Multicommodity Flow Applied to the Utility Model: A Heuristic Approach to Service Level Agreements in Packet Networks

by

Louis Lei Yu

B.Sc. Queen's University, 2003

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Supervisor: Dr. Eric G. Manning, P.Eng., ISP

Abstract

Consider the concept of the Utility Model [5]: the optimal allocation of resources of a server or network while meeting the absolute Quality of Service (QoS) requirements of users' multimedia sessions. Past algorithms and heuristics to solve the Utility Model mapped the problem onto a variant of the Combinatorial Knapsack Problem, with server utility (e.g. revenue) as the quantity to be optimized and with user QoS requirements expressed as constraints on the resource allocation. Both optimal (algorithmic) and fast but sub-optimal (heuristic) methods were derived to solve the resulting Multidimensional Multiconstraint Knapsack Problem (MMKP) and hence to perform admission control of proposed user sessions

However, previous algorithms and heuristics were restricted to solving the Utility Model on an *enterprise network* (a network of less than 30 nodes), owing to the need in admission control to solve the problem in real time, typically a few seconds or less. The methods used for the path finding and admission processes had unfavorable computational complexities. As a result, only small (i.e. enterprise) networks could be treated in real time. Also, considerable time was wasted on frequently unnecessary traversals during upgrading.

In this thesis we attempt to solve and implement the Utility Model using a modified version of a Multicommodity Flow algorithm, which has better computational complexity than Knapsack Algorithms or many heuristics and hence is capable of finding paths relatively quickly for larger networks. What's more, the Multicommodity flow algorithm used keeps essential information about the current networks and user sessions, thus further reducing the overall admission time.

Table of Contents

List of Figures

1 Introduction

In this Chapter we describe the problem to be solved and mention a few key related ideas.

1.1 The Problem

1.1.1 Multimedia Traffic

Over the last ten years, there has been a dramatic increase in the amount of traffic, specifically multimedia traffic, over the Internet, and this growth is expected to continue. As computing and telecommunication technology improves, more and more advanced techniques and protocols regarding network traffic and information routing are designed. Computers are getting faster and cheaper, while networks are being provided with higher bandwidths and lower error rates at lower costs. Now, people might like to watch, listen, and enjoy all kinds of multimedia applications, including interactive videos and high-quality videoconferencing, delivered over the Internet. Naturally, the next step is to develop applications to carry multimedia traffic efficiently over networks.

However, the proposed carriage of multimedia traffic implies specific performance standards or Quality of Service (QoS) requirements, high bandwidth and low latency being chief among them. To route multimedia traffic such as interactive video over the Internet, one requires a single path with high bandwidth and low latency for each multimedia session.

1.1.2 Quality of Service (QoS)

In addition to these challenges, an Internet Service Provider or ISP[1] must deal with rapidly changing patterns of usage within a network. The ISP must be able to react to these quickly, in order to minimize costs while at the same time being able to honor quality of service guarantees made to the customers. Hence some sort of adaptive approach should be taken to react to the changes of patterns of traffic flow within a network.

The ISP must be able to avoid overbooking of its resources, which may break quality of service (QoS) guarantees [20], but it must not purchase excessive amounts of these resources from the underlying facilities – the carriers who own them, or it may go bankrupt in today's highly competitive marketplace [4].

1.1.3 The Problem

Our problem is to select, from a set of proposed Service Level Agreements (SLAs) submitted by customers requesting admission to a packet-based network, a subset, one request per customers which will result in maximum utility (revenue) for the (single, monopoly) network operator, while fully respecting the allocations of resources necessary to honor absolute or hard QoS guarantees, made to both the newly admitted SLAs and to all other currently active SLAs.

Service Level Agreements (SLAs) [20] in this case, defined as network performance requirements by users for point-to-point communication, are needed to convey the expectation of users on the network providers. SLAs include the definition of several possible QoS levels, each of which defines an amount of bandwidth required, a maximum delay bound, duration, and an offered price (a bid). The requirements for bandwidth and latency are absolute: the provision of less bandwidth or more latency than requested is unacceptable. In an SLA-based system, an Admission Controller (such as the Utility Model) considers incoming SLA requests and assigns available

---

[1] By ISP we mean a network Service Provider (such as UUNet) as opposed to a local ISP.

network resources in order to provide QoS guarantees while maintaining efficient resource usage

1.2 Related Concepts

1.2.1 Network Configuration

Traditionally, ISPs would reconfigure their networks about once or twice a year. In addition the SLA – the service level agreement between an ISP and its customer - has been inflexible.

Now, however, ISPs need to reconfigure daily or even hourly, in order to meet the demanding QoS requirements of customers while minimizing their costs. In addition, customers requesting multimedia services such as video conferencing or streaming video would like more flexible SLAs, with features such as the ability to

- Make bids & adjust them if necessary (auctions – a form of resource allocation much favoured by economists for its efficiency).

- Ask for several levels of QoS at several prices and optionally be able to reserve resources for a defined time period.

- Be guaranteed at least one QoS level at a specified price, with the option to attempt to upgrade the QoS level later.

In some cases, a customer may wish to book a reservation for traffic several days (or weeks) in advance, for single or periodic events. For example, a company that holds a videoconference every Monday morning might like to schedule bandwidth for this purpose. In such cases we have ample time to do admission control and can therefore undertake lengthy computations in support of it. Unfortunately, not all of the bandwidth demands of ISP customers are scheduled in advance like this. More

typically, a customer may present a proposed SLA in order to watch a movie or buy a car, and the customer will expect the movie or infomercial to begin within a few seconds, hence the real-time constraints on the proposed algorithms and heuristics to do network admission control.

2 Previous Work

In this Chapter we review previous work related to our problem. We will also compare these studies, and reach conclusions by summarizing a list of things from previous methods needing improvement, as well as a list of advantages of these methods.

2.1 The Utility Model

The original purpose of the Utility Model [5] was to manage the consumption of resources in a multimedia server, in order to maximize the revenue generated by the server (or more generally, its economic *utility*), while keeping the quality of the services (QoS) provided to clients at guaranteed absolute levels. The Utility Model can manage resources such as CPU time, graphics coprocessor time, memory and video bus bandwidths, and primary and secondary memory. The Utility Model was later adapted for use in a network (as demonstrated by Watson [9]); in this case the resources are the bandwidths of the network links.

The Utility Model can be used to manage resources while handling the bidding process of selecting user SLAs for admission. In a multimedia environment with paying customers and legally enforceable SLAs, we must observe QoS requirements, and hence we must guarantee the availability of the necessary resources to each customer. Hence, given finite network resources, we need some notion of *admission control* [5].

A *session* [5] is an entity which requires the resources, at the level of granularity at which the Utility Model operates. Each session has a number of possible QoS levels. We use the generic terms: *Bronze, Silver, and Gold.*

A Service Level Agreement [20], or *SLA*, is a contract between an Internet Service Provider or ISP, and a customer. It is simply an agreement to provide a certain level of service, for a certain time, and at a given price. Customers submit proposed SLAs to network admission controllers to gain admission.

The carrier, on the other hand, wants to do *admission control* [5], i.e. to admit a subset of the SLAs, each at the QoS level which maximizes revenue, while fully respecting all terms and conditions (QOS guarantees)[2] of both the newly-accepted and all previously-accepted SLAs

Each QoS level for each session implies or specifies a set of resource requirements needed for that QoS level, and a value, or *utility*, assigned to each such set. The utility can be viewed as the amount of money that the session's owner – the customer - is willing to pay, to contribute to the ISP's revenue, if it is granted admission at that QoS level. QoS level *0* means no QoS (and hence no resource used); and thus means that the session is rejected. Level *0* is the lowest QoS level and each user SLA is guaranteed admission at this level -- or hopefully higher!

For each session i, a QoS level Q_i must be selected. This level then implies the *session utility* $u(Q_i)$, and the *session resource usage* $r(Q_i)$. In order to guarantee service at the level of quality Q_i selected, we must *bind* the necessary resources to the session before it begins, and for the duration of its existence.

Our goal, then, is to maximize the revenue U while respecting the system resource constraints R.

---

[2] In fact, given finite carrier resources and potentially unbounded service requests, the carrier *must* do admission control, if any credible attempt to guarantee QoS is to be made.

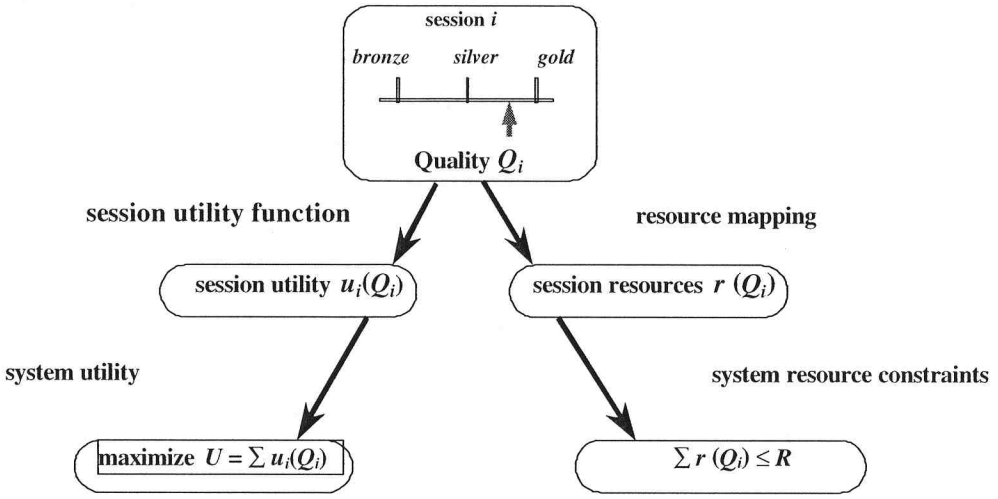


Figure 1: Relations Among Qualities, Utility and Resources

In Figure 1, each cusstomer session, *i,* specifies one or more levels, Gold, Silver or Bronze. Each level is then mapped to a session utility, which in a network, corresponds to the prices offered by customers; and a set of session resources, which in a network, correspond to bandwidth and latency, perhaps amongst other things. The goal then is to decide which level from each session to select so as to maximize the total utility (money) gained without exceeding the resource constraints (bandwidth and latency).

2.2 Applying the Utility Model to the Packet Network

We have noted that the Utility Model maps neatly to a simple model of a packet-switched network [10]. In this case, the relevant resources are the bandwidths and latencies of the links, and possibly the switches, of a packet network.

Now, our problem is to decide whether a new or changed traffic flow (that is, the traffic flow that will be associated with a Service Level Agreement, from source to destination) should be admitted, and if so, which path from source to destination it should use.

The problem may readily be split into two distinct sub-problems:

1. The admission of a new SLA into the network.

2. The modification, upwards or downwards, of the QoS level provided to an SLA which has been previously admitted to the network.

These sub-problems are very similar, and we approach them in much the same way.

In order to apply the Utility Model to our situation, we must map the SLAs' requirements to the resource requirements needed for the UM. We express the SLAs' requirements as levels of Quality of Service (QoS), mapping each into a set of resource requirements for the network.

For our simple network model, we restrict these requirements to be the bandwidth and latency of the connection. For each level, we also specify a value of utility, to express how much the SLA's owner - the customer - is willing to pay.

Latency as a resource is unlike bandwidth [9]: While a connection with a certain amount of latency may be required, it is not *consumed* in the same way as bandwidth. Thus latency resembles a constraint more than a consumable resource. In other words, total link bandwidth is conserved; total link latency is not.

Path latency l_p is defined here as

$$l_p = \sum l_i$$

Which is the sum of link latencies over the links l_i of the path.

Overall, a path meets a certain QoS level Q_i iff

1. The path's latency l_p is less than or equal to the QoS level's required latency l_q.

2. Each link in the path has uncommitted bandwidth greater than or equal to the QoS level's required bandwidth.

For example, consider an SLA needing a 500MBps connection, with latency to be no greater than 1.7 milliseconds, for which the customer is willing to pay $40 per hour. However, the customer will also accept other, lower levels of QoS, with lower utilities (offered prices). We can express this as an SLA for the Utility Model, as in Figure 2.

	Level 1 (Bronze)	**Level 2 (Silver)**	**Level 3 (Gold)**
bandwidth (Mb/s)	250	300	500
End-to-end latency (s)	2.5	2.0	1.7
utility value ($)	20	30	40

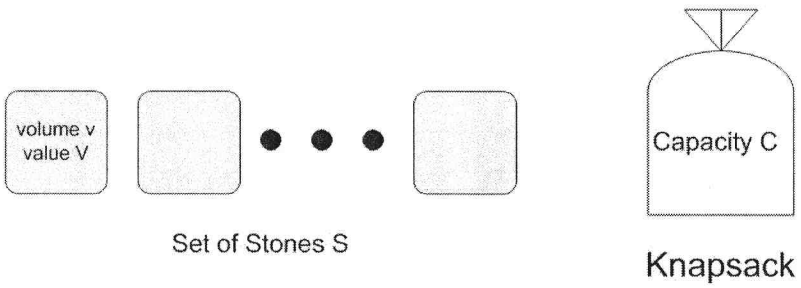
Figure 2: Example Session Profile

2.3 Knapsack Problem

The Utility Model can be solved by mapping it to a Multi-choice Multi-dimension 0-1 Knapsack Problem (MMKP) [5], which is a variation of the 0-1 Knapsack Problem [12]. A simple *0-1* Knapsack problem can be visualized as shown in Figure 2: Given a knapsack with capacity C, and a set of stones S, each stone S_j with volume $vol(S_j)$ and value $VAL(S_j)$, choose a subset of stones S' from S such that

$$vol(S') < C \text{ and } MAX(VAL(S'))$$

Figure 3 is a graphic illustration of the basic 0-1 Knapsack Problem.



Pick a subset S', in order to maximize V(S'),
subject to resource constraints: v(S')≤C

Figure 3: 0-1 Knapsack Problem

That is, we pick a most valuable set of stones that will fit in the knapsack. On one hand, the set of stones should be most valuable; on the other hand, the set of stones has to be smaller in total volume than the volume capacity of the knapsack.

Based on the concept of the 0-1 Knapsack Problem [12]; we define a Multi-choice Multi-dimension 0-1 Knapsack Problem (MMKP) [5], where we have multiple sets of stones.

$$S_1, S_2, S_3, \ldots S_n.$$

and each set of stones has multi-dimensional volume

$$v_1(S), v_2(S), \ldots\ldots v_n(S)$$

In this case, we will let each pile of stones represent a session S_n. and each stone represents that session at a particular level of QoS (bronze, silver and gold stones, as it were). The weight of each stone represents the utility offered for the session at that QoS level. We then wish to select one QoS level S'_m from each session (one stone from each pile of stones), such that we maximize the weight (revenue or utility).

$$MAX \sum V(S'_m)$$

while ensuring that none of the resources are overused – i.e. That the total resource consumption is less than the total resource constraint Rn.

$$\sum v_n(S'_m) < R_n$$

Figure 4 illustrates the Multi-choice Multi-dimension 0-1 Knapsack Problem (MMKP).

