The Traveling Miser Problem

David Breitgand, Danny Raz, Yuval Shavitt

***Abstract*— Various monitoring and performance evaluation tools generate considerable amount of low priority traffic. This information is not always needed in real time and often can be delayed by the network without hurting functionality. This paper proposes a new framework to handle this low priority, but resource consuming traffic in such a way that it incurs a minimal interference with the higher priority traffic. Consequently, this improves the network goodput. The key idea is allowing the network nodes to delay data by locally storing it. This can be done, for example, in the *Active Network* paradigm.**

In this paper we show that such a model can improve the network's goodput dramatically even if a very simple scheduling algorithm for intermediate parking is used. The parking imposes additional load on the intermediate nodes. To obtain minimal cost schedules we define an optimization problem called the *traveling miser problem*.

We concentrate on the *on-line* version of the problem for a predefined route, and develop a number of enhanced scheduling strategies. We study their characteristics under different assumptions on the environment through a rigorous simulation study.

We prove that if only one link can be congested, then our scheduling algorithm is $O(\log_2 B)$ competitive, where B is congestion time, and is 3-competitive, if additional signaling is allowed.

***Index Terms*— active networks, on-line algorithms, competitive analysis, network management.**

I. Introduction and Motivation

Off-line management applications, such as various logging facilities that subsequently transfer the accumulated data over the network, software distribution facilities, distributed backups, accounting and billing, long term history monitoring of the large-scale systems generate considerable amount of traffic. This traffic utilizes the same network resources as regular user traffic, and therefore, may affect the network goodput. While in on-line management applications, transmission of management data may have stringent timing requirements, in the off-line applications, there exists a considerable freedom regarding the exact transmission time. In this paper, we focus on handling the traffic produced by the off-line management applications.

Since in most cases the end-users are not directly interested in the services described above, the impact of the administrative traffic on the user-visible network services should be minimized. For instance, it would be beneficial to preempt some management traffic at times of network congestion and schedule it for later transmission when the congestion abate.

David Breitgand is with IBM Haifa Research Lab, Israel. Email: davidbr@ibm.il.com

Danny Raz is with the Computer Science Department, Technion, Haifa, Israel. Email: danny@cs.technion.ac.il. His work was supported, in part, by the fund for the promotion of the research at the technion.

Yuval Shavitt is with the School of Electrical Engineering, Tel-Aviv University, Tel-Aviv, Israel. Email: shavitt@eng.tau.ac.il

This work was done in part when the authors were at Bell Labs, Lucent Technologies, Holmdel, NJ, USA.

Thus, it is useful to differentiate between the higher priority user-visible traffic and the lower-priority management traffic. It is important to stress, however, that in this work, the terms *high priority* and *low priority* refer to the same best effort traffic being differentiated solely by the timing constraints. This is different from the service levels that are defined in Differentiated Services Architecture [1].

The user traffic, *e.g.*, HTTP packets, has to arrive at its destination within a short period of time (2-3 seconds for the HTTP example) while the low priority management traffic can be delayed much longer. It is important to note that both the high priority and the low priority traffic compete for the same limited amount of network resources.

Given this model, it is our objective to improve the goodput of the network by preferring the high priority (user) traffic over the low priority (management) traffic at times of high load, if timing constraints of the management traffic are flexible enough to admit extra delays.

In other words, if there are plenty of resources in the network there is no difference between the low and high priority traffic. However, when the resources become scarce, the user traffic is given a priority over the management traffic wherever possible. In this work, the resource that interests us is bandwidth.

One may notice that this approach is applicable also beyond the network management domain. For example, exactly for the same reasons as above, it may be beneficial to detain large e-mail messages in the network at times of congestion, to improve the overall goodput. Similarly, in peer-to-peer overlay networks it may be worthy to use intermediate peers in order to improve the overall efficiency of the data transfer. However, in order to focus the discussion, we will refer to the network management traffic as our primary motivation throughout the paper, and discuss other usages of the proposed framework in the concluding remarks. It is important to stress that our solution pertains to the application level.

We characterize every management message[1] originating at a management agent, the *source*, by a single parameter: *deadline*. This parameter defines the latest time by which the message should arrive at the manager station, the *destination*. In this work we are not concerned with a specific way the management data is obtained. In other words, we deal neither with the specific measurement techniques, nor with the data semantics. We simply view management applications as producers of the traffic that is a subject to the deadline constraint.

Based on this constraint, we attempt to create an individ-

[1]We use the term *message* to refer to the application-level messages. An application-level message is an individual piece of management information that should be delivered in its completeness to a specific destination to be useful. Each application-level message may be fragmented into more than one smaller *packets* by the underlying packet-switched network. This fragmentation is transparent to our algorithms.

ual *itinerary* for every low-priority application-level message, along the existing routing path between the source and the destination in such a way that the message would meet its deadline, and at the same time would incur minimal additional load on the routing elements and links along the path. There is no premium for arriving at the destination earlier than specified by the deadline. This goal can be achieved by preempting messages at internal nodes during transient congestion conditions.

One framework in which this is possible is the *active networks* paradigm. This paradigm allows to equip each message with an "autonomous intelligence" enabling the messages to react to the network conditions much like the rational car drivers do when they react to the road traffic conditions. In particular, an active message is capable of detaining itself in the network in case of unfavorable conditions, scheduling its transition for a later time. Storing a message in the network using the switching elements themselves is, obviously, infeasible. Fortunately, this need not to be the case since active messages may be diverted to a special purpose machine that is decoupled from the switching element, *e.g.*, the *active engine* in [2].

Figure 1 shows a very simple algorithm that can be used by messages in active network to achieve the above objectives.

```
while next link is congested
      wait for time interval t and check again
proceed to next hop.
```

Fig. 1. Simple Algorithm using the Intermediate Parking Ability

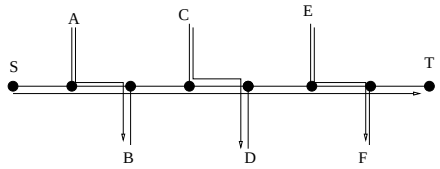

Fig. 2. A simple cross traffic scenario

A link is considered congested when its utilization crosses some threshold value. However, it is important to notice that *different* threshold values may be configured for low-priority and high priority traffic. Thus, these administratively defined threshold values can serve as a network management mechanism that forces the low priority traffic to "give way" to the high priority traffic when the competition over the resources becomes acute. When this mechanism for prioritizing the traffic is used, an additional advantage of the active network paradigm would be a higher reliability for the low-priority traffic.

Consider the scenario described in Figure 2. The source S is trying to send some management information to the target T. There is cross-traffic between A and B, and E and F respectively, which alternates, *i.e.*, whenever the cross traffic from A to B stops, the traffic from E to F starts. If the cross-traffic is intensive enough to raise the utilization level of the respective links above the low priority threshold, all attempts of S to send the information to T using regular means will fail[2] as the packets will experience loss, and no retransmissions will help.

In contrast, in our simple algorithm, the message will just park at the intermediate nodes waiting for the cross traffic from E to F or A to B to abate, and then it will continue towards T. If delays introduced by waiting are not too large, the low-priority information will arrive in time to be useful for the management application, and the user traffic (that is responsible for transient congestion) is unaffected. This simple example shows that, indeed, the extra functionality introduced by intermediate parking, adds an additional value to the network.

It is reasonable to assume that the timing constraints of the management traffic will permit practical usage of the above scheme because while the round trip time is typically less than a second, the typical management traffic deadlines for non-real time management applications is, at least, in the range of minutes. This provides the required maneuvering space for the low priority messages.

However, since parking in the network consumes valuable resources of the hosting nodes, *e.g.*, memory, our scheme may introduce considerable overhead on the infrastructure. In order to prevent an excessive load on the hosting nodes, we associate some time-dependent *cost* with parking at a given node, seeking to optimize packet parking schedules in terms of these costs.

We formulate the problem of creating the low-cost parking schedules for the low-priority traffic in abstract terms as a graph theoretic problem and study its different variants. We call this problem the *traveling miser problem* because the behavior of packets resembles a strategy of a savvy traveler that is allowed to make stops in hotels offering different time-dependent prices along some route, trying to keep the total cost of the trip as low as possible while still being timely.

We distinguish between the *on-line* variant, in which decisions should be made based on the partial information available locally at a given time, and the *off-line* variant in which all information about the network's behavior is known in advance.

Although solutions for both variants of the problem are relevant to the management applications, the on-line variant is of greater practical interest. For the off-line variant we explain its connection with other work done on the minimal weight path problem in time-dependent networks. For the on-line variant we present a scheduling algorithm called *Miser*. In addition to minimizing the interference of the management traffic with the user traffic (which is already achieved by the simple strategy above), this algorithm also minimizes the overhead on the nodes that contribute their local resources to enable intermediate parking. We prove that for the restricted case, in which only one link per route is congested at a time, the cost that our algorithm incurs on the active nodes is logarithmically competitive. We further demonstrate our algorithm usefulness by studying its performance under other, more complex, network conditions, and comparing it to to the simple strategy of Fig. 1.

We start with a simple model, that was introduced in [3]. In this model neither transmission costs, nor parking costs depend on the actual size of the low-priority message. Then, we study a model in which the parking and transmission costs linearly

[2] *E.g.*, due to an application/transport level time out, after too many retransmission attempts.

depend on the message size. It turns out that in this model, it is beneficial for a large management message to partition itself into the bulky main part, and a small "scout" part. This heuristics is called the *Scout* algorithm. The Scout algorithm is, at least, as good as the Miser algorithm, and under some conditions (*e.g.*, in case of an expensive network core) may result in schedules that are cheaper than those produced by the Miser algorithm by a factor of 2.

If we further enhance the basic model, and require the network nodes to emit small *trap* messages to signal (over the reverse forwarding path) that a congestion condition at the corresponding switching element is over, a 3-competitive scheduling strategy that we call *Trap* becomes possible.

The main contributions of this work are as follows.

