Investigating the Synchronous Bandwidth Allocation in an FDDI Network for Real-Time Communications

Fang Feng, Amit Kumar, and Wei Zhao
Department of Computer Science
Texas A&M University
College Station, Texas 77843-3112

Technical Report 95-022

Abstract

The Fiber Distributed Data Interface (FDDI) is recommended as a candidate for many hard real-time local-area communication systems. The Timed-Token Medium Access Control (MAC) Protocol of FDDI is capable of guaranteeing real-time traffic on the FDDI network with the synchronous transmission mode. This report addresses the issues in implementing real-time communications on an FDDI network using synchronous mode transmission, and guaranteeing the application-to-application deadlines of multiple message streams.

Extensive research has been done on using FDDI for real-time communication. Most previous research was based on a simplified model and had two main limitations: (1) delays and deadlines were only considered at the MAC level; and (2) in order to simplify the analysis, only one message stream per node was allowed. These limitations restrict the applicability of the past research to practical FDDI-based communication systems, since in these systems, multiple communication applications may have to run on the same node to send out messages, and the processing delays of higher layers than MAC layer may have great impact on the total delay.

Our work removes these limitations and shows a systematic way for managing end-to-end message delays at the application level on an FDDI network. We consider the integrated system which include host subsystems (sender and receiver hosts) and the network. A *synchronous server* is proposed to eliminate the restriction of one message stream per node. The subsequent problems, faced in bandwidth allocation and network access when multiple streams are allowed on one node, are addressed. The design and implementation of the synchronous server are discussed, and a methodology is proposed for estimating end-to-end message delays and guaranteeing deadlines when using such a server. Experimental results are provided to support our claims.

Keywords: real-time, FDDI, bandwidth allocation, synchronous server, capacity constraint, clock synchronization, device driver, STREAMS, SMT, time stamps

Contents

1 Introduction

In this report, issues in investigating real-time communication using the synchronous mode in an FDDI local area network are addressed.

The Fiber Distributed Data Interface (FDDI) is an ANSI standard [3, 4, 6] for high-speed local area networks with a bandwidth of 100 megabits per second. The Timed-Token Medium Access Control (MAC) Protocol adopted by FDDI controls station access time. This protocol defines two classes of service: synchronous and asynchronous. Synchronous messages arrive at the MAC (Medium Access Control) layer at regular intervals. Each station is allocated a synchronous bandwidth which is the time period a station may transmit its synchronous messages after receiving the token. The FDDI standard also guarantees a bounded response time for each station. After transmitting the synchronous messages, a station can then transmit its asynchronous messages if the time elapsed since the last token arrival at the same station is less than the value of the Target Token Rotation Time (TTRT), i.e., if the token arrived early. Thus, the Timed-Token Protocol is capable of guaranteeing bounded access time, which is crucial for real-time traffic. FDDI has been adopted by the Survivable Adaptable Fiber Optic Embedded Network (SAFENET) [18, 17]. SAFENET is a military standard for computer networks developed with the Navy's Next Generation Computer Resources (NGCR) program.

There has been extensive research on using FDDI networks for real-time communication. Johnson and Sevcik [9, 15] proved that when the network operates normally, the time elapsed between two consecutive visits of the token at a node is bounded by $2 \cdot TTRT$. In [1], this result was generalized to state that the time elapsed between any m consecutive visits is bounded by $m \cdot TTRT$, and it is also proved that the worst case achievable utilization is 33% using the Normalized Proportional Allocation scheme. In [2], a local scheme was presented which only uses the local information on a node to allocate its synchronous bandwidth. In [10, 13, 14], extensive results were reported on FDDI networks for guaranteeing hard real-time messages with arbitrary deadlines, and buffer requirements on both sender and receiver ends, and the concept of *guarantee probability* were proposed. Other related work can be found in [5, 7, 8, 11, 16, 19]. The previous work has two limitations. First, it focuses on the MAC layer. For practical systems, host systems must be taken into account. Second, only one message stream is assumed on each node. For many real-time systems, multiple applications can run on the same node giving rise to multiple message streams.

The main contribution of this paper is to extend the previous work to the integrated system including both the sender and receiver hosts, and the network, and allowing multiple applications, communicating with applications on other nodes through the FDDI-based network, to be run on one node. Thus, the limitations of previous work are removed so that the results can be directly applied to practical systems.

Our approach to solve these problems can be described as follows. In order to allow multiple message streams on one node, we propose the concept of a *synchronous server* that is responsible for multiplexing messages from different streams on one node onto the single FDDI interface. The various alternatives for the design and implementation of such a server are discussed.

For the stable operation of a network and the nodes on it, some constraints must be satisfied. We present a set of such constraints, and show how to properly allocate synchronous bandwidth based on these constraints. This paper also shows how to estimate application-to-application message delays based on these constraints. The synchronous server is implemented and allocates synchronous bandwidth such that these constraints are satisfied. The synchronous server uses an algorithm based on priority (assigned inversely proportional to the period of a message stream) to arbitrate access to the network. We claim that if the proposed set of constraints is satisfied, end-to-end deadlines at the application level can be guaranteed for synchronous message streams. Various experiments using the synchronous server are conducted to support this claim.

Our work shows a systematic way for measuring end-to-end message delays at the application level on an FDDI network, and can be viewed as a set of guidelines for implementing such a real-time system, determining the message parameters, estimating delays in such a system, and verifying whether a message can be delivered within the specified deadline. By studying a set of constraints on the different components, our work provide insights to the nature of such an integrated system.

This paper is organized as follows. The design and implementation of the synchronous server are discussed in Section 2. Section 3 provides details on the issues of guaranteeing application-to-application deadlines. Experimental results are given in Section 4. Concluding remarks are presented in Section 5.

2 The Synchronous Server

2.1 Motivation

In the current commercial FDDI products, not all of them implement the synchronous mode. In those which implement the synchronous mode, a First In First Out (FIFO) queue (SYNC) is used for outgoing synchronous messages. There is no way for the FDDI MAC hardware to distinguish among packets from different applications, and it has no knowledge of the individual synchronous bandwidth allocated to these applications. When the token arrives at a node, the MAC transmits messages in the SYNC queue in FIFO order until it exhausts its total synchronous bandwidth[1].

As a hardware solution, one network interface can be provided for each real-time message stream resulting in multiple FDDI interfaces on each node. This solution is not only prohibitively expensive, but also inflexible, since the number of message streams on a node would be limited by the number of interfaces available. Also, this would require a rather complicated device driver, or one instance of it per interface.

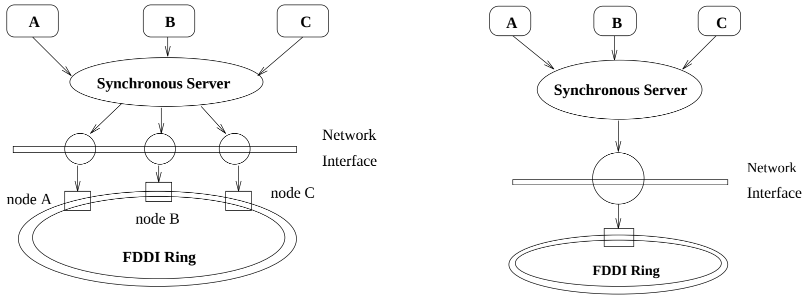
Another alternative is to redesign the MAC interface so that multiple synchronous queues can be provided at the MAC level and synchronous messages from different applications are dispatched to different queues. There is a one-to-one correspondence between an application and a synchronous queue at MAC level. This requires a complicated device driver and MAC control mechanism. This approach is not only costly but is also poor in scalability. The FDDI MAC standard must also be modified for handling multiple SYNC streams.

Our objective is to use the existing communication software, device driver, and FDDI interface to deal with multiple message streams on the same host. Logically, these streams can be thought of as multiple FDDI stations and each of them has its own FDDI hardware interface. Physically, they share the single communication path and the single SYNC queue on the FDDI MAC interface. This problem of multiple message streams on a host can be solved using hardware or software solutions. We now consider some of these solutions. As will be seen, several of these are infeasible.

In practical systems, multiple applications can share the same communication path and the same network interface. For real-time applications on an FDDI network, we assume they all use the synchronous mode to transmit messages. Thus, there is a synchronous bandwidth allocation problem. Consider an example: three applications, A, B and C, all executing on the same host, make connections to three (remote) destination applications, D, E and F, all of which may or may not reside on the same machine. On the sender machine, the outgoing packets must go through the same FDDI interface. This means that the message stream arriving at the network subsystem is multiplexed from three different message streams. The multiplexing process cannot be done arbitrarily in any order, since the bandwidth allocated to a stream must be respected, otherwise the network may be flooded.

To solve this problem, we try to simulate the three streams on one node as single streams on individual nodes on the ring. Logically, they appear as individual nodes on the FDDI ring with their own bandwidth allocated. Physical implementation must multiplex all the streams onto one communication path. We therefore need an auxiliary process, which we call the *Synchronous Server (SS)*, which constantly receives packets from these streams and passes them down to the

[1]After transmission of synchronous messages, it may continue to transmit asynchronous messages, for a period of time equal to the token holding time (THT), if the token arrived early.



(a) The Logical View (b) The Physical View

Figure 1: Two Views of the Synchronous Server

FDDI interface, according to the synchronous bandwidth allocated to them. Since we have only one processor and one communication path, each packet must go through the same processing path. All of the message streams need to be multiplexed onto this path. We use an application program to arbitrate the connections. The application sending messages must send them to the SS which then dispatches these message packets according to the individual synchronous bandwidth allocated to each application. Figure 1 shows the two views of the SS.

