論 文

A Distributed Processes Communication Control Mechanism Based on Monitor Concepts Implemented Through Simulation Using Semaphore

正會員 金 東 圭*

Dong Kyoo KIM* *Regular Member*

ABSTRACT A general purpose distributed processes communication control mechanism is a typical approach used for synchronizing concurrent processes involved in communication. The mechanism can provide a framework on which the layered communication architectures and protocols are efficiently implemented. The mechanism is realized under restricted environment where monitor facility is not available by means of simulation using semaphore.

要 約 범용의 분산 프로세스 통신제어 메카니즘은 통신하는 병렬 프로세스들을 동기시키는 데에 사용될 수 있는 일반적인 접근 방식이다. 이 메카니즘은 계층화된 통신구조와 프로토콜을 효율적으로 구현하기 위한 프레임워크를 제공할 수 있다. 모니터 설비가 제공되지 않는 제약된 환경에서 세마포를 이용하는 시뮬레이션을 통하여 이 메카니즘을 구현하였다.

1. Introduction

Implementation mechanism is a key factor of system performance of layered architectures and protocols of communications networks. The structure of functional modules within a layer, the way they interact, and interface mechanisms for different layers all together contribute to the resulting performance of network systems.

Sequential and concurrent processes approches can be used for implementing layered communication architectures and protocols, Each has merits and drawbacks. depending upon environments and application purposes. In this paper, a general purpose mechanism which can be used to implement arbitrary forms of communications control over distributed processes

*亞洲大學校 電子計算學科
Department of Computer Science, Ajou University
論文番號 : 91-36(接受1991. 3. 15)

in a network environment is suggested and developed. The mechanism is built on the basis of P. B. Hansen's monitor concept[2], employing typical forms of concurrent process interactions.

Since monitor facility is not available in the actual research environment, it was simulated by means of semaphore in order to realize its intended actions, showing that higher level IPC (Interprocess Communication) primitives in general can be realized through simulation using lower level primitives.

The mechanism is general in the sense that it can be easily tailored to meet specific requirements of specific applications. It also demonstrates network control programs can be made simple and reliable by using an abstract contruct based on any concurrent programming languages.

2. Design considerations of IPC Control

In this section, preliminaries, requirements (needed networks properties) and network environments for distributed processes communication are summarized.

• Logical Token Ring

It is assumed that processes are distributed among nodes in a logical token ring. However, there is no specific assumptions which may raise limitations over the physical configuration of the network.

• Nodes and Processes

On a node in a network, multiple processes can reside which may need to communicate with each other.

• Message Types

In other to reflect the four user-level primitives, a message is one of five types:

A-SEND; A-RECEIVE; A-FORCE-SEND; A-TOKEN; A-RECEIVE-ANY Description of each message type is given in Section 3.

• Processes, Ports, and Virtual Channels

Multiple application processes can reside in a node. Each process is initialized with access to one unique port number. This number can be used as either a send port or a receive port, depending upon whether it appears as the first or second parameter in an IPC operation. Using this scheme, two virtual channels that transmit data in opposite direction, exist between any pair of processes.

• Token As Access Right

The notion of a token is introduced into the network. Incoming messages from an input bus link which are not destined for the node that is receiving them are to be buffered and then output onto the node's output link regardless of whether the current node has possession of a token. However, a message origination from a node can only be output by that node when it is in possession of a token. Only on message at a time originating from th node that holds the token can be transmitted over the output bus link. Also, that message should follow all messages that are already buffered for transmission (FIFO basis) and it should be followed by a token. If the node receiving the token has no messages for transmission, then it simply buffers the token so that it will be sent onto the output link.

• FIFO Scheduling Policy

As already mentioned, messages are served with FIFO principle.

• Request for Transmission

No transmission takes place unless requested by a receiving party.

• Rendezvous Concept

A request for transmission and a corresponding response must be matched before transmission occurs. This means employment of the rendezvous concept.

• Strict Synchronization

With the tightly coupled synchronization policy provided in this mechanism, it is guaranteed that there is no message discarding due to buffer overflow and no deadlock.

