ELEKTRİK, VOL.6, NO.2, 1998, ©TÜBİTAK EMO

A Hierarchical Interconnection Network Architecture for Real-Time Systems

Bülent Örencik
İstanbul Technical University Electrical & Electronics Eng. Faculty,
Ayazağa 80626 İstanbul-TURKEY
Phone No : 212 - 285 35 90
e-mail : orencik@cs.itu.edu.tr

Abstract

A hierarchical network suitable for interconnection of real-time client processes in a distributed multiprocessor environment is presented in this paper. A multi-layer Communication Unit (CU) prototype is developed for this purpose. This unit offers the client processes communication services for real-time operation. These services are structured in such a way that they do not depend on the characteristics of the communication medium. The basic hardware of the CU consists of a PC compatible card connected to the VME Bus. Client processes run on target cards (Motorola MVME-162) which are attached to the same bus. Target cards together with a CU form a node. Communication between nodes is handled by the CUs over an FDDI based hierarchical multi-ring structure. Routers handle the message transfer between these rings. A 3-layer protocol is designed for this purpose. The Network Service Layer (NSL), the lowest layer of this protocol provides connectionless communication and routing services for the Transport Layer (TL) using a real-time Timed Token Protocol. The establishment and release phases of these connections are managed by a novel dynamic bandwidth allocation scheme. The TL enables real-time end-to-end communication over established connections. The Session Layer (SL), the topmost layer of the protocol provides to client processes location independent connection-oriented services and necessary support for migration. The Management Unit (MU) of the node enables cooperation between these layers and keeps the real-time clock. Clock synchronization in the network is handled by a hierarchical clock maintenance mechanism.

Key words: Network Architecture, Real-Time Communication, Transport Protocol, Session Protocol, Timed Token Protocol, Routing.

1. Introduction

This paper summarizes the objectives, paradigms and main results of the project "An Optical Interconnection Network Prototype For Distributed Multiprocessor Architectures (EEEAG-BAĞ3)" which was supported by TÜBİTAK (The Scientific and Technical Research Council of Turkey) and by İTÜ-KOSGEB (İstanbul Technical University Small & Medium Industry Development Organization), between 1993 and 1996 [1]. EEEAG-BAĞ3 was carried out with the following objectives:

To meet the requirements of real-time systems (*deterministic* network) [2, 3].

To meet the demands of hierarchical industrial systems (*scalable* network) [4].

To ensure high noise immunity crucial for operations in factory environment (*optical* network).

To cope both with periodic (speech and video applications) and non-periodic (signaling systems) message transfers (*timed token protocols, Fiber Distributed Data Interface* - FDDI) [5].

To reduce the design complexity (*layered* structure).

To design a cost effective network (by using easy-to-get network components).

Interprocessor communication network design for distributed real-time systems is a wide research area of intellectually challenging computer science problems with direct payoffs to current technology. But results have been few, and designers presently have little to enable them to handle the timing constraints of real-time systems effectively. In this context Goscinski's book [6] should be considered an important reference which describes most of the practical work realized in the eighties. The communication models of distributed operating systems like Accent, V, Charlotte, Amoeba, Eden, LOCUS and Mach, most of which have some unsuitable features for real-time applications, can be found in this reference. In the nineties new communication models for real-time distributed systems were proposed [7-10].

In the project EEEAG-BAĞ3 communication hardware and software of a network suitable for real-time distributed systems were developed [11, 12]. Processes of the distributed system (called client processes) can access this communication medium by using different functions. When needed new functions can be added to the library.

The developed software has a layered structure. If classified according to the Open System Interconnection (OSI) model, these layers are a) Session Layer (SL), b) Transport Layer (TL), c) Network Service Layer (NSL). Any kind of information exchange between client processes is realized over those created by the SL. These links are similar to those defined in the Charlotte communication model [13, 14] with additional parameters suitable for real-time operation. Information about the established links is kept in a central database in the system.

We designed a new Transport Layer (TL) suitable for high speed communication media below the SL [15, 16]. This layer is connection oriented. It provides loss recovery using the selective repeat method. It also supports flow control. The TL accomplishes the information exchange with the Network Service Layer (NSL) which operates below the TL, managing an up-stream and a down-stream message-flow-queue for each connection. These queues are handled according to the nature of the related message-flow (periodic real-time, non-periodic real-time, non-real-time).

The network service layer is an integrated entity which consists of the Physical Layer (PL), the Medium Access Control Layer (MAC) and the Routing & Frame Control Layer (R&FCL). It provides connectionless service. The NSL supports the FDDI protocol [17 - 19].

Clock synchronization is very important in real-time distributed systems to measure the duration of distributed activities and to coordinate the actions of different processors. In the nineties there have been many publications on this problem, most of which attempted to keep the relative deviation between individual clock values smaller than a given threshold value [20-25]. In our project we designed for hierarchical FDDI ring networks a new clock synchronization protocol with negligible overhead.

To realize a scalable network we designed gateways between subnetworks of different hierarchical levels. These gateways are also known as routers as they decode network addresses and route the information to the destination. The routers can handle frame passing between FDDI rings of the same level or of different

hierarchy levels. Another task of the router is the management of the bandwidth shares of the CUs located on the subrings. An entity of the router, the Bandwidth Supervising Entity (BSE), monitors the utilization rates of the allocated bandwidths to CUs and changes them dynamically according to their actual needs.

We used relatively cheap components in every stage of the hardware design. When new and faster network components are available improvements should be possible by making minor changes in MAC driver routines. In addition, by replacing solely the NSL of the CUs and routers, using new data transfer technologies (asynchronous transfer mode - ATM, distributed queue dual bus - DQDB) is also possible.

In this section the main motives for starting this project and the historical background are given. Section 2 discusses the network structure of the project. Features of the Network Service Layer are summarized in Section 3. A short description of our Transport Layer Protocol is given in Section 4. Section 5 describes the proposed Session Layer Protocol. An outline of the Management Unit can be found in Section 6. Section 7 outlines the structure of the routers in the network. A short summary of the project's software structure is given in Section 8. Section 9 is devoted to clarifying the features and limitations of the experimental system. The features of the simulated network architecture and test results are given in Section 10. Sections 11 and 12 are devoted to the summary of the contributions of the project and to the description of further research activities respectively.

2. The Hierarchical Network Structure

In the project EEEAG-BAĞ3 a deterministic, hierarchical, optical, layered network which can handle both periodic and non-periodic message transfer is designed with all its subproducts (communication units, routers, service access points for real-time nodes).

The structure of the network is shown in Figure 1. An FDDI backbone ring forms the top level of the hierarchy (FDDI ring level II). On this backbone there are r routers which enable interring passage. Each router can directly handle the frame passage between s attached subrings (FDDI ring level I) of the same cluster. If the frame is destined for another cluster, it is passed over a router-backbone-router chain to the related cluster. Each *Communication Unit* (CU) has a distinct static address consisting of the *cluster* number (log_2 r bits), *subring* number (log_2 s bits) and *node* number (log_2 t bits). In our implementation we used 16 bit network addresses with 6 bits allocated for the cluster number, 2 bits for the subring number and 8 bits for the node number (r=64, s=4, t=256). This decision limited the capacity of the entire network to 2^{16} CUs, organized in 64 clusters at most. Each cluster is limited to 4 subrings and 4 x 256 CUs. This static address structure enables a fast decoding mechanism in the Network Service Layers of the CUs and of the routers and we therefore preferred it in spite of its limitations compared with a dynamic address structure.

The main component of the network is the CU. This unit offers the client processes communication services for real-time operation. Client processes run on Motorola MVME-162 cards (P) connected to each other by the industry standard VME Bus. The CU on the other hand is realized on a VME PC (a PC equipped with an interface to the VME Bus). The structure of client processes was studied by Harmancı [26]. Only internode communication requests (a node consists of a VME PC and of 1-7 MVME-162 cards all attached to the same VME Bus) originated by client processes are handled by the CUs. A CU does not interfere with the communication between two client processes running on the same P or on two different Ps of the same node. This decision enabled us to realize a clear and relatively simple software structure for the CU.

3. The Network Service Layer (NSL)

The layered software structure of a Communication Unit (CU) and of a router is shown in Figure 2. The Network Service Layer NSL, the lowest one in this structure, matches the three bottom layers of OSI layered network model. NSL is responsible for handling the network addresses, evaluating the frames, emitting optical signals into the ring and extracting them from the ring. All message flow between the upper layers and the ring is coordinated according to the real-time needs of specific applications. The NSL consists of three sublayers: The Physical Layer (PL), the Medium Access Control Layer (MAC) and the Routing & Frame Control Layer (R&FCL).

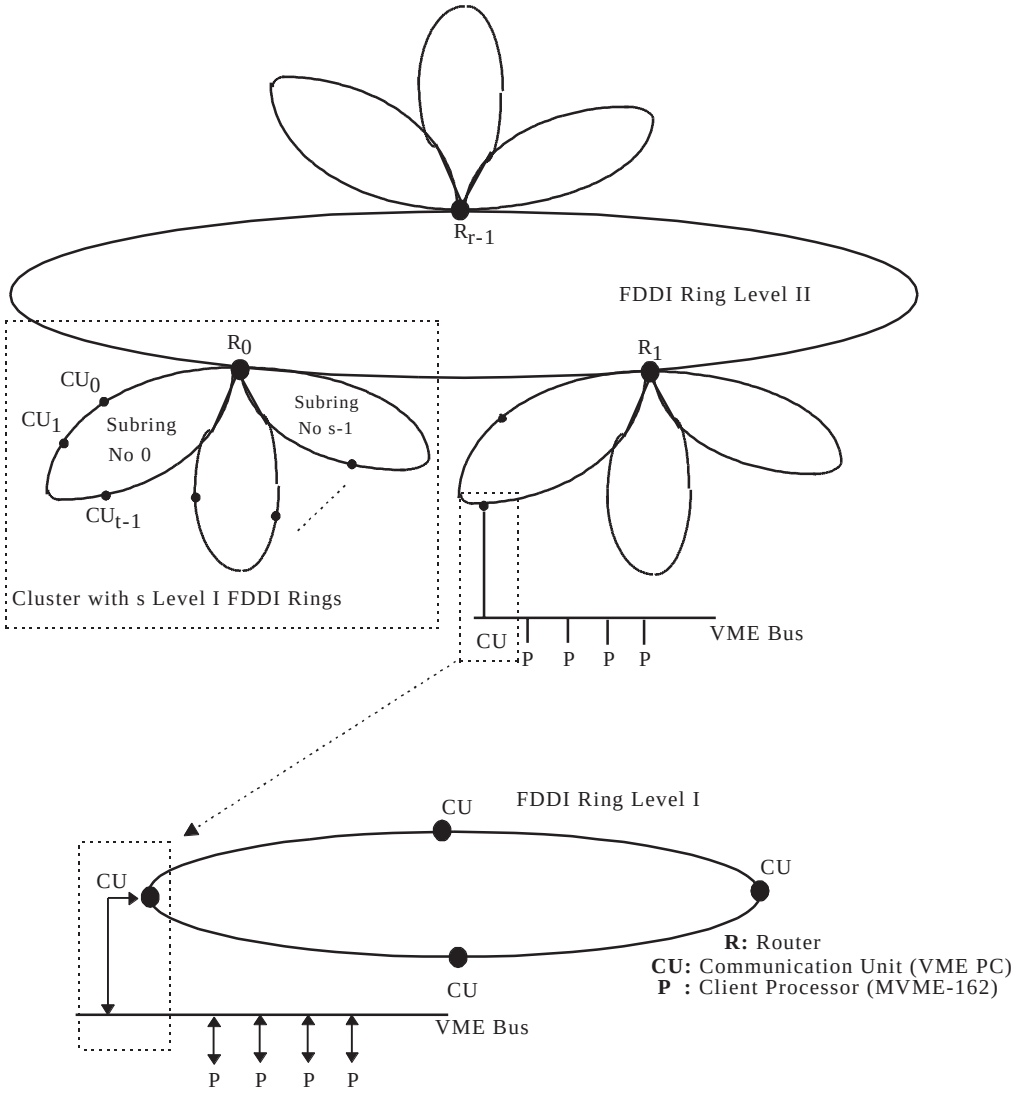


Figure 1. Hierarchical network structure of the project EEEAG - BAĞ3

3.1. The Physical Layer (PL)

This layer is functionally equivalent to the Physical & Physical Medium Dependent Layer of the Fiber Distributed Data Interface (FDDI) standard. The functions of the PL are executed by the chip DP83257

PLAYER+ on the FDDI development card DP83200MK attached to the VME PC. The receiver section of this chip obtains the clock signal from the optical receiver, detects the data, converts that from serial to parallel format and detects several connection failures. The transmitter section accomplishes parallel to serial conversion of data and submits the coded data to the optical transmitter. The chip operates in conformity with the standard FDDI X3T9.5. More information on DP83257 PLAYER+ and DP83200MK can be obtained from the references [27, 28].