Software implementation of the multiplexing adds overhead to the processing of messages which may degrade system performance since the host CPU must spend time doing the multiplexing. However, if the overhead is relatively small compared to the real processing jobs, and therefore, almost negligible, then it provides a good solution to the problem. As will be shown in this paper, this overhead is found to be tolerable.

2.2 Implementation Alternatives

It is ideal that the synchronous server be put into the protocol stack as a transport layer service. However, this would require rewriting the communication component of the operating system, and is not a feasible alternative in our case due to unavailability of the source code.

Alternatively, it is possible to incorporate it as added functionality into the MAC layer firmware or the device driver, but this seems incorrect, since synchronous bandwidth allocation involves knowledge of applications, and hence should be dealt with at a higher layer. The device driver and MAC firmware only deal with packets syntactically, without any knowledge of the semantics of the user data packets. On the other hand, the device driver and MAC firmware must be executed efficiently, and therefore it is not permissible for them to read the contents of the data packets in order to retrieve bandwidth information contained in the packets. Additionally, the significant overhead introduced by implementing the synchronous server at such a low level is definitely intolerable.

The only feasible solution is to implement the SS at the application layer. As will be shown, it is not very difficult to implement the SS at the application level. The main problem is the overheads it introduces. Improper implementation could greatly degrade the performance of the system and waste system resources. Several approaches were experimented with to implement the SS, some of which are discussed here.

In the first implementation, the SS is designed as a concurrent server running as a background process. It opens a UNIX domain socket and listens on the socket. Any local application that wishes to communicate with another application on a remote host must open a UNIX domain socket and pass all its data to the SS through this socket. The SS forks a child process to handle this application, and the parent process continues to listen on the socket for more application requests. The child process simply sends whatever it receives from the client process to the remote application. In this implementation, data is generated in the client process and the SS merely passes whatever it receives from the client to the remote receiver application. This alternating sequence of *read*s and writes in the SS results in poor performance for this implementation.

In the second approach, applications (clients) are allowed to only pass their parameters (those described in the traffic model in Sec. 3.1) to the SS. This type of data transfer between the clients and the SS can be interpreted as a form of message passing such as by shared memory, where the client writes the data and the SS reads it. This precludes the need for copying the data buffer twice, which is needed if the SS reads the data off a socket (from the client), and then writes the data to another socket for sending to the remote server, as in the first approach. Each client request is still handled by a forked child process. This approach enforces that each application only uses the bandwidth calculated from its parameters (i.e., only sends out J_i packets within each TTRT, and M_i packets in each P_i). The problem with this implementation is that all connections are handled independently by separate (child) processes, each of which has its own TTRT timer. Thus, there is no common reference point of timer control for these processes, which could cause bandwidth utilization to exceed the bandwidth allocated. Another problem with this approach is that it does not consider message priorities.

The current implementation takes into account not only the bandwidth allocated to every application, but also the priority assigned to each application[2]. A common timer is used for TTRT control so that the transmission tasks for all applications are based on one reference point. The parameters for the application requests are imported to the SS through a file. For simplicity,

[2]The priority mentioned here should not be confused with the priority assigned due a process based upon its class (real-time, time-sharing, or system) 4.3. The priority referred to here is implied by the period of a message stream, streams with shorter periods having higher priorities.

we assume all applications are launched and start generating messages simultaneously. It can be readily modified so that applications have distinct starting points, by simply delaying them a constant amount of time relative to a common starting point. Though, at first glance, this approach may look appear to have high overhead, it was found to have lower overhead than the previous two approaches mentioned above.

2.3 Algorithm Details

The implementation of the synchronous server controls the number of packets being sent within one TTRT for each client. Thus, for multiple clients, each packet is served in an predetermined order. This can be random, round robin, or priority-driven. Ordering based on priority is considered here. There exist many priority assignment schemes, such as Earliest Deadline First, Minimum Laxity First, Rate Monotone, Minimum Processing Time First, etc. How to assign priorities to clients is not explained here, since it out of the scope of this paper.

The input to the synchronous server is a set of triples that give the parameters of the message stream(s). Each triple is of the form

$$\{P_i, M_i, J_i\}$$

where P_i is the message period, M_i the message size (in packets), and J_i the synchronous bandwidth allocated to the application (also measured in packets). The clients are ordered by their priorities. Conquently, they are served in that order as well. The algorithm can be outlined as follows.

0. Initialization.
1. Select a packet: *select_packet(I)*. /* I is the index of the selected stream /*
2. Send the packet if $I \neq 0$.
3. If there is no packet to be sent (I = 0),
4.   sleep till the end of the current TTRT, or the closest period point.
5. Modify $TTRT$ and/or period counters.
6. Check for each application i whether M_i packets have been sent in the current period;
7.   If yes, set period quota counter to zero.
8. Check for each application i whether J_i packets have been sent for the current $TTRT$;
9.   If yes, set $TTRT$ quota counter to zero.
10. Check for each application i whether a new period is entered;
11.   If yes, reset period quota counter and increment total period counter by 1.
12. Check if a new $TTRT$ is entered;
13.   If yes, reset $TTRT$ quota counter.

14. Go to step 1.

A *period point* (line 4) is the instant in time when a new period starts. For example, the period points for application i are $k \cdot P_i, k = 0, 1, 2, \ldots$. In the algorithm, the applications are serviced by the synchronous server in the order of their priorities. Thus, if there is a higher priority packet ready to be sent, a lower priority one can not be sent before it. This is determined by the subroutine *select_packet*.

select_packet(I): /* assume message streams are ordered as 1...N. */

0. I := 0
1. For priority level i from 1 to N:
2.   If (M_i packets plus backlog have been sent in the current period)
3.   or (J_i packets plus backlog have been sent in the current $TTRT$)
4.   then continue;
5.   Else I = i;
6. Return I.

In each TTRT interval, at most J_i packets can be sent for application i. If the quota of J_i packets cannot be sent within a TTRT, the backlog is added to the quota for the next TTRT. Only when such a backlog exists can more than J_i packets be sent in one TTRT. The policy followed by the SS , therefore, is to transmit all packets for all applications, even if they are delayed. Similarly, in each P_i, at most M_i packets should be transmitted for application i, but backlog from previous period(s) may be transmitted in some period. Allowing the backlog to be sent may result in bursty arrival which may cause receiver buffer overflow and packet loss. We will enforce various contraints on the synchronous server such that backlog rarely occurs. Examples are provided in Section 4.

2.4 Other Possible Implementations

There are other possible algorithms to implement the synchronous server. Some of these described are described here.

Simple FIFO (First In First Out) and no priority control: Applications are served in the order in which their packets arrive. Priorities of message streams are not considered. Thus, a high priority message may be delayed by lower priority ones, which may cause it to miss its deadline. Furthermore, if multiple packets of one application arrive at the SS in a bunch, they are served one after another and other streams have to wait. This may result in unfair service and possible network or receiver congestion, since it may cause the bandwidth utilization to exceed the allocated value.

One modified version of the above is to allow applications to send their packets as long as they do not exceed the allocated bandwidth. High priority applications may still possibly be blocked by those with lower priorities.

Pure priority control: All connections are served according to their priority. For each priority level i, all packets within one period are sent out without interrupt if no other higher priority messages are available. So for each message stream, its traffic is burstier than with the chosen scheme. This may exceeding the bandwidth allocated to a connection and cause congestion at the network or at the receiver end.

As mentioned in the previous section, the order in which packets are selected by the algorithm can be in round-robin fashion. This appears to provide fair service to all applications. Our argument against this is that applications with higher priority have more stringent timing constraint, and hence, the application priorities should be respected as far as possible.

3 Management of Message Delays

The application-to-application delay for a message of a certain application is defined as the time span from the instant the first packet of the message is generated by the application, and is ready to be sent at the sender host (which, as stated in the assumptions, is the start of the corresponding period), to the instant when the last packet of the message is received by the receiver application on some remote host. For individual packets, packet delay is defined as the time span between the instant the sender host starts processing a packet and the instant that packet is received at the receiver host, both times being measured at the application level.

Note that a simple sum of the individual packet delays from a message is not equal to the corresponding message delay, the reasons for this being (i) the first packet of the message may not be sent immediately after it is ready, since packet transmission must follow the schedule determined by the bandwidth allocation and priority, (ii) the processing of the packets of an application is likely to be interleaved with processing of packets from other applications, so that all the packets from one message for any application may not be processed (and sent down to the MAC for transmission) consecutively, and (iii) even though the packets may sometimes be sent consecutively, the pipelining effect of sending multiple packets must be taken into account.

The delays can be estimated based on the scheduling algorithm and estimation of some parameters. With the implementation of the synchronous server, we are able to verify the message delays for an application. The delays of messages are predictable under normal conditions and can be estimated by analytical methods. In other words, if the packet delay is much smaller than message delay, one or several packets being disturbed by other processes does not greatly affect the delay of the message, and message delays can be estimated fairly accurately. These estimates can be verified by the obtaining experimental results.

