Index

Appendix

Abstract

In this paper, we mainly discuss the verification of two ns2 based Quality-of –Service routing (QoSr) simulation packages, *qosrns* package and *perd* package. These two packages are developed to evaluate the QoSr performance in an intra-domain. The verification result shows the QOSPF package works pretty well except that a further work is needed on its algorithm, while PERD seems not so good. Some background will be introduced before this.

1. Introduction

One of the important problems in QoS service offering is how to determine a route that satisfies multiple constraints (or QoS requirements) while simultaneously achieving high utilization of network resources. This problem is addressed by QoS routing. In general, routing consists of two basic tasks: distributing the state of the network and searching the feasible path with respect to given constrains.

QoS routing has recently received substantial attention. Most of the current proposals in this context rely on the link state approach. An overview for its framework can be seen in [1]. The benefits of QoS routing, in terms of improved network utilization and user service levels have been established through many research studies [2].

Despite many benefits brought with QoSr, doubts regarding the feasibility of implementing QoS routing protocols in IP networks remain because of the potential additional costs that support for QoS routing entails. These added costs have two major components: computational cost and protocol overhead. The former is due to the more sophisticated and more frequent route computations, while the latter is caused by the need to distribute updates on the state of network resources. Such updates increase network traffic and processing overhead at routers. Simulation can help us to investigate the performance of the network with QoSr enabled.

It is important to evaluate the LS approach based QoS routing performance. Two new QoS routing modules, *qosrns* and *perd,* are developed for this purpose. These two packages are based on the ns2 simulator, providing new functions. The package *qosrns* is developed according to RFC2676 [3]. And *perd* is an extension module derived from a new routing idea, PERD. About PERD routing principles, more will be introduced latter.

In this paper, after introducing some background for LS based QoSr and for these two packages, we present the implementation and verification for them. Verification results, conclusion and analysis are discussed. Some idea for QoSr and future work are shown at the last.

2. QoS Routing Background

Quality of Service (QoS) refers to the capability of a network to provide better service to selected network traffic over various technologies, including Frame Relay, Asynchronous Transfer Mode (ATM), Ethernet and 802.1 networks, SONET, and IP-routed networks that may use any or all of these underlying technologies. The primary goal of QoS is to provide priority including dedicated bandwidth, controlled jitter and latency (required by some real-time and interactive traffic), and improved loss characteristics. Also important is making sure that providing priority for one or more flows does not make other flows fail, that is, balancing the load.

QoS based routing is a routing mechanism under which flow paths are determined based on some knowledge of resource availability in the network as well as the QoS requirement of flows. It must extend the current routing paradigm in three basic ways: First, multiple paths between node pairs will have to be calculated according to different requirements and available resources. Some of these new classes of service will require the distribution of additional routing metrics, e.g. delay, and available bandwidth. If any of these metrics change frequently, routing updates can be becoming more frequently thereby consuming network bandwidth and router CPU cycles. Second, today's opportunistic routing will shift traffic from one path to another as soon as a "better" path is found. The traffic will be shifted even if the existing path can meet the service requirements of the exiting traffic. If routing calculation is tied to frequently consuming resources (e.g. available bandwidth) this change will happen more often and introduce routing oscillations as traffic shifts back and forth between alternate paths. Furthermore, frequently changing route may increase the variance of the delay and jitter experienced by the end user. Third, today's optimal path routing algorithms usually do not support alternate routing. If the best exiting path cannot admit a new flow, the associated traffic cannot be forwarded even if an adequate path exits [3] .

The problem of QoSr is difficult due to a number of reasons. First, distributed applications such as Internet phone and distributed games have very diverse QoS constraints on delay, delay jitter, loss ratio, bandwidth, etc. Multiple constraints often make the routing problem intractable. For example, finding a feasible path with two independent path constraints is NP-complete [4]. Second, any future integrated service network is likely to carry both QoS traffic and best effort (BE) traffic, which makes the issue of performance optimization complicated. It is hard to determine the best operating point for both types of traffic if their distributions are independent. Although the QoS traffic will not be affected due to resource reservation, the throughput of the best-effort traffic will suffer if the overall traffic distribution is misjudged. Third, the network state changes dynamically due to transient load fluctuation, connections in and out, and links up and down. The growing network size makes it increasingly difficult to gather up-to-date state information in a dynamic environment, particularly when the wireless communication is involved. The performance of a QoS routing algorithm can be seriously degraded if the state information being used is outdated.

2.1 Routing strategies

There are three routing strategies, *source routing, distributed routing* and *hierarchical routing*.

In the source routing, each node maintains the complete global state, including the network topology and the state information of every link. Based on the global state, a feasible path is locally computed at the source node. A control message is then sent out along the selected path to inform the intermediate nodes of their precedent and successive nodes. So under this condition, a link-state protocol is necessary to be used to update the global state from time to time at every node.

In the distributed routing, the path is computed by a distributed computation. Control messages are exchanged among the nodes and the state information kept at each node is collectively used for the path search. Most distributed routing algorithms need a distance vector protocol (or a link-state protocol) to maintain a global state in form of distance vectors at every node. Based on the distance vectors, the routing is done on a hop-by-hop basis.

In the hierarchical routing, nodes are clustered into groups (logical nodes), which are further clustered into higher-level groups (also logical nodes) recursively, creating a multi-level hierarchy. Each physical node maintains an aggregated global state. This state contains the detailed state information about the nodes in the same group and the aggregated state information about the other groups. Usually the source routing is used to find a feasible path on which some nodes are logical nodes representing groups. A control message is then sent along this path to establish the connection.

In our simulation *qosrns* and *perd* packages, LS protocol based source routing is used.

2.2 Routing protocol

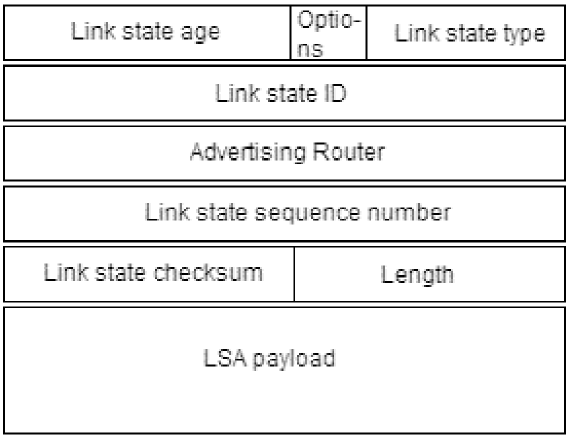
There are kinds of routing protocols. Usually link state protocols are used for source routing while distance vector protocols are used for distributing routing.

