Verteilte Systeme Übung 2

Alexander Steen , Max Wisniewski

Tutor : Philipp Schmidt

Unsere Implementierung benutzt für jedes Socket einen eigenen Thread. Dies ist keine besonders schöne Lösung und wird gerade bei hoher Belastung in die Knie gehen, aber für die einfachen Fälle, die wir hier behandelt haben, ist es völlig aureichend.

Kern dieser Lösung ist die Klasse ChannelFactory. In dieser erzeugen wir, wenn wir einen neuen Channel anfordern, zugleich einen *MessageDispatcher*, der auf das Socket aufpasst.

Die private Methode ist synchronized, damit wir nicht ausversehen zwei Channels auf dem selben Port öffnen und da es gleichzeitig passiert zwei Dispatcher für diesen Socket anlegen.

```
1  private static synchronized UdpChannel createChannel(int local_port){
2                  UdpChannel channel;
3                  if (dispatcher.containsKey(local_port)) {
4                          // port already exists, add new channel to according dispatcher
5                          channel = new UdpChannel(dispatcher.get(local_port).socket);
6                          MessageDispatcher m = dispatcher.get(local_port);
7                          m.addUnconnectedChannel(channel);
8                  } else {
9                          // channel on new port, so create new dispatcher and socket
10                         DatagramSocket newSock;
11                         try {
12                                 newSock = new DatagramSocket(local_port);
13                                 channel = new UdpChannel(newSock);
14                                 MessageDispatcher m = new MessageDispatcher(newSock);
15                                 dispatcher.put(local_port, m);
16                                 m.start();
17                         } catch (SocketException e) {
18                                 System.err.println("Socketcreation failed on port "
19                                                 + local_port + "\n");
20                                 return null;
21                         }
22                 }
23                 return channel;
24         }
```

Haben wir (1. Fall) schon einen *Dispatcher*, der auf den Socket hört, so holen wir diesen uns aus einem Wörterbuch (das wir uns gespeichert haben) und stecken den neuen Channel in den *Dispatcher* hinein.

Haben wir noch keinen Dispatcher für den Socket, so legen wir uns einen neuen an und starten ihn.

Da beide Factory Methoden anfänglich das gleiche tun (Channel anlegen und Dispatcher erzeugen/ermitteln) haben wir beides in eine private Methode ausgelagert. Ist der Channel nur mit *Port* erzeugt worden, so stecken wir ihn danach in die *unconnected* List rein. Haben wir den Endpunkt mitgegeben, stecken wir es in die verbundenen Endpunkte rein.

Der *Dispatcher* bekomt wie oben zu sehen ein Socket, auf das er achten muss. Nun fängt er an auf dem Socket zu lauschen.

```
this.socket.receive(incoming);
InetSocketAddress from = new InetSocketAddress(incoming
        .getAddress(), incoming.getPort());

if (!this.endpoints.containsKey(from)
        && this.unconnectedChannels.isEmpty()) {
                continue;
} else if (!this.endpoints.containsKey(from)
        && !this.unconnectedChannels.isEmpty()) {
                UdpChannel toChannel = this.unconnectedChannels.poll();
                toChannel.connect(from.getAddress(), from.getPort());
                this.endpoints.put(from, toChannel);
}
```

Bekommen wir nun ein Packet, holen wir uns den Absender herraus. Ist der Absender noch nicht in unsere Liste drin, schauen wir nach ob noch ein unverbundener Channel wartet. Haben wir so einen gefunden, dann verbinden wir diesen Channel und geben ihm danach die Nachricht weiter.

Ist der schon verbunden, werden wir ihm einfach die Nachricht weiterleiten

Wir haben für diesen Zweck eine neue Message Klasse erzeugt, damit wir die Bytearrays gesammelt weiter leiten können und nicht immer alle Bytes einzeln in die Messagequeue zu stecken.

Damit das Ganze funktioniert, müssen Senden und Empfangen auf dem Socket gleichzeitig funktionieren. Das wird uns von der API zugesichert und läuft in den Tests ganz gut.

Der Channel hält dann eine Blockingqueue, in die der Dispatcher die Messages einträgt. Der Channel wird dann beim receive auf die Queue Blocken.

Will der Channel etwas senden, verpackt er seine Message einfach in ein UDP Packet und sendet es über das Socket an seinen connecteten Endpunkt.

Probleme

Unsere erstes Problem ist zur Zeit, dass wir eine maximale Packetgröße von 512 Byte haben. Es gibt eine maximale Packetgröße in UPD, wir haben es jetzt erst einmal wahllos auf diesen Wert gesetzt. Wollten wir größere Packete schicken, müssten wir ein Protokoll entwickeln, dass Fragmentation implementiert (siehe TCP).

Das nächste Problem, dass sich wiederum über ein Protokoll lösen lassen würde, ist dass ein channelClose nicht an den Gesprächspartner übermittelt wird. Man könnte sich Beispielsweise eine bestimmte Bytefolge vorstellen, die nicht im payload vorkommen darf, die uns signalisiert, dass der Channel geschlossen wurde.