Comput. Lang. Vol. 16, No. 2, pp. 129–146, 1991
Printed in Great Britain. All rights reserved

DISTRIBUTED PROGRAMMING WITH SHARED DATA

HENRI E. BAL and ANDREW S. TANENBAUM
Department of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands

(*Received* 22 *January* 1990)

Abstract—Until recently, at least one thing was clear about parallel programming: shared-memory machines were programmed in a language based on shared variables and distributed machines were programmed using message passing. Recent research on distributed systems and their languages, however, has led to new methodologies that blur this simple distinction. Operating system primitives and languages for programming distributed systems have been proposed that support shared data without the presence of physical shared memory. We will look at the reasons for this evolution, the resemblances and differences among these new proposals, and the key issues in their design and implementation. It turns out that many implementations are based on replication of data. We take this idea one step further, and discuss how automatic replication can be used as a basis for a new model with similar semantics as shared variables. Finally, we discuss a new language, Orca, based on this model.

Programming languages   Distributed systems

1. INTRODUCTION

Parallel computers of the MIMD (Multiple Instruction Multiple Data) class are traditionally divided into two broad subcategories: tightly coupled and loosely coupled systems. In a tightly coupled system at least part of the primary memory is *shared*. All processors have direct access to this shared memory, in one machine instruction. In a loosely coupled (*distributed*) system, processors only have access to their own local memories; processors can communicate by sending messages over a communication channel, such as a point-to-point link or a local area network [1]. Tightly coupled systems have the significant advantage of fast communication through shared memory. Distributed systems, on the other hand, are much easier to build, especially if a large number of processors is required.

Initially, programming language and operating system designers strictly followed the above classification, resulting in two parallel programming paradigms: shared variables (for tightly coupled systems) and message passing (for distributed systems) [2]. Some languages and operating systems for uniprocessors or shared-memory multiprocessors support processes that communicate through message passing (e.g. MINIX [3]). More recently, the dual approach, applying the shared variable paradigm to distributed systems, has become a popular research topic. At first sight, this approach may seem to be against the grain, as the message passing paradigm much better matches the primitives provided by the distributed hardware. For sequential languages, however, we have become quite used to programming paradigms like functional, logic, and object-oriented programming, which do not directly reflect the underlying architecture either.

The purpose of this paper is twofold. First, we will classify existing techniques for providing conceptual shared memory by looking at their most important similarities and differences. Analysis of the semantics shows that many proposals are not strictly like message passing nor like shared variables, but somewhere in between. In other words, there exists a *spectrum* of communication mechanisms, of which shared variables and message passing are the two extremes. Most primitives towards the shared-variable end of the spectrum use *replication* of data for an efficient distributed implementation.

The second purpose of the paper is to discuss a new model providing conceptual shared memory and a new programming language, Orca, based on this model. Unlike most other languages for distributed programming, Orca is intended for distributed application programming rather than systems programming. A major issue in its design was to keep the language as simple as possible and to exclude features that are only useful for systems programming.

Some theoretical work has been done in the area of simulating shared memory in distributed systems [4, 5]. In these studies, a distributed system is usually regarded as a (possibly incomplete) graph, where nodes represent processors and arcs represent communication channels. These studies typically aim at minimizing the number of messages needed to read or write a simulated shared variable. In this paper, we are more interested in real-life distributed computing systems (like those advocated by V [6] and Amoeba [7]). In such systems, all processes can directly communicate with each other, although communication between processes on different processors is expensive. These systems frequently support additional communication primitives, like multicast and broadcast.

2. SHARED VARIABLES AND MESSAGE PASSING

Communication through shared variables probably is the oldest paradigm in parallel programming. Many operating systems for uni-processors are structured as collections of processes, executing in quasi-parallel, and communicating through shared variables. Synchronizing access to shared data has been a research topic since the early sixties. Numerous programming languages exist that use shared variables.

The semantics of the model are fairly simple, except for what happens when two processes simultaneously try to write (or read and write) the same variable. The semantics may either define simple reads and writes to be indivisible (conflicting reads or writes are serialized) or may leave the effect of simultaneous writes undefined.

The basis for message passing as a programming language construct is Hoare's classic paper on CSP [8]. A message in CSP is sent from one process (the sender) to one other process (the receiver). The sender waits until the receiver has accepted the message (*synchronous* message passing).

Many variations of message passing have been proposed [2, 9, 10]. With *asynchronous* message passing, the sender continues immediately after sending the message. Remote procedure call and rendezvous are two-way interactions between two processes. Broadcast and multicast are interactions between one sender and many receivers [11]. Communications ports or mailboxes can be used to avoid explicit addressing of processes.

Below, we will describe the most important differences between the two extremes of our spectrum: shared variables and simple (synchronous and asynchronous) message passing. Some of the extensions to message passing mentioned above make the differences less profound.

- A message transfers information between two processes, which must both exist (be alive) when the interaction takes place. At least the sender must know the identity of the receiver. Data stored in a shared variable are accessible to any process. Processes interacting through shared variables need not even have overlapping lifetimes or know about each other's existence. They just have to know the address of the shared variable.
- An assignment to a shared variable has immediate effect. In contrast, there is a measurable delay between sending a message and its being received. For asynchronous message passing, for example, this has some ramifications for the order in which messages are received. Usually, the semantics are *order-preserving*: messages between a pair of processes are received in the same order they were sent. With more than two processes, the delay still has to be taken into account. Suppose Process P_1 sends a message X to P_2 and then to P_3. Upon receiving X, P_3 sends a message Y to P_2. There is no guarantee that P_2 will receive X before Y.
- Message passing intuitively is more *secure* [12] than sharing variables. Security means that one program module cannot effect the correctness of other modules (e.g. by a "wild store" through a bad pointer). The feasibility of a secure message passing language was demonstrated by NIL [13]. Shared variables can be changed by any process, so security is a bigger problem. One solution is to use *monitors*, which encapsulate data and serialize all operations on the data.

- A message exchanges information, but it also *synchronizes* processes. The receiver waits for a message to arrive; with synchronous message passing, the sender also waits for the receiver to be ready. With shared variables, two different types of synchronization are useful [9]. *Mutual exclusion* prevents simultaneous writes (or reads and writes) of the same variable; *condition synchronization* allows a process to wait for a certain condition to be true. Processes can synchronize through shared variables by using busy-waiting (polling), but this behavior is undesirable, as it wastes processor cycles. Better mechanisms are semaphores, eventcounts, and condition variables.

The message passing model has some additional implementation problems, as noted, for example, by Kai Li [14]. Passing a complex data structure to a remote process is difficult. Processes cannot easily be moved (migrated) to another processor, making efficient process management more complicated. The shared variable model does not suffer from these problems.

3. IN BETWEEN SHARED VARIABLES AND MESSAGE PASSING

The shared variable and message passing paradigms each have their own advantages and disadvantages. It should come as no surprise that language and operating system designers have looked at primitives that are somewhere in between these two extremes, and that share the advantages of both. In this section, we will discuss several such approaches.

