# Verteilte Systeme 02

Yves Mueller

## Überblick

Meine Lösung gliedert sich ein drei Komponeten. Neben den durch Interfaces gegeben Komponten (Channel und Channelfactory) gibt es noch den UdpDispather. In dieser Klasse sind die eigentlichen Funktionalitäten abgebildet. Die Factory besteht aus ausschließlich statischen Methoden, die es erlauben neue Channels zu erzeugen und in ein sehr einfaches threadsicheres Singelton-pattern implmentieren, um immer genau einen Dispatcher zu verwenden.

## Dispatcher

Wie schon gesagt ist der Dispatcher die vermutlich intteresanteste Klasse. Ihm zur Seite steht noch die UdpTable ein Wrapper für Maps mit Mengen als Value. Diese Table benutze ich um Status über ein- und ausgehende Verbindungen zu halten. Zu den wichtigsten Aufgaben des Dispatcher gehört die Verwaltung dieser Informationen und der Sockets. Dafür verfügt der Dispather über Methoden, die UdpChannels zur Registrierung ihrer Empfangswünsche nutzen:

```
public void register(UdpChannel chan) {

    if (!channels.containsKey(chan.getLocalAddress().getPort())) {
        try {
            addPortToSelector(chan.getLocalAddress());
        } catch (IOException e) {
            log("Unable to open ports on socket: " + chan.getLocalPort());
        e.printStackTrace();
        }
    }
    table.addListner(chan.getLocalAddress().getPort(),chan);
}
```

Die Methode überprüft zu erst, ob ein Socket auf dem entsprechnden localen Port schon geöffnet wurde. Falls nicht wir dies mittel einer anderen Methode erledigt. Anschließend trägt sich der Channel als Listner für diesen Port ein. Es springt ins Auge, das der Channel keine Aussagen über seinen anderen Endpunkt treffen muss. Der Statusinformationen über die Gegenstelle werden nur im UdpChannel selbst gespeichert. Dies ist möglich, da die Steuerungslogik zum senden in der Channelimplmentierung ist und nicht wie die zum Empfangen im Dispatcher.

Wie gefordert habe ich die Netzwerkschnittstelle mittels java.nio entwickelt, was sich auch als eine der Größten Bremesen herrausstellte (Diese war mein erster Kontak mit nio). Ich hatte schon länger auf der Agenda die asynchronen Netzwerkqaulitäten von Java zu begutachten, und so habe ich die Aufgabe mit einem Selector implmentiert (wobei mir auch keine bessere Methode, außer mit sehr vielen Threads einfällt).

Mittels select() wartet der Dispatcher die meißte Zeit auf Pakete. Erhält er ein Event, so geht er die Liste der Listner auf dem betroffenen Prot durch und sucht zu erst nach einem Channel mit der Richtigen Gegenstelle. Dabei vermerkt er sich alle Channels ohne Gegenstelle. Findet er kienen mit Gegenstelle so wählt einen ungebunden und übermittelt ihm das Paket.

```
for (UdpChannel chan : listners) {
    if (chan.isClosed) {
        listners.remove(chan);
        continue;
```

```
    }

    if (chan.getRemoteAddress() == null) {
        unbound.add(chan);
        continue;
    }

    if (remote.equals(chan.getRemoteAddress())) {
        chan.queue.add(m);
        return;
    }
}

log("There are " + unbound.size() + " unbound chans.");
if (unbound.size() > 0) {
    UdpChannel u = unbound.iterator().next();
    u.queue.add(m);
    u.remote = remote;
    return;
}
```

# Channel

Zum Empfangen von Paketen verfügt jeder Channel über eine Blockinqueue, in die der Dispatcher neue Pakete pusht. Mit entsprechenden Schnittstellen lassen sich dann sowohl blockierendes als auch nicht blockierendes Empfangen realsieren (ich hab aber wie gefordert nur die nicht blockierende Variante implmentiert). Das Senden ist einfacher realsiert. Da sich der Channel bei der Konfiguration oder dem ersten empfangene Paket seine Gegenstelle merkt, kann er die send Methode des Dispatcher nutzen, die im Prinzip nur ein Wrapper für das send des Sockets ist.