3.1 Network and Message Models

The network consists of a set of N *nodes*, each of which has one host processor and one FDDI interface[3]. On node i, there are F_i outgoing and G_i incoming message streams. The jth outgoing message stream on node i, Ψ_{ij}, is described by a triple:

$$\Psi_{ij} = \{L_{ij}, P_{ij}, D_{ij}\},$$

where P_{ij} is the period, L_{ij} the message size, measured in bytes, and D_{ij} is the application-to-application deadline. Once the packet size σ is fixed, the number of packets M_{ij} to be sent in each period can be easily calculated as $M_{ij} = \lceil \frac{L_{ij}}{\sigma} \rceil$ [4]. Our goal is to allocate synchronous bandwidth J_{ij}, measured in number of packets, to each of the outgoing streams on its sender node. Once the bandwidth is allocated, the flow of message streams is restricted. Thus, maximum delays of messages can be calculated.

3.2 Delay Analysis

To estimate delays experienced by a message, we need to decompose the entire communication path, through which every packet passes, into subsystems. These delays can be itemized as follows, for each of S (Sender Host Subsystem), N (Network Subsystem), and R (Receiver Host Subsystem):

- S – Sender Host Subsystem.
  - Application level system call to write data to a socket;
  - Stream head processing;
  - UDP module processing;
  - IP module processing;
  - Driver processing;
- N – Network Subsystem.

[3] The term *host* usually refers to the host machine, not including the network inferface.
[4] For the purpose of our tests, we choose the packet size σ to be 2000 or 4000 bytes.

  - DMA data transfer to FDDI interface memory from main memory;
  - Queueing and transmission at the sender network interface;
  - Network propagation;
  - Reception and queueing at the receiver network interface;
  - DMA data transfer to main memory from the interface memory;

- R – Receiver Host Subsystem.
  - Device driver processing;
  - IP module processing;
  - UDP module processing;
  - Stream head processing;
  - Application processing.

Under normal conditions, a packet is processed continuously on the sender host till it is ready to be transferred to the network interface by DMA, before the processing of another packet is started. Thus, on the sender side, the packet is output from the processor to the device one after another. However, the processing on the receiver end is somewhat different. Since the reception process is slower than transmission, the processing of a packet may be interrupted by the arrival of new packets. In general, such interrupts can be amortized by simply adding a delay to a common base. Thus, we can reasonably combine all the processing delays in the S and the R components into two chunks, sending processing and receiving processing, respectively, without loss of generality.

With the help of timestamps in the device driver, the packet delay can be decomposed into three segments: application-to-driver delay (sender host), driver-to-driver delay (network), and driver-to-application delay (receiver host). Note that the DMA delay (waiting for DMA in addition to DMA transfer) is nearly a constant if transmission load is not very high and no other devices request DMA.

The reason the driver-to-MAC and MAC-to-driver delays are included in the network subsystem rather than the host subsystems, is that, with the available resources, it is not possible to obtain timing information at the MAC layer. By including these delays in the driver-to-driver delay, we obviate the need for this timing information. Approximate values of these delays for packets sizes of 2000 and 4000 bytes and the estimated MAC transmission time are presented in Table 1. The value given for packet delay is a sum of the application-driver, driver-driver and driver-application delays. The MAC transmission time includes overhead like IP header.

delays types	4000 (bytes)	2000 (bytes)
I_S	6	8
I_R	3	4
C_S	1700	1400
C_R	2500	1900
app-drv delay (μs)	680	570
drv-drv delay (μs)	1720	1060
drv-app delay (μs)	1150	1020
packet delay (μs)	3550	2650
MAC transmission	324.16	164.16

Table 1: Values of Various Delays

3.3 Constraints

Depending on the processor speed, the protocols chosen, and the packet size, maximum number I of packets that can be transmitted within one TTRT can be measured[5]. For a sender to be stable, the number of packets (called transmission load) should not exceed I in one TTRT, otherwise a backlog of packets forms and may be accumulated more and more. Thus, if in the real system, the bandwidth demand of multiple streams exceeds I, a faster machine has to be used, as we discussed earlier.

For the system to be able to run properly, several conditions should be satisfied: (i) the transmission load on the host processor in terms of number of packets transmitted in one TTRT is less than the maximum transmission load (I), so that no backlog occurs at the sender, (ii) for a pair of sender and receiver nodes, the throughput of the sender and receiver nodes must be controlled so that they are compatible, i.e. they are almost equal; (iii) the total synchronous bandwidth allocated to message streams is no more than TTRT; and (iii) interferences should be taken into account. All of these conditions can be summarized in the following constraints.

We now extend the theory established in [1, 13, 10]. The previous work was limited to single message streams per node, and consider only MAC-to-MAC delays. These limitations are removed here. The previous work is extended to the case where multiple message streams may be present on a node.

We first propose several constraints which must be satisfied in order to ensure that messages are transmitted before their deadlines, synchronous capacity allocation is not violated, and timed

[5]In our experimental FDDI network, at most 6 packets with size 4000 bytes or 8 with size 2000 bytes, can be sent within one TTRT.

token protocol requirements are satisfied.

(A) *Network Bandwidth Constraint:* The total sum of the synchronous capacities allocated to all message streams on all nodes must not exceed the available portion of the Target Token Rotation Time (TTRT), i.e.,

$$\sum_{i=1}^{N}\sum_{j=1}^{F_i} H_{ij} \leq TTRT - \tau, \tag{1}$$

where N is the number of nodes, F_i is the number of message streams on node i, and τ is the token walk time.

(B) *Sender Flow Control Constraint:* On node i, the total time for processing $\sum_{j=1}^{F_i} J_{ij}$ packets for all F_i streams (including any intermittent disturbances) must not exceed TTRT. That is, for $i = 1, \ldots, N$,

$$\sum_{j=1}^{F_i} J_{ij} \cdot C_S^i + \delta_S \leq TTRT, \tag{2}$$

where C_S^i is the sending processing time of one packet on node i, and δ_S the delay due to any disturbance.

(C) *Receiver Flow Control Constraint:* Similarly, on node i, the total time for processing $\sum_{j=1}^{G_i} K_{ij}$ packets from all incoming streams must not exceed TTRT. That is, for $i = 1, \ldots, N$,

$$\sum_{j=1}^{G_i} K_{ij} \cdot C_R^i + \delta_R \leq TTRT, \tag{3}$$

where C_R^i is the receiving processing time of one packet on node i, and δ_R the delay due to any disturbance. Note that every K_{uv} is equal to some J_{ij} and there is a one-to-one correspondence between them.

(D) *Synchronous Server Constraint:* All packets in a message of a message stream must be transmitted before the end of the corresponding period, otherwise a backlog will form and accumulated, message delay becomes longer and longer, and eventually all messages miss their dealines. Also all packets in a message of a stream must be transmitted before the deadline minus a maximum packet delay, else the last packet may not reach the receiver before the deadline. Thus, The total time needed to transmit all the packets from a message for a stream Ψ_{ij} on node i must be less than the corresponding period or the deadline minus the maximum packet delay, whichever is earlier, that is, for $i = 1 \dots n$, and $j = 1 \dots F_i$,

$$\left\lceil \frac{M_{ij}}{J_{ij}} \right\rceil \leq \left\lfloor \frac{min(P_{ij}, D_{ij} - PD_i)}{TTRT} \right\rfloor \tag{4}$$

(E) *Deadline Constraint:* If all constraints above are satisfied, the delay of a message from a message stream can be estimated and its value must be less than its deadline. That is, for $i = 1, \dots, N$ and $j = 1 \dots F_i$,

$$d_{ij} = (\left\lceil \frac{M_{ij}}{J_{ij}} \right\rceil - 1) \cdot TTRT + SP_{ij} + PD \leq D_{ij}, \tag{5}$$

where SP_{ij} is the delay of the start time of the last packet of a message on node i, relative to the last TTRT point. The meaning of SP_{ij} will be clear when we discuss the delay estimation. PD is the worst case packet delay, J_{ij} the number of packets to be sent out every TTRT for stream Ψ_{ij}, d_{ij} the message delay, and D_{ij} the message deadline. These parameters will be discussed further in the next section.

Constraint (A) enforces restrictions on each node at MAC level. Constraints (B) and (C) require two hosts to cooperate and keep the same pace. Contraint (D) requires the SS to process all packets of each period before the next period. Constraint (E) enforces all packets to be processed before the deadline.

Note that H_{ij} is the synchronous capacity expressed in terms of time at MAC level, whereas J_{ij} is the synchronous capacity expressed in terms of packets. Their relationship is

$$H_{ij} = J_{ij} \cdot \frac{\text{Packet size in bytes}}{\text{Time to transmit 1 byte}} + \delta \tag{6}$$

where δ is the protocol overheads, such as IP header, FDDI frame header, etc. The time to transmit one byte can be calculated knowing that the maximum bandwidth available is 100 Mbps (or 12.5 MB/s). Hence, the time required to transmit 1 byte is $\frac{1}{12.5 \times 10^6}$ seconds, or 80 ns.

3.4 Messsage Delay Estimation

Recall that the delay of a message is not simply the sum of the delays of all packets comprising the message. In general, it is difficult to obtain an analytical formula to describe the message delay. However, based on the implementation of the synchronous server, we are able to predict message delays fairly accurately.

We claim that Equation 5 can be used to estimate message delays and assign appropriate deadlines if the remaining constraints are satisfied. In this equation, $TTRT$ and M_{ij}, are known, PD can be measured by experimental means, and SP_{ij} and J_{ij} need to be estimated appropriately. We now show how to calculate appropriate values for SP_{ij} and J_{ij}.

