

## 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

<< interface >> MessageService extends Remote	
+ nextMessage(String clientID): String (throws RemoteException)	
+ newMessage(String clientID, String message): void (throws RemoteException)	

### Programmaufbau

#### Klassen:

Message.java

Message	
- df: DateFormat	
- clientsRequestsList: HashMap<String, Date>	
- <u>messageIDCounter</u> : int	
- nextMessage: Message	
- message: String	
- clientID: String	
- messageID: int	
- msgResendTime: int	
- date: Date	
+ Message(String clientID, String message, int msgResendTime)	
+ getValidMessage(String clientID): Message	
+ getNextMessage: Message	
+ getMessage: String	
+ getClientID: String	
+ getMessageID: int	
+ getMsgResendTime: int	
+ getDate: Date	
+ setNextMessage(Message): void	
+ toString: String	

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.

DeliveryQueue.java

<b>DeliveryQueue</b>
- 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 zurückgegeben werden und dabei nicht innerhalb der Gedächtniszeit  $t$  des jeweiligen Clients liegen.

MessageServiceServer.java

<b>MessageServiceServer extends UnicastRemoteObject implements MessageService</b>
- 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

<b>MessageServiceClient</b>
<ul style="list-style-type: none"><li>- <u>LOGFILE: File</u></li><li>- <u>stringBuffer: StringBuilder</u></li><li>- <u>messageService: MessageService</u></li><li>- <u>registry: Registry</u></li><li>- <u>messageToSend: String</u></li><li>- <u>serverAddr: String</u></li><li>- <u>serverName: String</u></li><li>- <u>clientID: String</u></li><li>- <u>serverTimeout: int</u></li><li>- <u>isResending: boolean</u></li><li>- <u>frame: JFrame</u></li><li>- <u>textServerAddr: JTextField</u></li><li>- <u>textServerName: JTextField</u></li><li>- <u>textClientID: JTextField</u></li><li>- <u>textServerTimeout: JTextField</u></li><li>- <u>textOutputArea: JTextArea</u></li><li>- <u>textInputArea: JTextArea</u></li><li>- <u>buttonConnect: JButton</u></li><li>- <u>buttonExit: JButton</u></li><li>- <u>buttonSend: JButton</u></li><li>- <u>buttonRecv: JButton</u></li></ul>
<ul style="list-style-type: none"><li>- <u>createAndShowGUI: void</u></li><li>- <u>writeLog(String message): void</u></li><li>- <u>connect: void</u></li><li>- <u>resetGUI: void</u></li><li>- <u>sendMessage: void</u></li><li>- <u>receiveMessage: void</u></li></ul>

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.

