DISTRIBUTED COMPUTING ON MICROCOMPUTER NETWORKS*

R. Bagrodia, K. M. Chandy, J. Misra

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-26 July 1987

Abstract

This paper proposes four constructs which, when added to a sequential programming language, yields a programming notation suitable for distributed implementation. An implementation of the constructs on a VAX 11/750 network is discussed.

*Research for this project was supported by a grant from the IBM Corporation.

1. Introduction

Low-cost microcomputer networks have brought distributed programming within reach of the microcomputer user. Distributed computing offers many advantages which have been extensively discussed in the literature [17], [19]. However, many microcomputer users are unwilling to learn new languages to obtain the advantages of distributed computing. This paper presents a simple language fragment and a kernel that can be used by application programmers in conjunction with familiar sequential languages (such as PASCAL and FORTRAN) to exploit the benefits of distributed program execution. An important objective is to stimulate rapid implementation of distributed programming environments on networks of microcomputers.

When developing distributed programs, a programmer familiar only with sequential programming languages faces a number of unfamiliar problems. The problems of deadlock prevention among communicating processes, termination detection, and the allocation of processes among processors are all problems specific to distributed programs. Although resource deadlocks may arise in a multiprogrammed uniprocessor environment, they are normally handled by the operating system. In distributed languages that are based on message-passing, processes may enter into communication deadlocks. In the absence of a sophisticated operating system, programmers writing distributed programs must ensure that the program is free of deadlocks. The primitives proposed by our language fragment provide a simple mechanism to prevent deadlocks. The kernel also provides a user-transparent facility to detect program termination in a distributed manner. Simple heuristics based on the utilization of each processor are provided to aid the user in allocating processes among the available processors in an efficient manner.

This paper does not propose a new language; rather, we isolate a few constructs that can be added to any sequential programming language to yield a distributed programming language. We thus hope to provide system and application programmers the advantages of distributed computing within the framework of the sequential programming language that they are most familiar with. This approach has the advantage of simplicity : simplicity in learning concepts about distributed programming and simplicity of implementation. The kernel described in this paper has been implemented to develop CMAY - a distributed programming language derived from FORTRAN. CMAY has been implemented on UTCSRES - a local area network of VAX 11/750s at the University of Texas at Austin. Efforts to implement CMAY on a network of IBM personal computers are in progress.

A variety of constructs have been proposed to design, specify and verify distributed programs. Various combinations of these primitives have resulted in the design of a large number of concurrent programming languages including CSP [14], Ada [1], DP [13], GYPSY [2], PLITS [10], MODULA [20], SR [3], NIL [18], STARMOD [7], CLU [16], and MESA [12]. The design of these languages have facilitated elegant solutions to many basic problems of distributed computation, operating system construction, program verification and the design of reliable, fault-tolerant systems. In some cases, the number and complexity of the new constructs proposed may hinder wide acceptance by the general programming community. In this paper, we show how incorporation of two basic primitives - **entities** and **messages**, in any general purpose sequential programming language like FORTRAN, PASCAL, C etc. can be used to obtain a distributed programming language.

The rest of the paper is organized as follows: section 2 presents an informal discussion of the basic notions of **entity** and **message**. Section 3 discusses the constructs provided by the language fragment: process representation, process creation and termination, process synchronization and process communication. This section describes how the above facilities may be provided by means of operations performed on **entities** and **messages** and compares our approach with the approach adopted in the design of CSP, DP and Ada. Section 4 illustrates the use of the language fragment in developing distributed programs. Section 5 describes the facilities offered by the operating system kernel to execute programs on computer networks. Section 6 is the conclusion.

2. Distributed Programming : An Overview

In this section we present an overview of our approach to distributed programming and informally describe the constructs introduced by our language fragment. The programming language in which these constructs are to be implemented is called the host language.

A distributed program is a collection of sequential processes which may execute concurrently. Each process executes independently of the others except for specific points in its computation when it sends or receives messages. Different sequential processes may run on different computers linked by a network. Processes communicate exclusively via messages. Processes may be executed on heterogeneous computers with different processor speeds, connected to different peripheral devices and running different operating systems. The individual operating systems may be single-user, multiprogrammed or multitasking. We assume that the communication software provides error-free virtual connection between any two computers on the network and apart from this feature make no assumptions regarding the operating system.

The distributed programs developed within the framework of our language fragment possess the following properties:

1. Flat program Structure: A distributed program consists of a collection of entity definitions and sequential modules (procedures, subroutines, etc.) of the host language. The fragment does not impose a hierarchical program structure or require a specific textual ordering of the entity definitions.

2. Program Modularity: The entity construct introduced by the language fragment adopts an object-oriented approach to program development. The entity definition completely encapsulates the description of the corresponding object and hides implementation details from the rest of the system. The language fragment thus directly facilitates the development of modular programs, irrespective of the specific host language being used.

3. Separation of program design from its implementation: By providing a consistent communication mechanism for communication between remote entities (entities executing on different processors) and local entities, the language separates the program design and development from its actual implementation on a specific processor configuration.

Entities are the basic building-blocks of a distributed program. An entity is a sequential program module implemented in the host language with the following additional features: an entity may

1. create other entities;
2. terminate itself;
3. send messages to other entities;
4. receive messages;

An **entity** is an independent self-contained unit, which is used to model processes. An **entity** type, like the SIMULA **class**, is used to define objects of a given type. Various instances of an entity type may be created dynamically to represent the many objects of a given type. An entity instance is created by executing a **let** statement. Hereafter, we shall use the term entity to mean an instance of an entity type. Unlike the local variables of a class-instance, the local variables of an entity cannot be accessed by other entities. Entities communicate via messages. On being created, an entity is assigned a unique identifier. The identifier is bound to the entity for the lifetime of the entity. In order for an entity to send a message to another, it must have access to the latter entity's identifier. A message is viewed as a specific instance of a message type. A message type consists of a name and a list of message parameters. An entity sends a message to another by executing an **invoke** statement. Message sending is non-blocking: messages sent by an entity are deposited in the receiving entity's message buffer; the sending entity is not delayed. An entity accepts messages from its buffer by executing a **wait** statement. If a desired message is not present in the buffer, the entity waits for the message. The wait may be indefinite or specify a time-out interval. In the first case, the entity ceases to wait only when the desired message is received by it. In the latter case, if the desired message is not received by the entity within the specified time period, the entity will eventually time out and thus cease to wait. On ceasing to wait, an entity proceeds to the next statement in its code. An entity refers to the type of the last message received by the keyword **message-type**. For instance, on ceasing to wait, an entity may execute a statement of the form

```
if (message-type = send) then  do x
else if (message-type = time-out) then do y
     else do z
```

Initially, every CMAY program consists of a single entity called **main** executing on one processor. The purpose of entity **main** is to initiate the execution of the program.