• Promising Network Properties

The network guarantees message delivery within a finite time and also makes sure uniform transmission time.

• Network Sizing

It is envisaged that space and time requirment are proportional to the size or the network.

3. User-level IPC Operations for Distributed Processes

The following primitives are defined for providing users with services necessary for communication.

RECEIVE (rev-port, snd-port, VAR buffer)

A user process wishing to receive messages from other processes must first issue its willingness to do so by sending this message to the appropriate destination node.

SEND (rev-port, snd-port, item)

This message is sent by a process when it has data items to deliver to a destination process. Receive request and corresponding send response must make rendezvous at the node issuing SEND primitive. However, the order of occurrence of these two events need not be specified. Whichever event may occur first a rendezvous takes place as a prerequisite for message transmission process.

RECEIVE-ANY (rev-port, VAR buffer, VAR snd-port)

In order to bypass the request-response rendezvous at the sending node and thus to speed up the communication process when necessary, this primitive and the next one, FORCE-SEND are employed. The process at the destination delays after sending RECEIVE ANY message to receive any maching FORCE-SEND message.

Therefore, the concept of rendezvous still is alive although this is the send-receive rendezvous which is different from the other one, request-reponse rendezvous.

FORCE-SEND (rev-port, snd-port, item)

Without wating for a request, any process can send data via this primitive to a destination process.

4. IPC structures and Mechanisms for Distributed Processes

With respect to the four types of primitives available at user-level processes, IPC structures and mechanisms for communication control over distributed processes are constructed based on P. B. Hansen's monitor concepts.

4.1 Network Operation

The IPC is performed via two major schemes: rendezvous of requests and response; direct transimission via RECEIVE-ANY and FORCE-SEND.

• Rendezvous of Requests and Responses

A process wishing to receive message from other processes first announces its willingness to do so by issuing RECEIVE message into the network through accessing INPUTS monitor. The RECEIVE message is destined for a particular destination by identifying the destination port as parameter in the message.

The receiving process, after sending receive request, waits in a queue until a maching response returns to it. At that time the receiving process is awaken to pick up the data