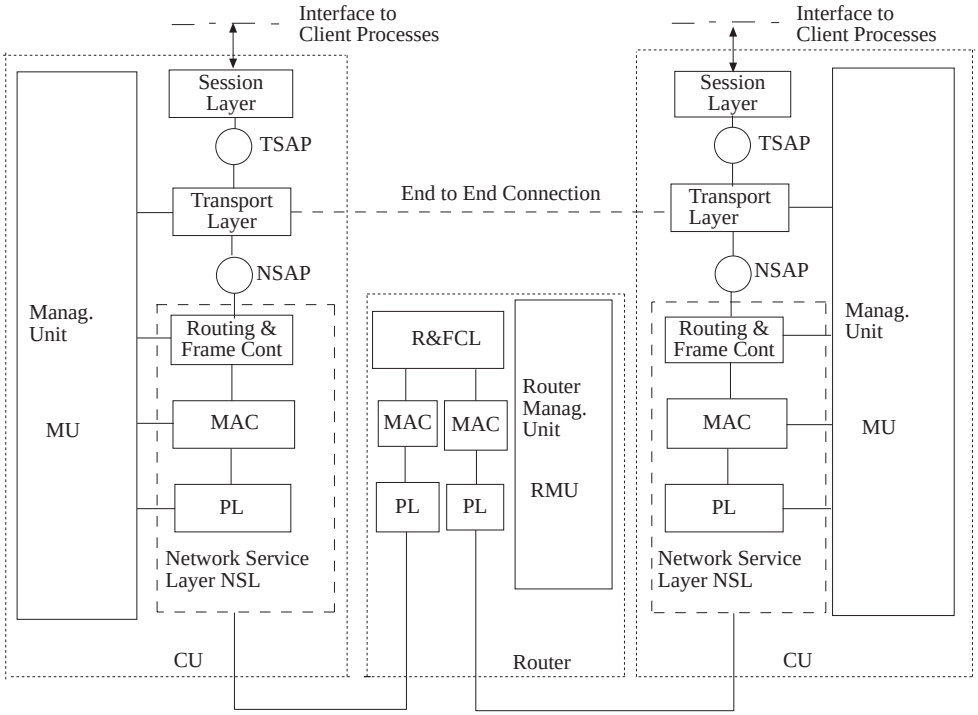


Figure 2. Layered software structure of the CU and of a router

3.2. The Medium Access Control Layer (MAC)

The MAC schedules and performs all data transfers on the ring based on the timed token protocol of the FDDI standard that provides a guaranteed amount of channel bandwidth to support timely delivery of interprocess messages. MAC functions are integrated on the chip DP83266 MACSI (Media Access Controller and System Interface), another component of the development card DP83200MK [29]. The receiver of the MACSI evaluates the parallel data produced by PLAYER+ and detects the frame type, the start and stop fields of the frame and checks whether the frame bears its address in the destination field. If so, it copies the information part of the frame in a buffer. The transmitter of the MACSI is responsible for catching a token and putting the frames through PLAYER+ on the ring as long as the time reserved for it has not expired. At the end of the reserved time (or as soon as it has exhausted all of the waiting frames) it generates a token to deliver the transmission turn to the next node on the ring.

Our research about the MAC layer has mainly focused on the optimization of *synchronous* transmission time periods of the nodes. Synchronous time (Hi) is the time allocated to the node i to inject into the ring the real-time data which has to be delivered to the destination in a restricted time interval. Periodic messages, such as sampled/digitized voice and video data, which arrive at regular intervals and have delivery time constraints, use synchronous time. Some deadline constrained non-periodic messages caused by alarms

and interrupts can also be considered in this category. If insufficient synchronous time were assigned, the percentage of the submitted data from the user between two token arrival instants to the actually transmitted data during the allocated synchronous time period would exceed 1. This would cause a buffer overflow in the node and loss of data waiting for transmission. On the other hand if the synchronous time were assigned longer than necessary, the rotation time of the token would become excessive. The nodes may use the time remaining from synchronous activity for transmitting data without deadlines (ordinary data). This time period is called *asynchronous* transmission time. Excessive token rotation time may lenghten the waiting time of the node for the token. In this case some of the processes may not meet their time limitations because of asynchronous transmissions [30-35].

The FDDI MAC uses a *Timed Token Rotation* (TTR) protocol to control access to the medium. Under this protocol, each node measures the time that has elapsed since a token was last received. The initialization procedures establish the *Target Token Rotation Time* ($TTRT$) equal to the lowest value that is bid by any of the nodes.

In this work, the duration of the synchronous transmission time periods during the operation are controlled dynamically. In this approach one master node (actually an entity of the router called *Bandwidth Supervising Entity* - BSE) is responsible for evaluating the real need of the nodes on the ring and to share the constant total synchronous time between the nodes according to their needs. Initially, when the ring becomes active, an initial synchronous time period is allocated to each node, as is usual in FDDI protocol [36]. The BSE monitors the utilization rate of the allocated time for each transport connection and when this rate decreases for any of the connections below a threshold value, this connection may be dropped to meet the extra bandwidth demand of new connection requests. For this change there is no need to interrupt the operation of the ring, which would not be permissible in a real-time environment.

Another synchronous bandwidth allocation scheme which only uses information locally available to the node and the sum of the synchronous bandwidths allocated to other nodes has been proposed [37-39]. The limited use of global data enables the elimination of a central BSE and makes the new scheme flexible and suitable for use in dynamic environments.

3.3. The Routing & Frame Control Layer (R&FCL)

The R&FCL sublayer, the top sublayer of MAC, provides connectionless datagram service to the Transport Layer (TL). R&FCL determines the MAC address which will be placed in the frame, checking the destination network address of the incoming primitive. Nodes sharing the same ring are addressed directly while for a node located on another ring, the MAC address of the router should be used. The frame type is determined to be synchronous or asynchronous depending on the priority and on the time-critical nature of data. The frame that is built is placed in one of the two outgoing (synchronous or asynchronous) frame queues to be delivered to the MACSI. Data extracted from the incoming frames over the ring are put by the MACSI into one of the two incoming frame queues depending on the synchronous or asynchronous type of the frame. These 4 queues (2 for incoming and 2 for outgoing frames) accommodate all the information flow between MAC and R&FCL.

Messages arriving at the R&FCL can be classified in 3 groups [40]:

a) *Real-time periodic* messages: Most video, audio and animation applications produce a stream of messages of this category. Taking t_{pj} as the period of the j^{th} message stream; t_{cj} as the total required time to inject one message of the j^{th} stream into the ring; u_j as the number of the routers which are involved in the transfer of the j^{th} message stream (this value may be 0, 1 or 2 in our network architecture depending

on the path of the stream); q_j as the minimum number of tokens arrived during one message period of the j^{th} message stream; τ as the sum of the minimum time allocated for asynchronous messages at each token rotation and of the token walk time around the ring; H_i as the maximum time that the i^{th} node on the ring is allowed to transmit its synchronous messages every time it receives the token; n as the number of nodes on a ring; t_{lj} as the message latency (end-to-end delay) of the j^{th} stream, we can calculate t_{lj} as follows:

It has been shown that the maximum amount of time that may pass between two consecutive token arrivals at a node is upper bounded with 2 $TTRT$ [41]. This bound holds regardless of the behavior of asynchronous messages in the network. The FDDI protocol also stipulates that the total bandwidth allocated to synchronous messages must be less than the available bandwidth; that is,

$$\sum_{i=1}^{n} H_i \leq TTRT - \tau. \tag{1}$$

Assuming that $t_{pj} \geq 2\ TTRT$,

$$q_j = \left\lfloor \frac{t_{pj}}{TTRT} \right\rfloor - 1 \tag{2}$$

We further assume that the q_j^{th} token arrived should be the last token used for the delivery of the message. To minimize the synchronous bandwidth $\sum_{i=1}^{n} H_i$ it is necessary to inject the message into the ring at each token arrival for the time t_{qj}, where

$$t_{qj} = t_{cj}/q_j \quad [32]. \tag{3}$$

With the above assumptions it can be shown that

$$t_{lj} \leq a_j + b_j \tag{4}$$

where

$$a_j = (2u_j + q_j + 1)TTRT; b_j = (u_j + 1)t_{cj}/q_j. \tag{5}$$

a_j is the upper bound on the total time elapsed between q_j consecutive token arrivals at the source node and eventually at the routers on the path between source and destination nodes. The value of a_j can be derived from an analysis given by Zhang and Burns [32]. b_j is the total injection time of the last portion of the message by the source node and by the related routers.

All the periodic messages are handled by the NSL according to the strategy given above so that at each token arrival each periodic message is sent partially using the formula given in Eq.(3). The synchronous bandwidth allocated to the i^{th} node should be sufficient to transmit $\sum_j t_{qj}$; that is,

$$H_i \geq \sum_j t_{qj}. \tag{6}$$

b) *Real-time non-periodic* messages: Many remote monitoring and control applications where measured data and alarms have to be delivered under some deadline constraints produce these type of messages. To set up the protocol parameters of the FDDI and to announce their bandwidth needs the nodes must sometimes broadcast short messages which should also be evaluated as messages of this category.

Assuming that such messages are transmitted using the first arriving token at the source node (and eventually at the routers) the upper bound of the message latency for the k^{th} message t_{lk} can be calculated as given below:

$$t_{lk} \leq a_k + b_k \tag{7}$$

where

$$a_k = 2(u_k + 1)TTRT;\ b_k = (u_k + 1)t_{ck}. \tag{8}$$

a_k is the upper bound on the total time elapsed until the arrival of the first token at the source node and eventually at the routers on the path between source and destination nodes. The expression for a_k can be directly derived from Eq.(5). b_k is the total injection time of the message by the source node and by the routers.

Messages in this category have the highest priority available. At each node an extra bandwidth has to be allocated which should be sufficient to transmit $\sum_k t_{ck}$, such that Eq.(6) becomes

$$H_i \geq \sum_j t_{qj} + \sum_k t_{ck}. \tag{9}$$

Although selecting the value for H_i as given in Eq.(9) guarantees that the node i will have sufficient time for transmitting all real-time periodic and non-periodic messages, it should not be considered as a good engineering approach actually to do so. It is worth noting that real-time non-periodic messages are created mostly by interrupts and alarms which do not happen frequently. Many applications may permit allocating a much smaller bandwidth than $\sum_k t_{ck}$. It is assumed here that the expected value of the total injection time of the real-time non-periodic messages transferred using the same token is much smaller than $\sum_k t_{ck}$.