- We study a novel framework for handling the lower priority, "overhead" traffic. This is in contrast to the usual case when the higher priority traffic is in the center of the problem. The proposed framework may be applicable in contexts different from the management, *e.g.*, in e-mail protocols, and peer-to-peer networks.
- We show that additional internal node capabilities (such as in the Active Network paradigm) indeed increase the network functionality.
- We formally define a relevant optimization problem. We study the on-line version of the problem, and present simple, yet efficient on-line algorithms for solving it. For the restricted, but practical case in which there is a single link congestion per path, we prove that our algorithm is $O(\log B)$ competitive, where B is the congestion duration.
- We evaluate our algorithms under realistic scenarios, and identify through an extensive simulation study the most significant network parameters affecting their performance.

The rest of the paper is organized as follows. In the next section we formally define the basic model. In Section III we formulate the problem, and then in Section IV we describe and analyze the on-line and off-line algorithms. In Section V we present the results of our simulation study for the basic model. In Section VI we extend our model to accommodate parking costs that linearly depend on the message size. Section VII discusses the on-line strategies in this extended model, and Section VII-A presents the simulation study of the new strategies. We survey the related work in Section VIII, and provide concluding remarks in Section IX.

II. Model

The motivation of the model presented in this section is to allow reasoning about the load of intermediate nodes and the cost associated with it. This model enables the definition and analysis of an optimization heuristics (see Section IV) that lowers the overhead imposed on the network nodes that accommodate the detained low-priority traffic. In Section VI we study an enhanced model in which the parking costs linearly depend on the size of the low-priority message.

Following [4] we consider a bi-directional *time-dependent* network $G = (V,\ E,\ W,\ P,\ L)$ with $V = \{1, 2, ..., n\}$ being a set of nodes, $E \subseteq V \times V$ being a set of links, and W, P, L being a set of *link weights*, *node parking weight densities*, and *link delays* respectively. $W = \{w_{i,j}(t) \mid (i,j) \in E\}$ are sets of time-dependent functions that represents the non-negative cost of traversing link (i,j) at departure time t. For simplicity, we assume that time is discrete.

$P = \{p_i(t) \mid i \in V\}$ are sets of time-dependent functions representing the non-negative cost of spending one time unit at node $i \in V$ at time instant t. If a message arrives at node i at time t_1 and departs from the node at time t_2 where $0 \leq t_1 \leq t_2 < \infty$, then the *parking weight* is $P_i(t_1, t_2) \stackrel{\text{def}}{=} \sum_{t=t_1}^{t_2-1} p_i(t)$.

The time it takes to traverse link (i,j) is called *link delay* and is denoted by $d_{i,j}$. For simplicity we assume a constant delay of 1 time unit for every link.

Let R be a simple path from s to d, $R = \langle s \equiv v_0, v_1, ..., v_{n-1} \equiv d \rangle$. The *Traveling Miser Problem (TMP)* is defined (informally) as follows. A miser starts at node s at time 0. The miser is required to reach the destination node d within the integer number $\mathcal{D}$ of time units, $0 \leq \mathcal{D} < \infty$, called *deadline*. At any given time instant t, the miser is at some node $v(t)$, and it can either park there for some time, or travel one of the outgoing links to one of the neighboring nodes.

We assume existence of a network level routing protocol, *e.g.*, IP routing protocol, such that there is only one relevant routing path from s to d. This path is not known to our application level protocol. However, when active message executes, the network routing is augmented by the application level routing decisions. Following the observations of [5] that over 85% of the AS links are symmetric and that the asymmetry is due to a small fraction of end-points, we assume, for the analysis sake, that the routes are symmetric.

Specifically, at any given time instant t, the miser is at some node $v_j \in R$, facing the following options for the next time unit:

- stay at the node v_j and pay $p_j(t)$ for parking;
- travel one link in the forward direction, *i.e.*, towards the destination, and pay $p_j(t) + w_{j,j+1}(t)$ for commuting;
- travel one link in the reverse direction, *i.e.*, towards the source, and pay $p_j(t) + w_{j,j-1}(t)$ for commuting.

The traveling miser problem is to devise a strategy that allows reaching the destination within $\mathcal{D}$ time units moving along a given path in either direction, and paying the minimal total cost for both commuting and parking. For convenience, we assume a fictitious self-link for every node in the path. The delay of this link is 1 (similarly to all other links), and its weight is 0. Traversing such a self-link corresponds to parking at the node for one time unit.

In order to state the problem more formally, we need the following definitions.

Definition 1: **Itinerary**:

For a given $s, d \in R$ an ***itinerary*** $\mathcal{I}(s,d) \stackrel{\text{def}}{=} \langle s \equiv v(0), v(1), v(2), ..., v(T) \equiv d \rangle$ such that

- $\forall i \in [0, T]\ \ v(i) \in R$;
- $\forall v(i-1), v(i) \in \mathcal{I}(s,d)\ \ (v(i-1), v(i)) \in R$, or $v(i-1) \equiv v(i)$.

Thus, $\mathcal{I}(s,d)$ is a, possibly non-simple, path from source s to destination d consisting of all the original path links plus the

fictitious self-links as described above. Sometimes we will use to $\mathcal{I}(s,d)$, as a shorthand notation for $\mathcal{I}(s,d)$.

The time T when the destination is reached is called the *termination time*, and the corresponding itinerary is referred to as ***T-termination itinerary***.

Definition 2: **Parking Itinerary**:
For some node $j \in R$, an itinerary that consists only of $0 \leq t < \infty$ fictitious links (j,j) is referred to as ***parking itinerary (or simply parking)*** at j for time t, and is denoted $\tilde{\mathcal{I}}(j,j)^t$.

Definition 3: **Itinerary Weight**:
For itinerary $\mathcal{I} = \langle s, v(1), ..., v(T-1), d\rangle$, *itinerary weight* is $W(\mathcal{I}) \stackrel{\text{def}}{=} \sum_{t=0}^{T-1} w_{v(t),v(t+1)}(t)$.

We are interested only in *finite weight* itineraries.

Definition 4: **Feasible Itinerary**:
An itinerary $\mathcal{I}(s,d)$, $s,d \in R$ is *feasible* for a given deadline $0 \leq \mathcal{D} < \infty$, being an integer number of time units, if and only if:

- ***The itinerary weight is finite***;
- ***The itinerary is timely***: $T \leq \mathcal{D}$.

Definition 5: **Reachable Node**:
Let M be an integer number of time units. Let R be some path in a time-dependent network G, and t be some time instance. A node $v_i \in R$ is ***reachable*** at time t within M time units from another node $v_j \in R$, $j \neq i$ if and only if there exists at least one feasible itinerary $\mathcal{I}(R')$ for the deadline M and a sub-path $R' = \langle v_j, ..., v_i\rangle$, where $R' \subseteq R$.

Definition 6: **Time Distance**:
The total delay of a shortest finite itinerary $\hat{\mathcal{I}}(i,j)$, is called ***time distance*** between the nodes i and j, and denoted $TD(i,j)$. Note that although in our case $d_i = 1$ for all i, $TD(i,j)$ may be different from $|j-i|$ due to infinite weights that some links may assume at certain time instances.

Property 1: Let $R = \langle 0, 1, 2, ..., n-1\rangle$ be a given path.

$$\forall \mathcal{I}(R):\ W(\mathcal{I}) \geq \sum_{i=0}^{n-1} \min_t \{w_{i,i+1}(t)\}.$$

This property simply states that in order to obtain the minimal weight itinerary over the given path, each link in the path should be traversed at time instance for which the weight of the link is minimal. This lower bound may be unattainable.

III. Problem Formulation

Now we can rigorously formulate the General Travelling Miser Problem (TMP) as follows.

Definition 7: **General TMP:** For a given path $R(s,d) \subseteq G$, where G is a time-dependent network, and a finite deadline $\mathcal{D} \in \mathbf{I}^+$, find a feasible itinerary $\mathcal{I}(s,d) = \langle s \equiv v(0), v(1), ..., v(T) \equiv d\rangle$ for which $W(\mathcal{I})$ is minimal.

As explained in Section VIII, the General TMP is an instance of a PSPACE-Complete problem, known as Canadian Traveler Problem. Fortunately, the general variant of TMP is of less practical importance since it corresponds to the cases when the network is highly unstable.

Therefore, in this work, we concentrate our efforts on a restricted variant of TMP, called K-block Recoverable TMP. This variation of TMP corresponds to scenarios in which the network is stable most of the time, but some links may become congested for certain time periods after which they recover. In K-block Recoverable TMP all parking weight densities are assumed to be constant in time $\forall t \leq \mathcal{D}$.

A. K-block Recoverable TMP

First, we define the notion of a *finite block.*

Definition 8: **Finite Block:**
Let (i,j) be a link in a given time-dependent path $R(s,d)$. We say that a finite block of duration $t_2 - t_1$ occurs at link (i,j) at time t_1 (***block start time***), and terminates at time $t_2 \geq t_1$ (***block termination time***) if $\forall t$: $t_1 \leq t \leq t_2$ $w_{i,j}(t) \equiv \infty$.
Note that fictitious self-links that correspond to parking at nodes are never blocked.

Given a finite deadline $\mathcal{D}$ and a time-dependent path $R(s,d)$, and a set of k finite blocks, the k-block recoverable traveling miser problem is to find a feasible itinerary $\mathcal{I}^*(s,d)$ such that $\forall \mathcal{I} \neq \mathcal{I}^*$: $W(\mathcal{I}^*) \leq W(\mathcal{I})$.

The k-block recoverable TMP is an instance of General TMP (see Definition 7), where at any given time instance weight of a link (except self-links) is either a constant, or infinity. The latter means that the link is blocked. The blocks are lifted after a finite time period.

The relation of this problem to the actual network scenarios is straightforward. The infinite weight of a link corresponds to the link congestion. The level of link utilization that should be interpreted as a congestion is controllable through the administrative means.

B. On-Line and Off-Line Problems

TMP can be formulated either as an *off-line* problem, or as an *on-line* one. In the off-line problem we assume that the input includes the values of link weight functions along the path for any given instant of time.

The off-line variant is of interest for advanced planning of the optimal transmission schedules based on the long observed, relatively static load patterns. This is useful, *e.g.*, for the traffic that is generated in regular hours of the day, and when the usual network behavior is known in advance.

In the strictly on-line variant of the problem the values of time-dependent link weight functions are not known in advance, the miser has only a local knowledge about the path, and the current value of a link weight observed by the miser for a certain link does not tell anything about its future value.

