Diss. ETH Nr. 10736

Hermes – Supporting Distributed Programming in a Network of Personal Workstations

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
Spyridon Gerassimos Lalis
dipl. Informatik–Ing. ETHZ
born October 4, 1965
citizen of Athens, Greece

accepted on the recommendation of
Prof. Dr. B. Sanders, examiner
Prof. Dr. N. Wirth, co–examiner

1994

Acknowledgment

I thank Professor Beverly Sanders for supporting the work presented in this thesis, and for her valuable suggestions which helped to enhance the structure of this book. I also thank Professor Niklaus Wirth for his comments on drafts of the text, as well as for providing me with the sources of the Oberon system so that I could make the changes required to implement my ideas. I am indebted to both my supervisors not only for advising me on this thesis, but also for letting me apprehend computer science through a thrilling blend of theory, practice, and pioneering.

My colleagues at the Institut für Computersysteme contributed to a friendly working atmosphere; being among them was great fun. In particular, I wish to thank Philipp Heuberger and Martin Gitsels for the numerous inspiring discussions we had together during the past years.

Last but not least, my deepest gratitude goes to my friends and family. I simply can not imagine having done my studies and research without their encouragement and support.

Contents

Abstract

With distributed programming it is possible to support cooperation among users in a network, and to develop programs that run simultaneously on several machines to enhance their availability and performance. This potential is even more important nowadays where networks of workstations become an increasingly attractive alternative to big mainframes for organizing businesses and computing environments.

In this work, a system that promotes distributed programming in a network of personal workstations is presented. It is implemented on top of an existing operating system as a collection of modules that introduce new functionality in an incremental fashion. The key idea of our approach is to view data items as abstract objects coupled to abstract descriptions indicating how their content can be written and read. This allows generic data transfer programs to be built without a priori knowing the type of the data items that are to be transfered. As an extension of this component, an asynchronous mechanism is developed for exchanging arbitrarily complex messages over the network. At the application level, sending and receiving of messages occur asynchronously to each other so that a decoupling between the communicating processes is achieved. Finally, using the provided primitives a component is built which supports programming of extensible application objects that can be dynamically installed, referenced, and collected over a network. Such objects are automatically notified about incoming messages, thus they operate like special state machines whose transitions are triggered by message events; this technique is appropriate for capturing a wide range of distributed programs.

Due to its clean internal structure, the system presents itself as a set of well separated, yet cooperating parts which can also be individually accessed by the programmer. This allows for great flexibility in application development. Furthermore, the system encourages a disciplined programming style and guarantees type safety which we consider to be an important property of development environments. Our implementation also demonstrates that the proposed approach leads to acceptable performance at only a modest software cost.

Kurzfassung

Mit verteiltem Programmieren ist es möglich die Zusammenarbeit zwischen Benutzern in einem Netzwerk zu unterstützen, und Programme zu entwickeln, die gleichzeitig auf mehreren Machinen laufen, um ihre Verfügbarkeit und Leistung zu erhöhen. Dieses Potential ist um so wichtiger heutzutage wo Netzwerke von Arbeitsplatzrechnern eine immer attraktivere Alternative zu Grossrechnern für die Organisation von Betrieben und Rechenumgebungen werden.

In dieser Arbeit wird ein System, dass verteiltes Programmieren in einem Netzwerk von Arbeitsplatzrechnern fördert vorgestellt. Es ist auf ein bestehendes Betriebssystem, als eine Sammlung von Moduln, die inkrementel neue Funktionalität einführen, implementiert. Die zentrale Idee unseres Ansatzes ist Daten als abstrakte Objekte zu betrachten, an denen abstrakte Beschreibungen angekoppelt sind, die angeben wie ihr Inhalt geschrieben und gelesen werden kann. Es ist somit möglich generische Programme zur Übertragung von Daten zu erstellen, ohne den Typ der Daten a priori zu wissen. Als Erweiterung dieser Komponente, wird ein Mechanismus für den Austausch beliebig komplexer Meldungen über ein Netzwerk entwickelt. Das Senden und Empfangen von Meldungen werden auf der Applikationsebene asynchron zueinander abgewickelt, so dass eine Entkopplung zwischen den kommunizierenden Prozessen erreicht wird. Schliesslich, wird mit Hilfe der angebotenen Primitiven eine Komponente gebaut, die das Programmieren von erweiterbaren Applikations–Objekten unterstützt, die über ein Netzwerk installiert, referenziert und entfernt werden können. Solche Objekte werden automatisch über ankommenden Meldungen benachrichtigt, sie funktionieren daher wie Zustandsautomaten deren Übergänge durch Meldugsereignissen angestossen werden; diese Technik ist gut geeignet, um eine breiten Bereich von verteilten Programmen abzudecken.

Wegen dessen klaren Struktur, präsentiert sich das System als eine Menge von sauber getrennten, jedoch zusammenarbeitenden Teilen, die der Programmierer auch einzeln zugreifen kann. Dies erlaubt grosse Flexibilität beim Entwurf von Applikationen. Ferner, fördert das System einen disziplinierten Programmierstil und garantiert Typen–Sicherheit, was wir für eine wichtige Eigenschaft von Entwicklungsumgebungen halten. Unsere Implementation beweist auch, dass der vorgeschlagene Ansatz zu akzeptablen Leistungen führt, und dies bei einem bescheidenen Software–Aufwand.

Chapter 1 Introduction and Overview

In contrast to conventional tools, computers are universal machines. The same hardware can be used to support different tasks, even several of them simultaneously, provided that the computer has been appropriately configured. But computers have another property that reveals unprecedented possibilities: computers can be connected together via communication networks so that programs residing on different machines can cooperate with each other.

Distributed programming is a large step past the old–fashioned view of a system in isolation, because it introduces the opportunity to perform tasks that lie far beyond the capabilities of a single machine. Time consuming tasks can be executed on idle processors, perhaps using several of them in parallel, thereby accelerating processing and reducing the workload of interactively used machines. It is also possible to distribute or replicate vital resources on multiple machines to achieve availability despite hardware and software failures. The network can also be exploited to support electronic interaction and sharing between users.

What is even more important, is that distributed programming is not a temporary trend whose importance is likely to diminish with the passage of time. On the contrary, as computers are becoming equiped with faster processors, more main memory, and abundant secondary storage, at a sinking cost, networks consisting of loosely coupled personal workstations that can be operated in isolation yet are capable of sophisticated cooperation over a network will become increasingly attractive. Indeed, there is already a significant trend towards downsizing from large mainframes to networks of smaller machines.

There is no free lunch though. Distributed programs are inherently more complex than non–distributed programs, and thus often far from easy to implement. The additional complexity stems from the fact that a distributed program, unlike a non–distributed program, is not "a" single program but consists of several parts that reside on different machines which execute (and fail) independently of each other. This makes communication and synchronization in distributed programs awkward to implement. It also introduces the possibility of having partial failures that must be dealt with to achieve robustness. Consequently, distributed programming remains a rather unattractive task unless sufficient support is provided so that these problems can be addressed in a straightforward way.

Motivated by the great potential of cooperative processing in a network, seeking for distributed programming support has been the focus of intensive research since the early days of computing. However, a "best" way to achieve this goal has not yet been found. In fact, the experience that has been acquired so far with respect to central issues of distribution suggests that there are no universal solutions. Not only does each computing environment serve its own specific purposes, but there is also a great variety of distributed applications with different goals and diverging semantics. It is therefore difficult to anticipate the requirements of future applications, and support which is ideal for a certain type of programs may turn out to be far less appropriate, or even totally inapplicable for others.

This drastically reduces the worth of ready–to–use tools for producing applications, and, by contrast, it underlines the importance of building new customized services in a straightforward way based on simple primitives. Hence, in this context, it does not seem meaningful to invest much effort in producing big and expensive pre–fabricated software blocks in the hope to cover the needs of coming applications; instead, making the development of application specific support services as easy as possible appears to be a better choice. This leads us to the vision of a small system that acts as a kernel providing only a few services which can be extended in a simple way, by programming new components, to introduce new functionality. After all, even for non–distributed systems promoting programming, rather than trying to eliminate it, is a key step towards making the computer a truly capable and flexible work bench, adaptable to the programmer's need. Given the manifold of distributed applications, we strongly believe that this holds even more for distributed programming.

The Hermes System

With the objective to investigate support for distributed programming according to this principle, we implemented the Hermes system which is described in this thesis. Hermes supports interprocess communication, and remote installation of programs that can be invoked over the network transparently, but lacks specialized application–oriented packages. The provided primitives combine flexibility with high level functionality in a balanced way, so that they can be directly used to program simple interactions over the network in a simple way, and still serve as a foundation upon which more elaborate support for specific application classes can be built.

Hermes is structured as a hierarchy of modules, each addressing a different problem of distribution in an incremental fashion (figure 1). Despite the tight coupling and interplay between the different components, the system consists of three essentially distinct layers that can be individually accessed by the programmers, depending on the requirements of the corresponding applications.

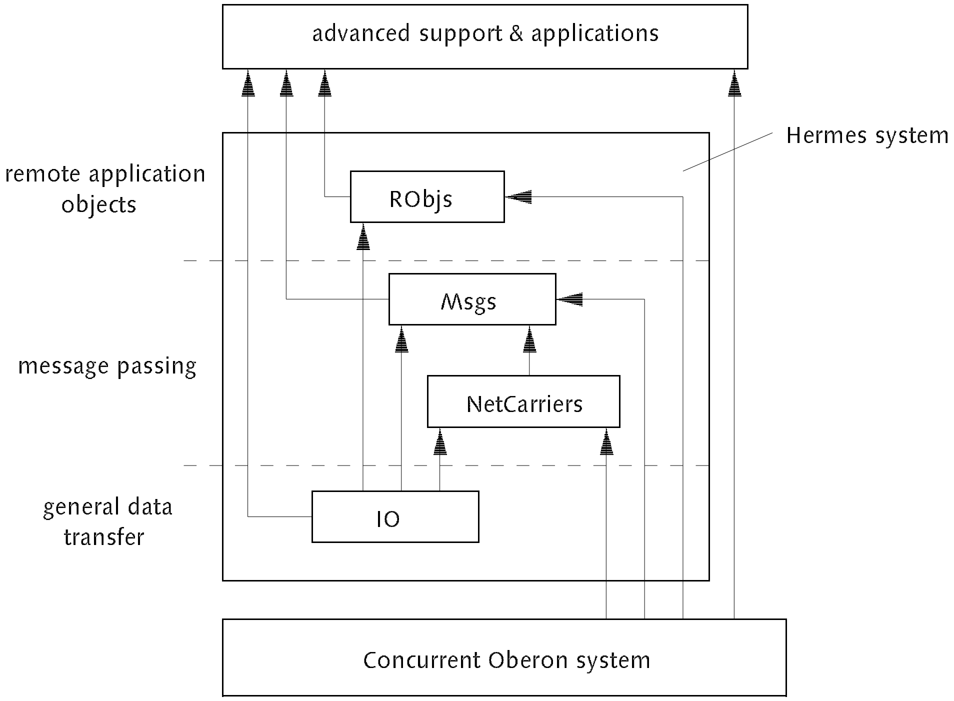

Figure 1 module structure and functionality layers of the Hermes system

The kernel of our approach is a general framework that supports input and output of arbitrarily complex data types. Due to the use of object–oriented techniques, the transfer code of a data type is directly attached to its instances so that it is possible to trigger transfer of a data object merely by holding a reference to it. Moreover, the same transfer code of a data type can be used to perform different transfers, for example to transfer instances thereof to and from disk, or to send them over the network. Both features considerably simplify the implementation of generic programs for transfering application objects.

As an extension of this framework, a facility is implemented to support high level data exchange between activities over the network. Communication is supported via asynchronous message passing primitives with messages being ordinary data types. In other words, applications can define their own messages as arbitrarily complex data types with rich internal structure yet still send them over the network as if they were single indivisible items. Data conversion and transmission is transparent, and messages are delivered to the application in their original form so that the contents of a message can be accessed directly after it has been received from the network. This combines the flexibility of message passing, in particular with respect to introducing parallelity, with the high level data abstraction of other interprocess communication paradigms, such as the remote procedure call.

The services of the message passing component are in turn used to implement dynamic installation and invocation of application programs over the network. Driven by the fact that a wide range of distributed programs such as asynchronous protocols and general purpose servers can be viewed as event driven machines with state that is preserved across invocations, support is given in an object–oriented way via special objects, also called remote objects, that can be invoked over the network by sending them messages. Remote objects can be developed in a true object–oriented way, i.e. it is possible to extend the behaviour of existing objects by augmenting their message handling capability with additional code for processing new messages. There is also support for introducing intra–object concurrency and sharing.

Hermes is written in the Oberon–2 programming language [Moessenboeck91] which is an extension of the Oberon language [Wirth88]. It runs on top of the Concurrent Oberon system [Lalis94], a special version of the original Oberon system [Wirth89a] we have implemented for the Ceres1 and Ceres2 personal workstations [Eberle87]. While Hermes is primarily conceived for supporting distributed programming in the Oberon environment (and on Concurrent Oberon), which undoubtedly affects its implementation, its key ideas and overall design principle apply to other platforms as well.

Reading this Thesis

The rest of the thesis is organized in a bottom up fashion as follows. Chapter 1 describes the changes made to the standard Oberon system with respect to concurrency and networking in order to make an implementation of our approach practically feasible. In the subsequent chapters 2, 3, and 4, the aforementioned layers of the Hermes system are introduced and their implementation is discussed in detail. Chapter 5 presents a few applications that have been implemented using Hermes. Finally, we summarize our work and conclude. The code sizes of the individual components of our implementation along with some performance data is given in the appendix. Familiarity with the Oberon environment is assumed in this thesis; no detailed knowledge is required though. For an introduction to the Oberon language and system the reader is referred to the literature.

Chapter 2 Extending the Oberon System

Our environment consists of networked Ceres personal computers that run the Oberon operating system. Oberon is a single user, single process operating system that was conceived both as a research vehicle for investigating the concept of object–oriented development and as an efficient programming platform for the Oberon and Oberon–2 programming languages to be used in our institute as well as in lower division computer science courses at ETH Zurich.

Although Oberon workstations are connected with each other via a local area network, network support is limited. This chapter motivates and describes changes and extensions made to the original system in order to increase its potential with respect to networking and to provide the basic tools for implementing distributed services. The new functionality is useful even if viewed in isolation to the rest of our work, since it can be exploited to enhance implementation of the existing network services significantly.

Introducing Concurrency

In contrast to systems that maintain a separate process and address space for each window, in Oberon, windows are implemented as passive objects with the capability of handling input events. Keyboard and mouse drivers are polled by a single process called the central (or Oberon) loop, and when an event is detected, a corresponding message is sent to the affected window with a call to its event handling procedure which processes it and returns control to the Oberon loop. The conventional method for executing code from the user interface is via special procedures called commands that are invoked from within the window handlers during message processing. When command execution terminates control returns to the window handler that invoked the command, and then back to the Oberon loop which continues to poll for input events.

Although this model of control works well for interactive applications, because handling of input events is essentially instantaneous and most commands require little time, there are cases where it is not appropriate. For example, it is inconvenient to implement a long computation as a command because its execution will block the system until it is finished, perhaps for several minutes or even hours. Also, there are applications which have to react to external events that can occur practically anytime and where polling via explicitly invoked commands is not acceptable. To accommodate such applications the Oberon system offers a mechanism called tasks. Tasks are special procedures that are invoked periodically from within the Oberon loop when all input handling has been performed. In other words, tasks can be viewed as commands that are executed repeatedly in the background without an explicit user request as soon as the machine becomes idle.