In the EEEAG-BAĞ3 Project we decided not to repeat real-time periodic messages in case of a loss as most of the applications which produce these messages do not have such a requirement. On the other hand it is crucial to deliver real-time non-periodic messages to their destination without any loss. To accomplish this, the NSL gives the highest priority to messages of this category when preparing frames to send. A very useful feature of the FDDI protocol is that the status of the destination node is automatically inserted into the rotating frame. When the frame comes back to the source node, the NSL determines the result of its previous transmission (successful delivery, frame error, address fault, insufficient copy buffer at the destination, etc.). In the event of any unsuccessful transmission the real-time non-periodic message is repeated by the source at the next catch of the token. In the event of successful delivery at the r^{th} repeat of the message, Eq.(7) becomes

$$t'_{lk} \leq a_k + b_k + 3rTTRT. \tag{10}$$

When the source and destination nodes are located on different rings, the message must travel from the source to the destination via one router (if both nodes are in the same cluster) or two routers (if they are in different clusters). In the event of unsuccessful transmission the source router repeats the message as described in the above paragraph.

c) *Non-real-time* messages: These messages are mostly of a non-periodic nature. File transfer and e-mail traffic might be considered as typical applications creating non-real-time messages. The asynchronous bandwidth of FDDI is suitable for these kind of messages as this bandwidth is dynamically shared among all the nodes on the ring so that each node sooner or later gets the chance to transmit non-real-time messages.

Asynchronous bandwidth approaches to *TTRT*, when no node has real-time traffic; it diminishes to a very small bandwidth, which can be occupied by only one FDDI frame, when all the nodes use their synchronous bandwidth completely. The guaranteed delivery of non-real-time messages is handled by the Transport Layer (TL), which will be discussed in Section 4.

Details of the NSL hardware and software structure can be found elsewhere [1, 42].

4. The Transport Layer (TL)

Recent advances in fiber optic and VLSI technologies have enabled the design of ultra high-speed networks with very low bit error rates. Existing transport layer protocols (Transport Control Protocol - TCP, OSI Class 4 Transport Protocol - TP4, etc.) cannot keep pace with these emerging technologies. While significant work is taking place to update these protocols, new protocols adapted to high bandwidth communication media (light-weight protocols) are also being proposed [8, 43, 44]. In the framework of our project a transport layer protocol (TL) was designed keeping the underlying FDDI protocol under consideration.

4.1. The Connection Structure

TL is a *connection oriented* protocol. Each connection serves to transport messages from one sender to one receiver. The connection requests originating from the sender's upper layer (Session Layer - SL) bear the message type (real-time periodic, real-time non-periodic, non-real-time) [40] and the value of the demanded bandwidth as parameters. If the TL cannot provide the memory buffer required, the connection request is rejected. Otherwise the request is directed via a packet (Transport Protocol Data Unit - TPDU) called TL CONNECT REQUEST TPDU to the Bandwidth Supervising Entity (BSE) of the ring which will be discussed in Section 7. The BSE checks the request and if the provided bandwidth is below the demanded rate the connection request is rejected. In this case the BSE sends the BSE CONNECT REJECT TPDU to the sender's TL. But if the requested bandwidth can be provided, the request is forwarded by the BSE CONNECT REQUEST TPDU to the receiver node's TL. When the BSE CONNECT REQUEST TPDU arrives at the receiver, the TL checks its memory reserve to find out whether it can allocate the necessary memory buffer to the connection. If it can, the connection is recorded in a table called SL_TL Table which is used mutually by the SL and the TL, and the SL is notified of the established connection. Next, the receiver's TL entity sends the TL CONNECT ACK TPDU to the peer entity, which also updates its own SL_TL Table and informs its upper layer about the acknowledgment of the connection request. Otherwise a TL CONNECT REJECT TPDU is sent to the BSE, which eventually forwards the reject information to the connection requester node via the BSE CONNECT REJECT TPDU. As before, the SL is notified of the result of the request.

The release of a connection is managed by the BSE in a similar way as its establishment except that the rejection of the disconnection request is not defined.

If the utilization percentage of a connection is below a threshold, the BSE may request that the source node's TL should release the connection. In this case the BSE sends the BSE DISCONNECT INITIALIZE TPDU to the related TL which, after having sent all of its remaining data, is forced to start the normal disconnection activity.

4.2. The Transport Protocol Data Unit (TPDU)

One fact that makes the existing protocols so processing intensive is the diversity of TPDU types used and the variable lengths of TPDU headers. To avoid this, only a single TPDU type is defined and is used for all purposes. The TPDU header is also fixed in order to quicken TPDU header parsing. Because the TPDU header is fixed, only some of the fields are meaningful at a time. This causes a loss of bandwidth, but this loss is insignificant when the increase in header processing speed is considered. All the other data structures used are also standardized to make memory access and allocation faster. The structure of a TPDU is given in Figure 3. In the TPDU the first 16 bits hold a unique connection identifier number. The first part of it (CONN_ID) is given at the connection setup by the TL entity which originates the connection. The second part (X_CONN_ID) is determined by the peer TL entity which accepts the connection. Next comes additional information (if any) which identify the type of TPDU (real-time periodic data, real-time non-periodic data, non-real-time data, connection, disconnection), the status of the sender, the TPDU sequence number, the TPDU length, the receive buffer status, etc. The rest is data with a maximum length of 996 bytes.

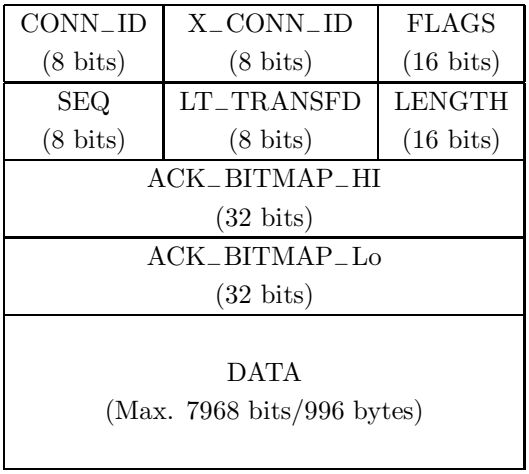


Figure 3. TPDU structure

4.3. Message Acknowledgment

In the EEEAG-BAĞ3 Project real-time periodic messages are not acknowledged at all. TPDUs for real-time non-periodic messages also do not need to be acknowledged as the FDDI frames transmitting them essentially handle the acknowledgment procedure - thanks to the ring architecture where each frame comes back to the source. On the other hand non-real-time messages are normally much longer than real-time non-periodic messages and hence they must be divided into many TPDUs. As the bandwidth provided for non-real-time messages is also typically very limited, most probably many successive tokens must be caught by the node until the entire non-real-time message can be transmitted. So there is a need for these messages to develop an improved acknowledgment mechanism which can only be provided by a transport layer.

In *lightweight* protocols it has become more important for the two sides of the connection to be completely aware of the state of each other. The full state information exchange is a common feature of many *highspeed* protocols [45]. Since the protocol processing tends to be more complicated at the receiving side, it seems more beneficial to let the sending side manage the flow of control information. In our project acknowledgment information is generated at the receiving side upon receipt of a command from the sending side [15].

The acknowledgment information bears the number of the last TPDU submitted to the upper layer and a bitmap describing the missing TPDUs and those that are still waiting in the receiver buffer. This information can be considered a full reflection of the receiver status. In this bitmap, if the corresponding buffer slot contains a TPDU the bit is set to 1, otherwise it is set to 0. A sample acknowledgment bitmap is shown in Figure 4. Normally the receive buffer size is capable of holding 64 TPDUs. But, for simplicity in the figure, it is assumed that the buffer size is limited to 16 TPDUs. The receiving entity sends the starting pointer of the buffer with the sequence number of the last TPDU passed to the upper layer. When the sending entity receives this information, it removes the TPDUs with sequence numbers smaller than the last TPDU passed to the upper layer from its send buffer. If there is a missing TPDU, the sender retransmits it.

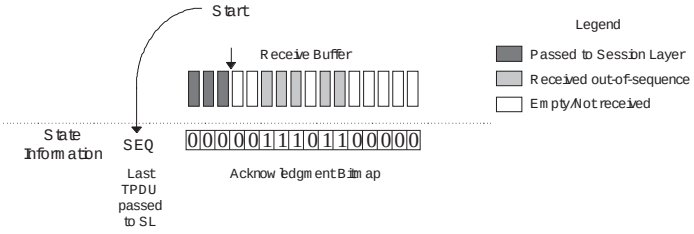


Figure 4. Acknowledgment bitmap

Loss recovery is an essential part of a transport protocol. It is assumed that the underlying network service layer provides error-free transmission of TPDUs and no TPDU with faulty data will arrive from the network. Thus, checksum fields are not used in EEEAG-BAĞ3. But TPDUs may be lost (dropped) because of the buffer limitation of terminal and router equipment. This fault can be corrected by acknowledging the lost TPDUs which will eventually be repeated by the sender.

4.4. The Finite State Machine (FSM) Model

The TL has been implemented using an FSM model [46]. In this model, every connection is assigned a separate FSM. There are a maximum of 255 FSMs running, corresponding to the maximum number of 255 connections. Every FSM accepts a primitive from an input queue as its input, performs an operation based on this input, writes a primitive to one of the input queues of another layer and makes transition to another state if necessary. All the machines are initially in their idle state, waiting for new connection requests. This is also the state into which all the machines go when the connection is lost or released.

The FSM states consist of an FSM (connection) number, next state pointers and a next state function and are defined as classes derived from a base class in the object oriented model. The base class is defined in C++ in Figure 5.

```
class base_state {
        public: unsigned char fsm_no; /* finite state machine (connection)
number */
            /* virtual next state function */
            virtual base_state * next_state (char /*primitive type*/,
                                             char* /*pointer to primitive parameters*/,
                                             int/*total length of parameters */) = 0;
    }
```

Figure 5. Definition of the base class in C++

In the EEEAG-BAĞ3 Project, communication between various layers is implemented using primitive queues. In order to activate an FSM, the next state function should be called giving a primitive as its parameter. The function returns the pointer to an instance of the base class, which is again a state, and this pointer is used for keeping track of the current state of the FSM.

Because the TL is seen as a single structure in the system, the input and output queues used to exchange primitives with other layers are common for all the FSMs. The problem of routing of incoming primitives is solved by using a distributer function (Figure 6). This function is also responsible for assigning new connection numbers when connection request TPDUs are received. In Figure 6, four of the FSMs (FSM 0, FSM 1, FSM 2 and FSM 3) are active while all the other FSMs are in their idle state. Queue i is the NSL queue for FSM i. Only incoming queues to the TL are shown in this figure while outgoing queues are ignored for the simplicity of the figure.