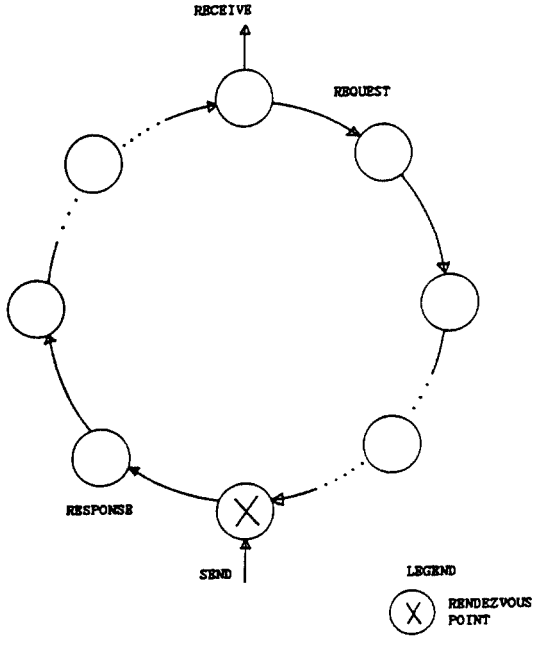

Fig. 1. Transmission Via Rendezvous of Requests and Responses

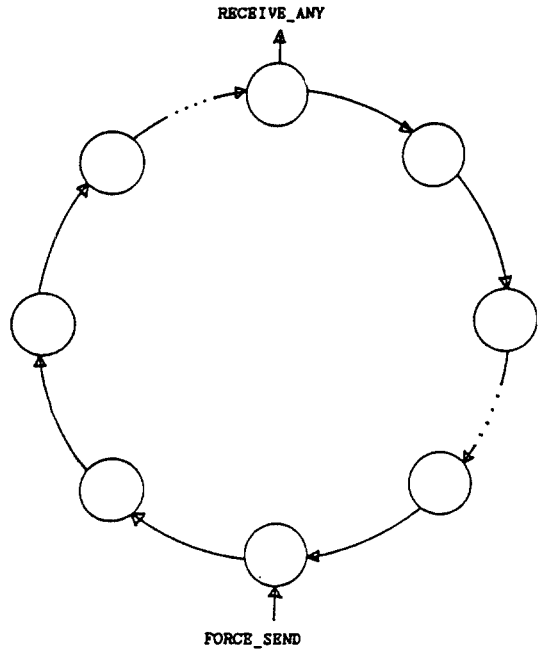

Fig. 2. Direct Transmission Via Receive-Any and Force Send

message.

On the other hand, a process wishing to send message to another process does so by issuing SEND message into the network through OUTPUTS monitor.

Rendezvous is required between two relevant processes at the sending node before the sending node can transmit.

Ready lists and monitor synchronization strategy are used for realization of the rendezvous. The order of which event occurs first by sender or receiver, however, doesn't make difference as far as the mechanism of rendezvous is concerned. Firgure 1 concerns about this process.

• Direct Transmission via RECEIVE-ANY and FORCE-SEND

Another strategy is provided which bypasses

the rendezvous concept, speeding up the com munication among processes. This is via intr oducing RECEIVE-ANY and FORCE SEND.

Any process which wishes to receive FOR CE-SEND message from other process unkn own to it can do so by issuing RECEIVE ANY message, waiting until arrival of FOR-CE-SEND message and being waken up upon occurrence of such event.

Meanwhile, other processes wanting to send without delay, message to that receiving pro cess can do so by issuing a FORCE SEND message. The sending process can proceed immediately in the OUTPUT monitor whenever the monitor is possible to access. Once in the monitor, there is no delay necessary for such thing as rendezvous.

4.2 IPC Structures and Mechanisms

A network node consists of the following system components: Application processes; Reader process; Writer process; INP monitor of type INPUTS; OUT monitor of type OUTPUTS; BUF of type BUFFER.

For the purpose of simulation in a single system, BUS monitor of type BUS LINK is added. BUS monitor is not required for real implementation. Figure 3 represents the conf iguration of system components for a node.

- INP (i) monitor

A process waiting to receive messages from other processes call INP monitor to ask it for issuing a receive request message (in this case the mssage type is A RECEIVE) and delayed until a maching send message (the message type is A SEND) arrives through the Reader process, which notifies the arrival to the waiting process by means of the signaling mechanism of the monitor construct.

A process can select another mechanism to bypass the above receive request-send response rendevzous process. This is done by a process simply accessing the receive-any entry in the INP monitor. In this case receiving process checks to see if the ready list associated with its receive parties set true or not, proceeding immediately to pick up messages or delayed

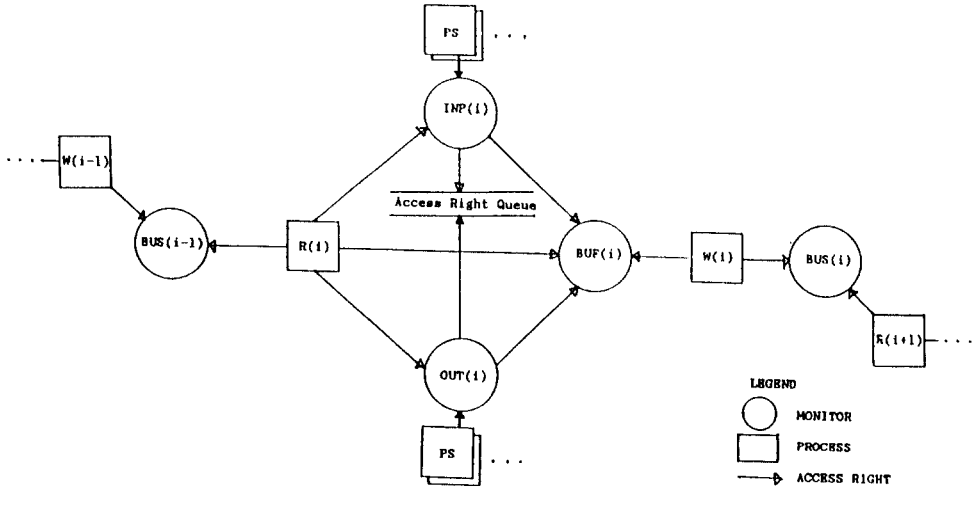

Fig. 3. Access Graph Among System Components of a Node

until being awoke respectively.

