論 文

A Remote Data Access Interface Model Using the Monitor IPC Mechanism

正會員 金 東 圭*

Dong Kyoo KIM* *Regular Member*

ABSTRACT In this paper, we construct a remote data access interface model using the IPC mechanism based on the monitor concept. A general purpose IPC mechanism which can be used to implement arbitrary forms of communications control over distributed processes in a network environment is developed as a basis to build the framework for the model. The interface model and synchronization mechanisms are presented for providing application processes with remote data access capability using the above IPC mechanism.

要 約 본 논문에서는 모니터 개념에 입각한 IPC 메카니즘을 사용하는 원격 데이타 액세스 인터페이스 모델을 구축하였다. 네트워크 환경에서 작동하는 분산 프로세스들에 대하여 유연성 있는 통신 제어를 구현하기 위한 일반성 있는 IPC 메카니즘이 모델의 프레임워크 확립을 위한 기반으로 개발되었다.

위의 IPC 메카니즘을 사용하여 응용 프로세스들에게 원격 데이타 액세스 기능을 부여할 수 있도록 인터페이스 모델과 동기 메카니즘이 기술되었다.

I. Introduction

The current data processing approach enjoying the spotlight of user attention is distributed systems and perhaps the most important and widely studied technical problem in distributed systems is the problem of synchroniz ation on distributed resources.

In this paper, we construct a simple and reliable remote data access interface model for synchronizing distributed concurrent processes on shared data. The model is built on the basis of P.B. Hansen's monitor concept[6].

As a basis, a reliable and general purpose IPC(Interprocess Communication) mechanism for communications control over distributed processes are suggested. It is a result of the research concerning distributed processes communication control mechanism[3], simulated by means of semaphore.

Because, even with a reliable network service, there is a need for synchronization to control remote data, access requests from distributed concurrent processes on shared data, we implement the model over the above IPC

*亞洲大學校 電子計算學科
Department of Computer Science, Ajou University)
論文番號 : 91-35(接受1991. 3. 15)

control. In designing the model, we considered two parts, one is for interfacing interaction between application processes and the IPC control, the other is for controlling operations of a data processing system from remote data access requests through the IPC control.

Finally, we give an overview of the structure and describe synchronization mechanisms over that structure.

Ⅱ. IPC structure and mechanism for distributed processes

In this section, a general purpose structure and a mechanism which can be used to implement arbitrary forms of communications control over distributed processes in a network environment are suggested and developed.

2.1. Design considerations

The network has the following properties and environments :

• Logical Token Ring

It is assumed that the network consists of a fixed number of processor nodes connected cyclically by unidirectional bus links. That is, processes and data are distributed among the nodes in a logical token ring.

• Virtual Channels

The processes distributed among the nodes are connected by virtual channels. Each channel can transmit one data item at a time from a single sender process to a single receiver process.

• Token As Access Right

A message originating from a node can only be proceeded by that node when it is in possession of a token circulating the network.

• FIFO Scheduling Policy

Only one message at a time originating from a node that holds the token can be transmitted. Also, that message should follow all the messages that are already buffered for transmission(FIFO basis).

• Rendezvous Concept

A request for transmission and the corresponding response must be matched before transmission occurs.

• Tightly Coupled Synchronization Policy

It is guaranteed that there is no message discarding due to buffer overflow and no deadlock.

• Message Types

In order to reflect the user-level primitive operations, a message is one of four types :

A-REQUEST ; A-RESPONSE ; A-FORCED ; TOKEN ;

2.2. User-level IPC Primitive Operations for Distributed Processes

The following four types of primitive operations are defined for providing users with communication services.

SEND(channel, item) ; RECEIVE(channel VAR item) ;

When two processes communicate on a virtual channel, the receiving node transmits a request message by calling RECEIVE primitive to the sending node which then responds by transmitting a response message by calling SEND primitive on the network.

RECEIVE request and SEND response must make a rendezvous at the node issuing SEND primitive. Fig. 1 concerns about the request-response rendezvous.

FORCED-SEND(channel, item) ; RECEIVE-ANY(VAR channel, VAR item) ;

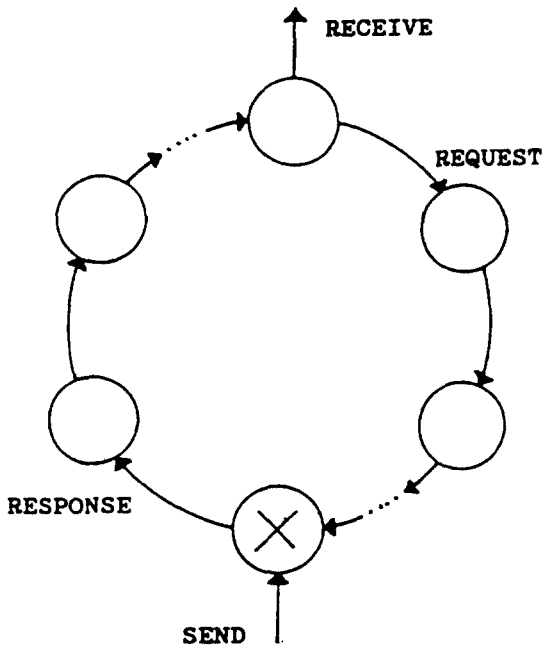
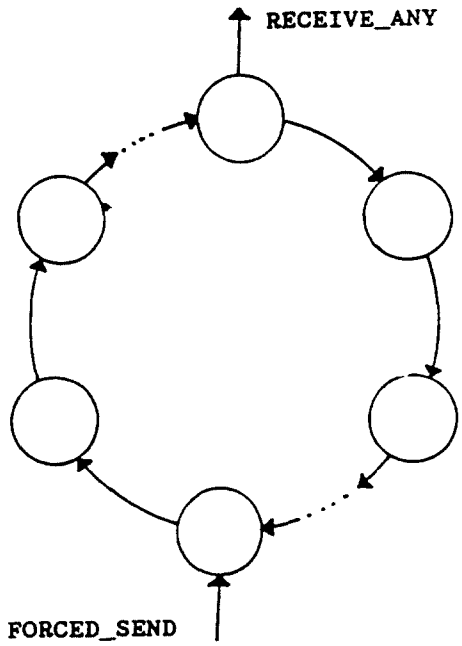