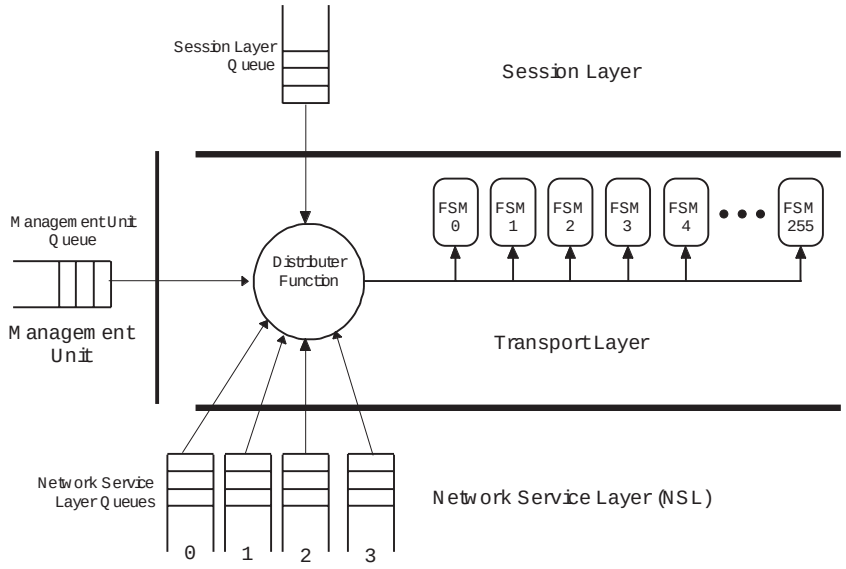


Figure 6. The role of distributer function in routing incoming primitives to FSMs

4.5. The NSL Services Access Point (NSAP)

There are 2 service primitives defined at the NSAP:

a) N_UNITDATA.req (TPDU pointer, length of TPDU, destination NSAP address): This primitive is used to send a TPDU to a destination node.

b) N_UNITDATA.ind (TPDU pointer, length of TPDU): This primitive is used to activate the related FSM with the arrived TPDU.

4.6. The Algorithm of the Transport Layer

The functioning of a TL entity is shown in Figure 7 in pseudocode to summarize its operation and to focus on its features. Note that there are as many operational TL entities as there are connections.

```
Procedure Transport_Entity;
Const
  max = 3;
```

```
Type
  message_type = (real_time, non_real_time);
Var
  repeat_count_data : integer;
  repeat_count_conn : integer;
  repeat_count_disc : integer;
  connection : message_type;
  wait_for_ack_timer : integer;
  wait_for_connect_timer : integer;
  wait_for_disconnect_timer : integer;
begin
   if SL requests a new connection by issuing T_CONNECT.req {Note 1}at TSAP then
   begin
      if necessary buffer allocatable then
      begin
         start the negotiation for the establishment of a new connection with
         the requested bandwidth
         by sending  TL CONNECT REQUEST TPDU to the  BSE;          {Note 2}
         initialize repeat_count_conn;
         start wait_for_connect_timer;
      end
      else
         issue T_CONNECT.conf at TSAP; {connection request rejected}
   end;
   if BSE CONNECT REQUEST TPDU arrives then {Note 3}
   begin
      if necessary buffer allocatable then
      begin
         issue T_CONNECT.ind at TSAP;
         send TL CONNECT ACK TPDU to the peer; {reflect the acceptance of the peer's
         connection attempt}
         connection := type of the TL CONNECT ACK TPDU;
      end
      else
          send TL CONNECT REJECT TPDU to the BSE;
   end;
   if (TL CONNECT ACK TPDU) or (BSE CONNECT REJECT TPDU arrives) then
   begin
      stop wait_for_connect timer;
      if connection established then
          connection := type of the TL CONNECT ACK TPDU;
      issue T_CONNECT.conf at TSAP; {result of connection request}
   end;
   if SL issues T_DISCONNECT.req {Note 1} at TSAP then
```

```
begin
   if the connection is idle {there is no data send - receive activity
   on the connection} then
   begin
      send  TL DISCONNECT REQUEST TPDU to the BSE;
      initialize repeat_count_disc;
      start wait_for_disconnect_timer;
   end
   else {defer the request until the end of the send - receive activity}
      record the pending status of the disconnection request originated by the SL;
     {update the SL_TL Table}
end;
if BSE DISCONNECT REQUEST TPDU arrives then {this TPDU reflects a disconnection request
originated by the peer and forwarded by the BSE}
begin
   issue T_DISCONNECT.ind at TSAP;
   send TL DISCONNECT ACK TPDU to the peer;
end;
if TL DISCONNECT ACK TPDU arrives then {the connection can be safely terminated}
begin
   stop wait_for_disconnect_timer;
   issue T_DISCONNECT.conf at TSAP;
end;
if BSE DISCONNECT INITIALIZE TPDU arrives then {this TPDU reflects the decision of the
BSE to shut down the connection}
begin
   disable all further T_DATA.req primitives at TSAP; {update  the SL_TL Table}
   if the connection is idle then {check the SL_TL Table}
   begin
      send TL DISCONNECT REQUEST TPDU to the BSE;
      initialize repeat_count_disc;
      start wait_for_disconnect_timer;
   end
   else {defer the request until the end of the send - receive activity}
      record the pending status of the disconnection request originated by the
      BSE; {update the SL_TL Table}
   end;
end;
if SL issues T_DATA.req {Note 1} at TSAP then {a new send request arrived}
begin
   disable all further T_DATA.req primitives at TSAP; {update the SL_TL Table}
   if connection = non_real_time then
   begin {only non_real_time messages should be acknowledged by the peer}
      while (send_buffer not full) or (last TPDU of the message not already in it) do
```

```
            put the next TPDU of the message into the send_buffer;
         send the send_buffer content to the peer accompanied by
         an acknowledge request; {Note 4}
         initialize repeat_count_data;
         start wait_for_ack_timer;
      end {non_real_time connection}
      else
      begin {send the whole real_time message}
         while last TPDU of the message not already sent do
         begin
         while (send_buffer not full) or (last TPDU of the message not already in it) do
           put the next TPDU of the message into the send_buffer;
         send the send_buffer content to the peer;
         end;
         issue T_DATA.ind at TSAP;
         if the disconnection request originated by the BSE or by the SL is pending then
         begin {give back the allocated bandwidth to the BSE}
             send TL DISCONNECT REQUEST TPDU to the BSE;
             initialize repeat_count_disc;
             start wait_for_disconnect_timer;
          end
          else
             enable further T_DATA.req primitives at TSAP;{update the SL_TL Table}
          end;
       end; {real_time_connection}
end;
if DATA TPDU received from the peer entity then
begin
  put the DATA TPDU into receive_buffer;
  if there are contiguous DATA TPDUs in the receive_buffer then
  begin
     resume the contiguous DATA TPDUs to the SL; {store them into the receive space
     allocated by the client process}
     update the receive_buffer;
  end;
  if DATA TPDU contains an acknowledgment request then
     send acknowledgment bitmap to the peer; {DATA ACK TPDU}
  if all DATA TPDUs including the last one related to the
  peer's last T_DATA.req have been
  received successfully then
  begin
     update the SL_TL Table;
     issue T_DATA.ind at TSAP;
  end;
```

```
end; {DATA TPDU received from the peer}
if DATA ACK TPDU received then
begin
   stop wait_for_ack_timer;
   update the send_buffer;
   if all TPDUs related to the last T_DATA.req successfully sent then
   begin
      issue T_DATA.ind at TSAP;
      if the disconnection request originated by the BSE
      or by the SL is pending then
      begin
         send TL DISCONNECT REQUEST TPDU to the BSE;
         initialize repeat_count_disc;
         start wait_for_disconnect_timer;
      end
      else
         enable further T_DATA.req primitives at TSAP; {update the SL_TL Table}
      end;
  end {all TPDUs successfully sent}
  else
  begin
     while (send_buffer not full) or (last TPDU of the message not already in it) do
       put the next TPDU of the message into the send_buffer;
     send the send_buffer content to the peer accompanied by an acknowledge request;
     initialize repeat_count_data;
     start wait_for_ack_timer;
  end; {all TPDUs related to the last T_DATA.req are not sent completely}
end; { DATA ACK TPDU received from the peer}
if wait_for_ack_timer expires then
    if (repeat_count_data) < (max) then
    begin
       repeat the send_buffer content;
       increment the repeat_count_data;
       start wait_for_ack_timer;
     end
     else
     begin {communication error}
        update the SL_TL Table;
        issue T_ERROR.ind {Note 1}at TSAP;
     end;
if wait_for_connect_timer expires then
    if (repeat_count _conn) < (max) then
begin
       repeat the TL CONNECT REQUEST TPDU;
```

```
       increment the repeat_count_conn;
       start wait_for_connect_timer;
    end
    else
begin {communication error}
   update the SL_TL Table;
       issue T_ERROR.ind at TSAP;
end;
if wait_for_disconnect_timer expires then
if (repeat_count _disc) < (max) then
    begin
       repeat the TL DISCONNECT REQUEST TPDU;
       increment the repeat_count_disc;
       start wait_for_disconnect_timer;
     end
     else
        begin {communication error}
             update the SL_TL Table;
                issue T_ERROR.ind at TSAP;
        end;
end; {Transport_Entity}

Note 1: Primitives (T_CONNECT, T_DISCONNECT, T_DATA, T_ERROR)
        exchanged between the TL and the Session Layer (SL) at Transport
        Services Access Point (TSAP) will be discussed in Section 5.1.

Note 2 : Sending a TPDU is accomplished by issuing an N_UNITDATA.req
         primitive at NSAP.

Note 3 : The NSL informs the TL about the arrival of a TPDU by
         issuing an N_UNITDATA.ind primitive at NSAP.

Note 4 : Sending the send buffer content is accomplished by issuing
         an N_UNITDATA.req primitive at NSAP for each TPDU in the send
         buffer.
```

Figure 7. Basic algorithm of the Transport Layer (TL) Entity

4.7. A Comparison of the TL With Some Other Transport Protocols

In Table 1, the TL [15, 1, 48] is compared with two lightweight protocols XTP and SNR, and two traditional protocols TCP and OSI/TP4. Detailed information about these protocols can be found in the literature [43, 44, 47].

5. The Session Layer (SL)

We designed a *Session Layer* (SL) above the TL. There are two important tasks that the SL is expected to perform:

To provide location independent connection oriented communication service to client processes.

To provide necessary support for process migration.

The design principles for the SL are listed below [49]:

The SL should use the services of the TL to perform its actions.

A unidirectional communication path should be established between a sender client process and a receiver client process. This path should be called a *link*.

Systemwide consistency should be assured by a global database that keeps all the information about all the links in the system.

The operation should be request-response type. All actions should be initiated by a request given by the client process and after completion of an action the necessary response should be produced by the SL. The SL should not send any signals, except the responses, to the client process. The memory space used for request-response exchange between the client process and the SL should be allocated by the client process when the request is initiated. This space should contain the parameters of the request and empty fields for the future responses of the SL. Some fields should also be allocated to record any unusual behavior of the network (network errors, unexpected shutdown of the connection, etc.).

The SL should not make any buffer allocation for send-receive activities. Necessary buffer spaces should be allocated by the client processes and the transmission should be accomplished directly from the memory space of one client process to the memory space of the other one.

A single Session Protocol Data Unit (SPDU) should be used for minimizing the processing overhead.

A wide variety of primitives should be offered to the client processes at the Session Services Access Point (SSAP) for creation and management of links, for data transmission and for gathering information about links.

5.1. The Transport Services Access Point (TSAP)

The TL provides reliable connection oriented communication service to the SL. The SL drives the TL with primitives that are similar to the corresponding primitives in the OSI Reference Model [47]. The following primitives are defined at TSAP:

T_CONNECT.req (connection identifier, peer address, message class, message characteristics): This primitive is used to set up a new transport connection between two transport entities. The requested message class and bandwidth is also declared. This request is forwarded to the Bandwidth Supervising Entity (BSE) of the ring, which will be discussed in Section 7.

T_CONNECT.ind (connection identifier, peer address, message class, message characteristics): This primitive is used to inform the SL about the establishment of a connection originated by the peer entity.