In [6], it was shown that finding an algorithm with bounded competitive ratio for the strictly on-line minimal delay path problem in time-dependent networks is P-SPACE-complete. If we treat the parking and commuting costs as time-dependent delays, and assume a very large deadline parameter for the message, then the on-line TMP becomes an instance of the on-line minimal delay time-dependent path problem.

IV. Algorithms

This section presents the algorithms for solving k-block recoverable TMP.

A. Off-Line Problem

The algorithm for finding minimal weight path in a general time-dependent network presented in [4] solves the general off-line traveling miser problem (as a special case) time for any constant deadline $0 \leq T < \infty$. Even though [4] solves the problem for the case of continuous (or piece-wise continuous) time-dependent weight and delay functions, the same algorithm may be used for our discrete case.

Since the off-line k-block recoverable TMP is a special case of the general off-line TMP, the algorithm of [4] also solves the off-line k-block recoverable TMP for any finite deadline.

It is useful to highlight some key ideas from the solution presented in [4]. The algorithm is based on the following *concatenation property* of the optimal (with respect to weight) time-dependent paths.

Lemma IV.1: **Every sub-itinerary of an optimal (minimal weight) itinerary is also optimal: Time-Dependent Path Concatenation Property**

Let $R(s,d)$ be a time-dependent path. Let $T \in \mathbf{I}^+$: $0 \leq T < \infty$ be some time instant. Let $\mathcal{I}^*(s,d) = \langle s \equiv v(t_0), v(t_1), ..., v(t_{k-1}), ..., v(t_{N-1}) \equiv d \rangle$ be a minimal weight finite T-termination itinerary of length at most N among all T-termination itineraries of length at most N. Then $\forall k,\ 0 \leq k \leq L(\mathcal{I}^*)$, itinerary $\mathcal{I}_k \stackrel{\text{def}}{=} \langle s \equiv v(t_0), v(t_1), ..., v(t_{k-1}) \rangle$ is a minimal weight finite T-termination itinerary of length at most k for destination $v(t_{k-1}) \in R$.

Proof: See [4]. ■

In practice, we are interested in feasible optimal itinerary only with respect to some finite deadline T being an integer number of time units. Thus we can find optimal T-termination, $(T-1)$-termination, $(T-2)$-termination *etc.*, itineraries with length at most l using the concatenation property stated by Lemma IV.1, and choose a minimal-weight feasible simple itinerary among them.

Indeed, the optimal time-dependent path concatenation property allows recursive constructing of the optimal t-termination itineraries for every $t \in \mathbf{I}^+ : 0 \leq t \leq T$ and for every length l.

Suppose, for a given path $R(s,k)$, $k \in R(s,d)$, $\mathcal{I}(s,k)$ is an optimal t-termination itinerary of length at most l where $0 \leq t \leq T$. Let us denote its weight by $W_k(t)$. Then we have the following recurrent relations:

$$W_k(t) = \min_{i \in \{k-1,k,k+1 \in R\}} (W_i(t - d_{i,k}) + w_{i,k}(t - d_{i,k})) \quad (1)$$

$$W_s(0) = 0 \quad (2)$$

$$\forall t:\ 0 < t \leq T, \forall i \in R : W_i(t) = \infty. \quad (3)$$

From these recurrent relations we can compute the weights for optimal t-termination itineraries for every node in R and also the minimal feasible itinerary for reaching the destination before deadline T as shown in [4].

Note that if we are given an explicit representation of the parking weight functions in advance for time D, the algorithm is polynomial. However, if the weight functions are specified in a more concise form (*i.e.*, analytically), the algorithm is pseudo-polynomial, as its running time also depends on D (deadline).

variables:
$j \in [1, \lfloor \log_2 B \rfloor]$: number of phase;
$dir \in \{FWD, BCKWD\}$: state;
v_b: currently blocked node in the forward direction;
v_i: current node;
s, d: source and destination;
v_j^*: intermediate destination of phase j;
T_j: parking time of phase j;
$\mathcal{W}$: set of known weights.

initially: $j \leftarrow 1;\ dir \leftarrow FWD;\ v_i \leftarrow s;$
$v_b \leftarrow nil;\ v_j^* \leftarrow nil;\ T_j \leftarrow 0;\ \mathcal{W} \leftarrow \emptyset$

while $v_i \neq d$
  if $dir = FWD \ \wedge \ w_{v_i,v_{i+1}} < \infty$
    $\mathcal{W} \leftarrow \mathcal{W} \bigcup \{w_{v_i,v_i}, w_{v_i,v_{i+1}}\};$
    proceed to node v_{i+1};
  else if $dir = FWD \ \wedge \ w_{v,v+1} = \infty$
    if $v_i = v_b$
      $j \leftarrow j + 1$;
    else
      $v_b \leftarrow v_i;\ j \leftarrow 1$;
    COMPUTE_ITINERARY$(j,\ v_b,\ \mathcal{W})$;
  else if $dir = BCKWD \ \wedge \ w_{v_i,v_{i-1}} < \infty \ \wedge \ v_i \neq v_j^*$
    proceed to node v_{i-1};
  else if $dir = BCKWD \ \wedge \ w_{v_i,v_{i-1}} < \infty \ \wedge \ v_i = v_j^*$
    park at node v_j^* for time T_j
  else if $dir = BCKWD \ \wedge \ w_{v_i,v_{i-1}} = \infty$
    $dir \leftarrow FWD$;
    $j \leftarrow 1$;
  upon termination time of phase j:
    $dir \leftarrow FWD$;
endwhile

compute_itinerary$(j,\ v_b,\ \mathcal{W})$
  using $\mathcal{W}$, find node v_j^* that is reachable from v_b within $t \leq 2^{j-1}$ time units, and such that v_j^* yields the minimum weight round-trip itinerary with the parking time: $T_j = max\{1, 2^{j-1} - 2 \cdot TD(v_j^*, v_b)\}$.

Fig. 3. Miser Algorithm

B. On-Line Problem (Miser Algorithm)

In this section, we present an algorithm that addresses the on-line k-recoverable TMP. We show that for the special case of $k = 1$ the presented algorithm is $5 + 4 \log_2 B$ competitive where B is the block duration. For $k > 1$ the algorithm serves as a heuristic, We study its in Section V via simulations.

The idea of the on-line algorithm is as follows. The miser advances towards the destination using the non-fictitious links only, as long as he does not reach a block. At the blocked node, the miser has two options: either to stay at the node where the block has occurred (this stay is simulated by traversing the corresponding self-link), or move backwards on the reverse forwarding path that has been discovered so far. Assuming that the miser knows an upper bound U for the block duration, a competitive strategy for the miser is to pick up the cheapest node (in terms of the total cost of getting there and back plus spending a certain amount of time at the node as explained below) situated within roughly $U/2$ distance from the current location on the way back to the source, to spend the block time there. In other words, the miser has to pick up an itinerary with the minimal weight for the block duration. The intuition behind the algorithm is that if the miser has to spend some time en route on the way to the destination due to a block, he better do this using the

cheapest itinerary. This minimizes the overall cost of the trip.

However, the miser does not know the exact duration of any block, only an upper bound, that is not necessarily tight. Thus, it is beneficial to have a strategy for finding a tighter bound. This is done by doubling the existing estimation of the block duration at each stage of the algorithm. In order to explain the algorithm in detail we need the following definition.

A simple round-trip itinerary between the two nodes $j, k \in R$ is an itinerary that is a *concatenation* of a shortest, possibly empty, itinerary that goes from node j to node k, possibly empty finite itinerary that uses only the (k, k) fictitious links, and the shortest, possibly empty, itinerary that goes from node k to node j. More formally:

Definition 9: **Simple Round Trip Itinerary:** Itinerary $\mathcal{I}_{rt}(j,k)^t \stackrel{\text{def}}{=} \hat{\mathcal{I}}(j,k)\&\tilde{\mathcal{I}}(k,k)^t\&\hat{\mathcal{I}}(k,j)$ where $\hat{\mathcal{I}}(j,k)$, and $\hat{\mathcal{I}}(k,j)$ being the shortest itineraries between j and k, and vice versa, and $\tilde{\mathcal{I}}(k,k)^t$ being a parking itinerary at node k for time t, and $\&$ being concatenation operation, is termed ***simple round trip itinerary*** with parking time t between two nodes $j, k \in R$.

Figure 3 shows the Miser algorithm. Suppose a block occurs at some node $v_b \in R$ on the link leading towards the destination. The algorithm works in stages. At stage $j \in [1, \lfloor \log_2 U \rfloor]$ the miser chooses node $v_j^* \in R$ reachable from v_b within $T \leq 2^{j-1}$ time units, such that $W(\mathcal{I}_{rt_j}(v_b, v_j^*))^{2^{j-1}-2\cdot TD(v_j^*, v_b)}$ is minimal. Then, the miser follows this minimal weight simple round trip itinerary for stage j.

In order to gain a logarithmic factor, the miser spends exponentially increasing periods of time away from the blocked node. After each such period the miser goes back to the blocked link and checks whether the block is lifted. If the block is lifted then the miser goes through, otherwise he proceeds to phase $j+1$. If the miser encounters a block while moving in the backward direction, he forgets the past iterations, switches the direction, and starts moving towards the destination.

C. Analysis

Obviously, if only one block may occur on the path before the miser's deadline expires, he will not go back and forth more than $\lfloor \log_2 B \rfloor$ times and cannot spend more than $2*B$ time units at v_j^* in any phase j, where B being the actual duration of the block. Thus, the algorithm terminates after at most $1 + log_2 B$ simple round trips, and the overall itinerary that he builds, is finite. Constructing the round trip itinerary for every phase clearly takes $O(n)$ time where n being the length of the path. This makes the running time of the strategy $O(n \cdot \log_2 B)$. As we show in Theorem 1, if $k = 1$ the Miser algorithm is logarithmically competitive.

In case of multiple blocks, *i.e.*, when $k > 1$, the Miser strategy serves as a heuristics. When the Miser algorithm computes the round-trip itinerary for stage j, it assumes that all the links that have been discovered in all previous phases will not be blocked during stage j. Naturally, this may not be true for a certain phase. It is easy to see, that when $k > 1$, the strategy is not competitive. If two blocks occur simultaneously on the two real links leading from the node (one in the forward direction for time B_1, and one in the opposite direction for time B_2), the weight of the resulting itinerary will increase by adding $B \cdot w_{v_b, v_b}$ where $B = \min\{B_1, B_2\}$. In contrast, the optimal off-line algorithm would build its itinerary in such a way that it will spend all the blocking time in the node with the minimal parking weight. Therefore, an adversary can always force the ratio between the on-line and the off-line algorithms to be the ratio between the minimal and the maximal link weights of the path. Fortunately, since blocks do not always follow the above worst case scenario in practice, the presented on-line strategy can be used as a heuristic. Its performance is studied using simulations in Section V under various assumptions on the blocks distribution.

