LUNDS TEKNISKA HÖGSKOLA
Datavetenskap

EDA095

Tentamensskrivning — Nätverksprogrammering 2006-05-27, kl 8-13

DEL 2 - Praktiska programmeringsuppgifter

Anvisningar

Tillåtna hjälpmedel för denna del av tentamen:

- Java snabbreferens.
- Kurslitteraturen inklusive boken Java Network Programming av Elliotte Rusty Harold.
- Valfri lärobok i Java.
- Utskrift av OH-bilder från föreläsningarna - ej separata exempelprogram.

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. För högre betyg krävs naturligtvis mer, så gör så många uppgifter du kan.

(vänd)

1. Synkronisering av klockor

I vissa tillämpningar kan det vara viktigt att veta när saker och ting hände i ett system med ganska hög precision. I Java finns det i klassen `System` möjlighet att få veta vad tiden är genom att anropa den statiska metoden `currentTimeMillis()`:

```
public static long currentTimeMillis();
```

Denna metod returnerar nuvarande tidpunkt mätt i millisekunder sedan midnatt 1/1 1970.

I ett distribuerat system blir problemet värre i och med att alla datorerna i nätverket har en egen klocka som inte nödvändigtvis är synkroniserad med klockorna i de andra datorerna. Det finns därför ett behov av att kontinuerligt synkronisera klockorna på de olika datorerna i nätverket. Vi antar härvid att det finns en metod `setCurrentTimeMillis()` i klassen `System` med vars hjälp vi kan ställa om den lokala klockan[1]:

```
public static void setCurrentTimeMillis(long newTime);
```

Din uppgift blir att skriva två klasser som kan användas för att automatiskt hålla klockorna synkroniserade i ett nätverk av datorer med dålig precision i sina klockkretsar enligt nedanstående klasskelett:

```
public class ClockSyncClient {
    /** Konstruktor. serverAddr och portNo anger dator respektive portnummer
        för den klocksynkroniseringsserver mot vilken alla datorers klockor
        ska synkroniseras. Parametern period anger hur ofta (i minuter) som
        den lokala klockan ska försökas synkroniseras med den centrala klockan.*/
    public ClockSyncClient(String serverAddr,int portNo,int period) {
        ...
    }

    ...
}

public class ClockSyncServer {
    /** Konstruktor. portNo anger portnumret som denna klocksynkroniseringsserver
        skall lyssna på. */
    public ClockSyncServer(int portNo) {
        ...
    }
}
```

En instans av klassen `ClockSyncClient` ska göra ett försök att synkronisera sin klocka med servern som implementeras av klassen `ClockSyncServer` med det tidsintervall som anges i parametern `period` i konstruktorn.

Ett synkroniseringsförsök går till på ungefär följande sätt:

1. Klienten skickar en begäran om tidsuppgift till servern. Samtidigt kommer den ihåg vad dess egen klocka visar just då:
   "`long time1 = System.currentTimeMillis();`"
2. Servern tar emot begäran och svarar med vad klockan är på serverns dator vid tillfället. Vi kallar denna tid för `serverTime`.
3. Klienten tar emot svaret och noterar tiden för mottagandet:
   "`long time2 = System.currentTimeMillis();`"
4. Klienten ställer om sin egen klocka baserat på tiden som servern sände och hur lång tid det tog att få ett svar från servern:
   "`System.setCurrentTimeMillis(serverTime+(time2-time1)/2);`"

1. I verkligheten existerar det ingen sådan metod, men för uppgiftens skull antar vi ändå detta. I alla operativsystem finns det något sätt att ställa klockan även om det inte kan göras direkt från ett Java-program eller av vilket applikationsprogram som helst.

