

TDTS04:

Distansvektorrouting i java

Juha Takkinen, Ph.D.

IDA, Institutionen för datavetenskap

1.0 Översikt

Labb 4 består av att utforma, implementera, testa och demonstrera ett program i java som implementerar den distribuerade och asynkrona distansvektorroutningsalgoritmen baserad på Bellman-Ford-ekvationen. Lösningen ska även inkludera ”giftig retur”-algoritmen (poison reverse) som delvis löser problemet med loopar i routingstopologin. Du får som också en följdfråga att besvara om labben.

2.0 Vad du ska implementera

Huvudutmaningen i den här labbuppgiften är att din kod måste vara asynkron. Du får i labben tillgång till en händelsedrivna simulator som hanterar kommunikationen mellan routrar. Du ska implementera den del av algoritmen som exekveras i varje router. Även om routrarna är aktiva i ett och samma program (simulatorn) så kan de endast kommunicera med varandra genom att skicka meddelanden via simulatoren.

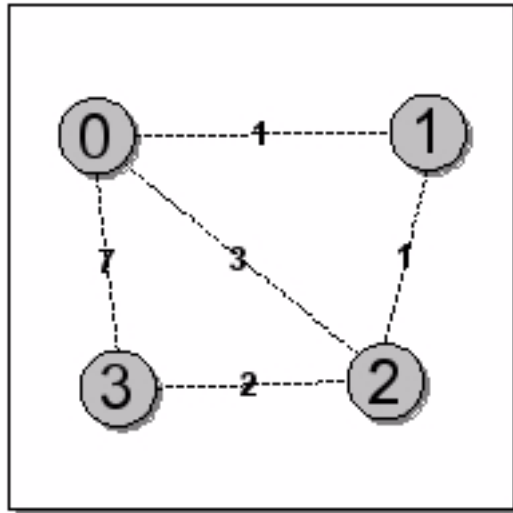
Du får följande filer i labben:

- RouterNode.java
- RouterSimulator.java
- RouterPacket.java
- GuiTextArea.java
- F.java

- Makefile

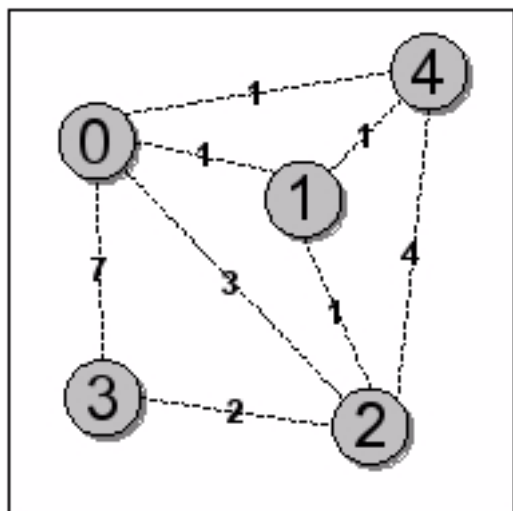
Det medföljer även en katalog kallad **test** där två versioner av simulatören finns tillgängliga för att testa din kodning.

RouterSimulator4.java å ena sidan är samma som **RouterSimulator.java** och hanterar ett nätverk med fyra noder enligt figur 1 nedan. **RouterSimulator5.java** å andra sidan simulerar ett nätverk med fem noder enligt figur 2.



FIGUR 1.

Nätverket med fyra noder i filen RouterSimulator4.java.



FIGUR 2.

Nätverket med fem noder i filen RouterSimulator5.java.

Den enkla klassen **F** i filen **F.java** kan du använda för att utföra enkla textformaterade in- och utmatningar.

All kod som du skapar ska du lägga i filen **RouterNode.java**. Du ska alltså endast ändra denna fil för att lösa labbuppgiften. Däremot kan du få behöva modifiera **RouterSimulator.java** och **RouterPacket.java** när du testar din kod, men vid demonstration av den slutliga lösningen ska all ny kod enbart finnas i **RouterNode.java**.

I filen **RouterNode.java** ska du lägga till kod för konstruktorer och metoder i klasserna som finns där. Du får endast kommunicera med andra routrar via metoden **sendUpdate()**. Du får inte lägga till några **static** medlemmar i en klass eller på annat sätt komma runt kravet att all information måste gå via **sendUpdate()**. Övriga metoder i filen är:

- **recvUpdate()**, som anropas av simulatoren när en nod får en uppdatering från en av sina grannar.
- **updateLinkCost()** exekveras när en kostnad på en länk som noden finns på förändras.
- **printDistanceTable()** används för avlusning och testning av koden och även för demonstration av lösningen. Denna metod ska skriva ut distansvektortabellen (routingstabellen) i ett format som du (och labbassistenten) kan läsa och förstå (OBS! Viktigt!). När du sedan demonstrerar din kod för labbassistenten så ska du kunna förklara exakt vad dina tabeller innehåller; listningen av tabellerna ska helst vara självförklarande.

