ELSEVIER

Computer Communications 24 (2001) 731–743

computer communications

www.elsevier.com/locate/comcom

Schemes for scheduling control messages by hierarchical protocols

E. Bortnikov, R. Cohen*

Department of Computer Science, Technion, Haifa 32000, Israel

Received 18 February 2000; revised 9 August 2000; accepted 23 August 2000

Abstract

The paper addresses the problem of designing efficient scheduling policies for the transmission of control messages by hierarchical network protocols. Such protocols encounter a tradeoff between the desire to forward a control message across the tree as soon as it is received, and the desire to reduce control traffic. Scheduling problems that arise in this context are defined and discussed. The paper mainly concentrates on minimizing the average extra delay encountered by the control messages under an upper bound on the number of outgoing messages a node can send during a fixed period of time. A polynomial-time algorithm is presented for the off-line version of the problem, and then several efficient on-line heuristics are presented and compared. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Hierarchical protocols; Scheduling algorithms; Dynamic programming; Competitive analysis

1. Introduction

The rapid growth of communication networks raises certain issues that require careful planning. One of the most important issues is protocol scalability. Scalability is usually achieved using hierarchical organization, with summarization of information between levels in the hierarchy. Protocol information maintained by nodes in different hierarchy levels has to be updated from time to time. The updates can be either due to some local activity, or triggered by state changes in the neighboring nodes. Update information is usually exchanged between neighboring nodes using control messages that are propagated in both directions of the tree representing the protocol hierarchy. Frequent updates give the desired effect of a flat network, but waste resources like bandwidth and CPU time. Infrequent updates, on the other hand, waste less resources but lead to a slower propagation of control information between the various hierarchy levels.

There are many protocol families that achieve scalability using hierarchical design. These protocols can be described by the following model:

1. Tree nodes propagate protocol control messages, to be referred as UPDATES, upstream and downstream.
2. The protocol is associated with several variables at each tree node. An UPDATE message may change the value of a variable at the receiving node, in which case the message is referred to as a *state-changing* message. Unless otherwise stated, any UPDATE message we mention throughout the paper is state-changing.
3. A received state-changing UPDATE message triggers the transmission of an outgoing UPDATE message.
4. In accordance with the *soft-state* approach [5], once in a *timeout period* every node sends an UPDATE to its parent in the tree, in order to refresh the state at the parent node.
5. Two or more consecutive UPDATE messages can be *merged* into a single one. Therefore, a node may not necessarily send an upstream UPDATE message immediately following the receipt of such a message. Rather, the node may wait until more UPDATE messages are received, either from the same downstream node or from other nodes. The node may react at some later time by sending a single UPDATE message that reflects its new state resulting from all these UPDATES.

There is an inherent tradeoff between the desire to forward a received UPDATE message across the tree as soon as it is received, and the desire to reduce control traffic. A node that receives an UPDATE from one of its neighbors has the following two options: (1) to send an UPDATE message immediately, with the value of its new state; and (2) to wait some time, in accordance with some scheduling policy, for the receipt of additional state-changing UPDATE messages, and then to send a single outgoing UPDATE. In this case, the incoming UPDATE message is said to encounter an *extra* delay.

* Corresponding author. Tel.: +-972-429-4305; fax: +-972-429-4353.
E-mail address: rcohen@cs.technion.ac.il (R. Cohen).

PII: S0140-3664(00)00276-0

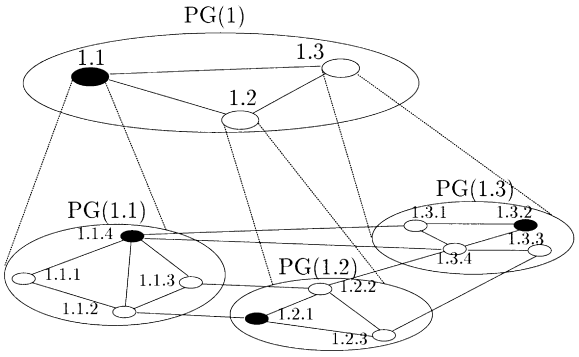

Fig. 1. An ATM PNNI network.

The problem of efficient scheduling of mergeable UPDATE messages arises mainly in highly dynamic environments, where the node states change rapidly and UPDATE messages are frequently exchanged. Unless properly controlled, these messages may flood the network and lay heavy processing load on the network nodes. Hierarchical protocols are probably the best example of the problem, because unless the upstream UPDATE messages are merged, the upper level nodes are likely to be heavily loaded.

The paper addresses the problem of designing efficient scheduling policies for the UPDATE messages. The rest of the paper is organized as follows. Section 2 introduces and discusses several protocols that may benefit from the scheduling policies presented in the paper. Section 3 presents a brief survey of the research in related areas. Sections 4 and 5 address the problem of minimizing the average extra delay that incoming UPDATE messages encounter under a bound on the number of outgoing messages that a node can send during a fixed period of time. Section 4 presents a polynomial-time algorithm that finds an optimal solution for the off-line problem, where the pattern of received messages is known in advance, whereas Section 5 presents some on-line heuristics for solving the problem under a more realistic assumption where the schedule of the incoming messages is not known in advance. Section 6 discusses the problem of minimizing the number of outgoing messages while bounding the extra delay, and Section 7 concludes the paper.

2. Hierarchical network protocols

This section describes several widespread protocol families whose design matches the model described in Section 1. These protocols may therefore benefit from the scheduling algorithms described in the following sections.

Consider first the PNNI routing protocol for ATM networks [1,2]. This protocol adopts the source routing approach, where the source has to determine a route to the destination over which a virtual channel will be set up. Source routing is not a scalable concept, because in large networks nodes cannot obtain up-to-date routing information for remote nodes. Hence, in PNNI the network nodes and links are organized hierarchically. At the lowest level of the hierarchy, each node represents an ATM switch and each link represents a physical link or an ATM virtual path. The nodes and links of each level are recursively aggregated into higher levels, such that a high-level node represents a collection of one or more lower level nodes, and a high-level link represents a collection of one or more lower level links. Nodes within a given level are grouped into sets of peer groups (PGs). Each PG is represented by a peer group leader (PGL). The PGL collects from lower level PGs routing information concerning its local PG, and propagates this information upstream. The messages containing this information can be viewed as the UPDATE messages discussed in Section 1. Consecutive UPDATE messages can be merged by a PGL before being forwarded upstream, even if they are received from different children. As opposed to traditional routing protocols [9], where topology changes occur only when physical links or nodes go up or down, in PNNI the set up and take down of every virtual channel (VC) is considered as a topology change. This is because every VC changes the available resources and QoS parameters, like available bandwidth, delay and jitter, associated with all the links it traverses.

As an example, consider Fig. 1 that shows a three-level ATM PNNI network. The upper level has a single logical node, PG(1). This node consists of three logical level-1 nodes: PG(1.1), PG(1.2) and PG(1.3). PG(1.1) consists of four level-0 (physical) nodes: 1.1.1, 1.1.2, 1.1.3 and 1.1.4. Similarly, PG(1.2) consists of three-level-0 nodes and PG(1.3) consists of four level-0 nodes. Suppose that a VC is set up on link 1.1.3–1.2.2. This VC reduces the available resources associated with this link. Hence, node 1.1.3 needs to broadcast an UPDATE message to all nodes in PG(1.1). However, this change affects also the resources and QoS parameters associated with logical link 1.1–1.2, because this link summarizes the properties of all physical links between PG(1.1) and PG(1.2). Hence, node 1.1.4 — the PGL of 1.1 — needs to send an UPDATE message to the PGLs of 1.3 and 1.2. Instead of sending such an UPDATE immediately, which may impose heavy processing burden on the PGLs, node 1.1.4 may delay the UPDATE until receiving more UPDATE messages, either from 1.1.3 regarding link 1.1.3–1.2.2 or from 1.1.2 regarding link 1.1.2–1.2.1. By merging several UPDATE messages, node 1.1.4 as well as the PGLs of PG(1.2) and PG(1.3), may significantly reduce the control traffic exchanged in PG(1) and the processing load they impose on each other.