Implementera klasserna `ClockSyncClient` respektive `ClockSyncServer` fullständigt enligt nedanstående anvisningar:

- Det är självklart tillåtet att lägga till egna privata attribut och metoder i klasserna. Vidare går det bra att komplettera med egna klasser.
- När man skapat en instans av någon av klasserna `ClockSyncClient` respektive `ClockSyncServer` sköts synkroniseringen automatiskt i bakgrunden oberoende av vad applikationen gör i övrigt.
- Den/de tråd(ar) som hanterar synkroniseringen bör köras med högsta prioritet eftersom tidmätningen annars kan störas. Detta gäller både klient och server.
- Förbindelsen till servern kan vara tillfälligt bruten. Om klienten inte mottagit ett svar från servern inom 200 millisekunder på en begäran om aktuell tid ska därför synkroniseringsförsöket avbrytas och nytt försök ska inte göras förrän efter den period som angavs när klienttråden startades.
  LEDNING 1: Med hjälp av metoden "`void setSoTimeout(int timeout);`" i socket-klasserna (`Socket/DatagramSocket`) kan man ange att ett `SocketTimedoutException` ska genereras angivet antal millisekunder efter anrop av read()/`receive()` om inget svar erhålls. Metoden `setSoTimeout()` kan generera ett `SocketException`. Se även kursboken.
- Om inget svar erhålls enligt föregående punkt och vi avbryter synkroniseringsförsöket kan det, beroende på ert designval, ändå vara så att ett svar bara var ovanligt mycket försenat och anländer strax efter att vi gav upp försöket. För att inte detta ska störa synkroniseringen behöver vi kunna sortera bort dessa meddelanden.
- **LEDNING 2:** För att vänta tills nästa gång det är dags att synkronisera klockorna kan du använda metoden `sleep(long milliseconds)` som finns i klassen `Thread`. Metoden kan generera ett `InterruptedException` som måste fångas. Det angivna tidsintervallet för synkroniseringsförsök är att se som ungefärligt.

(10p)

2. Distribuerad high-score-lista via RMI

I ett datorspel skrivet i Java vill man ha en gemensam high-score-lista för alla som spelar spelet på olika platser i världen. Tillverkaren av spelet har därför satt upp en server som håller reda på de tio bästa poängen och namnen på dem som åstadkommit dem. Spelet kopplar upp sig mot servern och hämtar ner high-score-listan och om spelaren skulle lyckas få en poäng som är högre än några av poängen på high-score-listan så ansluter spelet återigen sig till servern och skickar över information om det nya resultatet. Den nya poängen sorteras därvid in på rätt plats i high-score-listan (om den ska vara med vill säga; nya poäng kan ju ha kommit in från annat håll under tiden).

Javakoden för spelet innehåller bland annat följande kodsnuttar som har hand om kommunikationen med servern:

```
public class HiScores implements Serializable{
    public String[] player;
    public int[] score;

    public HiScores() {
        player = new String[10];
        score = new int[10];
    }
}

public class HiScoreCommunication {
    ...

    public HiScoreCommunication() {
        // Establishes a connection to the game server
        ...
    }

    public HiScores fetchHiScores() {
        // Gets the high-score list from the server
        ...
    }

    public void sendNewScore(String name,int score) {
        // Sends a new high-score to the server
        ...
    }
}
```

Spelet använder sig av klassen `HiScoreCommunication` för att kommunicera med tillverkarens server. För att erhålla en lista över de tio bästa resultaten från servern i form av en vektor sorterade efter fallande poängtal (score) anropar spelprogrammet metoden `fetchHiScores`. När spelprogrammet tror att ett nytt bästaresultat har uppnåtts skickar den detta resultat tillsammans med spelarens namn till servern genom att anropa metoden `sendNewScore`.

Använd kommunikationstekniken RMI för att implementera klassen `HiScoreCommunication` ovan tillsammans med ett serverprogram, inklusive main-operation, i Java som fungerar ihop med spelprogrammet beskrivet ovan och eventuella andra klasser som kan tänkas behövas på klienten och servern.

(10p)
