Verteilte Systeme: Übung 3

Max Wisniewski, Alexander Steen

May 23, 2011

Aufgabe 1: nrecv

Da wir die letzte Lösung mittels `BlockingQueue` implementiert haben, lässt sich das `nrecv` sehr einfach implementieren:

```
public Message nrecv(){
    return this.recieved.poll();
}
```

`BlockingQueue#poll` gibt nämlich null zurück, falls noch keine Nachricht vorhanden ist.

Aufgabe 2: Framework

Die zentralen Klassen, die es zu erklären gilt, sind in unserem Framework zum einen *CentralSequencer* und *CentralSyncProcess*. Diese beiden Klassen sind hinter jeweils einem Interface, bzw. einer abstrakten Klasse, versteckt. Diese sind alle aber wiederrum Package-Private, weil von außen niemand zu wissen hat, wie die Synchronisation genau abläuft.

Die Wahl der jeweiligen Synchronisation findet über die Klassen *Connection* und *ProcessConnect* statt. Diese beiden Klassen stellen einen Adapter dar, indem Verbindungen jeder Art aufgebaut werden können, in Channels verpackt und zuletzt an unsere beiden zentralen Klassne wieter geleitet werden können.

Die beiden Connectionklassen bedienen sich dazu einer Factory, die bisher nur die beiden zuerst genannten Klassen zur Verfügung stellt, aber innerhalb des Frameworks schnell zu anderen Lösungen erweiterbar ist.

(Denkbar sind viele zentrale Knoten um das ganze Ausfallsicher zu gestallten oder ein Tokenring, der nur über die Syncprocesses läuft). Betrachten wir nun einmal die zentralen Klassen.

Die Klasse *CentralSequencer* verwaltet intern 3 Listen von Channels.

```
private List<Channel> syncProcesses;
private List<Channel> newProcesses;
private List<Channel> delProcesses;
```

In syncProcesses, stehen alle Channel von Prozessen, die im Moment synchronisiert werden. Die Listen *newProcesses* und *delProcesses*, sind Zwischenspeicher, damit neue oder zu entfernende Channels nicht den Betrieb stören. (Da wärend einer ACK-Phase kein Channel wegfallen darf, da es in dieser Implementierung zu einem Deadlock führen würde.) Das hinzufügen, löschen, starten, stoppen das internen Threads ist nicht weiter spannend. Starten tut der Thread nur, wenn Channels zu verwalten sind. Beim löschen und hinzufügen, wie beschrieben, einfügen in die Extralisten, über die synchronisiert wird. Und stop setzt einfach eine Flag, damit nach der nächsten Runde der Prozess hält. Die Hauptschleife des Threads sieht nun folgender Maßen aus:

```
this.workAck();

while(running){
        //Normaler Ablauf eines Synchronen Algorithmuses
        this.allSend();  // Alle syncProcesses haben gesendet
        this.sendAck();  // Sende allen ein ACK für das Senden
        this.allWork();  // Alle Prozesse haben gearbeitet

        //Hinzufügen neuer Prozesse
        synchronized(newProcesses){
                syncProcesses.addAll(newProcesses);
                newProcesses.clear();
        }
        //Löschen nicht gebrauchte Prozesse
        synchronized(delProcesses){
                syncProcesses.removeAll(delProcesses);
                delProcesses.clear();
                if(syncProcesses.isEmpty()) running = false;
        }
        this.workAck();  //Sendet allen Prozessen ein ACK für das Arbeiten.
}
```

Bevor wir mit der Hauptschleife starten können, müssen wir allen Prozessen ein *WorkACK* schicken, was auf der Gegenseite als Startschuss aufgenommen wird.

Nun fangen wir als erstes an zu warten, dass alle Prozesse uns mitgeteilt haben, dass diese fertig sind. Sind alle Nachrichten eingetroffen, bestätigen wir dies allen. Genauso läuft auch work ab.

Da zwischen Recv und Work nichts mehr passieren kann, haben wir an dieser Stelle den Schritt zusammen gefasst, so dass aus *sendWork*, auch implizit ein receive Befehl vorraus geht. Dies setzt natürlich vorraus, dass der Gegenpart das Protokoll kennt. Am Ende jeder Runde, werden alle neuen Channels zu den synchronisierten hinzugefügt und alle fertigen gelöscht.

```
private void allWork(){
    boolean rightMessage;
    for (Channel process : syncProcesses) {
        do{
            byte[] tag = process.recv().getData();
            rightMessage = (tag.equals(SyncMessage.PROCESS_ENDED));
            if(rightMessage) remove(process);
            else rightMessage = (tag.equals(SyncMessage.PROCESS_WORKED));
        }while(!rightMessage);
    }
}
```

Dies ist einmal exemplarisch eine der Methoden aus der Hauptschleife.

Wir iterieren über jeden der synchronisierten Channel und holen uns das byte-Array aus der Nachricht (blockende Methode).

Im *WorkAck* testen wir auch noch, ob der Prozess uns ein Signal gesendet hat, dass er aufhören möchte. Ansonsten erwarten wir ein *PROCESS_WORKED*. Als kleine Optimierung könnte man hier noch einen Fehler zurück schicken, damit der Gegenpart weiß, dass die Nachricht falsch war.