When a packet is generated by the corresponding application, it may not be sent immediately, because the processor is processing a packet of higher priority or a packet from the same stream. Thus a packet may be delayed before it starts being processed. We call such a delay the *start delay*. The start delay Q_i[6] caused by one packet can be measured. For a packet of 4000 bytes, it is about 1800 μs. Using Q_i as a fixed constant, the maximum possible value of SP_{ij} can be calculated as

$$SP_{ij} = (J_{ij} - 1 + \sum_{l=1}^{j-1} J_{il}) \cdot Q_i. \tag{7}$$

SP_{ij} is dependent on the value of J_{ij} and those from higher priority streams.

In TTRT there cannot be more than J_x packets for any stream Ψ_x on a node, plus those from higher priority streams on the same node, there cannot be more than $\sum_{j=1}^{x-1} J_{ij}$ packets to be sent. Therefore, the last packet cannot be delayed by more than SP_{ij}.

Now consider J_{ij}. In the previous section, five constraints are discussed. The values of J_{ij}'s must satisfy all these constraints. The accurate solution to those inequalities will give a set of intervals in which J_{ij}'s take integer values. The problem is the complexity of the method. We propose a simpler method. We observe that Constraints (A), (B), and (C) require J_{ij}'s as small as possible, and Constraint (E) demands larger values for J_{ij}'s. Constraint (D) provides lower bounds to J_{ij}'s. Thus, our method works in an iterative fashion. First we derive all these lower bounds and use them as the real values, and then check whether constraint (E) is satisfied. Note that (E) is checked for each message stream on the sender node. If (E) is true, then check (A), (B), and (C). If any of (A), (B) and (C) is not true, then there is no way to assign sychronous bandwidth in this case. If (E) is not true, we can increase the values of some J_{ij}'s and try again. Note that the values of J_{ij}'s are all integers; the number of packets that can be sent within a TTRT is fixed; and (A), (B) and (C) will provide clues about whose value can be increased. This increase-check process is repeated until one of the following is held:

- (A), (B), (C) and (E) are all satisfied; or
- (E) is failed and one of (A), (B) and (C) is failed as well.

[6] Q_i is identical to C_S^i.

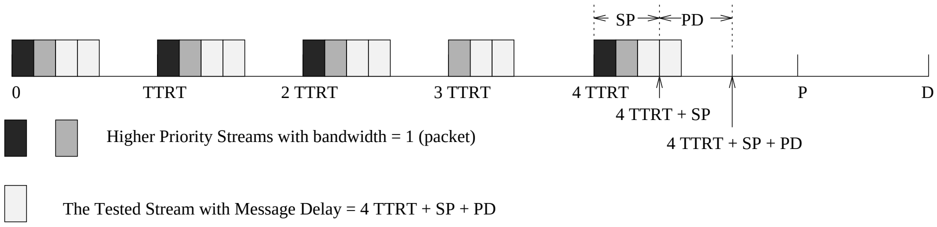


Figure 2: An Example of Message Delay Estimation

The first case implies that a set of values for J_{ij}'s are found. The second case means that this method can not find any set of values for J_{ij}'s.

To find the lower bound for J_{ij} of message stream Ψ_{ij}, let W_{ij} be the potential number of TTRT in which packets can be transmitted. Thus W_{ij} satisfies the following:

$$W_{ij} = \lfloor \frac{min\{P_{ij}, D_{ij} - PD_i\}}{TTRT} \rfloor. \tag{8}$$

Using W_{ij}, J_{ij} should satisfy:

$$J_{ij} \geq \lceil \frac{M_{ij}}{W_{ij}}. \rceil \tag{9}$$

Once J_{ij}'s is determined, K_{uv}'s are also determined. Increasing the values of J_{ij}'s can be done exhausively, or by heuritic. Exhasive search should be avoided, since some heuritics can reduce the search space greatly. Some of the heuristics are:

- For node i, if there are relatively more message streams on the node, each of their J_{ij}'s should be small. So try to increase such J_{ij} later.
- The maximum packet that can be sent within one TTRT would impose an upper bound on the sum of J_{ij}'s on one node.

We now consider an example. Figure 2 shows a stream with $J_{ij} = 2$ and $M_{ij} = 10$, and two other higher priority streams with $J = 1$. The estimated delay by formula 5 is about $4 \cdot TTRT + 3 \cdot 2000 + PD_i$, which is about 50000 μs for $TTRT = 10ms$ and packet size = 4000 bytes (thus, $PD_i = 3550$).

Equation refeq:proccons1 refers to intermittent disturbances. These may be due to other user processes or some system daemons. It is impossible to eliminate such interference. Usually, the duration of such sporadic interference is short and they affects several packets. So only a few messages may experience longer delays which may exceed their deadlines. Thus, in the strict sense of guaranteeing hard real-time deadlines, we are unable to provide a 100% guarantee. However, if we modify our definition of guaranteeing to allow a certain level of loss, then we still provide certain guarantees. This is explained in the next section.

3.5 Deadline Guarantee

For high-speed networks based on optic fiber (such as ATM, FDDI, etc.), the error rate is very low. Real-time applications require a certain quality of service (QoS), such as maximum packet loss ratio, maximum packet delay and delay variation, etc. For some delay-sensitive services, packet retransmission is out of the question, but a certain degree of packet loss is tolerable. For such network services, a new notion of guarantee is needed, called the *pth percentile guarantee*. This concept is not only a statistical measurement, but also a guide for the design, analysis, and implementation of a number of real-time communication systems.

Given a real-time communication service, and a set of parameters which define the quality of service, the pth percentile guarantee to the service requires that at least p percent of all the packets be delivered within their delay bound (the deadline) and delay variation be within the constraint.

Hard real-time guarantee studied extensively in past research is only a special case of the pth percentile guarantee when $p = 100$. This is different from the *guarantee probability*.

4 The Experiment System and Results

The aim of the experiments is to show that using the synchronous server implemented in Section **??**, actual message delays can be estimated and the application-to-application deadlines checked. If any of the constraints discussed in Section **??** is violated, message deadlines may not be guaranteed.

4.1 The Experimental FDDI System

For our tests to be run on the FDDI network, the very first thing is to implement the synchronous transmission mode on the firmware.

There has been a debate on the usage of the synchronous mode in FDDI protocol. The asynchronous mode can not guarantee bounded delay. The more stations on the network, the longer the delay might be. However, this only happens when the network is heavily loaded. People argued that it is hardly imagine that networks like FDDI could be saturated in a local area. The bandwidth is more than required, so the situation would rarely occur. Even though there may be some short period in which the network will become so heavily loaded that asynchronous mode

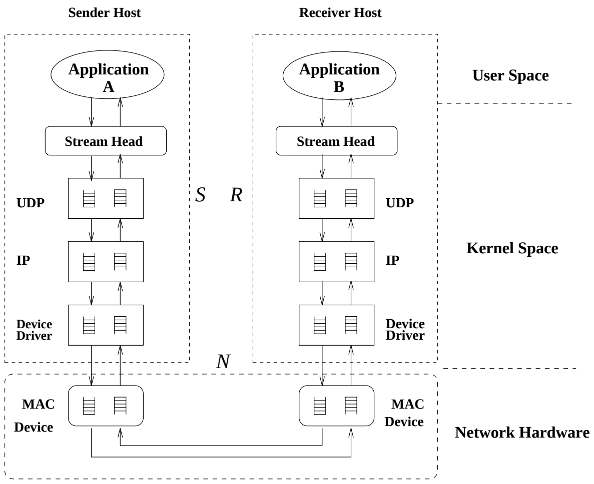


Figure 3: Decomposition of the Communication System

can not guarantee bounded access and delay, it is tolerable. Thus, there is no need to distinguish synchronous mode from the asynchronous one.

On the other hand, hard real-time communication require absolutely bounded delay, so that not single one message can be lost under any circumstances. This is very important to mission control, reactor control, space shuttle, types of systems. In such systems, absolute guarantee is required. Thus synchronous mode is necessary.

As far as commercial products are concerned, synchronous mode is optional. Thus not many commercial FDDI products implement the synchronous mode. The adapter we used implemented the synchronous mode, but it was not activated. So we have to activate it before it can be used. In order to do this, the source code for the FDDI device driver for the Solaris 2.x platform, and that for SMT ver. 6.2 was previously modified to support synchronous mode tranmission. See [12] for implementation details.

The communication protocol layers in our test systemm include the socket interface, the transport protocol UDP, network protocol IP, and the device driver that is responsible for the LLC sub-layer of the data-link layer; the FDDI network interface (NIF) board implements all functionalities of the MAC sub-layer of the data-link layer, and those of the physical layer. This is shown in Figure 3. The NIF board contains a 256KB buffer memory which consists of one synchronous queue (SYNC), three asynchronous queues (ASYNC0, ASYNC1 and ASYNC2), one receive queue, and a special frames area. [12] gives some details.

In SunOS 5.x (Solaris 2.x), most device drivers are implemented using the STREAMS mechanism. When an application makes a system call to transmit messages through a particular interface, the stream head copies the data from the data space of a user process to the kernel data space. The data is only copied once by the CPU on the sender side; thereafter, data passing between subsequent protocol processing modules are through pointers. Finally, the data is transfered by DMA from the kernel space to the interface device. A similar process exists at the receiver host, except in the reverse order.

Our experiments utilize the synchronous server described in Section 2. All test programs are based on the client/server model[7]. The synchronous server executes on one of the machines which is used as the client machine. Multiple clients (real-time applications), each described by a set of parameters, request the services provided by the synchronous server to multiplex their message streams and to transmit the data through the FDDI network interface. The servers (also real-time applications) reside on other machines and receive data through the network. All these processes can be run as time-sharing processes or as real-time processes on the Solaris 2.x platform.