INP monitor is also accessed by Reader process when any response or FORCE-SEND message destined to the node arrived.

INP monitor possesses access rigth to BUF monitor in order to pass receive request message to BUF monitor.

• OUT (i) monitor

Processes who want to send messages to other processes have two means to do so: first by accessing the send entry to OUT monitor and second by accessing the force send entry to OUT monitor.

The former is using the receive request-send response rendezvous mechanism and the latter is using the alternative which is faster than the former.

When accessing the send entry, processes need prior checking of the ready list associated with its end port. With the ready list set true, the process can immediately proceed to send message without being delayed. Oterwise the process wait in the destinated queue until signaled to awake by Reader process upon reception of maching receive request. The alternative is to access the force-send entry to OUT monitor. With this scheme processes can immediately output message to BUFFER monitor. There is no rendezvous and thus no delay in the OUT monitor.

OUT monitor is also accessed by Reader process when the process receives request destined for the node. With this event occuring the reader process, in the request entry to OUT monitor, sets a flag in the ready list associated with send ports of processes residing in this node.

In addition, OUT monitor has access right to BUF monitor in order to pass send or force-send messages heading for other nodes.

• BUF (i) monitor

Transmit messages are directly forwarded onto this BUF (i) monitor by reader process. Messages stored in BUF(i) are fetched by writer process whenever it wants to.

• BUS (i) monitor

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

• Access Right Queue

The process who issued RECEIVE message, SEND message, or FORCE-SEND message to INP monitor or OUT monitor is delayed to possess a token in access right queue according to the FIFO policy.

When a token(access right) comes into Reader process, the process delayed in the head of access right queue is awaked and the message is sent to BUFFER monitor. If access right queue is nill then incoming token is passed to BUFFER monitor immediatly.

• R(i) Reader process

This process reads in messages coming in from the line and appropriately processes them according to their types.

The process is in possession of access right to three monitors: INP, OUT, and BUF.

Upon reception of response or force-send messages the process calls response and any-reponse entries to the INP monitor, respectively, to pass them to destination processes.

Corresponding to receive request message coming into this node, the process accesses the request entry to OUT monitor to set the

flag of ready list true, causing rendezvous to occur.

For a message not destined to this node the process simply calls the BUF monitor to directly forward them onto the line.

• W(i) Writer monitor

This process picks up transmit messages stored in BUF(i) monitor and hands them to R(i+1) process through BUS(i) monitor.

5. Implementation

With the availability of CPASCAL the system implementation is straightforward. When monitor is not available, any form of simulation using low level synchronization methods like semaphores is inevitable.

In general, it is possible to simulate one IPC primitive into another among different levels of synchronization methods[2].

In either real implementation or simulation, programming a trace monitor as a development tool would be of value for keeping track of what is going on in the network.

The source program for each component of the mechanism is presented in the appendix with the monitor construct replaced with appropriate simulation using semaphore.

6. Conclusion and Remark

It has been shown that it is possible to construct a useful synchronization mechanism for controlling distributed processes communication in the network environment using monitor concept.

By virtue of the simple, clear and reliable structure intrinsic to the monitor, for providing mutual exclusion to shared resources among mutiple concurrent process, a clear-cut synchronization can be efficiently achieved.

The methodology is general in the sense that it can be easily tailored in order to fit to a variety of requirements under real situation.

The availability of the methodology for use in communication control under OSI architecture seems straightforward.

It desen't seem difficult to extend its structure and add features necessary for specific requirements for specific application.