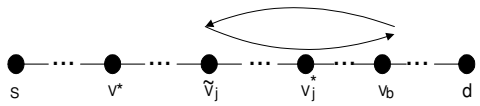

Fig. 4. Simple Round Trip w/o Parking

Theorem 1: Let $R(s,d)$ be a time-dependent path as defined in Section II. Let $\mathcal{D}$ be a finite deadline, and let U be the upper bound on the blocking time. If there exists only one block of duration B then the competitive ratio of the Miser algorithm is $5 + 4 \cdot \log_2 B$.

Proof: There are two cases that should be inspected.

Case 1: The block happens at node $v_b \in R$ at time $t_{start} > TD(s,d)$.

It is easy to see that in this case, parking in the intermediate nodes renders no gain in minimizing the total cost of the itinerary, since all parking costs are positive. The optimal algorithm just advances directly to the destination. Since the Miser algorithm behaves identically, the competitive ratio is 1.

Case 2: The block happens at node $v_b \in R$ for time $B = t^{end} - t^{start}$, $0 \leq t^{start} \leq t^{end} \leq \mathcal{D}$, and t^{start} such that $TD(s, v_{b+1}) > t^{start}$. In other words, the miser cannot reach the destination without encountering the block en route.

According to the lower bound given by Property 1, the cost of the itinerary built by the optimal algorithm is *at least* the weight of the path (when all link weights are finite):

$$W(\hat{\mathcal{I}}(s,d)) \leq W(\mathcal{I}_{opt}). \tag{4}$$

Also note that every algorithm, including the optimal one, must wait in this case at least $B + t^{start} - TD(s, v_b)$. The optimal algorithm will do this (*i.e.*, wait) at some node called v^* (see Figure 4), and therefore its parking cost is at least $(B + t^{start} - TD(s, v_b) \cdot w_{v^*, v^*}$. Thus, we have:

$$W(\hat{\mathcal{I}}(s,d)) + (B + t^{start} - TD(s, v_b)) \cdot w_{v^*, v^*} \leq W(\mathcal{I}_{opt}). \tag{5}$$

Note that since t^{start} is non-negative the following inequality holds:

$$W(\hat{\mathcal{I}}(s,d)) + (B - TD(s, v_b)) \cdot w_{v^*, v^*} \leq W(\mathcal{I}_{opt}). \tag{6}$$

For the on-line Miser algorithm we have:

$$W(\mathcal{I}_{miser}) \stackrel{\text{def}}{=} \sum_{j=1}^{\lfloor \log_2 B \rfloor + 1} [(2^j - 2 \cdot TD(v_j^*, v_b)) \cdot w_{v_j^*, v_j^*} + \quad (7)$$

$$2 \cdot W(\hat{\mathcal{I}}(v_b, v_j^*))] + W(\hat{\mathcal{I}}(s, d))$$

Let us find the upper bound for the cost of each phase j, $\forall j \in [1, \lfloor \log_2 B \rfloor]$ of the Miser algorithm. There are two cases.

Case 1: $1 \leq j \leq \lfloor \log_2 TD(s, v_b) \rfloor$.
Let $\tilde{v_j} \in [v^*, v_b]$ be the farthest node that is reachable from v_b in phase j by a round-trip itinerary without parking (see Figure 4). Since the Miser algorithm always chooses minimal weight round-trip itineraries for each phase, the cost of Miser itinerary in this phase is no greater than the cost of the round trip itinerary without parking to $\tilde{v_j}$. Therefore, for a *single* phase j:

$$W(\mathcal{I}_{miser}^j) \leq W(\mathcal{I}_{rt}(v_b, \tilde{v_j})) \leq 2 \cdot W(\hat{\mathcal{I}}(s, d)).$$

From Inequality 4 we obtain that for a single phase j $W(\mathcal{I}_{miser}^j) \leq 2 \cdot W(\mathcal{I}_{opt})$.
Case 2: $\lfloor \log_2 TD(s, v_b) \rfloor < j \leq \lfloor \log_2 B \rfloor$.

Observe that Miser *always* parks at v_j^* that belongs to the subpath $R(v^*, v_b)$ (including both ends). Let us consider a simple round trip itinerary from node v_b to node v^* with parking time $2^j - 2 \cdot TD(v_b, v^*)$ for each phase j. Since Miser chooses minimal weight itineraries for every phase, the cost of this round trip itinerary bounds the cost of phase in the Miser algorithm from above:
$W(\mathcal{I}_{miser}^j) \leq 2 \cdot W(\hat{\mathcal{I}}(v_b, v^*)) + (2^j - 2 \cdot TD(v^*, v_b)) \cdot w_{v^*, v^*}$
Since $W(\hat{\mathcal{I}}(v_b, v^*)) \leq W(\hat{\mathcal{I}}(s, d))$, we have:
$W(\mathcal{I}_{miser}^j) \leq 2 \cdot W(\hat{\mathcal{I}}(s, d)) + (2^j - 2 \cdot TD(v^*, v_b)) \cdot w_{v^*, v^*}$.

The maximal phase duration is $2 \cdot B$. Therefore, for any single phase j we have: $W(\mathcal{I}_{miser}^j) \leq 2 \cdot W(\hat{\mathcal{I}}(s, d)) + (2 \cdot B - 2 \cdot TD(s, v_b) \cdot w_{v^*, v^*} + 2 \cdot TD(s, v^*) \cdot w_{v^*, v^*}$.

Observing that $TD(s, v^*) \cdot w_{v^*, v^*} \leq W(\hat{\mathcal{I}}(s, d)) \leq W(\mathcal{I}_{opt})$, and using Inequality 6, we obtain that for a single phase j: $W(\mathcal{I}_{miser}^j) \leq 4 \cdot W(\mathcal{I}_{opt})$. Then from Equality 7 we obtain (using 4 again)

$$W(\mathcal{I}_{miser}) \leq (5 + 4 \cdot \log_2 B) \cdot W(\mathcal{I}_{opt}). \quad (8)$$

■

D. Trap over RFP Model

Can we improve the Miser's worst case performance? The logarithmic competitive ratio of Miser comes from the fact that it requires the whole message to traverse the entire path between the blocked node, and the node yielding the lowest cost round-trip itinerary of phase j of the algorithm.

If our model can be slightly extended to accommodate small signal messages, that are propagated over the reverse forwarding path each time a transient congestion condition on the previously blocked link is over, the Miser strategy can be modified as shown in Figure 5, to obtain the competitive ratio of 3. We call this modified algorithm *Trap*, to distinguish it from the original version of the Miser strategy. In our model, the primary contributor to an itinerary cost is due to parking. Trap messages do not park at the intermediate nodes. Therefore, for simplicity, we assume their cost to be 0.

```
while next link is not congested
      proceed to next hop.
if next link is congested:
      for each phase j ∈ [1, ⌊log2 B⌋] find a node
      v_j* that is reachable from v_b within
      t ≤ 2^(j-1) · T time units, where T is
      the time since the message
      departed from the source node, so that
      v_j* yeilds the minimum weight
      round-trip itinerary with the parking
      time 2^(j-1) · T − t.   Proceed to v_j*.
if by the end of phase j no trap
      message signaling end of blockage at
      v_b is received, proceed to phase j + 1.
else //a trap message arrived
      proceed towards the destination.
```

Fig. 5. Trap Algorithm using the signaling ability of intermediate active nodes.

To analyze the competitive ratio of the Trap algorithm, we first prove the following lemma.

Lemma IV.2: Let v_b be the blocked node. If in phase j, message in the Trap algorithm parks at node v_j^*, then in phase $j+1$, the node $v_{j+1}^* \in [s, v_j^*]$

Proof:

By the algorithm definition, phase $j + 1$ starts when no trap message is received from v_b at the end of phase j. Assume by contradiction that the statement of the lemma is wrong. *I.e.*, let v_{j+1}^*, the intermediate parking destination for phase $j + 1$, be located between v_j^* and the blocked node (see Figure 4). This means that the round trip itinerary yielded by v_{j+1}^* has lower cost than the one that would be yielded by v_j^*, and any other node being located between the source s and v_j^* (including both these nodes). But since the parking costs are strictly positive, and all nodes between the source and the blocked node are reachable in every phase, then node v_{j+1}^* would have been chosen as the intermediate parking destination in phase j. Thus, we have a contradiction that proves the lemma. ■

Theorem 2: In the single block case, the competitive ratio of the Trap on-line strategy is 3.

Proof: Note that according to Lemma IV.2, and by the fact that Trap builds minimal weight itinerary in every phase, the total cost of Trap for the entire block duration B is bounded from above by the cost of a simple round trip itinerary from node v_b to node v^* (that would have been chosen by the optimal algorithm) with parking time $B - 2 \cdot TD(v_b, v^*)$. Therefore the total cost of Trap is bounded from above as follows:

$W(\mathcal{I}_{trap}) \leq W(\hat{\mathcal{I}}(v_b, v^*)) + (B - TD(s, v_b)) \cdot w_{v^*, v^*} + W(\hat{\mathcal{I}}(s, v^*))$.

Noticing that $\hat{\mathcal{I}}(v_b, v^*)) \leq \hat{\mathcal{I}}(s, d))$, and $W(\hat{\mathcal{I}}(s, v^*)) \leq \hat{\mathcal{I}}(s, d))$, and using the lower bounds given by the inequalities 4, and 6, we obtain that: $W(\mathcal{I}_{trap}) \leq 3 \cdot W(\mathcal{I}_{opt})$. ■

E. Connection Oriented Transmission

The heuristics being discussed so far proposed a connectionless mode of operation. The active messages equipped with the autonomous intelligence took care of their own delivery to the specified destination. An arrival (or, alternatively, a loss) of an active message was not acknowledged by its destination. Since messages can get lost in the network, this approach applied as is may compromise reliability. Adding reliability does not imply having a connection oriented communication between the source and the destination. To add a fair level of reliability at the transport level, it is required to store copies of the active messages in the source node for possible retransmission. The destination is required to supply either a negative (or a positive) acknowledgement when a specific message is lost (or received) in order to enable retransmission and garbage collection at the source. A detailed discussion of the reliability mechanisms for the proposed algorithms is out of the focus of this work. It should be stressed, though, that reliability requires additional storage and bandwidth resources which may not always be available.