## Wireshark Paket-Analyse:

### Verbindungsaufbau:

No.	Time	Source	Destination	Protocol	Length	Info
10	1.79108700	192.168.1.16	192.168.1.3	TCP	66	50851->1099 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
13	1.89800000	192.168.1.3	192.168.1.16	TCP	66	1099->50851 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
14	1.89817400	192.168.1.16	192.168.1.3	TCP	54	50851->1099 [ACK] Seq=1 Ack=1 Win=65536 Len=0
15	1.90156100	192.168.1.16	192.168.1.3	RMI	61	JRMI, Version: 2, StreamProtocol
16	1.90813600	192.168.1.3	192.168.1.3	RMI	73	JRMI, ProtocolAck
17	1.90856600	192.168.1.16	192.168.1.3	RMI	72	Continuation
18	1.92755300	192.168.1.16	192.168.1.3	RMI	112	JRMI, call
19	1.93368000	192.168.1.3	192.168.1.16	TCP	54	1099->50851 [ACK] Seq=20 Ack=84 Win=17408 Len=0
20	1.93512300	192.168.1.3	192.168.1.16	RMI	370	JRMI, ReturnData
21	2.00328000	192.168.1.16	192.168.1.3	TCP	54	50851->1099 [ACK] Seq=84 Ack=336 Win=65280 Len=0
22	2.01990000	192.168.1.16	192.168.1.3	TCP	66	50852->49317 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
23	2.02561800	192.168.1.3	192.168.1.16	TCP	66	49317->50852 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
24	2.02570000	192.168.1.16	192.168.1.3	TCP	54	50852->49317 [ACK] Seq=1 Ack=1 Win=65536 Len=0
25	2.02584200	192.168.1.16	192.168.1.3	TCP	61	50852->49317 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=7
26	2.03027300	192.168.1.3	192.168.1.16	TCP	73	49317->50852 [PSH, ACK] Seq=1 Ack=8 Win=17408 Len=19
27	2.03036400	192.168.1.16	192.168.1.3	TCP	72	50852->49317 [PSH, ACK] Seq=8 Ack=20 Win=65536 Len=18
28	2.03709800	192.168.1.16	192.168.1.3	TCP	505	50852->49317 [PSH, ACK] Seq=26 Ack=20 Win=65536 Len=451
29	2.04364700	192.168.1.3	192.168.1.16	TCP	54	49317->50852 [ACK] Seq=20 Ack=477 Win=16896 Len=0
30	2.04584600	192.168.1.3	192.168.1.16	TCP	341	49317->50852 [PSH, ACK] Seq=20 Ack=477 Win=16896 Len=287
31	2.05082100	192.168.1.16	192.168.1.3	RMI	55	JRMI, Ping
32	2.05724100	192.168.1.3	192.168.1.16	RMI	55	JRMI, PingAck
33	2.05735100	192.168.1.16	192.168.1.3	RMI	69	JRMI, DgcAck
34	2.09620900	192.168.1.3	192.168.1.16	TCP	54	1099->50851 [ACK] Seq=337 Ack=100 Win=17408 Len=0
35	2.11265600	192.168.1.16	192.168.1.3	TCP	54	50852->49317 [ACK] Seq=477 Ack=307 Win=65280 Len=0

Frame 10: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0  
Ethernet II, Src: Azurewaw\_c4:19:0d (00:25:d3:c4:19:0d), Dst: IntelCor\_82:9e:92 (00:24:d6:82:9e:92)  
Internet Protocol Version 4, Src: 192.168.1.16 (192.168.1.16), Dst: 192.168.1.3 (192.168.1.3)  
Transmission Control Protocol, Src Port: 50851 (50851), Dst Port: 1099 (1099), Seq: 0, Len: 0

0000 00 24 d6 82 9e 92 00 25 d3 c4 19 0d 08 00 45 00 .\$. ....% .....E.  
0010 00 34 69 1b 40 00 80 06 0e 45 c0 a8 01 10 c0 a8 .4i@... ..E.....  
0020 01 03 c6 a3 04 4b ed 6d 17 11 00 00 00 00 80 02 .....K..m .....  
0030 20 00 fc 3e 00 00 02 04 05 b4 01 03 03 08 01 01 ...>.....  
0040 04 02 ..

**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 *UniquelIdentifier* für dessen Distributed Garbage Collection
10. Die Pakete 34 + 35 bilden die finalen *ACK's*, danach ist die Verbindung zwischen Server und Client hergestellt

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.... pwp4..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.

Versand von Nachrichten an den Server:

No.	Time	Source	Destination	Protocol	Length	Info
54	14.3778530	192.168.1.16	192.168.1.3	TCP	55	50852->49317 [PSH, ACK] Seq=477 Ack=307 win=65280 Len=1
55	14.3850460	192.168.1.3	192.168.1.16	TCP	55	49317->50852 [PSH, ACK] Seq=307 Ack=478 win=16896 Len=1
56	14.3852400	192.168.1.16	192.168.1.3	TCP	120	50852->49317 [PSH, ACK] Seq=478 Ack=308 win=65280 Len=66
57	14.4407840	192.168.1.3	192.168.1.16	TCP	76	49317->50852 [PSH, ACK] Seq=308 Ack=544 win=16896 Len=22
58	14.4877030	192.168.1.16	192.168.1.3	TCP	54	50852->49317 [ACK] Seq=544 Ack=330 win=65280 Len=0
96	19.1128520	192.168.1.16	192.168.1.3	TCP	55	50852->49317 [PSH, ACK] Seq=544 Ack=330 win=65280 Len=1
97	19.1191010	192.168.1.3	192.168.1.16	TCP	55	49317->50852 [PSH, ACK] Seq=330 Ack=545 win=16896 Len=1
98	19.1193010	192.168.1.16	192.168.1.3	TCP	119	50852->49317 [PSH, ACK] Seq=545 Ack=331 win=65280 Len=65
99	19.1368000	192.168.1.3	192.168.1.16	TCP	76	49317->50852 [PSH, ACK] Seq=331 Ack=610 win=16896 Len=22
100	19.1908830	192.168.1.16	192.168.1.3	TCP	54	50852->49317 [ACK] Seq=610 Ack=353 win=65280 Len=0
132	22.0566230	192.168.1.16	192.168.1.3	TCP	55	50852->49317 [PSH, ACK] Seq=610 Ack=353 win=65280 Len=1
133	22.0650320	192.168.1.3	192.168.1.16	TCP	55	49317->50852 [PSH, ACK] Seq=353 Ack=611 win=16896 Len=1
134	22.0653020	192.168.1.16	192.168.1.3	TCP	118	50852->49317 [PSH, ACK] Seq=611 Ack=354 win=65280 Len=64
135	22.0853730	192.168.1.3	192.168.1.16	TCP	76	49317->50852 [PSH, ACK] Seq=354 Ack=675 win=16640 Len=22
136	22.1439450	192.168.1.16	192.168.1.3	TCP	54	50852->49317 [ACK] Seq=675 Ack=376 win=65280 Len=0

[window size scaling factor: 256]  
Checksum: 0x6367 [validation disabled]  
Urgent pointer: 0  
[SEQ/ACK analysis]  
Data (66 bytes)  
Data: 50aced000577221d8c9b7d9eb73c226066280a0000014942...  
[Length: 66]  
0000 00 24 d6 82 9e 92 00 25 d3 c4 19 0d 08 00 45 00 .\$. ....% .....E.  
0010 00 6a 69 2a 40 00 80 06 0e 00 c0 a8 01 10 c0 a8 .ji\*@... ..  
0020 01 03 c6 a4 c0 a5 46 76 83 b2 9e 36 9d be 50 18 .....FV...6..P.  
0030 00 ff 63 67 00 00 50 ac ed 00 05 77 22 1d 8c 9b ..cg..P...w'...  
0040 7d 0e d7 3c 22 60 66 28 0a 00 00 01 49 42 23 9f j..< "f( ....IB#.  
0050 b6 80 01 ff ff ff ff 88 4b 3b 80 db 05 1e 64 74 ..<...kz...dt  
0060 00 08 4e 69 63 6b 6e 61 6d 65 74 00 0b 34 65 73 ..Nickna met..Tes  
0070 74 6d 65 73 73 61 67 65 tmessage

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*



## Empfang von Nachrichten vom Server:

Filter: (ip.src == 192.168.1.3 || ip.dst == 192.168.1.3) && (ip.src == 192.168.1.16)

No.	Time	Source	Destination	Protocol	Length	Info
284	64.5646390	192.168.1.16	192.168.1.3	TCP	55	50852→49317 [PSH, ACK] Seq=875 Ack=445 win=65024 Len=1
285	64.5721340	192.168.1.3	192.168.1.16	TCP	55	49317→50852 [PSH, ACK] Seq=445 Ack=876 win=16640 Len=1
286	64.5723710	192.168.1.16	192.168.1.3	TCP	106	50852→49317 [PSH, ACK] Seq=876 Ack=446 win=65024 Len=52
287	64.5998630	192.168.1.3	192.168.1.16	TCP	115	49317→50852 [PSH, ACK] Seq=446 Ack=928 win=16384 Len=61