In theory, a shared variable can simply be simulated on a distributed system by storing it on one processor and letting other processors read and write it with remote procedure calls. In most distributed systems, however, a remote procedure call is two to four orders of magnitude slower than reading local data. (Even Spector [15] reports an overhead of 150 microseconds for a certain class of remote references, despite a highly tuned, microcoded implementation.) This difference makes a straightforward simulation unattractive.

Most systems described in this section offer primitives that have some properties of shared variables and some of message passing. The semantics are somewhere in between shared variables and message passing. Often, the data are only accessible by some of the processes and only through some specific operations. These restrictions make the primitives more secure than regular shared variables and make an efficient implementation possible even if physical shared memory is absent.

We will discuss four key issues for every primitive:

- What are the *semantics* of the primitive?
- How are shared data *addressed*?
- How is access to shared data *synchronized*?
- How can the primitive be *implemented efficiently* without using physical shared memory?

We will first discuss proposals that are "close" to message passing; subsequent designs are increasingly similar to shared variables. The results are summarized in Fig. 1.

3.1. Communication ports

In CSP-like languages, interacting processes must explicitly name each other. For many applications (e.g. those based on the client/server model) this is inconvenient. A solution is to send messages indirectly through a *communication port* [16]. A port or mailbox is a variable where messages can be sent to or received from.

A port can be regarded as a shared *queue* data structure, with the following operations defined on it:

```
send(msg,q);  # Append message to end of queue.
msg := receive(q);  # Wait until queue not empty;
                    # get message from head of queue.
```

The latter operation also synchronizes processes. Ports can be addressed like normal variables. The implementation is fairly straightforward; a buffer is needed to store messages sent but not yet received.

technique	semantics	addressing	synchronization	implementation
Comm. ports	shared queues	variables	blocking receive()	straight message passing
Ada's shared variables	weird	variables	rendez-vous	replication, updates on rendez-vous
object model	shared objects	object-references	indivisible objects, blocking oper.	object migration, repl. read-only objects
problem-oriented shared memory	shared mem. with stale data	application-specific	through messages	replication, multicast
Agora shared memory	shared data struct. stale data	flat name space	pattern-directed	replication on reference
Tuple Space (Linda)	shared memory, no assignment	associatively	blocking read() and in()	compile-time analysis, replication, multicast
shared virtual memory	shared mem.	linear address space	semaphores, eventcounts, etc.	MMU, replication, multicast
shared logical variables	logical var. (single assignm.)	unification	suspend on unbound vars.	replication on reference

Fig. 1. Overview of conceptual shared memory techniques.

Although the semantics of ports are essentially those of asynchronous message passing, it is interesting to note that ports can be described as shared data structures with specialized access operations.

3.2. Ada's shared variables

Processes (tasks) in Ada® can communicate through the rendezvous mechanism or through shared variables. Shared variables in Ada are normal variables that happen to be visible to several tasks, as defined by the Ada scope rules. In an attempt to make the language implementable on memory-disjoint architectures, special rules for shared variables were introduced (Section 9.11 of the language reference manual [17]). Between synchronization points (i.e. normal rendezvous communication), two tasks sharing a variable cannot make any assumptions about the order in which the other task performs operations on the variable. In essence, this rule permits a distributed implementation to use copies (replicas) of shared variables and to update these copies only on rendezvous.

The semantics of Ada's shared variables are quite different from normal shared variables, as updates do not have immediate effect. Moreover, there are many deficiencies of shared variables in Ada, as discussed in detail by Shulman [18]. Introducing conceptual shared data this way does not seem like a major breakthrough in elegant language design, but it does illustrate the idea of replication.

3.3. The object model

Object-oriented languages are becoming increasingly popular, not only for writing sequential programs, but also for implementing parallel applications. Different languages have different definitions of the term "object", but in general an object encapsulates both *data* and *behavior*. Concurrent languages that are strongly influenced by the object-oriented programming paradigm include: ABCL/1 [19], Aeolus [20], ConcurrentSmalltalk [21], Emerald [22], Raddle [23], and Sloop [24].

An object in a concurrent object-based language can be considered as shared data that are accessible only through a set of *operations* defined by the object. These operations are invoked by sending a message to the object. Operation invocation can either be asynchronous (the invoker continues immediately after sending the message) or synchronous (the invoker waits until the operation has been completed).

Objects are usually addressed by an object *reference* (returned upon creation of the object) or by a global object name. To synchronize access to (shared) objects, several approaches are conceivable. Emerald uses a monitor-like construct to synchronize multiple operation invocations to the same object. Sloop supports *indivisible* objects, for which only one operation invocation at a time is allowed to execute. For condition synchronization, Sloop allows operations to suspend on a boolean expression, causing the invoking process to block until the expression is "true".

A key issue in a distributed implementation of objects is to locate objects on those processors that use them most frequently. Both Emerald and Sloop allow (but do not enforce) the programmer to control the locations of objects; these locations can be changed dynamically (object migration). Alternatively, the placement of objects can be left entirely to the run time system. For this purpose, Sloop dynamically maintains statistical information about the program's communications patterns. Some language implementations also support replication of immutable (read-only) objects.

The object model already presents the illusion of shared data. Access to the shared data is restricted to some well-defined operations, making the model more secure than the simple shared variable model. Synchronization can easily be integrated with the operations. In Sloop, operations are invoked by asynchronous messages, so the semantics of Sloop still resemble message passing. Emerald uses synchronous operation invocations, resulting in a model closer to shared variables.

3.4. Problem-oriented shared memory

Cheriton [25] has proposed a kind of shared memory that can be tailored to a specific application, the so-called problem-oriented shared memory. The shared memory can be regarded as a distributed system service, implemented on multiple processors. Data are stored (replicated) on one or more of these processors, and may also be cached on client workstations.

The semantics of the problem-oriented shared memory are tuned to the needs of the application using it. In general, the semantics are more relaxed than those of shared variables. In particular, inconsistent copies of the same data are allowed to coexist temporarily, so a "read" operation does not necessarily return the value stored by the most recent "write". There are several different approaches to deal with these *stale* data, for example to let the applications programmer worry about it, or to let the shared memory guarantee a certain degree of accurateness (e.g. a shared variable containing the "time of the day" can be kept accurate within, say, 5 seconds).

The problem-oriented shared memory is addressed also in an application specific way. Addresses are broadcast to the server processors. There is no special provision to synchronize processes (processes can synchronize using message passing).

The implementation significantly benefits from the relaxed semantics. Most important, it does not have to use complicated schemes to atomically update all copies of the same data.

3.5. The Agora shared memory

The Agora shared memory allows processes written in different languages and executing on different types of machines to communicate [26]. It has been implemented on closely coupled as well as loosely coupled architectures, using the Mach operating system.

The memory contains shared data structures, accessible through an (extendible) set of standard functions. These functions are available (e.g. as library routines) in all languages supported by the system. A shared data structure is organized as a set of immutable data elements, accessed indirectly through (mutable) *maps*. A map maps an index (integer or string) onto the address of a data element. To change an element of the set, a new element must be added and the map updated accordingly. Elements that are no longer accessible are automatically garbage collected.