We focus on OSPF and OSPF extensions because of our source routing simulation and *qosrns* and *perd* simulation packages. Package *qosrns* is developed earlier than *perd,* and it makes its basis on QOSPF.

Open Shortest Path First (OSPF) is a routing protocol developed for Internet Protocol (IP) networks by the Interior Gateway Protocol (IGP) working group of the Internet Engineering Task Force (IETF). OSPF is a link-state routing protocol that calls for the sending of link-state advertisements (LSAs) to all other routers within the same hierarchical area. Information on attached interfaces, metrics used, and other variables are included in OSPF LSAs. As OSPF routers accumulate link-state information, they use the some algorithm (usually Dijkstra algorithm) to calculate the shortest path to each node.

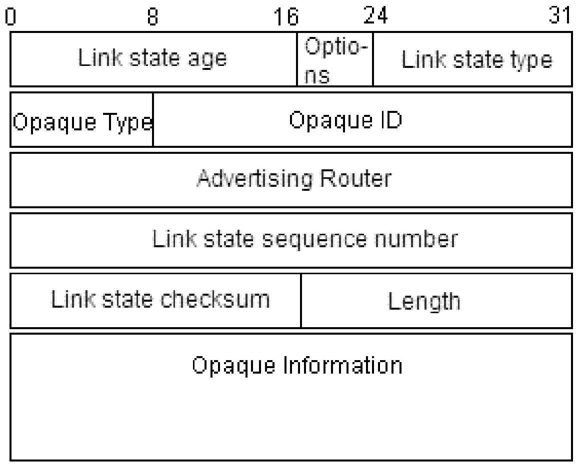
OSPF can operate within a hierarchy. An OSPF AS can consist of multiple areas Linked by routers.

“Native” OSPF LSA is shown as following:



TE is the process of selection the paths chosen by data traffic in order to facilitate efficient and reliable network operations while simultaneously optimization the resource utilization and traffic performance. OSPF-TE is built on the native OSPF foundation, processing the same as native OSPF.

OSPF- TE is using opaque LSA:



The draft [3] defines a new LSA, i.e., traffic engineering LSA. The LSA contains Link TLV, which includes the following sub-TLVs:

1- Link type (1 octet)
2- Link ID (4 octets)
3- Local interface IP address (4 octets)
4- Remote interface IP address (4 octets)
5- Traffic engineering metric (4 octets)
6- Maximum bandwidth (4 octets)
7- Maximum reservable bandwidth (4 octets)

8- Unreserved bandwidth (32 octets)
9- Resource class/colour (4 octets)

Based on the previous draft, the draft [9] proposes corresponding extension to OSPF for support of Traffic Engineering on a per-Class-Type basis. The draft add new sub TLVs:
TBD1 – Unreserved Bandwidth for Class-Type 1 (32 octets)
TBD2 – Unreserved Bandwidth for Class-Type 2 (32 octets)
TBD3 – Unreserved Bandwidth for Class-Type 3 (32 octets)
Meanwhile: sub-TLV 7 (maximum available bandwidth, which can be reserved) is referred to “Maximum available Aggregate Bandwidth, which can be reserved”; sub-TLV 8 (Unreserved bandwidth) is referred to “Unreserved Bandwidth for Class-Type 0”.

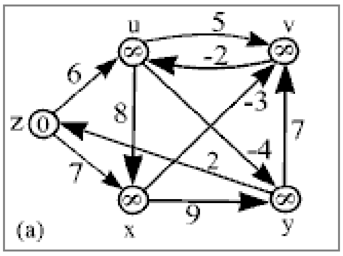
2.3 Route computation

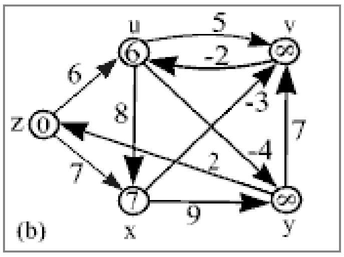
Routing QoS sensitive applications consists in finding a path, which complies with end-to-end constrains derived from the QoS user needs. This task translates into finding routes in a network scenario of multiple metrics. In general, routing optimally a flow with such end-to-end constraints (multiple metrics) is known to be an NP-complete problem. Many work have been done in this field. [6]

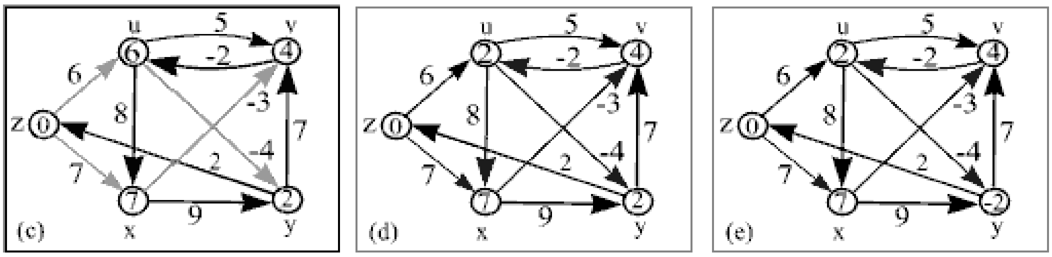
In our *qosrns* and *perd* package, the simulation routing module is made based on Bellman-Ford (BF) algorithm. This is because in considering routing with multiple constrains, it has been noticed that the Bellman-Ford (BF) algorithm can potentially solve a two metric routing problem, when one of the metric is the hop count. This fact prefers the Bellman-Ford than the Dijkstra algorithm. This is because Dijkstra algorithm does not search for a minimum path cost in ascending order of hops, as Bellman-Ford algorithm does.

The Bellman-Ford algorithm solves the single-source shortest-paths problem in the more general case. Weights may be negative. The algorithm initializes the distance to the source vertex to 0 and all other vertices to ∞. It then does V-1 passes (V is the number of vertices) over all edges relaxing, or updating, the distance to the destination of each edge. Finally it checks each edge again to detect negative weight cycles, in which case it returns false. The time complexity is O(VE), where E is the number of edges.

Following is a simple example for it:







The source is vertex z. The d values are shown within the vertices and blue edges indicate the π values (for example in figure (b) we see that $\pi[u] = z$ in $\pi[x] = z$). In this particular example, each pass relaxes the edges in lexicographic order:
(u, v), (u, x), (u, y), (v, u), (x, v), (x, y), (y, v), (y, z), (z, u), (z, x).

