Real-Time Communication in FieldBus Multiaccess Networks

Ching-Chih Han and Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, MI 48109-2122
{*cchan,kgshin*}*@eecs.umich.edu*

Abstract

There has been an increasing need of timely and predictable communication services for embedded real-time systems in automated factories and industrial process controls. Work has been done on real-time communication with deadline guarantees in point-to-point, token bus/token ring/FDDI, and DQDB (Distributed Queue Dual Bus) networks. However, due to the random access nature of the CSMA/CD type multiaccess networks, they are not suitable for applications with stringent timing constraints. In this paper, we consider real-time communication services with absolute deadline guarantees in multiaccess local area networks equipped with a centralized scheduler, such as the SP-50 FieldBus [1], an industrial standard protocol for process control and manufacturing applications.

Similar to most token-passing networks, in a centralized-scheduling multiaccess network, the access to the bus is controlled by a token. Only the station currently holding the token has the exclusive right to use the multiaccess bus. Unlike the token bus, token ring, or FDDI network, the multiaccess network uses a centralized token scheduling scheme and the token need not be allocated to the stations in a cyclic fashion. We show that the pinwheel [2] and the distance-constrained [3] scheduling techniques can be adapted to schedule the token in centralized-scheduling multiaccess networks to guarantee message deadlines.

1 Introduction

There has been an increasing need of timely and predictable communication services for embedded real-time systems in automated factories and industrial process controls. For example, an automated factory is usually composed of several workcells, each of which contains devices such as robots, sensors, and transport mechanisms. All devices in a workcell are connected via a local area network. Multiple workcells are then connected by bridges. A number of cooperating tasks collectively monitor and control manufacturing equipment and processes by communicating with one another via the underlying network. The ability to provide timely and predictable inter-process communication is, thus, of great importance to the underlying network architecture and protocol because failure to meet the message-transmission deadlines may lead to a disaster.

Several researchers have investigated the problem of guaranteeing the timely delivery of messages under different network architectures and protocols. The *real-time channel* concept originally proposed by Ferrari and Verma [4] for the problem of meeting message-transmission deadlines in a wide area point-to-point network has been widely studied [5, 6]. These studies are mainly concerned with the problem of establishing real-time point-to-point channels and providing guarantees of maximum delivery delays. For local area network, IEEE 802.4 token bus network [7], IEEE 802.5 token ring network [8], and FDDI [9] adopt the *timed-token medium access control* (MAC) protocol for providing bounded medium access times. Agrawal *et al.* [10, 11] and Han *et al.* [12, 13] attempted to solve the *synchronous bandwidth allocation* problem for FDDI networks to meet the protocol constraint while transmitting all synchronous messages before their deadlines. Another protocol which aims to provide time-constrained communication services is the DQDB (Distributed Queue Dual Bus) MAC protocol [14]. DQDB has been adopted by the IEEE as its candidate protocol for metropolitan and local area networks. Saha *et al.* [15] and Han *et al.* [16] studied the issue of guaranteeing the timely delivery of *isochronous* messages with hard deadlines in a DQDB network. However, due to the random access nature of the multiaccess networks that adopt the CSMA/CD (IEEE 802.3) protocol, it is hard to make deadline guarantees for these kinds of networks. Hence, CSMA/CD type multiaccess networks are not suitable for applications requiring absolute deadline guarantees.

In this paper, we consider real-time communication services with absolute deadline guarantees in centralized-scheduling multiaccess local area networks. Similar to most token-passing networks, in the centralized-scheduling multiaccess network, the access to the bus is controlled by a token. Only the station

The work reported in this paper was supported in part by the ONR under Grants N00014-92-J-1080 and N00014-94-1-0229, and by the NSF under Grant MIP-9203895.

that holds the token is allowed to transmit messages on the bus. Unlike a token bus, token ring, or FDDI network, which uses a distributed token-passing protocol, we propose a centralized token scheduling scheme for the multiaccess network. In order to guarantee that each station on the multiaccess bus is allocated sufficient bandwidth for transmitting its time-critical messages, a central controller is used to control the allocation and scheduling of the token. Since the central controller has the full control of the token, it has more flexibility to achieve the goal of allocating bandwidth to real-time traffic for meeting message deadlines. Although our proposed token scheduling scheme is not restricted to any particular multiaccess network and can be used in most centralized-scheduling multiaccess networks that meet some minimum architectural requirements (to be described in Section 2), to facilitate our discussion, in the following we briefly describe the most relevant features of a particular network — the SP-50 FieldBus [1], which is currently being studied by an Instrument Society of America (ISA) Standards Committee and will soon become an international standard to support time-critical communications between automation system devices in industrial control and manufacturing systems.

The entire network of the FieldBus is composed of several *links*, each of which is a multiaccess bus connecting all the devices in a workcell. These multiaccess buses are further connected via bridges. In order to reduce communication latencies, unlike the OSI seven layer model, the FieldBus has only three layers: physical layer, data link layer, and application layer. In the data link layer of FieldBus, a Data Link Entity (DLE) is a logically active object, such as a copy of the executing program, which can send/receive packets to/from the interconnection network and acts according to the data link layer protocol of FieldBus. Therefore, there could be more than one DLE on a station/node which is physically attached to the network. There are four classes of DLEs in the FieldBus data link layer: Basic, Link Master (LM), Link Active Scheduler (LAS), and Bridge. Basic and LM classes are conceptually the same, except that the Basic class DLEs have only the minimum functions which are absolutely necessary for adequate operations on a FieldBus network, while the LM class DLEs are equipped with more functions such as that of cooperating with other LMs on the same link in establishing and sharing the link mastership. Unlike other popular timed-token protocols (e.g., token bus, token ring, and FDDI), FieldBus has a central control unit, the LAS DLE, for each link (multiaccess bus). There is always a copy of LAS DLE physically residing in the same node with each LM, and hence, each LM is capable of being a LAS. For each link, exactly one LAS is active at any time for scheduling messages on the link. It receives and responds to scheduling requests from all DLEs on the same link by allocating a token to one of these DLEs which then assumes the exclusive right to use the link over some time period specified in the token. The token is returned upon completion of its use, or assumed upon its expiration. That is, the LAS DLE is responsible for allocating and scheduling the token for real-time messages on the local link by sending the token, according to a scheduling scheme, to the next scheduled station with a specified time period during which the station can hold the token. There is at least one active LM on each link, which is responsible for detecting and recovering from the failure of the LAS. The active LMs contend to become the active LAS at the initialization or upon detecting the absence of the LAS. A Bridge DLE, which acts just like a normal LM DLE within a single link, performs a store-and-forward function to connect two or more separate multiaccess links (the function of the Bridge DLEs is out of the scope of this paper).