Exclusive access to a data structure is provided by a standard function that applies a user function to a data structure. For condition synchronization, a pattern-directed mechanism is supported. For example, a process can wait until a certain element is added to a set.

The implementation is based on replication of data structures on reference. As in Cheriton's model, read operations may return stale data.

3.6. Tuple Space

The Tuple Space is a novel synchronization mechanism, designed by David Gelernter for his language Linda [27, 28]. The Tuple Space is a global memory containing *tuples*, which are similar to records in Pascal. For example, the tuple "["Miami", 305]" consists of a string field and an integer field. Tuple Space is manipulated by three atomic operations: **out** adds a tuple to Tuple Space, **read** reads an existing tuple, and **in** reads and deletes a tuple. Note that there is no operation to *change* an existing tuple. Instead, the tuple must first be removed from Tuple Space, and later be put back.

Unlike all other conceptual shared memory systems discussed in this paper, Tuple Space is addressed *associatively* (by contents). A tuple is denoted by supplying actual or formal parameters for every field. The tuple mentioned above can be read and removed, for example, by

```
    in("Miami", 305);
```

or by

```
    integer areacode;
    in("Miami", var areacode);
```

In the latter case, the formal parameter *areacode* is assigned the value 305.

Both **read** and **in** block until a matching tuple exists in Tuple Space. If two processes simultaneously try to remove (**in**) the same tuple, only one of them will succeed and the other one will block. As tuples have to be removed before being changed, simultaneous updates are automatically synchronized.

Although the semantics of Tuple Space are significantly different from shared variables (e.g. it lacks assignment), the Tuple Space clearly gives the illusion of a shared memory. The Tuple Space has been implemented on machines with shared memory (Encore Multimax, Sequent Balance) as well as on memory-disjoint machines (iPSC hypercube, S/Net, Ethernet based network of Micro-Vaxes). A distributed implementation can benefit from the availability of multicast [29]. As associative addressing is potentially expensive, several compile-time optimizations have been devised to make it reasonably efficient [30].

3.7. Shared virtual memory

Kai Li has extended the concept of *virtual memory* to distributed systems, resulting in a *shared virtual memory* [14]. This memory is accessible by all processes and is addressed like traditional virtual memory. Li's system guarantees memory *coherence*: the value returned by a "read" always is the value stored by the last "write".

The address space is partitioned into a number of fixed-size *pages*. At any point in time, several processors may have a *read-only* copy of the same page; alternatively, a single processor may have a *read-and-write* copy.

If a process tries to write on a certain page while its processor does not have a read-and-write copy of it, a "write page-fault" occurs. The fault-handling routine tells other processors to *invalidate* their copies, fetches a copy of the page (if it did not have one yet), sets the protection mode to read-and-write, and resumes the faulting instruction.

If a process wants to read a page, but it does not have a copy of the page, a "read page-fault" occurs. If any processor has a read-and-write copy of the page, this processor is instructed to change the protection to read-only. A copy of the page is fetched and the faulting instruction is resumed.

Shared Virtual Memory is addressed like normal virtual memory. An implementation may support several synchronization mechanisms, such as semaphores, eventcounts, and monitors.

The shared virtual memory can be used to simulate true shared variables, with exactly the right semantics. The implementation uses the hardware Memory Management Unit and can benefit from the availability of multicast (e.g. to invalidate all copies of a page). Several strategies exist to deal with the problem of multiple simultaneous writes and to administrate which processors contain copies of a page [14]. The entire scheme will perform very poorly if processes on many different processors repeatedly write on the same page. Migrating these processes to the same processor is one possible cure to this problem.

3.8. Shared logical variables

Most concurrent logic programming languages (PARLOG [31, 32], Concurrent Prolog [33, 34], Flat Concurrent Prolog) use shared logical variables as communication channels. Shared logical variables have the *single-assignment* property: once they are bound to a value (or to another variable) they cannot be changed. (In "sequential" logic languages, variables may receive another value after backtracking; most concurrent logic languages eliminate backtracking, however.)

Single-assignment is not a severe restriction, because a logical variable can be bound to a structure containing one or more other, unbound variables, which can be used for future communication. In fact, many communication patterns can be expressed using shared logical variables [33].

Synchronization in concurrent logic languages resembles data-flow synchronization: processes can (implicitly) wait for a variable to be bound. Shared logical variables provide a clean semantic model, resembling normal logic variables. Addressing also is the same for both types of variables (i.e. through unification).

The single-assignment property allows the model to be implemented with reasonable efficiency on a distributed system. An implementation of Flat Concurrent Prolog on a Hypercube is described in [35]. If a process tries to read a logical variable stored on a remote processor, the remote processor adds the process to a list associated with the variable. As soon as the variable gets bound (if it was not already), its value is sent to all processes on the list. These processes will keep the value for future reference. In this way, variables are automatically replicated on reference.

3.9. Discussion

Figure 1 summarizes the most important properties of the techniques we discussed. The systems differ widely in their semantics and addressing and synchronization mechanisms. A key issue in the implementation is *replication*. Multicast is frequently used to speed up the implementation.

Replication of data has already been used for a long time in distributed databases to increase the availability of data in the presence of processor failures. Replication introduces a severe problem: the possibility of having inconsistent copies of the same logical data. For databases, several solutions exist [36]. Typically, multiple copies of the same data are accessed when reading or writing data.

The techniques discussed in this section use replication to decrease the *access time* to shared data, rather than to increase availability. Therefore, it is unattractive to consult several processors on every access to the data. Instead, just the local copy should suffice for as many accesses as possible. With this restriction, different solutions must be found to deal with the consistency problem.

Figure 1 shows three different ways of dealing with inconsistency. Ada, the problem-oriented shared memory, and the Agora shared memory relax the semantics of the shared memory. The latter two systems allow "read" operations to return stale data. Higher level protocols must be used by the programmer to solve inconsistency problems. Ada requires copies to be updated only on rendezvous.

The second approach (used for objects, Tuple Space, and shared logical variables) is to replicate only *immutable* data (data that cannot be changed). This significantly reduces the complexity of the problem, but it may also introduce new problems. The approach is most effective in languages using single-assignment. Such languages, however, will need a complicated distributed garbage collection algorithm to get rid of unaccessible data. In Linda, tuples are immutable objects. A tuple can conceptually be changed by first taking it out of Tuple Space, storing it in normal (local) variables. After changing these local data, they can be put back in a new tuple. As tuples are accessed by contents, it makes little difference that the old tuple has been replaced by a new one, instead of being modified while in Tuple Space. As a major advantage of doing the modification outside Tuple Space, updates by different processes are automatically synchronized. On the other hand, a small modification to a large tuple (like setting a bit in a 100 K bitvector) will be expensive, as the tuple has to be copied twice.

The third approach to the consistency problem is exemplified by the shared virtual memory: use protocols that guarantee memory coherence. Before changing a page, all copies of the page are invalidated, so subsequent reads will never return stale data. Great care must be taken in the implementation, however, to avoid thrashing. For badly behaving programs, the system may easily spend most of its time moving and invalidating pages.