But since tasks are part of the Oberon loop, their execution can be delayed arbitrarily long by input handling and long running commands. This makes it impossible to guarantee that a certain task will be executed within a reasonable amount of time, which is vital for applications with real time constraints such as network protocols. Another problem is that tasks, like commands, run until they terminate. Thus to implement background processing that still allows interactive use of the system, a long computation must be broken down into a sequence of task invocations. The individual computation steps must execute "long enough" but not "too long" because otherwise, either the task will block the system, or processor time will be waisted mostly in control transfers. However, thinking about how much time is needed to execute a particular piece of code has little to do with the problem the programmer is actually trying to solve. Such calculations are annoying, machine dependent, and even impossible to make if execution time is a function of parameters that are not known at implementation time. In addition, since tasks and the Oberon loop are executed on the same stack, the intermediate state of a computation that runs as a task must be explicitly saved each time before returning control to the Oberon loop. This can be extremely cumbersome for programs that cannot be modeled as simple state machines with fine grained state transitions.

Our experience with tasks showed that these limitations are indeed of practical importance, and therefore we decided to develop a special version of the Oberon system, called Concurrent Oberon, that is free of these problems. An additional motivation for this development was the desire to employ Oberon, which is currently used in lower division computer science courses at ETH, for programming exercises in concurrent programming. Concurrent Oberon, provides threads (lightweight processes that share memory) along with a simple priority scheduler. Control of the processor is passed to different threads transparently and without requiring the programmer to explicitly save state. The scheduler recognizes three priority levels, and round robin scheduling is used to allocate processor time among threads of equal priority provided there are no available threads with higher priority.

Integrating concurrency in an elegant way into a system that is designed to be sequential posed an interesting engineering problem. Complicated globally shared data types are pervasive in Oberon, and applications are implemented using the implicit assumption that a sequence of operations is executed atomically, in other words it is often assumed that the state of an abstract data type will not change between successive operations. Hence, the obvious approach of adding synchronization to the operations of all abstract data types in the system (e.g. using monitors) is insufficient.

For this reason, and since the "single process multi–tasking" model works well for user driven applications, in Concurrent Oberon, the Oberon loop is a special thread which, by convention, is responsible for processing all keyboard and mouse events, controlling the screen and accessing globally shared data structures related to input and display handling. As a consequence, concurrency is transparent for ordinary Oberon programs that are executed from within the Oberon loop, hence such programs can freely access global data types without any explicit synchronization. Other threads may access such data structures only in a controlled way to avoid inconsistencies.

This design provides the desired functionality without adding significant complexity, and as an important practical matter, makes Concurrent Oberon compatible with the standard system. Existing applications can be run on our system without any modifications or recompilation. The only drawback of this approach is that programs that are to be run as threads must be implemented to explicitly synchronize with the Oberon loop when accessing I/O related data types. We do not feel that this is a crucial restriction though, because programming the required synchronization is not difficult, and we do not expect background threads to interact with the display often.

Programming Primitives

In Concurrent Oberon, the new capabilities are introduced by means of an additional module called Threads. This module provides procedures for creating and destroying threads, and a set of operations that can be used to implement synchronization tools (e.g. semaphores, monitors, signals). Module Threads also provides the operations for synchronizing with the Oberon loop.

Thread Creation and Destruction

Threads are implemented as pointers to a thread descriptor containing a stack and state information about the thread needed by the scheduler. Some of the record fields are exported so that their values can be obtained in a simple and efficient way. These fields must not be modified by the programmer.

```
DEFINITION Threads;

  CONST
    ready = 0; asleep = 1; suspended = 2; destroyed = 3; trapped = 4;
    low = 0; norm = 1; high = 2;

  TYPE
    Thread = POINTER TO ThreadDesc;
    ThreadDesc = RECORD
                   state, priority: SHORTINT;    (*read–only*)
                   incNo: LONGINT;               (*read–only*)
                 END;

    ThreadProc = PROCEDURE;

  VAR cur: Thread;    (*currently executing thread*)

  PROCEDURE Create (this: Thread; proc, trapproc: ThreadProc; wsp: LONGINT);
  PROCEDURE Destroy (this: Thread);
```

Threads are initialized with calls to procedure *Create*. The procedure that is to be executed as a thread and the size of the stack that will host the execution must be supplied as parameters. Optionally, a procedure that will be invoked if the thread experiences a run time error can be specified, giving programmers the ability to perform cleanup actions. The specified thread descriptor may be freshly allocated or can belong to a thread that has already terminated. To make failure detection possible despite recycling of descriptors, thread descriptors also have an incarnation number *incNo* that is incremented each time the descriptor is re–used (the incarnation number of freshly allocated descriptors is 0). By default, threads are created with low priority and are suspended, i.e. must be resumed explicitly to start execution.

Procedure *Destroy* is used to terminate the specified thread. Destruction occurs instantly only if a thread invokes the operation on itself. Otherwise, the specified thread is marked correspondingly and is destroyed as soon as it receives control of the processor.

Parameters can be passed to a thread by using the type extension facility of the Oberon language as shown for a thread that continuously increments a local variable by an amount that is passed to it as a parameter:

```
TYPE
  MyThread = POINTER TO MyThreadDesc;
  MyThreadDesc = RECORD (Threads.ThreadDesc)
                   inc: LONGINT
                 END;
```

```
PROCEDURE MyProc;
   VAR i: LONGINT;
BEGIN i := 0;
   LOOP INC(i, Threads.cur(MyThread).inc) END
END MyProc;

PROCEDURE Start;
   VAR t: MyThread;
BEGIN
   NEW(t); t.inc := 100;
   Threads.Create(t, MyProc, NIL, 128); Threads.Resume(t)
END Start;
```

New thread descriptor types are defined by augmenting the base descriptor with new fields. Thread procedures access the additional fields of extended descriptors using type guards on the global variable *Threads.cur* indicating the currently executing thread.

Scheduling and Control Operations

Module Threads supports three different priority levels. At each priority level, selection is round robin and a thread is considered for selection only if there are no available threads of higher priority. Consequently, processing at a given priority level remains unaffected even if there are many threads with lower priority. This avoids performance degradation of high priority activities despite the presence of other low priority threads.

While this selection strategy may lead to starvation of low priority threads, this is not a problem if threads are assigned priorities in a disciplined way. In Concurrent Oberon, the three priority levels are intended for urgent event handling, interactive processing, and long computations, respectively. This hierarchy makes starvation a desirable property, because, for example, it implies that user commands are executed at their customary speed even if there are several threads executing background computations, thereby avoiding the timesharing effects of multi–user systems. If a thread cannot be clearly placed into one of these three categories, then its priority can be changed with calls to procedure *SetPriority* whenever this is required.

```
DEFINITION Threads;

   PROCEDURE SetPriority (this: Thread; prio: SHORTINT);

   PROCEDURE Suspend;                    (*processor is released*)
   PROCEDURE Sleep (ticks: LONGINT);     (*processor is released*)
   PROCEDURE Resume (this: Thread);      (*processor is no released*)

   PROCEDURE Pass;    (*yield processor*)
```

Additional operations for changing the state of threads are provided. These can be used to avoid busywaiting when a thread must wait for some conditions to be met before it continues with its processing. The currently executing thread can be suspended or put to sleep for some amount of time with calls to *Suspend* and *Sleep*, respectively. In both cases, the state of the thread is changed correspondingly, and the scheduler is called to pass control to another thread. Threads are reverted to the ready state, if they are resumed with a call to *Resume*, or, in case they have been put to sleep, when the specified amount of time elapses. It is also possible to call the scheduler directly with a call to procedure *Pass*. This does not change the state of the calling thread.

Rather than using a separate scheduler thread through which control is switched among threads, the code for transfering control is executed directly from within the thread that is releasing the processor. To keep selection efficient, four separate lists are maintained, one for each priority level containing threads that are ready to run, and one for blocked threads. For each of the priority lists, a pointer denoting the next thread to receive control is used to implement a round robin scheme. Hence, choosing the next thread requires inspection of at most three pointers, which is fast and can be done in constant time independently of the number of threads in the system.

Even though, conceptually, threads change lists when the corresponding control operations are invoked, the lists are actually updated when the scheduler is called. This obviates synchronizing the operations that change the state of threads, which is particularly convenient if it is desirable to invoke such operations from within interrupt handlers. Notably, defering list updates until the next scheduling phase does not have any influence on the semantics, since the actual positioning of a thread within these lists is relevant only for the scheduling process.

Synchronization

In a shared memory system, concurrent access of data must be synchronized to avoid inconsistencies. This is typically achieved by using semaphores [Dijkstra72] or monitors [Hoare74]. It is also possible to combine these methods, for example, Modula–3 [Nelson91] introduces locks that are declared, acquired, and released like semaphores, but are automatically released when a thread blocks on a condition within a critical section, and reacquired when the thread is resumed. Other systems like CSP [Dijkstra68] and Ada [DoD80] achieve synchronization through synchronous rendezvous schemes. However, none of these approaches is clearly superior. While each one of them is a direct means of modeling synchronization for a particular type of process interaction, it can lead to cumbersome programs when it is applied to a different situation.

Motivated by this observation, in Concurrent Oberon, synchronization is supported by introducing the concept of atomic sections. Two operations *BeginAtomic* and *EndAtomic* can be used to implement code whose execution will not be preempted by transfering control to another thread. At the begining of an atomic section, the executing thread is marked so that it will not be taken control by force (interrupts are still serviced during execution of atomic sections). Hence, no transparent control transfer occurs during execution of atomic sections. It is nevertheless possible to release the processor by calling the scheduler (directly or indirectly) via the provided control operations. Since this does not unmark the thread, the remaining part of the atomic section will be atomic when the thread which yielded the processor resumes execution. An atomic section thus resembles a global monitor where atomicity can be violated if control is explicitly given away.

```
DEFINITION Threads;

   PROCEDURE BeginAtomic;
   PROCEDURE EndAtomic;
```

To hinder threads from monopolizing the processor when subsequently executing atomic sections, EndAtomic automatically yields the processor when the thread has kept the processor past a given time limit. Properly nested atomic BeginAtomic–EndAtomic pairs are also allowed, and in this case the processor is potentially released only on the outhermost invocation of EndAtomic.

Atomicity is intentionally introduced as an explicit concept that does not reveal how the system actually transfers control among threads. This forces the programmer to document atomicity requirements in a program and enhances maintainability and portability since the implementation of concurrency may vary without invalidating correctness of existing programs. This is in contrast to less transparent approaches where the method for achieving concurrency is publically known, and can be exploited by the programmer to hinder scheduler invocations.

Atomic sections are not intended as an all purpose synchronization device, but are to be used only for small critical sections. Tools for coarse grained synchronization such as semaphores and monitors can be implemented by using atomic sections in combination with the provided control procedures. The available primitives can also be used to extend the system by introducing other communication paradigms besides shared memory, such as message passing [BrinchHansen70, Gehani86] or pipes [Presotto85].

Atomicity Semantics

Semantically, the BeginAtomic and EndAtomic operations are equivalent to open and closed angle brackets "<...>" often used in the literature to denote atomic sequences of operations. Although nesting can occur in practice, from a formal point of view only the outhermost bracket–pair is relevant. Thus when argueing about the correctness of a program, all inner atomic sections can be ignored.

Procedures Suspend, and Sleep change the state of the calling thread and invoke the scheduler in a single atomic action, whereas Pass simply calls the scheduler. Although the exact effects of these operations differ, within atomic sections these calls have the same effect with respect to atomicity, since they all release the processor. They can therefore be substituted with a pair of angle brackets and an empty statement inbetween "> skip <" to indicate violation of atomicity.

These simple rules allow programmers to gain additional confidence in their implementations by exploiting the advantages of more formal and abstract descriptions. As an example, we show how the implementation of general semaphores can be carried over to a more formal description of the P and V operations by simple application of the given mappings:

```
concrete implementation              abstract description

  BeginAtomic;                         <
  WHILE s = 0 DO suspend END;            WHILE s = 0 DO > skip < END;
  s := s–1;                               s := s–1;
  EndAtomic;                           >

  BeginAtomic;
  s := s+1;                            < s := s+1; >
  IF waiting THEN resume any END;
  EndAtomic;
```

Correctness is guaranteed, because evaluation of the loop condition embodying the guard for decreasing the value of a semaphore, and execution of the corresponding statement are done in a single atomic action. The processor is released as long as the condition is not satisfied so that control is given to other threads, and eventually to one that will increase the semaphore, thereby establishing the awaited condition. This temporary violation of atomicity does not affect correctness though, since the thread which is polling the precondition will continue to execute atomically when it is given control again. In fact, such while loops within atomic sections are equivalent to await statements within angle brackets used in [Andrews91] to model synchronization.

Global Synchronization (with the Oberon loop)

The decision to give the Oberon loop free access on I/O related data types largely obviates synchronization, because these data types are typically accessed by window handlers, commands, and tasks, which are all executed from within the Oberon loop. It is however possible that other threads also wish to access the same data structures as the Oberon loop, for example to display output.

Since such concurrent accesses may cause arbitrary inconsistencies, it is mandatory that threads explicitly synchronize with the Oberon loop when executing critical operations that potentially interfere with the activities of "ordinary" Oberon programs. A separate set of primitives is provided especially for this purpose.

```
DEFINITION Threads;

  TYPE
    OberonActionProc = PROCEDURE (this: OberonAction);

    OberonAction = POINTER TO OberonActionDesc;
    OberonActionDesc = RECORD
                                       body: OberonActionProc;    (*code of action messages*)
                                    END;

  PROCEDURE QueueOberonAction (this: OberonAction);
  PROCEDURE DoOberonActions;

  PROCEDURE LockOberon;
  PROCEDURE UnlockOberon;
```

The first method is based on a mutual exclusion scheme between the Oberon loop and other threads using a lock maintained by module Threads. If critical data structures are to be accessed from within a thread, procedure *LockOberon* must be invoked prior to execution of the corresponding operations, and *UnlockOberon* must be called afterwards to release the lock. Below, a thread procedure that continuously prints a message on the system's log–viewer using locking is shown:

```
PROCEDURE PrintMsg0;
  VAR W: Texts.Writer;
BEGIN
  Texts.OpenWriter(W);
  LOOP
    Texts.WriteString(W, "hello world"); Texts.WriteLn(W);
    Threads.LockOberon;
    Texts.Append(Oberon.Log, W.buf);
    Threads.UnlockOberon
  END
END PrintMsg0;
```

Acquiring the lock suspends the calling thread if the lock is already held by another thread, and the thread is resumed when the lock becomes free. Blocking is likely to happen when the machine is used interactively since in this case the lock is held by the Oberon loop during most of the time. Nesting of these commands is allowed so that code containing them can be executed from within the Oberon loop as well as by another thread.

An alternative technique for accessing data types shared with the Oberon loop is via asynchronous message passing using so called actions messages. This requires more effort to implement than direct mutual exclusion but is more appropriate when it is important that a thread triggers an operation without blocking (e.g. displaying output when monitoring a network). The implementation of a thread procedure that produces exactly the same output as the above example using action messages instead of locking is given in the following (we assume that the global variable W has been appropriately initialized):

```
TYPE
   PrintAction = POINTER TO PrintActionDesc;
   PrintActionDesc = RECORD (Threads.OberonActionDesc)
                          s: ARRAY 32 OF CHAR
                      END;

PROCEDURE PrintProc (a: Threads.OberonAction);
BEGIN
   Texts.WriteString(W, a(PrintAction).s); Texts.WriteLn(W);
   Texts.Append(Oberon.Log, W.buf)
END PrintProc;

PROCEDURE PrintMsg1;
   VAR a: PrintAction;
BEGIN
   LOOP
      NEW(a); a.body := PrintProc; a.s := "hello world";
      Threads.QueueOberonAction(a)
   END
END PrintMsg1;
```

The main idea of this approach is to put critical operations into special *OberonAction* messages and place these messages into a a queue. This queue is processed with calls to procedure *DoOberonActions* which removes the messages deposited in the queue and executes the contained *ActionProc* procedures. By convention, processing of the queue is done from within the Oberon loop, asynchronously to the threads that queued the messages. Parameter passing is implemented in the same way this is done for threads, i.e. by extending the base message type with additional fields that are accessed by type guards.