T_CONNECT.conf (connection identifier, status): This primitive is used by the TL to give the results of a new connection request to the originating SL. The new connection is either successfully established or

not. The new connection attempt may fail due to the insufficient memory buffer of the local or remote TL entity or due to the lack of allocatable bandwidth. In the latter case the status also reflects the maximum provideable bandwidth.

T_DISCONNECT.req (connection identifier): This primitive is used to terminate a connection. The TPDU bearing this request is submitted to the BSE.

T_DISCONNECT.ind (connection identifier, status): This primitive informs the SL that the connection is being released. Its status parameter reflects the reason of the disconnection.

T_DISCONNECT.conf (connection identifier, status): Using this primitive the TL informs the SL that the connection is safely terminated. Its status parameter reflects the reason of the disconnection.

T_DATA.req. (connection identifier, data pointer, data length): The SL uses this primitive to send a message over a transport connection.

T_DATA.ind. (connection identifier, data pointer, data length): When the TL receives a message from its peer, it reassembles the data and signals the SL with this primitive. The sender's TL on the other hand, also issues this primitive to inform its upper layer that the message has been successfully sent.

T_ERROR.ind (connection identifier, reason): This primitive is used by the TL to notify any erroneous condition. It is produced when for instance a message could not be submitted to the peer entity.

Note that T_CONNECT.resp and T_DISCONNECT.resp primitives are not defined at TSAP, as a difference from the conventional OSI Model. It is assumed that the peer TL does the necessary actions without the intervention of its upper layer when a TPDU arrives requesting the connection or disconnection.

Table 1. Comparison of the TL with XTP, SNR, TCP, and OSI/TP4

Feature	TL	XTP	SNR	TCP	OSI/TP4
Number of Packet Types	1	2	3	1	9
Packet Header	16 bytes fixed	24 bytes fixed	24 bytes fixed	20 bytes variable	5 bytes variable
Connection Setup	2-way handshake	timer-based	3-way handshake	3-way handshake	3-way handshake
Connection Release	2-way handshake	3-way handshake	3-way handshake	3-way handshake	2-way handshake
User Data in Connection Setup Packet	No	Yes	No	No	Yes
Signaling	in-band	in-band	out-of-band	in-band	in-band
Piggyback	No	Yes	Yes	Yes	No
Stream	Messages	Bytes	Messages	Bytes	Messages
Window Size	fixed	variable	fixed	variable	variable
Flow Control	Yes	Yes	Yes	Yes	Optional
Rate Control	No	Yes	No	No	No
Important Data	N/A	N/A	N/A	Primitive	Urgent Flag
Acknowledgments	Sender dependent, based on commands by sender	Sender dependent, based on commands by sender	Sender independent	Sender dependent, based on data reception	Sender independent

5.2. The Link Structure

The SL uses links as the logical communication units. A link is a unidirectional logical channel between two client processes. Each link has two ends denoted by S and R. When two client processes attach themselves to the two ends of a link, the actual communication can begin. Only the process at the S end can send and the process at the R end can receive. In order to transmit data, the former process should execute a SEND primitive and the latter one should execute a RECEIVE primitive. Transmission starts when such a SEND-RECEIVE pair is matched. Activating a link causes the establishment of a new TL connection. When a client process migrates from one node to another all links attached to that process also travel with it. But as TL connections are node dependent, such a migration causes release of the related connections and establishment of new ones.

Since the links are location independent, the information about the links has to be consistent in the entire system. A dedicated machine in the system called the *Session Server* (SS) keeps all the information about the links in the *Global_Link_Table* (GLT). Because link information is stored in a single location, systemwide consistency is assured (Each link has a unique identifier- *link_id*).

5.3. The Session Services Access Point (SSAP)

The SL works in a request-response type operation. Before passing a request to the SL at SSAP, a client process allocates a block of memory called *Common Parameter Area* (CPA). It fills the related fields in this area and sends the address of CPA to the SL via the *Session Layer Service Request Queue* (SLSRQ). The SL fills the related fields in return to create a response. In CPA there is a special field called *Return_Code*. The SL continuously updates this field during the execution of the requests. The final value of Return_Code is either "*success*" or "*fail*". In this way the client processes can monitor the execution of the primitive if necessary. In a CPA, all the parameters used in SSAP requests can be found. But, only some of these parameters are meaningful related to the request being executed. This approach may increase the memory demand of the protocol, but it also results in much faster access to the parameters. SL does not buffer data. The data is copied from one process address space to its peer's address space. Therefore, SL does not need to copy the data. This quickens the message transmission and brings flexibility to the processes since SL does not impose any restrictions on the message size.

The CPA can also be used by the client process to monitor the connection establishment originated by the peer entity and the connection release originated either by the peer entity or by the BSE. Note that these are activities not initiated by the local client process.

The functioning of the SL entity is shown in Figure 8 in the form of an algorithmic language to summarize its operation and to focus on its features. Note that there are as many operational SL entities as there are links.

```
Procedure Session_Entity;
Var
  connected : boolean;
begin
   case primitive at SSAP of
      Open_Link:
      begin
         create a link with unique link_id and record information about the created link
```

```
   to the
   Global_Link_Table (GLT) at the Session Server (SS); {SPDU packet
   exchange necessary between the node and SS}
   update the Common Parameter Area (CPA); {acknowledge the Open_Link request}
end;
Close_Link:
begin
   if connected then {check the SL_TL Table}
       issue T_DISCONNECT.req at TSAP;
   else
   begin
      terminate the link and update the GLT; {SPDU packet exchange necessary
      between the node and SS}
      update the CPA; {acknowledge the Close_Link request}
   end;
 end;
 Attach_Link:
 begin
   attach the client process to the requested end (S or R) of an existing
   link and update the
   GLT; {SPDU packet exchange necessary between the node and SS}
   update the CPA; {acknowledge the Attach_Link request}
end;
Detach_Link:
begin
   if connected then {check the SL_TL Table}
       issue T_DISCONNECT.req at TSAP;
   else
   begin
       detach the client process from the link and update the GLT;
       {SPDU packet exchange necessary between the node and SS}
       update the CPA; {acknowledge the Detach_Link request}
   end;
end;
Suspend_Link:
begin
   if connected then {check the SL_TL Table}
       issue T_DISCONNECT.req at TSAP;
   else
   begin
      suspend the link and update the GLT; {SPDU packet exchange
      necessary between the node and SS}
      update the CPA; {acknowledge the Suspend_Link request}
   end;
```

```
end;
Resume_Link:
begin
   resume the link and update the GLT; {SPDU packet exchange necessary
   between the node and SS}
   update the CPA; {acknowledge the Resume_Link request}
end;
Activate_Link:
begin
   if (attached at the S end of the link) and not (connected) then
   {check the SL_TL Table}
   issue T_CONNECT.req at TSAP;
else
   update the CPA; {reject the Activate_Link request}
end;
Deactivate_Link:
begin
   if connected then {check the SL_TL Table}
       issue T_DISCONNECT.req;
   else
      update the CPA; {acknowledge the Deactivate_Link request}
end;
Send:
begin
   if (attached at the S end of the link) and (connected) then {check the SL_TL Table}
     if there is no active Send request on this link then {check the SL_TL Table}
        if (size of the receive space allocated at the other end of the link) >
        (amount of data to be transmitted) then {this information is obtained
        by exchanging SPDUs between the peer SL entities}
        issue T_DATA.req at TSAP;
 else
    update the CPA; {reject the Send request}
end;
Send_Report:
   return the amount of data that is still waiting to be transmitted;
   {updating the CPA}
Remote_Receive:
begin
   if attached at the S end of the link then
       return the size of the receive space allocated at the other end of the link;
      {SPDU exchange between the peer SL entities is necessary}
else
       update the CPA; {reject the Remote_Receive request}
end;
```

```
     Receive:
     begin
        if attached at the R end of the link then
           accept the size of the receive space and the pointer to it; {updating the CPA}
        else
           update the CPA; {reject the Receive request}
     end;
     Receive_Report:
       return the size of the remaining receive space; {updating the CPA}
     Receive_Cancel:
     begin
        if attached at the R end of the link then
        begin
           if there is an active Send request of the peer then
                force the peer to abort the sending activity;
                {SPDU exchange between the peer SL entities is necessary}
            abort the receiving activity; {updating the CPA}
         end
         else
             update the CPA; {reject the Receive_Cancel request}
     end;
     Link_Status:
        return the information about the particular link recorded in the GLT;
        {SPDU packet exchange necessary between the node and SS}
       Max_Quality:
            return the bandwidth allocated to the particular link;
            {accessing the SL_TL Table and updating the CPA}
       Transfer_Time:
             return the time necessary to transmit a given amount of data;
             {accessing the SL_TL Table and updating the CPA}
end; {primitive at SSAP}
case primitive at TSAP of
   T_CONNECT.conf:
     update the CPA; {acknowledge the Activate_Link request}
     update the CPA related to the Attach_Link request;
     {this CPA always reflects the recent status of the connection}
   T_DISCONNECT.conf:
   begin
      if local Close_Link, Detach_Link or Suspend_Link request is in progress then
         update the GLT; {SPDU packet exchange necessary between the node and SS to record
         the status of the link (closed, detached or suspended) to GLT}
      if local Close_Link, Detach_Link, Suspend_Link or Deactivate_Link
      request is in progress then update the CPA related to the local request;
      update the CPA related to the Attach_Link request; {this CPA always reflects the recent
```

```
       status of the connection}
    end;
    T_DATA.ind:
      update the CPA; {pause the send/receive activity until the next matching of SEND -
         RECEIVE pair}
    T_ERROR.ind:
      update the CPA related to the Attach_Link request; {record the fault}
    T_CONNECT.ind:
      update the CPA related to the Attach_Link request;
      {this CPA always reflects the recent status of the connection}
    T_DISCONNECT.ind:
      update the CPA related to the Attach_Link request;
      {this CPA always reflects the recent status of the connection}
 end; {primitive at SSAP}
end; {Session_Entity}
```

Figure 8. Basic algorithm of the Session Layer (SL) Entity

5.4. The Session Protocol Data Unit (SPDU) Structure

SPDUs are the packets exchanged between two communicating session entities. Since the SL is designed to operate on high-speed networks, the protocol processing has to be minimized. In order to speed up packet processing, there is a single SPDU type with a fixed header. This approach considerably reduces the delay for SPDU parsing. There are several flags in the SPDU for signaling purposes. Flag processing is a very time consuming process. To reduce the delay for flag processing, flags related to each other are placed on consecutive bits in the SPDU. For example, the flags used for link management are kept separate from the flags used during data exchange. Every SPDU has the same packet format, but according to the flags in the SPDU header, certain fields in the SPDU are processed and certain fields are ignored.

5.5. Session Services Access Point (SSAP) Primitives

In our work we have designed many request primitives at the SSAP. These primitives can be classified in three categories: link management, data transmission and information retrieval.

5.5.1. Link Management

There are eight primitives for link management: *Open_Link, Close_Link, Attach_Link, Detach_Link, Suspend_Link, Resume_Link, Activate_Link* and *Deactivate_Link*. A client process can open and close a link, attach to the end of a link, detach from the end of a link, set up a transport connection on a link or abort the established connection. The SL allows the client processes to suspend a link. When a link is suspended, all activity on the link is stopped until the client process resumes it. The client process can resume a suspended link at a different location. This flexibility provides a great support for the migration of the client processes. A client process can suspend the link and migrate. When it arrives at its new destination, it can resume the link and continue normal operation. For the client process, it seems as if it carries the link with itself during migration.