4. THE SHARED DATA-OBJECT MODEL

We have developed a new conceptual shared-memory model, called the *shared data-object* model [37]. In this model, shared data are encapsulated in passive objects. The data contained by the object are only accessible through a set of *operations* defined by the object's type. Objects are instances of *abstract data types*. Unlike, say, Emerald and Sloop, we do not consider objects to be active entities; neither do we consider all entities in the system to be objects.

Parallel activity originates from the dynamic creation of multiple sequential (single-threaded) processes. When a process spawns a child process, it can pass any of its objects as **shared** parameters to the child. The children can pass the object to *their* children, and so on. In this way, the object gets distributed among some of the descendants of the process that declared the object. All these processes *share* the object and can perform the same set of operations on it, as defined by the object's type. Changes to the object made by one process are visible to other processes, so a shared data-object is a communication channel between processes. This mechanism is similar to call-by-sharing in CLU [38].

The shared data-object model has many advantages over regular shared variables. Access to shared data is only allowed through operations defined by an abstract data type. All these operations are *indivisible*. Simultaneous operation invocations of the same object have the effect as if they were executed one by one, so access to shared data is automatically synchronized. Blocking operations are used for condition synchronization of processes, as will be explained later.

4.1. Implementing shared data-objects in a distributed system

Shared data-objects are a simple, secure mechanism for sharing data and synchronizing access to the data. The design cannot be judged, however, without also considering the implementation. We have worked on two distributed implementations of the model. One implementation [39] replicates objects on all processors and updates the copies through a fast, reliable broadcast protocol [40]. In this paper, however, we will study strategies based on *selective replication* and *migration* of objects, under full control of the run time system.

The compiler distinguishes between two kinds of operations:

- A "read" operation does not modify the object; it is performed on a local copy, if one exists.
- A "write" operation may read and write the object's data; it affects all copies.

For sake of clarity, we will use a simplified view of our model to describe its implementation. In particular, we will assume an object to contain a single integer and we will consider only two operations:

```
operation read(x: object): integer;
    # return current value
operation write(x: object; val: integer);
    # store new value
```

The run time system dynamically keeps track of how many times processors perform remote read and write operations on each object. If a processor frequently reads a remote object, it is profitable for it to maintain a local copy of the object. The execution time overhead of maintaining the statistics is negligible compared with the time needed to do remote references. The space overhead is not a real concern either, considering the current state of memory technology.

A major issue in implementing replication is how to propagate changes made to the data. Two approaches are possible: invalidating all-but-one copies of the data, or updating all copies [41]. Kai Li's Shared Virtual Memory uses invalidation. In our model, invalidation is indeed feasible, but it has some disadvantages. First, if an object is big (e.g. a 100 K bitvector) it is wasteful to invalidate its copies, especially if an operation changes only a small part (e.g. 1 bit). In this case, it is far more efficient to apply the operation to all copies, hence updating all copies. Second, if an object is small (e.g. an integer), sending the new value is probably just as expensive as sending an invalidation message. Although update algorithms are more complicated than invalidation algorithms, we think it is useful to study them.

A related issue is how to synchronize simultaneous operation invocations that try to modify the same object. To serialize such invocations we appoint one replica of the object as *primary copy*, direct all "write" operations to this primary copy, and then propagate them to the secondary copies. An alternative approach would be to treat all copies as equals and use a distributed locking protocol to provide mutual exclusion. The primary copy method, however, allows one important optimization: the primary copy can be migrated to the processor that most frequently changes the object, making updates more efficient. The statistical information described above is also used for this purpose.

4.2. Dealing with inconsistency

The presence of multiple copies of the same data introduces the consistency problem discussed in Section 3.9. As we do not want to clutter up the semantics of our model, the implementation should adequately solve this problem. In the following sections we will describe such implementations for different kinds of distributed architectures.

To deal with the consistency problem, we will first need a deeper understanding of the problem itself. Suppose we implement our model as follows. To update an object X, its primary copy is locked and a message containing the new value of X is sent to all processors containing a secondary copy. Such a processor updates its copy and then sends an acknowledgement back. When all messages have been acknowledged, the primary copy is updated and unlocked.

During the update protocol some processors have received the new value of X while others still use its old value. This is intuitively unappealing, but by itself is not the real problem. Far more important, not all processors will observe modifications to different objects in the same *order*. As a simple example, consider the program in Fig. 2.

P_2 tries to keep Y up-to-date with X; P_3 verifies that X is greater than or equal to Y. Clearly the latter condition should always be true. Now suppose X and Y are replicated as shown in Fig. 3. P_1 contains the primary copy of X, P_2 and P_3 have secondary copies. P_2 has the primary copy of Y, P_3 has a secondary copy. The following sequence of events may happen:

(1) X is incremented and becomes 1; P_1 sends an update message to P_2 and P_3.
(2) P_2 receives the update message, assigns 1 to the variable Y and sends an update message of Y to P_3.
(3) P_3 receives the update message from P_2, puts the value 1 in its copy of Y, and is surprised to see that Y is now greater than X (which still contains 0).
(4) P_3 receives the update message from P_1, and stores the value 1 in its copy of X.

P_3 observes the changes to X and Y in the wrong order. The problem is caused by the arbitrary amount of time that messages may take to travel from the source to the destination and by the inability to transfer information simultaneously from one source to many destinations. Such an implementation basically provides message passing semantics disguised in shared variable syntax.

The solution to the consistency problem depends very much on the architecture of the underlying distributed system. We will discuss solutions for three different classes of architectures: systems supporting point-to-point messages, reliable multicast, and unreliable multicast respectively.

4.3. Implementation with point-to-point messages

One model of a distributed system is a collection of processors that communicate by sending point-to-point messages to each other. A communication path between any two processors is

```
X,Y: shared object;    # initially 0

Process P1:
    for i := 1 to ∞ do
        write(X, i);

Process P2:
    repeat
        y := read(Y); x := read(X);
        if x > y then
            write(Y, x);

Process P3:
    repeat
        y := read(Y); x := read(X);
        assert x ≥ y;
```

Fig. 2. Example program.