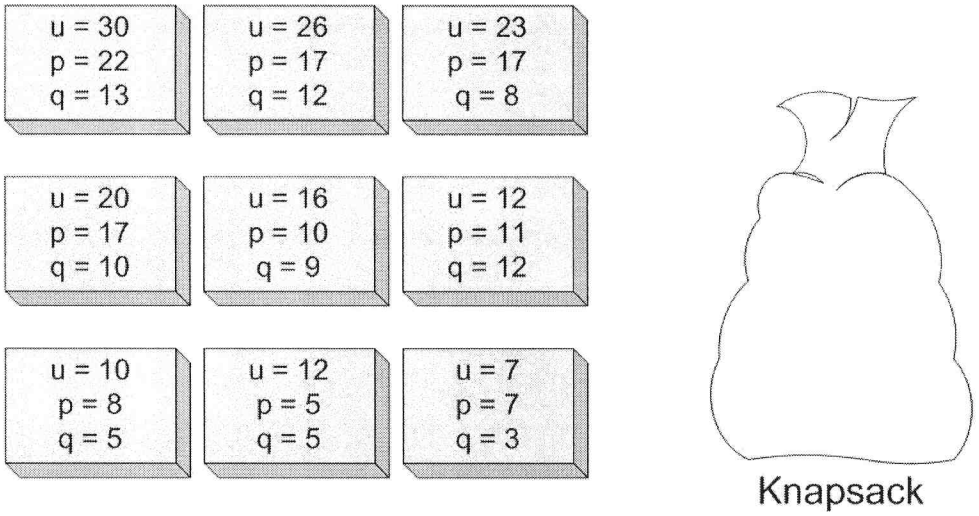


Figure 4: Multi-choice Multi- dimensional 0-1 Knapsack Problem (MMKP)

2.4 Solving the MMKP by HEU

After mapping the Utility Model into a Multi-choice Multi-dimension 0-1 Knapsack Problem (MMKP), we look for ways of solving the MMKP. Khan [5] presented two solutions to this problem: the branch-and-bound algorithm BBLP using linear programming for bound calculation, which achieves the optimal solution, and the heuristic HEU whose result often comes reasonably close to the optimal solution, but generally takes much less time. We will later compare our running time and solution to those produced by Khan's heuristic (HEU) [5]; therefore we now briefly describe it.

2.4.1 Admission Process For HEU

The admission process for HEU decides which levels of each SLA to admit. The process is as follows: we assume an empty system, i.e. no previously- admitted

sessions. HEU first orders the QoS levels for each candidate session according to revenue $V(S_m)$. It then selects the **least profitable** [10] (smallest value of V) of the feasible QoS levels for each session as its initial solution. Note that this includes denial of service (represented by a dummy level *0* of QoS which consumes zero resources and offers zero utility) if no nonvoid level is feasible.

HEU then considers possible *upgrades* [5] for each session. An *upgrade* is a move to a higher level of QoS, and hence greater utility earned, for a session. HEU calculates the net change in *aggregate resource consumption*[3] for each session's feasible upgrades. The calculation penalizes those upgrades that consume heavily used resources. If any upgrades are found to decrease the system aggregate resource consumption[4], the upgrade that yields the greatest decrease in aggregate resource consumption is selected. Otherwise, the upgrade that provides the greatest value gained per extra unit of aggregate resource is selected.

The loop returns to considering possible upgrades for each session. The heuristic terminates when no more *feasible* upgrades can be made. HEU only considers upgrades for each customer's SLA, and not downgrades[5]. The steps of Khan's HEU [5] are described in the pseudo-code of Figure 5.

---

[3] See Section 2.4.3 for the definition of Aggregate Resource Consumption

[4] While this may appear counter-intuitive, we can imagine that if the resources used by the current QoS level are in heavy use, and there exists an upgrade whose resources are in light use, then this upgrade may decrease the system aggregate resource consumption.

[5] Downgrading is here defined as a move to a lower level of QoS for a session.

```
for (i = 1, ..., n) ρ[i] = 1;                    /* initialize with smallest QoS levels */
C = Σ r[i][ρ[i]];                                /* calculate resource usage C */
while (1) {
        Δr_max = 0, Δp_max = 0;
        for (i = 1, ..., n; j = ρ[i] + 1, ..., l_I) {
                if (∃k : k = 1, ..., m, C[k] - r[i][ρ[i]][k] + r[i][j][k] > R[k]) continue;
                Δr = ((r[i][ρ[i]] + r[i][j]) • C) / C;      /* calculate aggregate resource */
                if (Δr > Δr_max)                            /* upgrade with max resource saving */
                        Δr_max = Δr, i' = i, j' = j;
                if (Δr_max ≤ 0) {
                        Δp = (v[i][ρ[i]] - v[i][j]) / Δr
                        if (Δp > Δp_max)                    /* upgrade with max value /aggr. resource */
                                Δp_max = Δp, i' = i, j' = j;
                }
        }
        if (Δr_max ≤ 0 and Δp_max ≤ 0) return ρ;
        C = C – r[i'][ρ[i']] + r[i'][j'];                   /* update C and return */
        ρ[i'] = j';
        return
}
```

Figure 5: Pseudocode for HEU

2.4.2 Path Finding and Matching Process for HEU

Before we apply the heuristic and attempt to upgrade a customer's service, we need to find paths in the network that have enough bandwidth to permit possible upgrades. There generally is more than one path from source to destination in a practical network (graph connectivity > 1). Because of the delay constraint, some of the paths will be considered infeasible to carry the multimedia traffic of an SLA. A K shortest paths discovery algorithm [13] is applied to the network. (There has been considerable work on K shortest paths problems; Khan selected a path-deviation style algorithm [5]. This algorithm has running time of $O(kN^2)$ for networks of 30 or fewer nodes, and the running time becomes quadratic for networks with more than 30

nodes. Hence HEU is only practical for *Enterprise Networks*, by which we mean networks of 30 or fewer nodes.)

We now have a list of candidate paths from source to destination for each SLA. For each SLA upgrade, the list of candidate paths is processed once, selecting the best choice to route the upgraded traffic[6]. That is, we select a path whose maximum link utilization is the smallest amongst the list of candidate paths[7], and we have to check if the path has enough bandwidth to accommodate the upgraded SLA. Because the utilizations of all the links in the network are constantly changing with each possible upgrade, the heuristic does not keep a profile of the utilizations of each path, and chooses to go through each path afresh, looking for maximum link utilization.

2.4.3 Aggregate Resource Consumption Calculation

Here we briefly define aggregate resource consumption [10]. Toyoda introduced the idea of a *penalty vector* [8], in order to obtain a one-dimensional resource consumption index based on the consumption of multiple limited resources. As an example, suppose we have two limited resources r_1 and r_2, and that the current consumption of these resources is

$$C = (c_1, c_2) = (0.5, 0.2)$$

Given two sessions S_1 and S_2 with required resource vectors (r_1, r_2)

$$R_{S1} = (0.1, 0.4)$$

---

[6] Given the assumption that there exists a feasible path to route the upgraded traffic

[7] Candidate paths in this case are all feasible paths, that is, a path such that the most heavily utilized edge can still accommodate the upgraded bandwidth demand. If there is only one feasible path, we just route the upgraded SLA upon that path

$\boldsymbol{R_{S2}}$ = **(0.4, 0.1)**

We prefer S_1 to S_2, as it requires less of r_1, which is currently in heavier use than r_2. $\boldsymbol{C}$ is used as a *penalty vector*, and we calculate the penalties as:

S_1: $\quad \boldsymbol{R_{S1}} \bullet \boldsymbol{C} = 0.1 \bullet 0.5 + 0.4 \bullet 0.2 = 1.3$

S_2: $\quad \boldsymbol{R_{S2}} \bullet \boldsymbol{C} = 0.4 \bullet 0.5 + 0.1 \bullet 0.2 = 2.2$

Geometrically, the effective gradient is the projection of the required resource vector onto the current consumption vector C. Put another way, $R \bullet C$ measures S's demands on resources weighted by their scarcity.

We can then scale the effective gradient by the magnitude of $\boldsymbol{C}$ (0.54) to obtain aggregate resource consumptions for S_1 and S_2 as 2.4 and 4.1 respectively. In HEU, we are more interested in the *change* in aggregate resource consumption when a session is upgraded.

2.5 Solving the Utility Model Using Watson's Approach

To adapt the Utility Model to a packet switching network, Watson [9] introduced another, faster but less efficient, heuristic for solving the Utility Model. Watson's heuristic does not directly map the Utility Model onto a Knapsack Problem; instead it combines the admission, path finding, routing, and upgrading processes, and attempts to solve the Utility Model in real time during the routing process. We now briefly describe Watson's heuristic.

2.5.1 The Admission Process for Watson's Heuristic

Suppose we have a network N and a proposed SLA has been submitted requesting service from (node) S to (node) D. Suppose that $SLA(S,D)$ is currently inactive. We need to find the following information:

- A routing for $SLA(S,D)$, and its QoS level Q_n
- The routing for any flows that may have been changed while routing (S,D).
- Any new link capacities

These, taken together, comprise the new state of the network N after we admit the SLA. In other words, we decide which path to use to route $SLA(S,D)$ and update the network status, before we route $SLA(S,D)$ on its path. We then update the status of network N as we admit the SLA to the network at the starting level.

2.5.2 The Upgrading Process for Watson's Heuristic

Similar to Kahn's path finding process for HEU [5], in order to activate a session *(S, D)* requested by an SLA, a path discovery algorithm (Dijsktra's K-shortest path [13]) is first used for each SLA to find possible routings for *(S, D).* These paths are then sorted by aggregate resource usage.

We then attempt to find a feasible path available to *(S, D)* for initial admission. If such a path exists, (S,D) is routed along it, and the initial admission process is finished. The goal of the initial admission process is to find a feasible path from S to D as a starting point: we can attempt to find upgrades to gain more utility later.

After the initial admission process, each of the SLAs SLA_i should have a feasible path from source Si to destination Di. The paths serve as a starting point for each SLA;

the task of finding a subset of QoS levels (one from each SLA) that optimizes overall utility is left for the upgrading step.

In the event that $SLA(S, D)$ wishes to increase its bandwidth by some amount, we need to find the following

- The new routing for $SLA(S, D)$.
- The new routing for any other flows that may have been changed.
- Any new link capacities.
- The new state of the network N.

Here is a more detailed description of the steps:

We first check whether the current routing for *(S, D)* has enough *surplus capacity*[8] to accommodate the increased flow. If this is the case, then we do not need to change any routing. We simply increase the bandwidth allocation of flow *(S, D)* and decrease the free bandwidth of all links along the path to reflect the increased allocation to (S, D).

If the current route for *(S, D)* does not have enough free bandwidth, we next attempt to locate some route for *(S, D)* that does have sufficient free bandwidth, while changing nothing else. This involves first finding the shortest possible paths with sufficient bandwidth and acceptable latency (or other properties implied by the definition adopted of QoS), and second, choosing the best of these paths.

---

[8] That is, if the most utilized edge amongst all the edges in the path still has enough surplus bandwidth to accommodate the upgraded demand for bandwidth.

From the above description one can see that the chief foreseeable disadvantage is that this approach will not generally give an optimal solution. This is because the selection of possible paths must be heuristic; the outcome of an upgrade can only depend on the choices of nearby paths and the solution for the shortest paths algorithm.

As mentioned before, to discover acceptable paths for this flow, we apply a *k-shortest path* algorithm [13] to source S and destination D of *N*. If the network is small (e.g., an enterprise network) we may be able to allow the algorithm to run to completion. If the network is large, we may be forced to reduce the number of paths considered; however, in that case, the number of choices of nearby paths will be reduced, and so the heuristic' s performance will fall further short of optimality, due to the limited path choices.

2.6 Comparison Between Heuristics

Both Khan's [5] and Watson's heuristics [9] can solve the Utility Model efficiently. In specific cases, depending on the shape, the size, and the connectivity of the network, both M-HEU [5] and Watson's [9] can solve the Utility Model in relatively short running time. We will now go through different phases of both and compare the two methods, outlining the advantages and disadvantages of both methods.

2.6.1 Path Finding Phase

Both Watson's heuristic and Khan's HEU start by finding a list of paths for each SLA (from source to destination). Both heuristics uses Dijkstra's K-shortest path [13] to find such lists of paths. The Dijkstra path algorithm's running time is approximately $O(kN^2)$ for a smaller network with low connectivity[9] [13]. For a network of more than 30 nodes, and with high connectivity, the Dijkstra's algorithm's running time

[9] A small network with low connectivity (or enterprise network) is here defined as a network of less than 30 nodes, each node with less than node degree [connections to other nodes] .

can increase exponentially [13]. In such a complex network, the path finding process will not be sufficient to produce results in a reasonable running time.

We can shorten the running time of the Dijkstra's K-shortest path algorithm by altering it to produce just some of the paths discovered, as Watson's heuristic does [9]. For example, the heuristic could select a random subset of the path choices from S to D. However, Watson' s heuristic will produce poorer results when only some of the paths are available for upgrading.

2.6.2 Admission Process

After the path finding process, the paths are mapped to each SLA. For Khan's heuristic [5], the list of paths discovered is processed each time an upgrade occurs for each SLA, and the best path is found for each upgrading step. For Watson's heuristic [9], only some of the paths discovered are mapped to an SLA for the upgrading process. In other words, for Watson's heuristic, we only will consider nearby paths for upgrades, therefore we may get excellent, acceptable or really poor path choices for upgrading depending on the initial routing.

From the above comparison, we can see that Khan's heuristic [5] considers all the path choices for each upgrade, and is the more efficient admission process of the two. If there exists a path solution for an upgrade, Khan's heuristic [5] will be more likely to find it. However, Khan's admission process is relatively slow, in part as every path is considered and updated for each upgrade step. What's more, various types of upgrading are tried before the heuristic terminates. This may be impractical if there are many other SLAs - in practice, perhaps hundreds - lined up waiting for admission.

Also, Khan's heuristic[10] [5] does not keep records of the edge utilization and flow information for each path discovered. Each time an upgrade occurs, the list of paths

---

[10] In Khan's implementation, only the maximum edge utilization of each path is stored.

discovered has to be checked, finding maximum link utilization and total bandwidth surpluses. If records are kept about link utilization of the path discovered, less time will be taken going through each path finding the best solution.

Watson's heuristic [5] does not consider every path choice from *S* to *D* (for a SLA) each time an upgrade occurs. Instead, it only considers nearby paths. We define a *nearby path* as one that has nodes in common with *(S, D)* – more specifically, in order:

1. Those paths that have start-point S and endpoint D, or vice-versa.

2. Those paths that have S or D as an endpoint or start-point.

3. Those paths that contain both S and D.

Because it does not consider every possible path choice at upgrade-time, the Watson heuristic may be inefficient with resources, but it can find a close-to-optimal solution much faster than Khan's. However, if there are only a few solution paths for an SLA upgrade, the Watson heuristic is not guaranteed to find one, nor is the heuristic guaranteed to find the best solution for an upgrade if there are several possible upgrades. Watson's heuristic concentrates on faster running time to solve the Utility Model, rather than on efficiency – an optimal or close to optimal solution.

2.7 Summary of Previous Work

From examining both the Khan and Watson heuristics, we conclude that there is room for improvement in running time. First, both Watson's [9] and Khan's heuristics [5] use the Dijkstra K-shortest paths algorithm [13] for the path finding process. In an

enterprise network [11], this is efficient; meaning an $O(kN^2)$ procedure such as Watson's or Khan's heuristic will produce efficient results in fast enough time in certain cases in a small network.

However, it is important that we look for scalable solutions, so we can treat networks larger than 30 nodes. This means that we need theoretically fast path finding algorithms that produce sufficient small running times, not just for enterprise networks with low connectivity, but for networks of any size, with any connectivity; This suggests an algorithm with theoretically smaller running time suitable for any size networks to replace the Dijkstra K-shortest path algorithm.

Secondly, both Watson's heuristic and Khan's HEU handle the SLA upgrading process by going through a list of paths [5, 9]. With Watson's method, only some of the paths (the nearby paths) are processed, and in both cases, a list of paths has to be checked.

If we can keep a list of vectors of information about maximum link utilization for each path, we can keep updating the list as we upgrade each SLA, and we won't have to go through unnecessary traversals. This is similar to matrix manipulations [22] done for graph traversals in linear programming; but instead of keeping rows and columns, we keep a list of vectors. This will be described in more detail in the next Chapter.