We illustrate the concepts described above by means of an example which implements the sieve of Eratosthenes [14]. This algorithm identifies successive prime numbers from a sequence of consecutive natural numbers. We define an entity type called *sieve* to implement the algorithm. Multiple instances of the *sieve* entity are created - one for each prime number that has already been identified in the sequence. The various *sieve* entities form a pipeline. Each *sieve* entity in the pipeline inputs numbers from its predecessor, suppresses those that are multiples of the original prime and passes the rest to the successor entity.

The CMAY code to implement this algorithm is displayed in pseudo-code in Figure 1. Entity **main** (lines 0-10) is used to initiate the program. The **main** entity creates the first *sieve* entity whose unique identifier is stored in its local variable *first_sieve* (line 5). **Main** sends a stream of integers 2,3,4,5.... to entity *first_sieve* via messages of type *next_number* (lines 7-8). The types of all messages that may be received by an entity must be defined within the corresponding entity type definition. For instance, the *sieve* entities may receive messages of type *next_number*. This message type is defined in line 17. The first element of the stream of numbers received by a *sieve* entity is a prime. For instance, the first number (i.e. 2) received by the *sieve* entity *first_sieve* is a prime number, and is stored in the entity's local variable *my_prime* (line 22). At this point, entity *first_sieve* creates a new instance of the *sieve* entity-type. In general, the ith *sieve* ($i>1$), say s_i, is recursively created by the (i-1)th *sieve* (line 25). Subsequently, s_i removes all multiples of *my_ prime* from the sequence of numbers received by it and passes the rest onto *sieve* s_{i+1} via messages of type *next_number* (lines 33-34). A *sieve* entity receives (or waits to receive) the next message of type *next_number* by executing the **wait** statement in line 29.

If desired, each *sieve* entity in the above algorithm may be assigned to a separate processor. The issue of entity allocation amongst processors is discussed in section 5.1. Program termination is discussed in section 5.3. The point we wish to emphasize by this example is that a straightforward implementation of our simple language fragment allows one to write recursive, modular programs even in languages such as FORTRAN.

```
0    entity main;
1    { Local Variable Declaration Section }
2    first_sieve : entity-identifier;
3     i:integer;

4    { Entity Body }
5        let first_sieve be sieve;
6    {   send a stream of numbers 2,3...1000 to first_sieve }
7        for i := 2 to 1000 do
8           invoke first_sieve with next_number(i);
9
10    end-entity;


11    entity sieve;
12
13    { Local Variable Declaration Section }
14    next-sieve : entity-identifier;
15    my_prime: integer;
16    { Message Receive Declaration Section }
17    message next_number(number:integer);
18
19    { Entity Body }
20    {     Wait  to receive the next integer in the sequence     }
21           wait for ( message-type = next_number);
22           my_prime:= number;

24     {     Create the next sieve process }
25           let next-sieve be sieve;
26
27           while true do
28           begin
29               wait for (message-type = next_number);
30
31      {     From the subsequent messages received, sieve out all multiples of
32            prime and pass the rest to  entity next-sieve.   }
33               if (mod(number,my_prime) <> 0) then
34                   invoke next-sieve with next_number(number);
35            end;
36     end-entity;
```

Figure 1: CMAY Implementation of Sieve of Eratosthenes

3. Primitives for Distributed Computing

The issues involved in the design of a distributed programming language are facilities for [4]:

1. Definition of units of concurrent computation (called processes) and specification of concurrent execution of processes;
2. Definition of communication primitives;
3. Definition of synchronization primitives.
4. Definition of a kernel to support distributed program execution on multiple computers.

In this section we mention the various approaches adopted in the literature to provide the above facilities. We describe the primitives implemented in CMAY and compare our approach to that adopted in the design of CSP, DP and Ada. Table I summarizes the primitives and facilities provided by CSP, DP, Ada, and CMAY to construct distributed programs.

3.1. Process Definition

A variety of program structures have been proposed in the literature to model concurrently executing sequential processes: UNIX **coroutines**, SIMULA **classes** [8], Concurrent PASCAL **processes** and **monitors** [6], CSP **parallel processes** [14], CLU **clusters** [16], ADA **task** and **packages** [11], DP **distributed processes** [13], and ARGUS **guardians** [17] are some examples.

In CSP, processes are independent units which interact with each other by means of communication statements. A process is defined by a list of commands, which may include input/output, alternative and repetitive commands. The list may be preceded by a label which serves to name the process. The local variables of each process are defined within its command list. Although CSP does not provide process valued variables, a family of processes can be described by using subscripted labels as the process name.

DP, like CSP, merges the traditional concepts of resources (abstract data types) and processes (active data types) into a single primitive, that of a DP process. A DP process defines local variables (similar to *own* variables of ALGOL), a set of service procedures that operate on the local variables and an initial statement. The variables and procedures represent the resources used by the process. A process cannot access the variables of another process, but may call the service procedures defined within another process. On being created, a process begins to execute its initial statements and continues until the statement is terminated or the process has to wait for some condition to become true. At this point it can satisfy the external requests of other processes. Thus in DP, the process definition itself serves as a computation unit. When one of its service procedures is called, a new server process is created to execute the body of the procedure. The various server processes and the host process execute concurrently on the host processor.