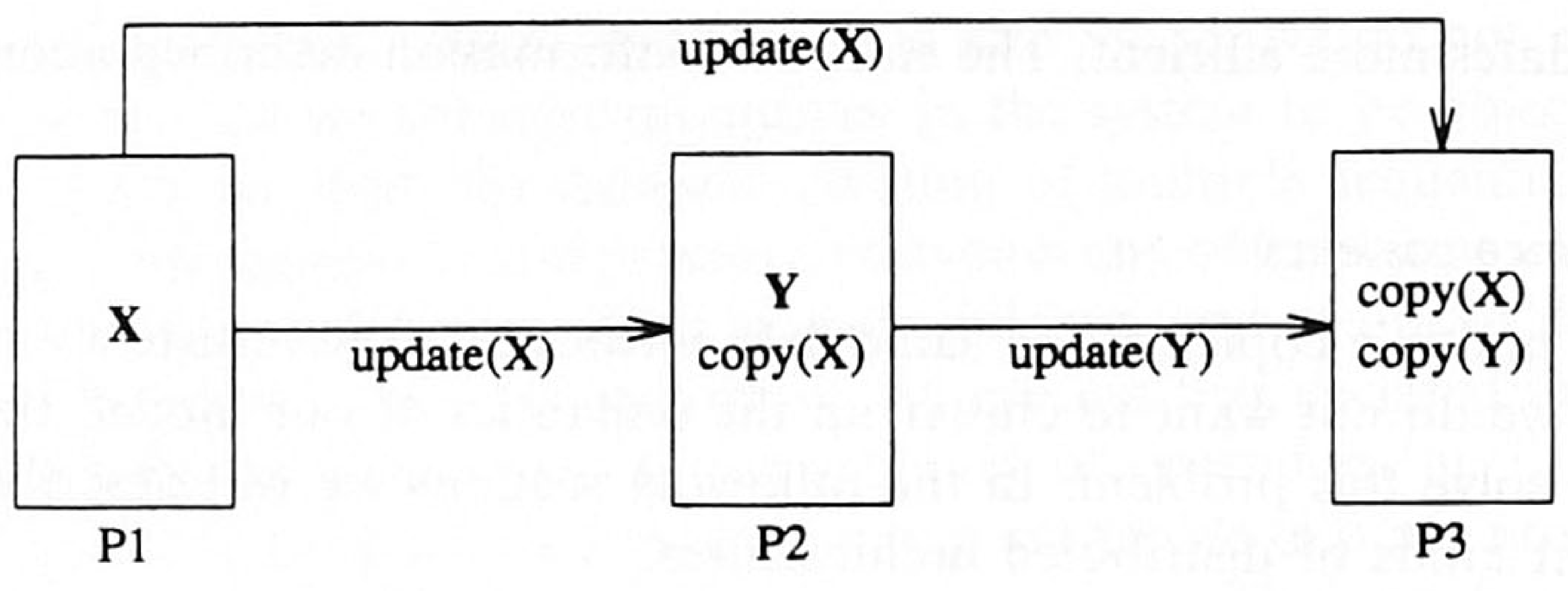

Fig. 3. Distribution of X and Y.

provided, either by the hardware or the software. Messages are delivered reliably, in the same order they were sent.

To implement consistent updating of objects in such a system, we use a *2-phase update* protocol. During the first phase, the primary copy is updated and locked, and an update message is sent to all processors containing a secondary copy. Unlike in the incorrect protocol outlined above, all secondary copies are locked (and remain locked) before being updated. A user process that tries to read a locked copy blocks until the lock is released (during the second phase). When all update messages have been acknowledged (i.e. all copies are updated and locked), the second phase begins. The primary copy is unlocked and a message is sent to all processors containing a secondary copy, instructing them to unlock their copies.

To implement the protocol, we use one *manager process* for every processor. We assume the manager process and user processes on the same processor can share part of their address space. Objects (and replicas) are stored in this shared address space. Write operations on shared objects are directed to the manager of the processor containing the primary copy; user processes can directly *read* local copies, although they may temporarily block, as described above. Each manager process contains multiple threads of control. One thread communicates with remote managers; the remaining threads are created dynamically to handle write-operations. So multiple write-operations to different objects may be in progress simultaneously; write-operations to the same object are serialized, as discussed below.

Upon receiving a request from a (possibly remote) user process W to perform an operation "write(X, Val)", the manager of X creates a new thread of control to handle the request:

```
receive write-req (X, Val) from W →
  fork handle_write(X, Val, W);
```

The process "handle_write" is defined by the following algorithm:

```
process handle_write(X, Val, W);
begin
  set write-lock on X;
  store Val in X;
  let S = set of processors having a copy of X;
  # first phase
  forall P ∈ S do
    send update-and-lock (X, Val) to manager of P;
  for i := 1 to |S| do
    receive ack;
  # second phase
  forall P ∈ S do
    send unlock (X) to manager of P
  unlock X;
  send ack to W;
end;
```

The process issuing the write request waits until it receives an acknowledgement. A manager responds as follows to messages from remote managers:

```
receive update-and-lock (X, Val) from P →
    set write-lock on local copy of X;
    store Val in local copy of X;
    send ack to P;

receive unlock (X) →
    unlock local copy of X;
```

The 2-phase update protocol guarantees that no process uses the new value of an object while other processes are still using the old value. The new value is not used until the second phase. When the second phase begins, all copies contain the new value. Simultaneous write-operations on the same object are serialized by locking the primary copy. The next write-operation may start before all secondary copies are unlocked. New requests to *update-and-lock* a secondary copy are not serviced until the *unlock* message generated by the previous write has been handled (recall that point-to-point messages are received in the order they were sent).

Deadlock is prevented by using multi-threaded managers. Setting a write-lock on a primary copy may block one thread of a manager, but not an entire manager process. Locking a secondary copy always succeeds within a finite amount of time, provided that all read-operations terminate properly.

If an object has N secondary copies it takes 3*N messages to update all these copies. Reading a remote object takes 2 messages (one request, one reply). So, objects should only be replicated on processors that read the object at least twice before it is changed again. This can be determined (or estimated) dynamically, as discussed earlier. The protocol can easily be optimized into a 1-phase update protocol if an object has only one secondary copy.

For a small object (like an integer) that is frequently changed, it may be more efficient to invalidate copies when the object is changed and to replicate it on reference. The first read-operation after a write fetches the object from a remote processor and creates a local copy. Subsequent reads use this local copy, until it is invalidated by a modification to the object.

4.4. Implementation with reliable multicast messages

The 2-phase undate protocol adequately solves the consistency problem, although at the cost of some communication overhead. The semantics provided by the implementation closely resemble those of shared variables. If a write-operation completes at time T_w, read operations issued at time $T_r > T_w$ return the new value.

This strict *temporal* ordering, however, is not a necessary requirement for programming MIMD-like systems, in which processors are executing *asynchronously*. Processors in such systems are not synchronized by physical clocks. Each sequential process in an asynchronous system performs a sequence of computation steps: $C_0, C_1, \ldots, C_i, \ldots$.

Within a single process, these steps are *totally* ordered; C_n happens after C_m if and only if $n > m$. There is no total ordering between computation steps of different processes, however, as discussed by Lamport [42]. There is only a *partial* ordering, induced by explicit interactions (like sending a message or setting and testing shared variables).

This lack of total ordering allows an implementation of shared data-objects to slightly relax the semantics without affecting the underlying programming model. Suppose Process P_1 executes "write(X, Val)" and Process P_2 executes "read(X)". If there is no precedence relation between these two actions (i.e. neither one of them comes before the other in the partial ordering), the value read by P_2 may be either the old value of X or the new value. Even if, physically, the write is executed before the read, the read still can return the old value. The major difference with systems that allow read-operations to return arbitrary old (stale) data is that our model supports a consistent logical ordering of events, as defined implicitly in the program. Programs like those in Fig. 2 still execute as expected.

In a distributed system supporting only point-to-point messages, a consistent logical ordering is difficult to obtain, because messages sent to different destinations may arrive with arbitrary delays. Some distributed systems (e.g. broadcast-bus systems) give hardware support to send a single message to several destinations simultaneously. More precisely, we are interested in systems supporting reliable, indivisible multicasts, which have the following properties:

- A message is sent reliably from one source to a set of destinations.
- If two processors simultaneously multicast two messages (say m_1 and m_2), then either all destinations first receive m_1, or they all receive m_2 first.

