Netzwerk Paket

Beschreibung, Verwendung und zusätzliche Informationen

Max Brauer

22.03.2016

Dieses Paket ist verantwortlich für die Verbindung zweier Computer über das Netzwerk. Es verwaltet alle Verbindungen und kümmert sich um den Datenaustausch.

1. Inhaltsangabe

2. Was kann dieses Paket?

- Verbindung mit anderen Computern aufnehmen (Client-Rolle)
- Verbindungen von anderen Computern annehmen (Server-Rolle)
- Bestehende Verbindungen verwalten und pflegen
- Automatische Verarbeitung von Verbindungsabbrüchen
- Ermittlung eines Pings (wird sekündlich aktualisiert)
- Schnelles wiederfinden der Zieladressen über eine eindeutige lokale ID
- Schneller Datenaustausch und –verarbeitung über das interne Nachrichtenformat
- Geringe Verzögerung bei der Verarbeitung von Daten (max. 10 Millisekunden)

3. Was kann diesem Paket hinzugefügt werden?

3.1. Gruppierung von Nutzern

Das verwaltungstechnische Zusammenfassen von Nutzern zu Gruppen, damit diese schneller angesprochen werden können. Dazu sollen beliebig viele Gruppen erstellbar sein und die Nutzer sollen auch beliebig vielen Gruppen zugeordnet werden können.

3.2. Zentral einheitliches Nutzerobjekt

Bis jetzt wird nur ein internes Nutzerobjekt erstellt, welches die Informationen über die Zielverbindung speichert. Von außen kann es nur über die interne ID angesprochen werden. Es besteht die Möglichkeit das System soweit zu öffnen, dass ein einheitliches Objekt für einen Nutzer erstellt wird, welches alle Informationen zu einem Nutzer speichert. Das spart Zeit beim Heraussuchen der Objekte über eine ID, da hier gleich die Objekte mit den notwenigen Informationen vorhanden sind.

4. Klassenbeschreibung

Hier werden nur die Klassen und Methoden beschrieben, die auch von außen benötigt werden.

4.1. ConnectionHandler

Diese Klasse stellt Methoden bereit, mit denen man die Verwaltung steuern kann.

4.1.1. ConnectionHandlerEvent events

(Siehe 4.2. ConnectionHandlerEvent)

4.1.2. void initAsClient(String ip, int port)

Initialisiert das Netzwerkmodul als Client und nimmt eine Verbindung mit den Server auf. Wenn die Verbindung fehlschlägt, so wird das Event unter 4.2.3. ausgelöst.

4.1.3. void initAsServer(int port)

Initialisiert das Netzwerkmodul als Server und nimmt alle Anfragen unter dem angegebenen Port an. Wenn die Initialisierung fehlschlägt (z.B. Port ist belegt), so wird das Event unter 4.2.3. ausgelöst.

4.1.4. void close()

Beendet das Netzwerkmodul und schließt alle Verbindungen. Wenn das Modul noch nicht initialisiert oder schon beendet wurde, dann passiert nichts.

4.1.5. boolean isConnected()

Gibt zurück, ob der Server oder der Client vollständig initialisiert werden konnte. Wenn die Rolle eines Servers inne ist, so muss keine Verbindung bestehen. Es reicht lediglich aus, wenn der Socket bereitgestellt werden konnte. Wenn stattdessen die Rolle eines Client inne ist, so gibt es auf jedem Fall zurück, ob die Verbindung mit dem Server hergestellt werden konnte.

4.2. ConnectionHandlerEvent

Dies ist ein Interface, welches Methoden bereithält, damit das Netzwerkmodul auch nach außen hin kommunizieren kann. Damit es verwendbar wird, muss eine Klasse, welches dieses Interface implementiert, instanziiert und an 4.1.1. gebunden werden.

4.2.1. void connectionAdded(long userId)

Diese Methode wird immer dann ausgelöst, wenn eine Verbindung mit einem Remotecomputer hergestellt werden konnte. Dabei ist es egal welche Rolle inne ist. Mitgegeben wird immer eine ID, mit der man den Nutzer später schneller wieder ansprechen kann. Wenn die eigene Rolle ein Client ist, so beträgt ID immer 0.

4.2.2. void connectionClosed(long userId)

Diese Methode wird immer dann ausgelöst, wenn die Verbindung mit einem Remotecomputer getrennt wurde. Dabei ist es egal welche Rolle inne ist. Mitgegeben wird immer eine ID, die den Nutzer repräsentiert. Wenn die eigene Rolle ein Client ist, so beträgt ID immer 0.