Finally, to solve the Utility Model, most of the overall running time is spent on the process of upgrading. This especially holds true for Khan's heuristic [5], as many different methods of upgrading are tried before the heuristic finally terminates. By

---

[11] Enterprise network is here defined as a network of less than 30 nodes, each node with less than 4 connections to other nodes.

improving the upgrading step of the heuristic, we will achieve a much smaller overall running time.

3 Solving the Utility Model Using a Multicommodity Flow Algorithm

In this Chapter, we discuss a new heuristic for solving the Utility Model by formulating parts of the Utility Model as a Multicommodity Flow problem [6].

3.1 The Problem

The Utility Model maps neatly to a simple model of a packet-switched network [10]. In this case, the relevant resources are the bandwidths and latencies of the links. The problem of solving the Utility Model may readily be split into two tasks

1. The path finding process for each customer's SLA. A path is found for the initial routing and new paths are found if the same path can't be used for the upgrading process.

2. The Admission Process, which decides which level of each SLA to admit. Together, all the picked (SLA, QoS level) pair must maximize the revenue generated with respect to the resource constraints.

In this Chapter, we will show that we can map both the path-finding step and the admission step onto a modified Multicommodity Flow problem [6]. Moreover, by solving the modified Multicommodity Flow problem, we can obtain a fast and efficient heuristic to solve the Utility Model. Unlike all of the previously-discovered heuristics, our heuristic scales well; thus making it feasible to be used in larger-than-Enterprise networks.

First we will explain the concept of the Multicommodity Flow problem. Next we will show how we can transform the Multicommodity Flow solution [6] into an Unsplittable Flow solution [11], thus giving us a partial solution to the Utility Model. Lastly, we will show how the Unsplittable Flow solution can be easily changed into a solution to the Utility Model.

3.2 Multicommodity Flow Problem

The Multicommodity Flow problem [6] involves simultaneously shipping multiple commodities' demands through a single network. Each commodity sends out some *flow* – a fraction of the demand (up to 100% of the demand), traveling through different paths (one or more), which can add up to the amount of *demand* each commodity can send from a source to a destination.

In a network with multiple nodes and edges, each edge is limited by a *flow capacity*. The *total flow* summed over all commodities for each edge[12] must not exceed the *flow capacity* of that edge. On top of that constraint, we must maximize the demand[13] each commodity can send out.

In another word, each commodity can send out *one demand* through the network, say 4 units of demand, that demand can be split into many flows traveling though different paths (a vector of flows). The objective is to maximize the sum of *demands* (sum of all the columns in that vector) summed over the commodities.

Raghavan and Thompson [2] documented the first solution for the general Multicommodity Flow problem in 1988. Since then, many researchers have given

---

[12] That is, the total flow of an edge is the sum of all the flows all commodities sent through that edge. For example, if commodity 1 sends out 4 units of flow traveling through edge 1, and commodity 3 sends out 5 units of flow traveling through edge 1; the total flow of edge 1 is 9 units.

[13] Here, maximize the demand means if there's a way of shipping a demand through the network, the algorithm must find it, else the algorithm should maximize the percentage of the demand being sent through the network

solutions for different variations of the Multicommodity Flow problem; the running times for the algorithms are also getting faster. So far, the best solution for the general Multicommodity Flow problem requires linear running time [14]. The solution however, also requires the maintenance of many large matrices, which is infeasible in a network application. The solution I decided to use is by Leighton et al [15]; it is the fastest combinatorial solution for Multicommodity Flow. This algorithm has a faster theoretical running time than some of the best linear programming algorithms: $O(klogn)$ where k is the number of commodities and n is the number of nodes in the graph. It is also easy to adapt this algorithm to the implementation of the Utility Model, as we will illustrate later.

3.2.1 The Underlying Theory

In this Section, we define a variation of the general Multicommodity Flow problem called *the Concurrent Flow problem* [6]. The Concurrent Flow problem differs from the general Multicommodity Flow problem in that there is an amount of *demand* for each commodity, and the object is to ship all of the demand across the network. So in this case we optimize the sum of flows with each flow bounded by the associated demand.

Formally, consider an undirected graph $G(V,E)$ with a positive capacity $u(v,w)$ for each edge $v,w \in E$. Consider also a set of commodities numbered *1* through *k*, where each commodity *i* is specified by a source-sink pair $s_i, t_i \in V$ and a positive demand di from its source s_i to its sink t_i . For each commodity *i*, we ship an amount proportional[14] to its demand d_i from its source s_i to its sink t_i . *A Concurrent Flow f* consists of *k* single commodity flows, one for each commodity.

A concurrent flow achieves *demand satisfaction* [6] if it ships an amount of each commodity equal to its demand from its source to its sink. It obeys the capacity

constraints [6] if no flow $f(v,w)$ on an edge $(v,w) \in E$ exceeds the capacity $u(v,w)$ of the edge. A *feasible concurrent flow* [6] achieves demand satisfaction while obeying the capacity constraints. Our objective is to find a feasible concurrent flow. If however, we do not find such a flow, we should ship at lease a percentage of the demand d_i across the network (that, we will see later, can be mapped to a downgrade in the utility model).

Figure 6 illustrates the concurrent flow problem. Suppose, in this network, we need to ship commodity 1, with demand $d_1 = 3$ across the network from node 1 to node 5 (source node for commodity1 $= s_1 = 1$ and destination node for commodity 1 = $t_1 = 5$). We have to find a path that we can use to send the demand of 3 units from source to destination.

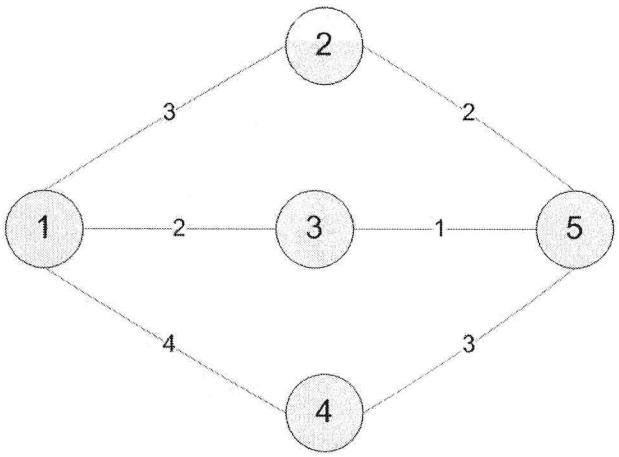


Figure 6: Example of a Concurrent Flow Problem

At this point, the definition of the Concurrent Flow problem does not limit ways that we can ship the 3 units of demand. That is, it is ok to split up the 3 units of demand

---

[14] Proportional here defined as a percentage, if not all

and ship parts of it through separate paths, as long as it achieves demand satisfaction while obeying the capacity constraint of each edge [15]. Hence a feasible solution is

- Ship 1 unit of bandwidth through path 1 3 5
- Ship 1 unit of bandwidth through path 1 4 5
- Ship 1 unit of bandwidth through path 1 2 5

While the following sample solution might not be the optimal solution to the example Concurrent Flow problem; it shows how the demand for each commodity can be *split* and routed through different paths (this is different than the requirement for the Utility Model implementation, as in multimedia routing we prefer to dedicate one single path to ship all the demand - in order to avoid jitter and synchronization problems. However, later we will show that by modifying the algorithm to solve the Concurrent Flow problem, we can change the algorithm to map splittable flows into Unsplittable Flows, and thus fit the requirements of the Utility Model).

3.2.2 The Concurrent Flow Algorithm

We now summarize the approximate algorithm of Leighton et al [15] for approximately solving the Concurrent Flow problem. Given an error parameter $\varepsilon > 0$,

---

[15] The usual concurrent flow problem allows demand from each commodity to be split up and shipped through several paths. However, there is a variation of the concurrent flow problem called the unsplittable flow problem, in which the demand can only be shipped through one path; more on that in Section 3.3.

the algorithm finds an ε-optimal flow [16] [15], Because we can make the error parameter ε arbitrarily small, we can find a solution arbitrarily close to optimal.

The rough concept of the algorithm is that for each commodity i, the algorithm would iteratively send a flow from source to destination. *A flow* [15] is defined here as a percentage of the demand d_i , routed through a path found from source to destination. For a commodity, each time we try to send a flow, we first discover a path from the commodity's source to destination; then we decide how what percentage (up to 100%) of the commodity's demand we wish to send through that path.

The algorithm begins with a flow *f* that achieves demand satisfaction but ignores the capacity constraints. In other words, the initial path finding process tries to route all of demand *di* on the initial path. If we can fit all of the demand into the path, we stop right there. If not, Leighton et al. [15] shows that if the flow is not *ε-optimal* [17], then there exists at least one *poorly routed commodity*; i.e. at least one commodity which is not routed onto an optimal path.

Leighton et al [15] then showed that after initial routing, by rerouting a fraction of the flow of a poorly routed commodity onto the edges of another path in an appropriately derived auxiliary graph, we get a decrease in path congestion. In other words, the function iteratively reroutes parts of a flow to a different path, until path congestion is within a (1 + ε)-factor of optimal. That is, the Leighton algorithm [15] will come within epsilon of the truly (one can scale the epsilon smaller and smaller until the

---

[16] In the algorithm proposed by Leighton et al [15], there is an error parameter. Buying using it, one can scale the degree of accuracy of the algorithm, in exchange for running time.

[17] ε-optimal is here defined as if we can't route all the demands of all commodities during initial iteration

solution is truly globally optimal), globally optimal solution, and therefore it is an algorithm, not a heuristic[18].

The basic idea behind rerouting (the iterative steps) is to move flow off highly congested edges. The algorithm achieves this by calculating and assigning high *cost function* values to highly congested edges and small cost function values to lightly congested edges. Given these *cost function values* [15], which correspond to the *dual variables* [14] of linear programming, a flow that uses edges with *high cost function values* marks a poorly routed commodity. To reroute a flow, we just iteratively find some different path with lower cost function value and reroute the flow (fraction of the total demand) onto it.

There are two general steps in solving the Concurrent Flow problem using the algorithm developed by Leighton et al [15]; the following is a descriptive breakdown of each step.

1. The first step of the Concurrent Flow algorithm [15] is initial routing. The goal is to use the minimum cost flow routine to find an initial path for the commodity. The initial path serves as a starting point for the rerouting step and ignores any capacity constraints. We don't care if the initial path found is the best possible path choice, or even a feasible choice, for the moment.

2. The second step is to reroute a fraction of the bandwidth from the initial path to another discovered path. We use a minimum flow algorithm by Bertsekas and Teng[19] called RELAXT-III [21] to discover more paths; and for every

---

[18] However, later we formulate Leighton's algorithm to solve the Utility Model, thus creating our own heuristic.

[19] The minimum flow algorithm by Bertskeas and Teng [23] is an improved version of the RELAX algorithm, implemented in Fortran.

path discovered, *a cost function* [15] is applied to each edge, thus determining the percentage of demand to reroute to newly- found paths.

The cost function for *edge vw*, denoted by $c(vw)$, is determined by the function:

$$c(vw) = e^{\sigma \lambda f_{(vw)} / u_{(vw)}}$$

σ in this case is a constant determined by edge utilization and the capacity of the edge. $\lambda f_{(vw)}$ is the utilization on edge vw given the current path. $u_{(vw)}$ is the capacity of that edge given the current path. The cost function is applied to every edge for each new path discovered. It is easy to see that the cost function penalizes edges with low capacity and high edge utilization. Thus, by looking at the cost function of each edge on a path, we can determine the percentage of bandwidth to route through that path.

Suppose path p_1 is a newly discovered rerouting path for commodity 1 in graph G. Each edge of path p_1 is assigned a cost function value. The next step is to select, from the list of edges in p_1, the edge with the highest cost function value. The highest cost function value discovered becomes the *representative cost function value* for path p_1; and is the basis for deciding the percentage of total bandwidth to route to path p_1.

We use the newly calculated representative cost function value to determine the percentage of bandwidth to reroute over the Path p_1. Therefore the total bandwidth gets split up and some part is routed onto p_1; the amount depends on the maximum cost function value over the edges of p_1.

Figure 7 illustrates the Pseudo code for the rerouting step.

```
/* Initialize variables before first iteration */
   if (iteration == 1)
      initialize(&costvalue, &lowestMaxCongestion, &lowestRatio)

 /* Find max congestion*/
   max_congestion = getMaxCongestion();

 /* Calculate cost function of each edge */
   for (i = 1; i <= n_edges; i++) {
      CostFunction[i - 1] = getCostFunction(congestions[i - 1], alpha,
max_congestion);
   }
 /* Check if done, If Not Done, continue*/

/* Find commodity, min cost flow, and percentage to reroute */
   commodity = firstCommodity;

/* Find min cost flow for commodity */
getMinCostFlow(commodity, max_congestion);
/* Find percent to reroute */
percent = getPercent(commodity, max_congestion, &CostFunction);
         commodity++;

 /* Reroute flow */
adjustFlows(percent, commodity);
   }
}
```

Figure 7: Pseudocode for the Rerouting Step

Putting the initial routing step and the iterative rerouting steps together, we reach an optimal solution for the Concurrent Flow algorithm. That is, the solution reached should be epsilon-optimal with respect to the routing of a commodity's demand through a network.

Figure 8 illustrates the pseudo code for the general Concurrent Flow algorithm.

```
/* Step 1: Find an initial route for each commodity group */
   for (i = 1; i <= n_commodities; i++) {
      findInitialRoute(i);
   }

   /* Compute total flow for each edge after initial routing, for calculation*/
   for (i = 1; i <= n_edges; i++)
      total_flows[i - 1] = 0;
   for (i = 1; i <= n_commodities; i++)
      for (j = 1; j <= n_edges; j++)
            total_flows[j - 1] += individual_flows[n_edges * (i - 1) + (j - 1)]);

   /* Compute congestion for each edge after initial routing, for calculation */
   for (i = 1; i <= n_edges; i++)
      congestions[i - 1] = total_flows[i - 1] / capacities[i - 1];

   /* Initialize variables, after initial routing, iteration counter is set to 1 */
   iteration = 1;

  /* Step 2 and 3, Reroute flows until result is satisfactory , iteration increase*/
   while (decongest(iteration) == 1) {
      iteration++;
    reroute flows with each iteration
   update total flows and edge utilizations with each iteration
}
```

Figure 8: Pseudocode for the General Concurrent Flow Problem

3.3 The Unsplittable Flow Problem

In the last Section, we described an algorithm for the Concurrent Flow problem. To adapt the Multicommodity Flow algorithm to the Utility Model, the procedure for the Concurrent Flow problem is unsatisfactory due to the splitting of the per-customer traffic flows.

As noted above, in a network application, we can't split the flow for a particular customer in all cases. For example, if a customer wants to surf the Internet and look for information, we can split up the information for that customer's SLA session and route it through different paths. However, if the application involves multimedia traffic such as watching a movie or listening to a radio stream over the Internet, then it is better not to split the information for that movie or radio session. This way, we don't introduce *jitter* caused by variations in delay among the paths.

With some modification, we can change our procedure for the Concurrent Flow problem into an implementation for an Unsplittable Flow problem [11]. By using the Unsplittable Flow implementation; we can use a single path for each customer session.

3.3.1 The Underlying Theory

In an Unsplittable Flow problem [11], we treat an undirected graph $G(V, E)$ with a positive capacity $u(v, w)$ for each edge $v, w \in E$. There is also a set of commodities numbered *1* through *k*, where each commodity *i* is specified by a source-sink pair $s_i, t_i \in V$ and a positive demand d_i from its source s_i to its sink t_i. For each commodity *i*, we ship an amount proportional (a percentage, if not all) to its demand d_i from its source s_i to its sink t_i on one single path. A positive edge flow

$f_i\ (vw) > 0$ denotes a forward flow of commodity i with respect to the direction of edge (vw) [20].