Ada provides two basic primitives, **packages** and **tasks**, to model passive resources and active data types respectively. A **task** has of a specification part and a task-body part. The specification part specifies the resources made available to the user by the task. These include **entry** resources which allow this task to communicate with other tasks. The task body consists of the sequence of statements (including communication statements) that are to be executed when the task is initiated. The body may define local variables which are not accessible by the task user. Ada also provides the mechanism to define a generic task, which may be multiply instantiated.

Many concurrent languages provide two distinct primitives to model abstract data types and processes respectively (e.g. the **package** and **task** primitives of Ada). In keeping with our goal of simplicity, CMAY provides a single abstract mechanism, called an **entity**, to serve *both* purposes of data abstraction and to model active processes.

The entity type declaration has a format that is similar to the declaration of a procedure or subroutine in the host language. The entity heading declares the name and formal parameters of the entity type in a manner similar to the declaration of a procedure heading in the host language. An entity type, however, is only allowed to have input parameters. The local variable section of an entity is identical to that of a procedure. The message declaration section of an entity type is used to declare the various types of messages that may be received by an entity. A message declaration consists of the string **message** followed by a name and a parameter list. The structure of the parameter list is similar to that of a formal parameter list in the host language.

The entity body consists of sequential statements of the host language (e.g. assignment statement, procedure call etc.) with the following additional statements:

- **let** statement : used to create new entities.
- **end-entity** statement : used by an entity to terminate itself.
- **invoke** statement : used to send messages to other entities.
- **wait** statement : used to wait to receive messages.

3.2. Process Creation And Termination

Depending on the nature of intended applications, the process representation structures (coroutines, entities etc.) may be used to specify a *static* number of process instances or permit processes to be created dynamically. In DP, processes are created statically, with one process being assumed for every processor in the network. Further, since a DP process is assumed to exist forever, no termination mechanisms are provided in the language. In CSP, processes are created dynamically. However, due to the declarative nature of the language, there exists a fixed upper bound on the number of processes that can be created. A CSP process terminates automatically when the command list has been executed. In Ada, tasks are created dynamically and may be created recursively. Ada views the instantiation and initiation of a task as two distinct operations. A generic task is instantiated using the **new** statement and may then be initiated by using the **initiate** statement. Ada provides a host of facilities for the termination of tasks. In particular, the **abort** statement may be used to terminate any task in the system. In CMAY, entities are created dynamically by executing a **let** statement and may be created recursively. All entities execute concurrently. An entity terminates itself by executing the **end-entity** statement in the entity definition. An entity cannot be terminated by another entity.

We define a scalar variable type called **entity-identifier**. Every entity in the program has a unique identification number which is stored in a variable of type **entity-identifier**. Variables of this type are used exclusively to store the identifier of entities and no operations (arithmetic, assignment, input/output etc.) can be performed on them.

An entity is created by the execution of a **let** statement which has the following form:

let *e1* **be** *entity-type-name*(actual parameter list)

The identifier of the new entity is stored in variable *e1* which must be of type **entity-identifier.**

The formal parameters of the entity type declaration are bound to the actual parameters in the **let** statement, as in the manner of a procedure call, at the point that the entity is created. The variable *e1* may be used by the creator of *e1* to send messages to it. Every entity type declaration contains a pre-defined local variable, called **myid**, which is of type **entity-identifier**. When a new entity is created, its identifier is automatically stored in its **myid**.

3.3. Process Communication

Concurrent processes communicate with each other to exchange information. The actions of a process may need to be conditionally delayed (or synchronized) in order to avoid interference with other processes in the system. Shared variables and message passing are the two fundamental mechanisms that have been used to provide process communication and synchronization facilities. In the shared variable approach, two or more processes access a common memory location to exchange information. A variety of protocols and constructs have been proposed to enable processes to synchronize their access to the shared resources with each other. Semaphores, conditional critical regions, path expressions, and monitors are some examples. Although the shared variable approach is very useful when large blocks of data need to be shared, it has two major drawbacks:

1. It is difficult to combine a high degree of parallelism and data sharing effectively [10].
2. Shared access to data can be efficiently implemented only on network configurations that provide hardware support for such access.

In the message based approach to process communication, the communication primitives may be provided as two separate commands - **send** and **receive**, or be packaged together as a single high-level primitive - (remote) procedure call. A remote procedure call has essentially the same semantics as a local procedure call. Executing a remote procedure call is equivalent to executing a **send** to transmit the input parameters to the invoked procedure, and then executing a **receive** to obtain the result parameters. In the message-based approach, a variety of related issues need to be resolved:

Blocking:	A primitive is considered non-blocking, if its execution does not cause the invoking process to be delayed. If both send and receive are implemented by blocking primitives, the communication is said to be fully synchronous (e.g. CSP); if neither of the primitives block, communication is fully asynchronous.
Addressing:	The source and destination names in the send and receive primitives may explicitly name a process (direct naming), name a port (port naming), or in the case of a receive not name any source (asymmetric naming).
Miscellaneous :	Some of the other issues that need to be addressed are message size (fixed versus variable length messages), types of permissible message parameters, etc.

In message-based distributed languages, three communication protocols have frequently been used in the literature:

1. Buffered Communication (BC): Execution of the **send** primitive causes the sending process to wait until the *message has been deposited in a buffer*.

2. Synchronous Communication (SC): The sending process waits until the *message has been received* by the other process.

3. Remote Procedure Call (RPC) : The sending process waits until it *receives a reply* to its message.

In each of the above protocols, execution of a **receive** primitive typically causes the process to block until it receives a message. We compare the three protocols described above in the areas of applicability, parallelism and program correctness.

Applicability: Buffered communication is the most flexible of the protocols described above and can be used to implement the other protocols. Remote procedure calls are a convenient abstraction for client/server type communications, where a client process must wait until the service requested from the server process has been completed.