With this multicast facility we can implement a simple update protocol. A "write(X, Val)" request is handled as follows by the manager of X:

```
receive write-req (X, Val) from W →
    set write-lock on X;
    store Val in X;
    let S = set of processors having a copy of X;
    multicast update (X, Val) to manager of every P ∈ S;
    unlock X;
    send write-ack (W) to manager of W;
```

After the *write-req* message has been handled, the acknowledgement is sent to the manager of W (the process that issued the request). The manager forwards it to W. This guarantees that the local copy of X on W's processor has been updated when W resumes execution. The manager can be a single-threaded process in this implementation. A manager handles all incoming *write-req*, *update*, and *write-ack* messages in the order they were sent. A manager containing a secondary copy responds as follows to messages from remote managers:

```
receive update (X, Val) →
    set write-lock on local copy of X;
    store Val in local copy of X;
    unlock local copy of X
receive write-ack (W) →
    send ack to W;
```

If a processor P reads a new value of an object X, an *update* message for X containing this value has also been sent to all other processors. Other processors may not have handled this message yet, but they certainly will do so before they handle any other messages. Any changes to shared objects initiated by P will be observed by other processors after accepting the new value of X. problems like those in Fig. 3 do not occur.

4.5. *Implementation with unreliable multicast messages*

A cost-effective way to build a distributed system is to connect a collection of micro-computers by a local area network. Such systems are easy to build and easy to extend. Many distributed operating systems have been designed with this model in mind [1].

Many LANs have hardware support for doing multicasts. An Ethernet, for example, physically sends a packet to every computer on the net, although usually only one of them reads the packet. There is no difference in transmission time between a multicast and a point-to-point message.

Unfortunately, multicasts in a LAN are not totally reliable. Occasionally, a network packet gets lost. Worse, one or more receivers may be out of buffer space when the packet arrives, so a packet may be delivered at only part of the destinations. In practice, multicast is highly reliable, although less than 100%. Unreliable multicast can be made reliable by adding an extra software protocol. Such a protocol has a high communication overhead and may result in multicasts that are not indivisible (as defined above). Instead, we have designed an implementation of shared data-objects that directly uses unreliable multicasts.

The basic algorithm is the same as that for reliable multicast. When a shared variable X is updated, some (or all) processors containing a secondary copy of X may fail to receive the update(X, Val) message. They will continue to use the old value of X. This is not disastrous, as long as the partial (logical) ordering of events is obeyed, as described above. To guarantee a consistent ordering, processors that failed to receive the update(X, Val) message must detect this failure before handling other update messages that logically should arrive after X's message.

This is realized as follows. Update messages are multicast to *all* processors participating in the program, not just to those processors containing a secondary copy. Every processor counts the number of update messages it sends. This number is called *mc-count*. Every processor records the *mc-count*s of all processors. These numbers are stored in a vector, called the *mc-vector* (initialized to all zeroes). For Processor P, *mc-vector* [P] always contains the correct value of P's *mc-count*; entries for other processors may be slightly out of date.

Whenever a processor multicasts a message, it sends its own *mc-vector* as part of the message. When a processor Q receives a multicast message from P, it increments the entry for P in its own *mc-vector* and then compares this vector with the *mc-vector* contained in the message. If an entry R in its own vector is less than the corresponding entry in the message, Q has missed a multicast message from Processor R. Q updates the entry for R in its own vector. As Q does not know which variable should have been updated by R's message, Q temporarily invalidates the local copies of all variables that have their primary copy on Processor R. It sends (reliable) point-to-point messages to the manager of R, asking for the current values of these variables. The reply messages from R also contain *mc-vector*s, and undergo the same procedure as for multicast messages. Until the copies are up-to-date again, local read operations of these copies block.

It is quite possible that lost update messages will remain undetected for a while. Suppose Processor Q misses an update message for a variable Y from Processor R and then receives an update message for X from Processor P. If P also missed R's message, the entry for R in the *mc-vector* of P and Q will agree (although they are both wrong) and the copy of X will be updated. However, as P contained the old value of Y when it updated X, the new value of X does not depend on the new value of Y, so it is consistent to update X.

If a process misses an update message for X, this failure will eventually be detected while handling subsequent messages. The assumption is that there will be subsequent messages. This assumption need not be true. For example, a process may set a shared flag-variable and wait for other processes to respond. If these other processes missed the flag's update message, the system may very well come to a grinding halt. To prevent this, dummy update messages are generated periodically, which do not update any copy, but just cause the *mc-vector*s to be checked.

The implementation outlined above has one considerable advantage: it takes a single message to update any number of copies, provided that the message is delivered at all destinations. There is a severe penalty on losing messages. As modern LANs are highly reliable, we expect this to happen infrequently. The implementation also has several disadvantages. Update messages are sent to every processor. Each message contains extra information (the *mc-vector*), which must be checked by all receiving processors. For a limited number of processors, say 32, we think this overhead is acceptable. The protocol can be integrated with the 2-phase update protocol described in Section 4.3. For example, objects that are replicated on only a few processors can be handled with the 2-phase update protocol while objects replicated on many processors are handled by the multicast protocol.

5. A LANGUAGE BASED ON SHARED DATA-OBJECTS

We have designed a simple, general purpose programming language called Orca, based on shared data-objects [37, 43]. Unlike most other parallel languages, Orca is intended for applications programming rather than for systems programming. Parallelism in Orca is based on dynamic creation of sequential processes. Processes communicate indirectly, through shared data-objects. An object can be shared by passing it as **shared** parameter to a newly created process, as discussed in Section 4.

5.1. Object type definitions

An object is an instance of an object type, which is essentially an abstract data type. An object type definition consists of a *specification* part and an *implementation* part. The specification part defines one or more operations on objects of the given type. For example, the declaration of an object type *IntObject* is shown in Fig. 4. The implementation part contains the data of the object,

```
object specification IntObject;
    operation value(): integer;   # current value
    operation assign(val: integer); # assign new value
    operation min(val: integer);
        # set value to minimum of current value
        # and "val"
    operation max(val: integer); # idem for maximum
end;
```

Fig. 4. Specification of object type IntObject.

code to initialize the data of new instances (objects) of the type, and code implementing the operations. The code implementing an operation on an object can access the object's internal data. The implementation of object type IntObject is shown in Fig. 5.