As a second example for hierarchical network protocols, consider distributed network management systems [8,12]. In such systems, multiple management servers are organized hierarchically. Every server maintains a Management Information Base (MIB), to allow distributed network monitoring and control. Each server manages a portion of the network agents and, in addition to that, plays the role of an agent to exchange information and accept control from a higher-level management server (its parent

in the management tree). This model is proposed to replace the traditional flat organization where a single server (and optionally a backup server) manages all the agents in the domain. The communication between neighboring nodes is realized via GET and TRAP control messages that update the server MIBs. The servers can merge incoming control messages before forwarding them to neighboring nodes. As an example, consider a server *S* that manages a domain of LANs connected by transparent bridges. Suppose *S* manages multiple servers, where each server is associated with a different bridge. If the spanning tree is re-configured, due to some failure, *S* will be informed of a topological change by all bridges. However, after receiving all UPDATES, *S* can send its parent server a single UPDATE that reports the change in configuration.

Other hierarchical protocols, which may benefit from the proposed scheduling policies, are the resource reservation protocol (RSVP) [5], TCP extensions for reliable multicast, hierarchical protocols for mobile user location in wireless networks [3], and loop-free routing using diffusing computations [7]. Resource reservation protocol has two phases. In the first phase a PATH control message is sent from the source to the destination over the route of the data packets. This message carries the traffic spec (Tspec), and creates a reverse routing state at every router on the path between the source and destination. In the second phase, the receiver responds to the received PATH message with a RESV message which specifies the resources reserved for the data packets by the routers along the path. RSVP is also used for multicast applications. In such a case the PATH messages are sent from the root of the tree towards the leaves, and RESV messages are sent on the reverse direction. RSVP allows the routers to merge two or more RESV messages which are received from different downstream interfaces for the same data flow. Therefore, in a dense multicast group a lot of control RESV messages can be saved if the routers wait before forwarding upstream every received RESV message, in order to increase the probability to merge two or more messages for the same data flow.

3. Related work

Although many practical applications face the control messages implosion problem, no one has defined the problem formally in the past. In those cases where the problem arises, a simple algorithm, tailored to the specific application, was given. One example is the "delayed ACK" scheme in TCP [13], which allows the receiving TCP to merge two ACKs into a single one. Another example for a tailored solution is in RSVP, where a control message is sent immediately if possible, or dropped otherwise.

The problem of optimal scheduling of UPDATE messages subject to some optimization criteria is related to a family of task scheduling problems real-time systems [4,10,14,15]. In such systems, real-time tasks need to be completed by certain deadlines. Every task is associated with an arrival time and an execution time. Tasks may preempt each other, but use the CPU exclusively while being executed. There are typically two kinds of tasks: hard-deadline tasks [10] and soft-deadline tasks [15]. A hard-deadline task has a fixed completion time called deadline. If the deadline is passed, the output of the task has no merit, and the task can be stopped. A soft-deadline task has a weaker requirement: it should be completed as soon as possible provided that the hard-deadline tasks meet their timing constraints. Tasks may also be classified as periodic or sporadic, depending whether they are invoked at constant intervals or at random. Finally, tasks may have priorities that affect the scheduling decisions. In this context, a task schedule is said to be feasible if it meets the hard tasks requirements. It is said to be optimal if it is feasible, and the completion times of the soft tasks meet some optimization criteria.

Another related problem is the scheduling of messages in synchronous networks [11]. In this model, a set of messages should be shipped over the network at certain deadlines. A single message may be sent over a link in a time unit. The problem is to build a feasible traffic schedule, such that all the messages are delivered on time.

In all of the above models, a limited resource (CPU time or link bandwidth) needs to be shared by multiple consumers. In the models addressed in this paper, incoming UPDATE messages may be viewed as consumers, whereas the budget of outgoing UPDATE messages is the resource to be shared. The concept of deadlines is applicable only to our second model (Section 6.1), where the incoming UPDATE messages are allowed to encounter only a limited extra delay. Except for that, there is not much in common between our models and those described above. The main difference is that in our models the "tasks" are *mergeable*, since a single outgoing UPDATE may serve a number of incoming messages, whereas in the previous models the server cannot save resources by serving two tasks together. Hence, solutions discussed in previous works are not applicable to the models described in this paper.

4. Minimum average delay scheduling

4.1. Problem definition and notation

In this section, we address the problem of optimal scheduling of UPDATE messages at intermediate nodes, assuming there exists a limit on the number of UPDATE messages a node can send upstream during a fixed period of time. Due to this limit, a node that sends an UPDATE message upstream each time it receives an UPDATE message from one of its children can rapidly exhaust its allowance. Thus, subsequently received messages will have to wait for the next time slot, and may therefore encounter a large extra delay. Conversely, a node can delay an incoming UPDATE message, hoping that more messages will arrive in the meantime. If

more messages indeed arrive, they are all merged into a single upstream UPDATE message. If, however, no additional UPDATE is received, the node will have to send an UPDATE upstream for the received message, and the extra delay turns to be a "mistake".

The problem is to design a scheduling algorithm that will run at each node, and determine when a new UPDATE message should be sent upstream, while minimizing the average delay of each received UPDATE message. In what follows, we define the problem formally.

Consider a division of the time domain into time slots of length τ. Suppose that each node must send an UPDATE message upstream at the end of every slot. Such a message is referred to as a *mandatory* outgoing message. In addition, every node is allowed to send upstream at most M UPDATE messages during every slot, namely, between every two consecutive mandatory messages. These messages are referred to as *optional* outgoing messages.

Let $S_{\text{in}} = \{I_1, I_2, \ldots, I_k\}$, where $0 < I_1 < I_2 < \cdots < I_k$, denote the arrival times of k incoming UPDATE messages during a time slot. Let $S_{\text{out}} = \{O_1, O_2, \ldots, O_l\}$, where $0 \le O_1 < O_2 < \cdots < O_l < \tau$ and $l \le M$, denote the transmission times of l outgoing UPDATE messages during a time slot. S_{in} and S_{out} will be referred to as the *input schedule* and the *output schedule*, respectively.

We define the *extra delay* of an incoming UPDATE message as the time interval between the arrival of the message and the transmission of the next outgoing message. We denote by Δ_i the extra delay of the ith UPDATE message, and define the *average extra delay* for the whole slot as

$$\frac{1}{k}\sum_{i=1}^{k}\Delta_i.$$

The problem we address is finding an *optimal* schedule for an (S_{in}, M) pair. The schedule $S_{\text{out}} = \{O_1, O_2, \ldots, O_l\}$, where $l \le M$, is said to be optimal if the following holds:

- There is no other schedule $S'_{\text{out}} = \{O'_1, O'_2, \ldots, O'_{l'}\}$, where $l' \le M$, that yields a smaller average extra delay than S_{out}.
- There is no other schedule $S'_{\text{out}} = \{O'_1, O'_2, \ldots, O'_{l'}\}$, where $l' < l$, that yields the same average extra delay as S_{out}. This condition guarantees that the optimal solution does not use unnecessary outgoing messages.

The optimal solution for the period $[0, i\tau]$, for every $i \ge 1$, is the union of the optimal solutions for slots $[0, \tau], (\tau, 2\tau], \ldots, ((i-1)\tau, i\tau]$. This is because during every time slot a node can send at most M optional messages regardless of the number of messages sent during previous slots, implying that the solutions for every two slots are independent of each other. Therefore, in order to find an optimal long-run schedule, it is sufficient to find the optimal schedule for each time slot. Without loss of generality, we shall concentrate upon solving the problem for the first time slot $[0, \tau]$ only.

4.2. An optimal off-line algorithm

In the off-line version of the problem, the input schedule S_{in} is known in advance, whereas in the on-line version I_i is known only when the ith message is received. In what follows, we analyze the optimal solution structure for the off-line problem and present a naive algorithm that computes the optimal solution in exponential time. We then present a dynamic programming algorithm whose running time complexity is $O(Mk^2)$.

Note that a schedule S_{out} minimizes the average delay if and only if it minimizes the total extra delay $\sum_{i=1}^{k}\Delta_i$. The minimum possible total delay for (S_{in}, M) will be denoted as $\Delta^{\text{OPT}}(S_{\text{in}}, M)$.