Due to the nature of workcells in an automated factory, most of time-critical communication is likely to take place between two peer DLEs on the same link, and hence, it can be handled by the local LAS. In this paper, we focus on the design of a token scheduling scheme that can be implemented in the LAS (or called the *link control unit*, LCU, in the following discussion) to allocate and schedule the token in such a way that each DLE on the local link will be allocated sufficient link bandwidth for guaranteeing the timely delivery of its real-time messages.

The rest of the paper is organized as follows. In Section 2, we describe the underlying network model and the real-time traffic characteristics. In Section 3, we propose a token scheduling scheme for real-time messages in a centralized-scheduling multiaccess network. In particular, we show how the *pinwheel* [2] and the *distance-constrained* [3] scheduling schemes can be adapted for the real-time communication problem addressed in this paper without considering the token dispatch overhead. In Section 4, we show how to incorporate the token dispatch overhead into the scheduling scheme proposed in Section 3. We conclude the paper with Section 5.

2 Network and message models

2.1 Network model

The local area network considered in this paper consists of N stations/nodes connected via a multiaccess link/bus with a central *link control unit* (LCU). The stations communicate with one another via the multiaccess bus. The stations' access to the bus is controlled by the LCU, which uses a token dispatch protocol for medium access control. The stations that have messages to transmit on the bus must first get the token from the LCU. When sending the token to a station, the LCU also specifies the duration, called the *Token Holding Time* (THT), that the station can hold the token for transmitting its messages. A station, after receiving the token, is entitled to transmit its messages on the bus for up to THT units of time. Either upon completion of its message transmission the station returns the token to the LCU, or when the THT expires the LCU generates a new token and sends the token to the next scheduled station. Messages to be transmitted on the bus are divided into fixed-length packets, or in ATM (Asynchronous Transfer Mode) term, cells. The transmission on the bus is slotted, i.e., data bits transmitted on the bus are divided into fixed-length slots. Each slot can hold one packet/cell and other information bits (e.g., the source/destination addresses, the framing bits, etc), i.e., each packet needs one slot time for its transmission. All stations listen to the bus all the time. If a station receives a packet destined for it, it stores the packet in its internal buffer; otherwise, it just discards the packet.

There are two salient differences between our centralized-scheduling multiaccess network and other token-passing networks such as token bus, token ring, or FDDI. First, in our network model, the token scheduling is controlled by a central LCU, while token bus/token ring/FDDI adopts a distributed timed-token protocol. Second, in our network model, the token need not be allocated to stations in a cyclic fashion as in token bus/token ring/FDDI. Note that our network model is compatible with the current draft proposal of the FieldBus protocol [1]. Therefore, the proposed token scheduling scheme to be discussed in the following sections can be readily incorporated in the FieldBus protocol.

2.2 Message model

Each station on the multiaccess bus may have real-time and/or non-real-time messages to transmit. Non-real-time messages do not have any timing constraints, while each real-time message belongs to a real-time message stream, which possesses some predefined characteristics, including the deadline of each message. Each station may have zero, one, or more real-time message streams emanating from it. Let $\mathbf{M} = \{M_1, M_2, \ldots, M_n\}$ be a set of n real-time streams in the multiaccess network. We consider the following message model, in which each stream M_i is characterized by a tuple (C_i, D_i), where

- C_i is the maximum number of packets (cells) in stream M_i that can arrive in *any* time interval of length D_i, and
- D_i is the transmission deadline (or simply, the deadline) for the messages in stream M_i, i.e., if a message of M_i arrives at time t, then it must be transmitted by time $t + D_i$.

This model is a generalization of the commonly-used real-time *peak-rate* message model [17], in which each stream M_i is characterized by a triple (C_i, D_i, P_i), where

- P_i is the minimum inter-arrival period for stream M_i, i.e., if the j-th message of M_i arrives at time t, then the $(j+1)$-th message in the stream will arrive at a time no earlier than $t + P_i$ for $j \geq 1$,
- C_i is the maximum message size measured in packets (cells) in stream M_i, i.e., C_i is the number of slots needed to transmit a maximum-size message in stream M_i, and
- D_i $(\leq P_i)$ is the transmission deadline for the messages in stream M_i.

Note that in the first message model, the inter-arrival time of two consecutive messages in stream M_i is not required to be larger than or equal to D_i (i.e., more than one message may arrive in a time interval of length $\leq D_i$). However, the total message size measured in packets (cells) in M_i that arrive in *any* time interval of length D_i should be bounded by C_i. In the second model, during any time interval of length P_i, at most one message with message size at most C_i will arrive. And, $D_i \leq P_i$ implies that the total message size in M_i that arrive in any time interval of length D_i is bounded by C_i. It is easy to see that the second message model is just a special case of the first one. Thus, unless otherwise specified, we will assume that real-time message streams conform to the first model.

For convenience of discussion, we will henceforth call D_i the deadline (constraint), and C_i the (maximum) message size (within a time interval of length D_i) of stream M_i. Moreover, without loss of generality, we assume that the time unit is one slot, D_i is measured in slot times, and the message arrival times align with the beginning of a slot. Note that as mentioned earlier, packet size matches the payload size of a slot, so we can also think of C_i measured in slots.

In order to guarantee the timely delivery of real-time messages, the LCU must allocate sufficient bandwidth to each real-time message stream. For the two message models defined above, it is easy to see that if we can guarantee that

(P1) during any time interval of length D_i the LCU will allocate the token to stream M_i and let M_i hold the token for (at least) C_i units of time,

then we can guarantee the deadline constraint of any message in stream M_i. Therefore, in the following discussion, we will concentrate on how to generate a token allocation schedule so that the above (P1) is satisfied. Especially, we propose a *centralized-scheduling multiple access* (CS/MA)[1] protocol for our multiaccess network. The CS/MA protocol discussed in this paper uses a centralized controller to allocate the token to stations/nodes such that conflict-free multiple access can be achieved and timely delivery of the real-time messages can be guaranteed.