Frame 286: 106 bytes on wire (848 bits), 106 bytes captured (848 bits) on interface 0  
Ethernet II, Src: Azurewav\_c4:19:0d (00:25:d3:c4:19:0d), Dst: IntelCor\_82:9e:92 (00:24:d6:82:9e:92)  
Internet Protocol Version 4, Src: 192.168.1.16 (192.168.1.16), Dst: 192.168.1.3 (192.168.1.3)  
Transmission Control Protocol, Src Port: 50852 (50852), Dst Port: 49317 (49317), Seq: 876, Ack: 446, Len: 52  
Data (52 bytes)  
Data: 50aced000577221d8c9b7d9eb73c226066280a0000014942...  
[Length: 52]  
0000 00 24 d6 82 9e 92 00 25 d3 c4 19 0d 08 00 45 00 .\$. ....% .....E.  
0001 00 5c 69 43 40 00 80 06 d5 f5 c0 a8 01 10 c0 a8 .\ic@... ..  
0002 01 03 c6 a4 c0 a5 46 76 85 40 9e 36 9e 48 50 18 .....6.HFV.tP.  
0003 00 fe d6 da 00 50 ac ed 00 05 77 0f 01 60 66 .@...Q...w...f  
0004 74 9e b7 3c 22 60 66 28 0a 00 00 01 49 42 23 0f ...< f( ...IB#...t.s  
0005 b6 80 01 ff ff ff ff 7a 96 7f fc bd fb f7 ca 74 .....7 .....t  
0006 00 08 4e 69 63 6b 6e 61 6d 65 .....  
0007 31 38 29

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)

Filter: (ip.src == 192.168.1.3 || ip.dst == 192.168.1.3) && (ip.src == 192.168.1.16)

No.	Time	Source	Destination	Protocol	Length	Info
284	64.5646390	192.168.1.16	192.168.1.3	TCP	55	50852→49317 [PSH, ACK] Seq=875 Ack=445 win=65024 Len=1
285	64.5721340	192.168.1.3	192.168.1.16	TCP	55	49317→50852 [PSH, ACK] Seq=445 Ack=876 win=16640 Len=1
286	64.5723710	192.168.1.16	192.168.1.3	TCP	106	50852→49317 [PSH, ACK] Seq=876 Ack=446 win=65024 Len=52
287	64.5998630	192.168.1.3	192.168.1.16	TCP	115	49317→50852 [PSH, ACK] Seq=446 Ack=928 win=16384 Len=61

Frame 287: 115 bytes on wire (920 bits), 115 bytes captured (920 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: 49317 (49317), Dst Port: 50852 (50852), Seq: 446, Ack: 928, Len: 61  
Data (61 bytes)  
Data: 51aced0005770f016066280a0000014942230fb6800d7400...  
[Length: 61]  
0000 00 25 d3 c4 19 0d 00 24 d6 82 9e 92 08 00 45 00 .\$. ....\$ .....E.  
0001 00 65 7a a9 40 00 80 06 fc 85 c0 a8 01 03 c0 a8 .eZ@... ..  
0002 00 10 c0 a5 c6 a4 9e 36 9e 48 46 76 85 74 50 18 .....6.HFV.tP.  
0003 00 40 16 80 00 00 51 ac ed 00 05 77 0f 01 60 66 .@...Q...w...f  
0004 28 0a 00 00 01 49 42 23 0f b6 80 0d 74 00 24 5b .....IB#...t.s  
0005 31 5d 20 4e 69 63 6b 6e 61 6d 65 3a 20 54 65 73 ...Nickn ame: Tes  
0006 74 6d 65 73 73 61 67 65 20 28 31 34 3a 33 31 3a tmessage (14:31:  
0007 31 38 29 18)