Parallelism: Buffered communication does not restrict the inherent parallelism in a program, since the sending process is not blocked. In the case of synchronous communications, the programmer must explicitly define buffer processes between two communicating processes, in order to insure that a process is not blocked unnecessarily. Finally, remote procedure calls completely inhibit parallelism, since the invoking process is forced to wait until its request has been processed.

Program Correctness: In synchronous communication, a process sending a message can locally assert that the message has been received by the other process. This greatly facilitates the task of constructing correctness proofs for the program. In buffered communications, some auxiliary information must be included in a message to facilitate the construction of correctness proofs.

We examine the specific communication primitives provided by some distributed programming languages. CSP provides a fully synchronous, direct naming communication primitive. This primitive is easy to use and implement and is very useful for programming pipelined process configurations. However, in other instances, this form of static direct naming may be awkward to use - for instance, in multiple client/single server situations. This problem may be solved by specifying a *port* as the source process as has been suggested in [15]. Further, fully synchronous communication can easily cause processes to deadlock.

DP provides remote procedure calls as its basic communication primitive. Although this primitive is especially convenient in modelling client/server relationships, it has some drawbacks:

1. If a majority of process-interactions do not require a reply, then the remote procedure call facility may be very inefficient.

2. Processes may get deadlocked for the following reason: when a process P calls a procedure R within another process Q, R is considered to be an indivisible operation within P and P is blocked till R completes. Thus if procedure R calls the service procedure R' defined within P, P and Q would be deadlocked.

Ada merges the remote procedure call of DP with the *rendezvous* concept of CSP. The process naming is asymmetric with only the calling task naming the called task. The *rendezvous* is achieved between an **accept** statement in the called task and an entry call in the calling task. However, **accept** statements are part of the task activity and do not constitute separate service operations packaged as procedures, as was the case in DP. This approach has the advantage of permitting different service requests to be processed differently by the servers.

CMAY provides a buffered communication protocol. Execution of the CMAY **send** primitive causes the sending entity to be delayed only if the processor has run out of buffer space. However, if virtual memory is used to implement buffers, a non-blocking **send** can be implemented effectively. Execution of the CMAY **receive** primitive normally causes a receiving process to block if the desired message is not present in its message buffer. The primitive may be used to specify a maximum time (called the *time-out* time) for which a process waits to receive a message. By specifying a zero *time-out* time, a non-blocking **receive** may be implemented (see discussion below under process synchronization).

Messages are sent by one entity to another using an **invoke** statement which has the following form:

invoke *e1* **with** *m1*(actual-parameter-list)

e1 must be of type **entity-identifier**. Execution of the above statement results in a message of type *m1* being sent to the entity *e1* provided entity *e1* exists. If the recipient entity is not ready to accept a message sent to it, the message is stored in a message-buffer associated with the recipient entity and may be accepted by it subsequently as discussed in the next section.

3.4. Process Synchronization

Distributed processes need to be synchronized with one another. From our discussion in the previous section, we note that the communication primitives may be used to synchronize processes in message based languages. However, in many cases, a more selective form of process synchronization is desirable. To take a simple example, an empty buffer process cannot satisfy a consumer process's request for data. In this case, the consumer process must wait till the buffer is non-empty. But if the buffer is non-empty, the consumer process's request for data should not be delayed.

The two primary mechanisms for process synchronization in message based languages are guarded commands [9] and conditional receives. CSP, DP and Ada all implement process synchronization using different forms of the guarded command. CMAY provides synchronization through the use of conditional receives.

In CMAY, an entity executes a **wait** statement to receive messages. The statement has the following form:

wait [*t*] [**for** *b*]

where *t* is an integer valued expression representing time; and *b* is a boolean expression that may reference any local variables of the entity. The predicate *b* is used by an entity to specify the message(s) it is ready to accept. The value *t*, referred to as the *time-out* time, represents the length of time the entity is willing to wait for the desired message(s). If the time value *t*, is omitted in a **wait** statement, the entity will wait indefinitely till the desired message is received. The **for** clause may be omitted from the **wait** statement, if a non-selective receive is to be implemented. Execution of a wait statement causes the entity to wait if the desired message is not present in its buffer and if the **wait** statement specifies a non-null *time-out* interval. If the entity is waiting for a specific message(s), other messages received by the entity are stored (in the order they were received) in a message buffer associated with the entity. A waiting entity ceases to wait when it is delivered a message that satisfies the condition *b* or if it receives a **time-out** message from the monitor. A **time-out** message is sent to the entity if no message satisfying condition *b* is received within *t* units of time from the time the **wait** statement was executed.

The time-out facility is a simple mechanism to provide facilities for real-time systems as well as to avoid process deadlocks. Due to the non-blocking nature of our send primitive, the only manner in which entities may enter a deadlock is on execution of a **wait** statement. The time-out facility can be used to insure that after a finite interval of time, an otherwise blocked entity ceases to wait. Further, the time-out mechanism may be used to implement a non-blocking receive. Consider, as an example, execution of the following **wait** statement by an entity *e1*:

wait *0* **for** *b1*

If a message satisfying predicate *b1* is present in the message buffer of entity *e1*, it is delivered to *e1*. However, if no such message exists, specification of a null time-out interval insures that the entity is not blocked; instead, a **time-out** message is sent to the entity to enable it to continue its execution.

4. CMAY Examples

This section presents some examples to illustrate how abstract data-types and processes may be represented as entities in our programming language fragment. The last example in the section illustrates the use of CMAY primitives to develop distributed applications.