Fig. 1. Transmission via rendezvous of request and response



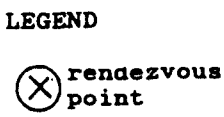


Fig. 2. Direct transmission via RECEIVE-ANY and FORCED SEND

In order to bypass the request response rendezvous and speed up the communication, any processes can send data via FORCED SEND primitive operation to a destination process without waiting for a request.

The process at the destination delays after sending RECEIVE-ANY message to receive any matching FORCED SEND message. Therefore, the concept of rendezvous still is alive although this is different from the request-response rendezvous. Fig. 2 concerns about this process.

2.3. IPC structure

A network node consists of the following system components : READER process ; WRITER process ; TOKEN-ACCESS monitor ; INP monitor of type INPUTS ; OUT monitor of type OUTPUTS ; BUF monitor of type BUFFER ; BUS monitor of type BUS-LINK ; Application processes.

• INP(i) monitor

An INPUTS monitor implements four operations :

RECEIVE sends a request and delays a calling process until a response arrives on a

given channel ; RESPONSE delivers a data item on a channel and continues the process that is waiting for receiving it ; A process can select another mechanism to bypass the above request-response rendezvous process. RECEIVE-ANY checks to see if the ready list associated with its receiving parties is set true or not, proceeds immediately to pick up messages or delays until being awoke respectively ; FORCE delivers a data item on a channel and continues a process that has issued RECEIVE-ANY.

• OUT(i) monitor

An OUTPUTS monitor implements three operations :

SEND delays a calling process until a given channel is ready for transmission. It then sends a data item through a local buffer ; REQUEST makes a channel ready to send and continues a process waiting for sending on that channel ; To bypass the above process, FORCED-SEND sends a data item immediately without waiting for request on a channel.

• BUF(i) monitor

Transmitted messages are directly forwarded onto this BUFFER monitor as long as the buffer is not full. Messages stored in the BUFFER monitor are fetched by the WRITER process as long as the buffer is not empty whenever it wants to.

• BUS(i) monitor

The monitor simply provides a place for a communication between the WRITER process and the READER process. It is only for simulation purpose, and not necessary in real implementation.

• TOKEN-ACCESS monitor

The process that issued RECEIVE, SEND or FORCED SEND to the INPUTS monitor or OUTPUTS monitor is delayed in TOKEN-ACCESS queue to possess a token according to the FIFO policy. When a token comes into

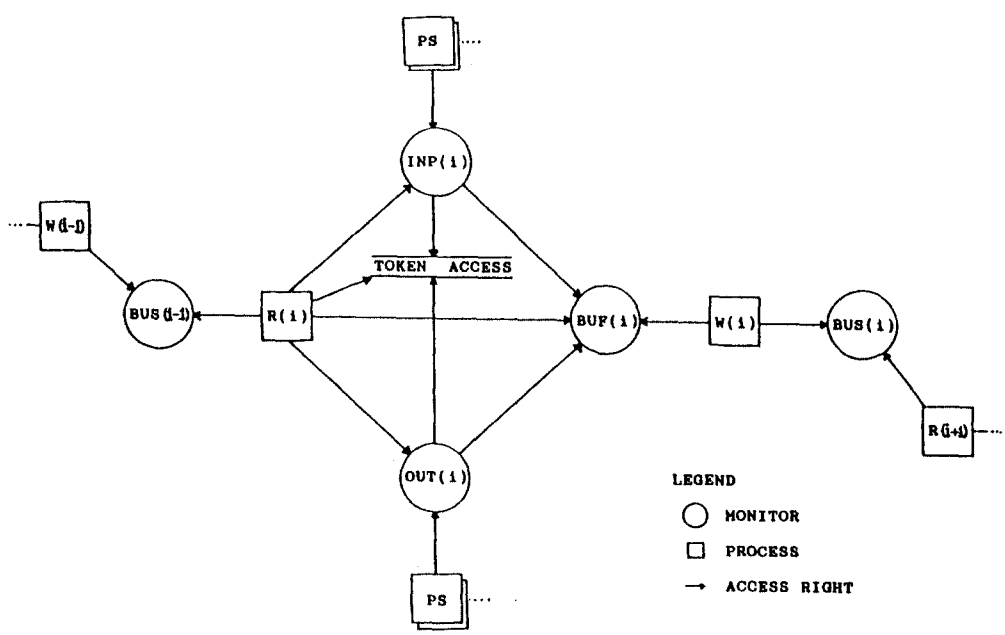


Fig. 3. IPC structure in a node.

the READER process, the process delayed in the head of TOKEN-ACCESS queue is aworke and the message is sent to the BUFFER monitor.

• R(i) process

A READER process receives one message at a time from the previous node through a bus link. The READER uses two constants defining the set of input channels and the set of output channels used by its node. If the node is the destination of a message, the READER process performs a RESPONSE, REQUEST, or FORCE operation on it. If the type of the message is token, it wakes a process waiting in TOKEN ACCESS queue and then passes the message immediately to the BUFFER monitor. Otherwise, it sends the message through a local buffer to the next node. These operations are implemented by INPUTS, OUTPUTS, BUFFER, and TOKE N-ACCESS monitors.