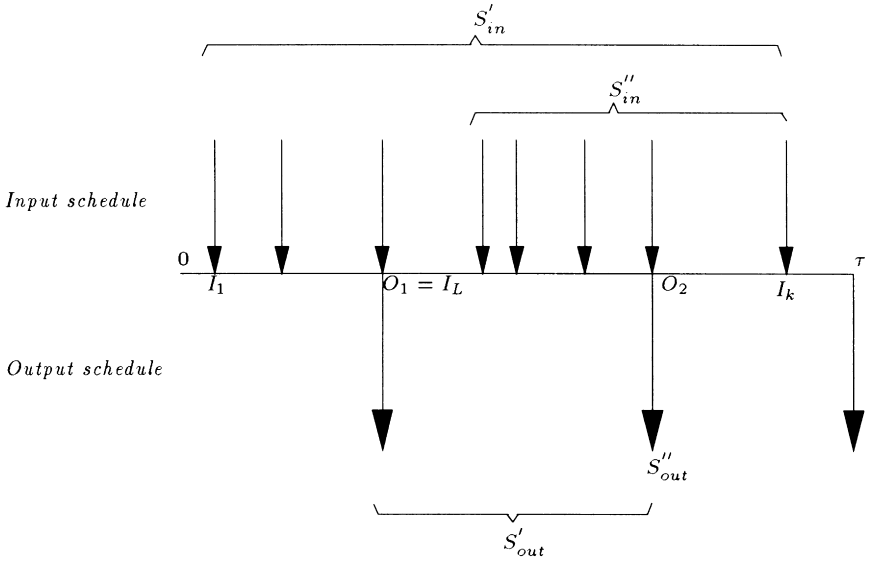

Fig. 2. The recursive structure of the solution for $k = 8$, $M = 2$.

In the following, if an UPDATE message is sent at time t following the receipt of an incoming UPDATE message, then the latter is said to be received at t^-. The following lemma is the key observation about the structure of the optimal solution.

Lemma 4.1. *Let* $S_{\text{out}} = \{O_1, O_2, ..., O_l\}$ *be an optimal schedule for* (S_{in}, M). *Then, for every* j, *where* $1 \le j \le l$, *there exists an* UPDATE *message whose arrival time is* O_j^-.

Proof. Consider an optimal schedule S_{out} for which the lemma does not hold. Hence, there exists a time O_j when an UPDATE message was sent after no message is received. Observe the last UPDATE message that arrived before O_j (if such a message does not exist, then the messages sent at $O_1, ..., O_j$ are redundant and S_{out} is not an optimal schedule, in contradiction to our assumption). Let I^- be the arrival time of this message. Now, let us change the transmission time of the j-th message from O_j to I. This results in a new schedule with the same number of outgoing messages, that does not increase the extra delay of any message, while decreasing the delay of at least one message. Hence the total delay of the new schedule is smaller than the total delay of S_{out}, in contradiction to our assumption. ■

By Lemma 4.1, each outgoing message in the optimal solution is sent when some incoming message arrives. Therefore, the optimal output schedule can be found by an exhaustive search over all $\binom{k}{M}$ possible schedules of M messages at k times. However, the running time of such algorithm is exponential. In the following, we present a dynamic programming algorithm that solves the optimal off-line scheduling problem in polynomial time. It relies on the following lemmas:

Lemma 4.2. *If* $M = 0$, *then*

$$\Delta^{\text{OPT}}(S_{\text{in}}, M) = k\tau - \sum_{i=1}^{k} I_i.$$

Proof. By definition, when $M = 0$, an UPDATE message is sent only at the end of the time slot. Hence, the ith received UPDATE message, that arrives at I_i, encounters a delay of $\tau - I_i$, and

$$\Delta^{\text{OPT}}(S_{\text{in}}, 0) = \sum_{i=1}^{k} (\tau - I_i) = k\tau - \sum_{i=1}^{k} I_i. \qquad \square$$

Lemma 4.3. *Let* $S'_{\text{in}} = \{I_1, I_2, ..., I_k\}$ *and* $M > 0$. *Let* $S'_{\text{out}} = \{O_1, ..., O_l\}$ *be an optimal schedule for* (S'_{in}, M). *Let* $S''_{\text{in}} = \{I_i | I_i \in S'_{in} \wedge I_i > O_1\}$. *Then:*

(a) $S'_{\text{out}} - \{O_1\}$ *is an optimal schedule for* $(S''_{\text{in}}, M-1)$. *(see* Fig. 2).

(b)

$$\Delta^{\text{OPT}}(S'_{\text{in}}, M) = \sum_{i=1}^{L} (O_1 - I_i) + \Delta^{\text{OPT}}(S''_{\text{in}}, M-1),$$

where I_L *is the arrival time of the last message that arrived before* O_1 (*see* Fig. 2).

Proof.

(a) To prove this part it is sufficient to show that if $S'_{\text{out}} - \{O_1\}$ is not an optimal schedule for $(S''_{\text{in}}, M-1)$, then S'_{out} is not an optimal schedule for (S'_{in}, M). Let S''_{out} be an optimal schedule for $(S''_{\text{in}}, M-1)$. An assumption that $S'_{\text{out}} - \{O_1\}$ is not an optimal schedule for $(S''_{\text{in}}, M-1)$ implies that it yields a larger total delay for the input schedule S''_{in} than S''_{out}. Hence, the output schedule $\{O_1\} \cup S''_{\text{out}}$ yields a smaller total delay for S'_{in} than S'_{out} does, in contradiction to our assumption.
(b) An UPDATE message that arrives at $I_i \le O_1$ encounters an extra delay of O_1–I_i. Hence, the total extra delay encountered by all the messages that arrived before O_1 is $\sum_{i=1}^{L} (O_1 - I_i)$. Since by (a) $S'_{\text{out}} - \{O_1\}$ is an optimal schedule for $(S''_{\text{in}}, M-1)$, the total delay encountered by the remaining UPDATE messages is $\Delta^{\text{OPT}}(S''_{\text{in}}, M-1)$, and the claim holds. □

We define the *i-suffix* of an input schedule $S_{\text{in}} = \{I_1, ..., I_k\}$ as the sub-schedule of i last messages in S_{in}, i.e. $S^i_{\text{in}} = \{I_{k-i+1}, ..., I_k\}$, where $1 \le i \le k$. We also define S^0_{in} as $\varnothing$. The problem of computing an optimal output schedule for (S^i_{in}, M') will be referred to as the (S^i_{in}, M')-subproblem. By definition, the solution of the original problem is the solution of the (S^k_{in}, M)-subproblem.

The dynamic programming algorithm operates as follows. It maintains a two-dimensional array, called $\Delta[0...k, 0...M]$. At the end of the execution, $\Delta[i,j]$ holds the total delay of the optimal schedule for (S^i_{in}, j), and therefore $\Delta[k, M]$ holds the total delay yielded by the optimal schedule for the original problem.

The algorithm consists of three phases. During every phase some of the table entries are filled in. The solid arrows in Fig. 3 depict the entries that are filled in every phase. The pseudo-code for the algorithm is given in Fig. 4.

During the first phase, row $\Delta[0,]$ is filled by 0's, because for every i the delay yielded by the optimal schedule for $(\varnothing, i)$ is 0. From Lemma 4.2 follows that for every $0 \le i \le M$:

$$\Delta[i, 0] = i\tau - \sum_{l=k-i+1}^{k} I_l.$$

Therefore, $\Delta[i+1, 0] = \Delta[i, 0] + \tau - I_{k-i}$, where $0 \le i \le k-1$. Phase 2 of the algorithm uses this equality in order to fill up all the entries of the column $\Delta[\ ,0]$.

The rest of the table is filled during the third phase, as

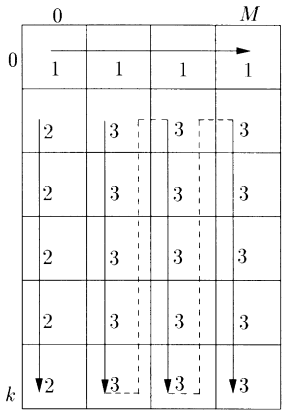

Fig. 3. The order of filling $\Delta[0...k, 0...M]$ in three phases.

depicted by Fig. 3 in the following way. By Lemma 4.1, the transmission time I of the first UPDATE message in the optimal schedule for (S^i_{in}, j), is $I \in \{I_{k-i+1}, \ldots, I_k\}$. Let $I = I_{k-l}$, where $0 \le l \le i-1$. Observe that $S^l_{in} = \{I_{k-l+1}, \ldots, I_k\}$. From Lemma 4.3 follows that

$$\Delta^{\text{OPT}}(S^i_{\text{in}}, j) = \sum_{t=k-i+1}^{k-l} (I_{k-l} - I_t) + \Delta^{\text{OPT}}(S^l_{\text{in}}, j-1).$$