Our experimental FDDI network consists of six SUN SPARCstations, csos2 and csos3 (SPARCclassic's), dsls2, dsls3 and dsls4 (SPARCstation 10's), and dsls5 (SPARCstation LX). Of these machines, three (csos2, csos3 and dsls3) run SunOS 5.3 (Solaris 2.3), and the others run SunOS 4.1.3. Our tests are conducted on the machines running Solaris, due to availability of the source code for the FDDI device driver running on these machines. The other machines running SunOS 4.1.3 are used mainly to generate additional traffic on the network. Figure 4 illustrates the experimental FDDI system.

These machines have different configurations. For example, the packet processing speed of a SPARC 10 is approximately twice that of a SPARCclassic. Their loads and network configurations are different in general. The internal hardware clocks of these machines may have different precisions. We, therefore, chose to use machines with similar architectures. Thus, all major tests are conducted on csos2 and csos3.

We tested two basic setups: (i) a single stream on each node, and (ii) multiple streams on a node. The former is used to demonstrate the impact of the network traffic and host load on various delay measurements, fundamental timing measurements, and transmission rates of the sender and receiver hosts. The latter is used to demonstrate the working of the synchronous server, the impact of priority of message streams and transmission load, and comparison with other implementations. Hence, two types of message sets are used in our experiments: multiple message streams and single message stream.

In all these experiments, all delays are measured in microseconds, and the UDP checksum

[7]The SS should not be confused with the server in the client/server model. The SS is named as such since it acts as a server for multiplexing the client data streams. In fact, in the client/sever model, the SS would actually be the client with the remote receiver application being the server.

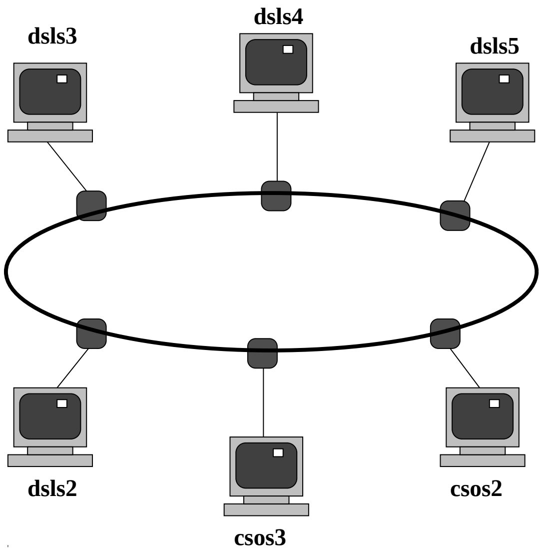


Figure 4: The Experimental FDDI network

option is disabled.

4.2 A Complete Example

Our experiments measure six timing quantities for each test run: start processing time of each packet, delay from application to driver (both on sender), driver to driver delay (sender to receiver), delay from driver to application (both on receiver), packet delay, and message delay. The start processing time of a packet is measured from its ready time, the beginning of the period of the corresponding message (we assume that all packets of one message have the same arrival time), to the time (T_1) when the packet is sent out. The application-driver delay is simply $T_2 - T1$. (Timestamps described in Section A.2). The driver-driver delay is $T_3 - T_2 \pm \Delta$. Since it involves two machines, the clock difference must be accounted for. It is added or subtracted depending on which clock is ahead. The driver-application delay is measured by $T_4 - T_3$. The packet delay is simply the sum of the previous three delays. Lastly, the message delay is from the beginning of the period of the message to the time when the last packet of the message is received at the receiver host. Figure 5 shows an example of these delays.

A single stream is used in this example:

$$\{P = 10, M = 3, D = 15000\},$$

and the packet size is 2000 bytes. Note that the scales for these plots are chosen differently for clarity. From (a), we can see an intermittent interference which delays the packet start

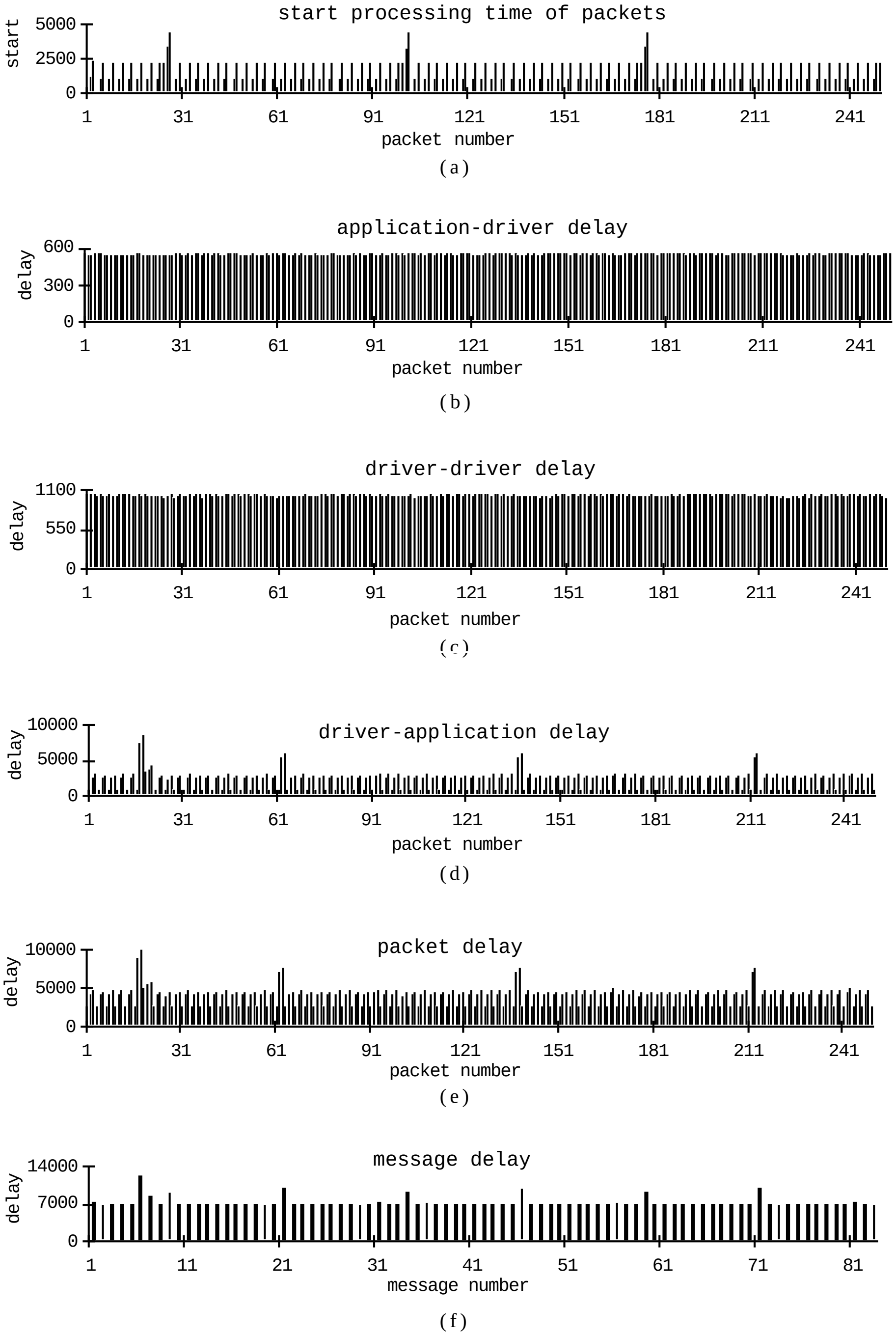


Figure 5: A Complete Example

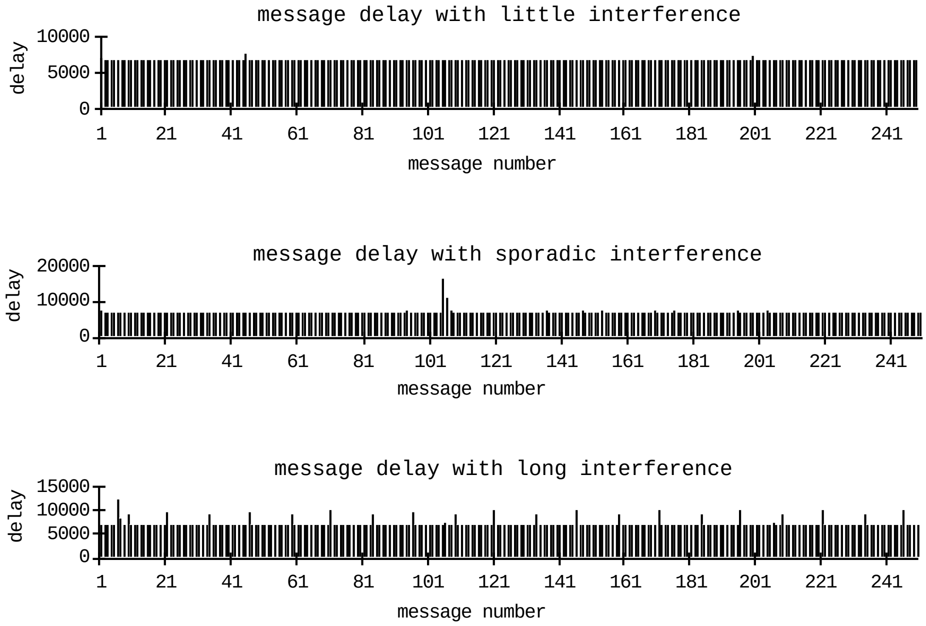


Figure 6: Impact of Time-sharing on Message Delays

time; similar interference also exists at the receiver machine, shown in (d). The application-driver delays in (b) and driver-driver delays in (c) are fairly constant. The sum of the application-driver, driver-driver, and driver-application delays gives the packet delay. Both sender and receiver delays contribute to the message delay, and three packets correspond to one message. The message delays are mostly within the estimated value using (5) in Section 3.3.