Above pictures (a-e) shows how the execution of the Bellman-Ford algorithm works on a graph with 5 vertices. (a) The situation just before the first pass over the edges. (b)-(e) The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

In QoSr simulation, the Bellman-Ford (BF) shortest path algorithm is adapted to compute paths of maximum available bandwidth for all hop counts. It is a property of the BF algorithm that, at its iteration, it identifies the optimal (e.g. maximal bandwidth) path between the source and each destination, among paths of at most h hops. In other words, the cost of a path is a function of its available bandwidth, i.e., the smallest available bandwidth on all links of the path, and finding a minimum cost path amounts to finding a maximum bandwidth path. However, because the BF algorithm progresses by increasing hop count, it essentially provides for free the hop count of a path as a second optimization criterion.

2.4 Path selection

The goal for the path selection is to select the feasible and efficient path.

Routing algorithms can achieve resource efficiency by limiting resource consumption while balancing the network load. For traffic with bandwidth guarantees, since the requested bandwidth has to be reserved, a natural way to limit resource consumption is to select a path with as few hops as possible. The network load can be balanced by selecting the least loaded path. However, these two “optimality” criteria sometimes conflict, for example a path with the fewest hops may contain heavily loaded links.

Several path selection algorithms have been proposed in the literature, including widest-shortest path [5], shortest-widest path [8] and utilization based path selection algorithms [7].

In our simulation, we just focus on the following two path selection algorithms:

Shortest-widest path: a path with the minimum hop count among all feasible paths. If there are several such paths, the one with the maximum available bandwidth is selected. If there are several such paths with the same bandwidth, one is randomly selected.
Widest-shortest path: a path with the maximum bandwidth among all feasible paths. If there are several such paths, then the one with the minimum hop count is selected. If there are several such paths with the same hop count, one is randomly selected.

A shortest-widest path algorithm usually based on the Bellman-Ford algorithm is proposed in [5][10]. We can also modify Dijkstra's algorithm to find the widest-shortest paths by defining two distance functions: the hop count is the primary function and another secondary function is added. When selecting the next node to mark, one selects the node with the minimal hop count. If several nodes have the same minimal hop count, one selects the node with the largest value of another function. The algorithm terminates when the destination is reached. [4]

A simple widest-shortest path algorithm applies Dijkstra's algorithm twice. First find a widest path; assume its bandwidth is B. Then find a shortest path with bandwidth B by using Dijkstra's algorithm on a network that only includes links of bandwidth B or higher. Note that the single pass link state shortest path algorithm given in [8] does not always find the shortest-widest path [4].

In RFC 2676, the extended FB algorithm is used to generate a matrix:

	Hops number 1	……	Max hops number
Destination 1	(bw, neighbour)	……	(bw, neighbour)
……	……	……	……
Destination n	(bw, neighbour)	……	(bw, neighbour)

While choosing the maximum bandwidth for each hops number, the shortest path is also got because of BF algorithm's increasing hop count progress. It is a widest-shortest way.

3. Verification For *qosrns* and *perd* packages

3.1 About **qosrns** *Package*

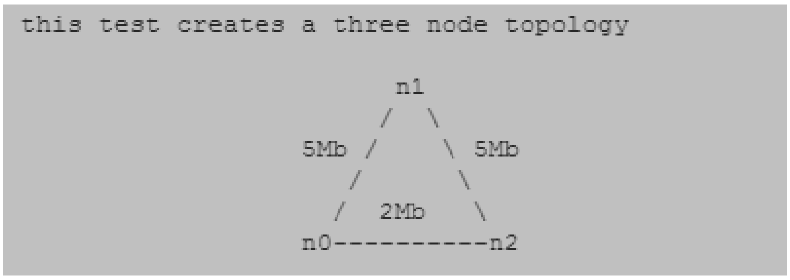
qosrns package implementation of QOS extensions to OSPF allows one to simulate different strategies. This package implements a new routing module for ns2.

It mainly consists of following three components:
1) A new routing logic, which computes shortest path between two nodes.
2) An agent — Agent/QOSPF, which exchanges link state routing packets between various nodes.
3) Routing module RtModule/QOSPF, which interfaces the above with Node.

3.2 Verification For qosrns Package

Test1

This test1 aims to test if its on-demand update is correct. Test1 is based on the example configuration file provided together with *qosrns* package.



```
this test creates a three node topology

                        n1
                       /  \
                  5Mb /    \ 5Mb
                     /      \
                    /  2Mb   \
                  n0----------n2
```

This test uses admission control (ADC) to make path reservation.
The following four sessions are created:

Sequence.	Source	Destination	Bandwidth	Flow ID
1	n0	n2	2.5M	1
2	n0	n2	1.5M	2
3	n0	n2	1.5M	3
4	n0	n2	3.0M	4

After each session is created, an on demand update procedure is called. Then we expect the path for the next session should be calculated according to the updated LS table.

Test1 result:

Flow ID	Path elected
1	n0-----n1-----n2
2	n0-----n2
3	n0-----n1-----n2
4	NULL

Result analysis:
After each session is created the available bandwidth of each link will change. If the on demand update is correct, the database in which the link state is stored will change correspondingly. We can get the path for each session by manually calculation for available bandwidth.

The expected result is:

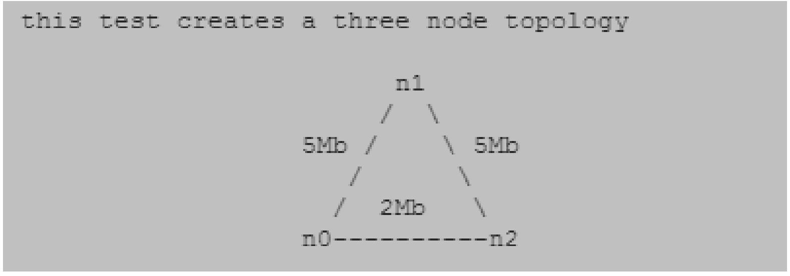
Flow ID	Path elected
1	n0-----n1-----n2
2	n0-----n2
3	n0-----n1-----n2
4	Unsuccessful as not enough bandwidth

We can see the result expected accords with the real test result, so the on demand update for the link state works well.

Test2

To make sure the update of the LS stored in QoSr node is really made on demand because of the procedure *bw-changed{}*, which is intended to inform LS agent about bandwidth changes ,we cancel the call for this procedure in this test.

In this test, the topology used and the session created are as the same as the former test. Also the path reservation is made by admission (ADC).



```
this test creates a three node topology

                        n1
                       /  \
              5Mb /      \ 5Mb
                   /        \
                  /   2Mb    \
              n0----------n2
```

And the following four sessions are created:

Sequence	Source	Destination	Bandwidth	Flow ID
1	n0	n2	2.5M	1
2	n0	n2	1.5M	2
3	n0	n2	1.5M	3
4	n0	n2	3.0M	4

Because the on demand update procedure is cancelled in this test, a different result is expected.