3 Proposed token scheduling scheme

The proposed token scheduling scheme is based on the *pinwheel* [2] and the *Distance-Constrained* (DC) [3] scheduling techniques. In this section, we first briefly describe the pinwheel and the DC scheduling problems and their scheduling schemes. Especially, we will describe the Schedulers **Sx** and **Sr** originally designed for scheduling pinwheel instances and distance-constrained task sets, respectively. We then show how **Sx**/**Sr** can be modified to schedule the token for real-time message streams on the stations of a multiaccess network.

3.1 Pinwheel and distance-constrained scheduling schemes

The theoretical base of our token scheduling scheme is grounded on some of the results in the pinwheel and the DC scheduling problems.

The Pinwheel Problem: ([2, 18]) Given a multiset of n positive integers $\mathbf{A} = \{a_1, a_2, \ldots, a_n\}$, find an infinite sequence (schedule) over the symbols $\{1, 2, \ldots, n\}$ such that there is at least one symbol "i" within any subsequence of a_i consecutive symbols (slots). □

For example, given a multiset $\mathbf{A} = \{2, 4, 5\}$, one solution sequence is $(1, 2, 1, 3, 1, 2, 1, 3, \cdots)$ where the subsequence $(1, 2, 1, 3)$ repeats forever. In this solution sequence, we can find one "1" in every $a_1 = 2$ consecutive symbols, one "2" in every $a_2 = 4$ consecutive symbols, and (at least) one "3" in every $a_3 = 5$ consecutive symbols.

The question of how to schedule a pinwheel instance has been studied in [2, 19]. Define $\rho(\mathbf{A}) = \sum_{i=1}^{n} 1/a_i$ to be the (*total*) *density* of the pinwheel instance $\mathbf{A}$. Holte *et al.* [18] have shown that if a pinwheel instance $\mathbf{A}$ with total density ≤ 1 consists solely of multiples (i.e., a_i (evenly) divides a_j for all $i < j$, and $\rho(\mathbf{A}) = \sum_{i=1}^{n} 1/a_i \leq 1$), then $\mathbf{A}$ is schedulable. For convenience of reference, we list this result in the following theorem.

Theorem 1: ([18]) Given a pinwheel instance $\mathbf{A} = \{a_1, a_2, \ldots, a_n\}$, if a_i divides a_j for $i < j$, and $\rho(\mathbf{A}) \leq 1$, then $\mathbf{A}$ is schedulable. □

Based on this result, Chan and Chin [2] have devised two schedulers, **Sa** and **Sx**, to schedule larger classes of pinwheel instances. The basic idea of **Sa** and **Sx** is the *single-integer reduction* technique, which aims to transform an arbitrary instance $\mathbf{A}$ to another instance $\mathbf{B} = \{b_1, b_2, \ldots, b_n\}$ which consists solely of multiples and $b_i \leq a_i$ for all i. From Theorem 1, we know that $\mathbf{B}$ can be feasibly scheduled (for example, by the algorithm **SpecialSingle** in [2]) if and only if $\rho(\mathbf{B}) \leq 1$. Since $b_i \leq a_i$ (i.e., $\mathbf{B}$ is more restricted than $\mathbf{A}$), if we find a schedule for $\mathbf{B}$, then the schedule also satisfies the original constraints for $\mathbf{A}$. However, since $b_i \leq a_i$ for all i, $\rho(\mathbf{B}) \geq \rho(\mathbf{A})$. Therefore, if the total density of $\mathbf{A}$ is larger than 1, it is impossible to find a feasible schedule for $\mathbf{A}$, i.e., total density less than or equal to 1 is a necessary condition for an instance to be schedulable. The density threshold ρ^* of $\mathbf{A}$ is then derived in such a way that as long as the total density of $\mathbf{A}$ is less than or equal to ρ^* then $\rho(\mathbf{B}) \leq 1$ (i.e., $\mathbf{B}$ is schedulable). In other words, with the single-integer reduction technique, one can schedule all pinwheel instances with total densities $\leq \rho^*$. Note, however, that if a pinwheel instance $\mathbf{A}$ has a total density larger than ρ^*, it does not necessarily mean that the instance is not schedulable by Scheduler **Sa** or **Sx**. $\mathbf{A}$ can be feasibly scheduled as long as the total density of the transformed set $\mathbf{B}$ is less than or equal to 1.

Without loss of generality, in the following discussion, we assume that $a_1 \leq a_2 \leq \cdots \leq a_n$. Let a denote the smallest number in $\mathbf{A}$, i.e., $a = a_1$. In Scheduler **Sa**, it finds a b_i for each a_i such that

$$b_i = a \cdot 2^j \leq a_i < a \cdot 2^{j+1} = 2b_i,$$

for some integer $j \geq 0$. Chan and Chin [2] call this operation *specializing* $\mathbf{A}$ *with respect to* $\{a\}$. Since the instance $\mathbf{B} = \{b_1, b_2, \ldots, b_n\}$ consists solely of multiples, as long as $\rho(\mathbf{B}) \leq 1$, **Sa** can then use the algorithm **SpecialSingle** [2] to find a feasible schedule for $\mathbf{B}$. And, since $b_i \leq a_i$ for all i, the schedule found for $\mathbf{B}$ is also a feasible schedule for $\mathbf{A}$.

Scheduler **Sx** is based on the same technique as Scheduler **Sa** except that $\mathbf{A}$ is specialized with respect to $\{x\}$, where x is an integer and $a_1/2 < x \leq a_1$. Starting from $x = a_1$, **Sx** specializes $\mathbf{A}$ with respect to $\{x\}$ until $x \geq a_1/2 + 1$ and chooses an x that minimizes

[1] To distinguish the type of protocol discussed in this paper from the well-known *carrier sense multiple access* (CSMA) protocol, we use CS/MA as the acronym for our protocol.