4.3 Impact of Timesharing on Message Delays

In Solaris 2.x, processes are divided into three classes: system, time-sharing, and real-time. Real-time processes have higher priorities than processes in the other two classes. When they are scheduled, real-time processes are always scheduled first. In each class, there are many levels which can be assigned to processes. A time-sharing process is assigned a time slice and priority based on its running time and its past running history, that is, depending on whether the process is CPU or I/O bound. Thus, the time slice and priority are dynamically changed. If the communication application is run in the TS class, it may be interrupted by the scheduler due to its limited time slice, or due to processes with a higher priority requiring the CPU. Thus, some packets and messages may be delayed. Figure 6 shows the impact of time-sharing processes.

The parameters of the application are:

$$\{P = 10, M = 3, D = 11000\},$$

and the packet size is still 2000 bytes.

In (a) and (b), there is little interference from other processes. Most message delays are constant and without much variance, except that, in (b), there are a couple of messages that experience long delays due to some sporadic interference[8]. In (c), there is a constant interference, which is due to a process running in the background that shares the CPU with the test application process. A time slice is allocated to that process, so the communication application process is interrupted at a regular interval. This type of interference can be eliminated if the real-time application is run in real-time process class.

4.4 Impact of Priority on Message Delays

In our implementation of SS, priority control is of secondary importance, bandwidth guarantee being the most important factor. Our data shows that changing the priority of a message stream does not greatly affect the delays of its messages, provided the utilization is moderate. However, if the tested message stream is of low priority and higher priority streams almost fully load the communication path (in terms of its maximum packet processing rate), then the delay may vary greatly. However, if the load becomes high, it can cause bursty arrival of packets at the receiver host causing overflow of the receiver buffer. Figure 7 shows the case in which priorities of message streams have certain impact on message delays when the transmission load is changed from light to moderate to heavy.

The packet size in this example is 4000 bytes. The parameters of the single stream in (a) are:

$$\{P = 91, M = 3, D = 30000\}.$$

The parameters of the three message streams in (b) are:

$$\{P_1 = 37, M_1 = 3, D = 30000\},$$

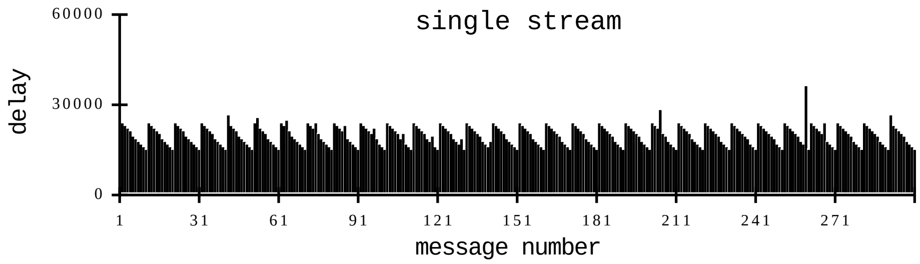
$$\{P_2 = 49, M_2 = 4, D = 40000\},$$

$$\{P_3 = 91, M_3 = 3, D = 35000\}.$$

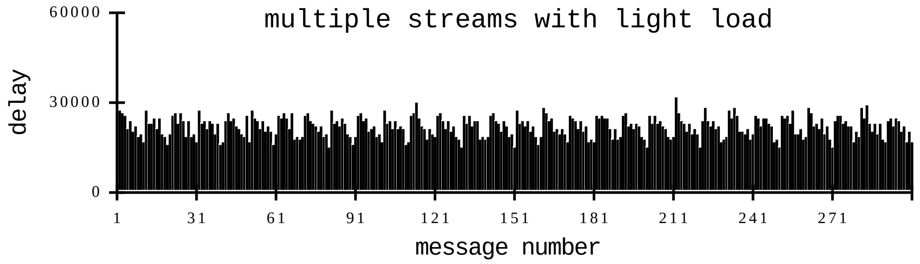
The parameters of the three message streams in (c) are:

$$\{P_1 = 37, M_1 = 6, D = 65000\},$$

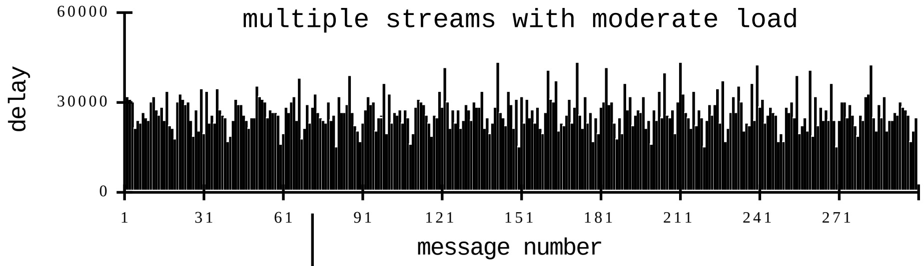
[8]It is very difficult to tell what this interference is.



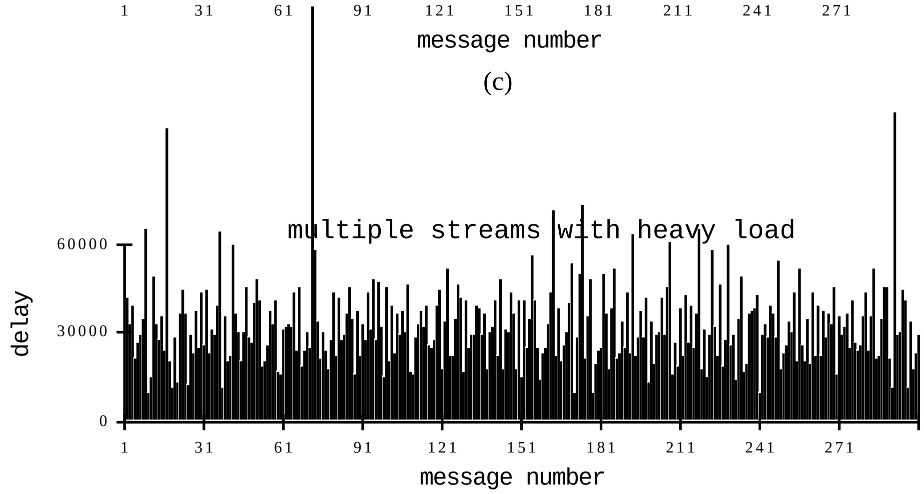
(a)



(b)



(c)



(d)

Figure 7: Impact of Priority on Message Delays

$$\{P_2 = 49, M_2 = 8, D = 85000\},$$

$$\{P_3 = 91, M_3 = 3, D = 40000\}.$$

The parameters of the three message streams in (d) are:

$$\{P_1 = 37, M_1 = 9, D = 95000\},$$

$$\{P_2 = 49, M_2 = 8, D = 90000\},$$

$$\{P_3 = 91, M_3 = 3, D = 60000\}.$$

In the last three cases, the third application, i.e. the one with lowest priority, is tested.

(a) shows a delay pattern which repeats every 10 messages, since the period is 91. This reveals an interesting phenomenon between the delay of a message and its period. The chart shows that in the case of single stream and load disturbance from other processes, message delays are well behaved. Some messages may be irregular due to unexpected interference such as the one with message number 259. For light and moderate transmission loads, message delays become more and more irregular and unpredictable, as shown in (b) and (c). When the load is even higher, the message delays are very long, and about 4% exceed its deadline. Using the concept of the percentile guarantee introduced in Section 3.5, we can still claim that a 96th percentile guarantee.

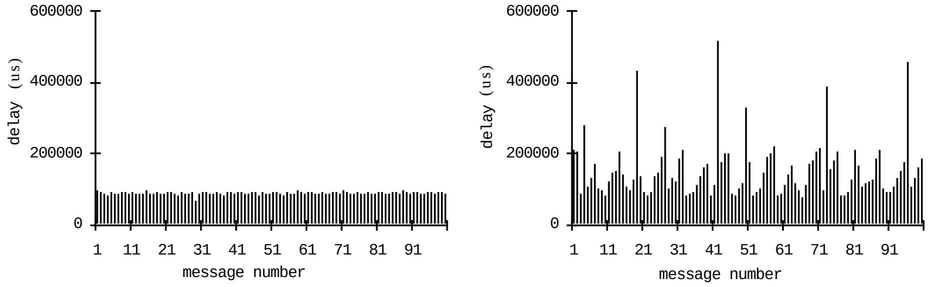
4.5 Impact of Host Load and Network Traffic

When we talk about host utilization, we consider other processes that compete for the CPU. Host utilization has a significant impact on message delays. On the sender host, it reduces packet processing throughput, which results in increased message delays. If the utilization on the receiver is high, received packets cannot be processed, so it can cause buffer overflow if the buffer is not sufficiently large. In this case, the transmission on the sender must be slowed down, or packets will be dropped at the receiver.

