

## Übungsblatt 7

**Ausgabe:** 6.12.2011

**Abgabe:** 16.12.2011

---

**Aufgabe 1** (Master-Worker-Pattern mit RMI):

(20 Punkte)

Ziel dieser Aufgabe ist es, ein Framework zu schreiben, das ein Problem in Teilprobleme zerlegt und diese dann verteilt berechnen lässt.

Es soll wie folgt funktionieren:

Der **Master** wird auf einem bekannten Rechner gestartet. Er hat 2 Schnittstellen. Die eine interagiert mit den Workern (Master-Interface), die andere mit einem Client (Server-Interface).

Die **Worker** registrieren sich bei dem Master. Nun laden diese nach Aufforderung durch den Master Code (Task-Interface) (und Parameter dafür) herunter und führen diesen lokal aus. Die Ergebnisse liefern sie dem Master zurück. Dies geschieht so lange, bis es keine Teilaufgaben mehr bei dem Server gibt.

Der **Client** kann dem Master einen Auftrag (Job) geben (Code und eine Funktion, die anhand der Anzahl der vorhandenen Worker Teilaufgaben erstellt).

- a) (4 Punkte) Laden Sie sich das Framework für die Aufgabe von der Veranstaltungsseite herunter. Benutzen Sie zum Lösen der Aufgaben die vorgegebenen Interfaces. Das Manipulieren bestehender Interfaces ist nicht erlaubt.

Implementieren Sie das *Worker*-Interface und schreiben Sie den gesamten Code für einen Worker. Dabei ist zu beachten, dass die **Anzahl** der gleichzeitig arbeitenden Threads einstellbar sein sollte, genau wie die Adresse (**IP und Port**) des Masters.

Die *start()*-Methode darf nicht blockieren und soll nicht erst bei Beendigung des Rechnens, sondern schnellstmöglich zum Server zurückkehren, damit dieser einfach in einer Iteration über eine Liste der verbundenen Worker Befehle erteilen kann.

- b) (3 Punkte) Implementieren Sie das Pool-Interface. Achten Sie dabei auf geeignete Sperr-Synchronisierung.
- c) (4 Punkte) Implementieren Sie den Master-Teil des Masters. Dieser muss speichern, welche Worker vorhanden sind und pro Auftrag einen Parameter- und Ergebnis-Pool bereitstellen.
- Vergessen Sie die RMI-Registry nicht. Diese sollte in der *main()*-Methode des Masters erstellt und benutzt werden.

- d) (6 Punkte) Implementieren Sie den Server-Teil des Masters. Mit *doJob()* kann ein Client einen Job starten. Dieser Aufruf soll schnellstmöglich zurückkehren und ein *RemoteFuture*-Objekt, das von dem Job-Objekt erzeugt wird, zurückliefern über welches der Client das Gesamtergebnis erhalten kann.

Der Master soll so vorgehen:

- zwei Pools erstellen
- mit *job.split()* geeignete Teilprobleme berechnen
- (jetzt das Future-Objekt zurückliefern)
- die Teilprobleme in den Argument-Pool stecken
- die *job.merge()*-Methode aufrufen
- Worker starten

So hat der Code vom Client die Freiheit, zu warten, bis alle Teilergebnisse fertig sind und dann ein Gesamt-Ergebnis (auf dem Master) zu berechnen, oder die Teilergebnisse über einen Callback ("push" - siehe Übung 6) oder das *RemoteFuture*-Objekt (dies würde Pulling entsprechen) sofort zum Client zurück zu leiten.

Hinweise:

- Um den Master einfach zu halten, gehen wir davon aus, dass Worker es immer schaffen, die Berechnung durchzuführen und niemals abstürzen.
- Es soll möglich sein, Jobs an den Master zu schicken, während bereits ein Job bearbeitet wird. Diese sollen nach den Vorberechnungen aber zwischengespeichert und in der Reihenfolge bearbeitet werden, wie sie eingetroffen sind.
- Wenn sich Worker beim Master anmelden, während ein Job läuft, sollen diese ebenfalls hinzugefügt werden.

- e) (4 Punkte) Schreiben Sie den Client, der einen simplen Job (z.B. sortieren einer sehr langen Liste von Zahlen, Häufigkeit von Wörtern in Texten, Patternmatching in Texten o.ä.) an Ihren Master schickt.

Testen Sie damit, ob ihr Framework korrekt funktioniert.

- optional: f) (2 Punkte) Schreiben Sie einen Job, der das gesamte System, also Worker, Master und Client in dieser Reihenfolge "gracefully" - also ohne abstürzen - beendet.

Dazu müssen u.a. die Worker abgemeldet werden und kurz bevor der Master herunterfährt soll eine Erfolgsmeldung an den Client geschickt werden.

optional: g) (4 Punkte) Schreiben Sie einen Job, der etwas (sinnvolles) tut - `sleep(1000)` o.ä. ist allerdings ausreichend. Er soll viele Teilprobleme erstellen und den Client bei jedem berechneten Teilproblem über Fortschritt informieren. (sichtbar durch eine Prozentanzeige oder eine Art (textuellen) Fortschrittbalken in dem Client)