If, however, connection-oriented communication between the source and the destination is needed, and can be facilitated by the capabilities of the communicating parties, a simple active algorithm that would retransmit a message after a predefined time-out can be considered as an alternative to the proposed *Miser* and *Trap* heuristics. This algorithm may improve reliability (exept for the pathological cases like the one explained in Section I), and facilitate the shorter control loops by having the destination informing the source immediately after getting the message.

We look at one specific algorithm of this kind that we call *Reset*. In this algorithm, a message is dropped at the blocked node, and is retransmitted by the source when twice the round-trip time-out between the source and the destination elapses, and no acknowledgement for the message is received.

Performance of this algorithm serves as a helpful yardstick in our simulation studies allowing to identify the favorable and unfavorable conditions for the proposed on-line heuristics.

Note that while this algorithm can be modified to gain a logarithmic factor, by introducing the exponentially increasing retransmission timeouts similarly to *Miser*, and *Trap*, it is not competitive. The total cost of a schedule in the *Reset* algorithm depends on the parking cost of the source.

If, however, the source node is the cheapest node of the path, and there is only one block per path during the message lifetime, the exponential *Reset* algorithm saves an additive factor of $\log_2 B$ compared to *Miser*, because in *Reset* the message is simply dropped at the congestion point, and does not backtrack. At the same time, the exponential *Reset* is by the additive factor of $\log_2 B$ more expensive than *Trap* because in *Trap*, the message is "retransmitted" only once when the blocked link recovery is signaled.

V. Simulations

In this section, we present a simulation study of the on-line miser algorithm proposed in IV-B, and its variation proposed in IV-D. We refer to them simply as *Miser*, and *Trap* for brevity.

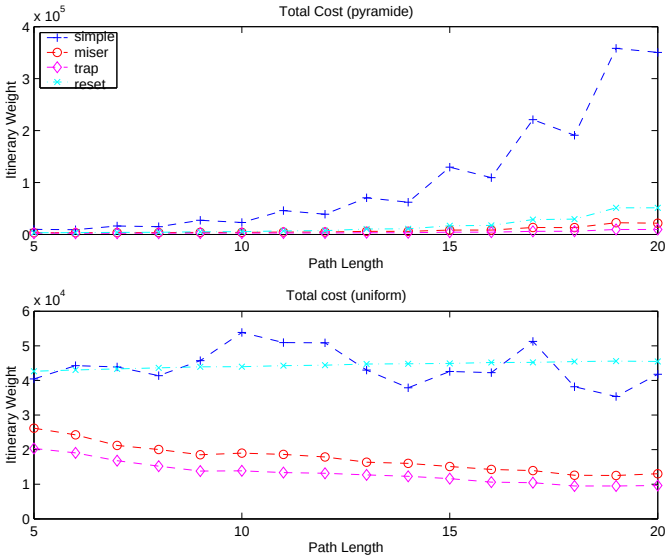

Fig. 6. Average Itinerary Weight (1 block, sample 100)

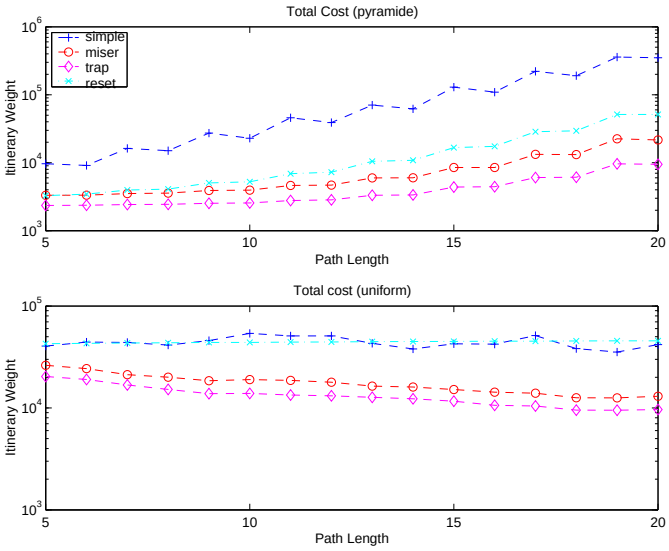

Fig. 7. Average Itinerary Weight (1 block; sample 100; semi-logarithmic Y)

Since competitive analysis provides indication only about the worst case, it is beneficial to study the actual performance of the active algorithms through simulation comparing them to other alternatives. We compare *Miser*, and *Trap* against each other, and also against the simple active algorithm explained in Section I (see Figure 1), referring to the latter as *Simple*, and the simple connection-like *Reset* algorithm of Section IV-E.

The overhead imposed on active nodes is quantified by the itinerary weight, *i.e.*, by the cost of the respective itineraries built by the algorithms (see Definition 3). In addition, we are interested in the propagation time of a packet through the network.

Our simulations consists of two sets. Section V-A investigates the behavior of the algorithms in a limited case when only one block may occur along the path during the specified deadline (*i.e.*, for 1-block recoverable TMP). Section V-B studies the algorithms for k-block recoverable TMP.

A. 1-*block Recoverable TMP*

We start by comparing averaged itinerary weights for *Simple*, *Reset*, Miser, and Trap algorithms when only one block may occur along the path during a specified deadline (*i.e.*, for 1-block recoverable TMP). Although each one of these algorithms works under slightly different assumptions on the model (see discussion in Sections IV-D, IV-E), showing them all in the same figure also helps understanding the relative benefits of the models.

Figure 6 represents the simulation results as the averaged itinerary weight (cost) being a function of the path length for two different cost models *pyramid*, and *uniform* respectively. The meaning of the cost models is as follows.

- ***Pyramid***: this cost model represents a deterministic distribution of parking densities to the nodes in the path simulating "cheap" edges and "expensive core". The parking weight density increases exponentially with the distance of a node from the edge, *e.g.*, $(1, 2, 4, 8, 16, 8, 4, 2, 1)$. The motivation for studying the pyramid cost model is that, usually, the load on the core switching elements increases sharply as we move deeper into the network cloud.
- ***Uniform:*** this cost model represents a random distribution of the parking weight densities to the nodes. Each cost is computed as 2^i, where i being an integer value uniformly distributed between 0 and 6.

For every cost model the graphs are obtained by averaging 100 runs of each algorithm for every path length ranging from 5 hops to 20 hops. In all the experiments the deadline was chosen to be $\mathcal{D} = 10000$ time units[3]

Every run for a given path length and a cost model is performed as follows. At the beginning of the run, we re-generate a path by allocating the parking weight densities to its nodes. Then, we randomly choose a link that will be congested. After that, we draw a random congestion duration b from $U(10, \mathcal{D}/2)$ distribution, and initiate a block at time 0 that will be removed after b time units. When this scenario is formed, we execute all algorithms on the same scenario recording their performance: itinerary weight (cost), and total transit time.

Figure 6 shows that *Miser* is always superior to *Simple*, and *Reset* for both cost models while being always inferior to *Trap* which is consistent with our analysis. The difference is larger for the *pyramid* cost model. The explanation is that the cost difference between the core and and the edges is very sharp. Therefore when in *Miser* the message is caught by a long block in the middle of the path, after a few iterations it will migrate far towards the source, dramatically decreasing its costs while the simple algorithm remain parked in a more expensive location. In case of the *uniform* model these differences in the parking costs are alleviated.

To better appreciate the difference between *Trap*, *Reset*, *Simple* and *Miser* cost model, consider Figure 7 showing the same graphs as Figure 6, but using the the logarithmic scale on the Y axis. First, consider Pyramid cost model. As one can observe, the *Reset* algorithm is only marginally better than *Miser* for shorter paths, and is about twice more expensive than *Miser* as paths become longer. The reason for this behavior is that since the blocked hops are drawn from the uniform distribution, approximately half of the blocked hops are beyond the most expensive node in the middle of the path (recall that the nodes' cost grow exponentially towards the middle, and then decrease exponentially towards the edge). Even if a blocked link is on the "other side" (*i.e.*, more than half-way towards the destination), *Reset* would still drop the message, and then retransmit it later paying all the way to the blocked node once again. *Miser*, in contrast, would not backtrack in this case, because all the nodes in the backward direction yield more expensive round-trip schedules. Thus, in approximately half of the blocks, *Miser* saves a factor of $\frac{B}{2 \cdot p}$ where B is the block duration, and p is the path length. In another half of the blocks *Miser* backtracks to the source and pays exactly the same parking price as *Reset*. The commuting price is different though. *Miser* pays $O(\log_2 B)$ for commuting while *Reset* pays $O(\frac{B}{2 \cdot p})$. Since *Miser* may over-estimate the block duration in the last phase, *Reset* may have marginal advantage when the blocks and the paths are short in approximately half of the cases.

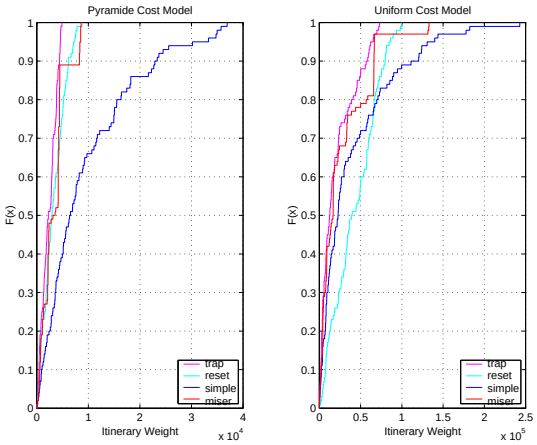

Fig. 8. CDF of Itinerary Weight (1 block; path 5; sample 100)

As expected, the *Trap* heuristics results in the best performance in both models being approximately twice cheaper on the average than *Miser* for any fixed path length. The exponential *Reset* would yield a curve that would lie in between that of *Miser*, and that of *Trap* (it is not shown in the figure).