BIBLIOGRAPHY

1. Dong Kyoo Kim, Computer communication network, Sung jo Publications Compony, Inc., 1986.
2. P.B. hansen, The architecture of concurrent programs, Prentice Hall, 1973.
3. V.E. Wallentine, W.J. Hankly, SIMMON A concurrent PASCAL based simmulation system, Kansas State University, 1978.

This paper is an outcome of the research performed under the support of 1986 research fund granted from Korea Research Foundation. Its main content was published in the Proc. of 2nd International Joint Workshop on computer Communications held in Japan in 1987.

APPENDIX

• System initialization

```
TYPE
    channelset =SET OF INTEGER;
    item       =ARRAY[1..max] OF CHAR;
    kinds      =(a_response, a_request, a_forsed, token);
    channel    =RECORD
                  sendport    : INTEGER;
                  receiveport : INTEGER;
                END;
    message    =RECORD
                  kind : kinds;
                  link : channel;
                  content : item;
                END;
```

• Monitor implementation

We need four monitors, INPUT monitor, OUTPUT monitor, BUFFER monitor and Access Right Queue in one node. Each enrties of the monitors and algorithm using semaphore are introduced

(1) INPUT monitor

INPUT monitor has four entries, RECEIVE, RESPONSE, RECEIVE_ANY, FORCED.

```
(* send a request through token access and deleay a calling
(* process untill a responses arrives
PROCEDURE receive (rcv_port, snd_port : VAR buffer);
VAR this : message
BEGIN
    WITH this DO
    BEGIN                    (* set request message
      kind:=a_request;
      link. receiveport:=rcv_port;
      link. sendport :=snd_port;
    END;
    delay_on_token ;                (* call delay_on_token
    put_buffer (this);              (* save this message onto buffer monitor
    p (response_delay);             (* wait on response
    p (inbuffermutex);              (* store responsed message to its buffer
```

```
      buffer :=inputbuffer;
    V (inbuffermutex);
    V (escmutex);                    (* make reader to be escaped from
                                     (* reponse entry
END; (* receive end *)

(* delivers response content and wake up the process that is waiting
(* to receive it
PROCEDURE response (m:message);
BEGIN
    WITH m DO
    BEGIN
        P (inbuffermutex);           (* store responsed content on
          inputbuffer : =content;    (* inputbuffer which task process
        V (inbuffermutex);           (* share
        V (response_delay);          (* signal response
        P (escmutex);                (* prevent another process entering
                                     (* until receive process complete
    END ;
END ; (* response end *)

(* ready to receive the foreced message
PROCEDURE receive_any (snd_port ; VAR buffer) ;
BEGIN
    P (forced_delay) ;               (* wait on forced message
    P (inbuffermutex) ;              (* store forced content to its buffer
      buffer : =inputbuffer ;
    V (inbuffermutex) ;
    V (escmutex) ;        (* make reader to be escaped from)
                          (* force entry
END ; (* receive_any end *)

(* delivers forced content and wake up the process that is waiting
(* to receive it
PROCEDURE force (m : message) ;
BEGIN
    WITH m DO
    BEGIN
        P (inbuffermutex) ;          (* store forced content on
          inputbuffer : =content ;   (* input buffer
```

```
        V (inbuffermutex) ;
        V (forced_delay) ;          (* signal force
        P (escmutex) ;              (* prevent another process entering
                                    (* until receive process complete
    END ;
END ; (* forced end *)
```

(2) OUTPUT monitior

OUTPUT monitor has three entries, SEND, REQUEST, and FORCED_SEND.

```
(* delays a calling process until a given channel is ready for
(* transmission
PROCEDURE send (rcv_port, snd_port, item) ;
VAR this : message ;
BEGIN
    WITH this DO
    BEGIN                           (* set response message
        kind : =a_response ;
        link. receiveport ; =rcv_port ;
        link. sendport : =snd_port ;
        content : =item ;
    END ;
    P (request_delay) ;     (* wait on request
    delay_on_token :        (* call delay_on_token
    put_buffer (this) ;     (* save this message onto buffer monitor
END (* send end *)
(* makes a channel ready to send and continue process waiting to
(* send on that channel
PROCEDURE request (m : message) ;
BEGIN
      (request_delay) ;     (* signal request
END ; (* request end *)

(* send directly message to process that ready to receive
(* in receive_any entry
PROCEDURE forced_send (rcv_port, snd_port, item) ;
BEGIN
    WITH this DO
    BEGIN                   (* set request message
      kind:=a_forced ;
```

```
        link. receiveport:=rcv port;
        link. sendport :=snd port;
        content : =item ;
    END;
    delay_on_token ;                    (* call delay on token
    put_buffer (this);                  (* save this message onto buffer monitor
END ; (* forced send end *)
```

