Tentamen
Nätverksprogrammering
Del 2

2015–06–04, 14.00–19.00

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.

Denna del består av två deluppgifter om 12 respektive 8 poäng. 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.

OBSERVERA! De två deluppgifterna hänger ihop och den andra deluppgiften bygger i viss mån på din lösning av den första. Därför rekommenderar vi att du läser igenom *båda* uppgifterna *innan* du börjar lösa uppgift 1. Båda deluppgifterna är i sin tur uppdelade i ett antal mindre steg. De är utformade så att även om du skulle köra fast på ett enskilt stg bör det inte hindra dig från att gå vidare till och lösa nästa. Så ge inte upp om de fastnar på ett enskilt steg.

---

1. MonoScan

Som nyanställd på marknadsundersökningsföretaget SpyBest har du fått i uppdrag att undersöka marknadsandelen för olika typer av webbservrar i utvalda delar av världen. Din lösningsidé är enkel:

1) Generera alla IP-adresser inom ett visst intervall. För att förenkla uppgiften håller vi oss till IPv4-adresser vilka består av fyra oktetter (åttabitarstal) vardera. Vi begränsar vidare intervallet till att bestå av alla IP-nummer där de tre första oktetterna är lika och den sista oktetten varierar inom intervallet [0, 255]. Exempelvis kan ett sådant intervall bestå av alla IP-nummer från 130.235.16.0 till 130.235.16.255.
2) Anslut till alla adresser i intervallet via deras HTTP-port (80) och (om svar erhålls):
3) Läs in informationen om den aktuella webbserverns typ och version.
4) Skriv slutligen ut statistik som visar vilka webbservertyper/versioner som hittats. Skriv för varje servertyp/version ut antalet servrar som hittats av denna typ.

Efter att din chef godkänt din plan skriver du ett program, *MonoScan*, som löser uppgiften. Detta gör du genom att gå igenom ett antal delmoment:

a) Om man ansluter till en given IPv4-adress, t.ex. 130.235.209.220 (webbservern för Institutionen för datavetenskap), på dess HTTP-port (normalt 80) och skickar ett GET-kommando svarar servern bland annat med en header innehållande följande rader där olika parametrars värde redovisas):

```
HTTP/1.0 200 OK
Accept-Ranges: bytes
Age: 0
Connection: keep-alive
Content-Length: 48405
Content-Type: text/html; charset=utf-8
Date: Mon, 01 Jun 2015 08:19:09 GMT
Server: Apache/2.2.15 (Red Hat)
Via: 1.1 varnish
X-Cacheable: NO
X-Powered-By: PHP/5.3.3
X-T3Cache: 1
X-T3CacheInfo: cacheContentFlag,loginAllowedInBranch,staticCacheable,
ClientCache,not_loggedin
X-Varnish: 121730987
X-Varnish-Cache: MISS
```

Vad är namnet på den parameter som anger servertyp/serverversion? Vilket värde har denna parameter i exemplet ovan?

(1p)

b) Använd klasserna `URL` och `URLConnection` för att skriva en metod `dotAddress2Server` med följande signatur:

```
String dotAddress2Server(String address)
```

Lägg metoden i en klass `MonoScan`. Metoden ska ansluta till den givna adressen, som t.ex kan vara ”130.235.209.220”, och returnera namnet/versionen för webbservern på den angivna adressen. Om ingen server svarar på den aktuella adressen så ska `null` returneras. Eftersom servrar ibland är långsamma på att svara och ibland inte svarar alls behövs en timeout efter en viss tid. Vi väljer att detta ska ske efter 200 ms. Detta kan du åstadkomma genom att göra metodanropen `setConnectTime(200)` och `setReadTime(200)` på din `URLConnection`. Om timeout sker kommer då ett `SocketTimeoutException` att kastas. Returnera `null` även i timeoutfallet.

Tips: Värdet för enskilda header-fält kan hämtas med metoden `getHeaderField` i klassen `URLConnection`.

(4p)

c) Skriv en separat klass `ServerStats` som ska användas för att lagra vilka servrar som använder de olika webbserverversionerna. Använd en map (lexikon) för att lagra informationen enligt:

```
Map<String, List<String>> serverCnt;
```

Låt nyckeln vara strängen som anger servertyp/version och värdet en lista över vilka servrar som använder denna typ av server. Exempel:

Apache/2.2.15 (Red Hat)=[130.235.209.216, 130.235.209.217, 130.235.209.218, 130.235.209.219, 130.235.209.220, 130.235.209.221, 130.235.209.240]

Här är nyckeln ”Apache/2.2.15 (Red Hat)” och värdena i listan är ”130.235.209.216”, ”130.235.209.217”, ”130.235.209.218”, ”130.235.209.219”, ”130.235.209.220”, ”130.235.209.221”, ”130.235.209.240”.