The simulation results for the uniform cost model show the same relative performance on the average for *Miser*, *Trap*, and *Simple* while *Reset* is often inferior on the average to *Simple*. This is due to the fact that since the nodes are priced randomly, the source is not necessarily cheap.

The variability of the itinerary weights obtained in different runs is high, because in each run the weight depends on the random block's location and duration. Therefore, although the average provides a useful indication, it is more informative to characterize the results by means of *empirical cumulative distribution function (ECDF)*. Figures 8 and 9, show the ECDFs for itinerary weights for both parking cost models for path length of 5 and 15, respectively.

As one can clearly see, in the *pyramid* cost model, *Simple*

[3]The error boxes are not shown. Although variance across the tests was substantial, it is small relatively to the values of the itinerary weights. Therefore showing the error boxes is not useful. Instead, as explained later in this section, we study empirical CDFs of the itinerary weights and travel time, which is much more informative.

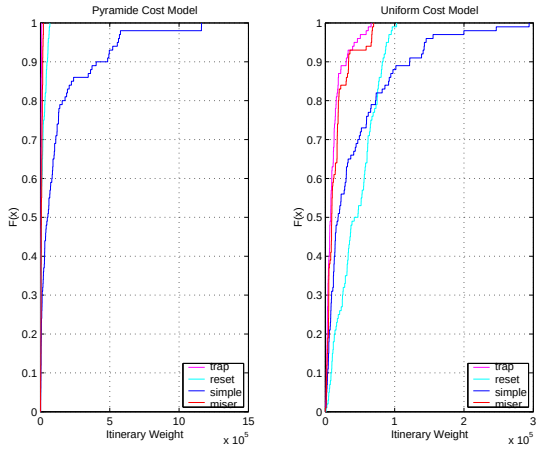

Fig. 9. CDF of Itinerary Weight (1 block; path 15; sample 100)

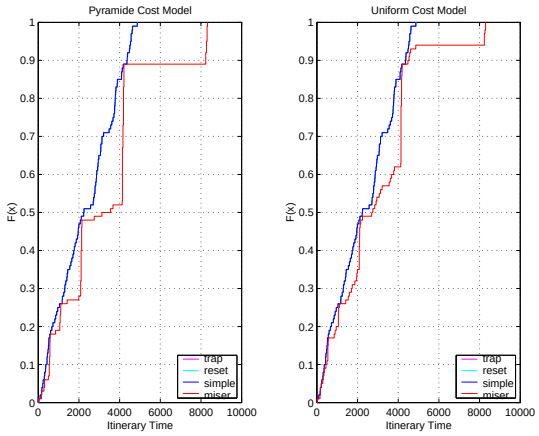

Fig. 10. CDF of Itinerary Time (1 block; path 5; sample 100)

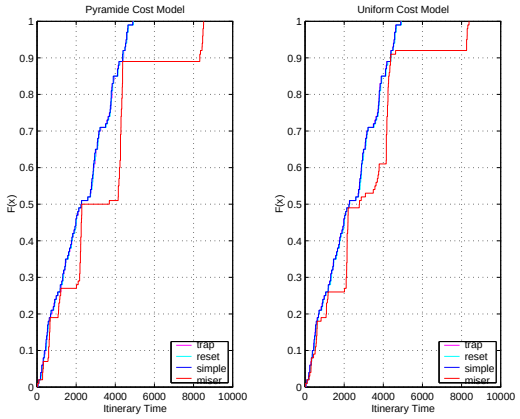

Fig. 11. CDF of Itinerary Time (1 block; path 15; sample 100)

exhibits high probability of very large itinerary weights. In this cost model, the total weight of the path increases exponentially with the path length. The reason that the graphs in Figures 8 and 9 pertaining to this cost model, appear to be shifted left as path length grows (observe the differences in the X axis scale across the graphs). This means that *Simple* would impose much higher overhead on the active infrastructure if accidentally caught by the block at an expensive (*i.e.*, loaded) active node. *Miser*, *Trap*, and *Reset*, in contrast, would smooth these outlying cases.

In case of the *uniform* cost model, *Reset*'s performance deteriorates due to the fact that the source node may have a very high parking cost. As one can see, in this model *Simple* has high probability for large outliers (heavy tail), which makes its performance worse on the average. However, *Reset* builds itineraries of larger weights more often. For instance, as Figure 9 shows, in over than 80% of the runs *Simple* builds cheaper itineraries than *Reset*.

In addition to the total cost of the itinerary we might be interested in the total time required by it. Figures 10 and 11 represent ECDFs for the total transit time of the messages for both cost models when the path length are 5 and 15. As one can see, *Simple*, *Trap*, and *Reset* spend approximately the same time for message transmission. In fact the differences in total time between these three algorithms are so small that their curves appear as a single one in the ECDF graphs.

Miser, however, pays by time for the cost optimization it achieves. This happens because in many cases *Miser* overestimates block duration (consider the last phase of the algorithm). In the *Trap*, and *Reset* algorithms, no overestimation happens. This is the reason why they behave similarly to *Simple* with respect to the total transit time. One should remember, however, that this is achieved through the slight relaxations of the model as discussed in Sections IV-D, IV-E.

Finally, it is beneficial to measure the dependency of itinerary weight on the block duration. Figure 12 depicts the cost as function of deterministically chosen block durations and *pyramid* cost model. As expected, the cost of *Miser* increases logarithmically, while the cost of *Simple* increases linearly. As one can clearly see, for the longer congestion periods, *Miser* is definitely superior. Another interesting behavior is the exponential height and width of the steps in the performance of *Miser*. This is a clear depiction of the 'nature' of the algorithm operation, that doubles its waiting time if the block is still present.

We want to attract the reader's attention to the fact that the encouraging results have been obtained even for relatively short paths which correspond to real topologies. The deadline assumed was also realistic given that the round trip time for such topologies is order of hundreds of milliseconds while the low-priority traffic deadlines is at minimum order of minutes. Finally, it is reasonable to assume that during a few minutes, indeed only one congestion would occur in the network, and the load on active nodes would not change dramatically, so that fixed parking weights are also reasonable. All this provides a strong motivation for the actual implementation of the proposed algorithm.

B. k-block Recoverable TMP

In this subsection we study the behavior of our algorithms for k-block Recoverable TMP. Figures 7 and 13 show the averaged total costs of the itineraries built by the algorithms as a function of path length for k, the number of blocks, being 1, and 14, respectively, for each of the two cost models. The costs are shown on the semi-logarithmic scale. The presented graphs have been obtained by averaging 100 runs for every value of k above, and path length varying from 5 to 20.

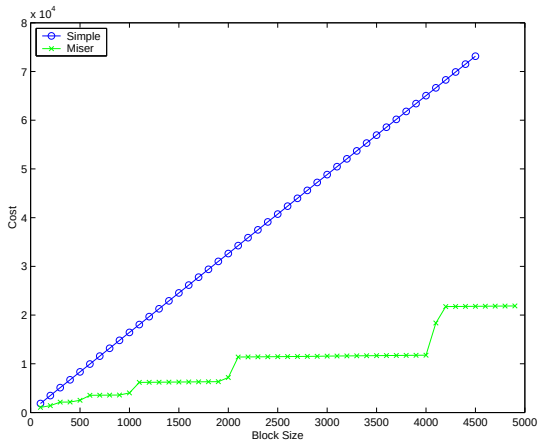

Fig. 12. Itinerary Weight as a Function of block duration

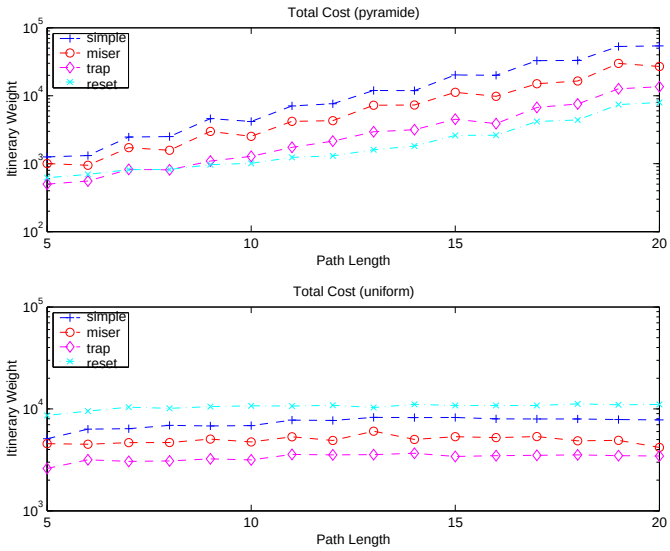

Fig. 13. Average Itinerary Weight (14 blocks; sample 100)

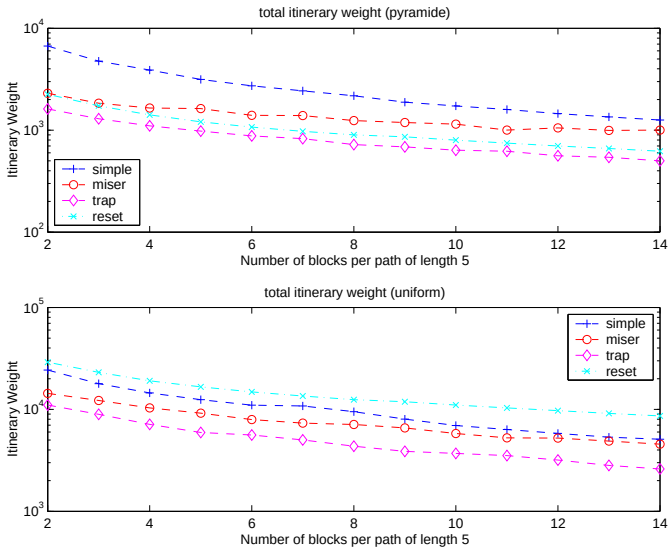

Fig. 14. Average Itinerary Weight (path length 5; sample 100)

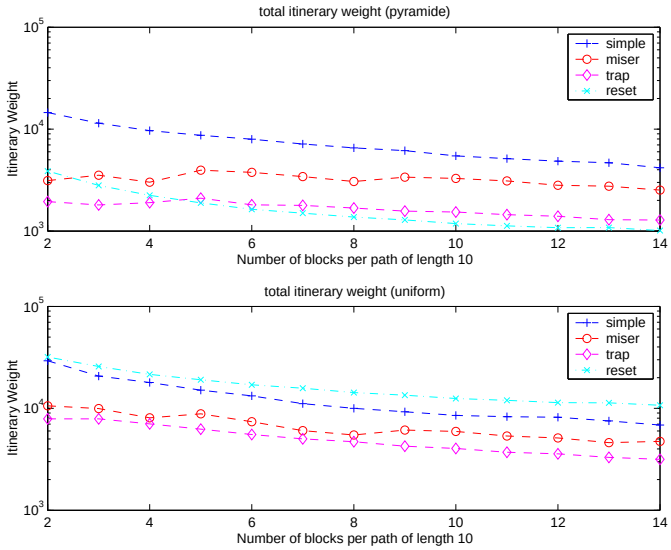

Fig. 15. Average Itinerary Weight (path length 10; sample 100)

Each run is performed as follows. In the beginning of the run, k links that will fail in this run are randomly chosen (repetitions allowed). Then a total block time is randomly chosen from $U(10, \mathcal{D}/2)$ as before. This total block time is split into k shorter blocks that are assigned to the chosen links. Then fail and recovery times are drawn from the exponential distribution for every link chosen in the first step. To make sure that there is at least one meaningful block, the earliest link failure time is treated as 0 time, and all failure and recovery times are shifted left by subtracting the earliest link block time from them.