The Unsplittable Flow problem [11] differs from the regular Concurrent Flow problem [6] by dedicating a single path to each commodity (SLA flow), instead of splitting the demand up and spreading it over two or more paths.

Figure 9 illustrates the difference between a solution for the Unsplittable Flow problem and a solution for the Concurrent Flow problem (Figure 6). Consider the following graph with a single commodity to flow from node 1 to node 4 and a demand of 4. Figure 6 showed a solution for the regular Concurrent Flow problem. Here we are allowed to split up each demand and send it through more than one path. To solve the Concurrent Flow problem, observe that there are three possible paths from node 1 to node 5. The demand of 4 gets separated and shipped through 3 paths. The 3 possible paths are

- Path 1, 2, 5.
- Path 1, 3, 5.
- Path 1, 4, 5.

Each path now gets a portion of the demand. By splitting the total demand and routing fractions of total flow through different paths, and by adjusting the amount of flow routed through each path, we can achieve the optimal solution.

However, in an unsplittable flow problem [11], each demand is restricted to one path; the best solution is determined by the edge capacities and the available surplus

---

[20] A negative flow denotes a flow in the opposite direction

for each edge (how much free bandwidth is left for each edge). In this example, the optimal solution is path 1, 4, 5 because the lowest edge capacity in path 1, 4, 5 is 3 units, the highest from the three path choices we have; the edge utilizations are all 0 right now. Therefore, path 1, 4, 5 is the best solution path for the unsplittable flow version of the problem.

The following Figure illustrates the solution for the Unsplittable Flow problem [11]; the solution path is highlighted.

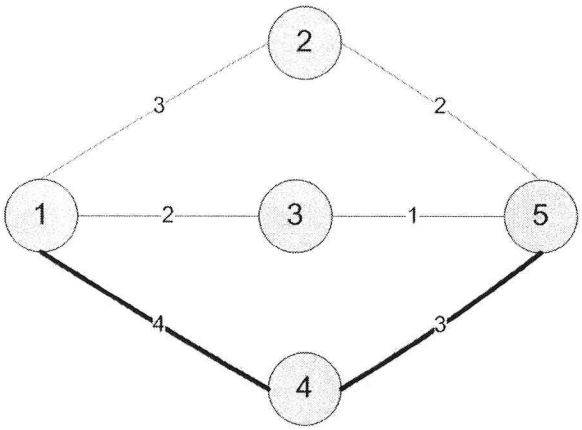


Figure 9: Example of a Unsplittable Flow Problem

Because the total demand in this example is 4 units, we are unable to route all 4 units through one single path; the best we can do is to route 3 units across a single path.

From the above comparison, we observe that in concurrent flow problem, the path with the largest percentage of bandwidth routed is the solution path for the unsplittable flow problem. We will explain why this is the case and how the distribution equation can assign a large percentage of the flow to the best possible path in the next Section.

3.3.2 The Unsplittable Flow Algorithm

In Concurrent Flow implementation, the rerouting class **decongest** handles the task of rerouting portions of the customer's demand to different paths. For each new path discovered, the distribution formula (a built in formula in decongest) determines what percentage of the total bandwidth each path can accept.

Recall that the main step of the distribution formula [15] is as follows

1) The cost function [15] first assigns each edge from the path with a cost value. The cost function is

$$c(vw) = e^{\sigma \lambda f_{(vw)} / u_{(vw)}}$$

σ in this case is constant. $\lambda f_{(vw)}$ is the congestion on edge vw given the current path. $u_{(vw)}$ is the capacity of that edge given the current path.

2) For each possible path discovered, we pick the edge with the worse cost function value and use it to determine the percentage of bandwidth that is routed through that path.

From this breakdown of the distribution function, we know that the percentage of the total bandwidth that gets routed for each path depends on the cost function value; and the cost function value depends on the capacity of the edge and the edge utilization. Therefore, the cost function favors lightly loaded edges and utilizes them to the extent allowed by their adjusted capacities. If a path discovered has the smallest *representative cost function value*[21], it means that such a path will have a set of edges

[21] See page 29 for the definition of representative cost function value

that are least utilized and thus have the biggest unused capacities amongst other path choices for the same commodity.

Therefore, the path with the greatest percentage of bandwidth assigned to it is the solution path for the Unsplittable Flow algorithm (for that commodity).

Now, with some adjustment, we know how to change the solution of a Concurrent Flow solution to an unsplittable one. Given the list of available paths and the percentage of bandwidth per path, we choose the path with the highest percentage of bandwidth routed. That path then becomes the solution path for the Unsplittable Flow problem.

3.4 Solving the Utility Model Using Unsplittable Flow Algorithm

We have illustrated the concepts of Concurrent Flow problem and Unsplittable Flow problem in the previous two Sections. Now we can combine the two concepts and explain the solution of the Utility Model.

3.4.1 The Underlying Theory

In order to understand this new solution technique for the Utility Model, we need to review some of the concepts mentioned in the previous sessions. Recall that the definition of the Unsplittable Flow problem [11] specifies: given an undirected graph $G(V,E)$ with a positive capacity $u(vw)$ for each edge $vw \in E$, we wish to find a single path to route each commodity i from s_i to t_i, with demand d_i (observe, that each commodity i from the Unsplittable Flow problem, can be one single level for one single SLA in the Utility Model).

So far, we only consider bandwidth as a resource constraint, and the demand *di* can be viewed as bandwidth demands from the customer SLA. We will deal with latency as a resource constraint in Section 3.4.4.

Figure 10 illustrates the idea. For example, if SLA 1 (with 3 levels: bronze, silver, gold) starts from node 1, ends at node 4, we then can map it into a Multicommodity Flow problem with a set of 3 commodities, all having $s_1 = 1$, $t_1 = 4$, but with three different values of demand.

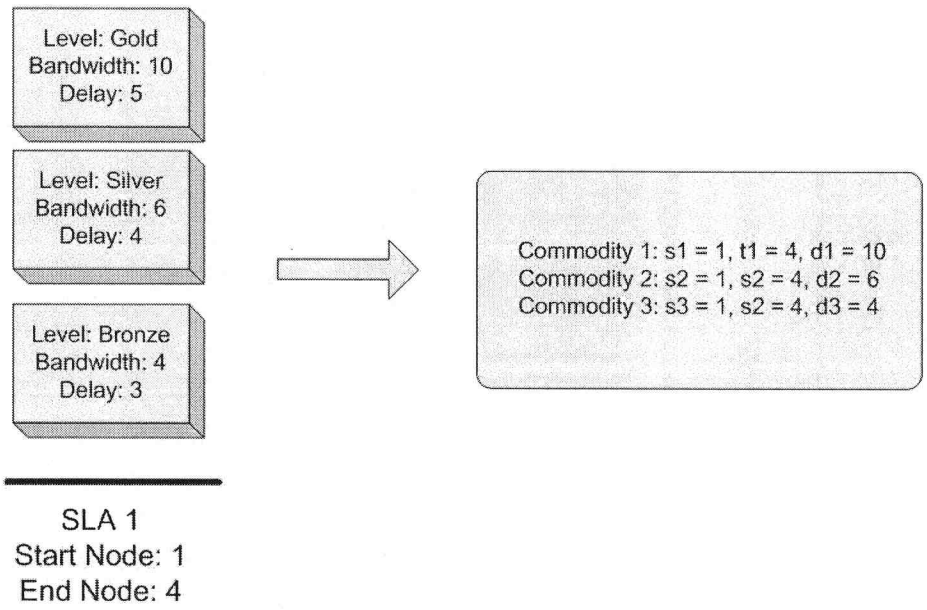


Figure10: Mappping a Customer SLA onto Several Commodities

However, we do not necessarily need to use three commodities to solve for a single SLA. Observe that for all the levels within a customer SLA, the source and destination nodes are the same. The only difference among levels is the bandwidth and latency demands. Therefore, we can map one of the levels of a SLA onto a commodity, and attempt upgrades by simply changing the demand of that commodity, scaling it upward or downward.

Taking all these points into consideration, two methods of admission for the Utility Model were devised. The first method is called the *top down method.* The top down method is used when there is little or no contention for bandwidth in the network. In other words, if the network is big enough, the bandwidth capacity of the edges is

large enough, and there are lots of path choices to choose from, then we can consider using the top down method, admitting all the customer SLAs from the highest level and then slowly downgrading the over utilized edges.

The second method is called the *bottom up method*. This method is for use when there is significant contention for bandwidth. In that case, if we cannot admit every customer SLA at its highest QoS level, we can start by admitting the level with the highest price/bandwidth ratio, and considering upgrading or downgrading after all the SLAs have been initially admitted. The bottom up method is a modified version of [5], incorporating concepts of Multicommodity Flow [6].

It's worth to mention that there is an underline assumption involved in solving the Utility Model using Multicommity Flow algorithm. That is, we are assuming that for each SLA, the bandwidth demand of each level is ordered in a monotonic increasing fashion. For example, for an SLA, we are assuming that the bandwidth demand for a higher level (say, the gold level) of SLA is greater or equal to a lower level of SLA (say, the silver level).

The reason for this assumption is that when we initially route one of the level of an SLA, that is, we allocate a path for one of the levels. Later when we upgrade or downgrade the current SLA, we are assuming upgrading to a higher level for an SLA means that the bandwidth demand increases, and we have to allocate more resource (bandwidth) for the SLA over the current path. We are also assuming that by upgrading to a higher level successfully, the user should gain more bandwidth for their session.

3.4.2 The Top Down Method

The top down method is used when there is no significant contention for bandwidth, i.e. when ample bandwidth is available. We first admit all SLAs at the highest level. Once all the SLAs are admitted and assigned paths, we then look for over-utilized

edges and downgrade SLAs with the lower price / bandwidth ratios[22]. The general steps are as follows

1) Route all the customer profiles at the highest QoS levels proposed in the SLAs.

2) Look for over-utilized edges.

3) For every over-utilized edge (occupied by >= *1* customer SLAs), we downgrade the customer SLA with the lower price/bandwidth ratio, until that edge's utilization is <= *100%*.

Edge utilization in this case means fraction (up to 100%) of bandwidth from an edge that is assigned to commodities (one ore more than one commodities can be assigned bandwidth from an edge). Therefore, an over-utilized edge in this case means an edge such that the fraction of bandwidth assigned to commodities is > 1.

It is clear that if there are many over-utilized edges, the top down admission method would go through each of them and downgrade all the SLAs sharing that edge, until its utilization is smaller than 1. Each of the over-utilized edges requires us to go through one or more iterations of downgrade (usually more). Hence this method could exhibit disastrous running times in heavily loaded networks. Therefore we should use the top down method, only for lightly-loaded networks.

3.4.3 The Bottom Up Method

The bottom up method is similar to Khan's [5]. However, because we are solving the problem by the unsplittable flow method [11], we know the maximum utilization and

---

[22] That is, we look for the over-utilized edge, and if that edge has two or more commodities (FLOWS) routed upon it, we downgrade the commodity with the lower price/bandwidth ratio. Otherwise, if that

the minimum bandwidth surplus of each path. These extra items of information make the upgrading process easier.

The steps of the bottom up method are:

1) Start with the lowest QoS level of each SLA (this is the starting point for the bottom up level).

2) Group all the QoS levels from customer SLAs together in one array. Sort the array by *price / bandwidth* .

3) From the top (the level with the highest price / bandwidth ratio), we pick the highest *price / bandwidth* level from each SLA. Those levels will be the foundation levels; we solve the Unsplittable Flow problem using the bandwidth requirement for the foundation level.

4) After solving the Unsplittable Flow problem, we upgrade or downgrade each SLA based on the free or unreserved bandwidth of the path,

This method differs from Khan's method [5] in that there is no unnecessary downgrading or upgrading step. After solving the unsplittable flow problem, the surplus or unallocated bandwidth on each edge is known. Hence we can finish upgrading in one simple step; there will more discussion of upgrading in Chapter 4.

---

edge has only one commodity routed upon it, we just downgrade that commodity.

3.4.4 Latency As a Resource Constraint

As mentioned in previous Sections, latency can also serve as resource[23]. However, latency as a resource is different than bandwidth, as it resembles a constraint more than a consumable resource. A path can satisfy a certain QoS level Q_i of an SLA iff

1. The path's latency l_p is less than or equal to the QoS level's required latency l_Q, and

2. Each link in the path has uncommitted bandwidth b greater than or equal to the QoS level's required bandwidth b_Q.

Based on the following procedure [9], we can treat latency as a constraint during the Concurrent Flow implementation. That is, after a list of paths has been discovered during the Concurrent Flow step, we calculate the total latency for each candidate path along with the percentage of bandwidth assigned to it. After solving the Unsplittable Flow problem, we try to find a level within the SLA such

1) The latency requirement of the level picked is greater than or equal to the total latency of the Unsplittable Flow path.

2) The bandwidth requirement of our SLA can be found on the Unsplittable Flow path.

For each SLA, we keep a list of 3 or 4 major paths (paths on which we route more than 10% of bandwidth, generated during the solution of the concurrent flow problem) as backup paths. In the (hopefully rare) case that, after we iterate through all the levels of an SLA (by upgrading or downgrading), and none of the latency constraints from any level of the SLA is satisfied by the unsplittable flow path, we

---

[23] For definition of latency as a resource in packet switching networks, see Section 2.2

consider routing the SLA to one of the other major paths, starting with the path with the second greatest percentage of bandwidth routed on it.

Overall, the steps of downgrading or upgrading with latency as a constraint are:

- We first execute the Concurrent Flow algorithm, to find a list of candidate paths, along with the percentage of bandwidth assigned to each path.

- We solve for the unsplittable flow problem. If the latency of the unsplittable path solution strictly greater than the latency constraint of the initial QoS level picked for the SLA, we try other QoS levels.

- If, after we try all the levels of an SLA, the latency of the unsplittable path solution is still strictly greater than the latency requirement of all the levels within the SLA, we give up on this path, and consider other candidate paths.

3.4.5 The Method Switcher

From the description of the top down and bottom up methods, we conclude that it is important to decide when to use each method. Hence a method switcher had to be designed.

The design of the method switcher is based on the rule that the top down method is used only when there is little or no contention for the bandwidth; otherwise, the bottom up method is used. The key for the design of the method switcher is to find over-utilized edges, and to set a limit on number of over utilized edges.

In MultiUtility, such a limit is set using the following formula

Switcher Number = CEIL[(Total Edge Number)/3]

For example, if there are 5 edges in the graph, so the switcher number will be 5 / 3 = 1.66667, taking the integer ceiling of that, the switcher number is 2. Hence if there are 2 or more over-utilized edges, we use the bottom up method.

The overall switching method is designed as follows:

1. Route all SLAs using the top down method.

2. Based on 1., find the switcher number. If the switcher number indicates that we need to use the bottom up method, we route all SLAs again at the bottom level.

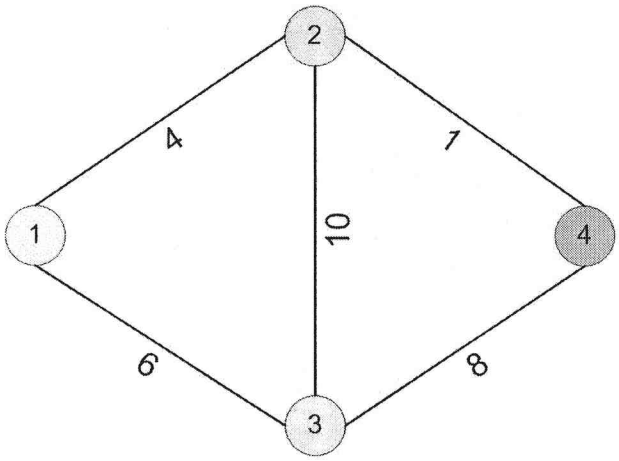
3. Otherwise, we proceed with the upgrading/downgrading process.