Integration of Concurrency into the Oberon System

Module Threads does provide the primitives for implementing concurrent programs, but it is not entirely responsible for introducing concurrency in the Oberon system. As shown in figure 1, full integration was achieved by appropriately modifying existing modules of the Oberon system.

The new module versions use the previously described operations of module Threads along with a few special systems programming tools (see below) to make concurrency transparent for the rest of the Oberon modules and applications. Specifically, low level synchronization, synchronization with the Oberon loop, as well as preemption, time sharing, and the desired interplay between interactive processing and background threads were achieved by changing five components of the standard Oberon system.

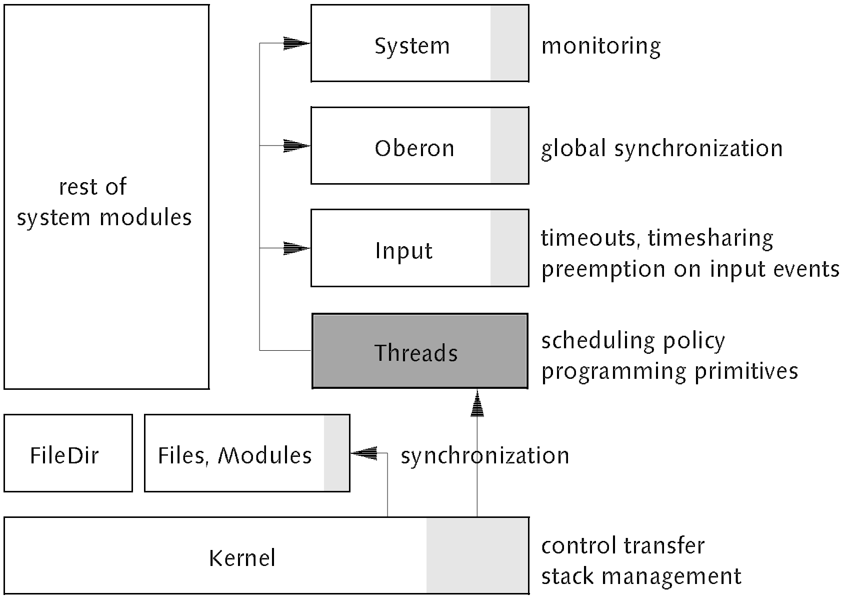

Figure 1 module structure of Concurrent Oberon

The system's kernel was also augmented with support for multi–threaded programming. These changes required less than 10 kilobytes of code, including the Threads module. A detailed accounting of the newly introduced complexity is given in the appendix.

Preemption, global synchronization, and monitoring

Although module Threads implements the scheduling policy, i.e. the algorithm according to which the threads are selected for execution, it does not introduce any transparent control transfer mechanism. Hence, to ensure that threads which must react quickly to events are indeed given the processor, and this sufficiently fast, it is mandatory to invoke the scheduler when the corresponding events occur.

To guarantee prompt reaction to user input events, the priority of the Oberon loop is set to normal and the scheduler called whenever keyboard and mouse events are present (module Input). Reaction to keyboard input is instantaneous, since the corresponding events are communicated to the system via interrupts and thus direct preemption is possible. Mouse events are not detected as quickly, because the mouse device is polled from within the timer interrupt handler.

The timer interrupt handler is also used to trigger control transfer periodically. This guarantees that processor time will be divided evenly among threads of equal priority, which is especially important if a workstation is to be used to perform several computations in parallel. Further, timer interrupts are used to notify the scheduler about the passage of time with calls to a special procedure called *Tick* that increments the internal time of module Threads and updates the state of sleeping threads.

The Oberon loop (in module Oberon) has also been extended as follows. After all input events have been processed, its priority is lowered, thereby giving low priority threads a chance to execute. Notably, the Oberon loop cannot be suspended when there are no input events present (as this would be natural to do), because tasks must still be executed. With this scheme low priority threads can nevertheless utilize cycles not needed for interactive processing without affecting the response time of the interactive system. The new Oberon loop is thus an extension of the old program:

```
LOOP
  Threads.LockOberon;
  WHILE keyboard or mouse events DO
    notify affected viewer
  END;
  execute next task
  Threads.DoOberonActions;
  Threads.UnlockOberon;
  Threads.SetPriority(Threads.cur, Threads.low)
END;
```

Additional changes are made to implement synchronization for globally shared data types using the provided primitives. In other words, the Oberon loop invokes LockOberon before commencing with input and task processing, and UnlockOberon when there are no more input events. This guarantees that programs executed from within the Oberon loop do not need to synchronize when accessing global I/O related data types. The OberonAction actions queued by threads are also processed from within the Oberon loop.

Finally, a few auxiliary commands are implemented (in module System) so that threads can be monitored and destroyed interactively. There is a command for opening a viewer that contains all running threads. For each thread, its identification number, procedure and state are displayed. The available threads are accessed via a special enumeration procedure *Enumerate* provided by module Threads. Additional commands can be used to obtain a stack trace of a thread or to selectively remove threads from the system.

Low–Level Synchronization and Stack Management

While it is meaningful to require that threads synchronize explicitly when accessing globally shared data structures, this approach is inconvenient for low–level objects such as files and modules, because the corresponding accesses are hardly visible for the programmer. Therefore, synchronization in the so called inner core modules was achieved by explicitly protecting critical code that must not be executed concurrently.

In order to keep the system's innner core independent of the concrete implementation of concurrency, so that it can serve as a common base for different approaches, a direct import relationship between these modules and the Threads module is avoided via an indirection. Two procedure variables *lock* and *unlock* defined in the kernel are used by modules Files and Modules to achieve mutual exclusion for all critical sections. These variables initially contain empty procedures and, at a later point in time, are overwritten by module Threads with real synchronization operations (in our case BeginAtomic and EndAtomic).

```
DEFINITION Kernel1;

  CONST
    0 = norm; 1 = trap; 2 = int;    (*mode codes*)

  TYPE
    Stack = POINTER TO StackDesc;
    StackDesc = RECORD
                  mode: SHORTINT;    (*execution mode*)
                  inSVC: BOOLEAN;    (*interrupted SVC code?; valid if mode >= 2*)
                END;

    StackProc = PROCEDURE;

  PROCEDURE NewStack (VAR stk: Stack; wsp: LONGINT);
  PROCEDURE InitStack (stk: Stack; proc: StackProc);
  PROCEDURE Transfer (stk: Stack);
  PROCEDURE Current (VAR cur: Stack);
  PROCEDURE StackState (stk: Stack; VAR fp, pc, used: LONGINT);

  VAR lock, unlock: PROCEDURE;  (*mutual exclusion*)
```

Module Kernel was also enriched with a rigorous stack management and operations for initializing stacks and control switching. This support is intentionally low level and tightly coupled with the compiler conventions for transfering control in procedure calls. It serves only as a foundation upon which models of multi–threading such as coroutines and threads with preemptive scheduling can be built. (To avoid changing the interface of module Kernel –this would make our system incompatible to the standard Oberon system– all new definitions are exported by a stub module called Kernel1.)

Stack objects are allocated with calls to *NewStack* where the maximum stack size must be passed as a parameter. Stacks are inserted into a list and are automatically collected as soon as they are no longer referenced so that the corresponding memory is returned to the system.

Initialization and control transfer procedures provide support for implementing coroutines [Marlin80]. Procedure *InitStack* arranges the contents of the stack so that the specified procedure will be invoked when control is transfered to the stack for the first time, but does not perform a context switch.

Control transfer is implemented by procedure *Transfer* that executes the appropriate actions depending on the state of the current execution. This information is recorded in the fields *mode* and *inSVC* of the current stack descriptor. The first field indicates whether the execution is in normal, trap, or interrupt mode. In the latter case, the second field can be used to determine if a supervisor call has been interrupted.

To avoid copying when transfering control, each stack comes with its own system stack region, hence switching control requires saving and reloading the two stack pointer and the frame pointer registers. If transfer is invoked from within an interrupt, then the general purpose and floating point registers are also saved on the stack of the suspended execution before executing the normal transfer code.

Additional changes to the kernel were modifications of the garbage collector to mark pointers on all stacks. After the main mark phase, the stack list is traversed, and for each stack the valid region between stack origin and stack pointer is checked. Values that could possibly be pointers are put into an array whose contents are eventually compared with the actual heap structure. Marking occurs only if pointer candidates indeed turn out to be pointers, i.e. are equal to addresses of unmarked objects.

To allow our implementation to be ported on machines which lack memory management hardware (e.g. Ceres–3 workstation [Heeb91]), stacks are allocated on the heap. An unpleasant consequence of this approach is that stacks cannot be dynamically extended when their physical boundary is reached, because the neighbouring memory locations may already be occupied by other heap objects. Instead, the full size of a stack must be claimed already at allocation time, thereby wasting memory if only a small potion thereof is actually used. Stack overflow is detected through checks made on each procedure call, and causes a run–time error. The corresponding instructions are generated by the compiler (the hardware does not support stack overflow checks) as part of the procedure entry protocol after the parameters have been deposited on the stack. Each stack region comes with a safety reserve that can be overwritten without any destructive effects so that delayed detection of overflow does not seriously endanger robustness of the system.

Enhancing the Network Support of Oberon

In the standard Oberon system, the network plays a minor role. Oberon workstations are completely autonomous, and the network is used merely to implement a few remote services [Wirth89b]. Client programs are implemented as commands that send a request over the network and remain blocked until the server completes processing, or a timeout occurs. The corresponding server parts are implemented in a single task that polls the network, processes incoming requests, and sends the replies back. Since both commands and tasks execute sequentially, during a client–server transaction, the processor on both machines is exclusively used for the purpose of the current data transfer.

This fact is reflected by the design of the network driver that allows packets that have been deposited into the system's network buffer to be inspected only by a destructive operation. Consequently, it is impossible to determine the type of a packet without actually removing it from the buffer which implies that programs polling the network are likely to intercept unexpected packets by accident, and –having no other choice– will have to discard them.

Due to this lack of provision for an interleaved use of the network, one cannot operate different communication protocols in parallel or conduct several conversations of the same protocol at the same time. Needless to say, this is unacceptable for a concurrent environment, such as Concurrent Oberon, where it is desired to have multiple threads using the network simultaneously. In fact, this limitation is crucial even in the standard Oberon system, because it contributes to the fragility of its network services. For example, a machine which is already engaged in a network conversation reacts only to packets that belong to this exchange, and thus during its entire duration does not respond (or react) to packets sent by other clients. Also, installing two tasks that poll the network on the same machine does not work, because they will receive –and thus discard– each other's packets.

These inconveniences were eliminated by modifying the network device driver so that the network can be used by several activities concurrently to each other. This work is presented in this section.

The Notion of Protocols in the Network Driver

Since the network is to be used by several activities, it is a shared resource. But unlike the disk or memory, it is also an active component that generates input events without any explicit request of programs residing on the local machine. In other words, the role of the network software is twofold. It must forward data originating from different clients to the network, and it must separate incoming data destined for different local clients. As opposed to the multiplexing function which is realized simply by putting data on the wire, demultiplexing can be achieved only if the notion of clients is explicitly supported by the network software.

For this reason, the network driver was augmented with operations for installing and removing *Protocol* objects which embody the reactive parts of network clients that can be either applications or general purpose protocols. Protocol descriptors contain a procedure variable that accepts a packet header and a complete packet buffer (including the header) that must be initialized with the code responsible for handling packets prior installation of the protocol object.

```
DEFINITION SCCDriver;

  CONST
    HSize = 10; MaxDataSize = 512;    (*header and data sizes*)
    ProtoNotFnd = 96;                 (*reserved packet type*)

  TYPE
    Header = RECORD                   (*header of Ceres network*)
                ...
             END;

    Buf = POINTER TO ARRAY 524 OF SYSTEM.BYTE;    (*data begins at buf[HSize]*)

    Handler = PROCEDURE (VAR head: Header; VAR buf: Buf);

    Protocol = POINTER TO ProtocolDesc;
    ProtocolDesc = RECORD
                     typeL, typeH: SHORTINT;    (*type range; must be unique*)
                     handle: Handler;           (*packet processing code*)
                   END;

  PROCEDURE Install (p: Protocol);      (*install (initialized) protocol object*)
  PROCEDURE Remove (p: Protocol);    (*remove protocol object*)

  PROCEDURE SendPacket (VAR h: Header; VAR data: ARRAY OF SYSTEM.BYTE);
```

Each protocol is also associated with a range of packet types, denoted by its lowest and highest value, which is used by the device driver to determine the protocol object responsible to which a packet must be delivered when it arrives from the network. If no matching protocol object is found for a network packet, a reply bearing the reserved packet type *ProtoNotFnd* is sent back to the machine that generated the packet and passed to the corresponding protocol. This allows applications to cleanly differentiate between unresponsive machines and machines where the appropriate protocol objects have not been installed.

Protocol handlers can handle incoming data in two different ways. Received data can be copied, or the entire buffer can be swapped with an empty one (notice that the buffer is passed as a VAR parameter). Since the latter method requires merely a few pointer assignments, it can be used to achieve efficient forwarding of big data blocks across protocol borders. This is particularly important if several protocol layers are stacked on each other as this is typically the case for network architectures like the ARPA–net [McQuillan77] or the ISO open systems interconnect architecture [Zimmermann80]. Moreover, this approach allows buffer management to be introduced in a simple way at higher layers of the system, because individual protocols may swap, copy, or ignore incoming data according to their own policy.

Also, since buffers are assumed to be free when control returns to the network driver, they are immediately recycled. This means that more than one low level buffer is filled only if several packets arrive during packet processing. Since this is quite improbable and given that packet processing is fast, the resources of the network driver module can be kept very limited without risking overflows.

Implementation in Concurrent Oberon

The asynchronous event of a packet arrival is communicated to the system via interrupts. The dispatching mechanism executes as a high priority thread that essentially extends the network interrupt handler which is only responsible for reading incoming data into a packet buffer. Communication between the interrupt routine and the dispatcher thread is implemented using a traditional producer consumer scheme over a slotted ring of packet buffers:

interrupt handler

```
read data into buffer
IF not full THEN
  IF empty THEN resume dispatcher END
  insert buffer into free ring slot
END;
```

dispatcher thread

```
LOOP
  await(non empty)
  get next full ring slot
  find protocol & call handler
END;
```

The dispatcher, in an endless loop, removes packets from the ring and uses their type to find the corresponding entry in the protocol list. The packet is then passed to its destination by invoking the handler procedure of the corresponding protocol object, which processes the arrived packet and returns control to the dispatcher. When there are no more available packets, the dispatcher suspends itself. It is resumed again from within the network interrupt handler as soon as a new packet arrives.

Notifying network clients via upcalls (figure 2) resembles the mechanism used in the Oberon system to communicate input events to the target windows.

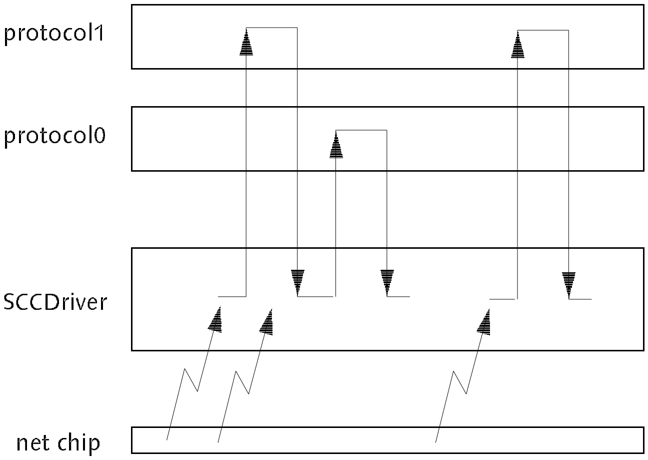

Figure 2 processing of arriving packets via upcalls

In this case, the scheme is used to perform packet processing which, as it has already been noted [Clark85], naturally fits into the upcall approach. Notification is indeed a very suitable method for implementing the reactive parts of protocols, because it intuitively reflects their state machine–oriented design. It also allows packet processing to be executed from within a single thread of control, thereby reducing the number of competing threads in the system without sacrificing responsiveness, since protocol handling is very fast and it is unlikely that several packets will reach the machine at the same time. Last but not least, acknowledgements to arriving packets can be generated directly from within the dispatcher thread, thereby guaranteeing fast response times, even though the machine may be heavily loaded. This does not only enhance the reliability of monitoring facilities, but also eliminates end–to–end synchronization between sending and receiving activities (threads), thereby minimizing the network delay experienced by applications.