Then we get the result:

Flow ID	Path elected
1	n0-----n1-----n2
2	n0-----n2
3	n0-----n1-----n2
4	NULL

Result analysis:
We see the result is the same as that from test1. That means update is still successful.
While the result we expect is:

Flow ID	Path elected
1	n0-----n1-----n2
2	n0-----n2
3	n0-----n2
4	n0-----n1-----n2

The result shows the on demand procedure *bw-changed{}* is not necessary in the test file. This brings the doubt on the realization for update LS information.

Another procedure *get-bw{}* is necessary to implement while using QOSPF module. This procedure is closely relative to ADC (admission control). ADC is used to make path reservation. The way used to make reservation should have nothing to do with the routing module itself.

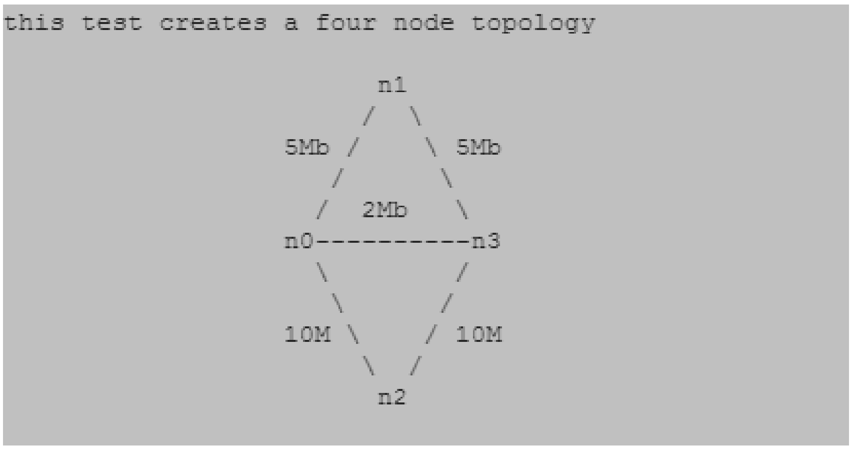
This procedure *get-bw{}* returned the available bandwidth, i.e. bandwidth left after path reservation. So there maybe some function overlapping between procedure *get-bw{}* and procedure *bw-changed{}*.

From the test1 and test2, we can see that, although *qosrns* package update its LS information successfully, its implementation should be improved.

Test3

This test2 file is configured to verify which path selection algorithm is used in this package, shortest - widest or widest – shortest.

The topology used in this test is:



```
this test creates a four node topology

                        n1
                       /  \
                  5Mb /    \ 5Mb
                     /      \
                    /  2Mb   \
                  n0----------n3
                    \        /
                     \      /
                  10M \    / 10M
                       \  /
                        n2
```

We create the session as following:

Source	Destination	Bandwidth	Flow ID
n0	n3	2.5M	1

Why we just create one session is because one is enough to make test for path selection.

The simulation result we get is:

Flow ID	Path elected
1	n0-----n1-----n3

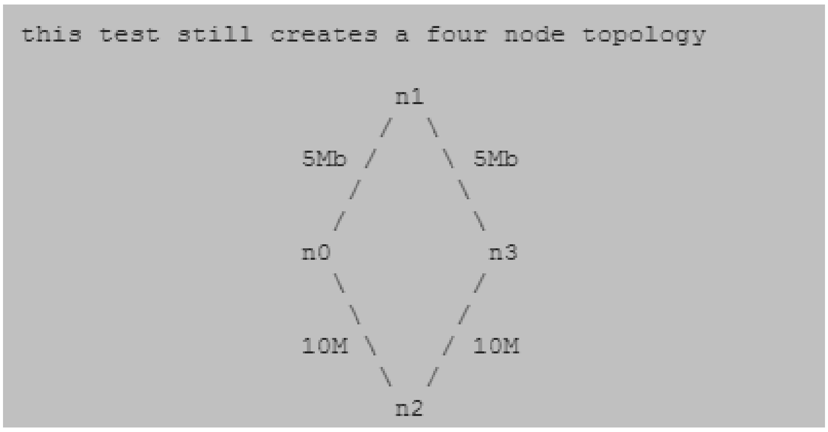
Result analysis:

We just create one session with 2.5Mbps. Because the bandwidth needed for this session is 2.5 Mbps, so it will not take the route n0-----n3. If the shortest-widest algorithm is used, when the hop count is the same in our context (n0-----n1-----n3 and n0-----n2-----n3), the path with the largest bandwidth will be chosen, that is, the path from n0 to n3 via n2 will be chosen. But the result shows that from n0 to n3 via n1 is chosen, we can get the conclusion that the shortest-widest algorithm is not used.

Test4

From the test2 we can see *qosrns* package does not use the shortest-widest algorithm. So we suppose that it uses the widest-shortest algorithm.

The topology used here is shown as following:



```
this test still creates a four node topology

                   n1
                 /   \
          5Mb /       \ 5Mb
             /         \
            /           \
          n0             n3
            \           /
             \         /
          10M \       / 10M
                \   /
                  n2
```

We create the session as following (one session is enough):

Source	Destination	Bandwidth	Flow ID
n0	n3	2.5M	1

The simulation result we get is:

Flow ID	Path elected
1	n0-----n1-----n3

Result analysis:

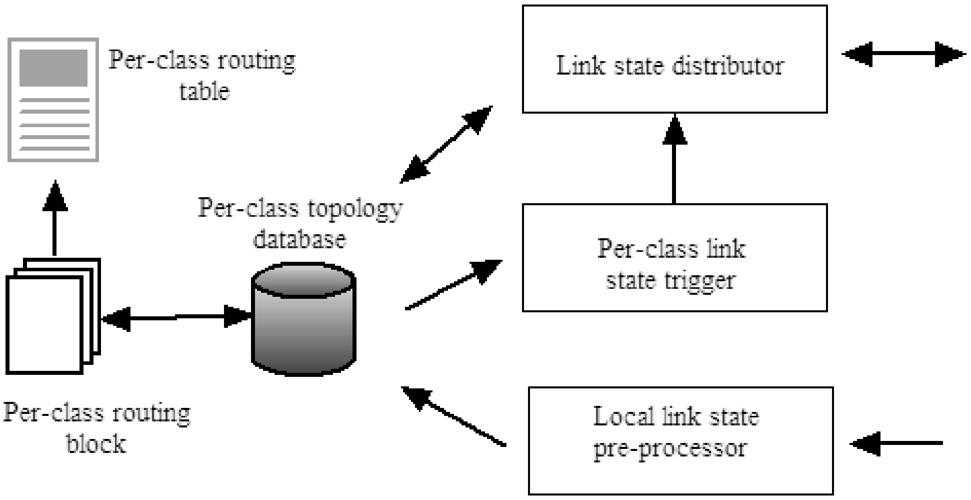
We just create one session. If the widest-shortest algorithm is used, when the hop number is the same (n0-----n1-----n3 and n0-----n2-----n3), the path with the largest width will be chosen, that is from n0 to n3 via n2 will be chosen.
But the result shows that from n0 to n3 via n1 is chosen, we can get the conclusion that the widest-shortest algorithm is not used.
From test3 and test2, we can see neither widest-shortest algorithm nor shortest-widest algorithm is used in *qosrns* package.