Hat man entweder das *Ende* oder das *Worked* erhalten, kann man mit dem nächsten fortfahren. Die Nachrichten, auf die wir uns hier beziehen, sind static Variablen in einer Message Klasse. Diese haben wir angelegt, um die Synchronisationsnachrichtenn nicht jedes mal erneut schreiben zu müssen.

Der Gegenpart dazu ist *SyncProcess*. Die Hauptschleife dieses Threads, macht nichts, außer die Nachrichten vom Server zu bekommen, den angeforderten Befehl auszuführen und danach ein Ack zu senden.

```
do{}while(!syncChannel.recv().getData().equals(SyncMessage.WORK_ACK));

do{
        callSend();
        callRecv();
        callWork();
    }
}while(!work.isEnded());

work.sendPhase();
```

In der while - Schleife werden immer sende, empfangs und Arbeitsphase ausgeführt. Kommt man aus der Schleife herraus, senden wir noch ein letztes mal unsere Nachricht. Sonst kommt es leicht vor, dass der nächste Prozess in der Reihe verhungert.

```
public void callSend() {
    work.sendPhase();
    syncChannel.send(new SyncMessage(SyncMessage.PROCESS_SENDED));
    do{}while(!syncChannel.recv().getData().equals(SyncMessage.SEND_ACK));
}
```

Die Methode ruft nun den eigentilchen Sendebefehl aus, sendet die Bestätigung, wenn sie damit fertig ist und wartet zuletzt auf das ACK vom Server. Damit nun ein Algorithmus auf diesem Framework ausgeführt werden kann, muss nur noch die Klasse *Work* geschrieben werden.

```
public interface Work {

    public void sendPhase();
    public void recvPhase();
    public void workPhase();

    public boolean isEnded();
}
```

Hat man so eine Klasse, wird diese nur noch in ProcessConnect hineingeschoben und sobald das Netzwerk gestartet wurde, läuft der Algorithmus.

Aufgabe 3: Timeslice

Legt man das Framework zugrunde ist das Schreiben der *TimeSliceWork* wirklich einfach. Durch eine abstrakte Klasse, werden uns auch noch senden und empfangen abgenommen. WIr müssen also nur noch die Zustandsüberführungsfunktion *workPhase()* schreiben.

```
public void workPhase() {
    String m = new String(this.received.getData());
    if(this.leaderID != -1){
        this.electionComplete = true;
    }else if(m.equals("")){
        if(!this.electionComplete && this.id == this.phase){
            System.out.println("GEWONNEN: "+this.id);
            this.message = new SyncMessage((this.id+"").getBytes());
            this.leaderID = this.id;
        }else{
            if(++round == anzProcess){
                round = 0;
                ++this.phase;
            }
        }
    }else{
        this.leaderID = Integer.valueOf(m);
        this.message = new SyncMessage((leaderID+"").getBytes());
    }
}
```

Zu Beginn der Working Phase holen wir uns das byte Array aus der Message. Als erstes Überprüfen wir, ob wir eine leader ID gesetzt haben. In dem Fall können wir uns einfach beenden. Dieser Teil ist unter Umständen in der neusten Implementierung obsolet, aber er schaden zur Zeit noch nicht.

Im nächsten Teil, wenn wir keine Nachricht empfangen haben, schauen wir zuerst nach, ob wir die ID der aktuellen Runde haben. Haben wir sie, sind wir Leader und schicken diese Nachricht weiter, das heißt setzen unsere Sendenachricht darauf.

Haben wir die ID nicht, setzen wir die Runde um 1 Hoch und testen, ob wir in die nächste Phase eintreten.

Haben wir eine Nachricht bekommen, muss dies die ID des neuen Leaders sein. Wir parsen die ID, speichern sie und schicken sie an den nächsten. In unserem Testcase erstellen wir Threads, mit der Anzahl, die uns vom User gegeben wurde. (War die EIngabe keine Zahl, lassen wir die NumberFormatExcpetion einfach durch).

Die IDs bestimmen wir einfach über einen Randomgenerator zusammen mit einem Bitfeld, so dass wir keine ID 2mal benutzen.

Danach verbinden wir die erstellten Threads einfach über Bidirectionale Pipes aus dem *vsFramework*. DIe Connecionclasses haben wir für unseren localen Fall sehr einfach Implementiert.

Connections bekommt einfach Channels und reicht diese weiter. ProcessConnect bekommt diese Connectionklasse. Kommt nun eine neue Workklasse, wird einfach eine Pipe erstellt und in den Prozess und den Sequencer gegeben. Das ist zu kaum etwas zu gebrauchen, aber für unseren Test völlig ausreichend.

```
----- Runde : 99 ------
--- Praticipants: 3
-- send round
.... received all ACKs
-- receive & statechange round
.... received all ACKs
```

Die Statusnachrichten sind alle nach diesem Shema aufgebaut. Die Tests sind folgender Maßen zu Bedienen:

Der erste Parameter gibt die Anzahl der Threads an. Ist keiner Angageben werden 5 genommen.

Der zweie Parameter fügt einen Versatz in den IDs der Prozesse hinzu. Bei dem Aufruf *TimeSlice* n k bekommen wir IDs aus dem Bereich

$k \leq id \leq 2n + k$.

Dies war nötig, da mit einer sehr hohen Wahrscheinlichkeit jedesmal die 0 unter den IDs war.

Alle unsere Test liefen bisher gut und das Programm terminiert auch.