Another possibility would have been to let each network client have its own thread polling for network packets and reacting to network events. This is typically the case in process–oriented designs [Requa85, Fenart86]. However, this overhead is unjustified if most clients are protocols with instantaneous reaction to network events. It also complicates the implementation, because elaborate synchronization and buffering is required to implement packet distribution correctly in the presence of concurrent polling.

Summary

In this chapter, we have described modifications and extensions to the Oberon system which provide the basic support for distributed programming. First of all, concurrency was introduced by augmenting the system with a single module which implements the essential programming primitives, and features a simple synchronization method that can be used to implement arbitrary customized synchronization and communication tools. To exploit concurrency with respect to networking, a modified version of the network driver supports simultaneous operation of multiple communication protocols. Unlike other process–oriented approaches, an efficient single threaded approach is used to distribute incoming network packets to their logical destinations. Also, fast forwarding of data blocks and flexible buffer management policies are supported by allowing application handlers to keep packet buffers. These changes promote the functionality of the Oberon system to a level comparable to that of other operating systems such as UNIX, yet without adding significant complexity.

Chapter 3 A General Data Transfer Facility

In this chapter, we introduce a general and extensible method for describing input and output of complex data. The proposed method proposed allows transfer operations of data types to be implemented without having to specify the medium where data is written on or read from, the format which is used for converting typed data such as integers and real numbers, or the method for resolving references. These functions can be introduced at a later point in time, even after the transfer code has been written. As a consequence, the same transfer code can be used to achieve persistence, i.e. to write data on disk, and to convey data through a communication channel. With our approach, it is also possible to program generic data transfers for instances of arbitrary data types even though their actual type is unknown at implementation time.

On Transfer of Data Structures

Transfer of complex data structures to and from disk is a typical operation for programs that want to achieve fault tolerance, or simply to preserve data across their executions to be used as input for other executions, even if the machine has been turned off in between. The complexity of data transfer, also referred to as *externalization* and *internalization*, depends on the kind of data that is involved. Storage and retrieval of typed data such as integers and reals is simple, because their values can be written and read directly as raw byte blocks. Externalization becomes difficult, however, if a data object contains dynamic structures. The reason is that pointers can be traced only in the context of their original address space, and therefore their actual value is of no use if the structure must be rebuilt in a different address space, or if the data transfer must have copy semantics. With pointers it is also possible to have aliasing and cycles, which complicates the transfer process even further.

Since the aforementioned problems have little to do with the fact that data is stored in memory or on disk, it is not surprising that they are encountered again in distributed systems where programs communicate data over a network; in this case, externalization and internalization are also called *marshalling* and *unmarshalling*, respectively. However, an important difference is that network communication involves cooperation between two different processes residing on different machines. This brings up the issue of robustness, because machines fail independently of each other and failure of one communication partner should not cause failure of the other. Further, the involved machines may have different internal data formats and different compiler implementations with the consequence that the values of standard data types and of records cannot be transmitted using their internal memory representation, but must be converted between the sender and the receiver. Notably, the same problem exists even in disk–oriented transfers, if the writer's environment may have different formatting conventions than the environment where data is read. For example, a file containing the externalization of a data structure may eventually be sent to another machine and used there as an input, possibly for the same program.

Given the importance of data transfer in a system, it is desirable to support it in a way that simplifies programming. The ideal solution would be to have a general primitive that takes the address of an object and performs the transfer, dealing with all necessary details without involving the programmer at all. Unfortunately, this cannot be done in a satisfactory way.

A major problem when considering transfer of complex data is the lack of extensive run–time type information about the contents of data items (records). For example, in Oberon, the address of a heap object suffices to determine its type, size, and the position of pointers inside it, but there is no information about the position and type of the rest of its contents. Consequently, it is impossible for a program to identify and handle the components of data objects individually, and thus data must be treated as an unstructured byte block. This approach is clearly insufficient though, because byte ordering and data arrangement within records is not the same across all hardware platforms and system implementations.

An additional difficultly is that the functionality and semantics of complex data types cannot be derived simply by looking at their data representation. Thus given the address of a data object, there is no way to train a program to make a selective approach, and the programmer it is left with no choice but to consider all components of a data type instance blindly. This is not always appropriate, because for some data types special actions may be required to preserve their semantic content or invariants. It is also likely that parts of a data type are redundant, hence should be ignored at externalization to reduce the amount of storage needed, and initialized according to the convention of the implementation when the data type is internalized again.

Therefore, generality cannot be achieved without involving in the transfer process the applications that implement the corresponding data objects. In other words, the application must be given the means to provide the programs that implement the data transfer with customized code that implements the required actions and which is called when instances of private complex data types are being externalized or internalized. The task of writing the externalization and internalization code can be simplified by using tools that generate this code automatically from a given data description or definition. Notably, this kind of support is purely auxiliary, and installing the appropriate code in the transfer programs still remains the programmer's responsibility.

But since the application is obliged to give the transfer code for its objects, it is desirable to keep this code sufficiently abstract so that it becomes a general applicable description of how data should be handled. Then, the same externalization and internalization code of a data type can be used for different kinds of transfers. It also becomes feasible to augment the system with programs that subject existing data objects to new transfers, despite the fact that the applications that define the corresponding data types have been implemented before these programs came into existence.

The IO Framework

According to these ideas, in Hermes, support for data transfer of complex data types is given in form of a module called IO (for input/output). Integration of application specific actions that are to be executed in the externalization and internalization process is achieved by introducing a special object type that has a type–bound procedure for transferring its contents. Applications must define their data types as an extensions of the base object type and implement the corresponding transfer procedures to handle the data type's contents. In other words, data items are modeled as intelligent objects with data transfer capabilities.

Writing and reading of data is supported via an object–oriented framework consisting of abstract *carrier*, *formatter*, and *linearizer* components with procedures for handling bytes, typed data, and pointers to complex data items, respectively. The provided types are merely the base types from which real component implementations are derived. Hence, they serve as stubs whose empty procedures can be used by the application to write the transfer code of data objects without specifying the concrete type of the components that are actually going to be used to handle the data. Binding of the code that implements the individual components to the application code occurs dynamically, at run time.

Functionally, each component presents itself as an abstract mechanism that can be used to write and read data items. This allows components to be arranged in a hierarchy according to the complexity of the data items they can handle. In fact, module IO explicitly introduces such a structuring by linking these components through a "has a" relationship so that operations of higher–level components can be implemented using the services of lower–level components (figure 1).

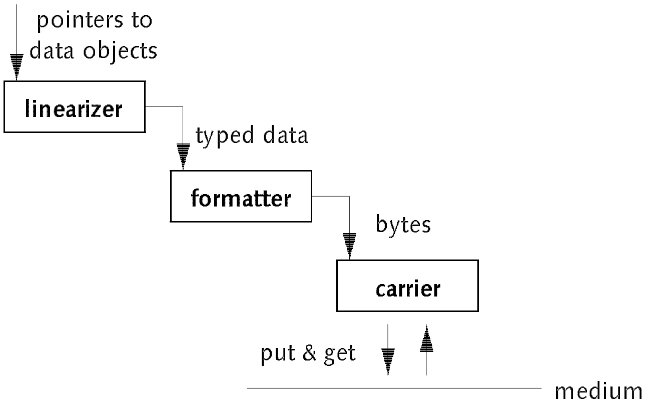

Figure 1 abstract data transfer components

Thus the linearizer, having the ability to handle pointers to data items, typed data (via a formatter component), as well as bytes (via a carrier component), is the right tool for transfering arbitrarily complex data structures.

Carriers

The data carrier where data is put in and retrieved from is modeled as an abstract uni–directional stream of bytes that can be accessed via sequential operations. This abstraction is introduced by defining a type called *Carrier* with type–bound procedures *Byte*, and *Bytes* for handling single bytes and byte blocks. The *writing* field of the carrier descriptor indicates the direction of the carrier, i.e. whether these operations write or read data.

```
DEFINITION IO;

  TYPE
    Carrier = POINTER TO CarrierDesc;
    CarrierDesc = RECORD
                    writing: BOOLEAN;
                    PROCEDURE (c: Carrier) Byte (VAR x: SYSTEM.BYTE);
                    PROCEDURE (c: Carrier) Bytes (VAR x: ARRAY OF SYSTEM.BYTE;
                                                  k: INTEGER);
                    PROCEDURE (c: Carrier) Synch ();
                    PROCEDURE (c: Carrier) Done (): BOOLEAN;
                  END;

    PosCarrier = POINTER TO PosCarrierDesc;
    PosCarrierDesc = RECORD (CarrierDesc)
                       PROCEDURE (c: PosCarrier) Pos (): LONGINT;
                       PROCEDURE (c: PosCarrier) Set (pos: LONGINT);
                     END;
```

Procedure *Synch* serves as a provision for allowing synchronization points to be injected in the data stream. While data carrier implementations may leave this procedure empty, it can be overwritten to put a special byte pattern in the stream (when writing) and to check against this pattern (when reading), or to implement a confirmation protocol when transfer occurs over a communication channel. Checks of this kind are mandatory to detect failures that occur due to inconsistencies between the code writing and the code reading data.

An additional procedure *Done* indicating the success or failure of the previous operations is provided. Failures are possible due to synchronization errors or failures of the medium, for example a network connection that can break down any time. The ability to check the state of the carrier is indispensable for tolerating errors when values received during internalization influence the internalization process itself, since in this case failure of the data carrier to deliver data correctly can affect robustness of the application. Below is is shown how to implement (writing and) reading an array of bytes so that an error of the carrier cannot cause an infinite loop. It is assumed that a special terminator signals the end of the array.

```
i := 0; c.Byte(a[i]);
WHILE c.Done() & (a[i] # Terminator) DO
   INC(i); c.Byte(a[i])
END;
```

The notion of positionable carriers is introduced via a subtype of Carrier called *PosCarrier* to describe special stream implementations that allow the insertion and removal point to be re–adjusted. This is to encourage implementors of carriers to choose the closest possible type as a base type for their implementations, because it may be possible (and desirable) to exploit the special properties of a carrier when programming the transfer code. The additional functionality is declared via a procedure *Set* for repositioning the carrier and a procedure *Pos* that returns its actual position, in numbers of bytes, within the stream. Positionable carriers have similar functionality to riders, a mechanism that is used in Oberon to access the contents of files, but are defined in a more general way so that they can be used to describe any kind of positionable stream.

Formatters

Handling of typed data is described via a *Formatter* type with procedures for converting standard types of the Oberon programming language to and from a sequence of bytes, respectively. Each formatter object also contains a carrier object so that concrete formatting operations are coded using abstract carrier operations. The direction of the formatter operations is determined by the direction of the included carrier.

```
  DEFINITION IO;

  TYPE
    Formatter = POINTER TO FormatterDesc;
    FormatterDesc = RECORD
                      c: Carrier;
                      PROCEDURE (f: Formatter) Bool (VAR b: BOOLEAN);
                      PROCEDURE (f: Formatter) Char( VAR ch: CHAR);
                      PROCEDURE (f: Fomatter) Int (VAR i: INTEGER);
                      ...
                      PROCEDURE (f: Formatter) Done (): BOOLEAN;
                    END;
```

Since formatting is a higher–level function than writing and reading bytes, the fact that the data carrier operations have been executed successfully does not imply that formatting has also succeeded. For this reason, the formatter has its own procedure *Done* indicating the result of its operations, which can be used to achieve robustness against failures in a similar way this is done for the carrier.

This separation underlines the independence between the formatting process and the process of storing and retrieving data, thereby allowing arbitrary combinations thereof. If formatting functions were directly coupled to the data carrier component, new versions of all existing formatting methods would have to be produced for each new data carrier. Conversely, introducing a new formatting convention would then result in updating and recompiling all existing data carrier implementations.

Linearizers and Data Objects

In accordance to this pattern, a third type, the *Linearizer*, captures the task of handling pointers. Hence, the linearizer must handle references, thereby converting dynamic data structures into a linear form from which they can be reconstructed again at a later point in time.

```
DEFINITION IO;

  TYPE
    Linearizer = POINTER TO LinearizerDesc;
    LinearizerDesc = RECORD
                      f: Formatter;
                      PROCEDURE (l: Linearizer) Obj (VAR o: Obj);
                      PROCEDURE (l: Linearizer) Done (): BOOLEAN;
                    END;

    Obj = POINTER TO ObjDesc;
    ObjDesc = RECORD
                PROCEDURE (VAR o: ObjDesc) Transfer (l: Linearizer);
              END;
```

Analogously to above, each linearizer object contains a formatter object, and has a type–bound procedure *Obj* for writing and reading pointers to data items depending on the direction of the carrier. Also, similar to carriers and formatters, linearizers have a procedure *Done* yielding the result of the *Obj* operation. Data items are modeled as objects of type *Obj* with a procedure *Transfer* for externalizing and internalizing their contents using a linearizer. Hence, the actual data transfer is performed by the application code, rather than an automatic mechanism.

Even though the roles of a linearizer and a data object seem alike, the two components serve entirely different purposes. The linearizer embodies the context of a transfer as well as the mechanism that produces type information (when writing) and uses it (when reading) to create object instances. On the contrary, the transfer code of a data object is simply responsible for handling its contents. The necessity of separating these tasks becomes most evident when considering internalization, since in this case an object instance must already be available in order to invoke the corresponding transfer code. Moreover, since the linearizer component is passed from one object to another as a parameter of the corresponding transfer procedures, the transfer state is preserved across individual invocations of the application code so that externalization and internalization decisions can be taken depending on it. As it will be shown in the next section, carrying and consistently updating state is necessary to deal with data structures with aliasing.

Programming Transferable Data Types

Implementation of the transfer procedures is straightforward and does not complicate the structure of applications, because it can be put near the corresponding type declarations isolated from the rest of the program. While it would be possible to provide the users with a tool that generates code from a given type definition, this is hardly required given the simplicity of this programming task.

An application data type with externalization and internalization capability is defined as an extension of the *IO.ObjDesc* type, and its transfer procedure is programmed to handle the newly added fields. Record fields that are not handled by the transfer procedure are not transfered. The descriptor fields need not to be handled in the order they appear inside the record. As an example, we show the implementation of a list element containing an integer and a pointer to the next element, and extension thereof that has an additional boolean field:

```
TYPE
   Elem = POINTER TO ElemDesc;
   ElemDesc = RECORD (IO.ObjDesc)
                  nxt: IO.Obj;       (*pointer to next list element*)
                  key: LONGINT    (*key of element*)
                END;
```

```
     ExtElem = POINTER TO ExtElemDesc;
     ExtElemDesc = RECORD (ElemDesc)
                    b: BOOLEAN
                  END;


  PROCEDURE (VAR e: ElemDesc) Transfer (l: IO.Linearizer);
  BEGIN
     l.f.LInt(e.key);   (*handle long integer value by calling formatter procedure*)
     l.Obj(e.nxt)       (*handle object value by calling linearizer procedure*)
  END Transfer;

  PROCEDURE (VAR e: ExtElemDesc) Transfer (l: IO.Linearizer);
  BEGIN
     e.Transfer↑(l);    (*handle base type fields with supercall*)
     l.f.Bool(e.b)      (*handle boolean value by calling formatter procedure*)
  END Transfer;
```

New data objects can be defined as extensions of existing objects, even if these have been implemented as opaque data types. The contents of the base record type can be handled via a supercall (indicated by ↑), i.e. by invoking the transfer procedure bound to the base type.