3.3 About PERD Routing mechanism

PERD is a routing mechanism, that is, per-class routing based on per-class dissemination. Network resource for each class are maintained in an extended link state database, which contains available link resources for each class. When the resources of a class change on a link, new link state will be advertised to the network. In addition, a trigger mechanism can be applied in order to reduce advertisement cost.

In diffserv network, class in PERD can represent a PHB or a group of PHB. For example, given 2 PHBs, PERB may use one class to represent the two PHBs. PHBs are realized by traffic scheduling mechanisms (e.g. CBQ, WFQ, PQ, RIO). Hence, the resources of PHBs can map to the parameters of the scheduling mechanisms. For example, the recourses can refer to maximum bandwidth, available bandwidth, and buffer (refer to delay) of queues.

The following figure gives the architecture of PERD based on link state protocol, e.g. OSPF.



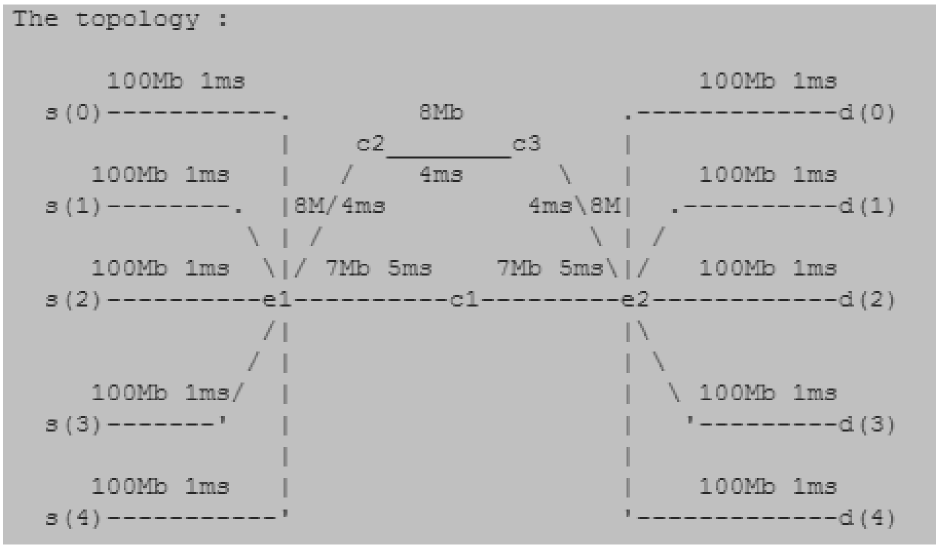
In this architecture, a “local link state pre-processor” determines when and how to input local link information into “per-class topology database”. One of the key functions of the processor is to transform various link state information into the variables that can be used by routing. The “per-class link state trigger” is to control when to update the LS information.

3.4 Verification For perd Package

Test1

This test file is to verify which routing selection algorithm *perd* package is using, shortest - widest or widest – shortest.

Topology used for test1 is as following:



```
The topology :

      100Mb 1ms                             100Mb 1ms
  s(0)-----------.        8Mb          .-------------d(0)
                 |    c2________c3     |
     100Mb 1ms   |   /    4ms      \   |    100Mb 1ms
  s(1)--------.  |8M/4ms         4ms\8M|  .----------d(1)
               \ | /                 \ | /
     100Mb 1ms  \|/ 7Mb 5ms    7Mb 5ms\|/   100Mb 1ms
  s(2)----------e1----------c1---------e2------------d(2)
                /|                     |\
               / |                     | \
     100Mb 1ms/  |                     |  \ 100Mb 1ms
  s(3)-------'   |                     |   '---------d(3)
                 |                     |
     100Mb 1ms   |                     |    100Mb 1ms
  s(4)-----------'                     '-------------d(4)
```

In this test file, e1 and e2 are configured as edge router, and c1, c2 and c3 are configured as core router. Access links have enough bandwidth, 100M. The bandwidths between routers are shown as above.

Five sessions are created:

Seq.	Traffic class	Source	Destination	Bandwidth	Flow ID
1	AF1	s(0)	D(0)	5.0Mb	1
2	AF1	s(1)	D(1)	3.5Mb	2
3	AF1	s(2)	D(2)	4.5Mb	3
4	AF1	s(3)	D(3)	3.0Mb	4
5	AF1	s(4)	D(4)	2.5Mb	5

We use same traffic class (AF1 in this test) because this test is to tell which routing selection algorithm the *perd* package uses. Different traffic class may result in difficult analysis for routing result from test.

Routing result from test:

Route Followed	Notation
e1-----c1-----e2	Session 1 created successfully
e1-----c2-----c3-----e2	Session 2 created successfully
e1-----c2-----c3-----e2	Session 3 created successfully
NULL	Creation for session 4 failed
NULL	Creation for session 5 failed

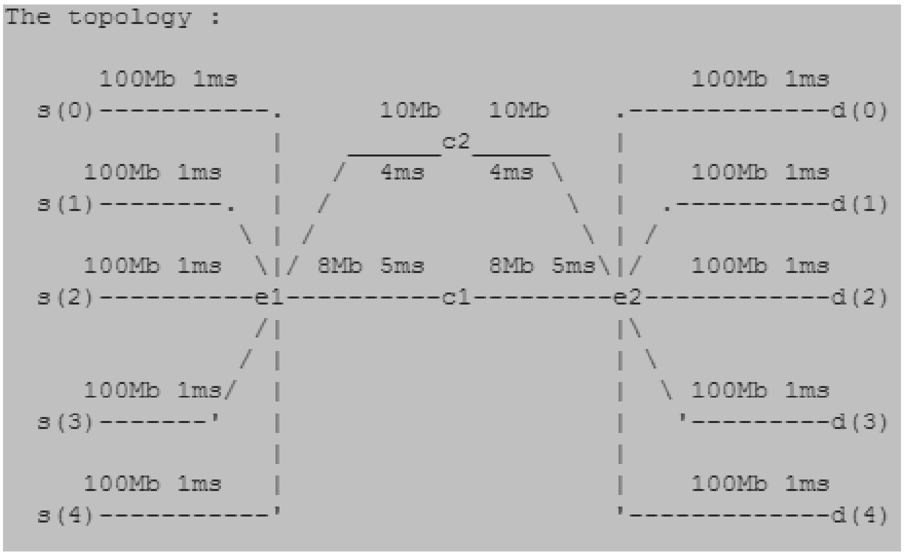
Result analysis:

We can see from the test result the session 1 (from s(0) to d(0), flow id is 0) chooses the route e1-----c1-----e2. The shortest path is taken. From this, we can suppose this *perd* package used the shortest-widest algorithm to make path selection. This proves that widest-shortest algorithm is not used.