• W(i) process

A WRITER process receives one message at a time from a local buffer and passes it to the next node through a bus link.

2.4. IPC mechanism

The IPC mechanism is performed via two major schemes :

Transmission via rendezvous of request and response ; Direct transmission via RECEIVE ANY and FORCED-SEND.

• Rendezvous of request and response

An application process calls a local INPUTS monitor when it wishes to receive data on a channel. A request is now sent through the network to the opposite side of the channel and the process is delayed until the correspo-nding response comes back.

An application process calls a local OUTP UTS monitor when it wishes to send data on a channel. The process is delayed until a request for data arrives on that channel. A response is then sent through the network to the opposite side of the channel.

So, a transmission on the network only takes place after rendezvous and causes a message to pass through all the nodes once.

• Direct transmission via RECEIVE-ANY and FORCED SEND

An application process calls RECEIVE ANY in a local INPUTS monitor when it wishes to receive data and bypass the request response rendezvous.

An application process calls FORCED-SEND in a local OUTPUTS monitor when it wishes to send data without waiting for request.

So, RECEIVE ANY-FORCED-SEND process provides application processes with direct tra nsmission without rendezvous.

Ⅲ. Remote Data Access Interface Model

In this section, a generalized remote data access interface structure and mechanisms using the above IPC mechanism is constructed.

To achive intended operations using the IPC user level primitive operations, an interface between application processes and the IPC primitive operations is specified.

To control the interaction between remote data access requests from the IPC control and a local data processing system, another inter-face is described. Finally, we give an overview of the structure and synchronization mechan-isms.

3.1. Properties and Preliminaries

Before we consider the remote data access

interface model, we summarize some properties and preliminaries of the model.

• Data Exchange by a Key

Suppose that all the data are accessed by a key. Thus, a process which wishes to access remote data has to call remote data access operations with a key.

• Data Transmission in a Record Unit

It is assumed that all the data are transmitted in the form of a record. To support this assumption, the following abstract operations are provided and the names of the operations imply what they have to do, but these are not implemented in detail.

CONVERT-BIT-STREAM-TO-RECORD (bit-stream, record) ;

CONVERT-RECORD-TO-BIT-STREAM(record, bit-stream) ;

• Uniform Data Transmission Time

All the data travel the same distance through all the nodes. The only exception is a data access in the same node. In this case, the data access request and the corresponding retrieved data pass through all the nodes once.

• Encapsulation and Decapsulation

According to the layered architecture and protocols, remote data access requests are encapsulated when these pass from a process to the IPC control and decapsulated when these proceed from the IPC control to a local data processing system. To support these procedures, the following two primitive operations are provided but these are not implemented in detail.

CONVERT-BIT-STREAM-TO-QUERY (bit-stream, query) ;

CONVERT-QUERY-TO-BIT-STREAM (query, bit-stream) ;

• Connection-oriented communication

The communication between application processes and a remote local data processing system begins with a connection establishment request and ends with a connection termination request.

3.2. Interface between Application Processes and the IPC Control

An ACCESS-CALL monitor implements the following primitive operations for providing application processes with an interface necessary for remote data access using the IPC mechanism.

• GEN-REQ(channel) ;

A process wishing to access remote data must first send the request of type GEN-CALL by calling GEN-REQ operation in the ACCESS-CALL monitor. It is delivered, without waiting, by FORCED-SEND in the OUTPUTS monitor of the IPC control. It makes a channel ready for data access and notifies the data processing system on the opposite side of the channel to carry out appropriate data initialization tasks.

• CLOSE-REQ(channel) ;

When a process accessing remote data wants to terminate its data access on a channel, it sends a request of type CLOSE-CALL by calling CLOSE-REQ operation in the ACCESS-CALL monitor, In this case, the channel is released and the data processing system on the opposite side of the channel carries out appropriate data closing operations.

• RETRIEVE(channel, key, VAR buffer) ;

After issuing GEN-REQ, the process can retrieve data by sending a data retrieve request of type RETRIEVE-CALL. It is performed by calling RETRIEVE operation in the ACCESS-CALL monitor.

The operation is implemented by two IPC primitives, SEND and RECEIVE. That is, the

data retrieve request is delivered by SEND in the OUTPUTS monitor of the IPC control, and the corresponding retrieved data is returned via RECEIVE in the INPUTS monitor of the IPC control.

The data retrieve operation begins with a GEN-REQ and ends with a CLOSE-REQ.

The request of type GEN-CALL is delivered without waiting. On the other hand, the request of type CLOSE-CALL and RETRIEVE-CALL is delivered under the RECEIVE request-SEND response rendezvous of the IPC control. It will provide distributed concurrent processes with remote data access in a synchronized manner.

3.3. Interface between the IPC Control and a Local Data Processing System

Primary aims of the interface are as follows

- To control interaction between the IPC control and a local data processing system, for example DBMS.
- To hide low level details concerning a local data processing system

In order to meet the above aims, the interface is implemented in two levels. In other words, the interface consists of the ACCESS-CONTROL monitor and the DATA SYSTEM process.

The ACCESS-CONTROL monitor defines the following primitive operations to control interaction between remote data access requests from the IPC control and operations of a local data processing system.

- ACCEPT INIT REQUEST(VAR channel, VAR query) :

This primitive is defined for accepting any one of the requests of type GEN-CALL from channels connected through the IPC control to a node. Because the request of type GEN-CALL from a channel is sent by issuing FORCED SEND in the OUTPUTS monitor of the IPC control, the operation is implemented by issuing RECEIVE-ANY in the INPUTS monitor of the IPC control.