The switcher method in MultiUtility is fairly simple: no doubt more efficient switcher methods can be devised.

4 Solution, By An Example

We will illustrate the details of our new method of solving the Utility Model by solving an example and describing it step by step. Consider the following network graph *G* and two customer SLAs, *SLA1* and *SLA2*, as illustrated in Figure 11. Each customer SLA has 3 levels, Gold, Silver and Bronze; and each level maps to a bandwidth demand and a bidding price for each customer. (We will discuss the latency constraint later.)

For simplicity, we only consider 3 levels of demands for each customer SLA. However, the implementation itself can support as many levels of demand as desired.



Customer Session	**Source/Destination**	**Level 3 (Gold)**	**Level 2 (Silver)**	**Level 1 (Bronze)**
SLA 1	Node 1/Node 4	Bandwidth = 10 Price = 5	Bandwidth = 6 Price = 4	Bandwidth = 4 Price = 3
SLA 2	Node 2/Node 4	Bandwidth = 2 Price = 10	Bandwidth = 1 Price = 2	N/A

Figure 11: The Utility Model: Example Network and SLA Profiles

4.1 Example: Top Down Method

4.1.1 Initial Routing: Top Down Method

The top down method is executed first, and so the levels of QoS that are considered initially are the highest levels for both SLA1 and SLA2.

Figure 12 illustrates the initial level and information that are considered for both SLAs:

Session	Source	Destination	Starting Bandwidth	Price
Customer Session 1	1	4	10	5
Customer Session 2	2	4	2	10

Figure 12: Top Down Method: Initial Routing Information

Given the starting levels, we solve for an Unsplittable Flow solution for both SLA1 and SLA2. That is, we look for the best possible single path to route both customer levels.

In order to solve for an Unsplittable Flow solution, we must first solve for a Concurrent (splittable) Flow solution. SLA1 is routed first, with 10 units of demand.

Recall that the steps of the Multicommodity Flow implementation are as follows:

- Find a path using the min-cost flow algorithm [22]

- For each path found, according to cost function $c(vw) = e^{\sigma\lambda f_{(vw)}/u_{(vw)}}$, assign a cost value to each edge.

- Pick the maximum cost function value from the list of edges in the path.

- From the maximum cost function value, determine the percentage of the total bandwidth to route to a particular path.

In this case, the min-cost flow algorithm found path 1 2 4, path 1 3 4 and path 1 2 3 4. According to the percentage calculation, the bandwidth distribution is as follows:

- Path1: 1, 3, 4, the main path, get 5.1982 units of the total 10 units of bandwidth (about 52% of the total bandwidth).

- Path2: 1, 2, 4, gets 3.422516 units of the total 10 units of bandwidth (about 34% of the total bandwidth) Note that this is more bandwidth than the path can carry – it is limited to 1 unit by edge 24.

- Path3: 1, 2, 3, 4, gets 1.379279 unit of the total 10 units of bandwidth (about 14% of the total bandwidth).

After solving the Multicommodity Flow problem, using the Unsplittable Flow algorithm, the path that receives the greatest percentage of the bandwidth would be the solution for the Unsplittable Flow problem. In this case, path 1 3 4 is the best path choice to route all 10 units of bandwidth for SLA1; even though this exceeds the path capacity, we still route 10 units of bandwidth upon path 1 3 4 (Later, we can downgrade the bandwidth demand to fit the path capacity, right now we only need to discover the best path choice.).

We do the same with SLA2; first we assign the cost function value to each edge, then we split up the 2 units of bandwidth demands for SLA2, and attempt to solve the Multicommodity Flow problem for SLA2 (Gold level), again using the Unsplittable Flow algorithm. The solution is as follows:

- Path 2 3 4 gets 1.379687 units of the total 2 units of bandwidth (about 70% of the total bandwidth)

- Path 2 4 gets 0.620313 units of the total 2 units of bandwidth (about 30% of the total bandwidth)

Hence, according to the Unsplittable Flow algorithm, path 2 3 4 is the best path to route the 2 units of bandwidth demand for SLA2.

Now, we have a solution for the initial routing of both SLAs. For SLA1, 10 units of bandwidth are routed along path 1 3 4. For SLA2, 2 units of bandwidth are routed along path 2 3 4.

After initial routing, the total flow and edge utilization of each edge are also outlined in Figure 13:

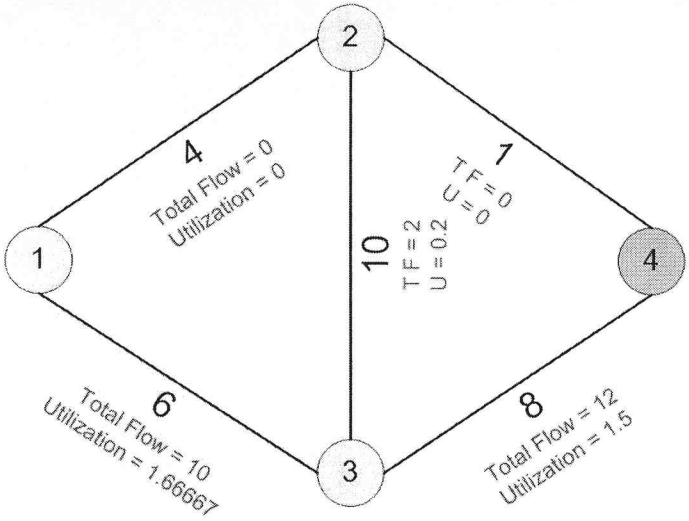


Figure13: Top Down Method: After Initial Routing

4.1.2 Upgrading and Downgrading: Top Down Method

According to Figure 13, we can see that edge 1 3, and edge 3 4 are both over utilized after the initial routing[24]. Recalling the top down approach, the steps of the approach are:

- Look for over-utilized edges in the graph

- For every over-utilized edge, we downgrade the customer SLA with the lowest price/bandwidth ratio, until that edge utilization is *less than or equal to 1*.

Knowing that edge 13 and edge 34 are over utilized, we go through edge 13 and edge 34 to downgrade the customer SLA with the lower price/bandwidth ratio. The process is as follows:

---

[24] An *over utilized edge* is defined as one having edge utilization of > 1.0

- Check edge 1 3, which is only occupied by SLA1, so we downgrade SLA1 from level 3 (GOLD, with 10 units of bandwidth demand) to level 2 (SILVER, with 6 units of bandwidth demand).

- Due to the downgrade, the total flow for edge 1 3 becomes 6 units; the edge utilization for edge 1 3 becomes 6/6 = 1.

- Edge 3 4 is also affected by the downgrading of SLA1; total flow on edge 3 4 lowered to 8 units, the edge utilization for edge 3 4 then becomes 1.0. There are no more over-utilized edges after the downgrading, thus we stop. The final routing information for SLA1 and SLA2 is illustrated in Figure 14:

Session	**Source**	**Destination**	**Final Bandwidth**	**Price**
Customer Session 1	1	4	6	4
Customer Session 2	2	4	2	10

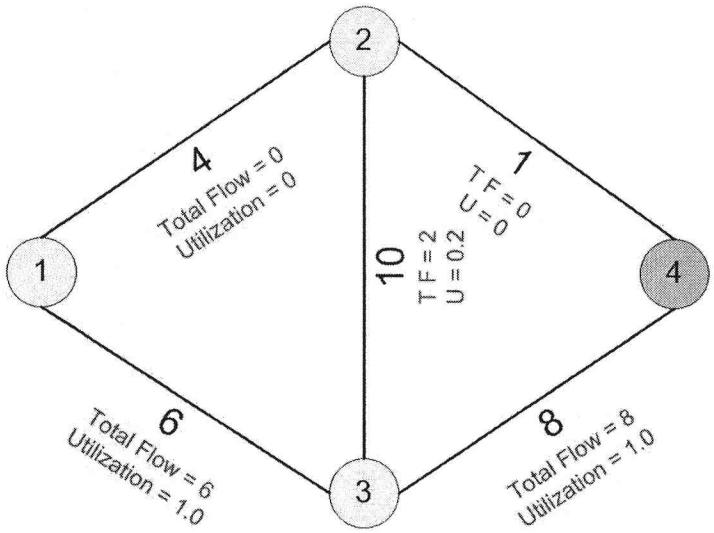


Figure 14: Final Solution Using Top Down Method

4.2 Example: Bottom Up Method

Suppose the bottom up method is chosen as the method of admission. Thus, the initial levels that are admitted are the lowest levels for both SLA1 and SLA2. SLA1 will be routed at BRONZE level and SLA2 will be routed at SILVER level. The following table - Figure 15 - shows the initial QoS level and other information that is considered for both SLAs:

Session	**Source**	**Destination**	**Starting Bandwidth**	**Price**
Customer Session 1	1	4	4	3
Customer Session 2	2	4	1	2

Figure 15: Bottom Up Method: Initial Routing Information

We then use the Multicommodity Flow algorithm to find a path for 4 units of bandwidth demand for SLA1 and 1 unit of bandwidth demand for SLA2. The path finding process is similar to the path finding step in the top down example[25].

After solving the Unsplittable Flow problem, the best path for SLA1 is path 1 3 4. The best path for SLA2 is path 2 3 4. Both SLA1 and SLA2 can be routed along these paths. The edge utilization after the initial routing is specified in Figure 16.

---

[25] See Section 4.1.1 for the top down method's initial routing process

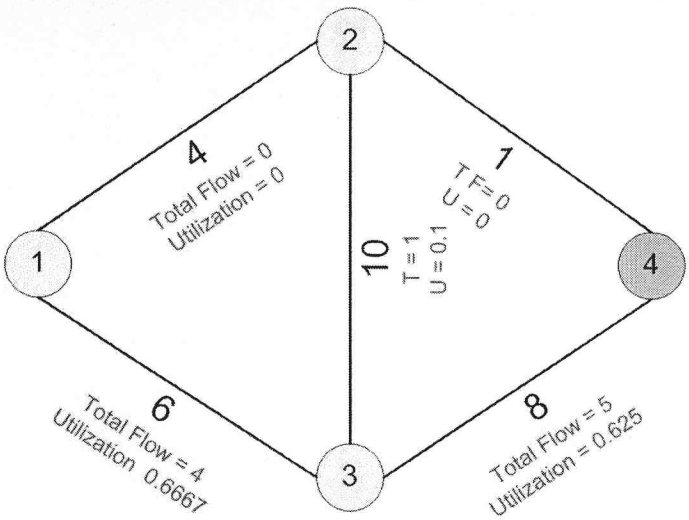


Figure16: Bottom Up Method: *After Initial Routing*

After the initial routing, we proceed with the upgrading/downgrading process. The steps are as follows:

- For SLA1, to route 4 units of bandwidth, we have to go through edges 13 and edge 35. After initial routing, we determine that the most congested edge that carries SLA1 is edge 1 3. As of now, edge 1 3's utilization is 0.667, with 2 units of surplus bandwidth.

- For SLA2, to route 1 unit of bandwidth, we have to go through edges 2 3 and 3 4. After initial routing, we determine that the most congested edge carrying SLA2 is edge 3 4. As of now, edge 3 4's utilization is 0.625, with 3 units of surplus bandwidth.

- We calculate the price/bandwidth ratio for SLA1's Bronze level and SLA2's Silver level. In this case, it's more feasible to upgrade SLA2 due to its higher price/bandwidth ratio. We upgrade SLA2 from silver level to gold level. SLA2 has now reached to the highest possible QoS level.

- We still can fit in 2 units of bandwidth on both edge 13 and edge 34, so we give the rest to SLA1. SLA1 is thus upgraded from Bronze Level to Silver level.

The final routing decision is that SLA1 is routed at Silver level and SLA2 is routed at GOLD level. For this example, by coincidence we find the same solution as the one produced by the Top Down Approach. The total flow and edge utilization using the bottom up approach is shown in Figure 17.

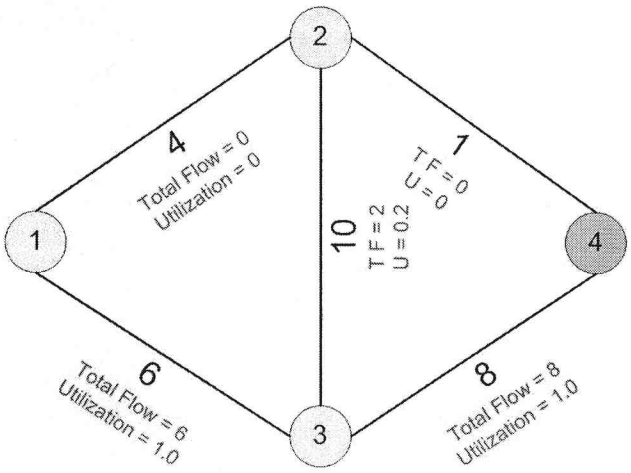


Figure 17: Final Solution Using Bottom Up Method

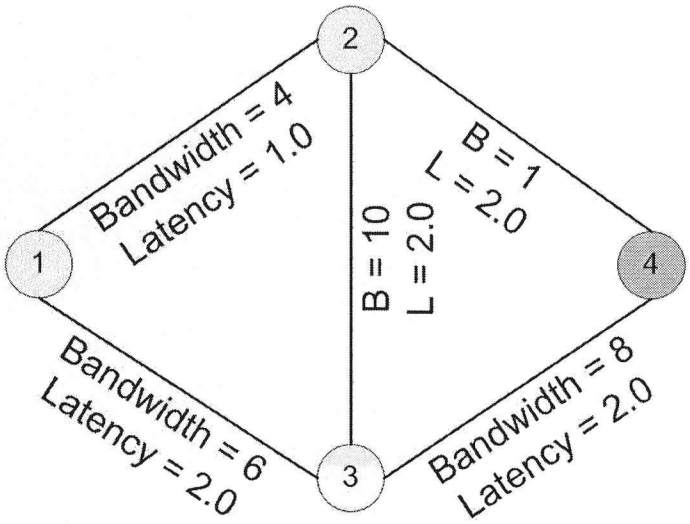
4.3 Example: Latency As Resource Constraint

As mentioned in previous Sections, latency can also serve as a resource[26]. Latency as a resource is unlike bandwidth: While a connection with a certain amount of latency may be required, it is not *consumed* in the same way as bandwidth. Put another way, bandwidth is *conserved* –if I have it, you can't have it. Latency is not conserved. Thus, latency resembles a constraint more than a consumable resource. Latency is handled as a constraint in the Multicommodity method. Furthermore, the latency

[26] For latency as a resource constraint, also see Section 22, and Section 3.4.4

constraint (of a customer SLA) acts as an upper bound – less latency than the constraint value is considered acceptable, in fact it is usually preferable.

Suppose we now add in latency for our top down example, then the new SLA profiles and the network graph are shown in Figure 18.



Customer Session	**Source/Destination**	**Level 3 (Gold)**	**Level 2 (Silver)**	**Level 1 (Bronze)**
SLA 1	Node 1/Node 4	Bandwidth = 10 Latency = 3.2 Price = 5	Bandwidth = 6 Latency = 4.1 Price = 4	Bandwidth = 4 Latency = 4.7 Price = 3
SLA 2	Node 2/Node 4	Bandwidth = 2 Latency = 5.2 Price = 10	Bandwidth = 1 Latency = 2.1 Price = 2	N/A

Figure 18: SLA and Network with Bandwidth and Latency As Constraints

We will now solve this example using only the top down approach, which suffices to show the treatment of latency. Just as before, we admit SLA1 and SLA2 at the gold level. SLA 1 is routed initially at 10 units of bandwidth; SLA s2 is routed initially at 2 units of bandwidth.

