Prof. Dr. Thomas Schmidt
HAW Hamburg, Dept. Informatik,
Raum 480a, Tel.: 42875 - 8452
Email: t.schmidt@haw-hamburg.de
Web: http://inet.haw-hamburg.de/teaching

Verteilte Systeme

Aufgabe 1: Client/Server-Anwendung "Verteilte Nachrichten-Queue"

Ziele:

1. Java RMI kennen und verstehen lernen
2. Eine interoperable Client-Server-Anwendung erstellen
3. Fehlersemantiken anwenden und Fehlertoleranzen implementieren

Vorbemerkungen:

In dieser Aufgabe wird eine chatgruppenartige Anwendung entwickelt: Clients tauschen über eine 'standardisierte' RMI-Schnittstelle, hinter welcher sich ein Reflektor-Server befindet, Nachrichten aus. Entscheidende Merkmale dieses kleinen verteilten Systems sind die Verteilungstransparenz, die Offenheit (Interoperabilität) und die Fehlertoleranz. Entsprechend kommt es vor allem auf die sorgfältige Umsetzung und das Testen dieser Kernmerkmale an.

Aufgabenstellung:

Entwickeln Sie ein Client/Server-Paar gemäß nachfolgenden Spezifikationen, welches auf beliebigen unterschiedlichen Laborrechnern über JAVA/RMI miteinander kommunizieren kann. Hierfür

1. konzipieren Sie zunächst den Programmaufbau, insbesondere die Verteilungskomponenten und ihre Fehlersemantiken;
2. implementieren Sie hiernach Client und Server unter strikter Einhaltung der Kommunikationsschnittstelle;
3. testen Sie dann Ihre Programme, insbesondere auf Interoperabilität mit Nachbargruppen;
4. untersuchen Sie die RMI-Kommunikation mit dem Netzwerk-Sniffer, dokumentieren Ihre Beobachtungen und begründeten Interpretationen in einem Protokoll.

Spezifikation:

Server:

1. Der Server nimmt über seine entfernte Methode `newMessage()` Nachrichten von Redakteur-Clients entgegen, speichert diese zusammen mit einer fortlaufenden Nachrichten-ID und dem Empfangszeitpunkt in seiner Delivery-Queue.
2. Die Delivery-Queue hat eine konfigurierbar begrenzte Länge *n*, so dass jeweils die neuesten *n* Nachrichten gehalten werden. Die Verdrängung operiert nach dem FIFO-Prinzip.
3. Clients können über die entfernte Methode `nextMessage()` Nachrichten-Strings abrufen, wobei sich der Server für eine begrenzte Zeit *t* merkt, welche Nachricht zuletzt an den jeweiligen Client ausgeliefert wurde. Ausgelieferte Nachrichten-Strings haben die Form `<clientID>"-"<messageID>":"<message>", "<timestamp>,` wobei die `clientID` als String `<userID>"@"<clientHostname>` definiert wird.
   Liegen keine Nachrichten für den Client mehr vor, wird `null` zurückgegeben.
4. Die Gedächtniszeit *t* wird für jeden Client individuell gehalten und stets aufgefrischt, wenn ein Nachrichtenabruf des Clients geschieht. Nach Ablauf der Gedächtniszeit wird der Client vollständig vergessen.
5. Der Server implementiert eine 'At-most-once' Fehlersemantik in der Nachrichtenauslieferung.
6. Der Server bleibt robust gegen Client-Fehlverhalten und protokolliert seine Aktionen in einem aussagekräftigen Logfile.
7. Seine entfernte Schnittstelle registriert der Server unter dem Namen "MessageService".

Schnittstelle:

Client und Server kommunizieren über folgendes Remote Interface:

```
public interface MessageService extends Remote {

// Requests the next message for `clientID`.
 public String nextMessage(String clientID) throws
                                          RemoteException;
 // Posts a new `message` from `clientID` into the chat.
 public void newMessage(String clientID, String message) throws
                                          RemoteException;
}
```

Client:

1. Der Client verfügt über zwei Funktionsbereiche: die Redaktion und die Rezeption. Beide Funktionen sind Bestandteil einer GUI, in welcher auch der korrespondierende Server konfiguriert werden kann.
2. Als Redakteur-Client werden Nachrichten mit der entfernten Methode `newMessage()` an den Server gesandt. Hierbei implementiert der Client eine 'At-least-once' Fehlersemantik in der Nachrichtenzustellung.
3. Als Lese-Client werden Nachrichten mit der entfernten Methode `nextMessage()` von dem Server geholt, wobei der Client einen Modus bereitstellt, in welchem stets alle verfügbaren Messages abgeholt und angezeigt werden. Hierbei implementiert der Client eine 'Maybe' Fehlersemantik im Nachrichtenabruf.
4. Der Client bleibt robust gegen Serverausfälle während eines konfigurierbaren Toleranzintervalls von *s* Sekunden.

Abgabe (Modalitäten wie in der Vorlesungseinführung angegeben):

Vor dem Praktikumstermin:

1. Konzeption von Programmaufbau und Verteilungskomponenten mit Fehlersemantik

Nach dem Praktikumstermin:

2. Überarbeitetes Konzept mit Implementierungsdokumentation und wohlbegründetem Versuchsprotokoll
3. Dokumentierter Programm-Code

Um die Interoperabilität zwischen den Gruppen zu ermöglichen, empfiehlt es sich die Sourcen in das Defaultpackage zu packen.

Online-Dokumentation zu Java RMI:

- http://docs.oracle.com/javase/tutorial/rmi/overview.html
- http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

Achten Sie besonders auf die erforderlichen Sicherheitseinstellungen für die RMI Registry!