4.2.3. void connectionFailed()

Diese Methode wird immer dann ausgelöst, wenn die Verbindungsaufnahme fehlgeschlagen ist. Wenn die eigene Rolle ein Server ist, so liegt es daran, dass der Server nicht bereitgestellt werden konnte (z.B. Port belegt, Firewall blockiert, …). Wenn die eigene Rolle ein Client ist, so liegt es daran, dass die Verbindung mit dem Server nicht hergestellt werden konnte (z.B. keine Verbindung mit dem Computer, Firewall blockiert, …).

4.3. Helper

Diese Klasse stellt Hilfsmethoden bereit, mit denen man schneller arbeiten kann.

4.3.1. byte[] intToByteArray(int value)

Wandelt eine Integer-Zahl (32 Bit) in ein Byte-Array mit 4 Byte um.

4.3.2. int byteArrayToInt(byte[] value)

Wandelt die ersten 4 Byte aus dem Byte-Array in eine Integer-Zahl (32 Bit) um.

4.3.3. int byteArrayToInt(byte[] value, int offset)

Wandelt 4 Bytes ab einer bestimmten Stelle aus dem Byte-Array in eine Integer-Zahl (32 Bit) um.

4.3.4. byte[] longToByteArray(long value)

Wandelt eine Long-Zahl (64 Bit) in ein Byte-Array mit 8 Byte um.

4.3.5. long byteArrayToLong(byte[] value)

Wandelt die ersten 8 Byte aus dem Byte-Array in eine Long-Zahl (64 Bit) um.

4.3.6. long byteArrayToLong(byte[] value, int offset)

Wandelt 8 Bytes ab einer bestimmten Stelle aus dem Byte-Array in eine Long-Zahl (64 Bit) um.

4.3.7. void byteCopyTo(byte[] source, byte[] target, int sourceIndex, int targetIndex, int length)

Übertragt eine bestimmte Anzahl an Bytes ab einer bestimmten Stelle aus einem Quell-Array in ein Ziel-Array ab einer bestimmten Stelle.

4.4. NetworkInput

Schnittstelle für alle empfangenen Nachrichten. Dazu muss lediglich die Klasse abgeleitet und davon die Instanz an 4.4.1. gebunden werden.

4.4.1. NetworkInput networkInput

Die globale Variable an der eine Instanz der Klasse gebunden wird. Alle Nachrichten werden dann an diese Instanz weitergeleitet.

4.4.2. void MessageReceived(PrimaryMessage message)

Diese Methode wird immer dann ausgelöst, wenn eine Nachricht eingegangen ist. Die Nachricht wird gleich mitgeliefert.

4.5. NetworkOutput

Diese Klasse stellt eine Methode bereit, über diese man Nachrichten nach außen verschicken kann.

4.5.1. void SendMessage(PrimaryMessage message)

Diese Methode sendet eine Nachricht ab. Alle notwenigen Informationen dafür befinden sich im Nachrichtenobjekt.

4.6. iLoadSaveAble

Diese Schnittstelle definiert Methoden, über diese man ein Objekt mit komprimierten Daten füllen oder diese abrufen kann. Mehr zu diesem System unter 5.3.

4.6.1. void load(byte[] data)

Das Objekt, welches dieses Interface implementiert, befüllt sich selbst mit Informationen mithilfe der komprimierten Daten des Byte-Arrays.

4.6.2. byte[] save()

Das Objekt, welches dieses Interface implementiert, komprimiert seine Informationen und gibt diese als Byte-Array aus.

4.7. Message

Diese Klasse fasst alle Informationen einer Nachricht zusammen. Sie enthält unter anderen Informationen über dem Remotecomputer (4.7.6), dem Grund (4.7.3.), die Daten, sowie ein paar Metainformationen (4.7.5.) für die Ableitungen der Klasse.
Sie kann Informationen in folgenden Varianten abspeichern:

- Rohe Byte-Array Daten
- Serialisierte Objekt Daten (nicht empfohlen, da viele Meta-Informationen)
- Komprimierte Daten (iLoadSaveAble, *siehe 4.6.*)
- Nachrichten – es ist möglich Nachrichten in Nachrichten abzulegen und wieder abzurufen.

Wichtig: Die Speicherformate können nicht beliebig gemischt werden. Wenn z.B. ein Byte-Array gespeichert wurde, kann es nicht zu einem Objekt serialisiert werden.

4.7.1. void load(byte[] data)

Diese Methode liest alle Informationen aus dem Byte-Array und befüllt sich damit. *(siehe 4.6.1.)*

4.7.2. byte[] save()

Diese Methode speichert alle Informationen in ein Byte-Array und gibt es zurück. *(siehe 4.6.2.)*

4.7.3. [get/set] Reason (int)

Gibt den Grund bzw. Zweck der Nachricht an. Damit ist es schneller möglich die Nachricht einem Ziel zuzuordnen. Ableitungen der Klassen können dieses Feld mithilfe eines Enumerators füllen.