We first solve for the two SLAs using the Concurrent Flow algorithm. For SLA 1, as shown in the previous Section, 3 paths were found. For SLA 2, 2 paths were found. However, this time, we will also take latency into consideration. The calculation is as follows

For SLA 1:

- Path 1, 3, 4, gets 5.1982 units of the total 10 units of bandwidth, the latency for path 1, 3, 4 is 2.0 on edge 13 plus 2.0 on edge 34 for a total latency of 4.0

- Path 1, 2, 4, gets 3.422516 units of the total 10 units of bandwidth, the latency for path 1,2,4 is 1.0 + 2.0 = 3.0

- Path 1, 2, 3, 4, gets 1.379279 unit of the total 4 units of bandwidth, the latency for path 1, 2, 3, 4 is 5.0

The above are all the major paths to consider for SLA1; they are ranked from the most feasible path (the path we route the greatest percentage of bandwidth to) to the least feasible path (least percentage of bandwidth). Our strategy is to consider the path with the greatest percentage of traffic routed first, because such a path is the solution for the Unsplittable Flow problem. If SLA 1's initial level has a latency requirement less than or equal to the latency of the most feasible path; we consider downgrading SLA 1[27].

We also keep other major paths (the rest of the path on the list) as backup choices. In case the most feasible path has excessive latency we consider the rest of the path choices.

---

[27] For this example we're using the top down method. If the method of admission is the bottom up method, we can look through other levels by upgrading.

For SLA 2:

- Path 2 3 4 gets 1.379687 of the total 2 units of bandwidth, the latency is 4.0

- Path 2 4 gets 0.620313 of the total 2 units of bandwidth, the latency is 2.0.

As we know, SLA 1 at Gold level of QoS requires at most 3.2 units of latency. We know that path 1,3,4, the Unsplittable Flow path for SLA 1, has total latency of 4.0, which is higher than the latency constraint for SLA 1's Gold level. The total path latency has to be less than or equal to the QoS level's latency constraint; therefore path 1, 3, 4 cannot meet the latency requirement of SLA 1's Gold level.

Hence we downgrade SLA 1. At Silver level, the Latency requirement is a less-demanding value of 4.1, which is fortunately higher than the latency of path 1,3,4 (with 4.0 total latency). Also, after solving the Unsplittable Flow problem for SLA 1, we know that path 1, 3, 4 is capable of routing at most 6 units of bandwidth, which fits the requirement for SLA 1's Silver level, thus the final admitted level for SLA 1 is Silver.

In this case, after downgrading SLA 1 from Gold level to Silver level, we were able to route SLA 1 onto the Unsplittable Flow path. However, suppose we had a case where, after we downgraded SLA 1 to Bronze level, path 1, 3, 4 's latency was still too high for the Bronze level of SLA 1. We would then consider routing SLA 1's traffic to path 1, 2, 4 (which is the path with the second greatest percentage of bandwidth routed on it) and so on.

For SLA 2, the path we route the greatest percentage of bandwidth on is path 2 3 4. The total latency for path 2, 3, 4 is 4.0. At Gold level, the latency constraint for SLA 2 is 5.2. The total latency for path 2, 3, 4 is less than the latency constraint of SLA 2's

Gold level. Thus we can route the traffic for SLA 2 on path 2, 3, 4. The final admitted level for SLA 2 is Gold

The final routing information (with Latency constraints) is as illustrated in Figure 19.

Session	**Source**	**Destination**	**Final Routing Information**	**Path**
Customer Session 1	1	4	Bandwidth = 6 Latency = 4.1 Level = Silver	1, 3, 4 Latency =4.0
Customer Session 2	2	4	Bandwidth = 2 Latency = 5.2 Price = Gold	2, 3, 4 Latency = 4.0

Figure 19: Final Routing Information (with bandwidth and latency as constraints)

5 Implementation

In the previous Chapters, we described ways of solving the Utility Model as formulated by Khan [5] and Watson [9], and we described our new heuristic approach to Service Level Agreements, routing paths, and Quality of Service mappings, which is formulated as a Multicommodity Flow Problem [6]. Both Khan [5] and Watson [9] used heuristics to solve the Utility Model, which they formulated as a Knapsack Problem [5]. Our heuristic focuses on selecting an optimal path for one of the level of each SLA (depending on the method of admission), then upgrade and downgrade from the path initially selected. For our heuristic; even though the path allocated for the initial level of each SLA is optimal, the final solution after upgrading and downgrading might not be the optimal solution; more on the optimality of our heuristic in Chapter 6.

In this Chapter, we will present implementation details of a prototype C implementation of the Multicommodity Flow based solution of the Utility Model called MultiUtility, including the inevitable differences between the implementation and the theoretical model.

5.1 Overview

Most of MultiUtility, including the initial admission process, the upgrading process, and the main data-structure is implemented in C. At the core of the Multicommodity Flow algorithm, the min-cost path finding algorithm that is responsible for finding

paths for each commodity, is coded in FORTRAN[28]. A FORTRAN to C converter is also coded to adapt the min-cost algorithm to rest of the implementation.

MultiUtility consist of several classes:

- The *Initial Routing Class* handles the initial routing for each customer SLA. That is, it allocates an initial routing path for each customer SLA by solving the Multicommodity Flow algorithm, which in turn leads to an Unsplittable Flow solution. Therefore, each SLA is admitted at its initial QoS level, which means at the highest QoS level for each SLA, if the method of admission chosen is the top down method, and at the lowest level if the method of admission chosen is the bottom up method. The initial routing class then attempts to find an initial path for all of the initially selected levels.

- The *Upgrading/downgrading Class* handles the upgrading or downgrading process for each SLA. It tries to increase or decrease the bandwidth demand, depending on the method of admission while maintaining the same path choice allocated for each SLA from initial routing.

- *The Unsplittable Flow Class* solves the Unsplittable Flow problem for selected level(s) of each SLA (depending on the method of admission). The class first follows the steps listed in Chapter 4 to obtain the unsplittable solution.

Multiutility also has input and output files, structured as follows:

---

[28] We chose to implement MultiUtility in C for efficiency. The path algorithm RELAX-III by Bertsekas and Teng is implemented in FORTRAN, so we made a translation class for interfacing purpose.

- The *Input File* is a text file that represents the network under analysis, and stores current capacities, cached paths, and node information. The input file also has the *SLA directory*, which stores SLA profiles, including their QoS mappings.

- The *output file*, which includes the final admission level and path information for each SLA

5.2 The Input File

5.2.1 Graph Information in Input File

To MultiUtility, a *network* consists of a set of nodes and a set of edges or links. Each node and each link is numbered in the input file. In the input file, each link identifies the connection of two numbered nodes. For example: link1 connects node 2 and node 4.

Figure 20 specifies the graph information in the format used in the input file of MultiUtility.

p mcf 4 5 2
a 1 2 4 1
a 1 3 6 2
a 2 3 10 2
a 2 4 1 2
a 3 4 8 2s

Figure 20: Graph Information in the Input File

The file starts with the header "p mcf" indicating the start of the file, followed by the number of nodes in the graph, the number of edges in the graph and the number of SLAs (customers) in the graph. In this particular example, "p mcf 4 5 2" indicates

that there are 4 nodes, 5 links in the graph and there are 2 customers' SLAs waiting for admission.

The line starting with the letter "a" indicates the link information. For example "a 1 2 4 1" indicates that link1[29] is connected between node 1 and node 2, and the bandwidth capacity of the link is 4 units, and the latency capacity of the link is 1 units. The links are numbered the same order as they appear in the input file.

Overall, if we translate the information in Figure 20, the graph information is as follows:

p mcf 4nodes 5links 2SLAs
Link1: between(node1, node2), bandwidth4, latency 1
Link2: between(node1, node3), bandwidth6, latency 2
Link3: between(node2, node3), bandwidth10, latency 2
Link4: between(node2,node4), bandwidth1, latency 2
Link5: between(node3,node4), bandwidth8, latency 2

Figure 21: Translation of Graph Information in the Input File

5.2.2 Customer Information in Input File

The input file for MultiUtility also specifies SLA information, listed as the combination of source node, destination node, and levels; each level has a combination of one bandwidth demand, one latency demand and one asking price.

When a SLA is being specified in an input file, we first list its source node number, and its destination node number, and then its level of service, which is specified by the price and the amounts of bandwidth and latency demanded.

---

[29] the link number is implicit, derived from the position in the input file; the first link appearing in the input file will be numbered link 1, and second link 2 and so on

Figure 22 gives an example of SLA information in the format used in the input file of MultiUtility.

SLA 1 4
Level 3: Bandwidth 10 Price 5
Level 2: Bandwidth 6 Price 4
Level 1: Bandwidth 4 Price 3

SLA 2 4
Level 2: Bandwidth 2 Price 10
Level 1: Bandwidth 1 Price 2

Figure 22: SLA Information in the Input File

The graph input information in Figure 20 and the Customer SLA input information in Figure 22 correspond to the graph and input information used in an example earlier, outlined in Figure 14. The example was explained in detail in Chapter 4.

5.3 Solving the Concurrent Flow Problem in MultiUtility

The Unsplittable Flow class in MultiUtility is responsible for solving the Unsplittable Flow problem for each SLA in the input.

Given the example input files in Figures 20 and 22, we suppose we are using the top down method as the method of admission (the same example had been illustrated in 4.1.1[30]); we have the following Concurrent Flow output listed in Figure 23.

[30] In this Chapter, we are only showing the format of the inputfile, concurrent flow output, unsplittable flow output, and final output file. The same example was explained in detail in Chapter 4.

```
COMMODITY 1


Path 1: Edge 1, Edge 4, Flow = 3.422516
Path 2: Edge 2, Edge 5, Flow = 5.1982
Path 3: Edge 1, Edge 3, Edge 5, Flow = 1.379279

  Accumulated Flow Edge 1      Flow = 4.801794
  Accumulated Flow Edge 2      Flow = 5.198206
  Accumulated Flow Edge 3      Flow = 3.422516
  Accumulated Flow Edge 4      Flow = 1.379279
  Accumulated Flow Edge 5      Flow = 8.620721


COMMODITY 2

Path 1: Edge 3, Edge 5, Flow = 1.379687
Path 2: Edge 4, Flow = 0.620313

  Accumulated Flow Edge 1      Flow = 0.000000
  Accumulated Flow Edge 2      Flow = 0.000000
  Accumulated Flow Edge 3      Flow = 1.379687
  Accumulated Flow Edge 4      Flow = 0.620313
  Accumulated Flow Edge 5      Flow = 1.379687

TOTAL FLOWS, CONGESTIONS
  Edge 1    Total Flow=4.801794     Congestion=1.200449
  Edge 2    Total Flow=5.198206     Congestion=1.299551
  Edge 3    Total Flow=4.802203     Congestion=0.480220
  Edge 4    Total Flow=1.999591     Congestion=1.999591
  Edge 5    Total Flow=10.000409    Congestion=2.000082
```

Figure 23: Concurrent Flow Output

The output for the Concurrent Flow first lists the list of paths allocated to route each commodity's flow, as well as the amount of flow that was routed (onto the path). For example, as we can see, for commodity 1, there were 3 paths allocated, and the 10 units of flow are split into 3.422516, 5.1982, and 1.379279 units and routed through those 3 paths.

The output file also lists the accumulated flow over each edge after routing traffic for a commodity. For example, for commodity 1, after splitting 10 unit of flow and routing it through 3 different paths, edge 1, which was used by path 1 (to route 3.422516 units of flow) and path 3 (to route 1.379279 units of flow), has accumulated flow of 4.801795 units.

Finally. The last Section of the output file lists the total amount of flow accumulated over each edge after all of the commodity had been routed (in this case, after all 2 commodity had been routed).

5.4 Solving the Unsplittable Flow Problem in MultiUtility

After obtaining the Concurrent Flow solution, it is easy to get an Unsplittable Flow solution. Figure 24 shows the output for Unsplittable Flow (the same example is illustrated in detail in Section 4.1.1).

```
COMMODITY 1
  Edge 1     Flow = 0
  Edge 2     Flow = 10
  Edge 3     Flow = 0
  Edge 4     Flow = 0
  Edge 5     Flow = 10

COMMODITY 2
  Edge 1     Flow = 0
  Edge 2     Flow = 0
  Edge 3     Flow = 2
  Edge 4     Flow = 0
  Edge 5     Flow = 2

TOTAL FLOWS, CONGESTIONS and LENGTHS
  Edge 1     Flow= 0      Congestion=0.0
  Edge 2     Flow=10      Congestion=1.66667
  Edge 3     Flow=2       Congestion=0.2
  Edge 4     Flow=0       Congestion=0.0
  Edge 5     Flow=12      Congestion=1.5
```

Figure 24: Unsplittable Flow Output

We can see that the path to route SLA1 and SLA2 is clearly outlined in the output result, along with the total flow on each edge, as well as the edge utilization on each edge. The latter data makes it easier and much faster to do the upgrading or downgrading step later.

5.5 Admission in MultiUtility

The admission process in MultiUtility includes the initial inputs and the upgrading/downgrading process.

5.6 The Output File

The output file of the MultiUtility implementation consists of the following

- A list of pre-admitted SLAs
- The Status of the SLAs after admission (admitted or not admitted)
- The Level each SLA is admitted at
- The bandwidth, latency and price information after admission for each SLA
- Flow information for each SLA
- Total flow information and edge utilization for the whole network

Figure 25 shows a sample output for the problem illustrated in Figure 19.

```
SLA 1
Status: Admitted
Admitted Level: 2
Admitted Bandwidth: 6
Admitted Latency: 3
Admitted Price: 4

SLA 1 Flow Information
  Edge 1    Flow = 0
  Edge 2    Flow = 6
  Edge 3    Flow = 0
  Edge 4    Flow = 0
  Edge 5    Flow = 6

SLA 2
Status: Admitted
Admitted Level: 3
Admitted Bandwidth: 2
Admitted Latency: 2
Admitted Price: 10

SLA 2 Flow Information
  Edge 1    Flow = 0
  Edge 2    Flow = 0
  Edge 3    Flow = 2
  Edge 4    Flow = 0
  Edge 5    Flow = 2

TOTAL FLOWS, AND EDGE UTILIZATION
  Edge 1    Flow= 0     Congestion=0.0
  Edge 2    Flow=6      Congestion=1.0
  Edge 3    Flow=2      Congestion=0.2
  Edge 4    Flow=0      Congestion=0.0
  Edge 5    Flow=8      Congestion=1.0
```

Figure 25: Output File Data for MultiUtility

6 Analysis, Experimentation and Results

In this Chapter we report the results of running our simulation on some sample data, and compare our results to that of the Watson [9] and Khan [5] implementations, with respect to both speed and optimality[31]. We also compare the performance of the Unsplittable Flow algorithm [11] to several existing path-finding algorithms. Finally we draw some conclusions.

6.1 Performance Tests

In this Section we first describe several small tests, performed to demonstrate the performance of the heuristic described in Chapter 3 and 4, and of the MultiUtiliy implementation, primarily the running time of the MultiUtility implementation and the optimality of the heuristic. We then give very preliminary performance results. As for tests, we report on four simple experiments, each designed to demonstrate the execution of a different phase of the heuristic.

- First, we isolate the path-finding phase (the Unsplittable Flow implementation [11]) and compare it to Dijkstra's k-shortest path algorithm [13] in sample networks of different sizes and connectivity. The goal is to illustrate that the Unsplittable Flow implementation can find a path faster than Dijkstra's k-shortest path algorithm [13], especially in bigger networks.

- Second, we ran both MultiUtility and Khan's heuristic implementation [5] (re-implemented by Eric Gowland [7]) on an enterprise network with a small set of

[31] The tradeoff between speed and optimality is that for an optimal solution, longer runtime is required. We can choose how close to optimal we want our solution to be in this case, by setting the error parameter ε: see Section 3.2.2 for definition of the error parameter.