Since the linearizer procedure Obj expects an object pointer of type IO.Obj as a reference parameter, it cannot be called with extended pointer types. Hence, if a data structure contains pointers to extensions of the IO.ObjDesc type, these have to be handled by introducing an auxiliary variable of type IO.Obj. When writing, the value of the pointer in question must be assigned to this variable which can then be used to write the value of the pointer. Conversely, when reading, the pointer value must first be obtained using the auxiliary variable, and then assigned to the extended pointer via a type guard.

This inconvenience is caused by the strong type checking of Oberon that requires pointer reference parameters to have the identical type as the formal parameter. This problem could only be solved by introducing a universal pointer type which is compatible with all possible pointer types. Although this type essentially exists in Oberon (*SYSTEM.PTR*), the compiler handles reference parameters of this type as OUT parameters, in fact, their original value is overwritten with type information. This is clearly not appropriate when the linearizer is externalizing a data type since in this case the pointer passed to the Obj procedure is used as an IN parameter.

Default Component Implementations

While it is likely that data transfering applications will employ their own carrier component to implement the transfer (e.g. a file for saving data on disk, or a network connection for transmitting data from one machine to another), each application will hardly require a customized formatting convention, or an own method for resolving references. With this motivation, module IO also provides default formatter and linearizer implementations to simplify programming of data transfering applications.

The Default Formatter

The default formatter arranges data according to the internal format of the Ceres workstation. In other words, its implementation on the Ceres simply handles typed data as byte blocks and does not perform any conversions. Implementations for other hardware platforms may need to swap bytes in order to achieve compatibility.

Even though formatters are responsible for handling typed data on top of a stream of uninterpreted bytes, in order to reduce the amount of data produced the default implementation does not tag the produced data. Considering the fact that the most commonly used typed items of the Oberon programming language consist of only 2–4 bytes, this yields significant savings (figure 2).

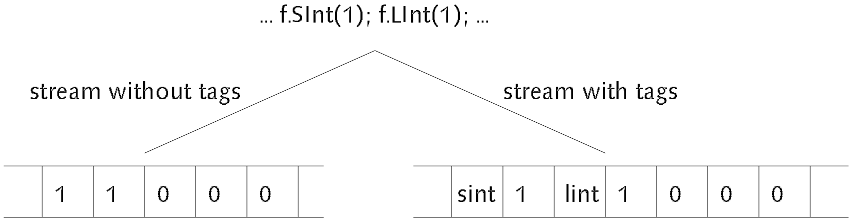

Figure 2 tagged vs. non–tagged output

A consequence of not using tags is that when reading it cannot be checked whether the incoming data actually corresponds to the items that are to be read. For example, it is possible to read a long integer first, and then a short integer out of the non–tagged stream of figure 2, despite the fact that this data was produced by writing a short integer followed by a long integer. Since the application can guarantee correctness by ensuring that the producer and consumer programs are implemented symmetrically, which is trivial given that in most cases the same code can be used both for writing and reading a data type, this limitation is not problematic. In fact, even with tags, errors at the application level cannot be detected, because it is still possible to swap the values of two data items of the same type if they are read in the reverse order they are written.

The Default Linearizer

Analogously to the default formatter, the default linearizer implements the Obj operation to linearize and delinearize pointers to data objects. Besides generating the type information that is required to create instances of data objects at internalization, the implementation handles aliasing and cycles. Also, motivated by the fact that type information is typically several bytes long, a simple mechanism is used to avoid replicating information that has already been generated during a transfer.

The primary task of the linearizer, namely to produce type information and use it to generate object instances at internalization, is accomplished by using the meta–programming facilities of the Oberon system. At externalization, a handle to the type of the data object is retrieved and used to write the corresponding module and type name on the data stream. This type information is used at internalization, first to obtain a handle to the corresponding module, then to get a handle corresponding to the right type, and finally to create an empty object instance.

Although complex data structures can consists of numerous nodes, typically these are instances of only a few different data types. For this reason, the same type information is never written twice during the transfer process, and known types are referenced via shorter identifiers. This is achieved by maintaining a type cache as follows:

write type info

```
t := Types.TypeOf(obj);
search in type cache
IF found THEN
    write type identifier
ELSE
    write new identifier
    write t.mod and t.type names
    insert type handle t in cache
END;
```

read type info & create instance

```
read type identifier
IF new THEN
    search for type handle in cache
ELSE
    read mod and type names
    m := Modules.ThisMod(mod);
    t := Types.This(m, type);
    insert type handle t in cache
END;
Types.NewObj(obj, t);
```

When writing, the contents of the cache are checked before producing type information to determine whether this data has already been written. If this is not the case, the type is given a unique identification number, it is inserted in the cache, and its identifier is written along with the type information. If the type is found in the cache, only its identification is written. Type identifiers are obtained by increasing a sequence number so that, at internalization, the value of a received identifier directly indicates whether the type is new. If this is indeed so, the type information is read and a type handle is created, else the appropriate type handle is retrieved by searching the type cache.

Once the type information corresponding to an object is written, or an empty instance of a data object is created, externalization, respectively internalization is continued by handling the object's contents with a call to its transfer procedure. To reduce the probability of having undetected inconsistencies in the transfer code of the application objects, the Synch procedure of the carrier is called before and after invoking the transfer procedure of an object. Provided that the actual carrier implementation has not left this procedure empty, this approach catches all programming errors whose effect is that the amount of data produced is not equal to the amount of data consumed.

When externalizing (and internalizing) data structures with aliasing, i.e. with objects that are reachable via several paths, the application code will encounter the shared objects more than once via repeated calls to the Obj procedure of the linearizer. However, shared objects should be externalized (and internalized) exactly once and aliasing must be reestablished in its original form when the data structure is reconstructed again. For this purpose, an object list is maintained in a similar way this is done for the type cache:

```
externalize object                          internalize object

search in object list                       read object identifier
IF found THEN                               IF new THEN
    write object identifier                     find object in list
ELSE                                        ELSE
    write new object identifier
    write type info                             read type info & create obj instance
    insert object in list                       insert object in list
    c.Synch(); obj.Transfer(l); c.Synch()       c.Synch(); obj.Transfer(l); c.Synch()
END;                                        END;
```

Transfer of object's contents is done only once for each object, when it is encountered for the first time. Cycles are automatically handled, because objects are inserted in the list prior to writing their contents so that recursive references to these objects are resolved correctly.

The object list and the type cache are reset with a call to procedure *Reset*; initialization of the type cache occurs only if this is explicitly specified. The object list must be initialized before each externalization and internalization. By contrast, the type cache can be preserved across several transfers which is useful if an application must transfer a set of objects rather than one data object. Using the same type cache for transfering different data structures strongly binds them together, because failure to internalize a single object will hinder internalization of the rest.

An example

To illustrate the way these mechanisms work, an example of a dynamic inhomogeneous structure using the previously introduced list data types and a pictorial representation of its externalization is given in figure 3.

Since the structure is traversed recursively using the application code to trace references, data objects are externalized in a nested fashion, i.e. the third data object is enclosed within the second, which in turn is enclosed within the root of the structure (for readability, the points within the data stream where the

Synch procedure of the carrier is called are marked with the character "@").

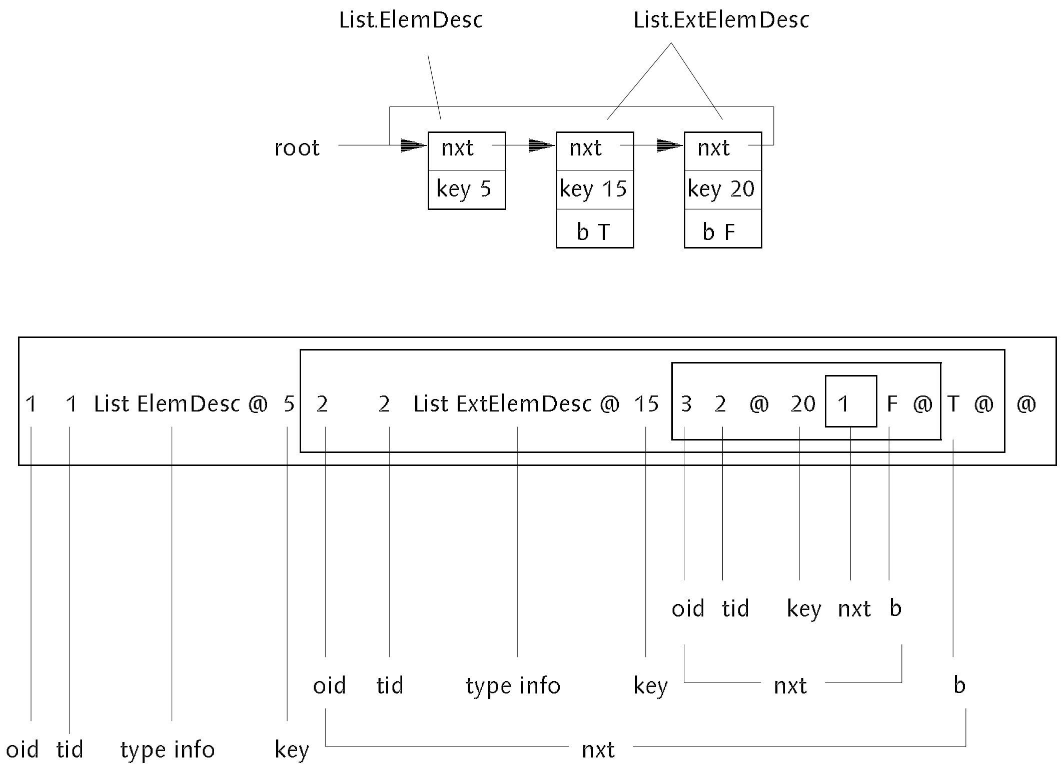
List.ElemDesc
List.ExtElemDesc
root
nxt
key 5
nxt
key 15
b T
nxt
key 20
b F
1 1 List ElemDesc @ 5 2 2 List ExtElemDesc @ 15 3 2 @ 20 1 F @ T @ @
oid tid key nxt b
oid tid type info key nxt b
oid tid type info key nxt

Figure 3 an inhomogeneous ring structure and its externalization

Despite their complexity, linearizations are systematically unraveled as the nodes of the data structures are created and initialized driven by the internalization code.

Failures

Obviously, having the application involved in the transfer process implies that the application code is mandatory for reconstructing a data structure. This is a problem if an attempt is made to reconstruct a data structure when one or several application modules that implement some of its nodes are not available. In Oberon, this is indeed possible, because applications may use (and thus transfer) objects of unknown type which are implemented in modules that are higher in the module hierarchy. Another failure possibility are synchronization errors that occur if the code that reads data is incompatible with the code that produced it.

If a failure occurs during internalization, then reconstruction of the data structure is aborted and the state of the linearizer is set accordingly so that further calls to procedure Done will return FALSE. In principle, for the special case where an object instance cannot be created due to missing application code, it would be possible to set the corresponding pointer to NIL and ignoring the object data. This automatic repairing mechanism is questionable, because it becomes impossible to differentiate between genuine and artificial NIL pointers. Also, a partially reconstructed data structure is of little use unless the application anticipates such errors.

In the latter case, where critical objects, whose code may be missing at internalization, are known in advance, the application programmer can handle failures in a simple way. The critical objects have to be separated from the main data structure, and transfered using a fresh transfer context so that internalization failures are contained and do not affect internalization of the main data structure. Further, it must be guaranteed that data produced by externalizing critical objects can be skipped in case internalization of the critical objects fail (figure 4).

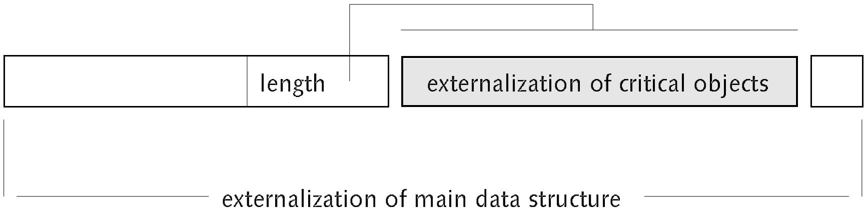
length
externalization of critical objects
externalization of main data structure

Figure 4 externalization with critical objects

This is particularly easy to achieve if the data carrier used to transfer the main data structure is a positionable carrier, because then the same carrier can be used to externalize the critical objects. The length of the externalization can be back–patched by repositioning the carrier. If a normal carrier is used instead, the critical objects have to be externalized on another medium (e.g. a file) and the length along with the data of the critical externalization must be copied on the main carrier.

Critical objects that share other objects must be transfered within the same transfer context, otherwise the aliasing information is lost and a proper reconstruction of the data structure is not possible. This limitation has proved to be of little practical importance since separate transfer contexts are intentionally used to handle parts of the data structure that are independent of each other and thus can be reconstructed atomically without affecting the main internalization process. Hence, aliasing across different transfer contexts is not very meaningful.

Related Work

The CLU language adopts a similar approach to ours for converting values of abstract types into intermediate representations [Herlihy82], where special encoding and decoding procedures must be supplied by the programmer for each data type. However, in CLU these procedures are used especially for putting values of data types into messages which are then sent over the network and used at the destination to reconstruct the data type. This feature is built in the language itself, and thus different methods of formatting, and transfer cannot be added in a simple way. In addition, CLU is not object–oriented, i.e. variables cannot hold objects of extended types. This considerably simplifies implementation of the conversion code, because the types of dynamic objects contained in data types are known already at implementation time. Since programs cannot possibly hold objects of unknown type, there are also no failures due to missing code.

Modula–3 also has a special mechanism named "pickles" to marshal and unmarshal complex data [Kalsow88]. The produced data can be stored on disk or used directly as input for other programs. However, pickling involves non–sequential accesses to the input and output streams and is therefore unsuitable for use with network protocols. Also, in contrast to our approach, marshalling is automatic, i.e. references are traversed using available run–time information, and values that are not pointers are put in a pickle "as is", hence data is handled using directly its internal memory representation. It is possible to install code that introduces customized handling, but unlike in our system this is an exception rather than the rule.

Contributions related to the problem of marshalling data structures have also been made by remote procedure call systems (RPC) [Bershad87, Birell84]. RPC systems support marshalling via special tools called stub generators that generate the data transfer and communication code automatically from a given specification or interface definition. In some cases, the programmer may influence marshalling either by installing custom–made transfer procedures that are invoked during externalization and internalization process, or by feeding the generators with code fragments that are patched into the produced code. The data conversion strategy is typically built in the code generators so that data conversion can be implemented using in–line code, rather than procedure calls. However, this means that the data format used cannot be changed without modification of the generator program.

The Sun external data representation library (XDR) [Sun90] also supports conversion of data structures into a fixed machine independent representation and vice versa. Although XDR is primarily used by Sun's RPC system as a protocol for transmitting parameters, it is possible to write and read data using different steam implementations. Similar to our approach, the direction of the transfer can be specified when creating a stream, and the transfer procedures for complex data structures must be implemented by the programmer. However, the transfer code can be generated automatically using special tools. An important difference is that when using XDR the transfer code must be provided as a parameter when starting externalization, with the consequence that the data objects to be handled must be known a priori, when implementing the program that performs the transfer. Also, internalization simply produces a pointer whose type cannot be determined or verified, hence typing errors remain virtually undetected.

