenen Senden entstandene RemoteException ab, setzt ein Flag zum Wiederversenden der Nachricht und ruft sich danach erneut auf.
Aufgrund des gesetzten Flags wird nun der else-Zweig der Funktion durchlaufen. Dieser wird in einem SwingWorker-Objekt abgearbeitet, um die GUI nicht einzufrieren.
Der User wird gefragt, ob er für die Zeit des vorher in der GUI angegebenen Server Timeouts versuchen möchte die Nachricht erneut an den Server senden zu lassen.
Stimmt er diesem zu, wird in einer Schleife versucht eine erneute Verbindung zur RMI Registry und dem Server zu erhalten, um die nicht zugestellte Nachricht zuzustellen. Jede Sekunde des zuvor definierten Server Timeouts wird versucht eine Verbindung herzustellen. Scheitert diese, wird die Exception abgefangen und es wird für eine Sekunde gewartet, bevor ein neuer Verbindungsaufbau getätigt wird. Dieses wird solange wiederholt, bis der Server Timeout abgelaufen ist. In dieser Zeit wird der Button zum Senden deaktiviert und sein Text in den Countdown des Server Timeouts abgeändert, um dem User ein Feedback über die bereits verstrichene Zeit zu geben. Schlägt nach Ablauf des Server Timeouts das Verbinden immer noch fehl, wird die GUI zurückgesetzt.

Protokollierung

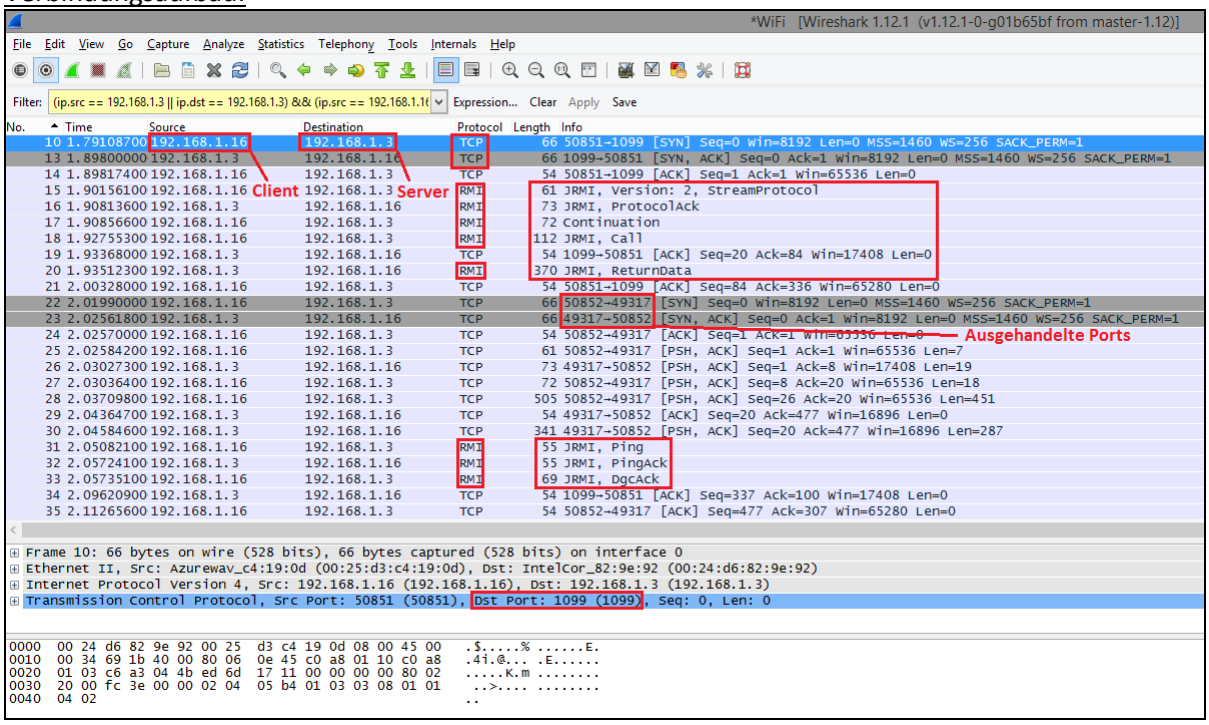
Test der Interface-Implementierungen zwischen den Praktikums-Gruppen:

Nachdem bei allen Teilnehmern ein Security-Manager eingerichtet war und alle die Policy-Dateien für die Zugriffsrechte korrekt eingebunden hatten, konnte zuerst noch keine Verbindung aufgebaut werden. Erst nachdem alle Teilnehmer einen gemeinsamen Namen für das Package der Class-Dateien definiert hatten, konnten sich die Clients mit dem Server verbinden.
Je nach Implementierung und Dauer der Ausfallzeit konnten sich Clients an einen zuvor beendeten Server wieder einwandfrei anmelden, wenn dieser wieder gestartet wurde.
Auch die Implementierung der Gedächtniszeit *t* und die damit verbundene at-most-one Fehlersemantik sorgten dafür, dass nur dann neue Nachrichten zugestellt wurden, wenn sie außerhalb der Gedächtniszeit lagen.
In diesem Zusammenhang konnten sich Beispielsweise mehrere Clients mit demselben Benutzernamen am Server anmelden und es bekam dann auch nur der Client die neuen Nachrichten, der den Empfang als erster eingeleitet hat.