Test2

From test 1, we suppose that the shortest- widest algorithm is used in *perd* package for path selection. This test is to verify if the shortest –widest algorithm is used. If shortest –widest algorithm is used, when there are several paths with the same hop count, the path, which has the widest bandwidth, will be selected.

Topology used by test2 is as following:



```
The topology :

      100Mb 1ms                             100Mb 1ms
  s(0)-----------.      10Mb   10Mb    .------------d(0)
                 |    ______c2_____    |
     100Mb 1ms   |   /  4ms    4ms \   |    100Mb 1ms
  s(1)--------.  |  /               \  |  .---------d(1)
               \ | /                 \ | /
     100Mb 1ms  \|/ 8Mb 5ms    8Mb 5ms\|/   100Mb 1ms
  s(2)----------e1----------c1---------e2-----------d(2)
                /|                     |\
               / |                     | \
     100Mb 1ms/  |                     |  \ 100Mb 1ms
  s(3)-------'   |                     |   '--------d(3)
                 |                     |
     100Mb 1ms   |                     |    100Mb 1ms
  s(4)-----------'                     '------------d(4)
```

In this test file, e1 and e2 are configured as edge router, and c1, c2 are configured as core router. Access links have enough bandwidth, 100M. The bandwidths between routers are shown as above. We can see the route, which is from e1 to e2 via c2, is better than the route, which is from e1 to e2 via c1. Its delay is smaller and its bandwidth is larger.

Also five sessions are created:

Seq.	Traffic class	Source	Destination	Bandwidth	Flow ID
1	AF1	s(0)	d(0)	5.0Mb	1
2	AF1	s(1)	d(1)	3.5Mb	2
3	AF1	s(2)	d(2)	4.5Mb	3
4	AF1	s(3)	d(3)	3.0Mb	4
5	AF1	s(4)	d(4)	2.5Mb	5

Because of the same reason we mentioned in test1, we use same traffic class (AF1 in this test) to tell which routing selection algorithm the *perd* package use.

Test2 result:

Route Followed	Notation
e1-----c1-----e2	Session 1 created successfully
e1-----c2-----e2	Session 2 created successfully
e1-----c2-----e2	Session 3 created successfully
e1-----c1-----e2	Session 4 created successfully
NULL	Creation for session 5 failed

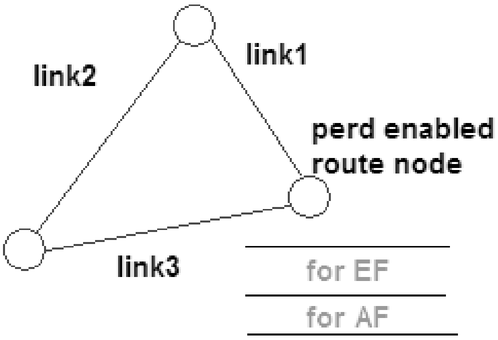
Result analysis:

If the shortest – widest algorithm is used by *perd*, then when the above topology is used, the better path (from e1 to e2 via c2) should be taken. That is e1-----c2-----e2 is expected for session 1. But we can see from the test result the session 1 (from s(0) to d(0), flow id is 0) chooses the route e1-----c1-----e2. The path with larger delay and smaller bandwidth is taken. From this, we can draw the conclusion that *perd* package does not use the shortest –widest algorithm, neither the widest –shortest from the result of test1.

From this test2 result, we can see the only session 5 cannot be created because the left bandwidth is not enough. According to this and correspond result in test1, we can get the conclusion that in *perd* package, link state for one class update correctly.

Test3

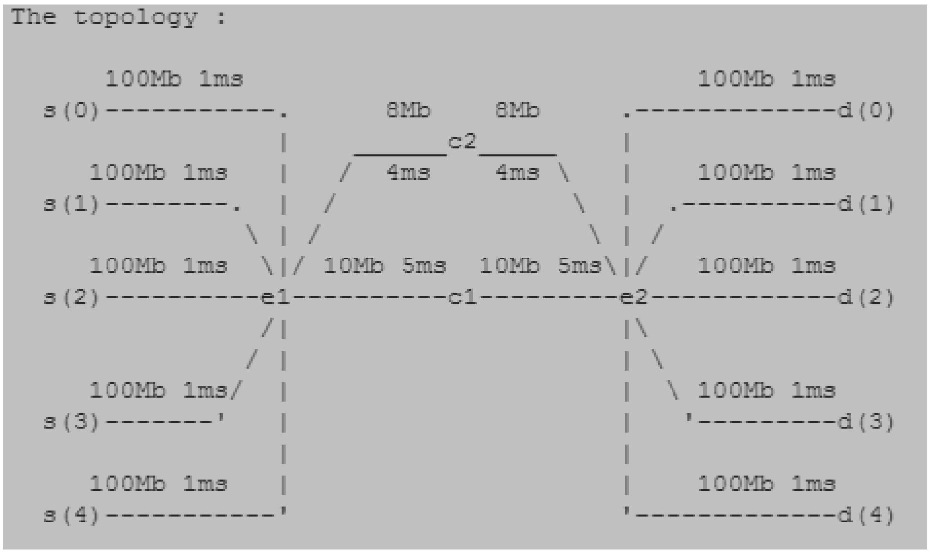
The most important idea in principle for PERD routing mechanism is that route selection is made differently according to different traffic class, which may represent one or several PHBs.



As shown above, the links between the route nodes, which is PERD mechanism enabled, have been set different available bandwidth for each traffic class in advance. For example, a 10M link, it is shared by EF and AF, 20% and 80% separately. The portion for each is fixed. When routing, the routing table should be separated too. The routing algorithms can be the same for all classes. It also can be different for different classes.

This test3 and following test4 will verify if the routing table is separated in this *perd* package.