4.1. Bounded Buffer

As a simple example, we model a bounded buffer process by an entity type called *buffer*. The entity receives messages of type *append* to add data to the buffer and of type *request* to remove data from the buffer. If the *buffer* receives *request* (*append*) messages when it is empty (full), the messages are stored in a queue associated with the *buffer* and accepted by it only when it is non-empty (non-full). We assume that when requesting (appending) data, a *consumer* (*producer*) entity sends its identifier as a message parameter to the *buffer* entity. Further, assume that the *consumer* entity contains the definition of a message *receive*, to receive data from the *buffer* and the *producer* entity defines a message called *ack* to acknowledge receipt of its data by the *buffer* entity.

```
entity  buffer(buffer_length:integer);

(*  Local Variable Declaration Section *)
    in, out  : integer;
    full, empty:logical;
    store:array[1..buffer_length] of integer;

(*  Message Receive Declaration Section *)
    message append(value:integer,producer:entity-identifier);
    message request(consumer: entity-identifier);

(* Entity Body *)
(*  initialize the buffer*)
    in:=0;
    out:=0;


    while true do
    begin
        full:= (in =  out+buffer_length);
        empty:= (in  =  out);
        wait  for (not empty and message-type=request)
              or  (not full and message-type  =  append)

        if (message-type  =  append) then
        begin
            store(( mod(in,buffer_length)) + 1)  := value;
            in  := in+1;
            invoke producer with ack;
        end
        else if (message-type  =  request) then
        begin
            value:=store(( mod(out,buffer_length)) +1);
            out:=out+1;
            invoke consumer with receive(value);
        end;
      end;
  end-entity;
```

Figure 2: CMAY Implementation Of A Bounded Buffer

4.2. Mergesort

This example illustrates how a recursive mergesort algorithm may be coded as an entity in our language fragment. The algorithm is used to sort an array of integers. It splits an array into two parts, sorts each part recursively and then merges the two sorted parts. For simplicity of exposition, we assume that the array to be sorted contains n elements, where for some $m >= 0$, $n = 2^m$.

We use an entity type called *merge_sort* in our implementation of this algorithm. An instance of entity *merge_sort*, say m_i, is created dynamically. An unsorted array called *unsort* is sent to m_i via a *sort* message. If the array has only one element, it is sent back to the creator of m_i via a *reply* message. Otherwise entity m_i splits the unsorted array into two equal parts. It recursively creates two instances of entity *merge_sort* say m_j and m_k to recursively sort each half. M_i then waits to receive the two sorted halves (which may be received in any order), merges them together and sends the sorted array to its creator by means of a *reply* message. At this point, m_i terminates itself by executing the **end-entity** statement.

Entity **main** initiates the program by reading in the unsorted array. It creates an instance of entity *merge_sort* and passes the unsorted array to it in a *sort* message. It then waits to receive the sorted array from this entity via a message of type *reply*. It prints the sorted array and then terminates itself. The CMAY program to implement the algorithm is presented in Figure 3 in PASCAL-like pseudo-code. The code for entity **main** has been omitted for brevity.

```
entity merge_sort(max_size:integer);

(* Local Variable Declaration Section *)
merged,temp1,temp2:array[1..max_size] of integer;
i,temp_size:integer;
m1,m2:entity_identifier;

(* Message Receive Declaration Section *)
message reply(size: integer; sorted:array[1..size] of integer);
message sort(sender_id:entity_identifier; size: integer;
                                    unsorted:array[1..size] of integer);

(*  entity body   *)
    wait for (message-type=sort)

    if size = 1 then
       invoke sender_id with reply(size,unsorted);
    else
    begin
 (*  Split unsorted array into two halves.  Create two merge_sort entities
     and send each half to a merge_sort entity                    *)
        temp_size:=size/2;
        temp1:= first-half of array unsorted;
        temp2:= second-half of array unsorted;

        let m1 be merge_sort(temp_size);
        invoke m1 with sort(myid,temp_size,temp1);

        let m2 be merge_sort(temp_size);
        invoke m2 with sort(myid,temp_size,temp2);

 (*   Wait to receive the two sorted parts.
      Store the two sorted parts in arrays temp1 and temp2 respectively *)
      wait for (message-type=reply);
      temp1:= sorted;
      wait for (message-type=reply);
      temp2:= sorted;

 (*   Call routine merge to merge the two sorted arrays temp1 and
      temp2 into array merged    *)
        merge(temp_size,temp1,temp2, merged);

 (*   Send the sorted array  merged to the creator process  *)
        invoke sender_id with reply(temp_size*2,merged);
    end;
 end-entity;
```

Figure 3: CMAY Implementation Of A Recursive Mergesort Algorithm

4.3. Inventory Management System

We illustrate the use of the language fragment described in this paper in developing distributed applications by considering the design of a simple inventory management system (IMS). The IMS is used to maintain satisfactory inventory levels of various items consumed by an establishment. The IMS uses the following three files:

1. *transaction* file: contains the daily summary of transactions processed for each item.
2. *inventory* file: contains the current inventory levels and the reorder points for each item; the reorder point refers to the minimum level of inventory that needs to be maintained for this item.
3. *supplier* file: contains the name and addresses of the supplier for each item.

The IMS is used to process the following commands:

- inquiry(item$_i$): determine the current inventory level for the ith item.
- consume(item$_i$, qty): qty units of the ith item have been consumed; if the inventory falls below the reorder point, fresh stocks have to be ordered from the supplier.
- receive(item$_i$, qty): qty units of the ith item have been received from a supplier.

A schematic diagram of the system is presented in Figure 4. The IMS receives commands from terminal-processes. Depending on the command, the IMS reads/updates the appropriate file(s) and sends an acknowledgement to the requesting process. We consider a distributed implementation of the above system. We define three file servers *inventory, transaction*, and *supplier* for each of the three files respectively. In order to minimize the response time for a command, each file server is assigned to a separate machine on the network. On receiving a command from a terminal-process, the IMS process forwards the command and the identity of the requesting process to the appropriate file server(s) for processing. Due to the non-bocking **send**, the IMS is immediately ready to receive the next command. When a command has been processed by a file server, it sends the acknowledgement directly to the requesting terminal-process. In this manner, multiple files can be accessed simultaneously in the system, and the IMS process is not blocked while a given command is being processed by a file server. Figure 5 presents a model of the implementation and illustrates the various messages exchanged by the processes in the system. In figures 6 and 7, we give the code for entity *IMS* and entity *inventory* which are used to represent the IMS process and the inventory file server respectively.