5.5.2. Data Transmission

Data transmission is carried out by two major primitives: *Send* and *Receive*. For data transmission, the client process at the S end of a link must execute a Send primitive and the process at the R end has to execute a Receive primitive. There are four more primitives that can be used during data transmission: *Send_Report, Receive_Report, Remote_Receive* and *Receive_Cancel*. Send_Report and Receive_Report are used to monitor the transmission or reception operations. Remote_Receive can be used by the sending entity to query the size of the receiving area allocated at the receiving entity. Receive_Cancel is used to cancel a Receive request.

5.5.3. Information Retrieval

There are three primitives for information purposes: *Link_Status, Max_Quality* and *Transfer_Time*. Link_Status is a general purpose primitive used to query all kinds of information from the Session Server. Max_Quality is used to determine the bandwidth allocated to the link. Transfer_Time is used to find out the time required to transmit a certain amount of data over a specified link.

5.6. The Finite State Machines (FSMs) of Session Layer Entities

The SL was implemented with an FSM model. In this model, every link is assigned a separate FSM. There are a maximum of 255 FSMs running, corresponding to the maximum number of 255 links. This model is very similar to the one which was discussed in Section 4.4.

5.7. An Operation Scenario

In Figure 9 a scenario is given to exemplify the operations of the SL and the TL and the roles of the SS and the BSE in these operations. In this scenario there are 2 nodes A and B. A client process on node A sends a message which occupies more than one TPDU to node B. Then the sender process suspends the link and migrates to node C. It resumes the link there and establishes a new connection to continue with the sending activity.

The detailed structure of SL is given in the literature [1, 48 - 50].

6. The Management Unit of the Communication Unit (MU)

We designed in our project a *Management Unit* (MU) to control the operations of the NSL, the TL and the SL. The MU consists of many subunits with different functions. The tasks of these subunits are summarized below:

The *Ring Management Entity* (RME) is responsible for initiating recovery functions on detecting faults like duplicate addresses.

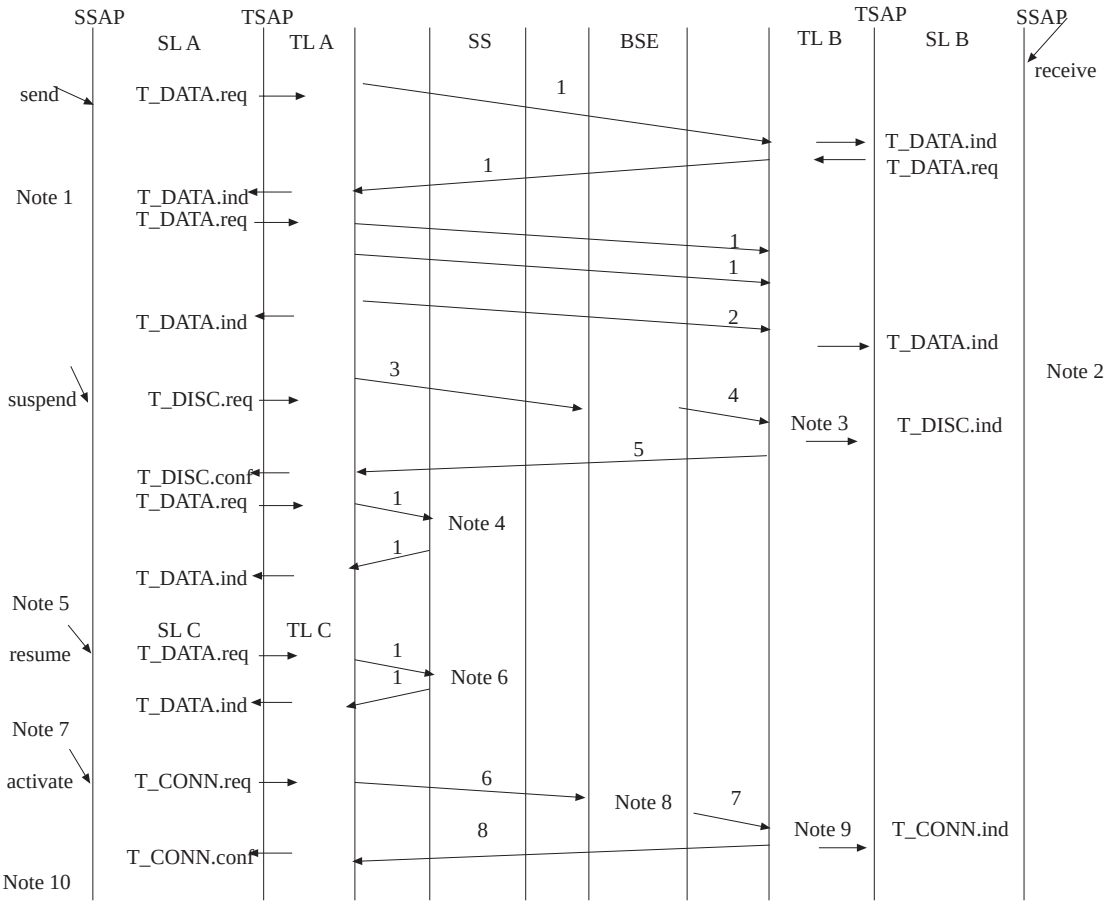
The *Frame Services Entity* (FSE) enables the MU to communicate with other MUs in the network.

The *Statistics Entity* (SE) keeps account of the error-free and error-prone frames.

The *Error Handler* (EH) processes the error indications coming from different layers.

The *Queue Management Entity* (QME) coordinates the activities in CU by allocating the communication processor to different layers and entity processes. This entity determines which layer to invoke, depending on which primitive queues in the system have data. It polls the primitive queues in a priority

order. If it detects that there is a primitive in a queue, it invokes that layer, which takes the primitive from this queue. The QME handles the data in higher priority queues first. When the higher priority queue is empty or a specified amount of primitive has already been taken from this queue, then the queue with the lower priority is served. The queues which belong to the MU have higher priority than other queues. For each TL connection, we create a new primitive queue between the TL and the NSL. The queue polling algorithm is similar to the FDDI token based access protocol. It has a circular service property with the period equal to $TTRT$. The j^{th} connection on the i^{th} node has an allocated synchronous bandwidth value of H_{ij}. The polling algorithm running on the i^{th} node enables the NSL on this node to read synchronous data with the length of H_{ij} in each $TTRT$ time unit from the queue of the j^{th} connection.



Note 1: Info about the remote receive space arrives
Note 2: Acknowledge the Receive request
Note 3: Info about the connection status
Note 4: Record the suspended link status to the GLT
Note 5: Acknowledge the Suspend_Link request
Note 6: Record the resumed link status to the GLT
Note 7: Acknowledge the Resume_Link request
Note 8: Requested bandwidth allocated by the BSE
Note 9: Info about the connection status
Note 10: Acknowledge the Activate_Link request

1: DATA TPDU 2: Last DATA TPDU 3: TL DISCONNECT REQUEST TPDU
4: BSE DISCONNECT REQUEST TPDU 5: TL DISCONNECT ACK TPDU
6: TL CONNECT REQUEST TPDU 7: BSE CONNECT REQUEST TPDU
8: TL CONNECT ACK TPDU

Figure 9. An operation scenario

The *Timer Entity* (TE) is a subunit of MU which is responsible for keeping the real-time clock and for correcting the clock deviations by exchanging clock information with other communication units' TEs. Especially in real-time systems the TE design is considered a very important task. Clock synchronization algorithms in the literature can be classified in two groups [51]:

Deterministic algorithms take into account the average value of the transfer delay of clock messages (messages bearing clock information between the TEs). These algorithms can be used in communication systems where the difference between the estimated and actual delays is relatively small [52, 53].

Probabilistic algorithms on the other hand use methods to find out whether the absolute delay value of the considered clock message exceeds an upper threshold limit. In this way these algorithms guarantee to use only clock messages with relatively small delays [54].

In both cases the elapsed time between the reading of the clock and outputting this value into the network and also the time between taking the clock message from the ring and evaluating it has to be kept as short as possible. To minimize the former time, whenever a clock message should be sent, MAC queues are inhibited from other accesses so that the clock message can access the ring without being delayed. To minimize the latter time the clock message is enabled to arrive at the destination TE through the most prioritized MAC receiving queue.

In our project the TE executes a deterministic algorithm. In this algorithm each node on the ring calculates the average clock value by using its own clock and the incoming clock information from other TEs on the ring and assigns this value to its clock. Excessive deviated clocks are not used in this calculation. In another form of this algorithm the Timer Entity on a master node called Clock Server (CS) makes the calculations and broadcasts the result to other nodes' TEs on the ring periodically, while these TEs eventually assign the broadcast value to their clocks. Note that a clock request frame injected into the ring from the Clock Server alerts all the TEs on this ring such that on the next token rotation each TE should put a special frame bearing its clock value on the ring. As all the clock values can be collected at the same tour of the token this synchronization method should be considered as a very effective one which can eliminate most errors due to variable network latency.

The above-mentioned clock synchronization methods are only valid for a single ring. To use them on the hierarchical ring network of the project (Figure 1), the TEs on the routers R_0, $R_1, \ldots, R_{r-1}$ were assigned as Clock Servers (CSs), each one responsible for the synchronization of the clocks on its subrings. The CS on each router periodically requests clock information from the TEs on its subrings and using the incoming information each CS calculates an average clock value as discussed in the paragraph above. Afterwards the CSs broadcast these values on the FDDI Ring Level II (backbone ring). Finding the average of the deviations of these broadcast clock values from their own clocks, the CSs finally obtain a global clock value which is eventually transferred to the TEs in the network.

MACSI and PLAYER+ chips which were used in our design provide many functions to facilitate the realization of the MU [1].

7. The Router Structure

We designed in this project *routers* which handle the message passage between hierarchical FDDI rings [55]. Only one router is involved in the communication between two nodes located on different subrings of the same cluster. The messages flow over two routers when the communicating nodes reside in different clusters. In a router attached to one FDDI Level II ring and s level I rings there are totally (s+1) PL layers, (s+1) MAC layers and 1 R&FCL layer operational (Figure 10).

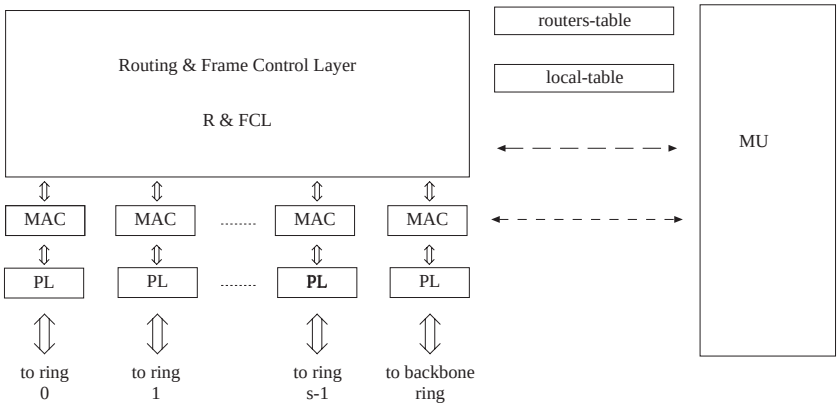


Figure 10. The structure of the router

The structure of the PLs and MACs in the router are very similar to that of their peers in the nodes. The single R&FCL in the router is responsible for checking the destination NSL address of the incoming frame and for constructing a new frame which bears the MAC address of the destination node, when the destination node is in the same cluster as the router. This address is obtained from the local_table using the subring number and the node number fields of the destination network address as search indexes of this table (Figure 11).

index	ring 0	ring 1	...	ring s-1
0	MAC $CU_{0,0}$	MAC $CU_{1,0}$	.....	MAC $CU_{s-1,0}$
1	MAC $CU_{0,1}$	MAC $CU_{1,1}$	.....	MAC $CU_{s-1,1}$
	:	:	:	:
t-1	MAC $CU_{0,t-1}$	MAC $CU_{1,t-1}$	.....	MAC $CU_{s-1,t-1}$

Figure 11. The local_table of the router

But when the destination node belongs to a different cluster, the MAC address of the router of the destination cluster will be located in the new frame. In this case the destination MAC address is obtained from the routers_table using the cluster number field of the destination network address as the search index of the table (Figure 12) [56].

index	MAC addresses of routers
0	MAC addr. of R_0
1	MAC addr. of R_1
:	
r-1	MAC addr. of R_{r-1}

Figure 12. The routers_table of the router

The Management Unit of a router has similar entities with the same functions as that of a node (RME, EH, QME). But these entities are multiplexed for each of the attached rings. The Timer Entity of the router is called the *Clock Server* (CS) which is responsible for the clock synchronization of the attached lower level rings. Its relation to the TE of a node is a typical master-slave relation which was discussed in Section 6.