Verteilte Systeme Praktikum WS2014
Teilnehmer: Michael Schmidt, Eugen Winter

Wireshark Paket-Analyse:

Verbindungsaufbau:



RMI Registry: 192.168.1.3
Server: 192.168.1.3
Client: 192.168.1.16

Der Verbindungsaufbau erfolgt in mehreren Schritten:

1. Der Client kontaktiert über das TCP die RMI Registry auf dem *well known* Port 1099
2. Der Client schickt nun via *JRMI* das gewünschte Protokoll samt Version an die RMI Registry
3. Die RMI Registry bestätigt via *JRMI ProtocolAck* das RMI Protokoll
4. Der Client startet einen *JRMI Call* mit dem Namen des Servers → *MessageService*
5. Die RMI Registry gibt dem Client mit *JRMI ReturnData* das RemoteObject samt Ziel-IP zurück
6. Nun kennt der Client die Server-IP und beide tauschen die TCP Ports und IP-Adressen für die Kommunikation aus. Anschließend kommunizieren beide noch die Konfigurationsparameter (DGC (*Distributed Garbage Collection*) Lease Time, Obj ID, Server UID, usw.)
7. Der Client prüft via *JRMI Ping*, ob der Server erreichbar ist
8. Der Server übermittelt seine Verfügbarkeit mit einem *JRMI PingAck*
9. Der Client sendet dem Server mit *JRMI DgcAck* einen *UniqueIdentifier* für dessen Distributed Garbage Collection
10. Die Pakete 34 + 35 bilden die finalen *ACK's*, danach ist die Verbindung zwischen Server und Client hergestellt

Verteilte Systeme Praktikum WS2014
Teilnehmer: Michael Schmidt, Eugen Winter

Problem-Identifizierung:

Während dem Testen der Interface-Implementierungen zwischen den Praktikums-Gruppen im Labor kam es wie oben beschrieben zu Verbindungsproblemen. An dieser Stelle lässt sich mit Hilfe des Paket-Sniffers auch feststellen wieso.

In Schritt 5 liefert die RMI Registry dem Client mit *JRMI ReturnData* das RemoteObject und die Ziel-IP des Servers zurück. Das serialisierte Datenpaket Nr. 20 hat dabei folgenden Inhalt:

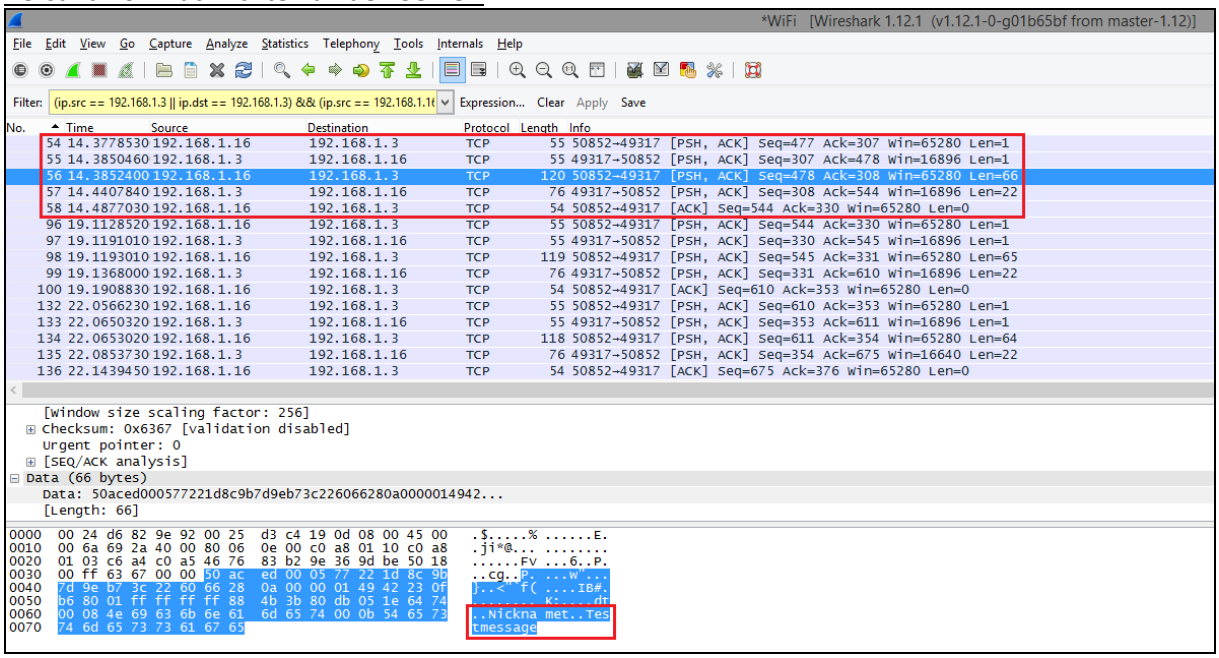
```
⊞ Frame 20: 370 bytes on wire (2960 bits), 370 bytes captured (2960 bits) on interface 0
⊞ Ethernet II, Src: IntelCor_82:9e:92 (00:24:d6:82:9e:92), Dst: Azurewav_c4:19:0d (00:25:d3:c4:19:0d)
⊞ Internet Protocol Version 4, Src: 192.168.1.3 (192.168.1.3), Dst: 192.168.1.16 (192.168.1.16)
⊞ Transmission Control Protocol, Src Port: 1099 (1099), Dst Port: 50851 (50851), Seq: 20, Ack: 84, Len: 316
⊟ Java RMI
    Input Stream Message: ReturnData (0x51)
    Serialization Data

0000  00 25 d3 c4 19 0d 00 24  d6 82 9e 92 08 00 45 00   .%.....$ ......E.
0010  01 64 7a 93 40 00 80 06  fb 9c c0 a8 01 03 c0 a8   .dz.@... ........
0020  01 10 04 4b c6 a3 5b 78  fc 6c ed 6d 17 65 50 18   ...K..[x .l.m.eP.
0030  00 44 e5 de 00 00 51 ac  ed 00 05 77 0f 01 60 66   .D....Q. ...w..`f
0040  28 0a 00 00 01 49 42 23  0f b6 80 05 73 7d 00 00   (....IB# ....s}..
0050  00 02 00 0f 6a 61 76 61  2e 72 6d 69 2e 52 65 6d   ....java .rmi.Rem
0060  6f 74 65 00 15 63 6f 6d  6d 6f 6e 2e 4d 65 73 73   ote..com mon.Mess
0070  61 67 65 53 65 72 76 69  63 65 70 78 72 00 17 6a   ageServi cepxr..j
0080  61 76 61 2e 6c 61 6e 67  2e 72 65 66 6c 65 63 74   ava.lang .reflect
0090  2e 50 72 6f 78 79 e1 27  da 20 cc 10 43 cb 02 00   .Proxy.' . ..C...
00a0  01 4c 00 01 68 74 00 25  4c 6a 61 76 61 2f 6c 61   .L..ht.% Ljava/la
00b0  6e 67 2f 72 65 66 6c 65  63 74 2f 49 6e 76 6f 63   ng/refle ct/Invoc
00c0  61 74 69 6f 6e 48 61 6e  64 6c 65 72 3b 70 78 70   ationHan dler;pxp
00d0  73 72 00 2d 6a 61 76 61  2e 72 6d 69 2e 73 65 72   sr.-java .rmi.ser
00e0  76 65 72 2e 52 65 6d 6f  74 65 4f 62 6a 65 63 74   ver.Remo teObject
00f0  49 6e 76 6f 63 61 74 69  6f 6e 48 61 6e 64 6c 65   Invocati onHandle
0100  72 00 00 00 00 00 00 00  02 02 00 00 70 78 72 00   r....... ....pxr.
0110  1c 6a 61 76 61 2e 72 6d  69 2e 73 65 72 76 65 72   .java.rm i.server
0120  2e 52 65 6d 6f 74 65 4f  62 6a 65 63 74 d3 61 b4   .RemoteO bject.a.
0130  91 0c 61 33 1e 03 00 00  70 78 70 77 34 00 0a 55   ..a3.... pxpw4..U
0140  6e 69 63 61 73 74 52 65  66 00 0b 31 39 32 2e 31   nicastRe f..192.1
0150  36 38 2e 31 2e 33 00 00  c0 a5 1d 8c 9b 7d 9e b7   68.1.3.. .....}..
0160  3c 22 60 66 28 0a 00 00  01 49 42 23 0f b6 80 01   <"`f(... .IB#....
0170  01 78                                             .x
```

Wie man sieht wird als *java.rmi.Remote* Objekt die Referenz *common.MessageService* an den Client übergeben. Das *common* steht dabei für den Package-Namen, in dem sich das Interface *MessageService* befindet. Wenn beim Client lokal ein anderer Package-Name verwendet wird, findet die Java VM das Interface nicht wieder und die Kommunikation schlägt fehl.