Objects can be created and operated on as follows:

```
myint: IntObject; # create object
...
myint$assign(83); # assign 83 to myint
...
x := myint$value(); # read value of myint
```

5.2. Synchronization

Access to the shared data is automatically synchronized. All operations defined in the specification part are indivisible. If two processes simultaneously invoke X$min(A) and X$min(B), the new value of X is the minimum of A, B, and the old value of X. On the other hand, a sequence of operations, such as

```
if A < X$value() then
    X$assign(A);
fi;
```

is not indivisible. This rule for defining which actions are indivisible and which are not is both easy to understand and flexible: single operations are indivisible, sequences of operations are not. The

```
object implementation IntObject;
    X: integer;  # local data

    operation value(): integer;
    begin
            return X;
    end

    operation assign(val: integer);
    begin
            X := val;
    end

    operation min(val: integer);
    begin
            if val < X then X := val; fi;
    end

    operation max(val: integer);
    begin
            if val > X then X := val; fi;
    end
begin
    X := 0;  # initialize internal data
end;
```

Fig. 5. Implementation of object type IntObject.

```
object implementation IntQueue;
    Q: list of integer;   # internal representation

    operation append(X: integer);
    begin    # append X to the queue
        add X to end of Q;
    end

    operation remove_head(): integer;
        R: integer;
    begin
        # wait until queue not empty,
        # then get head element
        guard Q not empty do   # blocking operation
            R := first element of Q;
            remove R from Q;
            return R;
        od;
    end
begin
    Q := empty;   # initialize IntQueue object
end;
```

Fig. 6. Outline of implementation of object type IntQueue.

set of operations can be tailored to the needs of a specific application by defining single operations to be as complex as necessary.

For condition synchronization, *blocking* operations can be defined. A blocking operation consists of one or more *guarded commands*:

```
operation name(parameters);
begin
   guard expr₁ do statements₁ od;
   guard expr₂ do statements₂ od;
   ...
   guard exprₙ do statementsₙ od;
end;
```

The expressions must be side-effect free boolean expressions. The operation initially blocks (suspends) until at least one of the guards evaluates to "true". Next, one true guard is selected nondeterministically, and its sequence of statements is executed. As an example, a type IntQueue with a blocking operation remove_head can be implemented as outlined in Fig. 6.

An invocation of remove_head suspends until the queue is not empty. If the queue is initially empty, the process waits until another process appends an element to the queue. If the queue contains only one element and several processes try to execute the statement simultaneously, only one process will succeed in calling remove_head. Other processes will suspend until more elements are appended to the queue.

5.3. An example program

We have used the object types discussed above to implement a distributed Traveling Salesman Problem (TSP)† algorithm, based on an earlier algorithm described in [44]. The algorithm uses one process to generate partial routes for the salesman (containing only part of the cities) and any number of worker processes to further expand (search) these partial solutions. A worker systematically generates all full routes that start with the given initial route, and checks if they are better (shorter) than the current best solution. Every time a worker finds a shorter full route, it updates a variable shared by all workers, containing the length of the shortest route so far. This variable is used to cut-off partial routes that are already longer than the current shortest route,

†The Traveling Salesman Problem is the problem of finding the shortest route for a salesman to visit each of a number of cities in his territory exactly once.

```
process worker(m: shared IntObject; q: shared TaskQueue);
  r: route;
begin
  do  # forever
      r := q$remove_head();
      tsp(r, m);
  od
end;

procedure tsp(r: route; minimum: shared IntObject);
begin
  if length(r) < minimum$value() then
    if "r" is a full solution then
      # r is a full route shorter than the current
      # best route, so update current best solution.
      minimum$min(length(r));
    else
      for all cities "c" not on route "r" do
          tsp(r||c, minimum); # search extended route
      od
    fi
  fi
end;
```

Fig. 7. Algorithm for TSP worker processes.

as these will never lead to an optimal solution. The basic algorithm for the worker processes is outlined in Fig. 7. (Figure 7 does not show how termination of the worker processes is dealt with; this requires an extension.) Conceptually, the distributed algorithm is as simple as the sequential TSP algorithm.

The shared variable is implemented as an object of type IntObject (see Fig. 4). As several workers may simultaneously try to decrease the value of this variable, it is updated using the indivisible min operation. The work-to-do is stored in an ordered task queue, the order being determined by one of the many heuristics that exist for the Traveling Salesman Problem, such as "nearest-city-first". The task queue is similar to the IntQueue data type of Fig. 6, except that the elements are "routes" rather than integers.

6. CONCLUSIONS

We have classified several communication primitives for distributed programming that support the shared variable paradigm without the presence of physical shared memory. Of the many programming languages for distributed systems that are around today [2], several recent ones present a computational model based on sharing data. More significant, novel programming styles are emerging. Examples include distributed data structures and the replicated worker model of Linda [28], and incomplete messages, difference streams, and the short-circuit technique of concurrent logic programming languages [33]. These techniques achieve a much higher level of abstraction than message passing languages, at the cost of some efficiency. More research is still needed to achieve the same level of efficiency for languages based on abstract shared data.

We also introduced a new model for distributed programming with abstract shared data and a new language, Orca, based on this model. At present, we have a compiler for Orca (based on Amsterdam Compiler Kit [45, 46] technology) and three prototype run time systems. One run time system runs on top of the Amoeba distributed operating system [7] and uses the 2-phase update protocol described in Section 4.3. The second run time system [39] runs on the bare hardware—a collection of MC68020s connected through an Ethernet—and uses the efficient reliable broadcast protocol described in [40]. For comparison, we have also written a run time system for a shared-memory multiprocessor.

Acknowledgements—The authors would like to thank Frans Kaashoek, Jack Jansen, Wim van Leersum, and Robert-Jan Elias for their work on the implementation of Orca. This research was supported in part by The Netherlands Organization for Scientific Research (N.W.O.) under grant 125-30-10.