Finally, our work has great similarities with other object–oriented systems like DB++ [Schmidt89], ETHOS [Szyperski92], and Oberon–3 [Gutknecht93], since in these systems data items are also implemented as objects with transfer procedures. DB++ and Oberon–3 focus on object persistence and thus are strictly file–oriented. In addition, these systems arrange persistent objects in collections that can be stored and read using special operations; hence, the unit of persistence is a collection not a single object. On the other hand, the ETHOS system provides a similar framework as ours, but objects have separate externalization and internalization procedures. Also, ETHOS restricts transfer of structures for positionable streams only, and there is no provision for supporting different formatting methods. Our design has also been an inspiration for Oberon–PVM [Bougnion94], that extends the PVM package to support object–oriented distributed programming in Oberon, and uses virtually the same approach to transfer data over a network.

Summary

We have presented a framework for describing transfer of complex data structures. Our approach can be viewed as a generalization of known methods that are integrated in remote procedure call mechanisms or are part of persistent object systems. Namely, in our system it is possible to combine arbitrary carrier, formatter and linearizer components with each other, and to develop such components independently of the applications that implement data objects. Also, the ability to externalize data structures directly on non–positionable streams eliminates copying and allows data objects to be directly transfered over communication channels with bounded buffering capability such as network transport protocols. Finally, due to the object–oriented design of our system, and the provision for checking consistency of the application code, generic transfer programs can be implemented in a type safe and robust way even though the actual type of the data objects that will be passed as parameters to these programs is not known at system implementation time.

blank

Chapter 4 The Message Passing Facility

This chapter describes a facility that supports interprocess communication via message passing primitives. Messages are conventional data types with structured contents, yet still can be exchanged over the network as if they were indivisible data items. Marshalling, data transmission, and unmarshalling occur transparently to the application in a type–safe way. As a consequence, conversion and consistency checks at the application level are completely eliminated, and the content of a message can be directly accessed as soon as it is received. Messages are sent to special queues, called mailboxes, that are uniquely identified throughout the entire system. Communication is asynchronous, meaning that the sender continues execution after sending a message and needs not to wait until the message is received or processed. Failure notification is supported via special messages that are sent to the sender when a transmission fails. There is also support for monitoring mailboxes in order to detect failures that occur during message processing. Data about the performance of our implementation can be found in the appendix.

Towards an Abstract Message Model

Interprocess communication is one of the key elements of multi–process systems, since it provides the means for data exchange between processes despite the fact that they may reside in separate address spaces. This is even more important in distributed systems where processes are located on different machines and therefore must cooperate over a network.

Message passing is a particularly convenient communication paradigm, because it embodies a model of interaction between processes that naturally fits into a distributed programming environment. Describing communication in form of message exchanges automatically decouples execution of the cooperating processes which is of outmost importance in a distributed system [Liskov86]. Indeed, parallelism is not only an inherent feature of distributed protocols that involve cooperations among several nodes in the system, but is also indispensable for enhancing efficiency of computations that can be performed concurrently. Also, message passing is a flexible approach, because its primitives can be combined without much effort to implement synchronous communication patterns that are typical for traditional client–server architectures.

However, message based communication in its most primitive form, as it is implemented by the network hardware, has profound limitations. Since network packets are small, large application message must be divided into several packets which must be sent in the correct order over the network and reassembled again at the destination. Moreover, transmission at the hardware level is only done in a best effort basis, and thus it cannot be guaranteed that data will indeed arrive at its destination. Last but not least, while the ability to send and receive byte blocks (packets) is the essence of interprocess communication, in practice, higher data abstractions are required, because programs are written in high level languages and use typed and structured data items.

In other words, only a high level message abstraction can make message passing truly attractive for implementing conversations at the application level. The introduction of "long" messages that are transparently cut into and assembled from several network packets, respectively, is definitely a step in this direction, but does not relieve the programmer from having to convert data into messages and vice versa. We believe that it is appropriate to allow application messages to be defined just like ordinary data types, and thus be arbitrarily complex objects whose contents can be accessed directly without any need for conversion. The network should consequently present itself as a special medium with the ability to handle such abstract messages in a single piece, i.e. to reliably convey them between sender and receiver while preserving their internal structure. Analogous mechanisms should be provided to communicate violation of this abstraction to the application, for example when the underlying network software fails to transmit data to its destination.

The Communication Primitives

Module Msgs implements communication in this spirit. It provides asynchronous message passing primitives that can be used to exchange entire data structures over the network. In addition, there is support for monitoring destinations to detect failures. The key to the implementation of this communication facility is that messages are defined as transferable data items with corresponding transfer procedures (see chapter 3). Thus message transmission is a special case of the previously described externalization and internalization scheme, for the case where data is sent over the network.

Mailboxes

Messages are not sent directly to application threads, instead they are deposited in special queues called mailboxes. Mailboxes serve as communication end–points identifying programs that cooperate with each other over a network, and are identified via addresses that are unique for the entire network and remain valid over shutdowns and crashes. This allows applications to detect failures reliably even if monitoring is slow compared to the time needed to restart the system. It also means that addresses can be reused after a shutdown which is especially useful for persistent programs that are restarted when the machine is turned on, and where it is desirable to use a single address for their entire lifetime.

```
DEFINITION Msgs;

  TYPE
    Adr = RECORD
           netadr: SHORTINT;
           mbxno: LONGINT
         END;

    Mbx = POINTER TO MbxDesc;
    MbxDesc = RECORD
                adr: Adr;      (*read–only*)
                avl: LONGINT   (*read–only*)
              END;

  PROCEDURE Open (mbx: Mbx; mbxno: LONGINT);
  PROCEDURE Close (mbx: Mbx);
```

A mailbox address consists of the network address of the machine where the mailbox resides and a mailbox number that identifies the mailbox within the address space of the host machine. Mailbox numbers are obtained by incrementing a 32–bit sequence number that is kept in stable storage to preserve its value despite failures. This guarantees uniqueness of mailbox addresses, since network addresses are unique and mailbox numbers are never duplicated.

To register a mailbox in the system's directory, a mailbox is opened with a call to procedure *Open* that accepts a mailbox number as a parameter. If the specified mailbox number is equal to 0, then a new mailbox number is generated, otherwise the mailbox address is initialized using this value. The address of the mailbox is recorded in the *adr* field of its descriptor. The mailbox numbers assigned to mailbox addresses by the system are always positive. Negative values are reserved for system services that must be publically accessible; they must be distributed by an administrator and specified on mailbox creation. When a mailbox is not needed anymore, it is explicitly disposed by procedure *Close*. Mailbox descriptors also contain an *avl* field indicating the number of messages held by the mailbox which is initially set to 0 when the mailbox is opened.

Sending and Receiving Messages

Transmission of messages is triggered by the *Send* operation that accepts a message and the address of the destination mailbox as parameters. Sending is non–blocking in the sense that the sender does not wait until the message arrives at or is retrieved from the destination mailbox, but continues execution after the message has been successfully transmitted over the network. This decoupling between sender and receiver allows concurrency to be introduced in a natural and straightforward way without requiring application programmers to create a separate thread for each individual message transmission.

```
DEFINITION Msgs;

  TYPE
    Msg = POINTER TO MsgDesc;
    MsgDesc = RECORD (IO.ObjDesc)
                  nfyadr: Adr    (*address to which error message should be sent*)
                END;

    NotifyProc = PROCEDURE (mbx: Mbx);

  PROCEDURE Send (VAR dst: Adr; msg: Msg);
  PROCEDURE Receive (mbx: Mbx; VAR msg: Msg);
  PROCEDURE InstallMsgNotifier (mbx: Mbx; notifier: NotifyProc; prio: SHORTINT);
```

The specified mailbox address is used to contact the corresponding host and to locate the destination mailbox. Messages arriving from the network are inserted in mailboxes in the order they are received. Our implementation guarantees that messages sent by the same application (thread) to the same mailbox will be placed in the mailbox queue in the order the were sent. Messages that originate from different activities are not ordered.

Messages that have been placed in a mailbox can be retrieved with calls to procedure *Receive*. This operation returns the first message of the mailbox queue. If the queue is empty, the caller is blocked until a message is deposited in the mailbox. Checking the number of available messages prior to attempting to withdraw a message allows blocking to be avoided.

Programming with messages is strongly event–oriented, i.e. when receiving a message, its type is used to decide which actions to take next:

```
Receive(mbx, msg);
IF msg IS MyMsgType0 THEN
    handle messages of type MyMsgType0
ELSIF msg IS MyMsgType1 THEN
    handle messages of type MyMsgType1
ELSE
    handle other messages
END;
```

This is particularly appropriate for describing distributed programs such as asynchronous algorithms or general purpose servers. Since messages are instances of conventional data types, their actual type can be determined by a type test. This scheme relies solely on the runtime support of the Oberon programming language and therefore it is not possible to have programming errors that are due to incorrect management of type information at the application level. Also, the fact that messages are structured items whose contents are directly accessible simplifies the application, because there is no need to extract data out of messages via special sequential access operations.

To enhance flexibility, an additional procedure *InstallMsgNotifier* is provided that can be used to associate a mailbox with a procedure which is invoked as soon as the mailbox becomes non–empty. Invocation occurs immediately if the mailbox already contains messages when this operation is called. The notifier procedure is invoked from within a thread that is created especially for this purpose and whose priority can be specified when installing the procedure. Notification occurs only once, for the first message that is deposited in the mailbox, hence if repeated notification is desired, the notifier procedure must be re–installed. This feature allows applications to be implemented in a pure event–oriented way without the need to maintain a thread solely for the purpose of polling the mailbox.

Failure Detection

Although the communication abstraction offered to the programmer is that of a network which reliably conveys messages to their destinations, message transmission and delivery may not be accomplished due to failures that cannot be repaired by the underlying communication software. These include node crashes or failure to create a message instance at the destination machine. It is also possible that a message cannot be delivered to the application, because it was sent to an address for which no registered mailbox exists.

To inform the application about such failures, messages contain a special *nfyadr* field containing a mailbox address. This address is used by the communication facility to notify the application when failures occur by sending it special error messages. If no notification is desired, then this address must be set equal to *nulladr*. Error messages are extensions of the base message type and come with an additional field indicating the cause of the failure while their nfyadr field is initialized to contain the mailbox address specified as a destination in the failed transmission.

```
DEFINITION Msgs;

   CONST
      NetErr = 1; ConfigErr = 2; MbxErr = 3; IOErr = 4;    (*error codes*)

   TYPE
      ErrMsg = POINTER TO ErrMsgDesc;
      ErrMsgDesc = RECORD (MsgDesc)
                          err: SHORTINT     (*error code*)
                       END;

      Mon = POINTER TO MonDesc;
      MonDesc = RECORD
                       dst: Adr;           (*destination to be monitored*)
                       secs: LONGINT;    (*polling period*)
                       msg: Msg            (*monitor message; nfyadr must be properly set*)
                    END;

   VAR nulladr: Adr;

   PROCEDURE StartMon (mon: Mon);
   PROCEDURE StopMon (mon: Mon);
```

For simplicity, we differentiate only between four types of failures. A network error is returned if the remote partner fails to respond to successive communication attempts, in which case the machine is presumably turned off or has crashed. Configuration errors indicate that module Msgs is not loaded on the corresponding destinations, whereas a mailbox error is raised if the specified mailbox address is invalid. Finally, an input/output error denotes failure of the destination to internalize a message, either due to missing application modules, or because there are inconsistencies between the externalization and internalization code.

This provision, however, suffices only if a failure coincides with transmission of an application message, hence failures that occur at later points in time, after the message has been successfully deposited in the specified mailbox, remain undetected. While the application can, in principle, assume a failure by the lack of an expected reply, the response time to application messages may vary significantly and is therefore difficult to predict. As opposed to this technique, failure detection can be achieved in a reliable way by sending monitoring messages to the desired destination, so that in case of failure the application will be automatically notified by the communication system as described above.

The task of sending messages essentially for the purpose of determining liveness of a destination is supported by introducing special monitor objects. Before starting a monitor object via procedure *StartMon*, its descriptor must be initialized with the address to be monitored, a monitoring message that will be sent periodically to the specified address, and the time (in seconds) that must elapse between two successive monitoring attempts. If a failure is indeed detected, the application is sent a corresponding notification message. Monitor objects must be explicitly stopped with a call to procedure *StopMon* when the state of the remote activity becomes of no interest.

Notifying the application in form of messages nicely integrates failure detection in the message–oriented communication paradigm. Also, even though failures and thus generation of notification messages occurs asynchronously to the application, handling of failures can be done synchronously to the application activity and in the same style as for ordinary messages.

Notably, it would have been possible to directly communicate failures to the application by augmenting the Send operation with an additional result parameter. The disadvantage of this scheme is that the sender must remain blocked throughout the entire transmission process and until the message is deposited into the destination mailbox, which unnecessarily restricts the implementation precluding asynchronous approaches. In addition, this would lead to having two different failure handling styles, one for failures occuring during message transmission, and one for failures detected by monitors.

Message Transmission

For each machine pair, message transmission is performed by two cooperating threads, one devoted to sending and one to receiving data over the network. The sending thread resides on the machine of the application that invoked the Send operation and the receiver on the machine of the destination mailbox. Both the sender and the receiver threads use the default linearizer and formatter implementations of module IO described in chapter 3 to implement message transfer. A transport protocol that reliably conveys data over the network serves as a data carrier coupling the two threads in a producer–consumer fashion. In other words, the message is externalized directly on the network by the sender while the receiver uses the incoming data to reconstruct an exact copy of the message at the destination.

The Network Carrier Protocol

The abstraction of a reliable uni–directional byte stream is implemented in module NetCarriers as a special protocol on top of the network device driver. A stream is essentially a communication channel coupling a sender and a receiver in a producer–consumer fashion. Each stream end is conceptually associated with a pointer indicating the location of the next byte to be inserted or retrieved, depending on whether the carrier is a sender or receiver. The pointer is advanced –but never decreased– by the corresponding access operations.

Hence, neither the sender nor the receiver may go backwards on a network stream. Transmission is reliable, i.e. it succeeds unless the machine of a carrier fails in which case the state of the other carrier is set accordingly to indicate this failure and to suppress further access operations.

The sending and receiving ends of a network stream are both implemented as extensions of the base carrier type and allow the stream to be accessed via their Byte and Bytes procedures. In other words, the sender uses the sending carrier to directly externalize a data object on the network and the receiver uses the corresponding receiving carrier to pick up incoming data and reconstruct an exact copy thereof at the destination.

```
DEFINITION NetCarriers;

  CONST
    RespErr = –1; Ok = 0; PeerErr = 1; SynchErr = 2; ProtoErr = 3;

  TYPE
    Carrier = POINTER TO CarrierDesc;
    CarrierDesc = RECORD (IO.CarrierDesc)
                    port: INTEGER;      (*read–only*)
                  END;

  NotifyProc = PROCEDURE (c: Carrier);

  PROCEDURE InitSender (c: Carrier; adr: SHORTINT; port: INTEGER);
  PROCEDURE InitReceiver (c: Carrier; port: INTEGER);
  PROCEDURE Purge (c: Carrier);

  PROCEDURE InstallTPhaseNotifier (c: Carrier; notify: NotifyProc);
  PROCEDURE TPhaseSynch (c: Carrier; VAR res: SHORTINT);
```

Since network streams are intended for hosting transfer of data objects according to the scheme presented in chapter 3, data travelling on network streams are not only produced by the (presumably correct) communication software, but also by application code implementing the transfer procedures of the corresponding data objects. For this reason, there is support for detecting and tolerating synchronization failures. Network carriers implement the Synch procedure so that synchronization errors due to inconsistencies between the externalization and the internalization code are detected. No special actions are needed if an object instance cannot be created at the destination machine since this immediately results in a synchronization error. Synchronization errors lead to a graceful termination of the corresponding transfer so that the stream can be used for further transfers after such a failure.