(3) Access Right Queue

Access right queue has two entries DELAY ON TOKEN and WAKE_TOKEN_DELAY.

```
PROCEDURE delay_on_token ;
BEGIN
    P (token_delay) ;
END ;

PROCEDURE wake_token_delay ;
BEGIN
    IF NOT tokendelay=empty THEN
        V (tokendelay) ;
END ;
```

(4). BUFFER monitor

BUFFER monitor has two entries, GET_BUFFER, and PUT_BUFFER.

```
(* delays a calling process as long as buffer is empty.
(* it then gets a message from the buffer and continue
(* the execution of another process waiting to send a message
PROCEDURE get_buffer (VAR m : message)
BEGIN
    P (buffermutex) ;
    IF buffer=empty THEN            (* delays a calling process
      P (bufferfull) ;              (* as long as buffer is empty
    get a_buffer from buffer pool ;
    m : =a buffer ;
    V (bufferempty) ;               (* wake up delaying process
                                    (* in buffer full
    V (buffermutex) ;
END ; (* get_buffer entry end *)
(* delays a calling process as long as buffer is full.
```

```
(* it then puts a message into the buffer and continue
(* the execution of another process waiting to receive message
PROCEDURE put_buffer (m : message)
BEGIN
    p (buffermetex) ;                (* delay a calling process
    If buffer=full THEN              (* as long as buffer is full
      P (bufferempty) ;
    put a_buffer onto buffer pool ;
    V (bufferfull) ;                 (* wake up delaying process
    V (buffermutex) ;                (* in buffer empty
END ; (* put_buffer entry end *)
```

(5) BUS LINK buffer

Bus link buffer has two entry, INPUT_FROM_BUS and OUTPUT_INTO_BUS.

```
(* delay a reader process as long as bus_link is empty.
(* it then gets a message from the bus_link and continue
(* the execution of reader process
PROCEDURE input_from_bus (VAR m : messge) ;
BEGIN
    P (busfull) ;
    get a_message from bus link buffer ;
    m : =a_message ;
    V (busempty) ;
END ; (* input_from_bus entry end *)

(* delay writer process as long as bus_link is full.
(* it then puts a message into bus_linke and continue
(* the execution of write process
PROCEDURE output_onto_bus (m : message) ;
BEGIN
    P (busempty) ;
      put a_message into buslinkbuffer ;
    V (busfull) ;
END ; (* output_into bus entry end *)
```

• Process Implementation

We need Reader process, Writer process, and Application processes which exist in each node.

(1) Reader process

```
(* input one message at a time from previous node
(* through a bus link.
(* if the channel is destination of a message, the reader performs
(* a response, request, or forced operation on it.
(* otherwise, send the message through a local buffer to next node
READER PROCESS
BEGIN
    WHILE true DO
    BEGIN
        input a message from buslink
        IF channel# of this message in my channel set
        THEN
        CASE kind OF this message ;
        a_response : inputmonitor.response (mesg)
        a_request  : outputmonitor.request (mesg) ;
        a_forced   : inputmonitor.forced (mesg) ;
        token      : wake up token delay process ;
                     put buffer (token) ;
        END case
        ELSE
          put bufer (this message) ;
END while
```

(2). Writer process

This process operate as discussed in above section.

```
(* receives one message at a time from a local buffer and
(* output it to the next node through a bus link.
WRITE PROCESS
BEGIN
    WHILE true DO
    BEGIN
        get a_message from buffer monitor ;
        output a_message into buslink ;
    END ;
END (* end writer process)
```

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