Hence,

$$\Delta[i,j] = \min_{0 \le l \le i-1} \left(\sum_{t=k-i+1}^{k-l} (I_{k-l} - I_t) + \Delta[l, j-1] \right) \quad (1)$$

The computation of a single table entry during phase 3 requires $O(k)$ iterations. If in every iteration $\sum$ is computed from scratch, the execution time will be $O(k)$ for each iteration, and $O(k^2)$ for each entry. However, the following equality can be employed in order to reduce the entry computation time to $O(k)$:

$$\sum_{t=k-i+1}^{k-l} (I_{k-l} - I_t) = (i-l)I_{k-l} - \sum_{t=k-i+1}^{k-l} I_t$$

$$= (i-l)I_{k-l} + \left(\sum_{t=1}^{k-l} I_t - \sum_{t=1}^{k-i} I_t \right). \quad (2)$$

In order to use Eq. (2) the algorithm employs an additional vector called δ. In phase 0, the algorithm sets $\delta[0] = 0$ and $\delta[i] = \sum_{l=1}^{i} I_l = \delta[i-1] + I_i$ for $1 \le i \le k$. Substituting Eq. (2) into Eq. (1) yields:

$$\Delta[i,j] = \min_{0 \le l \le i-1} ((i-l)I_{k-l} - (\delta[k-l] - \delta[k-i]) + \Delta[l, j-1]). \quad (3)$$

This computation is performed during phase 3 of the algorithm. Each step of the loop requires $O(1)$ operations, implying that the time complexity for the whole computation of each table entry is $O(k)$.

At the end of phase 3, $\Delta[k, M]$ holds the total delay that the optimal schedule yields. In order to determine the schedule itself, the algorithm in Fig. 4 maintains another array, called $next[0...k, 0...M]$. An entry $next[i,j]$ in this array serves as an index for the UPDATE message arriving at $I_{next[i,j]}$, which is also the transmission time of the first outgoing message in the optimal schedule for (S^i_{in}, j). If the latter is empty, then $next[i,j] \triangleq_{\text{nil}}$.

Phase 4 of the algorithm recovers the linked list of the outgoing messages transmission times starting from $next[k,M]$. Observe that if $next[i,j] = l \ne \text{nil}$, then the first message in the optimal schedule for (S^i_{in}, j), S' say, is transmitted at I_l. From Lemma 4.3, follows that $S' = \{I_l\} \cup S''$, where S'' is the optimal schedule for $(S^{k-l}_{\text{in}}, j-1)$. Hence, the list recovery should continue from $next[k-l, j-1]$. The pointers are followed in this way until nil is reached.

The analysis of the time complexity of the algorithm is as follows. Phases 0,1 and 2 require $O(k)$, $O(M)$, and $O(k)$ operations, respectively. During phase 3, $O(Mk)$ entries are computed, where each entry computation requires $O(k)$ operations. Hence, the running time of phase 3 and of the whole algorithm is $O(Mk^2)$.

Input: $S_{in} \triangleq \{I_1, \ldots, I_k\}$
M.

Output: An optimal schedule for S'_{in}, $S_{out} \triangleq \{O_1, \ldots, O_l\}$.
$\Delta^{\text{OPT}}(S_{in}, M)$.

```
0. δ[0] ← 0
   for i := 1 to k do
       δ[i] ← δ[i − 1] + I_i

1. for j := 0 to M do
       Δ[0, j] ← 0
       next[0, j] ← nil

2. for i := 0 to k − 1 do
       Δ[i + 1, 0] = Δ[i, 0] + τ − I_{k−i}
       next[i + 1, 0] ← nil

3. for j := 1 to M do
       for i := 1 to k do
           Δ[i, j] ← ∞
           next[i, j] ← nil
           /* Consider Δ(S^l_in, j − 1), for 0 ≤ l < i */
           for l := 0 to i − 1 do
               Δ' ← (i − l)I_{k−l} − (δ[k − l] − δ[k − i]) + Δ[l, j − 1]
               if (Δ' < Δ[i, j]) then
                   Δ[i, j] ← Δ'
                   next[i, j] ← k − l

4. i ← k, j ← M
   S_out ← Ø
   while (next[i, j] ≠ nil) do
       S_out ← S_out ⋃ {I_{next[i,j]}}
       i ← k − next[i, j]
       j ← j − 1

return (S_out, Δ[k, M])
```

Fig. 4. A dynamic programming algorithm for off-line minimum total delay scheduling.

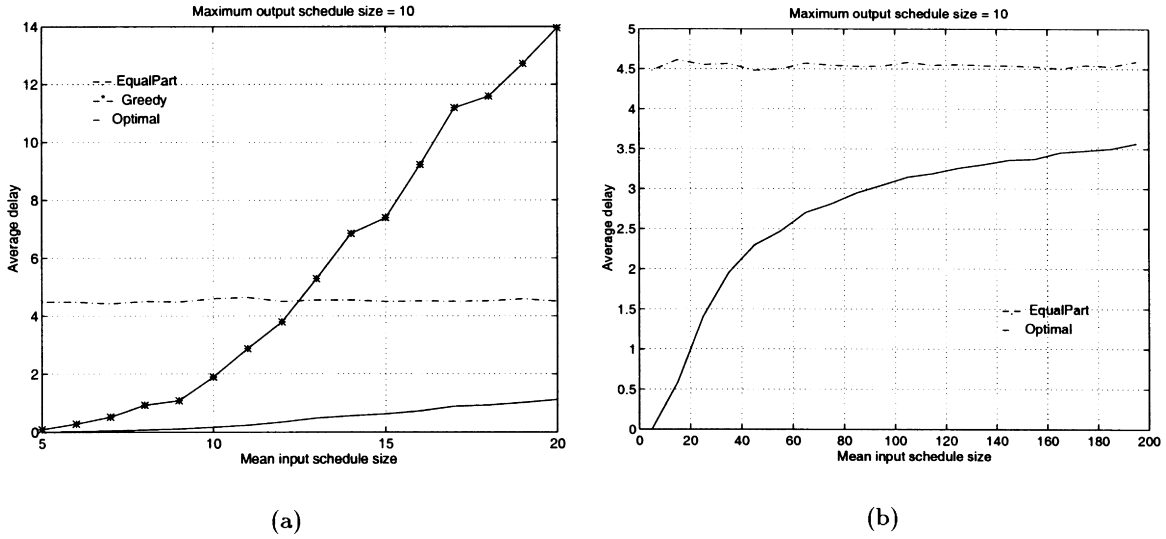

Fig. 5. The performance of *Greedy* and *Equalpart*.

4.3. An alternative definition for "extra delay"

The extra delay definition implicitly assumes that the UPDATE messages are independent of each other, because it aggregates the extra delays regardless of which state variables each UPDATE affects. In practice, this is not always the case. In the following discussion, two UPDATE messages that affect the same state variable are said to have the same type. Suppose that two succeeding UPDATEs of the same type are received by a node, and suppose that no outgoing message is sent between their arrivals. Then, the effect of the first UPDATE terminates upon the arrival of the second one. Hence, the extra delay the first UPDATE encounters is actually the time interval between the two arrivals.

A small modification in the dynamic programming algorithm can be made in order to compute an optimal schedule under the new definition. First, note that Lemma 4.1 still holds. Now, let O and O', where $O < O'$, be the transmission times of two consecutive outgoing messages. Let the arrival times of all the UPDATEs of the same type received between O and O' be $I'_1 < \cdots < I'_l$. By the new definition, these messages encounter extra delays of $I'_2 - I'_1, I'_3 - I'_2, \ldots, I'_l - I'_{l-1}$, and $O' - I'_l$, respectively. Hence, the total extra delay encountered by all these messages is $O' - I'_1$. This is exactly the extra delay encountered by the first UPDATE according to the original definition. Therefore, in order to use the existing algorithm for computing an optimal schedule under the new definition, one has to limit the summation of extra delays only to the first message of every type the node receives after the last UPDATE transmission.

5. On-line algorithms for minimum average delay scheduling

In this section, we present a family of simple on-line heuristics for the minimum average delay-scheduling problem. These heuristics try to imitate the following behavior of the optimal algorithm. On one hand, the optimal algorithm "recognizes" bursts of incoming messages in the input schedule, and reacts on each burst by sending an UPDATE message. On the other hand, the optimal algorithm balances the delays that incoming messages encounter, by not delaying a large group of messages for a long period of time while serving another group of comparable size quickly.