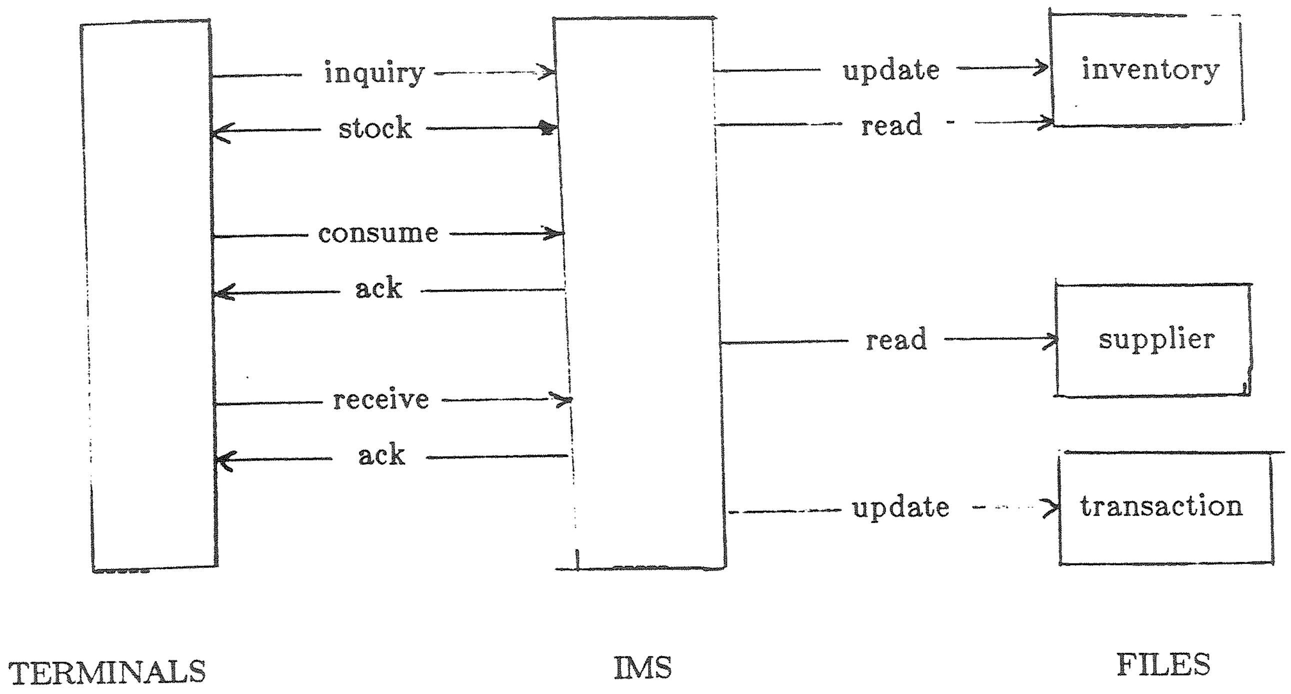

Figure 4: A Schematic Of The IMS

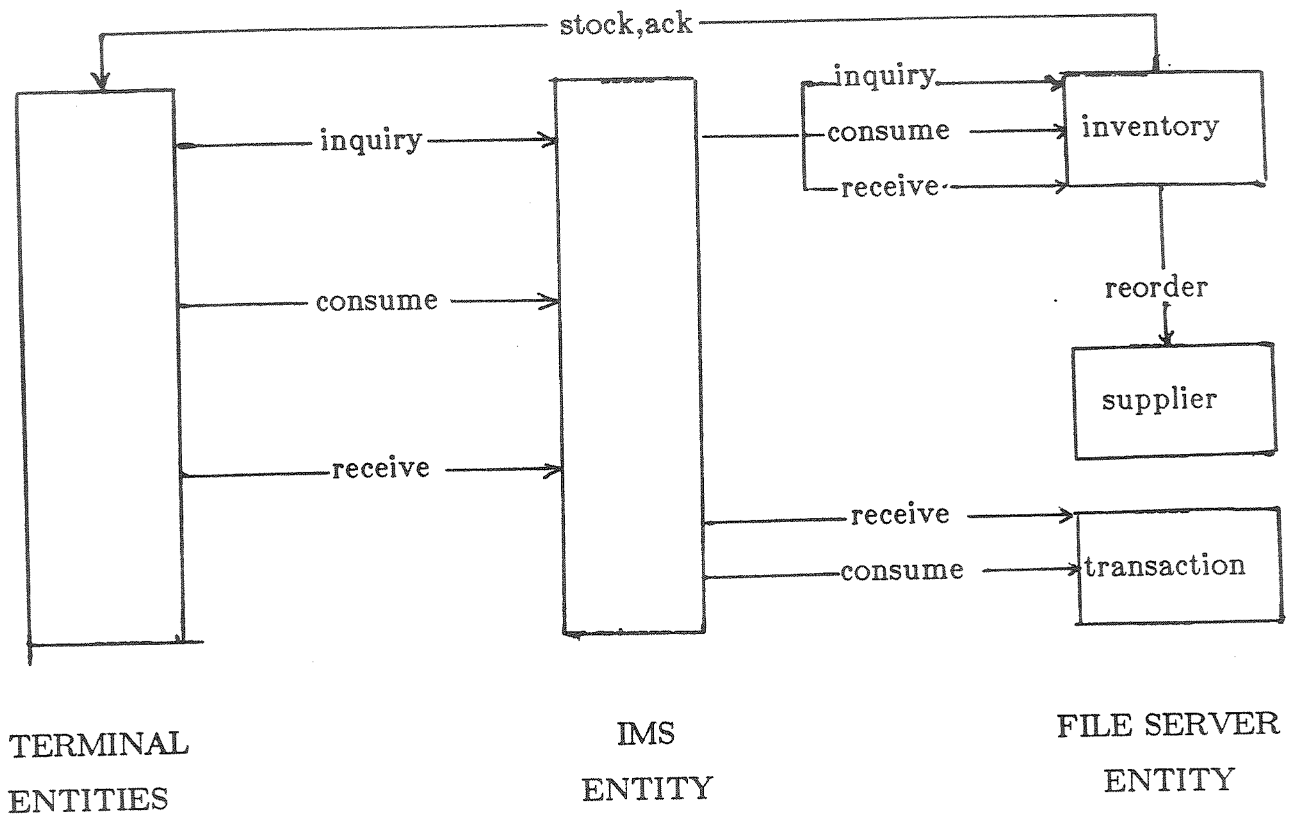

Figure 5: A LOGICAL MODEL OF THE IMS

```
entity ims(inventory, transcation: entity-identifier);

message inquiry(term-id: entity-identifier,itemno:integer);
message consume(term-id: entity-identifier,itemno, qty:integer);
message receive(term-id: entity-identifier,itemno, qty:integer);


while true do

begin
   wait till the next message is received;

   if (message-type = inquiry) then
        invoke inventory with inquiry(term-id,itemno);

   else if  (message-type = consume) then
   begin
        invoke inventory with consume(term-id,itemno, qty);
        invoke transaction with consume(term-id,itemno, qty);
   end

   else if  (message-type = receive) then
   begin
        invoke inventory with receive(term-id,itemno, qty);
        invoke transaction with receive(term-id,itemno, qty);
   end;
 end;
 end-entity;
```

Figure 6: CMAY code for entity IMS

```
entity inventory(supplier: entity-identifier);

message inquiry(term-id: entity-identifier,itemno:integer);
message consume(term-id: entity-identifier,itemno, qty:integer);
message receive(term-id: entity-identifier,itemno, qty:integer);


while true do

begin
   wait till the next message is received;

   if (message-type = inquiry) then
   begin
     read inventory file record item_rec for item itemno;
     invoke term-id with stock(item_rec.quantity);
   end

   else if  (message-type = consume) then
      begin
         update inventory file record item_rec for item itemno;
         if (item_rec.quantity < item_rec.reorder-point)
                and (not item_rec.order-placed) then
         begin
            invoke supplier with reorder(itemno);
            item_rec.order-placed:= true;
            update inventory file record item_rec for item itemno;
         end;
         invoke term-id with ack;
      end

    else if  (message-type = receive) then
       begin
          update inventory file record item_rec for item itemno;
          invoke term-id with ack;
       end
end;
end-entity;
```

Figure 7: CMAY Code For Entity Inventory.

5. Distributed Program Execution

A distributed operating system kernel is incorporated in the user program to support distributed program execution on computer networks. The kernel facilities are provided as separate functional entities which are included in the program at the option of the user. This approach allows a user the flexibility of developing CMAY programs in a uniprocessor environment. The kernel provides three major facilities:

1. It supports the dynamic creation of and communication between processes on different processors.

2. It provides built-in facilities for distributed termination detection.

3. It collects appropriate statistical information on the relative utilization of the various processors on the network (the utilization statistics ignores all computing load other than the CMAY programs that may be running on the machines).

5.1. Remote Process Creation