$\rho(\mathbf{B})$, or until it finds an x which makes $\rho(\mathbf{B}) \leq 1$ (or until it finds that no such integer exists). Note that finding an x that minimizes $\rho(\mathbf{A'})$ can be done in $O(n)$ time [2]. Therefore, **Sx** is more powerful than **Sa** in the sense that every pinwheel instance that can be scheduled by **Sa** can also be scheduled by **Sx**. For example, **Sa** specializes $\mathbf{A} = \{4, 7, 8, 13, 24, 28\}$ (with a total density of $0.672 \cdots \approx 2/3$) with respect to $\{4\}$ to get $\mathbf{B} = \{4, 4, 8, 8, 16, 16\}$ with a total density of 7/8. In comparison, **Sx** specializes **A** with respect to $\{3\}$ to get $\mathbf{B'} = \{3, 6, 6, 12, 24, 24\}$ with a total density of 5/6 ($< 7/8$).

It has been shown in [2] that the density thresholds for Schedulers **Sa** and **Sx** are 1/2 and 13/20, respectively. That is, as long as the total density of **A** is less than or equal to 1/2, the total density of the resulting set after specializing **A** with respect to $\{a\}$ will be less than or equal to 1, and hence, the resulting specialized set is schedulable (and so is the original set **A**). Similarly, as long as the total density of **A** is less than or equal to 13/20, the minimum total density of the resulting sets after specializing **A** with respect to $\{x\}$, for $a_1/2 < x \leq a_1$, will be less than or equal to 1.

Distance-constrained task system model. In [3] we proposed a new real-time system model, called the *Distance-Constrained Task System* (DCTS), to characterize real-time tasks that have *temporal distance constraints* [20]. In the conventional real-time task system model [21], it is assumed that every task must be executed once during a certain fixed period. The execution of a task in one period is independent of the execution of the same task in any other period. In the DCTS model, we assume that two consecutive executions of the same task must be "close" to each other. Specifically, given a DCTS task set $\mathbf{T} = \{T_1, T_2, \ldots, T_n\}$, where each task T_i has an execution time C_i and a (temporal) distance constraint D_i, if f_{ij} denotes the finish time of the j-th execution/invocation of task T_i, then the distance constraint D_i for T_i requires that $f_{i1} \leq D_i$ and $f_{i,j+1} - f_{ij} \leq D_i$ for all $j \geq 1$. In [3], we proposed a scheduling algorithm, Scheduler **Sr**, for DCTS task sets. Given a DCTS task set $\mathbf{T} = \{T_i = (C_i, D_i) \mid 1 \leq i \leq n\}$ with $D_1 \leq D_2 \leq \cdots \leq D_n$, Scheduler **Sr** first specializes the distance constraint (multi)set $\mathbf{D} = \{D_1, D_2, \ldots, D_n\}$ with respect to $\{r\}$, where r is a (real) number chosen from the interval $(D_1/2, D_1]$ such that it minimizes the total density of the specialized task set, where the density of a DCTS task set **T** is $\rho(\mathbf{T}) = \sum_{i=1}^{n} C_i/D_i$. Scheduler **Sr** then uses an approach similar to the *rate-monotonic* (RM) scheduling algorithm [21] to schedule the specialized DCTS task set whose distance constraint (multi)set consists solely of multiples. It has also been shown that as long as the total density $\rho(\mathbf{T})$ of the task set **T** is less than or equal to $n(2^{1/n} - 1)$, Scheduler **Sr** can guarantee to generate a feasible schedule for **T**. Note, again, that if $\rho(\mathbf{T}) > n(2^{1/n} - 1)$, it does not necessarily mean that **Sr** cannot generate a feasible schedule for **T**. As long as the total density of the task set after specializing **D** with respect to $\{r\}$ is less than or equal to 1, **Sr** can generate a feasible schedule for **T**. Another important property of Scheduler **Sr** is that during any time interval of length D_i, the scheduling algorithm will allocate C_i time units for task T_i for all i. Note that both D_i and C_i are not necessarily integral; they can be any positive real numbers.

As mentioned earlier, if the token scheduling scheme of the LCU can allocate (at least) C_i slots to stream M_i during any time interval of length D_i for all i, then all the messages are guaranteed to be transmitted before their deadlines. Therefore, we can think of the token scheduling problem as an extension of the pinwheel problem (C_i isn't necessarily equal to 1), and our token scheduling scheme uses a discrete version of the DCTS scheduling algorithm (both D_i and C_i are integers).

In what follows, we describe how to adapt the scheduling algorithms originally designed for the pinwheel and the DCTS scheduling problems to schedule the token so that within any time interval of length D_i, the token is guaranteed to be allocated to stream M_i for at least C_i slots.

3.2 TokenAllocator and TokenScheduler processes

We now describe in detail how to allocate the token to message streams. We first assume that token dispatch overhead is small (as compared to the message size) and can be ignored or is included in C_i's (note that token dispatch overhead corresponds to preemption/context-switching overhead in task scheduling). Later, we will relax this assumption and show how to deal with the token dispatch overhead.

Let $\mathbf{M} = \{M_i = (C_i, D_i) \mid 1 \leq i \leq n\}$ be a set of real-time message streams. Define the (*message*) *density* of stream M_i to be $\rho(M_i) = C_i/D_i$, and the (*total*) *density* of the set **M** to be $\rho(\mathbf{M}) = \sum_{i=1}^{n} C_i/D_i$. Given a set **M** of streams with D_i divides D_j for all $i < j$, and $\rho(\mathbf{M}) \leq 1$, the algorithm **TokenAllocator** (Figure 1) will allocate the token to the streams in such a way that during any time interval of length D_i, the algorithm will allocate (not necessarily consecutive) C_i time slots to stream M_i. The **TokenAllocator** process uses the rate-monotonic (RM) [21] scheduling algorithm to assign message priorities so that the streams with tighter deadline constraints get higher priorities. Specifically, **TokenAllocator** treats each stream $M_i = (C_i, D_i)$ as