The simplest heuristic is referred to as *Greedy*. As long as it has used less than M outgoing messages, *Greedy* sends an outgoing message in response to each incoming message. If the allowance is exhausted, it delays each arriving message until the end of the slot. Hence, *Greedy* performs optimally on input schedules of size M or less, but may yield an average delay close to τ for heavily loaded inputs.

The second heuristic is referred to as *EqualPart*. The scheduler divides the interval $[0, \tau]$ to $M + 1$ equal sub-slots, and outgoing messages are sent only at the end of each sub-slot. If no incoming message arrives during some sub-slot, an outgoing message is not sent at the end of the sub-slot.

The *EqualPart* algorithm provides the following guarantees regardless of the input schedule size. First, the maximum delay a message encounters is $(\tau/(M + 1))$. Second, if the incoming messages are generated by a Poisson process, the expected average delay is $(\tau/2(M + 1))$. The main disadvantage of this algorithm is its performance under a lightly loaded arrival schedule, where *Greedy* serves incoming messages very fast.

In the following, we use simulations in order to compare the performance of the heuristics to the performance of the optimal algorithm. The slot size is $\tau = 100$, the average size of the schedules varies from 5 to 200, and the maximum number of outgoing messages is 10. For every average size

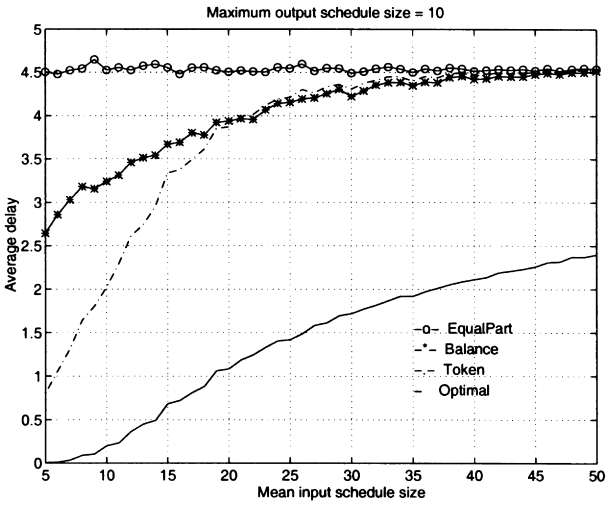

Fig. 6. Performance of timer based heuristics.

of the input schedule, each algorithm was executed on 500 different exponentially distributed input schedules, and the received results were averaged.

Fig. 5 depicts the performance of *Greedy* and *EqualPart* versus the optimal (off-line) algorithm. As expected, *Greedy* performs well for lightly loaded schedules, but becomes impractical as the input load increases (see Fig. 5(a)). In contrast, the performance of *EqualPart* improves as the input schedule becomes heavier loaded (Fig. 5(b)).

EqualPart does not use outgoing messages that are scheduled to be sent at the end of empty sub-slots. Hence, it may send less than M outgoing messages even when the number of incoming messages is larger than M. In what follows, we propose two variations of *EqualPart* that avoid this weakness.

The first variation, referred to as *TokenAlgorithm*, uses the notion of *tokens*. At the beginning of a slot, the algorithm has 0 tokens. When an UPDATE message arrives, the algorithm operates as follows:

- If the number of tokens is greater than 0, an outgoing message is immediately sent and the number of tokens decreases by 1.
- Otherwise, the message is delayed until the end of the sub-slot.

At the end of the sub-slot, the algorithm operates as follows:

- If there is no waiting message, the token pool grows by 1.
- Otherwise, an outgoing message is sent immediately, and the number of tokens does not change.

BalanceAlgorithm is another variation of *EqualPart*. Like *TokenAlgorithm*, it uses the notion of tokens, but uses them in a less greedy way, because each saved token reduces the length of all remaining sub-slots. Let n denote the number of UPDATE messages a node can send until the end of the slot, and T denote the time remaining until the end of the slot. When the slot starts, n is initialized to $M + 1$ and T to τ. The length of each sub-slot is determined at the end of the previous sub-slot as T/n. Hence, the first sub-slot ends at $(\tau/(M + 1))$. If during a sub-slot i there is no incoming message, no UPDATE message is sent at the end of the sub-slot, and $(T \leftarrow T - (T/n))$ whereas n does not change. Consequently, each of the remaining sub-slots is shorter than sub-slot i. If an incoming message does arrive during sub-slot i, an UPDATE is sent at the end of the sub-slot, $(T \leftarrow T - (T/n))$ and $n \leftarrow n - 1$. Consequently, the length of sub-slot $i + 1$ is equal to the length of sub-slot i.

Fig. 6 depicts the performance of *TokenAlgorithm* and *BalanceAlgorithm*. The curves of the optimal scheduler and *EqualPar*t are given as a reference. It is obvious that for heavily loaded input both algorithms perform the same as *EqualPart*, because the likelihood of having an empty sub-slot is small. For lightly loaded input, *TokenAlgorithm* performs much better, like *Greedy*, since a single saved token may cause all the following messages not to be delayed at all. In contrast, *BalanceAlgorithm* uses a saved token only to decrease the delay encountered by subsequently received messages. An analysis of the performance of *TokenAlgorithm* is presented in Appendix A.

Recall that by Lemma 4.1, the optimal scheduler sends every UPDATE immediately after having received an incoming message. However, under heavy load, the transmission of an outgoing UPDATE message by *TokenAlgorithm* and *BalanceAlgorithm* is time driven, and therefore it is rarely triggered immediately after an incoming message is received. This happens because the node has no a-priori information about the statistical behavior of the incoming messages. Suppose the algorithm knows the mean rate of UPDATE message arrival. Hence, the expected input schedule size k can be pre-computed. Recall that the number of outgoing messages that can be used during a slot, including the mandatory one, is $M + 1$. This number can be referred to as a potential that needs to be exhausted before the slot ends. The *VirtualClock* heuristic aims in spending an equal share of the potential on every received message. It defines the *clock rate* as $(r = (M + 1)/k)$.

This ratio expresses the expected part of potential associated with each incoming message. A received message is always delayed, and the virtual clock is increased by r. If the new value is ≥ 1 (i.e. there is enough potential for an outgoing message), and the algorithm has sent less than M outgoing messages in the current slot, an outgoing message is sent and the clock value is decreased by 1. In addition to this procedure, a mandatory message is sent at the end of the slot.

The algorithm uses an estimate on the input schedule size for evaluating the correct clock rate. If the input schedule is loaded more than expected, the algorithm will run out of reserve messages faster than expected, and the last incoming messages will encounter a relatively long delay. If, on the other hand, the input schedule is loaded less than

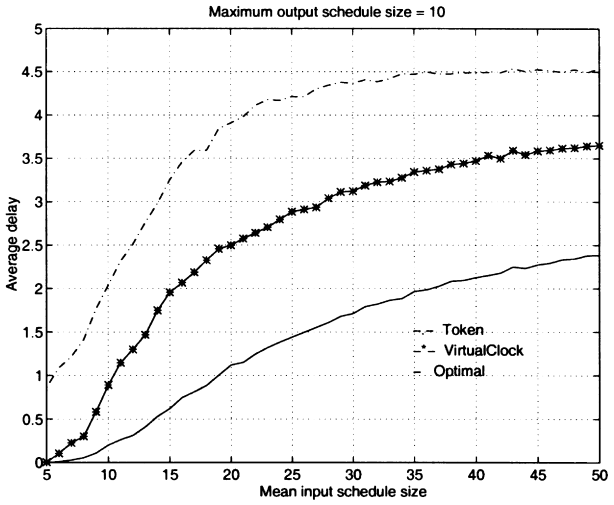

Fig. 7. Performance of *TokenAlogrithm* and *VirtualClock*.

expected, the algorithm may use less than M optional messages.

VirtualClock overcomes an inherent problem of all the timer-based approaches, namely their inability to recognize bursts in the input schedule and to react accordingly. When a burst of incoming messages is received, *VirtualClock* may trigger the transmission of an outgoing UPDATE message even if the previous message has just been sent. On the other hand, it may delay for a long time an isolated incoming message that does not affect the average delay significantly. The ideas behind this heuristic are similar to those behind Lixia Zhang's traffic control algorithm for packet switching networks, which is also called *VirtualClock* [6,16].

