Tentamen
Nätverksprogrammering
Del 2

2017–06–02, 8.00–13.00

Tillåtna hjälpmedel för denna del av tentamen:

- Java snabbreferens.
- Kursboken: Java Network Programming av Eliotte Rusty Harold.
- Valfri lärobok i Java.
- Utskrift av OH-bilder från föreläsningarna.

Denna tentamen i kursen Nätverksprogrammering består av två delar – en del som innehåller frågor av teoretisk/principiell/utredande karaktär och en del som innehåller praktiska programmeringsuppgifter. Detta är del 2. Den ska du ha erhållit tillsammans med ett färgat tentamensomslag när du lämnade in din lösning på del 1 av tentamen.

För godkänt betyg på tentamen krävs sammanlagt minst 20 poäng på tentamen, varav minst 8 poäng på vardera deltentamen.

Massiva parallella beräkningar

Uppgiften handlar om att implementera ett ramverk för massiva parallella beräkningar. Ramverket förutsätter att beräkningarna betår av många oberoende funktioner, d.v.s. beräkningar utan tillstånd och utan sidoeffekter. Ramverket ska distribuera beräkningarna över att stort antal datorer. För att utnyttja alla processorkärnor på varje dator ska beräkningarna göras i flera trådar på varje dator. Systemet betår av en server som har en lista med alla beräkningar som ska utföras. Klienterna, som körs på datorerna som utför beräkningarna, hämtar arbeten från servern, utför beräkningarna och skickar sedan tillbaka resultaten till servern. Din uppgift är att implementera klient-delen av ramverket.
Klienten består av tre delar:

1. Kommunikation med servern.
2. En monitor, `JobCache`, för kommunikation mellan trådarna i klienten. Monitorn har en cache med beräkningar som ska utföras och lagrar även resultat innan de skickas till servern.
3. Ett antal trådar som utför beräkningarna. Dessa kallas för arbetstrådar.

Representation av beräkningar

Beräkningar kapslas in i subklasser till `Job`:

```
abstract class Job {
   /**
    * Creates a Job object from the data sent by the server.
    * @param data A byte array containing the parameters of the job.
    */
   static Job createJob(byte[] data) { ... }

   /**
    * Preform the computation.
    */
    abstract void compute();

   /**
    * Returns the result as an array of bytes to be sent to the server.
    * compute() must have been called before calling getResult().
    * @return The result of this job, an byte array to be sent to the server.
    */
    abstract byte[] getResult();
}
```

Monitorn

Klienten består av flera trådar och de kommunicerar via en monitor, `JobCache`. Arbetstrådarna hämtar nya jobb från monitorn och registrerar resultaten när de är klara med beräkningen. Monitorns uppgift är även att bestämma när nya arbeten ska hämtas från servern. När färre än 10 arbeten finns i cachen ska fler arbeten hämtas från servern för att säkerställa att de finns tillgängliga när arbetstrådarna blir klara med sina nuvarande uppgifter. Detta åstadkoms genom att `waitUntilRefillIsNeeded()` blockerar tills det är dags att hämta nya beräkningar från servern. Av prestandaskäl vill man inte skicka enstaka resultat, utan monitorn slå ihop minst 10 resultat så de kan skickas samtidigt under en TCP-uppkoppling.

```
interface JobCache {
   /**
    * Adds new jobs to the cache.
    * @param jobList contains a list of new jobs.
    */
   synchronized void addJobs(Job[] jobList);

   /**
    * Blocks until JobCache contains less than 10 jobs
    * that has not been started.
    */
   synchronized void waitUntilRefillIsNeeded();

   /**
    * Adds the result of one job.
    * @param result, the result of one job.
    */
   synchronized void addResult(Job result);

   /**
    *  Returns one job from the job list. The job is removed from the list.
    *  After the job is done, the thread is responsible for returning the result
    *  using the addResult() method. Blocks if no jobs are available.
    * @return a job to be performed
    */
```

```
    synchronized Job getJob();

    /**
     * Retrieves a list of all finished jobs.
     * The list contains at least 10 results.
     * Block if less than 10 results are available.
     * @return A list of all finished jobs.
     */
    synchronized Job[] getResults();
}
```

Server-Klient kommunikation

Kommunikationen med servern sker med TCP. Då systemet består av många klienter kan inte alla hålla en uppkoppling öppen hela tiden. En ny uppkoppling görs därför varje gång som nya arbeten hämtas eller resultat lämnas. Protokollet mellan klienten och servern beskrivs nedan.

hämta arbete

1. Klienten gör en TCP-uppkoppling mot `workserver.info`, port `9090`.
2. Klienten skickar strängen ‘‘`get more work`’’, utan citattecken.
3. Servern svarar då med en sekvens av bytes. Den första byten är antalet arbeten som skickas.
4. Sedan följer de enskilda arbetena. Varje arbete inleds med en byte som anger längden på parametrarna (antalet bytes), som sedan följer. Det är denna sekvens av bytes som ska användas som parameter till `Job.createJob()`.
5. Servern stänger TCP-uppkopplingen.

skicka resultat

1. Klienten gör en TCP-uppkoppling mot `workserver.info`, port `9091`.
2. Klienten skickar strängen ‘‘`put`’’, utan citattecken.
3. Klienten skickar sedan en sekvens av bytes. Första byten som skickas är antalet resultat som följer.
4. Sedan följer de enskilda resultaten. Ett resultat inleds med en byte som anger längden på resultatet (antalet bytes), som sedan följer. Sekvensen av bytes är det som `Job.getResult()` returnerar.
5. Klienten stänger TCP-uppkopplingen.

uppgiften

a) Skapa en klass som implementerar interfacet `JobCache`.

(6p)

b) Implementera kommunikationen med servern. Nya arbeten ska läggas in i monitorn du skapade i a). Resultaten som skickas till servern hämtas även från samma monitor.

(6p)

c) Implementera en tråd, `Worker`, som utför beräkningar. Arbeten hämtas från monitorn du skapade i uppgift a). I samma monitor lämnas även resultaten.

(3p)

d) Implementera en klass, `WorkingClient`, som startar upp klienten, d.v.s. klasserna i uppgifterna a)-c). 4 arbetstrådar ska skapas.

(3p)

Slut – lycka till!