As a result of accepting that type of request, connection establishment and data initialization tasks will occur.

- ACCEPT ACCESS-REQ(channel, VAR query) :

After establishing a connection by GEN-CALL from a process, the process may send the request of type RETRIEVE-CALL. In order to accept the request, we define another operation, ACCEPT-ACCESS-REQ. It is implemented by RECEIVE in the INPUTS monitor of the IPC control. Thus, it will make a request-response rendezvous at the node issuing RETRIEVE or CLOSE-REQ operation. Even if it is possible to accept CLOSE-CALL as well as RETRIEVE-CALL by issuing the operation, there is a difference in the sense that RETRIEVE-CALL must be returned with a retrieved data but CLOSE-CALL doesn't need to do so.

The course of the data exchange ends as a result of accepting CLOSE-CALL.

- RETURN(channel, record) :

The data record which is the result of RETRIEVE-CALL must be returned. To carry out this operation, we define another operation, RETURN. The process which has issued RETRIEVE operation is delayed to accept the retrieved data and it is implemented by RECEIVE in the INPUTS monitor of the IPC control. So, the retrieved data is delivered by SEND in the OUTPUTS monitor of the IPC control when we implement RETURN operation. That is, another rendezvous will take place in the node in which the local data

processing system is located.

The DATA SYSTEM process is a continuously operating process which is provided to hide details of the operations of a local data processing system and to solve a concurrency problem from the distributed concurrent processes.

To conduct the above two tasks, the process references both the ACCESS CONTROL monitor and the operations of a local data processing system.

Because we are only interested in the remote data access interface, we describe operations of a local data processing system only in the abstract level.

To do so, the DB SYSTEM monitor is defined for specifying the operations in a natural language forms. There are three types of abstract operations in the DB SYSTEM monitor, INIT DB, ACCESS DB, and CLOSE DB. According to the type of the requests from processes, one of them is called to perform appropriate operation in a local data processing system.

The DATA SYSTEM process listens to the channels which are connected to this node via ACCEPT-INIT REQ operation in the ACCESS-CONTROL monitor until a request of type GEN-CALL arrives. On arrival, the process calls a local data processing system via INIT-DB in the DB-SYSTEM monitor to perform initialization tasks. After conducting initialization, the process accepts data retrieve operations on that channel previously established until CLOSE-CALL is accepted.

Since the process only listens to the channel delivered GEN-CALL, the concept of mutual exclusion is guaranteed among distributed concurrent porcesses.

3.4. Remote Data Access Structure and Mechanisms

As already mentioned, we implement two interfaces for providing distributed concurrent processes with remote data access. These interfaces are combined with the IPC structure to construct a system which can be used to perform remote data access in a network environment. The overall structure of the system is shown in the Fig. 4 and the appendix contains the complete text of the program written in concurrent pascal.

The remote data access mechanisms over the structure shown in the Fig. 4 can be thought of with the following three phases ; connection establishment, data exchange, and connection termination.

- Connection Establishment and Termination

Even with a reliable network service, there is a need for connection establishment and termination procedures to support connection oriented service. Connection establishment serves two main purposes :

- It allows the DATA SYSTEM process to assure that an application process is connected through a channel.
- It allows the DATA SYSTEM process to perform initialization tasks before a process asks for data access.

Connection establishment can be accomplished by a simple set of commands. To begin with, an application process wishing remote data access from a local data processing system sends GEN CALL by calling GEN-REQ operation in the ACCESS-CALL monitor. Then, it is delivered through a channel by FORCED-SEND in the IPC control. In all the nodes in which a local data processing system is located, the DATA SYSTEM process is listening to channels connected to its node by

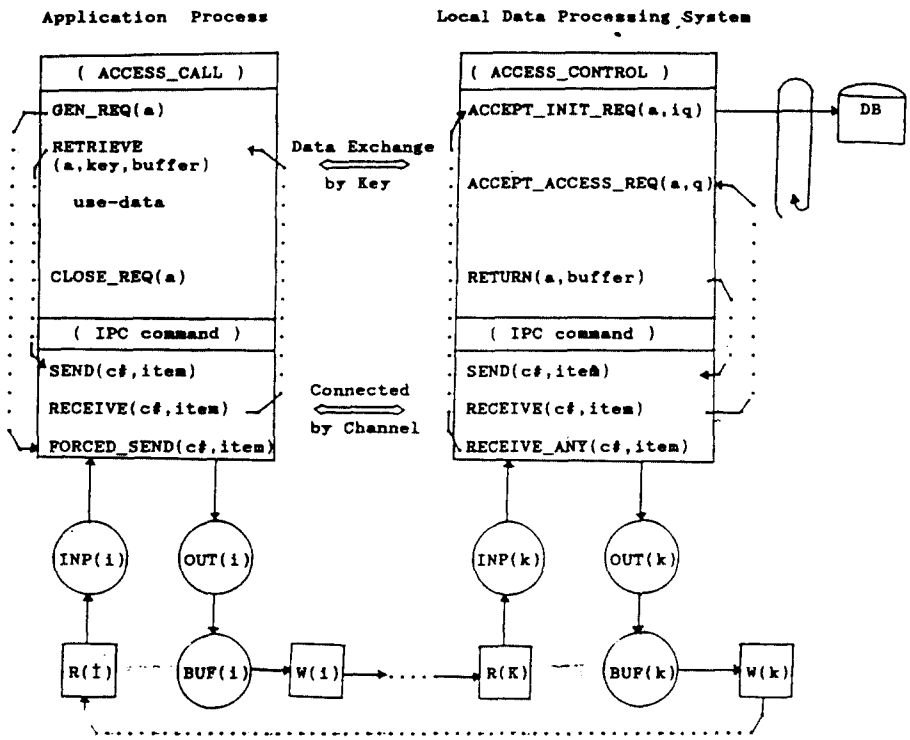


Fig. 4. Remote data access interface model over the IPC control

calling ACCEPT-INIT REQ in the ACCESS-CONTROL monitor. ACCEPT INIT REQ is implemented by RECEIVE ANY in the IPC control to accept GEN-CALL delivered by FORCED-SEND. On accepting GEN CALL, the process performs an initialization of a local data processing system by issuing DB-INIT in the DB SYSTEM monitor.