Deshalb konnte man mit der Einigung auf einen einheitlichen Package-Namen das Problem lösen.

Verteilte Systeme Praktikum WS2014
Teilnehmer: Michael Schmidt, Eugen Winter

Versand von Nachrichten an den Server:



Server: 192.168.1.3
Client: 192.168.1.16

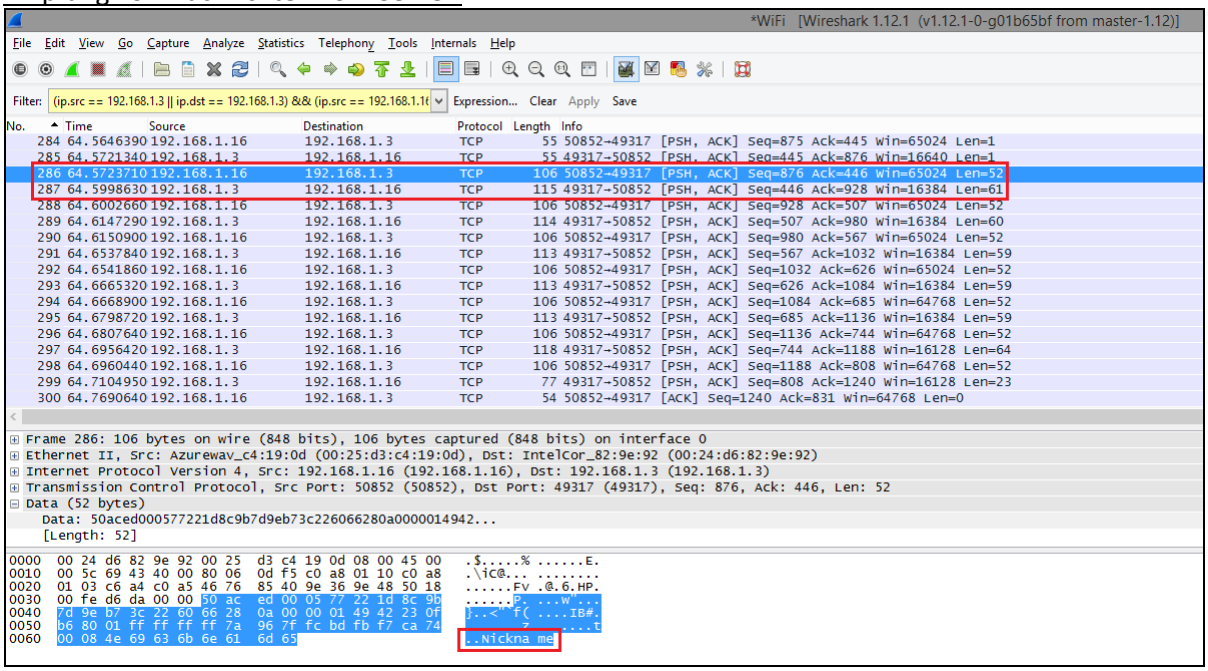
Der Versand von Nachrichten an den Server erfolgt in mehreren Schritten:

1. Client meldet den Versand beim Server über *[PSH, ACK]* an
2. Server bestätigt die Annahme über *[PSH, ACK]*
3. Die Nachricht wird vom Client an den Server übertragen
4. Der Server bestätigt den Empfang der Nachricht
5. Der Client bestätigt die Bestätigung des Servers über den Empfang der Nachricht

Wie man anhand der Daten im Paket sehen kann, sind die Inhalte der Nachricht einsehbar und unverschlüsselt. Im Bild zu sehen ist die Client-ID, mit der die Nachricht verschickt wurde: *Nickname*
Der Inhalt der Nachricht war: *Testmessage*

Verteilte Systeme Praktikum WS2014
Teilnehmer: Michael Schmidt, Eugen Winter

Empfang von Nachrichten vom Server:



Server: 192.168.1.3
Client: 192.168.1.16

Der Empfang von Nachrichten vom Server erfolgt in zwei Schritten:

1. Der Client schickt die Client-ID, für die die Nachrichten abgerufen werden sollen, dem Server
2. Der Server schickt, wenn vorhanden und noch nicht zugestellt, die passende Nachricht zurück

Auf dem Bild oben sieht man, dass der Client dem Server die Client-ID *Nickname* übermittelt, um dessen Nachrichten abzurufen.
Das Bild unten zeigt die Antwort des Servers mit der Nachricht zur passenden Client-ID *Nickname*:

[1] Nickname: Testmessage (14:31:18)