TokenAllocator

```
/* d_i: slack time of the current message of M_i. */
/* c_i: remaining transmission time w.r.t. current d_i. */
/* send(P,message): send a message to process P
   and wait for its reception by P. */
/* receive(P,message): wait for and receive a message
   from process P. */
/* M = {M_i = (C_i, D_i) | 1 ≤ i ≤ n} is a set of message
   streams with D_i|D_j for all i < j and ρ(M) ≤ 1. */

1. receive(TokenScheduler, M);
/* note that D_1 ≤ D_2 ≤ ··· ≤ D_n */
2. for i := 1 to n do { c_i := C_i; d_i := D_i; }
3. do {
4.     i ← 1;
5.     while (i ≤ n and c_i = 0) { i := i + 1 }
6.     if i ≤ n then {
7.         H := min(c_i, d_1);
/* allocate H slots to M_i */
8.         send(TokenScheduler, i, H);
9.         c_i := c_i − H;
10.    } else {
11.        H := d_1;
/* allocate H slots to non-real-time traffic */
12.        send(TokenScheduler, 0, H);
13.    }
14.    for j := 1 to n do {
15.        d_j := d_j − H;
16.        if (d_j == 0) { c_j := C_j; d_j := D_j; }
17.    }
18. } forever
```

Figure 1: The **TokenAllocator** process.

TokenScheduler

```
/* Assume M = {M_i = (C_i, D_i) | 1 ≤ i ≤ n}, where
   D_1 ≤ D_2 ≤ ··· ≤ D_n. */

/* Upon system initialization */
1. collect M and the station id that each message
   stream emanates from;
2. specialize the deadline constraint multiset D of M
   to get D' (M');
3. if (ρ(M') > 1) reject M and exit;
4. else {
5.     send(TokenAllocator, M);
6.     do {
7.         receive(TokenAllocator,i, H);
8.         if (i ≠ 0) {
9.             send the token with THT H to the
               node containing stream M_i;
10.        } else {
11.            send a non-real-time token with THT H
               to the next scheduled node;
12         }
13.        wait for the token to expire or until the
           (non-real-time) token is returned early;
14.        if token is returned early by H' slots and
               H' − τ > 0 then {
15.            send a non-real-time token with THT
               H' − τ to the next scheduled node;
16.            goto Line 13;
17.        } /* if */
18.    } forever
19. }
```

Figure 2: The **TokenScheduler** process.

a periodic task with execution time C_i and period (= deadline) D_i. **TokenAllocator** always allocates the token to the stream with the highest priority among those *active* streams, where an active stream is one whose slot requirements are *unfulfilled* at the current period interval. In **TokenAllocator**, the while-loop (Line 5) is used to locate the highest-priority stream (i.e., the smallest index i) that has unfulfilled slot requirement (i.e., $c_i > 0$) with respect to the current d_i. After locating the highest-priority active stream M_i, **TokenAllocator** allocates $\min(c_i, d_1, d_2, \ldots, d_{i-1}) = \min(c_i, d_1)$ time slots to M_i. The **TokenScheduler** process (Figure 2) implements the token scheduling scheme which sends the token with token holding time H to the station that contains (the source of) the stream M_i (more on **TokenScheduler** will be discussed later). If there is no stream with the unfulfilled slot requirement (i.e., $c_i = 0$ for all i) with respect to the current d_i, the LCU issues a non-real-time token with a THT equal to the beginning time of the next closest scheduled activity minus the current time, i.e., $\min(d_1, d_2, \ldots, d_n) = d_1$ (Line 11 of **TokenAllocator**). Note that the index 0 in Line 12 represents that the token is to be sent to a station for transmitting its non-real-time messages. This non-real-time token can be issued to the stations on the network in a predefined fashion (such as round-robin).

Note that the **TokenAllocator** process will generate exactly the same schedule that the RM algorithm will generate given a periodic task set with the i-th task having a period D_i and an execution time C_i, and D_i divides D_j for all $i < j$.

Theorem 2: For a set of real-time message streams

$\mathbf{M} = \{M_i = (C_i, D_i) \mid 1 \leq i \leq n\}$, if D_i divides D_j for all $i < j$, and $\rho(\mathbf{M}) \leq 1$, then **TokenAllocator** will allocate C_i slots to stream M_i in any time interval of length D_i. □

The proof of the above theorem is omitted due to limited space.

Suppose the LCU sends a real-time token with a token holding time H slots to stream M_i which emanates from station N_k. If the total size of the real-time messages of M_i currently waiting to be transmitted is less than H slots (packets), then, after transmitting these messages of M_i, node N_k can use the remaining unused token holding time to transmit its non-real-time messages. However, whenever new messages of M_i arrive, the transmission of non-real-time messages should be preempted, and N_k should use the remaining THT to transmit the newly-arrived real-time messages of M_i. The token will not be returned to the LCU even when node N_k has neither real-time messages of M_i nor non-real-time messages to transmit because new messages of M_i might still arrive at N_k before the current token holding time expires. When the current token holding time expires, the LCU will generate a new token and send it to the next scheduled station.

However, if the LCU sends a non-real-time token with H slots of THT to a node and the node does not use up all H slots to transmit its non-real-time messages, the node should return the token to the LCU. After receiving the returned non-real-time token, the LCU sends the (non-real-time) token to the next scheduled node according to the predefined sequence with H' slots of THT, where H' is the amount of time (number of slots) by which the token is returned early.

We now describe the function of the LCU scheduler, the **TokenScheduler** process (Figure 2), which incorporates **TokenAllocator** to allocate token to the message streams. Note that in Lines 14 and 15 of **TokenScheduler**, τ is the token dispatch time to be discussed in the next section. Currently, we can simply think that $\tau = 0$ since the token dispatch overhead is ignored.

To schedule the token for a set of general message streams $\mathbf{M}$ (i.e., D_i may not evenly divides D_j for some i and j), the scheduler **TokenScheduler** does the following steps.

Step 1. Upon system initialization, gather/maintain the required information on real-time connection requests; in particular, (C_i, D_i) for each stream M_i and the station id that M_i emanates from.