The process accessing the data may initiate a close request when it wishes to terminate its data access on that channel. The request is implemeted by calling CLOSE REQ operation in the ACCESS-CALL monitor and being delivered via SEND in the IPC control. After establishing a connection, the DATA SYSTEM process on the opposite side may accept a data retrieve request or a close request through the IPC control. Accepting a close request of type CLOSE CALL from a process connected, The DATA SYSTEM process performs the following two tasks :

- Close its own data
- Release the connection so that another process wishing remote data access on that channel can be initiated.
- Data Exchange

Fig. 5 shows a typical sequence of events in a data exchange phase. As already mentioned, the data exchange phase is entered after connection establishment.

The sequence of events is as follows :

1. The DATA SYSTEM process accepts a GEN CALL from a channel by issuing ACCEPT INIT REQ operation in the ACCESS CONTROL monitor.
2. After initialization tasks, The DATA SYS

TEM process issues ACCEPT-ACCESS-REQ operation in the ACCESS-CONTROL monitor to accept data access requests from the channel. It is implemented by RECEIVE in the INPUTS monitor.

3. The RECEIVE request causes a rendezvous in the node in which the process sending GEN-CALL is waiting for permission of data access. As a result of the rendezvous, the process can transmit the request of type RETRIEVE CALL to the DATA SYSTEM process. It is delivered via SEND in the OUTPUTS monitor of the IPC control. Then the process is delayed by issuing RECEIVE in the INPUTS monitor of the IPC control until the retrieved data comes back.
4. On arriving of the request of type RETR IEVE-CALL, the DATA SYSTEM process accepts the request and processes it accor ding to its type.
5. After processing the request, the DATA-SYSTEM process calls RETURN operation in the ACCESS-CONTROL monitor to send the retrieved data to the process currently connected. To carry out this, there will be another rendezvous in this node.
6. The process waiting for the retrieved data is aworke when the data returns.
7. The sequence from 1 through 6 will be continued until the process currently conn ected issues CLOSE-REQ in the ACCES-S-CALL monitor.

   After connection termination, the DATA-SYSTEM process gives the access right of its data to a process waiting for access(if there are any).

Ⅳ. Conclusion and Discussion

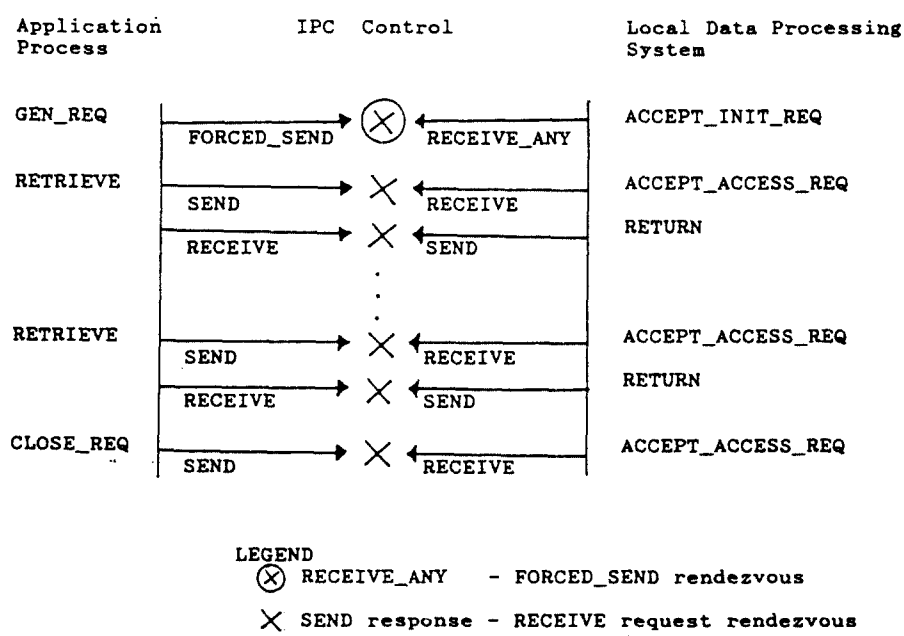


Fig. 5. Sequence of events in a data exchange phase

This far, we have shown how to realize an interface model simply and reliably with an abstract language for synchronizing data access requests among processes distributed in a network environment.

Since this model was constructed in accor dance with layered architecture, it can be easily extended to be used for specific appli cation. That is, in order to tansport data or queries in string format rather than in record units, a preprocessor may be needed : an additional component may be required in order to provide more efficiency for traveling the network distinguishing whether the requested data are located in the same node as the requesting process or in different nodes.

Another issue is delivery of data access requests via request response rendezvous and subsequent delivery of retrieved data. The rendezvous solution guarantees the mutually exclusive access of shared data among distr ibuted processes, but is entails delay problem.

Acknowledgement : This paper is an outcome of a research project supported by a grant from the 1988 research fund of Korea Research Foundation. Its main content was published in the Proc. of 4th. International Workshop on Computer Communications held in Japan in 1989.