4.7.4. MessageDataType getDataType()

Gibt den Typ des Nachrichteninhalts zurück.

4.7.5. [get/set] Header (byte[])

Speichert zusätzliche Informationen in den Header der Nachricht. Es können maximal 127 Bytes (Byte.MAX_VALUE) hier abgelegt werden. Es ist vor allem als Speicherplatz für abgeleitete Klassen gedacht, die extra Felder definieren wollen.

4.7.6. [get/set] SenderId (long)

Speichert die ID, an dem die Nachricht gesendet werden soll, oder von wem sie empfangen wurde.

4.7.7. [get/set] DataAsByteArray (byte[])

Liest oder speichert die Daten als Byte-Array.

4.7.8. [get/set] DataAsObject (<T>)

Serialisiert oder Deserialisert ein beliebiges Objekt in seine Daten. Dies kann theoretisch mit jedem Objekt gemacht werden. Der Nachteil liegt darin, dass diese Methode sehr langsam im Vergleich zu den anderen Optionen ist.

4.7.9. [get/set] DataAsLoadSaveAble (<T>)

Speichert ein Objekt in seine komprimierten Daten. Diese Methode ist im Bezug zu 4.7.8. Ressourcenschonend und Zeitsparend. Die Typinformationen werden aus dem generischen Aufruf geholt und nicht gespeichert.
Es können nur Typen die iLoadSaveAble (4.6.) implementieren verwendet werden.

4.7.10. [get/set] DataAsMessage (<T>)

Vergleichbar mit 4.7.9., da ein Nachrichtenobjekt immer iLoadSaveAble implementiert. Der Unterschied liegt darin, dass beim Abruf die Senderinformationen (4.7.6.) gleich mit übernommen werden. Dennoch ist es nicht möglich die Methoden 4.7.9. und 4.7.10. zu mischen.

4.8. MessageDataType

Gibt den Speichertyp der gespeicherten Informationen eines Nachrichtenobjekts (4.7.) an.

4.8.1. BinaryData

Die Daten wurden als Byte-Array gespeichert und können nur als solches abgerufen werden.

4.8.2. Object

Die Daten wurden als serialisiertes Objekt gespeichert und können nur als solches abgerufen werden.

4.8.3. ILoadSaveAble

Die Daten wurden komprimiert ohne extra Informationen gespeichert.

4.8.4. Message

Die Daten wurden komprimiert ohne extra Informationen gespeichert. Die Daten sind immer eine Nachricht.

4.9. PrimaryMessage

Von Message (4.7.) abgeleitet und vereinfacht den Umgang mit diesem. Dieser Nachrichtentyp wird vor allem vom Netzwerksystem und den äußeren Basissysteme verwendet.

4.9.1 [get/set] Type (PrimaryMessageType)

Gibt den Typ der Nachricht zurück oder setzt diesen. Dazu wird Reason (4.7.3.) verwendet.

4.10. PrimaryMessageType

Bestimmt den Typ der Nachricht *(siehe 4.9.)*.
Hinweis: Die Liste ist nicht vollständig uns muss noch erweitert werden.

4.10.1. None

Die Nachricht ist keinem Typ zugeordnet und kann sofort verworfen werden. (Vermutlich ist sie fehlerhaft.)

4.10.2. Ping

Die Nachricht wird als Ping vom Netzwerk-Modul verwendet. Von einer Verwendung außerhalb wird abgeraten.

4.10.3. PingAnswer

Die Nachricht wird als Ping vom Netzwerk-Modul verwendet. Von einer Verwendung außerhalb wird abgeraten.

4.10.4. RawText

Diese Nachricht ist für den Debug gedacht und enthält als Daten nur ein Byte-Array, welches einen String repräsentiert. Von einer Verwendung im live-System wird abgeraten.

5. Hinweise

5.1. Einbindung

Wichtig ist, dass die Schnittstellen ConnectionHandlerEvent (4.2.) und NetworkInput (4.4.) eine Implementierung erhalten, die dann an den jeweiligen Stellen abgelegt werden müssen. Diese stellen die Kommunikationswege des Netzwerksmoduls nach außen hin dar.

