Praktikum Simulative Untersuchung von Protokollfunktionen des Internet

Aufgabenblatt 4[1]

Vorbereitung

Aufgabe 1: Iteratoren
Lesen Sie sich zur Vorbereitung den folgenden Text bitte gut durch und versuchen Sie das Konzept von Iteratoren zu verstehen. Wenn Sie Fragen haben, stellen Sie diese bitte sofort am Anfang des praktischen Termins, da Sie Iteratoren zur Lösung der Aufgabe dringend benötigen:

Sie haben beim letzten Praktikum die Datenstruktur `map` kennengelernt und gelernt, wie Sie über den Schlüssel mit dem `[]`-Operator direkt auf ein Datenelement wie auf ein Arrayelement zugreifen können. Manchmal weiß man jedoch nicht, welche Elemente sich in einer `map` befinden. Jedes Element einer `map` ist ein `pair`, das die beiden Attribute `first` und `second` besitzt. Ersteres enthält den Schlüssel, letzteres den Wert.

Um nacheinander auf die einzelnen Elemente einer `map` zugreifen zu können, benutzt man Iteratoren. Iteratoren sind im Prinzip eine Verallgemeinerung des Zeigerkonzeptes. Zur Anwendung von Iteratoren müssen Sie folgendes wissen:

- Sie können eine Iterator-Variable mit folgendem Typ definieren: *`ContainerTyp`*`::iterator`. Da Container-Typen meist recht kompliziert aussehen, wurden von uns Aliase für die benutzten Typen definiert. Iteratoren gibt es nicht nur für `map`s, sondern auch für andere Container der C++-Standardbibliothek.
- Mit der `map`-Methode `begin()` erhalten Sie einen Iterator, der auf das erste Element der `map` "zeigt".
- Das Element, auf den ein Iterator sich bezieht, erhalten Sie, indem Sie ihn wie einen Zeiger dereferenzieren, d.h. den $*$-Operator benutzen. Wenn Sie direkt auf ein Feld des Elements zugreifen möchten (im Fall von `map`s z.B. `first` oder `second`) können Sie auch wie bei Zeigern den `->`-Operator benutzen.
- Sie können einen Iterator mit dem `++`-Operator inkrementieren. Er zeigt dann auf das nächste Element im entsprechenden Container.
- Die `map`-Methode `end()` gibt einen Iterator zurück, der **hinter** das letzte Element im Container zeigt. Sie dürfen diesen Iterator weder dereferenzieren noch inkrementieren. Allerdings können Sie durch einen Vergleich mit diesem Iterator herausfinden, ob Sie schon alle Elemente einer `map` gesehen haben.
- Häufig benutzt man `for`-Schleifen um mit Iteratoren einen Container zu durchlaufen.

Die Distanzvektoren in dieser Aufgabe sind als `map` mit Schlüsseln vom Typ `NodeAddressT` (letztendlich eine ganze Zahl) und Datenelementen vom Typ `RouteInfo` definiert. Der Schlüssel bezeichnet hierbei den Zielknoten einer Route, der Wert enthält die nötigen Informationen (`nextHop`, `cost`) über die Route.

Damit Sie den recht kompliziert aussehenden Ausdruck für den `map`-Typ nicht immer hinschreiben müssen, haben wir ein Typalias mit dem Namen `DVT` (Distance Vector Type) definiert.

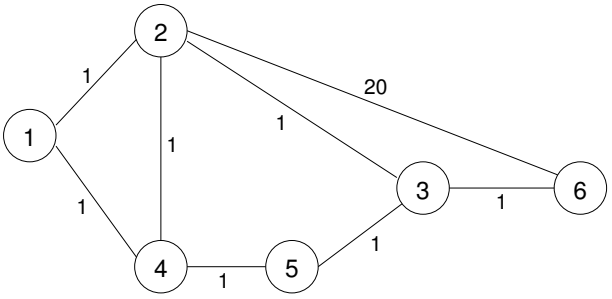
Damit das Ganze nicht zu abstrakt wird, hier nun ein Beispiel, wie Sie die Kosten zu jedem Ziel in einem Distanzvektor an das Environment ausgeben könnten:

```
1  for (DVT::iterator i = myDV.begin(); i != myDV.end(); ++i) {
2      ev << "Ziel"<< i->first
3         <<",Kosten:"<< i->second.cost;
4  }
```

[1] Stand: 12. Oktober 2023

Aufgabe 2: Distance Vector Routing
In der folgenden Grafik sehen Sie wieder das Ihnen schon bekannte Netz. Führen Sie nun den Distance Vector Routing Algorithmus für den Zielknoten 6 aus. Fertigen Sie dafür eine Tabelle an, die für jeden Schritt für alle Knoten die Kosten und den Next-Hop-Knoten zum Zielknoten 6 enthält. Gehen Sie der Einfachheit halber von einem getakteten System aus.