BIBLIOGRAPHY

1. C.A.R. Hoare, "Monitors : An operating system structuring concept", Comm. ACM, Vol. 17, pp. 549~557, Oct. 1974.
2. Dong Kyoo Kim, Computer communication networks, Sang jo Publications Company, Inc., 1986.
3. Dong kyoo Kim, "A distributed processes communication control mechanism based on monitor concepts implemented through simulation using semaphore", 2nd International joint Workshop on Computer Communications, Tsukuba, Japan, June 18~20, 1987.
4. Gorge A. Champine, Distributed computer systems impact on management, design, and analysis, North Holland, 1980.
5. Morris Sloman, Jeff Kramer, Distributed systems and computer networks, Prentice-hall, 1987.
6. P.B. Hansen, The architecture of concurrent programs, Prentice Hall, 1973.
7. Peter Wegner, Scott A. Smolka, "Processes, tasks, and monitors : A comparative study of concurrent programming primitives", IEEE Trans. Software Eng., vol. SE P, No. 4, July 1983.
8. V.E. Walletine, W.J. Hankly, SIMMON A concurrent pascal based Simulation System, Kansas State University, 1978.

APPENDIX

```
(*System Variables Initialization
CONST
    nmax        =…;
    cmax        =…;
TYPE
    item        =…;
    node        =1..nmax;
    channel     =1..cmax;
    channelset  =SET of channel;
    kinds       =(a-request, a-response, a-forced, token);
    message     =RECORD
                     kind     : kinds;
                     link     : channel;
                     contents : item
                 END;
TYPE
    primarykey=…;
    recordtype=…;
    querytype   =…;
    query       =RECORD
                     type : querytype;
                     key  : primarykey
                 END;
(********** IPC control implementation **********)
(*  The IPC control structure has following components.          *)
(*  An INPUTS monitor implements four operations ; RECEIVE,     *)
(*  RESPONSE, RECEIVE-ANY, and FORCE.                           *)

TYPE INPUTS=
MONITOR(buf : buffer ; token : token-access);
VAR
    this, msg : message ;
    receiver  : ARRAY[channel] OF QUEUE ;
    sender, receiver-any : QUEUE ;
(*  RECEIVE sends a request through a token access and del ays a *)
(*  calling process until a response arrives on a given channel. *)

PROCEDURE ENTRY receive(c : channel ; VAR v : item);
```

```
BEGIN
    WITH this DO
    BEGIN
        kind : =a-request ;
        link : =c
    END;
    token.wait;
    buf.send(this);
    delay(receiver[c]);
    v : = this.contents
END;

(*  RESPONSE delivers a data on a channel and continues the          *)
(*  process that is waiting to receive it.                           *)

PROCEDURE ENTRY response(m : message) ;
BEGIN
    this : =m;
    continue(receiver [m.link])
END ;

(*  RECEIVE-ANY receives or waits any FORCED SEND message without    *)
(*  request-response rendezvous at sending node.                     *)

PROCEDURE ENTRY receive_any(VAR c : channel ; VAR v : item);
BEGIN
    IF msg.empty THEN delay(receiver_any);
    WITH msg DO
    BEGIN
        v : =contents;
        c : =link;
        continue(sender)
    END
END;

(*  FORCE delivers a forced data on a channel and continues the      *)
(*  process that is waiting to receive any FORCED SEND messgae.      *)

PROCEDURE ENTRY force(m : message) :
BEGIN
```

```
    IF  msg.full  THEN delay(sender);
    msg : =m ;
    continue(receiver-any)
END;
BENGIN msg.reset  END ; (*  INPUT monitor *)

(*  An OUTPUTS monitor implements implements three operations ; SEND, REQUEST, *)
(*  and FORCED-SEND.                                                            *)
TYPE OUTPUTS=
MONITOR(buf : buffer ; token : token-access);
VAR
    c    : channel;
    this : message;
    list : ARRAY[channel] OF RECORD
                                ready : BOOLEAN;
                                sender : QUEUE
                             END;
(*  SEND delays a calling process until a given channel is ready                *)
(*  for transmission. It then sends a data item through a local                 *)
(*  buffer.                                                                     *)

PROCEDURE ENTRY send(c : channele ; v : item);
BEGIN
    WITH list[c] DO
        IF NOT ready THEN delay(sender);
    WITH this DO
    BEGIN
        kind : = a_response;
        link : =c;
        contents : =v
    END;
    token.wait;
    buf.send(this);
    list[c].ready : =false
END;

(*  REQUEST makes a channel ready to send and continues a process               *)
(*  (if there are any) waiting to send on that channel. At first,               *)
(*  no channels are ready.                                                      *)
```

```
PROCEDURE ENTRY request(m : message) ;
BEGIN
    WITH list [m.link] DO