Koden som du skapar ska dessutom implementera ”giftig retur”-algoritmen (poison reverse).

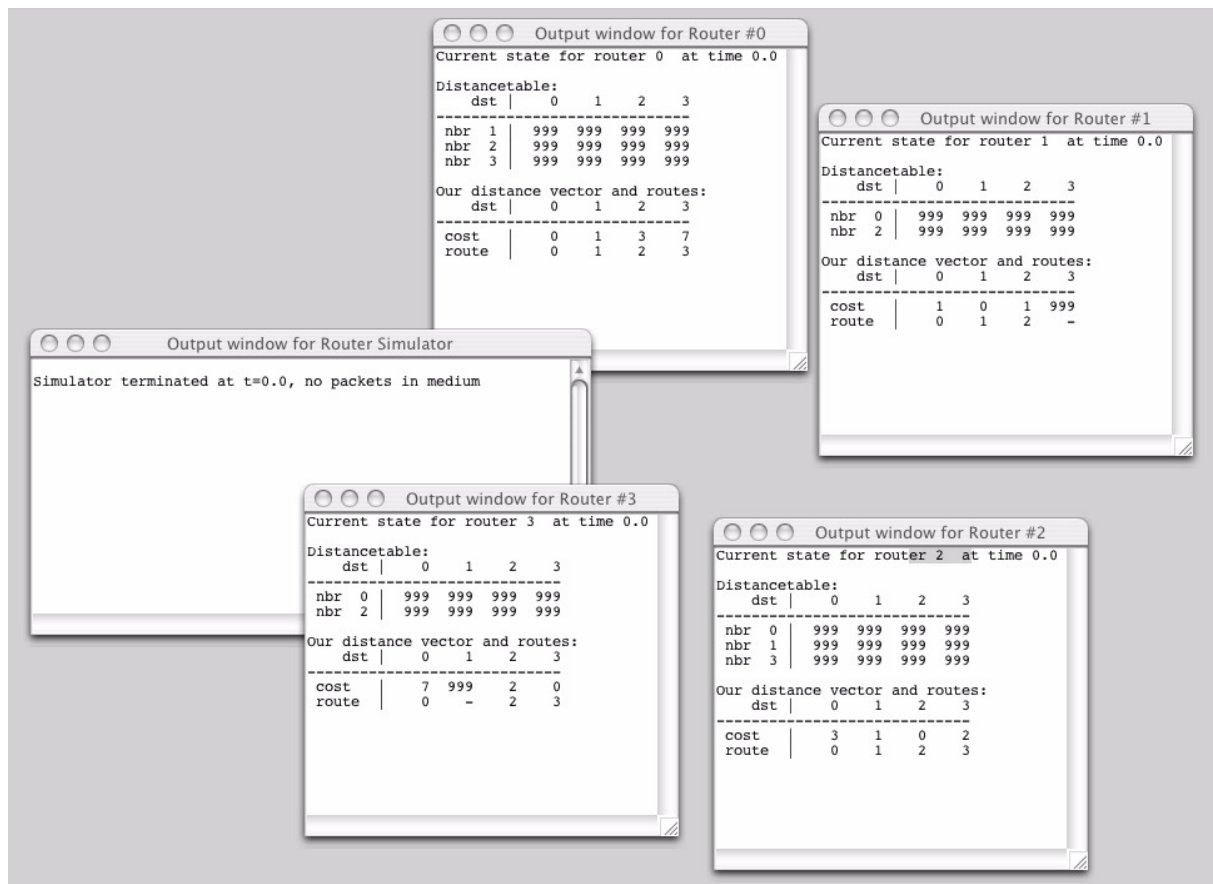
3.0 Exekvering av simulatoren

Du startar simulatoren genom att skriva

```
% java -DTrace=3 RouterSimulator
```

Simulatoren börjar med att initialisera varje router. Ett antal fönster öppnas på skärmen, en för simulatorns utdata och en för varje router-nod som initialiseras.¹ Se figur 3 på nästa sida. Titeln på fönstret beskriver vad som finns i det.

1. Det grafiska gränssnittet finns implementerat i filen **GuiTextArea.java**.



FIGUR 3.