SLAs. The goal was to see if there are any improvements in running time over Khan's heuristic [5] in a small network. Then, we ran both MultiUtility and Khan's heuristic implementation [5] on larger networks[32]. The goal it to test our hypothesis that the MultiUtility produces a result in less running time than Khan's heuristic [5], especially in a larger network with lots of SLAs. What's more, in test four, we will show that in addition to offering shorter running time in a larger network, our heuristic doesn't appear to sacrifice accuracy.

- For the previous test, we assume that the system is empty. That is, we are assuming that there haven't been any SLAs admitted into the network, all the link utilizations are 100% from the start. In reality, when a SLA is admitted, the network will already be routed with hundreds of SLAs; therefore it is unrealistic to assume an empty system from the start. In this test, we test both heuristics on networks that are 50% utilized, 75% utilized and so on. That is, we assume that before the first example SLA is admitted, the average utilization of each link is already 50% and 75%[33]. The goal is to test how well the admission step is going to function with networks of various states.

- For the fourth test, we compare the solution obtained using MultiUtility to the truly optimal solution, obtained by using the branch and bound algorithm [4][34]. The goal is to test the optimality of our heuristic, to learn whether our heuristic can get to a reasonably close to optimal solution in less running time.

---

[32] A larger network is here defined as a network bigger than an enterprise network, with more than 30 nodes and average degree of connectivity of more than 4 per node.

[33] We achieve this by setting up example network which the average link utilization is pre-determined.

[34] The branch and bound method was invented and documented by Khan as another way of solving the Utility Model. This method requires putting all levels of all user SLAs into a branching tree and searching for the best solution; this method is slow but does yield the optimal solution.

For each experiment, we used NETGEN or RMFGEN[35] to generate random network information.

6.2 Experiment 1: Unsplittable Flow Implementation

We have tested only the Unsplittable Flow implementation [11] (by isolating only the path finding step and ignoring the admission step) on a variety of problems and compared its performance to theoretical bounds. We used two different random network generators, NETGEN and RMFGEN.

Furthermore, for large numbers of commodities, the Unsplittable Flow implementation outperforms the standard Dijkstra's algorithm [13]. We provide some data as evidence of the running time improvement in Section 6.2.3.

6.2.1 Analysis: Dependence on the Size of the Graph

As we mentioned before (in Section 3.2), as the graph gets bigger, the Unsplittable Flow algorithm [11] takes longer to find a list of paths to route all of the demands of each SLA. Overall, (path finding plus routing…) the Unsplittable Flow algorithm finds an *ε-optimal* solution in the order of

$$O(k^2(\log k + \epsilon^{-2})\log n)$$

running time, where k is the number of SLAs being routed, ε is the degree of accuracy we want the algorithm to reach (the error parameter), and n is the number of nodes in the graph.

Thus, by looking at the big O notation, as the number of nodes increases, the overall running time should increase logarithmically.

---

[35] NETGEN and RMFGEN are random network generation programs; we use the data generated by these programs as our test data.

6.2.2 Analysis: Dependencies on Number of Commodities

By looking at the order of running time, k indicates the number of commodities. That is, as the number of SLAs waiting for admission increases, the overall running time should also increase. The overall running time being $O(k^2(\log k+\epsilon^{-2})\log n)$, as k increases, the overall running time should increase roughly linearly.

6.2.3 Experiment: Comparison to Other Path Algorithms

We ran the path finding process in the MultiUtility implementation and the path finding process in Khan's heuristic [5] over 5 types of graphs, and compared the results. In the MultiUtility implementation, the path finding process used is the Unsplittable Flow implementation [11]; in Khan's heuristic [5], the path finding process used is the standard Dijkstra's K-shortest path algorithm [13]. Here is a list of the descriptions of all of the graphs that were used:

1. *100 nodes, 213 edges graph*: This is a typical randomly generated graph, and the goal is to test how each algorithm performs on normal graphs with no extreme cases.

2. *150 nodes, 325 edges graph:*

3. *200 nodes, 560 edges graph*: In Eric Gowland's thesis [7], he mentioned that the path finding process of Khan's heuristic works well with network node counts under 50 and low connectivity. He did not mention however, how well the heuristic will do with network of more than 50 nodes. Here we test both path-finding algorithms on graphs with a high number of nodes and edges and high connectivity. The goal is to test how both algorithms perform on larger graphs with high connectivity.

4. 250 nodes, 660 edges graph:

5. *300 nodes, 700 edges graph*: This is a really big network. The goal is to test the performance of both algorithms on a relatively large network.

For each of the 5 test graphs mentioned above, we submitted 6 batches user SLAs, the sizes of the SLA batch are: 100 SLAs, 300 SLAs, 500 SLAs, 700 SLAs, 900 SLAs, 1100 SLAs.

6.2.4 Experimental Results: Path Finding Algorithm

Figure 26 shows the running time for each of the experiments described above.

(nodes, edges, SLAs)	Dijkstra's Algorithm	Unsplittable Flow
(100, 213, 100)	320 ms	124 ms
(100, 213, 300)	635 ms	136 ms
(100, 213, 500)	912 ms	180 ms
(100, 213, 700)	1042 ms	240 ms
(100, 213, 900)	1347 ms	300 ms
(100, 213, 1100)	1628 ms	356 ms
(150, 325, 100)	520 ms	212 ms
(150, 325, 300)	780 ms	214 ms
(150, 325, 500)	902 ms	240 ms
(150, 325, 700)	1230 ms	310 ms
(150, 325, 900)	1550 ms	387 ms
(150, 325, 1100)	1781 ms	412 ms
(200, 560, 100)	678 ms	312 ms
(200, 560, 300)	921 ms	367 ms
(200, 560, 500)	1140 ms	410 ms
(200, 560, 700)	1459 ms	478 ms
(200, 560, 900)	1720 ms	512 ms
(200, 560, 1100)	2103 ms	535 ms
(250, 660, 100)	1090 ms	345 ms
(250, 660, 300)	1402 ms	410 ms
(250, 660, 500)	1892 ms	489 ms
(250, 660, 700)	2420 ms	523 ms
(250, 660, 900)	3529 ms	578 ms
(250, 660, 1100)	4189 ms	610 ms
(300, 700, 100)	2198 ms	378 ms
(300, 700, 300)	2890 ms	419 ms
(300, 700, 500)	3458 ms	487 ms
(300, 700, 700)	3910 ms	510 ms
(300, 700, 900)	4920 ms	578 ms
(300, 700, 1100)	5821 ms	689 ms

Figure 26: Running Time Comparison Between Path Algorithms

From the above table, we can see that for smaller networks (a network of 100 nodes), the MultiUtility implementation uses half of the running time as Khan's heuristic [5]. However, as the numbers of network nodes and edges increase, the advantage of MultiUtility is clearly shown. For network of 300 nodes (700 edges) MultiUtility uses only 0.4 seconds of the running time to find a result. Compare that to Khan's heuristic [5] which took 5 times longer to find a result in the same network, this is a huge improvement.

The above table is only for the path finding process of both MultiUtility and Khan's heuristic [5]. For the admission process, extra running time will be added for the upgrading/downgrading process.

6.3 Experiment 2: MultiUtility Versus Khan's Heuristic

The second experiment conducted was to compare both methods of solving the Utility Model as a whole. That is, we will test MultiUtility against Khan's heuristic [5], on their ability to find paths and handle admission in various types of networks and various numbers of SLA batches.

6.3.1 Analysis: the Complexity of Khan's Heuristic and MultiUtility

Gowland [7], stated that his re-implementation of Khan's heuristic has running time of $O\ (nk^2)$ with network of 30 nodes or less. However, he also stated that with networks of more than 30 nodes, the running time increases exponentially, and could be as high as $O\ (nk^3)$.

We stated in previous Section that the overall running time for MultiUtility is $O(klogn)$. Also, the running time complexity does not increase no matter how large the network is. Therefore, we predict that the MultiUtility will have a better overall running time than Khan's heuristic, and this characteristic will be especially marked

for networks with more than 50 nodes, as Khan's heuristic will have $O(nk^3)$ running time and the running time of MultiUtility will continue to be $(klogn)$.

6.3.2 Experiment: Comparison of Overall Running Time

We again tested the same three example networks as the ones that were used in Section 6.2.3. Again, they are:

1. 100 nodes, 213 edges graph: testing the performance of both heuristics (both the admission step and the path finding step) against smaller networks.

2. 150 nodes, 325 edges graph

3. 200 nodes, 560 edges graph: testing the performance of both heuristics on large networks with lots of connections per node.

4. 250 nodes, 660 edges graph

5. 300 nodes, 700 edges graph: testing the performance of both heuristics on extremely large networks.

Figure 27 shows the running time for all example networks mentioned above.

(nodes, edges, SLAs)	Khan's HEU	Multicommodity Heu
(100, 213, 100)	520 ms	234 ms
(100, 213, 300)	785 ms	354 ms
(100, 213, 500)	1022 ms	410 ms
(100, 213, 700)	1542 ms	490 ms
(100, 213, 900)	1707 ms	521 ms
(100, 213, 1100)	2019 ms	589 ms
(150, 325, 100)	689 ms	312 ms
(150, 325, 300)	981 ms	390 ms
(150, 325, 500)	1048 ms	489 ms
(150, 325, 700)	1810 ms	590 ms
(150, 325, 900)	2103 ms	670 ms

(150, 325, 1100)	2596 ms	721 ms
(200, 560, 100)	1049 ms	489 ms
(200, 560, 300)	1590 ms	543 ms
(200, 560, 500)	2496 ms	670 ms
(200, 560, 700)	3049 ms	734 ms
(200, 560, 900)	3940 ms	867 ms
(200, 560, 1100)	4859 ms	923 ms
(250, 660, 100)	2038 ms	523 ms
(250, 660, 300)	3940 ms	678 ms
(250, 660, 500)	4058 ms	893 ms
(250, 660, 700)	4890 ms	934 ms
(250, 660, 900)	5234 ms	1045 ms
(250, 660, 1100)	5930 ms	1289 ms
(300, 700, 100)	3028 ms	654 ms
(300, 700, 300)	4593 ms	780 ms
(300, 700, 500)	5038 ms	894 ms
(300, 700, 700)	6050 ms	1045 ms
(300, 700, 900)	6893 ms	1459 ms
(300, 700, 1100)	8821 ms	1689 ms

Figure 27: Overall Running Time Comparison: MultiUtility and Khan's Heuristic

6.3.3 Analysis: Comparison of Overall Running Time

From the running times listed in Figure 27, we can see that for the first network example, the network of 100 nodes and 213 edges, Khan's heuristic[5] took slightly longer than MultiUtility. That is because on smaller networks, the admission step doesn't have to spend much time looking for the appropriate upgrade in Khan's heuristic [5]; therefore, the difference in running times is not apparent.

With larger networks, such as the last network example, the running time differences are very apparent. For example, in a 300 nodes network, MultiUtility takes about 1.5 second to admit 900 customer SLAs; Khan's heuristic [5], takes about 7 seconds. The difference becomes more apparent as the number of nodes increases.

6.4 Experiment 3: Testing On Utilized Networks

The previous set of tests tested the running time of Khan's heuristic and the MultiUility when the network empty. That is, we assume that no other SLAs had been previously admitted. This is an infeasible assumption. In a real network, it's incorrect

to assume that prior to the admission of a user SLA that the system would be empty. In this test, we set up the network to have an average link utilization of 50%, and 75%. Figure 28 and 29 illustrates the running time comparison of both Khan's heuristic and the MultiUtility when we run both heuristics on 50%, and 75% utilized networks.

(nodes, edges, SLAs)	Khan's HEU	Multicommodity Heu
(100, 213, 100)	620 ms	259 ms
(100, 213, 300)	985 ms	389 ms
(100, 213, 500)	1522 ms	434 ms
(100, 213, 700)	1842 ms	523 ms
(100, 213, 900)	2307 ms	592 ms
(100, 213, 1100)	3419 ms	678 ms
(150, 325, 100)	1629 ms	356 ms
(150, 325, 300)	2981 ms	452 ms
(150, 325, 500)	3058 ms	534 ms
(150, 325, 700)	4510 ms	598 ms
(150, 325, 900)	5721 ms	721 ms
(150, 325, 1100)	6904 ms	782 ms
(200, 560, 100)	3049 ms	534 ms
(200, 560, 300)	4691 ms	673 ms
(200, 560, 500)	5603 ms	737 ms
(200, 560, 700)	6184 ms	884 ms
(200, 560, 900)	7829 ms	901 ms
(200, 560, 1100)	8104 ms	1204 ms
(250, 660, 100)	3586 ms	653 ms
(250, 660, 300)	4960 ms	728 ms
(250, 660, 500)	5604 ms	910 ms
(250, 660, 700)	6295 ms	1489 ms
(250, 660, 900)	8068 ms	1820 ms
(250, 660, 1100)	9402 ms	2289 ms
(300, 700, 100)	4067 ms	854 ms
(300, 700, 300)	5513 ms	930 ms
(300, 700, 500)	6023 ms	2943 ms
(300, 700, 700)	8593 ms	4045 ms
(300, 700, 900)	10394 ms	5423 ms
(300, 700, 1100)	15038 ms	6293 ms

Figure 28: Overall Running Time Comparison: On 50% Utilized Networks

(nodes, edges, SLAs)	Khan's HEU	Multicommodity Heu
(100, 213, 100)	820 ms	329 ms
(100, 213, 300)	1078 ms	459 ms
(100, 213, 500)	1922 ms	536 ms
(100, 213, 700)	2242 ms	627 ms
(100, 213, 900)	3021 ms	699 ms
(100, 213, 1100)	4205 ms	728 ms
(150, 325, 100)	2329 ms	455 ms
(150, 325, 300)	3181 ms	567 ms
(150, 325, 500)	4258 ms	627 ms
(150, 325, 700)	5312 ms	748 ms

(150, 325, 900)	6723 ms	828 ms
(150, 325, 1100)	7922 ms	882 ms
(200, 560, 100)	4023 ms	699 ms
(200, 560, 300)	5921 ms	729 ms
(200, 560, 500)	7462 ms	888 ms
(200, 560, 700)	8293 ms	1023 ms
(200, 560, 900)	9302 ms	1124 ms
(200, 560, 1100)	15822 ms	1344 ms
(250, 660, 100)	6532 ms	777 ms
(250, 660, 300)	7284 ms	882 ms
(250, 660, 500)	8304 ms	917 ms
(250, 660, 700)	10425 ms	1024 ms
(250, 660, 900)	15023 ms	1203 ms
(250, 660, 1100)	18294 ms	1344 ms
(300, 700, 100)	6728 ms	891 ms
(300, 700, 300)	9213 ms	1032 ms
(300, 700, 500)	14523 ms	3204 ms
(300, 700, 700)	1993 ms	5923 ms
(300, 700, 900)	25394 ms	6273 ms
(300, 700, 1100)	30380 ms	7888 ms

Figure 29: Overall Running Time Comparison: On 75% Utilized Networks

From Figure 9 we can see that when the network is 50% and 7% utilized, Khan's HEU takes even longer to sort through all the path choices and admit all the SLAs. This characteristic was especially shown as the network becomes more utilized. For example, for network of (300, 700, 100), when the network is 0% utilized, the running time is 3028; when the network is 75% utilized, the running time is 4067. Because our algorithm only traverse through edge utilizations and edge capacities already stored in vectors, there isn't much change in running time between when the network is 0%, when the network is 50% utilized, or when the network is 75% utilized.

6.5 Experiment 4: Solution Optimality Test

We tested the solution produced by MultiUtility against the solution from the branch and bound method [4]. The goal is to judge the optimality of our solution. In the

example we used, there are three levels for each SLA. The percentage of optimality for an SLA admitted by MultiUtility is compared to the global optimum, as produced by the branch and bound algorithm.