REFERENCES

1. Tanenbaum, A. S. and Renesse, R. van. Distributed operating systems. *Comput. Surv.* **17**(4): 419–470; December 1985.
2. Bal, H. E., Steiner, J. G. and Tanenbaum, A. S. Programming languages for distributed computing systems. *ACM Comput. Surv.* **21**(3): 261–322; September 1989.
3. Tanenbaum, A. S. *Operating Systems: Design and Implementation.* Englewood Cliffs, NJ: Prentice–Hall; 1987.
4. Upfal, E. and Wigderson, A. How to share memory in a distributed system. *J. ACM* **34**(1): 116–127; January 1987.
5. Karlin, A. R. Sharing memory in distributed systems—Methods and applications. STAN-CS-87-1164 (Ph.D. dissertation), Stanford University, CA; January 1987.
6. Berglund, E. J. An introduction to the V-system. *IEEE Micro.* **6**(4): 35–52; August 1986.
7. Mullender, S. J. and Tanenbaum, A. S. Design of a capability-based distributed operating system. *Comput. J.* **29**(4): 289–299; August 1986.
8. Hoare, C. A. R. Communicating sequential process. *Commun. ACM* **21**(8): 666–677; August 1978.
9. Andrews, G. R. and Schneider, F. B. Concepts and notations for concurrent programming. *Comput. Surv.* **15**(1): 3–43; March 1983.
10. Andrews, G. R. Paradigms for process interaction in distributed programs. TR 89-24, University of Arizona, Tucson, AZ; October 1989.
11. Gehani, N. H. Broadcasting sequential processes (BSP). *IEEE Trans. Software Engng* **SE-10**(4): 343–351; July 1984.
12. Hoare, C. A. R. The emperor's old clothes. *Commun. ACM* **24**(2): 75–83; February 1981.
13. Strom, R. E. and Yemini, S. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Engng* **SE-12**(1): 157–171; January 1986.
14. Li, K. Shared virtual memory on loosely coupled multiprocessors. RR-492 (Ph.D. dissertation), Yale University, New Haven, CT; September 1986.
15. Spector, A. Z. Performing remote operations efficiently on a local computer network. *Commun. ACM.* **25**(4): 246–258; April 1982.
16. Mao, T. W. and Yeh, R. T. Communication port: A language concept for concurrent programming. *IEEE Trans. Software Engng* **SE-6**(2): 194–204; March 1980.
17. U.S. Department of Defense. *Reference Manual for the Ada Programming Language.* ANSI/MIL-STD-1815A; January 1983.
18. Shulman, N. V. The semantics of shared variables in parallel programming languages. Ph.D. thesis, New York University, N.Y.; June 1987.
19. Shibayama, E. and Yonezawa, A. Distributed computing in ABCL/1. In *Object-Oriented Concurrent Programming* (Edited by Yonezawa A. and Tokoro M.), pp. 91–128. Cambridge, MA: MIT Press; 1987.
20. Wilkes, C. T. and LeBlanc, R. J. Rationale for the design of Aeolus: A systems programming language for an action/object system. *Proc. IEEE CS 1986 Int. Conf. on Computer Languages*, pp. 107–122. Miami, FL; October 1986.
21. Yokote, Y. and Tokoro, M. Experience and evolution of ConcurrentSmalltalk. *SIGPLAN Not. (Proc. Object-Oriented Programming Systems, Languages and Applications 1987)* **22**(12): 406–415. Orlando, FL; December 1987.
22. Black, A., Hutchinson, N., Jul, E., Levy, H. and Carter, L. Distribution and abstract types in Emerald. *IEEE Trans. Software Engng* **SE-13**(1): 65–76; January 1987.
23. Forman, I. R. On the design of large distributed systems. *Proc. IEEE CS 1986 Int. Conf. on Computer Languages*, pp. 84–95. Miami, FL; October 1986.
24. Lucco, S. E. Parallel programming in a virtual object space. *SIGPLAN Not. (Proc. Object-Oriented Programming Systems, Languages and Applications 1987)* **22**(12): 26–34. Orlando, FL; December 1987.
25. Cheriton, D. R. Preliminary thoughts on problem-oriented shared memory: A decentralized approach to distributed systems. *Oper. Syst. Rev.* **19**(4): 26–33; October 1985.
26. Bisiani, R. and Forin, A. Architectural support for multilanguage parallel programming on heterogenous systems. *Proc. 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 21–30. Palo Alto, CA; October 1987.
27. Gelernter, D. Generative communications in Linda. *ACM Trans. Prog. Lang. Syst.* **7**(1): 80–112; January 1985.
28. Ahuja, S., Carriero, N. and Gelernter, D. Linda and friends. *IEEE Comput.* **19**(8): 26–34; August 1986.
29. Carriero, N. and Gelernter, D. The S/Net's Linda kernel. *ACM Trans. Comput. Syst.* **4**(2): 110–129; May 1986.
30. Carriero, N. The implementation of tuple space machines. RR-567 (Ph.D. dissertation), Yale University, New Haven, CT; December 1987.
31. Clark, K. L. and Gregory, S. PARLOG: Parallel programming in logic. *ACM Trans. Prog. Lang. Syst.* **8**(1): 1–49; January 1986.
32. Gregory, S. *Parallel Logic Programming in PARLOG.* Wokingham, England: Addison–Wesley; 1987.
33. Shapiro, E. Concurrent Prolog: A progress report. *IEEE Compt.* **19**(8); 44–58; August 1986.
34. Shapiro, E. *Concurrent Prolog: Collected Papers.* Cambridge, MA: MIT Press; 1987.
35. Taylor, S., Safra, S. and Shapiro, E. A parallel implementation of flat Concurrent Prolog. *Int. J. Parallel Prog.* **15**(3): 245–275; 1987.
36. Bernstein, P. A. and Goodman, N. Concurrency control in distributed database systems. *Comput. Surv.* **13**(2); 185–221; June 1981.
37. Bal, H. E. *Programming Distributed Systems.* Summit, NJ: Silicon Press; 1990.
38. Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C. Abstraction mechanisms in CLU. *Commun. ACM.* **20**(8): 564–576; August 1977.
39. Bal, H. E., Kaashoek, M. F. and Tanenbaum, A. S. A distributed implementation of the shared data-object model. *USENIX Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 1–19. Ft Lauderdale, FL; October 1989.
40. Kaashoek, M. F., Tanenbaum, A. S., Flynn Hummel, S. and Bal, H. E. An efficient, reliable broadcast protocol. *Oper. Syst. Rev.* **23**(4): 5–20; October 1989.
41. Bal, H. E., Kaashoek, M. F., Tanenbaum, A. S. and Jansen, J. Replication techniques for speeding up parallel applications on distributed systems. Report IR-202, Vrije Universiteit, Amsterdam, The Netherlands; October 1989.

42. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7): 558–565; July 1978.
43. Bal, H. E., Kaashoek, M. F. and Tanenbaum, A. S. Experience with distributed programming in Orca. *Proc. IEEE CS Int. Conf. on Computer Languages*, pp. 79–89. New Orleans, LA; March 1990.
44. Bal, H. E., Renesse, R. Van and Tanenbaum, A. S. Implementing distributed algorithms using remote procedure calls. *Proc. AFIPS Nat. Comput. Conf.*, pp. 499–506. Chicago, IL: AFIPS Press; 1987.
45. Tanenbaum, A. S., Staveren, H. van, Keizer, E. G. and Stevenson, J. W. A practical toolkit for making portable compilers. *Commun. ACM* **26**(9): 654–660; September 1983.
46. Bal, H. E. and Tanenbaum, A. S. Language- and machine-independent global optimization on intermediate code. *Comput. Lang.* **11**(2): 105–121; 1986.

About the Author—HENRI E. BAL received an M.Sc. in mathematics from the Delft University of Technology in 1982 and a Ph.D. in computer science from the Vrije Universiteit in Amsterdam in 1989. His research interests include programming languages, parallel and distributed programming, and compilers. Dr Bal has implemented the global optimizer of the Amsterdam Compiler Kit. Since 1985, he has been working on Orca, a new programming language for implementing parallel applications on distributed systems. The language design and implementation are described in several published research papers and in the book *Programming Distributed Systems*, which is a minor revision of his Ph.D. thesis. At present, Dr Bal is a staff member of the Department of Computer Science at the Vrije Universiteit in Amsterdam..

About the Author—ANDREW S. TANENBAUM received the S.B. degree from MIT and the Ph.D degree from the University of California at Berkeley. He is currently a Professor of Computer Science at the Vrije Universiteit in Amsterdam, The Netherlands. His research interests include distributed operating systems, programming languages, and compilers. Dr Tanenbaum is the author of three well-known textbooks (*Structured Computer Organization*, *Computer Networks*, and *Operating Systems: Design and Implementation*) as well as over 50 research papers on operating systems, computer architecture, networks, languages, and other topics. He is also the author of the MINIX operating system, and one of the principal designers of the Amsterdam Compiler Kit and Amoeba distributed operating system.