On the other hand, high network traffic does not greatly affect the sender host, since its synchronous bandwidth is guaranteed. Actually, it may ease the congestion on the receiver host, since it slows down the arrival rate of the packets.

The figures here only show the cases of low and high utilization. Under very low CPU utilization, shown in (a), message delays are fairly constant and their variance is small. On the other hand, if CPU utilization becomes high (about 50%) [9], as shown in (b), the message delays are

[9] CPU utilization was measured using the UNIX command *top*.



(a) Test A: Host Load Low

(b) Test B: Host Load High

Figure 8: Impact of Host Load on Message Delays

several times longer, which can cause several messages to miss their deadlines. The parameters of applications in this example are:

$$\{P_1 = 237, M_1 = 12, D = 120000\},$$

$$\{P_2 = 349, M_2 = 10, D = 110000\},$$

$$\{P_3 = 673, M_3 = 10, D = 150000\}.$$

We tested the third application.

5 Concluding Remarks

Using the synchronous server, delays experienced by messages of a message stream can be predicted under light load condition and can be verified by various tests. They are nearly independent of priority under light loads of both hosts (sender and receiver) and the network, but dependent on priority if host load or network load are relatively high. Since the delays are predictable, we can assign deadlines for messages based on their worst case delay and provide a margin for safety. Our approach gives a step-by-step approach to measure delays. The significance of our work is that it shows how to design a real-time system and measure delays for message streams in a real system.

Our experiments shows that other host processes on both sender and receiver hosts greatly affect the message delays. When we design a real-time system, we must consider what kind of load will be exerted on the hosts attached to the network. Or, host load must be kept under a threshold. Under normal conditions, the network load usually is very light, especially for a high-speed network. Unfortunately, instantaneous overload situations may occur, a fact that has been proved by many prototype systems. Thus, when we design a hard real-time communication network, we must consider the instantaneous heavy network traffic situation.

The impact of the priority of the processes on both sender and receiver may not play an important role for most cases on an FDDI network. We can not say that it is not important in other network configurations. As a matter of fact, message priority may be an important way to guarantee QoS in such high-speed networks as ATM, Frame Relay, etc.

Network bandwidth allocation determines host scheduling policy. Host scheduling of processes must take underlying network bandwidth allocation and architecture into account. Our experiments show that pure priority-driven type of scheduling policies yield poor performance, since they ignore the bandwidth allocated, and may easily overload the network.

Packet size plays an important role in system performance. The larger the packet size, the better the performance of a transmission system. This is because relative overhead is lower for larger packet size, and invoking a protocol module takes the same amount of time for both small and large packet. Also, once a module has been invoked, the processing time a packet is not directly proportional to its size.

The most time-consuming processing in a communication job is on the hosts. On any communication path, there may be one or more bottlenecks that can be on the hosts or on the network. Network congestion has been extensively studied. Host congestion may also degrade the system performance to a large extent. If the communication process on a host cannot obtain enough CPU time, the entire communication path must be slowed down.

Our experiments show that the disturbance due to other network traffic or other CPU activities is the cause of delay fluctuations. These fluctuations can result in loss of packets by flooding buffers or in long packet delays. So, if a host is connected to several different networks, all these disturbances must be taken into account when estimating delays. Under such circumstances, the disturbance can cause delays as long as several thousand microseconds or even several hundred milli-seconds. This makes it impossible to predict delays with microsecond precision.

References

[1] Gopal Agrawal, Baio Chen, Wei Zhao, and Sadegh Davari. Guaranteeing synchronous message deadlines in high speed token ring networks with timed token protocol. In *Proceedings of the 12th IEEE International Conference on Distributed Computing Systems*, pages 468–475, June 1992.

[2] Gopal Agrawal, Biao Chen, and Wei Zhao. Local synchronous capacity allocation schemes for guaranteeing message deadlines with the timed token protocol. In *Proceedings of INFOCOM'93*, pages 186–193, 1993.

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

[4] ANSI Standard X3T9.5/88-139, Rev 4.0. *FDDI Media Access Control (MAC)*, October 1990.

[5] D. Dykeman and W. Bux. Analysis and tuning of the FDDI media access control protocol. *IEEE Journal on Selected Areas in Communications*, 6(6), July 1988.

[6] ISO International Standard 9314-2. *Information Processing Systems — Fibre Distributed Data Interface (FDDI) — Part 2: Token Ring Media Access Control (MAC)*, 1989.

[7] Raj Jain. Performance analysis of FDDI. *Digital Technical Journal*, 3(3):78–88, Summer 1991.

[8] Raj Jain. Performance analysis of FDDI token ring networks: Effect of parameters and guidelines for setting TTRT. *IEEE LTS*, 2(2):16–22, May 1991.

[9] M. J. Johnson. Proof that timing requirements of the FDDI token ring protocols are satisfied. *IEEE Transactions on Communications*, COM-35(6):620–625, June 1987.

[10] Sanjay Kamat, Nicholas Malcolm, and Wei Zhao. Performance evaluation of a bandwidth allocation scheme for guaranteeing synchronous messages with arbitrary deadlines in an FDDI network. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 34–43, December 1993.

[11] Mark H. Klein, John P. Lehoczky, and Ragunathan Rajkumar. Rate-monotonic analysis for real-time industrial computing. *IEEE Computer*, 27(1):24–33, January 1994.

[12] Amit Virendra Kumar. Real-time communication in FDDI-based networks. Master's thesis, Texas A&M University, 8 1995.

[13] Nicholas Malcolm and Wei Zhao. The timed-token protocol for real-time communications. *IEEE Computer*, 27(1):35–41, January 1994.

[14] Nicholas Malcolm and Wei Zhao. Hard real-time communication in multiple-access networks. *Journal of Real-Time Systems*, January 1995.

[15] K. C. Sevcik and M. J. Johnson. Cycle time properties of the FDDI token ring protocol. *IEEE Transactions on Software Engineering*, SE-13(3):376–385, 1987.

[16] Mary Jane Strohl. High performance distributed computing in FDDI networks. *IEEE LCS*, 2(2), May 1991.

[17] U. S. Department of Defense. *Survivable Adaptable Fiber Optic Embedded Network (SAFENET) Network Development Guidance*, 1994. MIL-HDBK-818A.

[18] U.S. Department of Defense. *Survivable Adaptable Fiber Optic Embedded Network (SAFENET)*, 1994. MIL-STD-2204A.

[19] Qin Zheng and Kang G. Shin. Synchronous bandwidth allocation in FDDI networks. In *Proceedings of ACM Multimedia'93*, pages 31–38, August 1993.

A Clock Synchronization

A.1 The Need for Clock Synchronization

In a communication system, the accuracy of clock readings is critical when we try to estimate the delays of real-time messages. Clock precision requirement depends on the applications. If the application runs for more than several seconds, then clock precision within several milli-seconds may be satisfactory. However, if we consider packet delays that are of the order of 10^2 or 10^3 milli-seconds, we must ensure that the clocks on the machines differ by less than a few hundred microseconds.

It is very difficult to synchronize the clocks on two machines to the exactly same value, since the constantly changing physical environmental conditions of the two clocks affect the clock drift. But, for our purpose of measuring message delays, we do not require the actual clock readings to be the same. What we need is to know the relative clock difference on the two machines in question and the variance of the clock difference. Using an experimental approach, we are able to estimate the clock difference to an accuracy of tens of microseconds that satisfies our requirements.

In UNIX, either of the commands *rdate* and *ntp* can be used to synchronize the clocks on two machines to within one second[10]. However, this precision is too coarse for our purpose. We need to design some simple algorithms to synchronize the clocks more accurately. These algorithms depend upon the timestamps which are obtained at various stages.

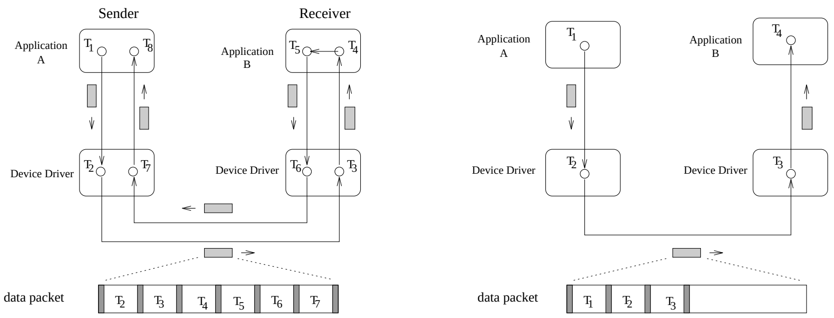
A.2 Timestamps

Timestamps are frequently used for obtaining timing information during execution of a task or process in a system. A task usually consists of a number of steps or processing stages, which are often programmed as separate subroutines or functions. The performance and correctness of the execution of a real-time task depend on the time of execution of these modules. To determine

[10] Super-user access is required to run *rdate* or *ntp*. See UNIX *man* pages for details.

the timing of a step of the task, the execution time of the corresponding module can be obtained by reading the system clock at certain fixed point in the module and saving the clock readings (or timestamps). The timestamping method is also useful for measuring message delays. We use timestamps to find out the exact time at which a packet goes through the various stages in the communication path, and use these timestamps to calculate the various processing times and delays.

The difference in the clocks on two machines can be calculated fairly accurately by using various timestamps obtained at the application and device driver level at both the sender and the receiver hosts. The timestamps are obtained using the system command *gettimeofday* (at the application level) or *uniqtime* (at the device driver level) which returns the number of seconds and microseconds elapsed since a reference point[11]. For our experiments, since the data portion of a packet carries no significant information, we overwrite a portion of it with some of these timestamps. Totally, eight timestamps may be obtained, four on the sender host, and the remaining four on the receiver host. Figure 9(a) shows the points along the transmission path where these timestamps are obtained.