Innehållet på skärmen när simulatören startas.

Från detta startläge ska din kod alltså se till att uppdateringar sker i nätverket, d.v.s routrarna i de olika fönstren. Simulatören kommer att exekvera tills routingstabellerna har konvergerat, d.v.s när det inte längre finns några meddelanden eller händelser kvar i systemet.

I konstruktören för simulatören kan man via kommandoraden sätta spårningsnivån (trace level) med flaggan **-DTrace=3** som du gjorde när du startade simulatören ovan. Ett spårningsvärde på 1 eller 2 gör att simulatören kommer att skriva ut detaljer om vad som händer inuti simulatören, t.ex. vad som sker med paket och timerfunktioner. Värde 0 stänger av spårningen helt. Ett värde över 2 gör att simulatören kommer att visa all möjlig information, som kan vara bra för avlusning av din kod.¹

1. I verklighetens nätverk finns inte en sådan här fin spårningsfunktion att tillgå.

Du kan även förändra startvärdet pseudoslumptalen i simulatoren, t.ex. till värdet 123, med flaggan **-DSeed=123**:

```
% java -DSeed=123 RouterSimulator
```

Ett annat startvärde på pseudoslumptalen ger en annan händelsekedja för meddelandena i ditt nätverk.

4.0 Övrigt

Du behöver inte ändra kopplingarna mellan noderna i topologin, d.v.s. noder som ej är uppkopplade mot varandra i början av simuleringen ska inte bli uppkopplade senare via uppdateringar.

När din labbassistent eventuellt testar din kod så kommer han/hon att använda andra värden på länkkostnaderna i topologin. Du kan själv ändra på värdena och därmed testa din kod mera genom att initialisera datastrukturen **connectcosts[][]** i **RouterSimulator()**. Se dock till att sätta samma länkkostnader åt båda hållen på varje länk; vi använder endast symmetriska länkkostnader i kursen. Det är även här som olika topologier kan kodas.

Simulatoren skickar aldrig **RouterPacket**-meddelanden själv. Din kod måste se till att det utbyts information mellan routrarna i nätverket.

Filen **RouterNode.java** har en variabel kallad **costs** som initialiseras av konstruktorn. Simulatoren sätter länkkostnaderna som hör till en nod via **costs**. Till exempel, i nätverket i figur 1, så sätts länkkostnaderna i nod 1 via en vektor bestående av {1, 0, 1, **RouterSimulator.INFINITY**} som innebär att kostnaden är 1 till nod 0, 0 till noden själv, 1 till nod 2 och ”oändlig” till nod 3. Förändringar i länkkostnaderna sker genom anrop av **updateLinkCost()**, som du ska implementera.

Kostnaderna för alla länkar sätts i konstruktorn för **RouterSimulator**. Längre ned i koden läggs två händelser (events) till som gör att länkkostnaderna kommer att ändras vid specifika tillfällen under simuleringstiden. Länken mellan **event** och **dest** ändras till **cost** vid tiden **evtime**. Observera att länkkostnaderna inte kommer att ändras om du inte ändrar värdet på konstanten **LINKCHANGES** till sant som finns i början på filen. Ändra gärna länkkostnaderna och händelserna här för att testa din kod.

Mer information om implementation av distansvektorroutning kan du läsa om i nedanstående källa:

- C. Hedrick, C. (1988), *RFC 1058 Routing information protocol*. IETF, June 1988.

5.0 Följdfråga om labben

1. "Giftig retur"-algoritmen löser problemet med loopar i de nätverkstopologier som du testat i labben. Ge ett exempel på en topologi, t.ex. en utökning av ett av nätverken i labben, där algoritmen inte skulle fungera och förklara även varför. Ge en lösning på detta nya problem.

6.0 Redovisning

För att slutföra labben måste du demonstrera din lösning för labbassistenten. Vid demonstrationen kan det ske att en annan topologi än de två givna i uppgiften används av labbasistenten för att testa din kod. Var beredd på att förklara vad `printDistanceTable()` visar på skärmen och hur du har implementerat "giftig retur"-algoritmen.

Kontrollera att du har svarat utförligt på frågan som ställs i labbuppgiften och använd vederhäftiga källor som referenser.

Innan du demonstrerar din lösning, ge din labbassistent en papperskopia av din kod (allt ska finnas i `RouterNode.java` enbart) samt `Makefile`-filen.

Lämna sedan in din laborationslösning på papper i ett korrekt ifyllt och underskrivet IDA-labbomslag till laborationsassistenten.