Step 2. Specialize $\mathbf{D} = \{D_1, D_2, \ldots, D_n\}$ using the chosen specialization operation to get the specialized stream set **M'** with the specialized deadline constraint (multi)set **D'** $= \{D'_1, D'_2, \ldots, D'_n\}$. For example, if the specialization operation is the same as that used in Scheduler **Sx**, then we find D'_i for each D_i such that $D'_i = x \cdot 2^j \leq D_i < x \cdot 2^{j+1} = 2D'_i$, for some integer $j \geq 0$, where x is an integer $\in (D_1/2, D_1]$ that results in the minimum $\rho(\mathbf{M'})$. (Note that D'_i divides D'_j for all $i < j$.)

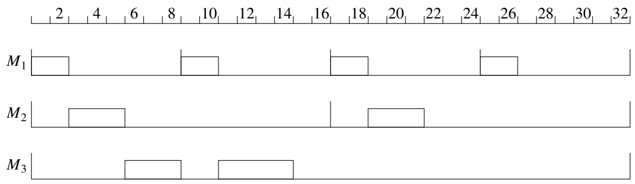


Figure 3: The token allocation schedule for the set of streams in Example 1.

Step 3. Check whether or not the total message density of the specialized stream set $\mathbf{M'} = \{M'_i = (C_i, D'_i) \mid 1 \leq i \leq n\}$ is less than or equal to 1, i.e., whether $\rho(\mathbf{M'}) = \sum_{i=1}^{n} C_i/D'_i \leq 1$, or not. If not, reject **M** and stop. Otherwise, proceed to the next step.

Step 4. Use the **TokenAllocator** process to get the stream id i and the token holding time H. If $i > 0$, assign the token to stream M_i (send the token to the station that M_i emanates from) with a token holding time H. If $i = 0$, send a non-real-time token with H slots of THT to the next node according to the predefined sequence.

Step 5. Wait for the expiration of the token (holding time), or the early return of the (non-real-time) token. (Note that the real-time token will not be returned.) If the token expires, repeat Step 4 for the next scheduled activity. If the (non-real-time) token is returned early by H' slots and $H' - \tau > 0$, send the token to the next node according to the predefined sequence with $H' - \tau$ slots of THT, where τ is the token dispatch time (to be discussed in the next section).

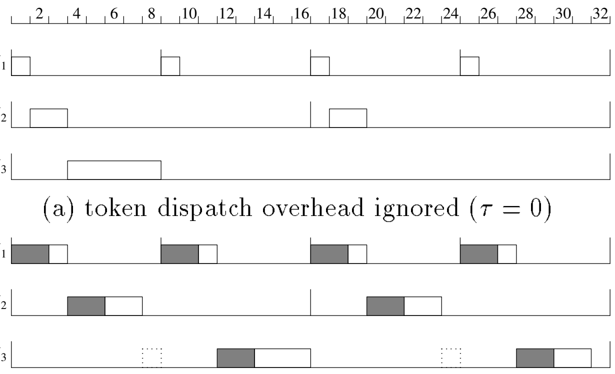
Example 1: Consider a set of real-time message streams $\mathbf{M} = \{M_i = (C_i, D_i) \mid i = 1, 2, 3\} = \{(2, 9), (3, 17), (7, 35)\}$. Assume that M_i emanates from node N_i for $i = 1, 2, 3$, respectively. Also, assume that non-real-time tokens are sent to the nodes in a round-robin fashion staring from node N_1. If we specialize the deadline constraint set $\mathbf{D} = \{9, 17, 35\}$ with respect to $\{8\}$, we will get the specialized deadline constraint set **D'** $= \{8, 16, 32\}$. Since D'_i divides D'_j for all $i < j$, and $\rho(\mathbf{M'}) = 2/8 + 3/16 + 7/32 = 21/32 < 1$, by Theorem 2, we know that **M'** is schedulable by **TokenAllocator**. The subschedule from slot 1 to slot 32 produced by **TokenAllocator** is shown in Figure 3. According to this schedule, the LCU will first send a token to node N_1 and let it hold the token for two time slots to transmit messages of M_1. When the token expires, the LCU will send another token to N_2 with 3 slots of THT. Then, upon expiration of the token, another token is sent to N_3 with 3 slots of THT. The next two steps are: send a token with 2 slots of THT to N_1, and then, send a token with 4 slots of THT to N_3. Now, since there is no real-time token scheduled for the next two slots, the LCU will send a non-real-time token to N_1 for transmitting its non-real-time messages, if any. (Note that non-real-time token allocation is not shown in the schedule of Figure 3.) The following steps are left for the reader. This subschedule repeats (virtually) forever.

It is easy to see that **TokenAllocator** will generate an infinite sequence of stream ids and token holding times for the set of real-time streams such that there are C_i time slots allocated to stream M_i in any time interval of length D'_i ($\leq D_i$). So, all real-time messages are guaranteed to be transmitted before their deadlines. □

4 Incorporating token dispatch overhead

In the previous discussion, we ignored the token dispatch overhead. We now show how to deal with the token dispatch overhead and how to incorporate it into the **TokenAllocator** process. We will use τ to denote the token dispatch time, i.e., it takes the LCU τ time slots to send/dispatch the token to a station. We first use an example to show the effect of the token dispatch overhead.

Consider a set of real-time streams $\{M_i = (C_i, D_i) \mid i = 1, 2, 3\} = \{(1,8), (2,16), (5,32)\}$. (Note that D_i divides D_j for $i < j$.) If the token dispatch overhead is ignored or included in C_i's, the schedule generated by the **TokenAllocator** process is shown in Figure 4(a). In the schedule, the token will be allocated to each stream M_i for C_i time slots within any time interval of length D_i slots. However, if the token dispatch overhead needs to be considered, some time slots must be used to send the token to the nodes that have messages to transmit. For example, if token dispatch needs two slots,[2] then time slots 1 and 2 must be used to send the token to the node that contains stream M_1. Note that the nodes do not return the real-time token to the LCU. The real-time token is assumed upon its expiration, and the LCU will issue a new token to the next scheduled node upon expiration of the current (real-time) token. So, no real-time token return overhead needs to be considered. However, as discussed earlier, a non-real-time token will be returned and sent to another node if the token does not expire and there is no non-real-time message to transmit on the node that currently holds the (non-real-time) token. The (non-real-time) token return overhead is easily taken care of by the **TokenScheduler** process in Figure 2. (Note that the new THT, H', in **TokenScheduler** is defined to be the amount of time the token is returned early, and hence, the token return time is not included in H'.)