An important issue in process creation is that of the allocation of processes amongst available processors. In one approach, the distributed program configuration is controlled entirely by the system (language or operating system) which allocates processes and may also cause processes to migrate amongst processors in order to optimize system utilization parameters. This approach has been adopted in the design of a number of programming languages. In the other approach, the programmer creates the processes on specific processors and maintains complete control over the distribution of the program. The first approach has the advantage of making the mapping of the program onto the network completely transparent to the user. However, an efficient allocation would require the presence of a sophisticated distributed operating system, a requirement that contradicts our goals. In CMAY, the programmer has the option of controlling the allocation of the different parts of the distributed program among the available computers or of giving up control to the system. If all entities are created on one processor, we have a centralized implementation of CMAY. Regularly updated statistical information on the utilization of each computer (e.g. processor utilization and number of currently active entities) is provided to aid the user or the system in dynamically allocating new entities in an efficient manner amongst the various processors. For instance, an entity may be created on the processor with the lowest utilization or on the processor with the smallest number of executing entities.

We introduce a scalar variable type called **processor-identifier** . Variables of type **processor-identifier** are used exclusively to store the identifiers of the processors on the network. In order to create an instance of an entity on a remote processor, the **let** statement introduced earlier may be modified as follows:

let *e1* **be** *entity-type-name*(actual parameter list) **at** *p1*;

where *p1* is a variable of type **processor-identifier** . The entity will execute on the processor identified by variable *p1*. If the **at** clause is omitted, the entity executes on the local processor.

5.2. Program Initiation

Typically, a CMAY program runs on a collection of autonomous computers linked by a communication network. Each computer in the network is statically assigned an unique identifier which is stored in a variable of type **processor-identifier** declared in the kernel. A processor's unique identifier is available to the entities executing on that processor in a compiler defined variable called **my_processorid**. Every CMAY program must declare the processors from the network, on which the program may (potentially) execute. In addition the program must also specify one of these processors as the **initiator** processor. The CMAY program is initiated by executing entity **main** on the **initiator** processor. Subsequently, other entities may be created dynamically on the various processors as explained previously. A statement type called the **mayprogram** statement is provided to specify the **initiator** processor as well as the other processors on which the program may execute. This statement has the following form:

mayprogram <program-name> (**processors** $<p_1>$,$<p_2>$...$<p_n>$;
initiator $<p_j>$)

where $<p_i>$, i:1..n, are constants of type **processor-identifier** which specify the processors on which the program may execute; and $<p_j>$, is the **initiator** processor. If this statement is omitted in a program, the program is executed on a single processor.

5.3. Program Termination

A distributed program is said to have terminated if the following two conditions are satisfied [chandyterm]

1. All messages that were sent have been received (messages stored in the message buffer of an entity are presumed to have been 'received').
2. No entity will send another message in the future (all entities are waiting indefinitely).