The topology used in test3 is quite similar with the test1 and test2.



```
The topology :

      100Mb 1ms                                     100Mb 1ms
  s(0)-----------.      8Mb    8Mb      .-------------d(0)
                 |    ______c2_____    |
     100Mb 1ms   |   /  4ms    4ms \   |    100Mb 1ms
  s(1)--------.  |  /               \  |  .----------d(1)
               \ | /                 \ | /
     100Mb 1ms  \|/ 10Mb 5ms  10Mb 5ms\|/   100Mb 1ms
  s(2)----------e1----------c1---------e2------------d(2)
                /|                     |\
               / |                     | \
     100Mb 1ms/  |                     |  \ 100Mb 1ms
  s(3)-------'   |                     |   '---------d(3)
                 |                     |
     100Mb 1ms   |                     |    100Mb 1ms
  s(4)-----------'                     '-------------d(4)
```

Tset3 contains two steps. We do not separate the different traffic class for different bandwidth portions for each link in this test3 file. We will see from test3 result that if the *perd* package can separate them by itself.

In the first step, five sessions list below are created:
We create different traffic class for five sessions, one is EF, three are AF1 and the fifth is BE

.

Seq.	Traffic class	Source	Destination	Bandwidth	Flow ID
1	EF	s(0)	d(0)	5.0Mb	1
2	AF1	s(1)	d(1)	3.5Mb	2
3	AF1	s(2)	d(2)	4.5Mb	3
4	AF1	s(3)	d(3)	3.0Mb	4
5	BE	s(4)	d(4)	2.5Mb	5

Test3 Step1 result:

Route Followed	Notation
e1-----c1-----e2	Session 1 created successfully
e1-----c1-----e2	Session 2 created successfully
e1-----c2-----e2	Session 3 created successfully
e1-----c2-----e2	Session 4 created successfully
NULL	Creation for session 5 failed

In the second step of test3, we create different traffic class for five sessions, none is EF, four are AF1 and the fifth is BE.

Seq.	Traffic class	Source	Destination	Bandwidth	Flow ID
1	AF1	s(0)	d(0)	5.0Mb	1
2	AF1	s(1)	d(1)	3.5Mb	2

3	AF1	s(2)	d(2)	4.5Mb	3
4	AF1	s(3)	d(3)	3.0Mb	4
5	BE	s(4)	d(4)	2.5Mb	5

Test3 Step2 result:

Route Followed	Notation
e1-----c1-----e2	Session 1 created successfully
e1-----c1-----e2	Session 2 created successfully
e1-----c2-----e2	Session 3 created successfully
e1-----c2-----e2	Session 4 created successfully
NULL	Creation for session 5 failed

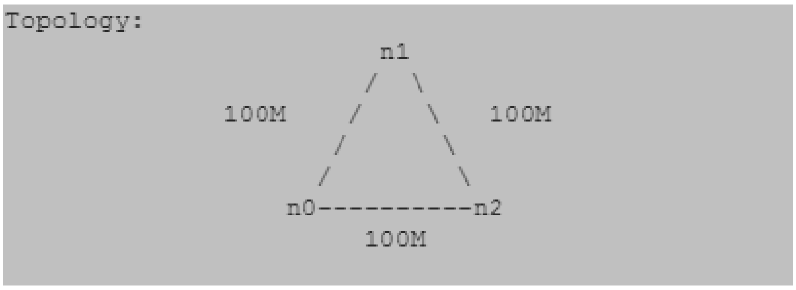
Result analysis:
We can see, although we change the traffic class for session, the routing result is the same. If there is preset for the width allocation in the inner *perd* package, then there should be some difference at least for the flow2 flow3 flow4, because now flow 1 is also AF1, also needs the bandwidth. The same routing results show that no inner mechanism is preset in *perd* package to separate the different traffic class.

From the result, we get that, under this condition, the *perd* package cannot separate the traffic by itself, and the *perd* package cannot separate the routing table for each different class.

Test4

From test3, we know that there is no inner separation mechanism in *perd* package, that is, the *perd* package cannot separate the traffic class by itself. Test4 is used to verify that if we separate the bandwidth portion in links for each traffic class, after that, if the *perd* package can separate the routing table. That is to verify if the different routing tables are used to find path for different traffic class after we separate the traffic.

Following is the topology used in test4:



```
Topology:
                         n1
                        /  \
            100M       /    \    100M
                      /      \
                     /        \
                   n0----------n2
                        100M
```

The bandwidth for each link is set as 100M so that every session will be created successful. And this will also eliminate the affect from reservation mechanism used by *perd* package.

We now consider an extreme situation. If we make the traffic separated for EF and AF, then we expect get different routing result from *perd* package. That is, in above topology, if we let link n0-----n2 can only used by AF traffic, and n0-----n1-----n2 can be only used by EF traffic. Then when PERD mechanism is used, we of course should get n0-----n1-----n2 for EF, and n0-----n2 for AF.

Following sessions are created:

Sequence	Traffic class	Source	Destination	Bandwidth	Flow ID
1	AF	n0	n2	2.5M	1
2	EF	n0	n2	1.5M	2

The sequence of creation for AF and EF session is very important, because a *dynamic allocation* for EF and AF is used in test4:

1) Each route node is PERD enabled.
2) Create CBQ links between each node. And give them big enough bandwidth to avoid being affected by the ways used for reservation.
3) Create an AF session, because the bandwidth of link in my test is big enough, so this session will be create successful.
4) By setting CBQ parameters, we let the link bandwidth is wholly occupied by it.
5) Update using provided trigger procedure.
6) Create an EF session to see its route.

Result for test4 under above configuration:

AF session	n0-----n2
EF session	n0-----n2

When we create session as following:

Sequence	Traffic class	Source	Destination	Bandwidth	Flow ID
1	EF	n0	n2	2.5M	1
2	AF	n0	n2	1.5M	2

We get result:

EF session	n0-----n2
AF session	n0-----n2

Result analysis:
In first test configuration, AF session is created first. So it gets path n0-----n2. Then n0-----n2 is wholly used by AF session. Then EF session is created. If *perd* package separate the routing table for different traffic clad as PERD mechanism does, EF should get path n0-----n1-----n2 as expected, because the bandwidth of the link (between n0 and n2) is wholly used by AF.

Based on the same reason, when EF session is created first, it will take path n0-----n2. Then n0-----n2 is wholly used by EF session. When AF session is created, it should take route n0-----n1-----n2.
Now, we see the result is the same. All are n0-----n2. We can get the conclusion that *perd* package does not separate the routing tables according to different traffic classes efficiently.