Fig. 7 presents simulation results for the performance of *VirtualClock* versus *TokenAlgorithm*. The curve of the optimal scheduler is given again as a reference. It is evident from the figure that *VirtualClock* beats all the timer-based approaches. Its performance is much closer to the optimum for both lightly and heavily loaded input schedules. The results were obtained by modeling a Poisson message generation process. If this process schedules the received messages in bursts, *VirtualClock* is even more advantageous.

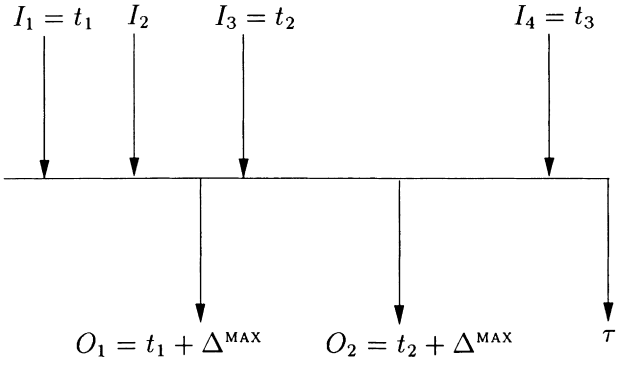

Fig. 8. An example of *GreedyMaxDelay* operation.

6. Minimum outgoing messages

Suppose that in order to meet control message delivery deadlines, there exists a constraint on the extra delay that every UPDATE can encounter at every node. A weaker constraint is to bound the *average* extra delay at every node. This section presents off-line and on-line algorithms that minimize the number of outgoing messages while meeting these two constraints.

6.1. Minimum outgoing messages under a maximum delay constraint

Suppose there exists an upper bound Δ^{MAX} on the delay an incoming UPDATE message can encounter at the node. In such a case an optimal schedule will be the one that guarantees this bound while using a minimum number of outgoing messages.

In this context, an output schedule $S_{out} = \{O_1, O_2, ..., O_l\}$ is said to be legal for (S_{in}, Δ^{MAX}) if it yields a delay of at most Δ^{MAX} for every message in $S_{in} = \{I_1, ..., I_k\}$. It is said to be optimal if it is legal and there is no other legal schedule $S'_{out} = \{O'_1, O'_2, ..., O'_{l'}\}$ such that $l' < l$. The optimal schedule size for (S_{in}, Δ^{MAX}) will be denoted as $M^{OPT}(S_{in}, \Delta^{MAX})$.

In what follows, we show that the problem can be solved by a simple on-line algorithm, referred to as *GreedyMaxDelay*. The algorithm sends the ith UPDATE message at $O_i = t_i + \Delta^{MAX}$ (unless $O_i > \tau$), where $t_1 = I_1$ and for $i > 1$ $t_i = \min\{I_j | I_j > O_{i-1}\}$.

This algorithm can be easily implemented using a timer as follows. At the beginning of the slot, the timer is off. When an UPDATE message is received, at time t say, the algorithm performs as follows:

- The incoming message is delayed.
- If the timer is off and $t + \Delta^{MAX} < \tau$, the timer is initialized to Δ^{MAX} and starts counting down.

When the timer expires, an outgoing message is transmitted. Fig. 8 demonstrates the operation of *GreedyMaxDelay*.

Theorem 6.1. *GreedyMaxDelay produces an optimal schedule for* (S_{in}, Δ^{MAX}).

Proof. It is easy to see that the output schedule $S_{out} = \{O_1, ..., O_l\}$ generated by the algorithm is legal, in the sense that no incoming message encounters, a delay larger than Δ^{MAX}.

To prove that S_{out} is also optimal, suppose exists a legal schedule S'_{out} that uses less than l outgoing messages. Let us define l *phases* in the considered slot, where the ith phase starts at t_i and ends at O_i. Since the schedule S'_{out} uses less

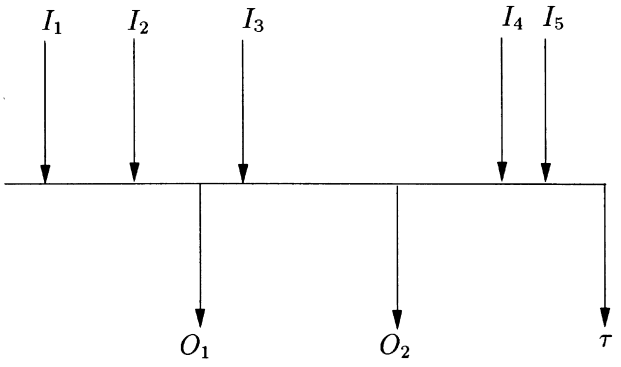

Fig. 9. An example of *GreedyAverageDelay* operation.

than l messages, there is necessarily a phase, say i, where S'_{out} sends no message. This implies that the incoming message that arrives at t_i encounters a delay of more than Δ^{MAX}. Hence, S'_{out} is illegal, in contradiction to our assumption. □

6.2. Minimum outgoing messages under an average delay constraint

The last problem we address is finding an optimal schedule while bounding the *average* delay. In this context, an output schedule $S_{\text{out}} = \{O_1, O_2, \ldots, O_l\}$ is said to be legal for $(S_{\text{in}}, \Delta^{\text{AVG}})$ if it yields an average delay of at most Δ^{AVG} for $S_{\text{in}} = \{I_1, \ldots, I_k\}$. It is said to be optimal if it is legal and there is no other legal schedule $S'_{\text{out}} = \{O'_1, O'_2, \ldots, O'_{l'}\}$ such that $l' < l$. The optimal schedule size for $(S_{in}, \Delta^{\text{AVG}})$ will be denoted as $M^{\text{OPT}}(S_{\text{in}}, \Delta^{\text{AVG}})$.

An off-line solution for the problem can be found using a variation of the algorithm for minimum average delay scheduling (Section 4). By definition, for every $M \geq 0$ and S_{in}

$$\Delta^{\text{OPT}}(S_{\text{in}},M) \geq \Delta^{\text{OPT}}(S_{\text{in}},M+1).$$

Hence, if $(\Delta^{\text{OPT}}(S_{in}, M))/k > \Delta^{\text{AVG}}$ holds for some M, no output schedule of size M or less can yield an average delay of Δ^{AVG} or less for S_{in}. Therefore, $M^{\text{OPT}}(S_{\text{in}},\Delta^{\text{AVG}})$ is the smallest value of M that fulfills:

$$\frac{\Delta^{\text{OPT}}(S_{in}, M)}{k} \leq \Delta^{\text{AVG}}.$$

Once M^{OPT} is determined, a possible optimal schedule is the one that yields the minimum average delay for $(S_{\text{in}}, M^{\text{OPT}})$.

$M^{\text{OPT}}(S_{\text{in}}, \Delta^{\text{AVG}})$ can be found by performing a binary search on the interval $[1, \ldots, k]$, invoking the dynamic programming algorithm described in Section 4 at each step. Such a search would require $\log_2 k$ steps. However, a single invocation of the dynamic programming algorithm would suffice if the following modification is made. The modified algorithm maintains a table $\Delta[0 \ldots k,]$ of $k+1$ rows and a dynamic number of columns. Columns are added to Δ and computed one by one, until j for which $(\Delta[k,j]/k) \leq \Delta^{\text{AVG}}$ is found. The time complexity of the algorithm is $O(M^{\text{OPT}}(S_{\text{in}}, \Delta^{\text{AVG}})k^2)$. Since $M^{\text{OPT}}(S_{\text{in}}, \Delta^{\text{AVG}}) \leq k, O(k^3)$ is an upper bound.

In what follows we discuss two possible on-line heuristics for the problem. The *GreedyMaxDelay* algorithm presented in Section 6.1 is one possible solution. It produces a legal output schedule by guaranteeing a delay of at most Δ^{AVG} for each incoming message. However, the produced schedule is likely to be inefficient because an optimal schedule may delay some messages longer than Δ^{AVG}.

A better heuristic is the *GreedyAverageDelay* algorithm. This algorithm delays the next outgoing message as long as the average delay of all the messages received after the transmission of the last outgoing UPDATE does not exceed Δ^{AVG}. Let $B_i = \{I_j | O_{i-1} < I_j \leq O_i\}$, where $O_0 \triangleq 0$. Then, the algorithm fulfills the following for every $1 \leq i \leq l$:

$$\frac{\sum_{I \in B_i} (O_i - I)}{|B_i|} = \Delta^{\text{AVG}}. \quad (4)$$

The only difference between this algorithm and the *GreedyMaxDelay* is that if an UPDATE message arrives at time t and the timer is on, scheduled to expire at t', its expiration is *rescheduled* to $t' + [\Delta^{\text{AVG}} - (t' - t)]/l$, where l is the number of waiting messages at t^+ (i.e. including the new one). If $t' \geq \tau$, the timer is turned off. It is easy to verify that the algorithm indeed maintains invariant (4). Fig. 9 demonstrates the operation of *GreedyAverageDelay*. By the scheduler's definition, $(O_1 - (I_1 + I_2)/2) = \Delta^{\text{AVG}}$, $O_2 - I_3 = \Delta^{\text{AVG}}$, and $\tau - (I_1 + I_2)/2 \leq \Delta^{\text{AVG}}$ hold.