The *Bandwidth Supervising Entity* (BSE) monitors the bandwidth utilization percentage of each node on the ring. When an extra bandwidth request arrives from the TL of a node to establish a new connection, the BSE checks the difference between the allocated and actually utilized bandwidths of each node on that ring. If there is enough non-utilized bandwidth in reserve, the bandwidth share of the requesting node is increased to meet the extra bandwidth need of the requester. This change may cause a decrease in the bandwidth share of some of the nodes on the ring which can only utilize some part of their shares. By sending some control TPDUs to the related nodes the BSE forces some of the non-utilized or not fully utilized connections to be released. Using the relinquished bandwidth the BSE eventually enables the requesting node to establish the new connection. This method can dynamically adapt the bandwidth allocation mechanism to the actual bandwidth utilization of the nodes.

The procedure becomes more complex when the source and destination nodes of the candidate connection are not located on the same ring but on different rings of a cluster. In this case the BSE has not only to allocate extra bandwidth to the source node, but it also has to extend its own bandwidth share on the ring on which the destination node is situated.

When the source and the destination nodes are not in the same cluster, two BSEs should be involved in the bandwidth regulation mechanism: one on the source node's ring and on the backbone ring (FDDI Ring Level II in Figure 1) and the other one on the destination node's ring.

More details about the designed router are given in the literature [1].

8. The Software Structure

The software of the project was written in C++. All layers in the system were designed as finite state machines. Each state of a machine is an object. The state objects were derived from a base object. The base object includes global data, which is shared between the state objects. Each state object has a transition function which defines the actions to perform in this state of the layer (Figure 13). There is a pointer for each layer which points to the current state. When a layer is invoked, the transition function of the current state is run. This function reads a primitive from its incoming queue, performs necessary actions and returns the value of the pointer to the next state. The key statement of the program is:

$$\text{current\_state\_pointer} = \text{current\_state\_pointer} \rightarrow \text{transition}(\text{queue\_number}); \tag{11}$$

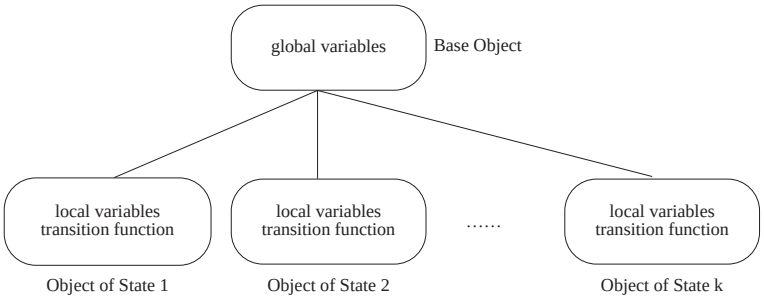


Figure 13. Derivation of state objects

All of the primitives exchanged between the layers and other entities are structured according to the definition given in Figure 14. This fixed length primitive structure contributes to achieving a simple and fast memory management scheme by preventing the memory from discontinuities. More details about the software structure are given in the literature [38, 1].

```
struct node_S{
         char type;                  //type of the primitive
          char *datpar_ptr;     //pointer to the primitive's data or parameters
          int length;                 //length of the data or parameters
          node_S *next;         //pointer to the next node
};
```

Figure 14. Structure of the primitives

9. An Experimental System

An experimental system consisting of a) 4 CUs on a ring (A, B, C, D) and b) 2 CUs on two different rings (A and B) and one interconnecting router (R) were built using 4 FDDI development cards (DP83200MK) attached to either VME PCs or to ordinary PCs. Test programs which pass different requests to the SL and evaluate returning responses in operation scenarios similar to that given in Figure 9 were used for this purpose. The following observations have been made [1]:

Because of the address overlapping the development cards, at most 2 of them can be attached to the ISA bus of a PC. As a consequence of this limitation, a router can handle the message transfer between only two FDDI rings.

Lack of direct memory access (DMA) support and limitation of the data transfer over the ISA bus to only 8 bits in width, are two other restrictions of the DP83200MK. Because of these limitations a frame of 1 kB length can be transferred between the CPU main memory and DP83200MK data buffer not less than in 1.1 ms (7.5 Mbps). Having only 4 development cards in hand we are far from even moderately loading the 100 Mbps FDDI bandwidth (maximum 15 Mbps, while there is a continuous message flow between source A and destination B and another message flow between source C and destination D).

The receiving entity is always the bottleneck of the system as any receiver in the system must be ready to accept messages from all of the sending entities in the network at the same time. Since the transfer rate over the ISA bus may not exceed 7.5 Mbps, each sender must limit its output rate to maximum 7.5/n Mbps, where n is the maximum number of the sending entities, so that even in the worst case no input buffer overflow of any receiver occurs (with n nodes, sending continuously messages to the same receiver).

10. Simulation of the Network Architecture

To test the network with more than 4 CUs and to overcome the hardware limitations of the FDDI development card we prepared a test program which can simulate any number of CU session layers on the same PC. In this program a multiplexing/demultiplexing software module enables these multiple SLs to direct their connection/disconnection requests to a common TL by sharing the same primitive queues at TSAP. This approach alone would not suffice for a thorough test of the network if the transfer rate bottleneck between the main board and the FDDI development card were not eliminated. Therefore, instead of transferring the

real data over the ISA bus we decided to pass for each data transfer request arriving at TSAP a direction to the development card to force it to send repeatedly the same 4 kB test pattern stored in its output buffer until a repeat counter decrements to zero.

With the help of these tricks and with only minor modifications of the software developed for the experimental system we succeeded in testing our network architecture with any number of simulated CUs at different loads.

The characteristics of some of the simulated cases are given in Tables 2-7 [37]. In these tables S denotes the source CU, T_c indicates the average time between two succeeding connection requests, T_d shows the average duration of a connection, D represents the destination unit (CU or the router R); P is taken as the data transfer period and C as the data transmission time in each P. At most one router was used throughout the tests and TTRT was selected as 50 ms in all cases.

Table 2. Features of case A

S	T_c[ms]	T_d[ms]	C[ms]	P[ms]	D
1	1000	1500	5	155	2
2	1500	1000	10	200	R
3	1000	1500	5	160	4
4	1600	2000	8	255	3

Table 3. Features of case B

S	T_c[ms]	T_d[ms]	C[ms]	P[ms]	D
1	800	1500	5	180	2
2	1000	1500	10	205	R
3	1000	1500	5	160	4
4	1600	2000	8	260	3

Table 4. Features of case C

S	T_c[ms]	T_d[ms]	C[ms]	P[ms]	D
1	800	1500	5	155	2
2	1000	1500	10	200	R
3	1000	1500	5	150	4
4	1600	2000	8	250	3

Table 5. Features of case D

S	T_c[ms]	T_d[ms]	C[ms]	P[ms]	D
1	800	1500	9	150	2
2	800	1500	10	200	R
3	1000	2500	8	150	4
4	1600	2000	8	250	3

Table 6. Features of case E

S	T_c[ms]	T_d[ms]	C[ms]	P[ms]	D
1	800	1500	9	150	2
2	800	1500	10	200	5
3	1000	2500	8	150	4
4	1500	2000	8	150	3
5	1600	2000	8	250	2
6	1000	1400	7	100	5

Table 7. Features of case F

S	T_c[ms]	T_d[ms]	C[ms]	P[ms]	D
1	800	1500	9	150	2
2	800	1500	10	200	5
3	1000	2500	8	150	4
4	1500	2000	8	150	3
5	1600	2000	8	250	2
6	1000	1400	7	100	5
7	900	2000	7	250	8
8	1100	1500	7	100	7

Case A represents a network with low connection request rate and low data transmission load. A working environment with a higher connection request rate than in Case A is simulated in Case B. Case C represents a network with a higher data transmission load than in Case A. Case D depicts an application with both higher connection request rate and higher data transmission load than in Case A. In these 4 tests 4 source CUs are simulated.

In the next case (Case E) a higher data transmission load is generated on the ring than in the former cases. Some of the connection requests have to be rejected by the BSE. An overloaded network with 8 CUs is simulated in Case F. The BSE has to reject a lot of connection requests.

The test results are shown in Table 8. U_{avg} denotes the ratio of the average utilized bandwidth to the maximum bandwidth of an FDDI ring calculated for the first 500 token tours. During this time N_r connection requests are directed to the BSE, while N_a of them are accepted. OVR_{BSE} represents the ratio of the connection/disconnection message exchange overhead to the average utilized bandwidth during the first 500 token tours. The value of OVR_{BSE} remains stable after the first 50 tours. The results demonstrate clearly that for a wide variety of traffic loads the overhead of the proposed dynamic bandwidth allocation scheme is negligible [37].

Table 8. Protocol overhead of the dynamic bandwidth allocation scheme

Case	U_{avg}	N_r	N_a	OVR$_{BSE}$
A	0.212	81	81	0.0017
B	0.272	94	94	0.0024
C	0.277	92	92	0.0020
D	0.447	103	98	0.0025
E	0.568	140	128	0.0047
F	0.696	190	152	0.0091

11. Conclusion

Main contributions of the project EEEAG-BAĞ3 are given below:

We developed a real-time, hierarchical, layered network architecture which can effectively handle both real-time and non-real-time message transfers and ensure high noise immunity. Standard off-the-shelf components were used for its implementation.

The token usage time of a node can be adapted dynamically to its varying needs by a novel bandwidth regulation mechanism. The implementation of the dynamic synchronous bandwidth regulation throughout a hierarchical FDDI multi-ring network has not been realized before. Simulation results stipulate the low overhead of this mechanism.

A new *lightweight* transport protocol was designed which can effectively handle real-time periodic, real-time non-periodic and non-real-time messages.

We realized a session layer protocol which handles the interconnection of user tasks through unidirectional "links". A central link data base is designed for link management.

Necessary support for process migration is provided using node independent links.

We developed a new deterministic clock synchronization protocol suitable for hierarchical FDDI rings.

12. Future Work

Based on our previous work in the project EEEAG-BAĞ3 we plan to work further on the following topics:

The session layer of the designed communication unit manages the interconnection between tasks by using one to one links. Only two tasks can be attached to one link. At the existing structure conference type communication can only be handled by creating links between each pair of the tasks which take part at the conference. To provide support for this type of communication we are now working on removing the limitation that at most two tasks can use a link simultaneously. Multicast and broadcast features of FDDI protocol will be used for this purpose.

We are currently working on the performance analysis of the communication network at loads of different characteristics (different periodic and non-periodic loads, data lengths and deadlines). The performance is mainly limited by the FDDI development card DP83200MK, which can access the VME PC memory at 7.5 Mbps at its peak transfer rate. When these cards can be replaced by PCI Bus compatible units which can handle 32 bit memory to memory transfer, the performance can increase to almost 100 Mbps, the peak bit injection/extraction rate of the FDDI ring.