(a) token dispatch overhead ignored ($\tau = 0$)

(b) token dispatch time $\tau = 2$

Figure 4: The effect of the token dispatch overhead.

In order to guarantee that during any time interval of length D_i, there must be at least C_i slots allocated to stream M_i, a schedule as shown in Figure 4(b) must be generated (assuming token dispatch to require two slots). In Figure 4(b), the shaded areas are the slots used to send the token to the nodes that contain the corresponding streams. The token with $C_1 = 1$ slot of THT is allocated to M_1 once every $D_1 = 8$ time slots. Each time the token is allocated to the node that contains M_1, two slots must be used to send the token. If we think of the token dispatch overhead as part of the message transmission time (message size), it is equivalent to allocating $C'_1 = C_1 + 2 = 3$ slots to M_1 within every time interval of length D_1. We will call C'_1 the *effective* message size of M_1. Also, the token with $C_2 = 2$ slots of THT is allocated to M_2 once every $D_2 = 16$ time slots. Slots 4 and 5 (20 and 21) are used to send the token to the node that contains stream M_2. Thus, the effective message size of M_2 is $C'_2 = 4$. Similarly, the token is allocated to M_3 twice with 3 and 2 slots ($C_3 = 5$) of THT, respectively, every $D_3 = 32$ slots. Slots 12 and 13 (and also slots 28 and 29) are used to send the token to the node that contains M_3. Note that the token is allocated to M_3 twice in the interval $[1, D_3]$ in Figure 4(b), instead of once as in Figure 4(a). Also, notice that slot 8 (and also slot 24) is not enough to transmit the token to the node that contains M_3, and

[2] The token dispatch time (τ) is, in general, much smaller than the time needed to transmit a message of size C_i. The numbers for τ and C_i's in this example are chosen for the purpose of illustration.

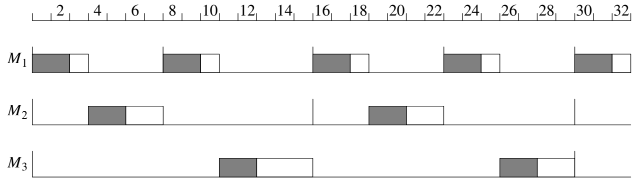


Figure 5: The "compressed" schedule.

thus, the LCU will not send the token to the node that contains M_3 at these two time instants. Instead, the LCU will leave time slot 8 (and 24) idle intentionally. Although we will later show that these idle slots can actually be "compressed" by "advancing" the schedule, we will still consider these idle slots (8 and 24) as the overhead caused by token dispatch. Thus, the effective message size of M_3 is $C'_3 = 11$, instead of $C_3 + 2 \cdot 2 = 9$. Note that slot 32 is also left idle intentionally. However, it is not considered as the token dispatch overhead for any real-time stream, and hence, it is not included in the effective message size of any real-time stream.

Given a set of streams $\mathbf{M} = \{M_i = (C_i, D_i) \mid 1 \leq i \leq n\}$ with D_i dividing D_j for all $i < j$, we define the total *effective* message density $\rho'(\mathbf{M})$ to be

$$\rho'(\mathbf{M}) = \sum_{i=1}^{n} \frac{C'_i}{D_i} = \sum_{i=1}^{n} \left(\frac{C_i + O_i}{D_i} \right),$$

where O_i is the total token dispatch overhead (including the slots left idle intentionally due to insufficient time for sending the token) for stream M_i in the interval $[1, D_i]$ by using **TokenAllocator**.

As mentioned earlier, we don't really need to leave the slots idle if they are not long enough to send the token to a station. We can just "compress" the schedule by sending the token to the next scheduled station. The compressed version of the schedule in Figure 4(b) is shown in Figure 5. It is easy to see that the compressed schedule still satisfies the requirement that during any time interval of length D_i, the token is allocated to M_i for at least C_i time slots (excluding the token dispatch overhead).

We now show how to modify the **TokenAllocator** process in Figure 1 to incorporate the token dispatch overhead. Let τ denote the token dispatch time. The modified **TokenAllocator** process which incorporates the token dispatch overhead is shown in Figure 6. Note that if $H \leq 0$ in Line 8 or Line 13, it means that $H = d_1 - \tau \leq 0$. Thus, in Line 16, $d_j := d_j - H - \tau$ is equivalent to $d_j := d_j - d_1$, which, in turn, is equivalent to compressing/advancing the schedule by d_1 slots. Also, notice that the token holding time H sent to the **TokenScheduler** process does not include the token dispatch time τ.

TokenAllocator (modified)

1. receive(**TokenScheduler**, $\mathbf{M}$);
/* Note that $D_1 \leq D_2 \leq \cdots \leq D_n$ */
2. **for** $i := 1$ **to** n **do** { $c_i := C_i$; $d_i := D_i$; }
3. **do** {
4. $\quad i \leftarrow 1$;
5. $\quad$ **while** ($i \leq n$ **and** $c_i = 0$) { $i := i + 1$ }
6. $\quad$ **if** $i \leq n$ **then** {
7. $\quad\quad H := \min(c_i, d_1 - \tau)$;
8. $\quad\quad$ **if** $H > 0$ **then** {
9. $\quad\quad\quad$ send(**TokenScheduler**, i, H);
10. $\quad\quad\quad c_i := c_i - H$; }
11. $\quad$ } **else** {
12. $\quad\quad H := d_1 - \tau$;
13. $\quad\quad$ **if** $H > 0$ **then** send(**TokenScheduler**, 0, H);
14 $\quad$ }
15. $\quad$ **for** $j := 1$ **to** n **do** {
16. $\quad\quad d_j := d_j - H - \tau$;
17. $\quad\quad$ **if** ($d_j == 0$) { $c_j := C_j$; $d_j := D_j$; }
18. $\quad$ }
19. } **forever**

Figure 6: The (modified) **TokenAllocator** process.