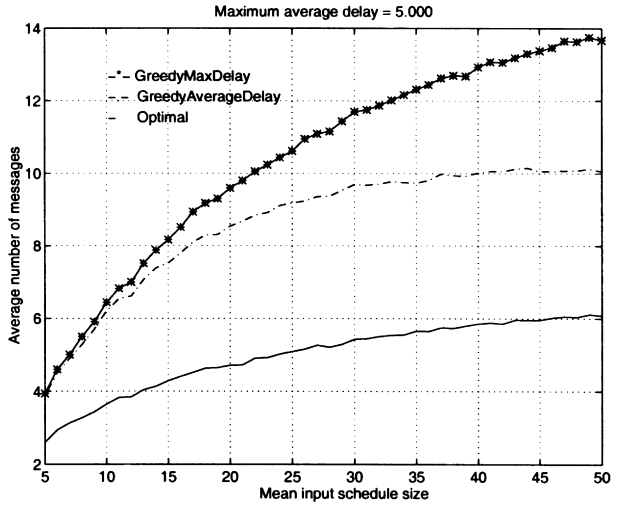

Fig. 10. Performance of GreedyMaxDelay and GreedyAverageDelay.

Fig. 10 presents simulation results of the two heuristics versus the optimum. As in the previous simulations, the process of UPDATE generation is Poisson. It is evident that *GreedyAverageDelay* can meet the average delay constraint while using significantly less messages than *Greedy-MaxDelay*.

Denote the sizes of the output schedules that *GreedyMaxDelay* and $(S_{in}, \Delta^{\text{AVG}})$ by $M^{\text{GMD}}(S_{\text{in}}, \Delta^{\text{AVG}})$ and $M^{\text{GAD}}(S_{\text{in}}, \Delta^{\text{AVG}})$, respectively. We define the competitive

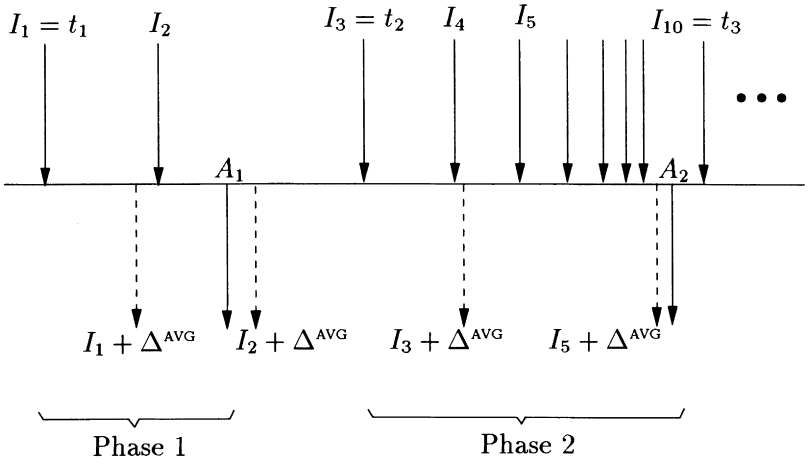

Fig. 11. Phase definition.

ratio of *GreedyMaxDelay* versus *GreedyAverageDelay* as

$$r_{GAD}^{GMD} = \max \frac{M^{GMD}(S_{in}, \Delta^{AVG})}{M^{GAD}(S_{in}, \Delta^{AVG})},$$

for every (S_{in}, Δ^{AVG}) pair such $M^{GAD}(S_{in}, \Delta^{AVG}) > 0$.

Lemma 6.1. *The competitive ratio of GreedyMaxDelay versus GreedyAverageDelay is O (log k).*

Proof. Let $\{A_1, \ldots, A_l\}$ be the output schedule generated for a pair (S_{in}, Δ^{AVG}) by *GreedyAverageDelay*. Let $t_1 = I_1$, and $t_i = \min\{I_j | I_j > A_{i-1}\}$ for $1 < i \leq l$. Define l phases in the considered slot, where the ith phase starts at t_i and ends at A_i (see Fig. 11). Note that these phases do not cover the whole time axis. They do cover, however, all the arrival times of incoming messages.

We shall now show that *GreedyMaxDelay* sends at most $\lfloor H_k \rfloor$ messages during every phase, where $H_i \triangleq \sum_{j=1}^{i} \frac{1}{j}$ is the ith harmonic number. These messages are depicted by dashed arrows in Fig. 11

As a first step, we show that every phase is not longer than $H_k \Delta^{AVG}$. This is proved by showing that the ith incoming message in each phase schedules the timer of *GreedyAverageDelay* to expire no later than $H_i \Delta^{AVG}$ from the beginning of the phase. The proof is by induction on i. The basis is trivial. Suppose the claim holds for some i, and consider now the case for $i + 1$. By the induction hypothesis, the ith message sets the timer to expire at $t' \leq H_i \Delta^{AVG}$. Message $i + 1$. arrives at $t \leq t'$, because otherwise *GreedyAverageDelay* would send an outgoing message before its arrival, defining the end of the phase. When this message arrives, the timer expiration is rescheduled to

$$t' + \frac{\Delta^{AVG} - (t' - t)}{i + 1} \leq H_i \Delta^{AVG} + \frac{\Delta^{AVG}}{i + 1} = H_{i+1} \Delta^{AVG}.$$

Since at most k messages arrive during a phase, the last of them schedules the timer to expire no later than $H_k \Delta^{AVG}$ from the beginning of the phase. This is the latest time an outgoing message can be sent by *GreedyAverageDelay*. *GreedyMaxDelay* sends at most $\lfloor H_k \rfloor$ messages during the phase, because every two consecutive messages it sends are spaced by at least Δ^{AVG}. Unlike *GreedyAverageDelay* which, by definition, does not send messages between the defined phases, *GreedyMaxDelay* sends at most one message. This is because otherwise the phases would not cover all the arrival times of incoming messages, since there is an UPDATE arrival between every two outgoing messages sent by *GreedyAverageDelay*. Hence, every message sent by *GreedyAverageDelay* is "covered" by at most $\lfloor H_k \rfloor + 1 = O(\log k)$ messages of *GreedyMaxDelay*, and the claim holds. □

Lemma 6.2. *The competitive ratio of GreedyMaxDelay versus GreedyAverageDelay is Ω(log k).*

Proof. Suppose that $\Delta^{AVG} = 1$. Consider an input schedule $S_{in} = \{I_0, \ldots, I_{k-1}\}$, where $I_0 = 0$ and for $1 \leq i < k$ $I_i = H_i^-$.

Consider a division of the slot into sub-slots of one unit duration. If *GreedyMaxDelay* starts the timer countdown at sub-slot i after receiving some UPDATE message, then an outgoing message will be sent at sub-slot $i + 1$. Since at least one UPDATE arrives at every sub-slot $0, \ldots, \lceil H_{k-1} \rceil$, *GreedyMaxDelay* sends at least one outgoing message every two consecutive sub-slots from the above set. Therefore, the total number of outgoing messages sent by *GreedyMaxDelay* is:

$$M^{GMD}(S_{in}, \Delta^{AVG}) \geq \frac{\lceil H_k \rceil}{2} = \Omega(\log k).$$

Consider now the output schedule generated by *GreedyAverageDelay* on the same input schedule. After the ith message is received at H_i^-, the timer is rescheduled to expire at $(H_i + (1/i + 1) = H_{i+1})$. In other words, every

incoming message reschedules the timer to expire immediately after the next incoming message arrives. The scheduler therefore sends a single message at H_k, and the claim holds. □

Theorem 6.2. *GreedyMaxDelay is $\Theta(\log k)$-competitive versus GreedyAverageDelay.*

Proof. By Lemmas 6.1 and 6.2.