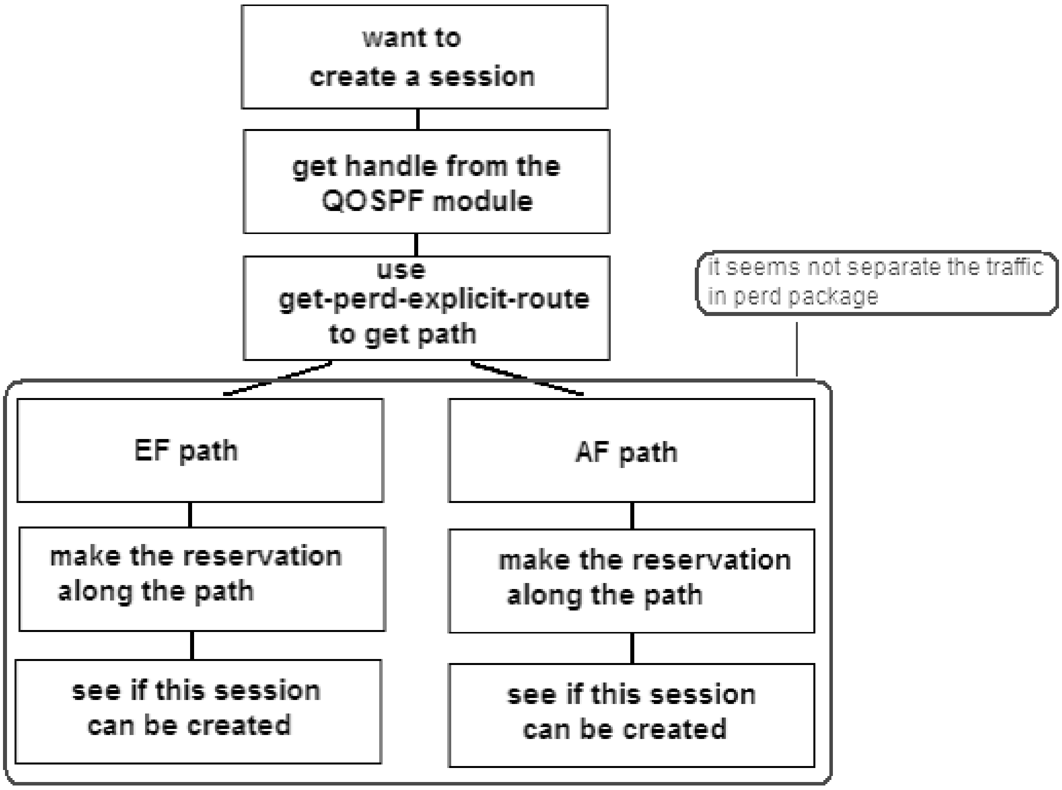
3.5 TEST Conclusions And Analysis

From the above tests for *qosrns* package and results from them, we can see the LS information update works well. But its implementation makes a little confusion and should be improved so as to avoid overlapping or close relationship between routing module and path reservation way. The routing algorithm used by this *qosrn* package does not accord with the RFC 2676.

From the above tests for *perd* package and results from them, we can see the *perd* package cannot separate the routing table for different traffic class, which is the most important idea in PERD mechanism. Also the path selection algorithm it uses is neither shortest-widest nor widest- shortest.

The following shows personal opinion for why *perd* package cannot work well.

perd package works, that is routing for sessions, like this:



perd package still uses routing module “*QOSPF*”. This means in *perd* package, the routing algorithm used must be same as that used in *qosrns* package. From results of the tests made for *qosrns* package, we can understand that *perd* package does not develop its own routing algorithm. So the path selection for both packages is not shortest–widest, and is not widest–shortest, neither.

perd package routing uses following two hooks from its environment

1. *Link instproc get-bw{}*

This procedure is used in *qosrns* package to return the current total available bandwidth on the link. *perd* package does not change it. This makes little sense in PERD, because PERD should know the current available for each traffic class. May be it should bring some parameter to show the available bandwidth for which traffic class, like *get-bw{traffic_class }*

2. *RtModule/QOSPF/perd-bw-changed{}*

This procedure used in *perd* package does not update the database correctly because it does not separate the traffic class. That is, maybe it should bring some parameters to show which database should be updated, e.g. *perd-bw-changed{traffic_class}*

perd package should provide a new way, while not standard ADC to make path reservation. Because standard ADC, just like what *qosrns* package uses, just control the admission for whole available bandwidth. What we need is to separate the admission for different traffic class.

Although *perd* package does not work well as we expected, it provide a train of thought to implement the PERD routing mechanism. It is worth being referenced for our further work.

4. Future work

The routing algorithms should be modified to try its best to fit two constrains optimization. In our context, two constrains are bandwidth and hop count. *perd* package should be made a big change to implementation PERD routing mechanism.

Local optimization and global optimization are to be considered for further study.

Thanks to the work done from different people and from different ideas, we are on the way going forward. What they did, all, are important for our further work.

5. References

[1] E. Crawley, R. Nair, B. Rajagopalan, and H. Sandick, “A Framework for QoS - based Routing in the Internet.” RFC 2386, Category Informational, August 1998

[2] Z. Wang, and J. Crowcroft, “Quality of Service Routing for Supporting Multimedia

Applications.” IEEE Journal Selected Areas in Communications, 14(7): 1228-1234, 1996

[3]R.Guerin,et al. QoS routing mechanism and OSPF extensions, RFC2676, Aug 1999

[4] S. Chen and K Nahrstedt. An overview of quality-of-service routing for the next generation high-speed networks: Problems and solutions. IEEE Network Magazine Special Issue on Transmission and Distribution of Digital Video, 1998

[5] R. Guerin, A. Orda, and D. Williams. QoS Routing Mechanisms and OSPF Extensions. *IETF Internet Draft <draftguerinqosroutingospf00.txt>*, November 1996.

[6] Q. Ma and P. Steenkiste. On path selection for traffic with bandwidth guarantees, In Proceedings of IEEE International Conference on Network Protocols, Atlanta, GA, October 1997.

[7] Q. Ma, P. Steenkiste, and H. Zhang. Routing High Bandwidth Traffic in Max-Min Fair Share Networks. In *ACM SIGCOMM96*, pages 206–217, Stanford, CA, August 1996.

[8] Z. Wang and J. Crowcroft. Quality of Service Routing for Supporting Multimedia Applications. *IEEE JSAC*, 14(7): 1288–1234, September 1996.

[9] F. Faucheur,et al. Extensions to OSPF for support of Diff-Serv-Aware MPLS traffic engineering. IETF draft. Feb. 2001

[10] T. Korkmaz, et al. An Efficient Algorithm for Finding a Path Subject to Two Additive Constraints, Measurement and Modeling of Computer Systems, 318-327, 2000

Appendix

qosrtest1.tcl
qosrtest2.tcl
qosrtest3.tcl
qosrtest4.tcl
perdtest1.tcl
perdtest2.tcl

perdtest3.tcl
perdtest4.tcl