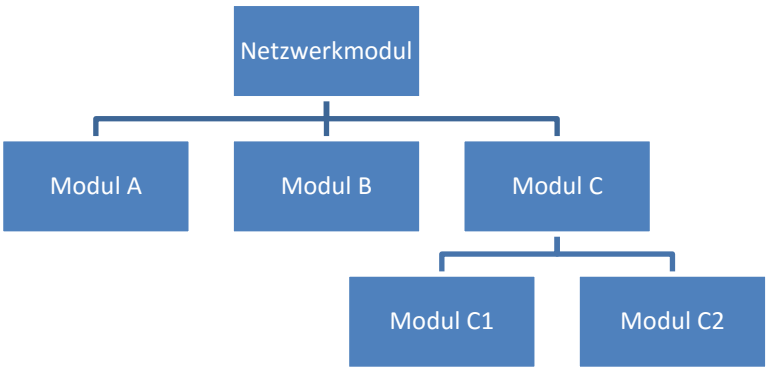
Für den Start reicht es aus das Netzwerkmodul zu initialisieren (4.1.2. oder 4.1.3.). Den Rest übernimmt das Modul selber. Für einen erfolgreichen Start ist es unumgänglichen alle Schnittstellen vorher bereitgestellt zu haben.

Zum Beenden reicht es aus, über die Verwaltung das Modul zu beenden (4.1.4.).

5.2. Absenden einer Nachricht

Zum Absenden einer Nachricht, erstellt man ein Objekt vom Typ Message (4.7.) oder eines seiner Ableitungen. Diese kann man dann bequem mit den Daten füllen, die man übermitteln will. Für den Empfänger muss nur noch seine ID unter SenderId (4.7.6.) abgelegt werden.
Enthält die Nachricht nun alles, was sie benötigt, so muss sie einfach an 4.5.1. übermittelt werden.

Durch die Schachtelung von Nachrichten kann man die eigene Softwarearchitektur nachbilden und so den Datentransport vereinfachen:



5.3. Vor- und Nachteile der Datenformate einer Nachricht

	Binär	Objekt	iLoadSaveAble	Message
Komfort beim Speichern.	Daten müssen vorher in das binäre Format übertragen werden.	Daten können direkt eingefügt werden.		
Komfort beim Laden.	Daten müssen nachträglich verarbeitet werden.	Objekt liegt sofort im Typ vor, der beim generischen Aufruf angegeben wurde.		
Extra Codeaufwand	Verarbeitung der Daten vor dem Speichern und nach dem Laden	Keiner	Programmieren der Load/Save-Methoden vom Interface	Keiner
Datenmenge	gering	Hoch, da Typ- und Metainformationen gespeichert werden.	gering	
Geschwindigkeit	Relativ hoch	Relativ gering	Relativ hoch	

Der große Nachteil beim Speichern der Daten im Objekt-Format liegt darin, dass beim Serialisierungsvorgang alle Typ- und Metainformationen gespeichert werden. Dadurch entstehen mehr Bytes, die übertragen werden müssen. Wenn die Informationen an der Gegenstelle angekommen sind, werden beim Deserialisierungsvorgang die Typ- und Metainformationen ausgelesen und anhand dessen werden die Objekte gesucht und aufgebaut. Dabei können auch Fehler entstehen (z.B. Objektinformationen existieren nicht, Objekt enthielt Daten, die nicht gespeichert wurden, …). Erst wenn das Objekt fertig ist, wird es in den Typ gecastet und zurückgegeben.
Hier geht nicht nur viel Zeit verloren, sondern auch die Fehleranfälligkeit ist deutlich höher.

Deshalb wird auch die Verwendung mit dem iLoadSaveAble (4.6.) empfohlen, da hier genau festgelegt werden kann, was wo und wie gespeichert werden muss. Hierbei kann man gleichzeitig die Daten komprimiert ablegen. Die Typinformationen der Objekte werden nicht mitgespeichert, sondern erst an der Zielstelle aus dem generischen Aufruf gewonnen. Dadurch müssen die Typinformationen nicht erst vom System gesucht werden, sondern sind sofort vorhanden. Außerdem kann man hier die Daten fehlertolleranter auslesen.

5.4. Ableitungen von Message

Es ist jederzeit möglich eine Ableitung von Message (4.7.) zu machen. Wenn dies gemacht wird, so dürfen die Ableitungen keine extra Datenfelder anlegen, sondern sollen die Felder aus dem Header-Bereich (4.7.5.) verwenden. So haben alle Message-Objekte und ihre Ableitungen dieselbe Informationsbasis und können beliebig verwendet werden. Es wird dringend davon abgeraten die load- und save-Methode (4.7.1. und 4.7.2) zu überschreiben.
Ableitungen dienen nur einem Zweck: Die Vereinfachung des Zugriffes auf die Daten.

6. Informationen

Quellcode:

Stand:	22.03.2016
Programmierer:	Max Brauer

Dokumentation:

Stand:	22.03.2016
Autor:	Max Brauer