As one can observe from Figures 7 and 13, the relative behavior of *Miser*, *Simple*, and *Trap* algorithms remains similar as number of blocks increases. However, the absolute weights of itineraries built by the algorithms decrease with the number of blocks. This is natural since the more blocks can occur per fixed deadline D, the higher is the probability that the blocks will overlap. This effectively reduces the total blocking time by the additive factor proportional to the number of blocks.

Reset exhibits somewhat more interesting behavior that requires a more involved explanation. As one may see, in the *pyramid* cost model, *Reset* becomes clearly superior as the number of blocks grows. In the *uniform* cost model, however, *Reset* becomes clearly inferior even to the *Simple* algorithm as more blocks occur. To understand this behavior, we must recall that in our simulations, the total block time remains the same throughout all the runs. Thus, on the average, the block time per link decreases linearly as the number of blocks increases. Figures 14, and 15 show the total itinerary weights of the algorithms as a function of the number of blocks for path lengths 5, and 10, respectively.

When more blocks occur per given path length, the probability of repeated failures occurring on the same link increases. This increases the probability for *Miser*, and *Trap* to build more expensive itineraries in both parking cost models since either repeated backtracking occurs in these algorithms due to multiple link failures, or *Miser*, and *Trap* have no effect since the packet gets locked between two blocks, and thus both behave more like *Simple*. In case of the *pyramid* cost model, source node is always the cheapest for roughly half of the path. Thus,

in case of many short-term blocks it may be beneficial in this model simply to wait at the source until the failures abate. Note that *Reset* does exactly that. In case of the random cost model, the source's parking cost may be very high. Thus, spending the blocking time in the cheap place inside the network may be advantageous. This is the strategy pursued by *Miser*, and *Trap*.

Therefore, the trade-off between the strategies can be put as follows. When the network is very unstable, and the resources in the source node are sufficient (*i.e*, this node is cheap for parking), *Reset* is likely to yield the cheapest schedules. In cases when the network is unstable, but the source does not have sufficient resources (*i.e.*, it is expensive), the on-line strategies like *Miser*, *Trap*, and even *Simple* are likely to be more economical.

VI. Size Dependent Parking Costs Model

The basic model of Sections II, III does not take into account the actual size of the messages. Thus, commuting and parking costs do not differentiate between large and small messages. In this section we refine our basic model by introducing costs that depend linearly on the message size. More specifically, in the definition of k-block recoverable TMP of Section III, $w_{i,j}$ changes as follows.

Given a finite deadline $\mathcal{D}$ and a time-dependent path $R(s,d)$, block sequence $\{b_1, b_2, ..., b_k\}$, $\forall (i,j) \in R$ let link weight function

$$w_{i,j}(t) \stackrel{\text{def}}{=} \begin{cases} \infty & \text{if } \exists l \text{ s.t. } b_l = \langle (i,j), t_l^{start}, t_l^{end} \rangle \\ c_{i,j} \cdot z + x_{i,j} & \text{otherwise} \end{cases}$$

where $c_{i,j}$, and $x_{i,j}$ being finite positive constants, and z being a size of the message.

As one can easily verify, the analysis of Section IV-C still holds for the Miser strategy in our new model. In particular, the competitive ratio of the strategy in case of the single block per path throughout the deadline duration remains $O(\log B)$.

However, if one partition the message into smaller pieces, the Miser strategy can be improved in the new model. Indeed, in the Miser strategy, the message goes back and forth between the blocked node, and the node resulting in the cheapest round-trip itinerary for phase j of the algorithm. The cost of these recurring round-trip itineraries can be minimized if we slightly change our basic on-line strategy. In the following section we describe a modified Miser algorithm that takes advantage of the extended cost model. We call this version *Scout*.

As the size of the minimal message that can be sent over the network decreases, so does the cost of probing the blocked link. Thus, the Scout heuristics asymptotically converges to the Trap strategy discussed in Section IV-D.

VII. Scout Strategy

Let s_{min} be the *minimal MTU* allowed by the underlying communication network. Then we may take an additional advantage of the active network paradigm, by allowing the miser message to spawn a minimal size, zero cost, *scout* message on demand as explained below.

As long as no link *en-route* to the destination is blocked, the message moves forward, as it was the case with the Miser strategy. If a blocked link is encountered at node v_b, the algorithm proceeds in stages exactly as in the Miser algorithm with one important difference. When the parking time of phase j terminates, the message that parks at node v_j^* (of phase j) spawns a scout message of size s_{min} that follows the shortest round-trip itinerary $\mathcal{I}_{rt}(v_j^*, v_b)$ in order to check the status of the blocked link, and inform the parked message about it.

If the previously blocked link recovered, the message moves towards the destination. Otherwise, it behaves like the miser. It computes the cheapest round-trip itinerary for the next phase for all the nodes that are reachable from the current location of the message, and follows this itinerary. Then the process repeats itself at the end of the next phase.

To analyze this strategy, let us first notice that the running time of the algorithm remains the same as for the Miser algorithm up to a small constant factor. This is because each of $1 + \log_2 B$ phases of the algorithm may take slightly longer because of the delay introduced by the scout message.

It is important to observe that if the link is reported by the scout message as blocked at the end of phase j, in phase $j+1$, the message can either move backwards (*i.e.*, towards its original source), or stay in place. However, it would **never** move in the forward direction unless the scout reports that the previously blocked link became passable.

This means that the large message would not travel the path more than three times, while the scout message would not go back and forth more than $\log_2 B$ times. Let $k = \lceil \frac{z}{s_{min}} \rceil$ where z is the original message size. Following the logic similar to the one that was used in proving Theorem 1, we obtain:

$$W(\mathcal{I}_{scout}) \leq 3 \cdot W(\mathcal{I}_{opt}) + \cdot (4 \cdot \log_2 B + 5) \cdot \frac{W(\mathcal{I}_{opt})}{k}.$$

Thus, the competitive ratio of the Scout algorithm is $\frac{(4 \cdot \log_2 B + 5)}{k} + 3$. In other words, we obtain a linear gain in the total cost which is roughly the ratio between the minimal size active message (the scout), and the average size of the management message. Section VII-A shows a comparative simulation study of the *Scout* strategy in the size-dependent parking cost model.

A. Size-Dependent Model Simulation Study

The simulation study of this section is similar to the one of Section V. We use the same two cost models, *pyramid*, and *uniform* as before. However, the parking weights generated at the beginning of each run are treated as coefficients $c_{i,i}$ (parking self-links weights) in the model above. Throughout all the simulations we use message deadline of $10,000$ time units, the total blocking time is drawn from $U(10, \mathcal{D}/2)$, the message size is fixed at 1024, and the scout size is fixed at 128, *i.e.*, the ratio between the bulk of the message and the scout is 8.

Figures 16 and 17 show the relative performance of all algorithms in the size dependent linear cost functions model.

The *Scout* heuristics becomes more advantageous when paths become longer. This is natural since longer paths exhibit sharper differences in parking costs. *Scout* is superior also over short paths when there exist large differences in the parking costs.

Obviously, as the ratio between the whole message and the scout increases, the *Scout* algorithm saves more. We assumed

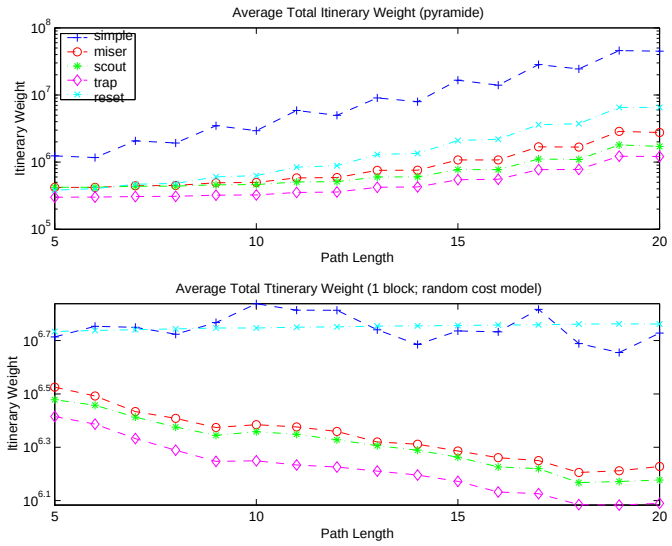

Fig. 16. Average Itinerary Weight as a Function of Path Length (1 block; sample 100)

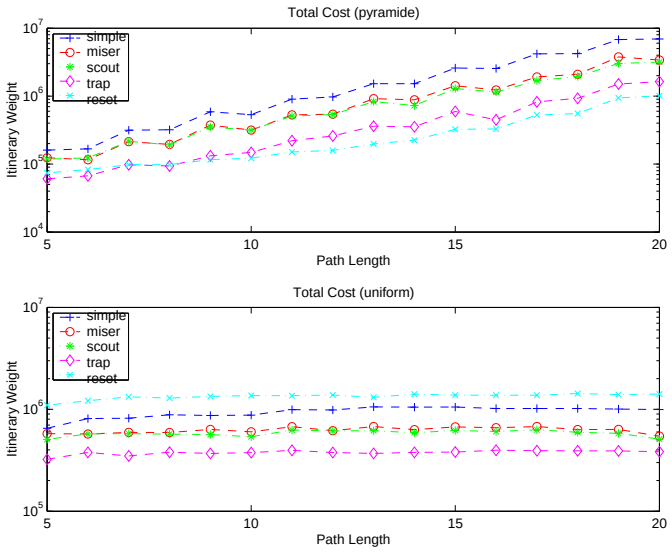

Fig. 17. Average Itinerary Weight as a Function of Path Length (14 blocks; sample size 100)

ratio of 8 in our simulations, but the ratio may easily be at least two orders of magnitude in practice, which would make *Scout* a preferable on-line strategy.