For the time being we are using a single real-time quality of service parameter characterizing the link, namely the bandwidth (actual bit transfer rate). Other parameters may also be considered in this respect: latency, jitter (delay variation), loss etc. The analysis of the relation of these real-time parameters to the queue management strategies in the CU and in the router is one of our future research topics.

A research team in our department which is working on distributed operating systems is cooperating with us on a new interface design enabling the user tasks to access the primitives defined at SSAP more efficiently.

References

[1] B. Örencik, An Optical Interconnection Network Prototype for Distributed Multiprocessor Architectures, *Tech. Report of TÜBİTAK - EEEAG-BAĞ3,* (1997) (in Turkish).

[2] L. Sha L, S. Sathaye, A Systematic Approach to Designing Distributed Real-Time Systems, *IEEE Computer,* Vol: 26, No 9, 68-78, (1993).

[3] N. Swaminathan, W. Zhao, Issues in Building Real-Time Systems, *IEEE Software,* Vol: 9, No 5, (1992).

[4] M. Holliday, Z. Stumm, Performance Evaluation of Hierarchical Ring-Based Shared Memory Multiprocessors,*IEEE Trans. on Computers,* Vol: 43, No 1, 52-67, (1994).

[5] N. Malcolm, S. Kamat, W. Zhao, Real-Time Communication in FDDI Networks, *Journal of Real-Time Systems,* Vol: 10, No 1, (1996).

[6] A. Goscinski, *Distributed Operating Systems. The Logical Design,* Addison Wesley, (1992).

[7] S. Hinrichs, Simplifying Connection - Based Communication, *IEEE Parallel Distributed Technology,* Vol: 3, No 1, 25-36, (1995).

[8] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. Verma, H. Zhang, The Tenet Real-Time Protocol Suite: Design, Implementation and Experiences, *IEEE/ACM Trans. on Networking,* Vol: 4, No 1, 1-10, (1996).

[9] A. S. Tanenbaum, M. F. Kaashoek, H. E. Bal, Parallel Programming Using Shared Objects and Broadcasting, *IEEE Computer,* Vol: 25, No 8, 10-19, (1992).

[10] P. Steenkiste, A Systematic Approach to Host Interface Design for High-Speed Networks, *IEEE Computer,* Mar. 1994, 47-57, (1994).

[11] B. Örencik, Overview of a Hierarchical Network for Real Time Distributed Interprocess Communication, *8th Symp. on Computer and Microprocessor Applications (mP'94),* Budapest-Hungary, 357-361, (1994).

[12] B. Örencik, F. Buzluca, E. Harmancı, O. Aliefendioğlu, C. Özden, O. Akgün, A Hierarchical Interconnection Network for Real Time Distributed Multiprocessors Architectures, *9th Int'l Symp. on Computer and Information Sciences (ISCIS IX),* Antalya, 361-368, (1994).

[13] Y. Artsy, H. Chang, R. Finkel, Interprocess Communication in Charlotte, *IEEE Software,* Jan., 1987, 22-28, (1987).

[14] R. Finkel, M. Scott, Y. Artsy, H. Chang, Experience with Charlotte: Simplicity and Function in a Distributed Operating System, *IEEE Trans. on Software Engineering,* Vol: 15, No 6, 676-684, (1989).

[15] S. Soysal, B. Örencik, A Comparison of the BAG3 Transport Protocol with Other Light-Weight Protocols, *1st Symp. on Computer Networks (BAS'96)*, İstanbul, 26-33, (1996).

[16] S. Soysal, B. Örencik, D. Ünal, Implementation of BAG3 Light-Weight Real Time Transport Protocol, *2nd Symp. on Computer Networks (BAS'97)*, Ankara, 58-65, (1997).

[17] ANSI, *Fiber Distributed Data Interface (FDDI) - Token Ring Media Access Control (MAC)*, American National Standard for Information Systems, ANSI X3.139, (1987).

[18] A. Shah, G. Ramakrishnan, *FDDI: A High Speed Network,* PTR Prentice Hall, New Jersey, (1994).

[19] F. E. Ross, An Overview of FDDI: The Fiber Distributed Data Interface, *IEEE Journal on Selected Areas in Communications,* Vol: 7, No. 7, 1043-1057, (1989).

[20] D. Palumbo, The Derivation and Experimental Verification of Clock Synchronization Theory, *IEEE Trans. on Computers,* Vol: 43, No 6, 676-686, (1994).

[21] K. Arvind, Probabilistic Clock Synchronization in Distributed Systems, *IEEE Trans. on Parallel and Distributed Systems,* Vol: 5, No 5, 474-487, (1994).

[22] C. Fidge, Logical Time in Distributed Computing Systems, *IEEE Computer,* Vol: 24, No 8, 28-33, (1991).

[23] N. Suri, M. Hugue, C. Walter, Synchronization Issues in Real-Time Systems, *Proc. of the IEEE,* Vol: 82, No 1, 41-54, (1994).

[24] P. Ramanathan, K. Shin, R. W. Butler, Fault-Tolerant Clock Synchronization in Distributed Systems, *IEEE Computer,* Vol: 23, No 10, 33-42, (1990).

[25] A. Olson, K. Shin, Probabilistic Clock Synchronization in Large Distributed Systems, *IEEE Trans. on Computers,* Vol: 43, No 9, 1106-1112, (1994).

[26] E. Harmancı, A Real Time Distributed Operating System Prototype For Multiprocessor Architectures, *Tech. Report of TÜBİTAK - EEEAG-BAĞ2,* (1997) (in Turkish).

[27] National Sem., *Fiber Distributed Data Interface (FDDI) Databook,* National Semiconductor Corp., (1991).

[28] National Sem., *Desktop FDDI Handbook,* National Semiconductor Corp., (1992).

[29] R. Macomber, *A Software Engineers's Guide to the System Interface of the DP83266 MACSI Device,* National Semiconductor, Application Note 964, (1994).

[30] N. Malcolm, W. Zhao W, The Timed-Token Protocol fo Real-Time Communications, *IEEE Computer Mag.,* Vol: 27, No 1. 35-41, (1994).

[31] K. G. Shin, Q. Zheng, FDDI-M: A Scheme to Double FDDI's Ability of Supporting Synchronous Traffic, *IEEE Trans. on Parallel and Distributed Systems,* Vol: 6, No 11, 1125-1131, (1995).

[32] S. Zhang, A. Burns, An Optimal Synchronous Bandwidth Allocation Scheme for Guaranteeing Synchronous Message Deadlines with the Timed Token MAC Protocol, *IEEE/ACM Trans. on Networking,* Vol:3 No 6, 729-741, (1995).

[33] Q. Zheng, K. Shin, Synchronous Bandwidth Allocation in FDDI Networks, *IEEE Trans. on Parallel and Distributed Systems,* Vol: 6, No 12, 1332-1338, (1995).

[34] M. Hamdaoui, P. Ramanathan, Selection of Timed Token Protocol Parameters to Guarantee Message Deadlines, *IEEE/ACM Transaction on Networking,* 340-351, (1995) .

[35] G. Agrawal, B. Chen, W. Zhao, S. Davari, Guaranteeing Synchronous Message Deadlines with the Timed Token Medium Access Control Protocol, *IEEE Trans. on Computers,* Vol: 43, No 3, 327-339, (1994).

[36] F. Feng, A. Kumar, W. Zhao, *Investigating Synchronous Bandwidth Allocation in an FDDI Network for Real-Time Communications,* Tech. Report 95-022, Department of Computer Science, Texas A&M University, (1995).

[37] F. Buzluca, Design of an Efficient Real Time Communication Structure for an FDDI Based Network System, Doctoral Dissertation in İstanbul Technical University, İstanbul, (1997) (in Turkish).

[38] F. Buzluca, E. Harmancı, An Efficient and Dynamic Synchronous Bandwidth Allocation Scheme for Guaranteeing Synchronous Messages with Arbitrary Deadlines in an FDDI Network, *12th International Symp. On Computer and Information Sciences (ICSIS XII),* Antalya, 301-308, (1997).

[39] F. Buzluca, E. Harmancı, An Efficient and Dynamic Synchronous Bandwidth Allocation Scheme for FDDI Networks, *2nd Symp. on Computer Networks (BAS'97),* Ankara, 50-57, (1997) (in Turkish).

[40] T. C. Kwok, Residential Broadband Internet Services and Applications Requirements, *IEEE Communications Magazine,* June 1997, 76-83 (1997).

[41] K. C. Sevcik, M. J. Johnson, Cycle Time Properties Of The FDDI Token Ring Protocol, *IEEE Transactions on Software Engineering,* Vol: 13, No 3, 376-385, (1987).

[42] B. Örencik, F. Buzluca, O. Akgün, O. Aliefendioğlu, C. Özden, Design of a VME PC Based Communication Unit for Real-Time Interprocess Communication, *8th Symp. on Computer and Microprocessor Applications* (μP'94), Budapest-Hungary, 369-376, (1994).

[43] G. Lundy, H. Tipici, Specification and Analysis of the SNR High-Speed Transport Protocol, *IEEE/ACM Trans. on Networking,* Vol: 2, No 5, 483-496, (1994).

[44] G. Chesson, The Xpress Transfer Protocol (XTP)-A Tutorial, *Computer Communications Review,* Vol: 20, No 5, 67-80, (1990).

[45] Doeringer W. A., et al., A Survey of Light-Weight Transport Protocols for High-Speed Networks, *IEEE Trans. on Communications,* Vol: 38, No 11, (1990).

[46] M. Simms, Using State Machines as a Design and Coding Tool, *Hewlett Packard Journal,* Vol: 45, No 6, 27-32, (1994).

[47] A. S. Tanenbaum, *Computer Networks.* 3rd ed., New Jersey, Prentice Hall, (1996).

[48] B. Örencik, C. Özden, Design of Interprocess Communication Primitives and Structures for Real Time Systems, *8th Symp. on Computer and Microprocessor Applications* (μP'94), Budapest-Hungary, 362-368, (1994).

[49] S. Soysal, B. Örencik, A. Yavaş, Implementation of BAG3 Link Based Session Protocol, *2nd Symp. on Computer Networks (BAS'97),* Ankara, 66-73, (1997).

[50] B. Örencik, S. Soysal, The Design of BAG3 Real Time Light-Weight Transport and Session Protocols, *12th International Symp. On Computer and Information Sciences (ICSIS XII),* Antalya, 333-340, (1997).

[51] S. Neeraj, M. H. Michelle, C. J. Walter, Synchronization Issues in Real-Time Systems, *Proceedings of the IEEE,* Vol: 82, No 1, 41 -54, (1994).

[52] L. Lamport, P. M. Melliar-Smith, Synchronizing Clocks in the Presence of Faults, *Journal of the ACM,* Vol: 32, No 1, 52-78, (1985).

[53] J. Lundelius, N. Lynch, An Upper and Lower Bound for Clock Synchronization, *Information and Control,* Vol: 62, 190-204, (1984).

[54] F. Christian, Probabilistic Clock Synchronization, *Distributed Computing,* No: 3, 140-158, (1989).

[55] R. Cohen, Y. Ofek, A. Segall, A New Label-Based Source Routing for Multi-Ring Networks, *IEEE/ACM Trans. on Networking,* Vol: 3, No 3, 320-328, (1995).

[56] T. Pei, C. Zukowski, Putting Routing Tables in Silicon, *IEEE Network Magazine,* Jan., 42-50, (1992).

CONTENTS