    BEGIN
        ready : =true ;
        continue(sender)
    END
END;

(*  FORCED-SEND sends directly message to the process that ready        *)
(*  to receive any message in RECEIVE-ANY entry.                        *)

PROCEDURE ENTRY forced-send(c : channel ; v : item);
BEGIN
    WITH this DO
    BEGIN
        kind : = a_forced :
        link : =c;
        contents : =v
    END;
    token.wait;
    buf.send(this)
END;
BEGIN
    FOR c : =1 To cmax DO list[c].ready  : =FALSE
END ; (*  OUTPUT monitor  *)

(*  BUFFER monitor implements two operations ; SEND and RECEIVE.        *)

TYPE BUFFER=
MONITOR
CONST bmax=…"cmax / nmax+1";
VAR
    buf : SEQUENCE[bmax] OF message ;
    sender, receiver : QUEUE ;

(*  SEND delays a calling processes as long as the buffer is full.      *)
(*  It then puts a message into the buffer and continues the            *)
(*  execution of another processes(if there are any) waiting for        *)
(*  receiving the message.                                              *)
```

```
PROCEDURE ENTRY send(m : message) ;
BEGIN
    IF buf.full  THEN delay(sender) ;
    buf.put (m);
    continue(receiver)
END;

(*  RECEIVE delays a calling processes as long as the buffer is      *)
(*  empty. It then gets a message from the buffer and continues the  *)
(*  execution of another processes(if there are any) waiting for     *)
(*  sending a message.                                               *)

PRCEDURE ENTRY receive(VAR m : message);
BEGIN
    IF buf.empty  THEN delay(receiver) ;
    buf.get(m);
    continue(sender)
END;
BEGIN buf.reste  END ; (*  BUFFER monitor  *)

(*  BUS_LINK implements two operations : INPUT-FROM-BUS and          *)
(*  OUTPUT-ONTO-BUS.                                                 *)

TYPE BUS_LINK=
MONITOR
VAR link : message ;
      sender, receiver : QUEUE ;

(*  INPUT-FROM-BUS delays a READER process as long as bus link       *)
(*  is empty. It then gets a message from bus link and continues     *)
(*  the execution of the READER process.                             *)

PROCEDURE ENTRY input_from_bus(VAR m : message);
BEGIN
    IF link.empty  THEN delay(sender);
    link.get(m);
    continue(sender)
END;

(*  OUTPUT_ONTO_BUS  delays a WRITER process as long as bus link is  *)
```

```
(* full. It then puts a message onto bus link and continues *)
(* the execution of the WRITER process. *)

PROCEDURE ENTRY output-onto-bus(m : message) ;
BEGIN
    IF link .full THEN delay(sender);
    link .put (m);
    continue(receiver)
END;
BEGIN link .reset END ; (* BUS-LINK monitor *)

(* TOKEN-ACCESS monitor implement two operations ; WAIT and WAKE *)
TYPE TOKEN_ACCESS=
MONITOR
VAR tokendelay : QUEUE ;

(* WAIT delays a calling process until a token arrives. *)

PROCEDURE ENTRY wait ;
BEGIN
    delay(tokendelay)
END;

(* WAKE wakes up a process(if there are any) waiting for a token. *)

PROCEDURE ENTRY wake;
BENGIN
    continue(tokedelay)
END;

BEGIN END ; (* TOKEN_ACCESS monitor *)

(* WRITER process receives one message at a time from a local *)
(* buffer and passes it to the next node through bus link. *)

TYPE WRITERPROCESS=
PROCESS(buf : buffer ; linkbuf : bus-link);
VAR m : message ;
BEGIN
    CYCLE
```

```
        buf .receive(m);
        bus_link.output_outo_bus(m)
    END
END ; (*  WRITER process  *)

(*  A READER process fetches one message at a time from the          *)
(*  previous node through bus link. If the channel is the            *)
(*  destination of a message, the READER performs a RESPONSE,        *)
(*  REQUEST, TOKEN, or FORCE operation on it. Otherwise, it sends    *)
(*  the message through a local buffer to the next node.             *)

TYPE READERPROCESS=
PROCESS(inpset, outset : channelset ; inp : inputs;
          out : outputs ; buf : buffer ; linkbuf : bus_link;
          token : token_access);
VAR m : message ;
BEGIN
    CYCLE
        input_from_bus(m);
        WITH m DO
            IF(kind=a_response) and (likn in inpset) THEN
               inp .response(m)
            ELSE IF(kind=a_request  and (link in outset) THEN
                out .request(m)
            ELSE IF(kind=a_forced) and (link in inpset) THEN
                inp .force(m)
            ELSE IF(kind=token) THEN
                BEGIN
                    token .wake;
                    buf .send(m)
                END
            ELSE buf .send(m)
    END
END ; (*  READER process  *)

(**********  Remote Data Access Model Implementation  ********
****)
(*  The model has following components.                              *)
(*  An ACCESS_CALL monitor implements three operations for           *)
(*  providing application processes with an interface necessary      *)
```