However, we need to map a mix of SLAs admitted at various QoS levels onto utility (revenue). For our tests we assumed the following mapping. We assume that the revenue generated by an SLA is $1 at Gold, $0.66 at Silver and 0.33 at Bronze. Figure 30 lists all the combinations that can be reached with solution reached by the Branch and Bound algorithm and by MultiUtility, as well as the percentage of optimality attached to those combinations.

Solution Reached By Branch and Bound	Solution Reached By MultiUtility	Optimality
Bronze Silver Gold	Bronze	100% 66% 30%
Silver Gold	Silver	100% 66%
Gold	Gold	100%

Figure 30: Optimality Test Rules: Percentage of Optimality

Given the optimality percentage table in Figure 30, we compare our results obtained by MultiUtility to the optimal solution solved by the branch and bound method [4]. Two types of tests were performed.

1. For the first test, we used a randomly- generated 100 nodes network and admitted 20 different SLAs, ranging from a light 30 SLA batch, to a heavy, 600 SLAs batch. The network was randomly generated using NETGEN. The goal is to test the optimality of MultiUtility as the number of customer SLAs increases. As the customer SLAs increase, the network remains unchanged.

2. For the second test, we admitted 100 customer SLAs on 20 different networks, ranging from 30 nodes to 600 nodes. The goal of this test is to test the optimality of MultiUtility as the size of the graph increases, thus getting a better idea of the performance of MultiUtility in a larger network.

Figures 31 and 32 illustrate the result of the optimality tests mentioned above.

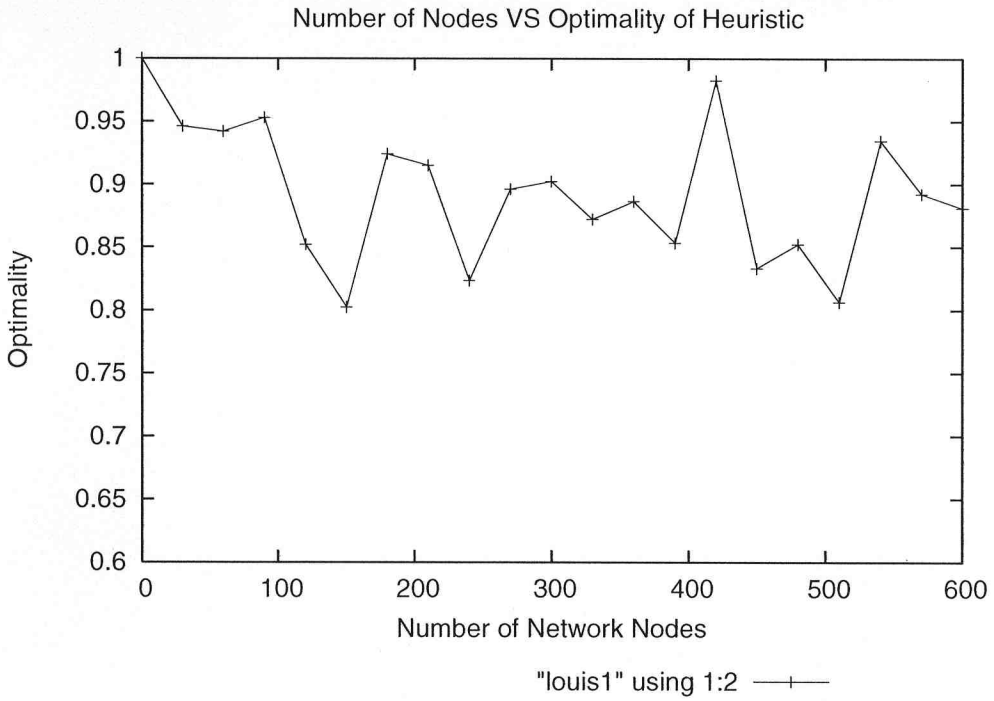


Figure 31: A Chart Showing the Result of the Optimality Test 1

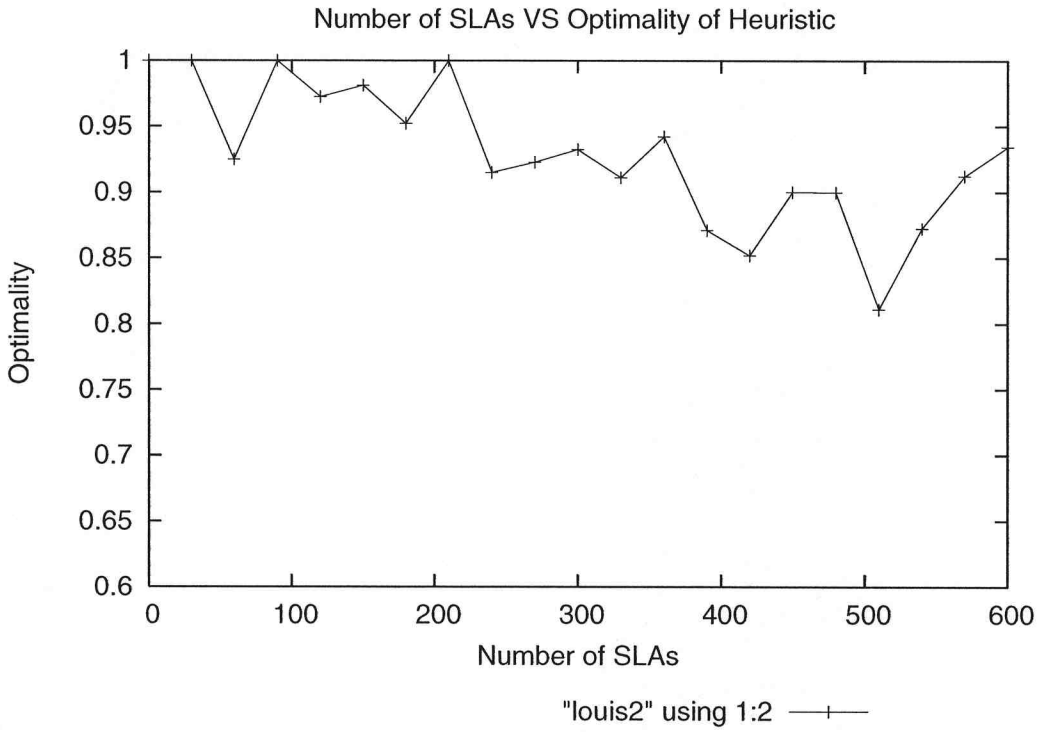


Figure 32: A Chart Showing the Result of the Optimality Test 2

From the Figure we can reach the following conclusions:

1. As we can see from figure 10, for test 1: optimality gradually decreases at first, then it came back up. There isn't a specific trend overall. As the network node increases, the optimality sometime drops, and then comes back up. The overall optimality is around 90%

2. The same can be said about test 2, as the number of SLAs admitted increases, the overall optimality is 90%.

3. An explanation is that the decreasing and increasing of optimality is due to the randomness of the network and SLA data. For example, from figure above, we can tell that when we submit 100 SLAs to a 200 nodes network, the optimality decreases; however that doesn't mean that for every network of 200 nodes, the optimality will decease, it just means that particular randomly generated 200 nodes network causes decrease of optimality.

In conclusion, our heuristic is very close to optimal, and it has a very fast running time, especially in larger networks with more than 50 nodes.

Conclusions and Future Work

Here we discuss the conclusions that can be drawn from this work, along with the major contributions made. We also introduce some topics related to the Multicommodity Flow algorithm [6] and the Utility Model based Network Admission Controller [7, 9], which both deserve further research.

6.6 Conclusions

In the first two Chapters, we looked at Khan's heuristic [5] for optimal admission of customers to a server while fully respecting QOS constraints. We confirmed that the prototype implemented for Khan's heuristic [5] is only suitable for use in enterprise networks– defined here as networks of less than 50 nodes - due to the poor path finding algorithm. What's more, the admission step of Khan's heuristic [5] goes through lots of iterations per SLA to find the right path for each upgrading; some of the overall running time are wasted in the unnecessary iteration and path matching during upgrading.

In making improvements to Khan's heuristic [5]; we wanted to keep the efficient, i.e. close to optimal, solutions which Khan's heuristic [5] was able to produce. At the same time, however, we wanted to improve the path finding algorithm used, making it faster and more accurate and able to treat networks bigger than an enterprise network[36]. We also wanted to eliminate the upgrading iteration for each of the SLAs: It is much easier if we keep information such as edge utilization and total flow carried on each edge in a vector. This way, we could handle the upgrading in three easy steps by

---

[36] Here. A network bigger than an enterprise network is defined as a network of more than 30 nodes.

1. Looking up the information in a vector,

2. Determining if there is enough surpluses for upgrading,

3. Upgrading.

By using concepts from the Multicommodity Flow Problem [6], a classic problem from combinatorial mathematics, we were able to map parts of the Utility Model problem into a Concurrent Flow problem [6], solving the Concurrent Flow problem using a modification of a approximation algorithm first invented by Leighton et al [15].

As a result, our new and improved heuristic is able to perform a fast and efficient path search on a network of a large size (more than 50 nodes); while the efficiency of the heuristic remained about the same as Khan's [5]. Also, the overall running time was cut significantly[37] due to the extra information we keep in a vector for upgrading purposes.

6.7 Further Work

6.7.1 Application Models

In this work, a new concept was presented to solve the Utility Model. However, the work done in this thesis was still very conceptual and focused on the path finding aspect. For the MultiUtility model to be used in engineering practice, a lot more work is necessary.

---

[37] In some cases, the overall running time of MultiUtility was reduced to 1/5 of the time as Khan's heuristic, see Figure 26

For example, Figure 31 illustrated the graph and SLA interface for Khan's heuristic implementation [5]. For the purpose of testing different types of data, we only used a simple input and output file with numeric data as our interface. Also a lot of network parameters such as queuing delays are ignored.

To adapt our work to admission control in real networks, we need to develop an interface, network and SLA environment similar to Khan's implementation. We also need to study more networks which represent practical design, as demonstrated in Gowland's thesis [7].

6.7.2 Proper Network Simulations

To adopt the MultiUtility algorithm to practice, we need proper network simulation. That is, the experiments done in Chapter 6 are only to test the running time and efficiency of the MultiUtility algorithm. To adopt the algorithm to a real life network environment, we need to test it against a model of a real network. For example, in Gowland's thesis [7], he re-implemented Khan's heuristic[5] in C and did proper testing using a network topology based on MichNet, an actual backbone network installation in the State of Michigan. Bandwidths and Latency tested on Gowland's implementation [7] are based upon those found on a map of this topology.

It would be useful to conduct such proper testing and simulation with the MultiUtility heuristic.

6.7.3 Multicast Resource Allocation

A potential criticism of the MultiUtility design and implementation presented in this work is its use of a single source and destination specification for each SLA. Depending upon the application, it may be far more efficient to provide a multiple-destination or even multiple-source model.

This concept ties in closely with multicasting [1], which has been an area of substantial research. Modification of the existing model to include such capabilities would involve incorporating research on efficient multicast tree construction and tying it in with the Multicommodity Flow algorithm. Also, there are variations of Multicommodity flow problems that involve multicasting, for example, the algorithm by Garg and Konemann [22]. Multicommodity flow algorithms might be a very promising way to solve the multicasting version of the Utility Model.

Bibliography

[1] A Borodin & R El-Yaniv, "Online Computation & Competitive Analysis", *Cambridge Univ Press*, 1998, ISBN 0-521-56392-5.

[2] E. Chang & A. Zakhor, "Cost Analysis for VBR video servers", *IEEE Multimedia*, Winter 1996, pp 56-71.

[3] P. Klein, S. A. Plotkin, C. Stein, and E Tardos, "Fast Approximation Algorithmsfor the Concurrent Flow Problem with Applications to Routing and Finding Sparse Cuts", *Proceeding of the 22nd Annual ACM Symposium on Theory of Computing*, 1990, pp 310 – 321.

[4] S. Khan, E. Manning & K. F. Li, "The Utility Model for Adaptive Multimedia Systems", *Proceeding of the International Conference on Multimedia Modeling (MMM 97)*, Singapore, Jan. 1998. World Scientific Publishing Ltd., ISBN 981 - 02 - 3351 - 5, pp 111 - 126.

[5] S. Khan, " Quality Adaptation in a Multisession Multimedia System: Model, Algorithms and Architecture", Ph D thesis, ECE, University of Victoria,1998.

[6] A.A.Assad, "Multicommodity Network Flows – a Survey", *Networks*, 8:37 –91, 1978

[7] Eric Gowland, "Performance Evaluation of an Utility Model Based Network Admission Controller", M.sc. thesis, ECE, University of Victoria, 2004.

[8] Y. Toyoda, "A Simplified Algorithm for Obtaining Approximate Solution to Zero-one Programming Problems", *Management Science*, 21:1417–1427, 1975.

[9] R. K. Watson, "Packet Networks Networks and Optimal Admission and Upgrade of Service Level Agreements: Applying the Utility Model", M.Sc thesis, ECE, University of Victoria, 2001.

[10] M.M.Akbar, "The distributed Utility Model Applied to Optimal Admissoin Control & QoS Adaptation in Multimedia System and Enterprise Networks," Ph.D. thesis, Department of Computer Science, University of Victoria, 2002.

[11] S. G. Kolliopoulos, C. Stein, " Improved Approximation Algorithm for Unsplittable Flow Problems", *Proceeding of the 38th Annual Symposium on Foundation of Computer Science (FOCS '97)*, pp 426 – 430, 1997

[12] S. Sahni, "Approximate Algorithms for the 0/1 Knapsack Problem", *Journal of the ACM (JACM)*, Volume 22, Issue 1 (January 1975), pp 115 – 124, 1975

[13] D.B.Johnson, "A Note on Dijkstra's Shortest Path Algorithm", *Journal of ACM (JACM)*, Volume 20, Issue 3 (July 1973), pp 385 - 388, 1973

[14] S. Kapoor and P.M.Vaidya, "Fast Algorithm for Convex Quadratic Programming and Multicommodity Flows", *Proceeding of the 18th Annual ACM Symposium on Theory of Computing*, pp 147 – 159, 1986

[15] T. Leighton, F. Makedon, S. Plotkin, C. Stein, E. Tardos, and S Tragondas, "Fast Approximation Algorithms For Multicommodity Flow Programing", *Proceeding of the 23rd Annual ACM Symposium on Theory of Computing*, pp 101 –111, 1991

[16] E. Chang & A. Zakhor, "Cost Analysis for VBR video servers", *IEEE Multimedia*, Winter 1996, pp 56-71.

[17] I.R. Chen & T.H.Tsui, "Performance Analysis of Admission Control Algorithms based on Reward Optimization", *Performance Evaluation*, Vol 33, No 2, pp 89 – 112.

[18] L. Chen, K. F. Li, S. Khan, Eric Manning, "Parallel & Distributed Processing", *Lecture Notes in Computer Science* 1586, Springer Verlag, pp 289-298, ISBN 3-540-65831-9.

[19] A Kamath, O Palmon & S. Plotkin, "Routing & Admission Control in General Topology Networks with Poisson Arrivals", *Technical Report, STAN-CSTR-95-1548*, 1995.

[20] X. Xiao and L. M. Ni, "Internet QoS: A big picture", *IEEE Network*, 13(2), pp 8 - 18, Mar 1999.

[21] D.P.Bertsekas and P.Teng, "RELAXT-III: A new and improved version of the RELAX code", *Technical Report LIDS-P-1990, MIT*, July 1990.

[22] N. Garg and J. Konemann, Faster and simpler algorithms for multicommodity flow and other fractional packing problems, *39th Annual IEEE Symposium on Foundations of Computer Science*, pp 300–309, 1998