If all entities (including **main**) in a CMAY program have been terminated then the above conditions are trivially satisfied. This would automatically cause the CMAY program to terminate. In addition, CMAY provides a **terminate** statement to unconditionally terminate the execution of the program on all the processors. This statement is analogous to a **stop** statement in a sequential program. However, there exist a variety of distributed applications, in which program termination can be detected only in a distributed manner based on the two conditions specified above being satisfied. For this purpose the language provides a built-in, user-transparent facility to detect program termination in a distributed manner. This facility is provided as a library entity, and is activated by the simple mechanism of including this entity in a user program. Program termination is detected by using the distributed termination detection algorithm suggested in [chandyterm].

6. Conclusion

In this paper, we suggest that general purpose distributed programming languages may be designed by incorporating two basic notions - **entities** and **messages** in any sequential programming language. A key feature is that a programmer familiar with a sequential programming language - say FORTRAN - can use all the facilities of the language including procedure libraries and needs to learn only three new primitives (**create**,**send**, and **receive**) to use distributed systems effectively. With message types defined suitably, it should be possible to execute distributed programs which consist of communicating entities written in different host-languages and running on different processors. Our distributed programming fragment was compared with the approaches adopted in the design of CSP, DP, and Ada. A comparison of the four languages is presented in tabular form in table I. The process representation and message passing features introduced in this paper have also been implemented to develop a message based simulation language [5].

The ideas discussed in this paper were implemented to develop CMAY - a FORTRAN based distributed programming language. CMAY supports a library facility whereby user programs may include routines and entities from the CMAY library. A versatile trace facility is also provided. This facility may be used to trace messages exchanged between entities. The tracing may be performed at different levels depending on the detail to which the state information of an entity is desired. Tracing may be initiated dynamically, for instance, on encountering a condition corresponding to a possible error in the program.

CMAY has been implemented on a network of VAX 11/750s each running the 4.2 BSD Unix operating system. The IPC (Inter-Process Communication) facilities provided by the operating system were used to set up the communication channels between the various processors on the network. Although this makes the current implementation of the language unportable to systems not running 4.2 BSD Unix, the communication facilities are provided as subroutine calls and can conveniently be replaced by whatever network communication facilities are provided by other operating systems.

Acknowledgements

We are grateful to Professor Mike Molloy and to Ted Briggs for their guidance in implementing the language on the UTCSRES network.

HEADING	CSP	DP	ADA	CMAY
System Configuration	distributed or centralized	distributed	distributed or centralized	distributed centralized
Process Representation	parallel processes	distributed processes	tasks	entities
Process Creation	dynamic	static	dynamic	dynamic
Process Termination	automatic	----	explicit, uncontrolled	explicit, autonomous
Process Variables	restricted	No	Yes	Yes
Communication Primitive	rendezvous	remote procedure call	remote proc. call via rendezvous	buffered message passing
Synchronization Primitive	rendezvous and input guards	guarded region conditional delay	rendezvous and input guards	conditional receives
Non-determinism	guarded command	guarded region	guarded command	programmer responsibili
Recursion	No	No	Yes	Yes
Termination Detection	programmer responsibility	programmer responsibility	programmer responsibility	built-in facility

Table 1: Comparison of CSP, DP, Ada and CMAY

References

[1] *Reference Manual for the Ada Programming Language*
United States Department Of Defense, 1982.

[2] Ambler, A., Good, D.I., and Burger, W.F.
Report on the Language Gypsy.
Technical Report ICSCA-CMP-1, Dept. of Computer Science, University of Texas at Austin, August, 1976.

[3] Andrews, G.R.
Synchronizing Resources.
ACM Trans. Program. Lang. Syst. 3(4):405-430, October, 1981.

[4] Andrews, G.R. and Schneider F.B.
Concepts And Notations For Concurrent Programming.
Computing Surveys 15(1):3-43, March, 1983.

[5] Bagrodia, R., Chandy, K.M., and Misra, J.
A Message Based Approach to Discrete Event Simulation.
Report TR LCS - 8403, Dept. of Computer Science, University of Texas at Austin, May, 1984.

[6] Brinch Hansen, P.
The Programming Language Concurrent Pascal.
IEEE Trans. on Software Engg. 1(2):199-207, June, 1975.

[7] Cook, R.P.
*MOD - A Language For Distributed Programming.
IEEE Trans. on Software Engg. SE-6(6):563-571, November, 1980.

[8] Dahl O.J., Myhrhaug B. and Nygaard K.
Simula 67 Common Base Language.
Norwegian Computing Centre, Oslo, 1970.

[9] Dijkstra, E.W.
Guarded commands, nondeterminacy, and formal derivation of programs.
Communications of the ACM 18(8):453-457, August, 1975.

[10] Feldman, J.A.
High Level Programming For Distributed Computing.
Communications of the ACM 22(6):353-367, June, 1979.

[11] Gehani N.
Ada: An Advanced Introduction.
Prentice-Hall, 1983.

[12] Geschke, C.M., Morris Jr., J.H., Satterthwaite, E.H.
Early Experience With Mesa.
Communications of the ACM 20(8):540-553, August, 1977.

[13] Hansen P.B.
Distributed Processes:A Concurrent Programming Concept.
CACM 21(11):934-941, November, 1978.

[14] Hoare C.A.R.
Communicating Sequential Processes.
CACM 21(8):666-677, August, 1978.

[15] Kieburtz, R.B., and Silberschatz, A.
Comments on 'communicating sequential processes.
ACM Trans. Program. Lang. Syst. 1(2):218-225, October, 1979.

[16] Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C.
Abstraction Mechanisms in CLU.
Communications of the ACM 20(8):564-576, August, 1977.

[17] Liskov, B.
Primitives For Distributed Computing.
In *Proceedings of the seventh Symposium on Operating Systems Principles.*
ACM, December, 1979.

[18] Parr, F.N. and Strom, R.E.
NIL: A High-level Language For Distributed Systems Programming.
IBM Sytems Journal 22(2):111-127, 1983.

[19] Scherr,A.L.
Distributed Data Processing.
IBM Sytems Journal 17(4):324-364, 1978.

[20] Wirth, N.
Modula:A Language for Modular Multi-programming.
Software - Practice and Experience (7):3-35, 1977.