The competitive ratio of *GreedyMaxDelay* and *GreedyAverageDelay* versus the optimal algorithm can be defined similarly. Observe that each of the two algorithms sends no more that k outgoing messages, whereas the optimal algorithm sends at least one message for any input schedule of size k. Consequently, the competitive ratio of both *GreedyMaxDelay* and *GreedyAverageDelay* versus the optimum is $O(k)$. It can be proved, however, that the competitive ratio of any deterministic on-line algorithm for this problem is $\Omega(\sqrt{k})$. □

7. Conclusions and future research

In this paper, we have defined a general model for scheduling the transmission of control messages that embraces many widespread network protocols. We have addressed two major scheduling problems in the context of this model. The first is the problem of minimizing the average extra delay a received message encounters under a constraint on the number of outgoing messages a node can send during a fixed period of time. The second problem is minimizing the number of outgoing messages under a constraint on the maximum or average delay.

A naive exponential-time algorithm for the off-line variant of the first problem has been presented. This algorithm has been modified into a polynomial-time algorithm using dynamic programming. Several heuristics for the on-line version of the problem have been proposed and compared: *TokenAlgorithm*, *BalanceAlgorithm*, and *VirtualClock*. The latter can achieve better performance if the mean input schedule size is known in advance.

A simple on-line algorithm, called *GreedyMaxDelay*, has been presented for minimizing the number of outgoing messages under the maximum delay constraint. Finally, a polynomial-time off-line algorithm for solving the same problem under the average delay constraint has been proposed, and two on-line heuristics, *GreedyMaxDelay* and *GreedyAverageDelay*, have been presented and compared.

In this paper, we have concentrated on local optimization only. We do believe that local optimization solutions can be employed in order to achieve global optimization, namely minimizing overall delays for a fixed number of control messages. However, the relationship between local and global optimizations in the general case is nontrivial, and it is therefore left for a future work.

Appendix A. Performance analysis of TokenAlgorithm

This appendix analyzes the performance of *TokenAlgorithm*. It is assumed that the incoming UPDATE messages are performed by a Poisson process with a mean of k messages in a slot, or $(k/(M+1))$ messages in a sub-slot.

Let Δ^{AVG} be a random variable denoting the average extra delay, $\Delta^{i,\mathrm{AVG}}$ be a random variable denoting the average extra delay in sub-slot i, A be a random variable denoting the number of UPDATE messages arriving during a sub-slot, and T^i be a random variable denoting the number of tokens at the beginning of sub-slot i. The probability of j arrivals during a sub-slot is

$$a_j = P[A=j] = \frac{e^{-k/(M+1)}\left(\frac{k}{M+1}\right)^j}{j!},$$

and the probability of more than j arrivals is

$$a_{j^+} = P[A>j] = 1 - \sum_{l=0}^{j} a_l.$$

Denote by t_j^i the probability of having j tokens at the beginning of sub-slot i, where $0 \le i \le M$. By the algorithm's definition, $t_j^i = 0$ for $j > i$, whereas for $i \ge j > 0$

$$t_j^i = P[T^i = j] = t_{j-1}^{i-1}a_0 + t_j^{i-1}a_1 + \cdots + t_{j-1}^{i-1}a_{i-j}$$

$$= \sum_{l=j-1}^{i-1} t_l^{i-1}a_{l-j+1}. \tag{A1}$$

The probability of having 0 tokens when sub-slot $i > 0$ starts is:

$$t_0^i = t_0^{i-1}a_{0^+} + t_1^{i-1}a_{1^+} + \ldots t_{i-1}^{i-1}a_{(i-1)^+} = \sum_{l=0}^{i-1} t_l^{i-1}a_{l^+}. \tag{A2}$$

Starting with $t_0^0 = 1$ and using Eqs. (A1) and (A2), we obtain t_j^i for every (i,j) pair.

Suppose there are j tokens in the system at the beginning of sub-slot i. Suppose l UPDATES arrive during this sub-slot. If $l \le j$, each UPDATE encounters no delay. Hence,

$$E[\Delta^{i,\mathrm{AVG}} | (T^i = j) \wedge (A = l \le j)] = 0.$$

Otherwise, the last $(l-j)$ UPDATES will wait until the end of the sub-slot. Since we consider a Poisson model of message arrival, the last $(l-j)$ messages are expected to arrive in the last $(l-j)/l$ interval of the sub-slot. Therefore, each of these messages encounters an expected delay of $1/2((l-j)/l) \times (\tau/(M+1))$. Consequently, we obtain:

$$E[\Delta^{i,\mathrm{AVG}} | (T^i = j) \wedge (A = l > j)]$$

$$= \frac{\frac{\tau}{2(M+1)}\left(1 - \frac{j}{l}\right)(l-j)}{l} = \frac{\tau}{2(M+1)}\left(1 - \frac{j}{l}\right)^2.$$

The extra delay is well defined only if at least one message

is received during a sub-slot. The probability of receiving $l > 0$ UPDATES during a sub-slot given that the sub-slot is non-empty is

$$p[A = l > 0 | A > 0] = \frac{a_l}{a_{0^+}}.$$

Hence,

$$E[\Delta^{\mathrm{i,AVG}} | T^i = j] = \frac{\tau}{2(M+1)} \sum_{l=j+1}^{\infty} \left(1 - \frac{j}{l}\right)^2 \frac{a_l}{a_{0^+}},$$

and

$$E[\Delta^{i,AVG}] = \frac{\tau}{2(M+1)} \sum_{j=0}^{i} t_i^j \sum_{i=j+1}^{\infty} \left(1 - \frac{j}{l}\right)^2 \frac{a_l}{a_{0^+}}.$$

Since the incoming message generation process is Poisson, an UPDATE message has an equal probability to be received in every sub-slot, namely $(1/(M+1))$. For every $0 \le i \le M$, a message arriving at sub-slot i encounters an expected extra delay of $E[\Delta^{\mathrm{i,AVG}}]$. Hence

$$E[\Delta^{\mathrm{AVG}}] = \frac{\sum_{i=0}^{M} E[\Delta^{i,\mathrm{AVG}}]}{M+1}.$$

These results give the same performance curve shown in Fig. 6

References

[1] A. Alles, ATM internetworking, Engineering InterOp, Las Vegas, March 1995.

[2] ATM Forum PNNI SWG 94-0471R13. ATM Forum PNNI Draft Specifications, April 1994.

[3] B. Awerbuch, D. Peleg. Concurrent on-line tracking of mobile users, SIGOMM, September 1991, pp. 221–233.

[4] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, On-line scheduling in the presence of overload, 32nd Symposium on Foundations of Computer Science, 1991, pp. 100–110.

[5] R. Braden, L. Zhang. S. Berson, S. Herzog, S. Jamin, Resource reservation protocol (RSVP), RFC-2205, September 1997.

[6] D. Clark, S. Shenker, L. Zhang, Supporting real-time applications in an integrated service packet network: Architecture and mechanisms, SIGCOMM, 1992.

[7] J.J. Garcia-Luna-Aceves, Loop-free routing using diffusing computations, IEEE Transactions on Networking 1 (1) (1993).

[8] G. Goldszmidt, Distributed system management via elastic servers, IEEE First International Workshop on Systems Management, Los Angeles, CA, April 1993, pp. 31–35.

[9] C. Huitema, Routing in the Internet, Prentice-Hall, Englewoods Cliffs, NJ, 1995.

[10] C.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in a hard real time environment, JACM 20 (1) (1973) 46–61.

[11] K.-S. Lui, S. Zaks, Scheduling in synchronous networks and the greedy algorithm, WDAG, 1997.

[12] K. Meyer, M. Erlinger, J. Betserand, C. Sunshine, G. Goldszmidt, Y. Yemini, Decentralizing control and intelligence in network management, The Fourth International Symposium on Integrated Network Management, May 1995.

[13] W. Richard Stevens, TCP/IP Illustrated, Addison-Wesley, Reading, MA, 1994.

[14] J.K. Strosnider, J.P. Lehoczky, L. Sha, The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments, IEEE Transactions on Computers 44 (1) (1995) 73–89.

[15] T.-S. Tia, J.W.-S. Liu, M. Shankar, Algorithms and optimality of scheduling soft aperiodic request in fixed-priority preempted systems, Journal of Real-Time Systems 10 (1): 23–43, January 1996.

[16] L. Zhang, Virtual clock: A new traffic control algorithm for packet switching networks, ACM Transactions on Computer Systems 9 (2) (1991) 101–124.