Creating and Purging Carriers

Before data transfer can commence, a sending and a receiving carrier is initialized with calls to procedures *InitSender* and *InitReceiver*, respectively. Binding between the two carrier ends and thus establishment of a stream is achieved by the use of identification numbers called ports. When initializing a receiver carrier, the port number under which the stream is to be registered may be directly supplied. If the specified number is equal to 0, a free port number is chosen by the system. Both operations are non–blocking and simply initialize the internal state variables of the carrier and register it so that it can be reached via the network.

Binding does not involve any packet transmissions at all. This is in contrast to 3–way handshake protocols [Tomlinson75] which are typically employed by other transport implementations (e.g. TCP [Cerf74]) to select the initial sequence numbers of a transport session. Our implementation avoids packet delivery problems by the use of unique stream identifiers and sequence numbers indicating the amount of data packets that have been sent in a given stream. A bind request is therefore implicitly present whenever a data packet is received with a new stream identifier and a sequence number equal to a default value (in our implementation 0), and thus is generated when the first data packet is sent over the network.

When a carrier is not needed anymore, it is destroyed with a call to procedure *Purge* in order to inform the local protocol software –but not the communication partner– that the corresponding stream shall not be used for further data exchanges. Purged carrier descriptors are kept registered for a while so that packets generated by the other stream end can be replied to appropriately (last acknowledgement problem). In other connection–oriented approaches [Sunshine78], closing results in communication over the network and changes the state of the protocol descriptor at the remote machine. Since applications –and our implementation in particular– use high level protocols to ensure that conversations are terminated in a synchronized way, this is not necessary.

Transmission and Flow Control

Once a stream is established, data can be transfered using the Byte and Bytes operations. The writing carrier puts data directly into a packet buffer that is sent over the network when it becomes full. Reliability is achieved via a stop–and–wait protocol, which means that successful receipt of data is acknowledged on a per–packet basis, and that the sender remains blocked until the acknowledgement arrives. Adopting a window protocol that allows the sender to continue execution concurrently to data transmission (and acknowledgement) [Tanenbaum81] was not considered necessary, because local area networks have small round–trip delays and there is at most one transient packet at a time. Also, in our system, acknowledgements are generated very fast, because the reactive part of the protocol software is invoked directly from within the network dispatcher which is activated as soon as packets arrive. Unlike the receiver–driven protocol employed in the network services of the Oberon system, this scheme achieves decoupling of the sender and receiver. This is essential for obtaining throughput in a concurrent system, because the receiving thread may be delayed due to local load and there is also a large variance in the time between packet transmissions.

Decoupling between the threads communicating over a stream is further supported by allowing several packets to be buffered at the receiver before suspending the sender. Destination buffering is particularly helpful if several data packets may be produced by the sender while the receiver does not have control of the processor [Sunshine76]. Such bursts are even more likely to occur in local area networks where round–trip delays are typically small compared to the process scheduling intervals. A typical situation where the sender profits from this support is shown in figure 1.

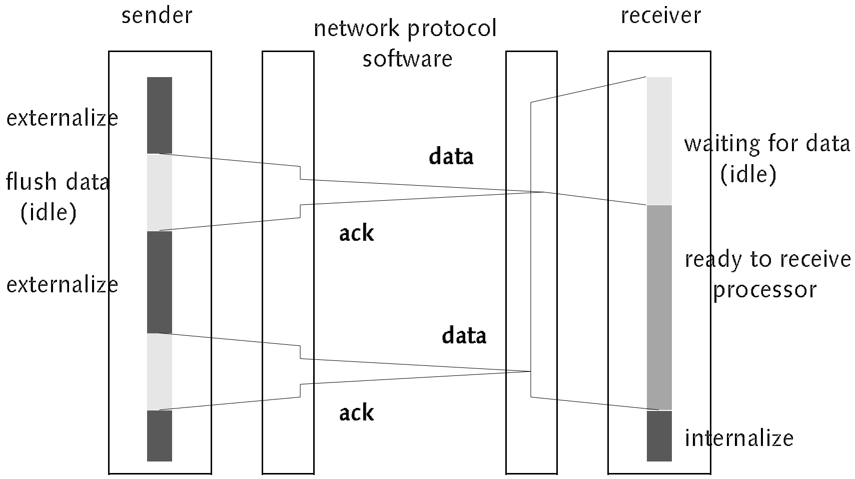

Figure 1 decoupling between sender and receiver threads

Buffering is kept simple and efficient. The receiving carrier maintains a list of free packet buffers which are subsequently swapped with data packets arriving from the network when the protocol software is notified from within the network dispatcher (see chapter 1). Occupied buffers become free again as data is gradually consumed by the receiving end of the stream. Since data is directly retrieved from the received packet buffers, copying is completely eliminated.

To suppress transmission of packets that cannot be accepted due to lack of space at the receiver, acknowledgements sent back to the sender to confirm receipt of a data packet also carry information indicating the number of packets the sender may produce without causing an overflow. When data is produced at a faster rate than it is consumed, this information is used to block the sender in advance, before it produces a packet that must be ignored. The sender is unblocked again by the receiver when free buffers are created. The thread receiving data over a stream blocks too when there are no occupied buffers left. In both cases, the blocked threads periodically send monitoring packets to the remote machine and transfer is aborted if several consecutive monitoring attempts are left without reply.

Transfer Phases and Synchronization

To tolerate synchronization failures, transfer over a network stream is divided into so called transfer phases. Each transfer phase carries a piece of data while its boundaries serve as firewalls isolating it from the rest of the data conveyed over the stream. Hence, the data of a transfer phase is either successfully processed or ignored without affecting processing of the data following it. Transfer phases are thus ideal for hosting transfer of data objects that do not belong together (in our case different messages).

To differentiate between packets corresponding to different transfer phases, each data packet bears a transfer phase number indicating the transfer it belongs to. This information is mainly used by the receiver in order to discard data packets belonging to a transfer phase when a synchronization error occurs. The end of a transfer phase and the begining of the next one are communicated to the protocol software with calls to procedure *TPhaseSynch* which increases the carrier's internal transfer phase number. The state of the carrier with respect to the terminated transfer phase is recorded in parameter *res* and it is automatically reset in case a synchronization error has occured. If, however, transfer fails due to other reasons, the carrier is purged and may not be reused for additional data transfer.

When sending, the end of a transfer phase results in flushing the contents of the output buffer so that data of different transfer phases are not placed in the same packet. This is essential for correctly resetting the receiving carrier if it experiences a synchronization error so that it can commence reading data of the next transfer phase. The last data packet of a transfer phase is marked to detect synchronization errors when the receiver attempts to read more data that actually produced. Due to this support, robustness against synchronization failures is achieved without introducing any end–to–end communication between the cooperating partners.

At the receiver, when a transfer phase ends, a check is made to determine whether there are any unprocessed data left. If this is indeed the case, then a synchronization error has occured and all occupied buffers belonging to the failed transfer phase are set free. Arriving packets that belong to a failed transfer phase are rejected and the sender is notified as to abort the current transfer. Data packets of new transfer phases are always accepted. Thus the sender may start a new transfer phase despite the fact that the receiver has not finished processing the data of the last one. This overlapping is semantically acceptable because the receiver can still detect and overcome synchronization errors, and desirable since it decouples the sender from the receiver.

To enhance flexibility, a special operation *InstallTPhaseNotifier* is offered that allows a notification procedure to be installed in a receiving carrier. This procedure is invoked from within the protocol software to trigger application specific processing at the begining of a new transfer phase. If a notification is desired for each individual transfer phase, then this procedure must be invoked at the end of each transfer phase. With this feature, a fresh receiving carrier may be created as soon as an already initialized carrier is bound to a stream so that there is no risk of missing incoming binding requests. This provision also eliminates the need to always have an idle receiver thread waiting for binding requests to arrive, since the notification procedure can be used to create a thread at exactly the required moment.

Network carriers also implement the *Synch* operation of the base carrier type to perform additional checking. When writing, this operation injects a tag into the stream, and when reading, the retrieved value is compared to this tag. If a mismatch occurs, a synchronization error is present and the state of the carrier is set accordingly. The current version uses a tag consisting of two identical byte values and implements byte stuffing to guarantee that the tag will not be re–produced by the application so that synchronization errors are reliably detected. If the application sends a byte that is equal to the value used for the tag, a filler byte with a different value is subsequently written in the stream. Analogously, when the application reads a byte that is equal to the byte used to form the tag, the next byte is removed from the stream. If it not equal to the filler byte, then a synchronization error has occured. According to our measurements, this filtering increases the cost of the carrier operations about 5–10%; given the reliability achieved with this approach, we believe that this performance penalty is justified.

Transfer Environments

The network carrier transport protocol is the foundation upon which message transfer is based. When the Hermes system is loaded, a single receiving carrier is installed at the local machine to intercept incoming transmission requests. Also, a notification procedure is installed, programmed to create a receiver thread responsible for servicing the carrier as soon as a corresponding stream is established, and to initialize a new receiver carrier for intercepting new bind requests. The sending thread and the corresponding carrier are created by the applications that wish to send a message, and the first data packet produced by the sender automatically results in establishment of a stream and in creation of a corresponding receiver thread at the remote machine. For brevity, a network stream and the associated sending and receiving threads are referred to as a transfer environment.

Once a transfer environment is initialized, both the sender and receiver threads use the default formatter and linearizer implementations to perform the message transfer. In other words, the sender externalizes and the receiver internalizes the message with a call to the Obj procedure of the linearizer component. Transfer is terminated by completing the corresponding transfer phase, thereby allowing for synchronization checks to be performed:

```
sender thread                  receiver thread

l.Obj(msg);                    l.Obj(msg);
resume application             end transfer phase
end transfer phase             IF done THEN queue & notify ELSE error END;
```

On the destination machine, depending on the state of the transfer phase, the receiver either creates a notification message or queues the message at the destination mailbox and notifies the application using the notifier procedure installed in it, if any. In the current implementation, notification of the application is achieved by creating a thread that invokes the notification procedure of the corresponding mailbox. To avoid creating a new thread for each new call, notifier threads are not destroyed as soon as the corresponding notification call terminates, but are kept in a pool to host additional calls. While, in principle, a direct notification from within the communication thread is possible, this would make the communication system vulnerable to delays occuring due to extensive application processing. Also, given the flow control mechanism we adopt, the latter approach would introduce the possibility of deadlock if applications used the notification mechanism to send messages.

On the sender side, the application remains blocked during externalization and is resumed before the internal buffers of the transport protocol are flushed. This guarantees that the contents of messages will not be modified after they have been forwarded to the communication subsystem while avoiding copying the message to system buffers. Eliminating buffering enhances efficiency (copying is a well known disease of network protocols and operating systems) and allows our system to be implemented with a moderate amount of memory resources. In fact, even for arbitrarily long messages the only memory used for sending and receiving a message are the internal packet buffers of the underlying communication protocol.

Despite the fact that the application remains bound to the sending thread during data transmission, it still is largely decoupled from the receiving thread performing the internalization due to the implementation of network carriers. As a consequence, the sending thread can perform data transfer even if the receiving thread is temporarily delayed perhaps due to other computations residing on the same machine, or is still busy internalizing the last message sent to it. Blocking occurs only if the protocol at the destination machine is unable to accept incoming packets due to slow processing of the receiver thread. In fact, if the sender and receiver are comparably fast, a considerable degree of parallelity is achieved for long messages that span over several packets, because internalization is performed hand in hand with externalization, at full transfer speed, and message reconstruction is completed essentially at the moment the sender transmits the last data packet. Small delays occuring during the transfer are ironed out by the destination buffering mechanism. Notably, transmission is non–blocking when message data can be hosted within a single network packet, since then the application is resumed before data is sent over the network and continues execution concurrently to data transmission.

Motivated by the fact that the startup and initialization costs of a transfer environment are comparable to the time required to send a message over the network, it is attempted to amortize transfer environments over several message transmissions. For this reason, transfer environments are not destroyed as soon as the corresponding data transmission completes, but are kept in a pool so that they can be reused for transferring additional messages.

The pool of passive transfer environments is searched when applications wish to send a message, and new transfer environments are created only if a matching environment for the desired destination is not found. Our implementation guarantees that a single transfer environment is created for each remote machine. Applications that wish to send messages using a transfer environment that is already active put their messages in a queue and are suspended until message transmission has been completed. Notably, this scheme guarantees FIFO ordering of messages between application–mailbox pairs, because messages sent by the same application (thread) to the same mailbox are transmitted by the same transfer environment in the order they were handed to the communication system.

Since eager creation of transfer environments can easily result in having several transfer environments that remain idle after hosting a few message transmissions, transfer environments are automatically collected if they have not been used for some time (the current version has a timeout of 30 seconds). To avoid race conditions between the sender and receiver, the sender explicitly signals destruction of a transfer environment to the receiver via an empty transfer phase containing a single byte before purging the carrier and destroying itself. Upon receipt of this signal, the carrier and the receiving thread on the remote machine are immediately destroyed.

Related Work

The Mach system [Accetta86] also supports communication between processes via asynchronous message passing, according to a scheme that was originally implemented in the Accent network operating system [Fitzgerald86, Rashid81]. However, this facility is low–level and language independent so that it is more a foundation upon which language specific support can be implemented, rather than a communication facility to be used directly by applications. Messages therefore simply consist of unstructured byte buffers and data conversion is left to be implemented by the clients of the system.

Often, message–oriented support is provided in the form of arbitrarily long messages that are sent reliably over the network, in a single piece. One approach is to implement long messages as a chain of buffers that are transparently accessed via special input and output operations [Szyperski90, Weck90]. Another possibility is to let the application provide the buffer chain directly [Linton86]. However, while this abstraction addresses the problem of putting long data items into messages, in both cases the application programmer must still pack data into and unpack data out of messages, respectively.

Asynchronous message passing with so called "active" messages is also used to support communication in multiprocessor machines [Eicken92]. Active messages contain at their header the address of an application–specific handler that is immediately executed as soon as the head of a message arrives at its destination to fetch the (rest of) the message out of the network. As opposed to the transfer procedure of Hermes message though, the handler is only responsible for reconstructing the message at the destination, and thus the buffers containing the message data must be explicitly constructed by the sender. The handler of an active message also implements the task of passing the message to the corresponding computation ongoing at the destination machine. Consequently, there is no receive operation and the application in notified about a message event directly from within the handler of that message. This allows applications to integrate message arrival in their computations in a flexible way. In Hermes, a similar effect can be achieved by using the provided notification mechanism.

Message passing is also used to implement synchronous communication with the ultimate objective to serve as a transport protocol for implementing remote procedure calls. A typical example is the V distributed system [Cheriton84] which supports interprocess communication in a blocking request–response fashion. In other words, after a process has sent a message, it does not continue execution (as in our system), but waits for a corresponding reply to arrive from the destination. Further, there is no support for high level data types, and messages are of fixed size. Transfer of large data items that cannot be placed in a single packet must be transfered via a special block–transfer operation that writes and reads entire memory segments over the network.

A refined version of this communication scheme is supported by the versatile message transaction protocol (VMTP) [Cheriton86]. Message transactions consist of a request and the receipt of the corresponding reply. As in the V system, messages are unstructured and of limited length, but in this version a message may be bigger than a network packet. VMTP maintains stable transaction records (objects similar to mailboxes) that retain their meaning across several message exchanges and that can be re–used to perform several high level conversations. However, separate descriptors are created for each client at the server machine. Also, clients remain blocked during a message transaction, and thus may have only one outstanding exchange at a time. On the contrary, in our system, a server may use the same mailbox to serve all clients, and a client may issue several requests one after the other so that remote processing is performed in parallel.

Another communication paradigm which actively supports data exchange in a programmer friendly fashion –typically unparalleled by most message based work– is the remote procedure call [Bershad87, Birell84]. The main feature of RPC systems is that they come with strong marshalling support so that the application can use remote procedures as if they were local. The code needed to implement binding, data conversion and transfer is typically produced by special tools called stub generators. However, stub generators typically require interfaces and data type definitions to be written in a special description language. In addition, even though RPC system typically support quite a few built–in data types, some implementations come with restrictions. For example, in Matchmaker [Jones86], a system built on top of the Mach operating system, pointers are not traced and thus it is not possible to send dynamic data structures over the network. While our system does not come with automatic generators, a single notation is used to write programs and there are no limitations with respect to transfer of data types. We also expect that the task of implementing a tool for generating transfer code in our system would be considerably simpler than that of writing a typical stub generator.