Låt klassen ha en metod för att uppdatera din map och en metod för att skriva ut innehållet i densamma på stdout:

```
void addServer(String key, String value);

void print();
```

Låt `print` skriva ut en rad för var typ av server/version där dess beteckning anges följt av antalet servrar av denna typ som hittades. Utskriftsordningen är godtycklig. Exempel:

```
Apache/2.2.15 (Red Hat): 7
Apache/2.2.22 (Debian): 1
Boa/0.94.14rc21: 2
Microsoft-IIS/4.0: 1
Microsoft-IIS/8.0: 1
Virata-EmWeb/R6_0_1: 1
Virata-EmWeb/R6_2_1: 2
Web-Server/3.0: 3
```

För att förbereda för framtida utökningar vill vi att alla servrars adresser lagras även om vi för tillfället bara skriver ut antalet funna servrar. Skriv klassen så att den även fungerar i en multitrådad miljö. Detta behövs senare i uppgift 2.

(2p)

d) Komplettera din klass `MonoScan` med en metod `scan` enligt:

```
void scan(String subnetAddress);
```

Låt argumentet `subnetAddress` vara en ofullständig IPv4-adress bestående av de tre första oktetterna i en adress, t.ex. ”130.235.209”. Generera kompletta adresser genom att komplettera adressen med de 256 möjliga värdena på den sista oktetten och extrahera serverinformationen för var och en av adresserna m.h.a metoden `dotAddress2Server` du skrev tidigare. Lagra resultaten i ett attribut i klassen MonoScan enligt:

```
public class MonoScan {
   private ServerStats servers;

   ...
}
```

(3p)

e) Skriv ett huvudprogram och en konstruktor (om nödvändigt) till klassen `MonoScan`. Låt huvudprogrammet ta en ofullständig IPv4-adress av typen ”130.235.209” som argument på kommandoraden, skapa ett objekt av klassen `MonoScan`, sätta igång avsökningen samt slutligen skriva ut resultatet av densamma.

(2p)

2. MultiScan

I jakt på större utmaningar söker du dig vidare till det mera välmående företaget MasterSpy, en konkurrent till SpyBest. För att göra skäl för din höjda lön bestämmer du dig för att utöka sökningen som ditt program gör från att bara variera den sista oktetten i adressen till att variera de *två* sista oktetterna. Det betyder att om man anger en adress ”130.235” när man startar programmet så kommer den att genomsöka alla adresser från 130.235.0.0 till 130.235.255.255.

a) Hur många adresser kommer programmet att undersöka med denna modifikation?

(1p)

b) Givet timeout-tiderna i uppgift 1, *ungefär* hur lång tid skulle det i ett värstafall kunna ta att skanna igenom hela serien av IP-nummer efter servrar med hjälp av en enkeltrådat program (som i uppgift 1)? Motivera ditt svar.

(1p)

c) För att snabba upp avsökningen vill vi parallellisera den så att vi kan ha upp till 32 samtidiga avsökningar igång samtidigt (för att inte överbelasta nätverket vill vi inte ha fler igång samtidigt). Du ska därför skriva ett nytt program, `MultiScan`, som använder sig av en trådpool implementerad med hjälp av klasserna `Executor` och `Future`. Varje jobb (task) ska bestå av en instans (av en något uppdaterad version) av klassen `MonoScan` från uppgift 1. Varje enskilt jobb (task) kommer alltså att ansvara för att skanna 256 enskilda IP-adresser (adresser där de tre första oktetterna i adressen är lika). De enskilda `MonoScan`-objekten ska nu dela på ett gemensamt `ServerStats`-objekt där all information om de funna servrarna lagras.

Modifiera din klass `MonoScan` så att den implementerar interfacet `Callable` och kan fungera som ett jobb i trådpoolen. Du kan eventuellt även behöva modifiera konstruktorn i klassen så att alla `MonoScan`-objekt delar på samma `ServerStats`-objekt[1]. Du behöver bara redovisa de förändringar du behöver göra av klassen `MonoScan`.

(3p)

d) Skriv en ny klass `MultiScan` innehållande ett huvudprogram som:

1) Läser in en ofullständig IPv4-adress bestående av två oktetter från kommandoraden (t.ex. ”130.235”).
2) Genererar 256 parallella jobb representerade av `MonoScan`-objekt – ett för varje tre-oktett-variant av den givna adressen (i vårt exempel ”130.235.0” till ”130.235.255”) och lägger in dessa som jobb i trådpoolen.
3) Väntar, genom att använda sig av en `Future` för varje jobb, på att samtliga jobb ska ha avslutats.
4) Skriver ut resultatet av avsökningen på skärmen.

(3p)

Slut – Glad Sommar!

[1] Har du inte redan sett till att `ServerStats` fungerar i en multitrådad miljö så är det dags att göra det nu.