Aufgabe 3: Count-to-Infinity
Was versteht man unter dem Count-to-Infinity-Problem und in welchem Fall tritt es auf?



Aufgabe 4: Konvergenzgeschwindigkeit
Ausgehend von einem getakteten System, von welcher Eigenschaft des Netzwerkes hängt die Konvergenzgeschwindigkeit ab? Welcher Zusammenhang besteht zwischen dieser Eigenschaft und der Konvergenzgeschwindigkeit? Geben Sie hierfür eine informale Begründung an.

Praktischer Teil

Dieses Mal werden Sie das Distance Vector Routing implementieren. Dafür müssen Sie die Datei `routing/DistanceVectorRoutingDaemon.cc` erweitern. Dabei sollen Sie zunächst die nicht-optimierte Version implementieren. Die Implementierung der optimierten Version ist optional. Die Sourcen für diese Aufgabe befinden sich im Verzeichnis `protsim04`.

Denken Sie bitte – wie immer bei den praktischen Aufgaben – an aussagekräftige Debug-Ausgaben.

Aufgabe 5: Distance Vector Routing (nicht-optimiert)
Für diese Variante erweitern Sie bitte den **ersten** Teil der Datei.

Für diese Variante benötigen wir eine Datenbank, die von einigen Routern im Netzwerk einen Distanzvektor enthält. Diese Distance Vector Database ist auch wieder eine `map`. Ihr Schlüssel ist die Adresse des Routers, von dem der Distanzvektor stammt, der Wert ist der Distanzvektor (also vom Typ `DVT`). Die Distance Vector Database hat den Namen `DVdb` und ist vom Typ `DVdbT`.

Bei jeder Routenänderung, die entweder lokal durch eine `LocalLinkChangeNotif` oder von anderen Routern durch eine `DistanceVectorMessage` mitgeteilt wird, werden alle Routen neu berechnet. Dies geschieht in der Methode `calculateRoutes()`, die Sie nun bitte ergänzen. Alle weiteren notwendigen Hinweise stehen im Quelltext.

Aufgabe 6: Distance Vector Routing – Count-to-Infinity
Probieren Sie das Programm aus der letzten Aufgabe aus. Lassen Sie es mindestens so lange laufen, bis der Schwall von Distance-Vector-Nachrichten abgeflaut ist. Unter Umständen ist es vorteilhaft, das Senden einzelner Applikationen abzuschalten, um die Ping- und Pong-Nachrichten besser verfolgen zu können (die Applikationen antworten aber weiterhin auf Pings). Deshalb sind zunächst alle Applikation außer Applikation 2 in Knoten 1 abgeschaltet. Schalten Sie nach Bedarf andere Applikationen wieder ein. Dies geschieht in `omnetpp.ini`, indem man den Parameter `interval` der entsprechenden Applikation auf einen Wert ungleich 0 setzt (z.B. `*.Node2.application[1].interval = 0.1;`). Beachten Sie bei Ihren Tests bitte, dass der Link zwischen Knoten 2 und 6 die Kosten 20 und alle anderen Links die Kosten 1 haben.

Nachdem der Routing-Algorithmus nun eingeschwungen ist, ändern Sie bitte, so wie im zweiten Termin, die Kosten zwischen Knoten 3 und 6 in **beide** Richtungen auf 1000. Beobachten Sie bitte das Verhalten von Nachrichten zwischen Knoten 1 und 6. Wie können Sie dieses Verhalten erklären? Setzen Sie danach die Kosten wieder zurück auf 1 und erklären Sie das Verhalten.

Aufgabe 7: (optional) Distance Vector Routing (optimiert)

Wenn Sie noch Zeit haben, implementieren Sie bitte auch die optimierte Version des Algorithmus. Für diese Variante entfernen Sie bitte die Kommentarzeichen vor `#define OPTIMIZED_DV` und erweitern Sie den **zweiten** Teil der Datei.

Für die optimierte Fassung genügt es, dass wir nur den eigenen Distance Vector speichern. Dies geschieht in der Variable `DVdb`, die in diesem Fall vom Typ `DVT` ist.

Die Distance Vector Database soll mit den direkten Nachbarn initialisiert werden. Dies geschieht in der Methode `initializeDataBase()`. Ergänzen Sie bitte den entsprechenden Code.

Routen können auf zwei Arten geändert werden, entweder bei einer lokalen Linkänderung oder bei Änderungen in anderen Routern.

Erweitern Sie bitte die entsprechenden Methoden `calculateRoutes(LocalLinkChangeNotif)` und `calculateRoutes(DistanceVectorMessage)` entsprechend der dort gegebenen Hinweise.

Wiederholen Sie anschließend das Experiment aus Aufgabe 6. Können Sie sich das Verhalten erklären?