VIII. Related Work

The problems studied in this paper have much in common with the well known problem of finding a minimal weight path. Finding a minimum weight path in a static network has long been the subject for extensive research [7]. The same problem for networks with time-dependent edge lengths has been most extensively studied studied by [8]. Pseudo-polynomial time algorithms for computing the minimal delay path in such networks when delay functions are known, have been demonstrated.

As was pointed out by [4], if real (even if strictly positive) time-dependent link weight functions are associated with traversing the links and parking in the nodes, the problem of finding a minimal weight path is, generally speaking, different from that of finding a path with minimal delay. The resulting paths are not necessarily simple and not necessarily minimal delay paths since, in general, delays are independent from the weights.

The well-known optimality principle [9], stating that a subpath of any optimal path is itself an optimal path, widely used for finding a minimal weight path in a static network, is not valid for time-dependent networks.

In [4], the most general model for time-dependent networks has been studied, and similar, but more restricted optimality principle has been established. We have discussed this principle in Section IV-A.

In [4], the weight functions are assumed to be known in advance. Thus, this work, essentially solves our *off-line* problem.

Very few studies of the on-line variant of the problem exist. One special case is known as the *Canadian Traveler Problem* [10], [6]. In the original version of this problem, studied in [6], a topology of the network (the map) is known in a advance. Each link is assigned a strictly positive finite delay. However, the map is unreliable. When a traveler following a minimal delay path, arrives to a certain hop, it may be impossible to proceed to the next hop since the next link is blocked (has infinite weight). Once a link is blocked it remains blocked forever. The problem is to travel from a source to a destination minimizing total trip time. In [6], it was shown that if the number of blocks is infinite then finding an on-line strategy with a bounded competitive ratio is P-SPACE Complete. The problem remains hard even if the blocks later recover as shown in [10] for *Recoverable CTP*. In [10] an exponential algorithm for a variant of *Recoverable CTP* that was called k*-block recoverable CTP* was presented. The difference of this problem from the original variant is that the number of blocks is fixed, k, and that links recover after a finite period of time.

Also, in [10], a polynomial time strategy minimizing the total travel time from one vertex to another in such time-dependent network was presented. The crucial assumption made for achieving this solution was letting the link down time periods be *smaller* than the all internodal delays. This allowed an elegant recursive solution. Indeed, it is obvious how to solve the problem for $k = 1$. We just need to construct a minimal delay path, between the source and the destination, and for every vertex on this path, we need to construct a minimal delay path from this vertex to the destination. As long as there is no block, the traveler just advances along the minimal delay path. If a blocked link occurs, the traveler switches to an alternative path built from this vertex to the destination. Since only 1 block in the *whole network* may occur when $k = 1$, this algorithm is guaranteed to work. And it, obviously, has a polynomial running time. Now assume that we solved the problem for $k = m$, and wish to solve it for $k = m + 1$. If the traveler gets stopped by the block it may use an alternative (next minimal delay path), and because of the assumption on the inter-nodal delay being larger than any blocking time, when the traveler arrives to another node there are $k = m$ blocks in the networks. The case that we already solved.

Unfortunately, the assumption on the blocks being shorter

than link delays renders the solution impractical for most networking environments where the situation is just the opposite: an internodal delay of a non-faulty link is smaller than the typical down time period.

In our study we, in fact, revisit the k-block Recoverable Canadian Traveler Problem for the special case of the topology being a single path, and we do not make any assumptions on the ratio between the internodal delay and the down time. We limit ourselves to finding only the "interesting" itineraries along this path. Namely, the itineraries that take less than the finite predefined time for the traveler to arrive from the source to the destination.

In [11] studies source-based routing approach to minimizing delay in time-dependent networks by trying several alternative paths in sequence. In this work the network topology is assumed to be known in advance along with the probabilities of the link failures. Under the assumption that during sufficiently long time no new link failures occur, [11] shows that a simple greedy strategy is optimal in a tree-like network. For general network they show that the problem of devising minimal source-based routes can be solved in polynomial space and is #P-Hard.

An active networks paradigm [12] allows seamless deployment of routing protocols at the level of individual *active messages*. Active networks turn out especially useful for wide range of network management applications [2]. In this work we maintain that this paradigm may be successfully deployed for transferring large volumes of management traffic while minimizing its impact on the user applications that consume the same network resources.

IX. Conclusion and Future Work

We presented a novel framework for handling low priority administrative traffic with non-stringent timing constraints. We demonstrated that the active networks paradigm may help in significantly reducing the competition for network resources between this low priority traffic and the application traffic at times of congestion thanks to the intermediate parking capability provided by active networks. Intermediate parking enables to increase the network's goodput, and, as a side effect, makes the management traffic more robust in presence of network congestion.

We proposed an efficient scheduling algorithm that minimizes the overhead imposed on the active network infrastructure itself by the intermediate parking. We showed that the desired effect may be achieved by constructing efficient parking schedules along a fixed routing path. In restricted, but practical case of a single congestion per-path, the proposed algorithm is logarithmically competitive. In less restricted cases, it serves as a useful heuristic, as demonstrated by simulations. Interestingly enough, the encouraging results have been obtained for relatively short paths which corresponds to the real networks' dimensions.

Based on this study, we believe that the active network paradigm may be of practical usefulness in reducing the impact of large amount of management traffic on the user applications that consume the same network resources.

References

[1] S. Blake et. al., "IETF RFC2475 An Architecture for Differentiated Services," `http://www.ietf.org`.

[2] Danny Raz and Yuval Shavitt, "Active networks for efficient distributed network management," *IEEE Communications Magazine*, vol. 38, no. 3, Mar. 2000.

[3] David Breitgand, Danny Raz, and Yuval Shavitt, "The Traveling Miser Problem," in *INFOCOM'02*, New-York, NY, USA, September 2002.

[4] Ariel Orda and Rafael Rom, "Minimum weight paths in time-dependent networks," *Networks*, vol. 21, pp. 295 – 319, May 1991.

[5] He, Y. and Faloutsos, M. and Krishnamurthy, S.V., "Quantifying the Routing Asymmetry in the Internet at the AS Level," in *The Global Internet Symposium, IEEE GLOBECOM'04*, Dallas, TX, 2004.

[6] C.H. Papadimitriou and M. Yannakakis, "Shortest paths without a map," in *16th ICALP*, July 1989.

[7] R. Bellman, "On a routing problem," *Quart. Appl. Math.*, vol. 16, pp. 87 – 90, 1958.

[8] Ariel Orda and Rafael Rom, "Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length," *Journal of the ACM*, vol. 37, pp. 607 – 625, July 1990.

[9] R. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, New Jersey, 1957.

[10] Amotz Bar-Noy and Baruch Schieber, "The canadian traveller problem," in *Second Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, San-Francisco, California, 1991, pp. 261 – 270.

[11] A. Itai and H. Shachnai, "Adaptive source routing in high-speed networks," *Journal of Algorithms*, vol. 20, pp. 218–243, 1996.

[12] Konstantinos Psounis, "Active networks: Applications, security, safety, and architectures," *IEEE Communications Surveys*, vol. 2, no. 1, First Quarter 1999, `http://www.comsoc.org/pubs/surveys/1q99issue/psounis.html`.

PLACE PHOTO HERE

David Breitgand David Breitgand (S'97-M'03) received his doctoral degree in CS from the Hebrew University, Israel, in 2003. Starting October 2003, David Breitgand has been a Member of Technical Staff at IBM Haifa Research Laboratory, Israel, where he is working on advanced management technologies for networked systems and storage. David's research interests are in network and systems management, performance management, networking, distributed computing, and fault tolerance.

PLACE PHOTO HERE

Danny Raz Danny Raz (M'98) received his doctoral degree from the Weizmann Institute of Science, Israel, in 1995. From 1995 to 1997 he was a postdoctoral fellow at the International Computer Science Institute, (ICSI) Berkeley, CA, and a visiting lecturer at the University of California, Berkeley. Between 1997 and 2001 he was a Member of Technical Stuff at the Networking Research Laboratory at Bell Labs, Lucent Technologies. In October 2000, Danny Raz joined the faculty of the computer science department at the Technion, Israel. His primary research interest is the theory and application of management related problems in IP networks.

PLACE PHOTO HERE

Yuval Shavitt Yuval Shavitt (S'88-M'97-SM'00) received the B.Sc. in Computer Engineering (cum laude), M.Sc. in Electrical Engineering and D.Sc. from the Technion — Israel Institute of Technology, Haifa in 1986, 1992, and 1996, respectively. From 1986 to 1991, he served in the Israel Defense Forces first as a system engineer and the last two years as a software engineering team leader. After graduation he spent a year as a Postdoctoral Fellow at the Department of Computer Science at Johns Hopkins University, Baltimore, MD. Between 1997 and 2001 He was a Member of Technical Stuff at the Networking Research Laboratory at Bell Labs, Lucent Technologies, Holmdel, NJ. Starting October 2000, Dr. Shavitt is a faculty member in the School of Electrical Engineering at Tel-Aviv University. He served as TPC member for INFOCOM 2000 – 2003 and 2005, IWQoS 2001 and 2002, ICNP 2001, IWAN 2002-2005 and other conferences, and on the executive committee of INFOCOM 2000, 2002, and 2003. He was an editor of Computer Networks, and served as a guest editor for JSAC and JWWW. His recent research focuses on Internet measurement, mapping and characterization; and QoS routing and Traffic Engineering.