(a) Timestamps in two-way traffic packets

(b) Timestamps in one-way traffic packets

Figure 9: The Timestamps in Packets

Two major uses of these timestamps in our experiments are clock synchronization and delay measurement. When used for clock synchronization (two-way traffic), eight timestamps may be used, of which T_2 through T_7 are put into packets, and T_1 and T_8 are actually captured at the sender. When used for delay measurement (one-way traffic), only four timestamps are used, of which T_1, T_2 and T_3 are put into the packets.

At the application level, four timestamps are obtained as follows: T_1 and T_5 are obtained at the sender and receiver, respectively, just before the packet is sent out, and T_4 and T_8 right after the packet is received at the receiver and sender, respectively, At the driver level, obtaining

[11]The reference point generally used is 00:00 GMT, January 1, 1970.

timestamps is tricky. T_2 and T_7 are taken at sender and receiver, respectively, just after the TBD (transmit buffer descriptor) is set up and before the data is really copied to the device by DMA, T_3 and T_6 after the packet has been copied by the DMA into kernel memory (from the RCV queue on the device), but before the packet is processed by the driver. Caution must be taken about where to put instructions into the driver code to take the timestamps, or the driver may be messed up.

Other than the two major uses mentioned above, the timestamps are also used for the following purposes:

- To calculate the processing times for transmission and reception;
- To study the impact of packet size on message and packet delays;
- To examine the possible cause and location of packet loss;
- To verify the allocated synchronous bandwidth; and
- To compare synchronous and asynchronous transmission modes.

These uses of timestamps will not be discussed here in detail due to limitations on space.

Next we show how to use the timestamps to calculate the difference between the clocks on two different machines.

A.3 Clock Synchronization Algorithms

If the two machines have the same hardware configuration, we can reasonably assume that their processing speeds are approximately the same, and transmission and reception processing take the same amount of time. Thus, a simple round trip time algorithm can be used to estimate clock difference. We call this algorithm the *Homogeneous Clock Synchronization Algorithm*. See Figure 10(a) for illustration. From the figure, we can see that the round-trip time is calculated by:

$$RTT = T_8 - T_1 - (T_5 - T_4)$$

Assume the difference between the two clocks CLK_A and CLK_B is Δ. Then, if CLK_A is ahead of CLK_B,

$$T_4 + \Delta - T_1 = \frac{RTT}{2};$$

or, if CLK_B is ahead of CLK_A, then

$$T_4 - \Delta - T_1 = \frac{RTT}{2}.$$

Combining these two cases, we have

$$\Delta = \left| T_4 - T_1 - \frac{RTT}{2} \right|,$$

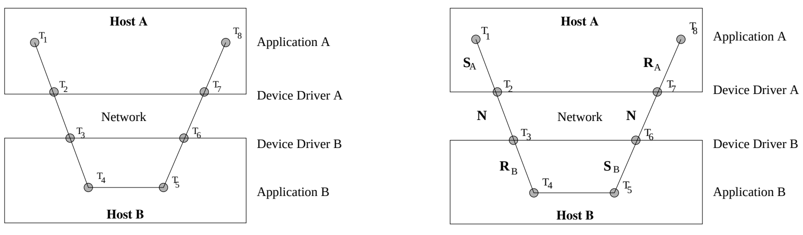
or

$$\Delta = \left| \frac{T_4 - T_1 + T_5 - T_8}{2} \right|. \tag{10}$$

In these formulae, we only use four timestamps: T_1, T_4, T_5, and T_8. From them we obtain the clock difference at the application level. Similar calculations can be done at the driver level using the other four timestamps, which should give more accurate estimation, since host interference has less impact on it.

To estimate clock difference, machine A sends a packet to machine B which then sends it back. This is repeated thousands of times. The timestamps from each packet are used to calculate the value of Δ. The final value of Δ can be taken as the average of all calculated values. When it is necessary to make the results more accurate, a statistical technique, the *confidence interval*, can be used to eliminate some values that are too large or too small compared to the other values, and then to find the average of the remaining values.

Synchronizing the clocks on two machines with different configurations is slightly more complicated. We can not assume that one-way delay is half the round-trip time unless transmission processing is equal to reception processing, which, in general, is not true. We need to use some more information provided by the eight timestamps. A more complicated algorithm than the previous one is proposed for this case. We call this the *Heterogeneous Clock Synchronization Algorithm.*



(a) Homogeneous Clock Synchronization (b) Heterogeneous Clock Synchronization

Figure 10: Two Kinds of Clock Synchronization

Using the timestamps (see Figure 10(b)), transmission and reception processing times can be calculated on the two machines. Assume that $\theta_1 = \frac{S_A}{S_B}$, and $\theta_2 = \frac{R_A}{R_B}$, and that machine A is slower than machine B, thus $\theta_i > 1$ ($i = 1, 2$). Note that, in general, $\theta_1 \neq \theta_2$, i.e., the processing times are not proportional. Test data shows that reception processing takes longer time than transmission processing (i.e., $R_A > S_A$ and $R_B > S_B$). The time interval between T_4 and T_1 is shorter than that between T_8 and T_5, and that the difference between these two intervals is:

$$\eta = R_A + S_B - R_B - S_A = (\theta_2 - 1) \cdot R_B - (\theta_1 - 1) \cdot S_B.$$

Again, we assume that clock difference is Δ. If CLK_A is ahead of CLK_B, then

$$T_4 - T_1 + \Delta + \eta = T_8 - T_5 - \Delta$$

Solving for Δ, we have

$$\Delta = \frac{1}{2}(T_8 - T_5 + T_1 - T_4 - \eta).$$

Combining this with the case where CLK_B is ahead of CLK_A, we have

$$\Delta = \left| \frac{1}{2}(T_8 - T_5 + T_1 - T_4 - \eta) \right|. \tag{11}$$

The values of S_A, R_A, S_B, and R_B can be obtained either by measuring multiple packets and averaging the calculated values beforehand, or by calculating them on the fly. Once this is done, a routine similar to the one used in the homogeneous algorithm can be followed to calculate Δ.

A.4 Adjustment of Clock Readings

Once the clock difference has been estimated, it can be used as a basis for measuring message delays. It is difficult to simultaneously synchronize clocks and measure message delay. That is, synchronizing clocks on the fly is virtually impossible, since during the clock synchronization using either of the synchronization algorithms, we use two-way (round-trip) traffic which may need all eight timestamps; on the other hand, one-way traffic is used to measure the message delays.

The clock difference (CD) constantly changes, mainly due to different clock drift rates (CDR). Even the CDR is not a constant and varies a little as the the environmental conditions of the machine change. We must be able to keep track of the clock difference and its variance caused by different clock drift rates, so that measurements are more accurate.

More precisely, we need to add an adjustment to the clock difference after the initial synchronization. This adjustment can be positive or negative, depending upon which clock is ahead and the relative values of clock drift rates. Details are given below.

We start from a reference point, record the clock difference at that instant, and add an adjusted amount to the clock which has a slower clock drift rate to make up the drifted difference. The

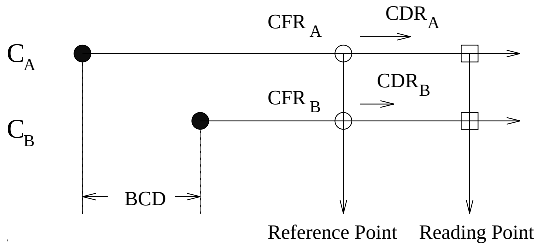


Figure 11: Adjustment of Clock Readings

effect is as if we keep the clock difference as a constant. Refer to Figure 11. The clock difference estimation at the reference point provides a base clock difference (BCD), and two clock face readings (CFRs). The different clock drift rates will make the clock difference vary afterwards.

Depending upon the initial values of CFRs and the relative values of CDRs, we have the following:

$$\begin{cases} CDR_A > CDR_B \Rightarrow \begin{cases} CFR_A \geq CFR_B \Rightarrow \text{CD increases} \\ CFR_A < CFR_B \Rightarrow \text{CD decreases} \end{cases} \\ CDR_A = CDR_B \Rightarrow \text{CD is unchanged} \\ CDR_A < CDR_B \Rightarrow \begin{cases} CFR_A > CFR_B \Rightarrow \text{CD decreases} \\ CFR_A \leq CFR_B \Rightarrow \text{CD increases} \end{cases} \end{cases} \tag{12}$$

$CFR_A > CFR_B$ means that CLK_A is ahead of CLK_B. It is not necessary to find out the the accurate clock drift rate (CDRs), instead only their difference is needed (i.e., the clock drift rate difference or CDRD). It can be obtained by running many tests and taking the average, since the CDR is not a constant.

When we save BCD at the reference point, we also save CFR_A as the reference point value. Suppose that we need to find out the clock difference after the reference point. We get the new clock face reading on machine A, CFR_A^{new}. Then the adjustment to clock difference (ACD) can be calculated as follows:

$$ACD = (CFR_A^{new} - CFR_A) \cdot CDRD.$$

Adding or subtracting ACD from BCD would give the estimated clock difference. Whether ACD is added or subtracted from BCD depends on whether the two clocks are drifting away from or towards each other. This can be determined by calculating a few CDs and comparing them.