A theorem parallel to Theorem 2 can also be derived for the modified **TokenAllocator** process.

Theorem 3: For a set of real-time streams $\mathbf{M} = \{M_i = (C_i, D_i) \mid 1 \leq i \leq n\}$, if D_i divides D_j for all $i < j$, and $\rho'(\mathbf{M}) = \sum_{i=1}^{n} C'_i / D_i \leq 1$, then the (modified) **TokenAllocator** process in Figure 6 will allocate C_i (not C'_i) slots to stream M_i in any time interval of length D_i. □

The proof of the above theorem is omitted due to limited space.

Now, let's go back to the **TokenScheduler** process in Figure 2. In Line 14, if a token is returned early by H' slots, and if H' is long enough (i.e., $H' - \tau > 0$), then the LCU (the **TokenScheduler** process) will issue a non-real-time token with THT $= H' - \tau$ to the next scheduled station. The "$-\tau$" in the THT $H' - \tau$ is because the THT sent to a station does not include the token dispatch time. If the current token expires, or if the token is returned early but $H' - \tau \leq 0$, then the LCU just uses the **TokenAllocator** to schedule the next activity (in the latter case, it is equivalent to compressing the schedule by H' slots).

5 Conclusion

In this paper, we proposed a token scheduling scheme for allocating the bandwidth of a multiaccess bus and guaranteeing the timely delivery of real-time messages in a centralized-scheduling multiaccess network such as the soon-to-be industrial standard, FieldBus. Our proposed scheme is based on the pinwheel and the distance-constrained scheduling schemes. It is predictable since it can feasibly schedule a set of streams with total message density less than or equal to a certain density threshold (e.g., 0.65 for using the specialization operation of Scheduler **Sx**). It is also simple and easy to implement since it uses a fast on-line scheduling algorithm to decide which station the next token should be sent to and how much time the station can hold the token for transmitting its messages.

The most significant contribution of this paper comes from the fact that timely and predictable communication services are essential to embedded real-time applications, such as automated factories and industrial process controls. Our proposed scheme provides an effective and simple mechanism for supporting intra-workcell time-critical communications in a computer-integrated manufacturing system, in which the ability to provide timely and predictable inter-process communication is of great importance because failure to complete specified message transmissions before their deadlines may risk failure of the functions of the communicating processes, risking equipment, plant and even human safety.

References

[1] "Industrial Automation Systems—Systems Integration and Communications—Fieldbus (draft) (ISA/SP50-1993)." Instrument Society of America, 1993.

[2] M. Y. Chan and F. Chin, "Schedulers for larger classes of pinwheel instances," *Algorithmica*, vol. 9, pp. 425–462, 1993.

[3] C.-C. Han and K.-J. Lin, "Scheduling distance-contrained real-time tasks," *Proc. IEEE Real-Time Systems Symposium*, pp. 300–308, Dec. 1992.

[4] D. Ferrari and D. C. Verma, "A scheme for real-time channel establishment in wide-area networks," *IEEE Journal on Selected Areas in Communications*, vol. SAC-8, pp. 368–379, April 1990.

[5] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multi-hop networks," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, Oct. 1994.

[6] Q. Zheng and K. G. Shin, "On the ability of establishing real-time channels in partially-connected networks," *IEEE Transactions on Communications*, vol. 42, pp. 1096–1105, February/March/April 1994.

[7] "Token passing bus access method and physical layer specifications." ANSI/IEEE Standard, 802.4–1985, 1985.

[8] "Token ring access method and physical layer specifications." ANSI/IEEE Standard, 802.5–1985, 1985.

[9] "Fiber Distributed Data Interface (FDDI) — Token Ring Media Access Control (MAC)." American National Standard, ANSI X3.139-1987, 1987.

[10] G. Agrawal, B. Chen, W. Zhao, and S. Davari, "Guaranteeing synchronous message deadlines with the timed token medium access control protocol," *IEEE Trans. on Computers*, vol. 43, pp. 327–350, March 1994.

[11] B. Chen, G. Agrawal, and W. Zhao, "Optimal synchronous capacity allocation for hard real-time communications with the timed token protocol," in *Proc. of Real-Time Systems Symposium*, Dec. 1992.

[12] C.-C. Han and K. G. Shin, "A polynomial-time optimal synchronous bandwidth allocation scheme for the timed-token MAC protocol," in *Proc. of IEEE INFOCOM'95*, (Boston, Massachusetts), April 1995.

[13] C.-C. Han, K. G. Shin, and C.-J. Hou, "Synchronous bandwidth allocation for real-time communications with the timed-token MAC protocol." submitted to J. of ACM, 1994.

[14] "IEEE Standards for Local and Metropolitan Area Networks: Distributed Queue Dual Bus (DQDB) Subnetwork of a Metropolitan Area Network (MAN)." IEEE 802.6, July 1991.

[15] D. Saha, M. C. Saksena, S. Mukherjee, and S. K. Tripathi, "On guaranteed delivery of time-critical messages in DQDB," in *Proc. of IEEE INFOCOM'94*, vol. 1, pp. 272–279, June 1994.

[16] C.-C. Han, C.-J. Hou, and K. G. Shin, "On slot allocation for time-constrained messages in DQDB networks," in *Proc. of INFOCOM'95*, April 1995.

[17] C. M. Aras, J. F. Kurose, D. S. Reeves, and H. Schulzrinne, "Real-time communication in packet-switched networks," *Proceedings of the IEEE*, vol. 82, pp. 122–139, January 1994.

[18] R. Holte, A. Mok, L. Rosier, I. Tulchinsky, and D. Varvel, "The pinwheel: A real-time scheduling problem," in *Proc. of the 22nd Hawaii International Conference on System Science*, pp. 693–702, January 1989.

[19] M. Y. Chan and F. Chin, "General schedulers for the pinwheel problem based on double-integer reduction," *IEEE Trans. on Computers*, vol. 41, June 1992.

[20] C.-C. Han, K.-J. Lin, and J. W.-S. Liu, "Scheduling jobs with temporal distance constraints," *to appear in SIAM J. on Computing*, 1995.

[21] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. of ACM*, vol. 20, no. 1, pp. 46–61, 1973.