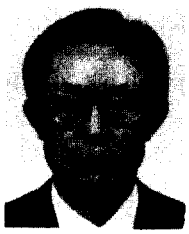
```
(*  for remote data access using the IPC mechanisms : GEN_REQ,        *)
(*  RETRIEVE, and CLOSE_REQ.                                          *)

TYPE ACCESS_CALL=
MONITOR(inp : inputs ; out : outputs);
VAR a_query : query ;

(*  GEN_REQ sends a request for remote access through a channel to    *)
(*  the appropriate destination node where a local data processing    *)
(*  system is located.                                                *)

PROCEDURE ENTRY gen_req(a : channel);
VAR v : item;
BEGIN
    WITH a_query DO
      type : ='GEN_CALL';
    convert_query_to_bit_stream(a_query, v);
    out .forced_send(a, v)
END;

(*  RETRIEVE sends a request for remote data retrieve and waits for   *)
(*  receiving returned data.                                          *)

PROCEDURE ENTRY retrieve(a : channel ; pk : primarykey;
                         VAR buffer : recordtype) ;
VAR v, str : item;
BEGIN
    WITH a_query DO
    BEGIN
        type : ='RETRIEVE CALL';
        key : =pk
    END;
    covert_query_to_bit_stream(a_query, v);
    out .send (a, v);
    inp. receive(a, str);
    convert_bit_stream_to_record(str, buffer)
END;

(*  CLOSE_REQ informas the DATA-SYSTEM connected that the process     *)
(*  accessing remote data wishes to terminate the data access.        *)
```

```
PROCEDURE ENTRY close_req(a : channel);
VAR v : item;
BEGIN
    WITH a.request DO
        type : ='CLOSE_CALL';
    convert_query_to_bit_stream(a.query, v);
    out.send(a, v)
END;

BEGIN END ; (* ACCESS_CALL monitor *)

(* ACCESS_CONTROL monitor implements three operations for *)
(* controlling the remote data access from the IPC *)
(* control : ACCEFT_INIT_REQ, ACCEPT_ACCESS_REQ, and RETURN. *)

TYPE ACCESS_CONTROL=
MONITOR(inp : inputs ; out : outputs);

(* ACCEPT_INIT_REQ, accepts a remote data access request from *)
(* channels connected to this node. *)

PROCEDURE ENTRY accept_init_req (VAR a : channel ; VAR q : query);
VAR v : item ;
BEGIN
    inp.receive_any(a, v);
    convert_bit_stream_to_query(v q)
END;

(* ACCEPT_ACCESS_REQ accepts a remote data retrieve request from *)
(* the process through the channel selected previously. *)

PROCEDURE ENTRY accept_access_req (a : channel ; VAR q : query);
VAR v : item ;
BEGIN
    inp.receive(a, v);
    convert_bit_stream_to_query(v, q)
END ;

(* RETURN delivers a retrieved data to the process through the *)
(* channel selected previously. *)
```

```
PROCEDURE ENTRY return(a : channel ; r : recordtype) ;
VAR
    str : item ;
BEGIN
    convert_recored_to_bit_stream(r, str);
    out.send(s, str)
END ;
BEGIN END ; (* ACCESS CONTROL monitor *)
(* A DB-SYSTEM monitor is defined for specifying the operations in *)
(* a local data processing system in the abstract. It is called *)
(* according to the type of data access request from a channel. *)

TYPE DB_SYSTEM=
MONITOR

PROCEDURE ENTRY init db;
BEGIN
    "Open and Initialize Data"
END ;

PROCEDURE ENTRY access db(q : query ; VAR buffer : recordtype);
BEGIN
    "Access data according to query and key
     and put the result onto buffer"
END ;

PROCEDURE ENTRY close_dp;
BEGIN
    "Close and Terminate using Data"
END ;

BEGIN END ; (* DB_SYSTEM monitor *)

(* An DATA_SYSTEM accepts remote data access requests, data *)
(* retrieve requests, and close requests from channels through *)
(* the ACCESS_CONTROL monitor and then according to the type of *)
(* request, performs appropriate operations through the DB-SYSTEM *)
(* monitor. It can synchronize the operations between remote *)
(* application process and a local data processing system *)
```

```
TYPE DATA-SYSTME=
PROCESS(control_data : access_control ; a_dbms : db_system);
VAR
    close : BOOLEAN ;
    a     : channel ;
    q, iq : query ;
BEGIN
    CYCLE
        control_data.accept_init_req(a, iq);
        IF (iq .type ='GEN_CALL' THEN
           BEGIN
               a_dbms.init_db;
               close : =false ;
               repeat
                  control_data.accept.access_req(a, q);
                  IF (q .type ='CLOSE_CALL') THEN close : =true
                  ELSE
                  BEGIN
                      a-dbms .access -db(q, buffer) ;
                      control_data.return(a, buffer)
                  END ;
           until close ;
           a_dbms.close_db
        END
    END
END ; (* DATA-SYSTEM process *)

(* An application process can access data on a channel. These      *)
(* operations are implemented in the ACCESS-CALL monitor described *)
(* before.                                                         *)

TYPE approcess=
PROCESS(inp : inputs ; out : outputs ; call_data : access_call);
VAR
    a      : channel ;
    key    : primarykey ;
    buffer : recordtype ;
BEGIN
    call_data.gen_req(a);
    call_data.retrieve(a, key, buffer) ;
```

```
        USE_DATA
    call_data.close_req(a)
END ; (* Application process *)

(* This initial process initializes all other processes and          *)
(* monitors and defines their access right to one another.           *)

VAR
    inp             : inputs ;
    out             : outputs ;
    buf             : buffer ;
    linkbuf         : bus_link;
    token           : token_access;

    reader          : readerprocess ;
    writer          : writerporcess ;

    call_data       : access_call;
    control_data    : access_control;
    dataps          : data_system;
    a_dbms          : db_system;

    process         : approcess ;
    inpset, outset  : channelset ;

    BEGIN
        inpset : =[…] ; outset : =[…] ;
    INIT
        inp(buf, token), out(buf, token), buf,
        linkbuf, token,

        reader(inpset, outset, inp, out, buf, linkbuf, token),
        writer(buf, linkbuf),
        call_data(inp, out),
        control_data(inp, out), a_dbms,
        dataps(control_data, a_dbms),
        process(inp, out, call_data),

END.(* MAIN *)
```



金 東 圭(Dong Kyoo KIM) 종신회원
1947年 2月 7日生
1972年 2月 : 서울대학교 공과대학 졸(공학사)
1979年 2月 : 서울대학교 자연과학대학원 졸(이학석사)
1984年 7月 : 미국 Kansas State University 대학원 졸(Ph.D. 정보통신 전공)
1972年 ~ 1976年 : 한국과학기술연구소 (KIST) 연구원
1976年 ~ 1979年 : 한국전자통신연구소 (KIET) 선임연구원
1981年 ~ 1982年 : 미국 Kansas State University 전자계산학과 Instructor
1979年 ~ 현재 : 아주대학교 전자계산학과 교수
연구관심분야 : 데이타 통신 / 컴퓨터 네트워크, 정보통신 Protocal engineering, Security, CIM 분산처리