Since the remote procedure call was originally conceived as the exact copy of the local procedure call, most implementations support communication in a strict request–response pattern. Hence, the caller is blocked until remote processing completes and the results are sent back. For this reason, systems that focus on interactions where it is desirable to achieve parallelity have adopted a special type of remote procedure calls, often referred to as asynchronous remote procedure calls, or promises [Arbenz94, Liskov88]. As opposed to remote procedure calls, asynchronous remote procedure calls return immediately, and a calling program is blocked only if it attempts to read the result of a call that has not yet terminated. To allow for more parallelity, in some cases it is possible to block on a set of results in order to obtain the one that is computed first. However, introducing asynchrony through a tool that is traditionally used to model synchronous cooperation, seems somewhat confusing. This is especially so, since programming with asynchronous procedure calls is quite different than with ordinary procedures; asynchronous calls are namely placed very early in the program text, possibly several lines before their results are actually needed. As a consequence, fully exploiting the potential of this mechanism leads to message–oriented program designs, which speaks in favour of a message based approach.

A somewhat generic disadvantage of procedure–oriented communication is that programs performing long computations cannot pass incremental results to their clients, and thus computations must be broken up into pieces. Also, the throughput of a synchronous remote procedure call system cannot be improved by aggregation of several sequential calls, because a call must complete before the next one may be started. The same problem exists in asynchronous RPC systems that do not support ordering of concurrent calls. Remote pipes or channels [Gifford88] are an approach towards addressing these problems more effectively. Channels allow a process to issue a series of calls one after the other without having to wait for the server to complete processing. Calls are executed in the order they were issued and it is possible to synchronize, i.e. to wait for the last call to terminate. Asynchronous message passing offers comparable flexibility to this approach, since a client may send several requests before starting to collect the results. It also allows server programs to communicate intermediate results of a computations to their clients in a very simple way.

Unlike all mentioned communication methods that are built for supporting communication across distinct address spaces, another paradigm, called distributed shared memory [Fleisch89, Li88], is used to simulate a conventional shared memory environment on top of distributed system. But while distributed shared memory is appropriate for parallel computations with a high degree of sharing, it is not convenient for implementing fault tolerance, because the location of data is transparent for the programmer with the consequence that any part of the shared address space may suddenly become unreachable due to machine crashes. In addition, implementations of this scheme rely on the ability of the hardware to detect memory faults, which makes them inappropriate for systems with no memory management hardware. Last but not least, since data exchange is page–oriented, data conversion is hindered because the type and the location of data items within a page cannot usually be determined at run time.

Summary

A message passing facility was presented. In contrast to other message based systems, our implementation allows instances of arbitrarily complex data types that may even contain dynamic data structures to be exchanged over a network using simple communication primitives. Comparable support for transfering complex data is found in remote procedure call systems, but we achieve this functionality without the use of special description languages and complex tools for generating code. Instead, the applications that implement the messages also provide the code for their transfer in a type safe way, a feature that enhances flexibility yet hardly complicates programming. Message transmission is efficient, because the sender marshals data onto the network without any intermediate buffering, and at the receiver data is retrieved directly from within arriving packet buffers. Due to the adopted flow control mechanism, it also possible to transmit arbitrarily long messages using bounded buffers (at the network protocol level). Since we support decoupling between message passing activities, parallelity can be introduced at the application level without the need to explicitly create threads for each data exchange. Synchronous communication can still be achieved in a simple way, by sending messages and blocking until a corresponding reply is received. Finally, error notification occurs via sending error messages to the application, an approach that allows failures to be handled by the application in the same style as conventional messages.

Chapter 5 Remote Objects

In this chapter, we present a facility of the Hermes system that supports remote installation, invocation, and removal of application components called remote objects. Remote objects can be allocated over the network and are referenced via references. References to remote objects can be copied to achieve sharing, and a remote object is removed when all of its references have vanished. The system automatically detects crashes and garbage collects orphan objects. Remote objects can be invoked transparently over the network by sending them messages. Message events are notified to the corresponding object as soon as they occur with calls to a message handling procedure that implements the behaviour of the object. Remote objects are programmed in a true object–oriented way, i.e. additional functionality can be introduced by refining existing object implementations. Message handling is by default sequential, but intra–object concurrency can be introduced at arbitrary places within the message handling code to achieve parallelity.

An Event–Oriented Execution Model

In a distributed environment, the ability to install and remove program parts over the network from within other programs is of great importance, because it allows applications to incrementally occupy machines in the network depending on their processing and reliability requirements. A typical example where this flexibility is desirable is a program that distributes or replicates computations using idle machines to enhance performance and availability.

The functionality which will be required of components of distributed programs varies considerably. On the one hand, components may be active programs that perform a certain task on behalf of the application that initiated them, and on the other hand it is also possible that they implement a passive resource which can be accessed over the network. From a more abstract point, however, there is no substantial difference between these two extremes. An activity performing a computation can be viewed as a passive object that has been invoked and thus contains a thread executing the corresponding request, while a resource may be viewed as a server process that remains suspended waiting for incoming service requests to arrive. It is therefore appropriate to capture both types of functionality using a single abstraction.

We have done this by adopting an event–oriented model. Components of distributed programs are implemented as encapsulated entities, called remote objects, associated with a mailbox whose address serves as an identification for the corresponding object. A remote object is primarily passive; it becomes active as soon as a message is deposited into its mailbox, and reverts to its passive mode when message processing completes. Notably, with this approach, the only difference between embodying an activity and a passive resource is that the former type of objects will typically remain active for a longer time than the latter. A remote object can be arbitrarily complex though, in particular it may contain additional threads that execute concurrently to message processing.

The behaviour of remote objects is captured by a type–bound handler procedure that is invoked when a message arrives from the network and is placed in the object's mailbox. Thus the handler of remote objects is programmed in an event–oriented way, with events being message arrivals. Since the handler is a type–bound procedure, it can access data local to the object instance that received the message, and the modifications made to the object's state are preserved after the corresponding handler invocation terminates. In other words, remote objects essentially are state machines whose transitions are triggered by messages arriving from the network. This design achieves persistence of state across invocations in a natural way, in contrast to RPC systems where special "static" data declarations must be inserted in the interface description of a service to accomplish a similar effect. It also allows several instances of the same type to be placed in the same address space without interfering with each other. Finally, it becomes possible to develop remote objects incrementally, because existing object implementations can be extended by adding both new state and novel message processing capability.

Installation

Remote objects can be installed over the network via calls to procedure *Install*. The machine where the instance is to be allocated as well as the object's type, consisting of the corresponding module and type names, are supplied as parameters. Invocation of this procedure results in sending an installation request containing specified type information to the object server of the target machine. On the remote machine, the request is processed and a reply is sent back, thereby unblocking the caller.

```
DEFINITION RObjs;

   CONST
      Ok = 0; NetErr = 1; TypeErr = 2; ModErr = 3; KeyErr = 4;
      ProtErr = 5; CheckErr = 6; ConfigErr = 7; RefErr = 8;

   TYPE
      Ref = RECORD
               adr: Msgs.Adr
            END;

   PROCEDURE Install (target: SHORTINT; mod, type: ARRAY 32 OF CHAR;
                      VAR ref: Ref; VAR res: SHORTINT);
   PROCEDURE Fingerprint (mod: ARRAY OF CHAR; id: LONGINT);
```

As opposed to conventional objects that are allocated locally, creation of a remote object may fail due to several reasons. Besides being unable to communicate with the target machine, the machine may be protected, the specified type information may be invalid, or there may be inconsistencies between the local and the remote module versions. In fact, it may not even be possible to load the required modules, either due to key mismatches, or because the corresponding object files are not available on the machine. Loading a module and generation of an object instance do not need to be implemented explicitly since these services are already provided by the Oberon system.

If installation succeeds, a reference containing the mailbox address and the type of the allocated object is returned. This address is used to communicate with the object using the primitives described in the previous chapter. Hence, communication with remote objects is message–oriented and asynchronous, which allows client programs to resume operation after sending a request to a remote object and pick up the corresponding results, if any, whenever this is appropriate.

Version Checking

In Hermes, installation may fail despite the fact that the required modules have been successfully loaded on the destination machine due to a special check that determines compatibility between the code residing on the machine that invokes the Install operation and the code available on the destination machine. In a distributed system where several instances of the same module may interact with each other, this is necessary to guarantee that the code driving the cooperation is the same on each of the involved machines. This is even more so in our system since the transfer code of messages is provided by the application and a perfect symmetry between the externalization and internalization code is required to accomplish message transfer.

Compatibility in this spirit, however, cannot be checked by comparing interfaces as this is done in a non–distributed system, because different implementations of a module can have exactly the same interface. Instead, a thorough way of determining whether two instances of the same module are equal to each other is needed. To achieve this, we suggest that each module exports a parameterless procedure named "Fingerprint" that invokes procedure *Fingerprint* with the module name and a version identifier as parameters. We also demand that a module's fingerprinting procedure invokes the fingerprinting procedures of all imported modules. Hence, under the assumption that programmers correctly implement the fingerprinting procedures of their modules to indicate version changes, a fingerprint of a complete module hierarchy can be obtained by invoking the fingerprinting procedure of the topmost module. Our implementation does exactly this for both the caller and callee environments and compares the resulting fingerprints to determine compatibility. If a mismatch occurs, then the corresponding installation attempt is aborted and an error result denoting this failure is returned to the application.

Introduction of this mechanism was motivated by the fact that the Oberon system allows the address of any exported parameterless procedure to be retrieved using only its name. In a system that lacks this service, checking in this fashion is considerably more difficult to implement. Another –and perhaps better– solution would be to change the compiler so that it automatically produces fingerprints for each module which would uniquely identify its version in a machine–independent way. This could be achieved by using a module's source text to compute a signature, in a similar way this is done to generate keys for interface descriptions. Of course, a single universal coding algorithm would have to be employed by all compilers to ensure that checking can be performed across different platforms.

Sharing and Garbage Collection

Once a remote object is successfully installed and a reference to it is obtained, special operations can be used to achieve sharing and garbage collection of remote objects. Additional references to a remote object are obtained with calls to procedure *CopyRef* that takes a reference as a parameter and produces a new one, provided that the object type denoted by the specified module and type names is assignment compatible to the type of the object pointed by the reference which is being copied.

```
DEFINITION RObjs;

  PROCEDURE CopyRef (cpy: Ref; mod, type: ARRAY OF CHAR;
                     VAR cpy: Ref; VAR res: SHORTINT);
  PROCEDURE PurgeRef (VAR ref: Ref);
```

Therefore, the CopyRef operation can be viewed as the analogue of the pointer assignment for conventional heap objects. Copying across machine boundaries is achieved by transfering the original reference over the network and using it as a prototype for the CopyRef operation at the remote machine. In other words, transfering a reference over the network does not create a new reference on the destination machine, but simply gives the receiver the information needed to contact the remote object. References are disposed of via procedure *PurgeRef*. Remote objects are removed from the system when all corresponding references are destroyed.

These operations conceptually implement a reference counting scheme. The reference count of an object is initially set to 1 when the object is successfully created, and is incremented as new references are produced by copying. Conversely, the reference count of an object is decremented each time a corresponding reference is purged, and the object is removed when its reference count becomes equal to 0. Node crashes are automatically detected by the system, and references residing on failed machines are considered invalid, i.e. the reference count of the corresponding objects is decreased.

With these primitives, applications can implement sharing according to the following rule: reference variables must never be assigned values directly, and the prototype of a CopyRef operation may not be purged before the operation completes. Sharing errors at the application programs lead to object collection which is detected when trying to obtain a copy of a reference to that object. Collection is achieved by purging references when they are no longer needed. In contrast to sharing errors, failure to purge a reference is not detected, and the corresponding object remains allocated until the machine where the orphan references reside is switched off. While automatic collection of references would considerably simplify programming, unfortunately, our system does not support this. The reason for this limitation is that, in Oberon, applications cannot determine whether an object –be it a record on a stack or a heap object– becomes unreachable so that they can take special correcting actions. Other systems such as ETHOS [Szyperski92] and Modula–3 [Nelson91] provide support for this kind of notification, and thus are ideal platforms for implementing this functionality.

A Formal Specification

Garbage collection in our system can be described more formally via five properties P1–P5 as shown below. We define the expression *alive* to be equal to 1 if the specified machine is alive, otherwise 0. Analogously, installed equals to 1 only if the object was installed by that machine. Finally, the terms *nCPY* and *nPRG* denote the number of (successfully completed) CopyRef and PurgeRef operations invoked by a machine. The terms *RefCount* and *UsedRefs* indicate the number of references recorded by the system and the number of references used by the application, respectively.

System

```
invariant P1: ∀ m: RefCount(m) = alive(m)* (installed(m) + nCPY(m) − nPRG(m))
invariant P2: (+ m : RefCount(m)) > 0 ⇒ object not collected
progress  P3: (+ m : RefCount(m)) = 0 → object collected
```

Application

```
invariant P4: UsedRefs(m) <= installed(m) + nCPY(m) − nPRG(m)    (*sharing*)
invariant P5: UsedRefs(m) >= installed(m) + nCPY(m) − nPRG(m)    (*collection*)
```

According to the first property P1, our system guarantees that the reference information of an object with respect to any machine is consistently updated according to the operations that machine has issued, provided that it has not failed. It is also guaranteed that an object is not collected as long as at least one operational machine still references it (P2). The progress property P3 is used to express the fact that an object will eventually be collected if at some point in time all of its references disappear, either as a result of successive PurgeRef operations or due to machine crashes. In other words, once the reference count of an object becomes 0, it is no longer possible to increment it and to avoid collection.

From the above specification it is clear that our system does not implement sharing and collection, but only provides the tools that can be used to achieve this. This is reflected by the last two invariants which must be preserved by the application programs. Invariant P4 indicates that sharing is achieved as long as the application has not purged references it actually needs, while P5 states that object collection will not take place unless all of the corresponding references have been purged. Notably, the conjunction of P4 and P5 (i.e. maintaining strict equality in both cases) implies that reference information is updated consistently with the application needs, and thus is the specification of an application that implements the functionality of automatic garbage collection.

Implementation

Reference information of remote objects residing on a machine is recorded in form of a list containing one entry for each individual reference. Each list entry contains a physical reference to a local object and also holds the mailbox address of the object server of the machine where the corresponding remote reference resides. The reference list implicitly implements the object's reference counts, hence it is augmented with a new entry when a reference is copied, and entries are removed as references are purged. Obviously, a remote object is collected as soon as there is no associated entry in the list. The entries of the list are also used to monitor the machines where references to remote objects reside. In case of failure, the effects of a purge operation are simulated, thereby leading to removal of the corresponding entries. Since the object servers of machines get a new mailbox number when a machine reboots, race conditions due to slow monitoring and fast restarts are completely avoided. Machines that reboot simply appear as new machines so that failures are always reliably detected without invalidating any new reference information.

The list is updated by sending reference messages with information indicating the actions that must be done each time a reference is copied or purged. The CopyRef operation sends a message carrying the mailbox address of the local object server and awaits a reply confirming that the message has been processed. Purging generates a message with the analogous information, so that the entry corresponding to the destroyed reference can be removed from the list.

Waiting for reply when copying a reference ensures that a copy operation followed by a purge operation on the reference that served as a prototype will indeed produce the expected result. A non–blocking implementation of the CopyRef operation, would allow the calling thread to continue execution and possibly trigger execution of the purge operation, before the reference count of the object is actually updated (figure 1).

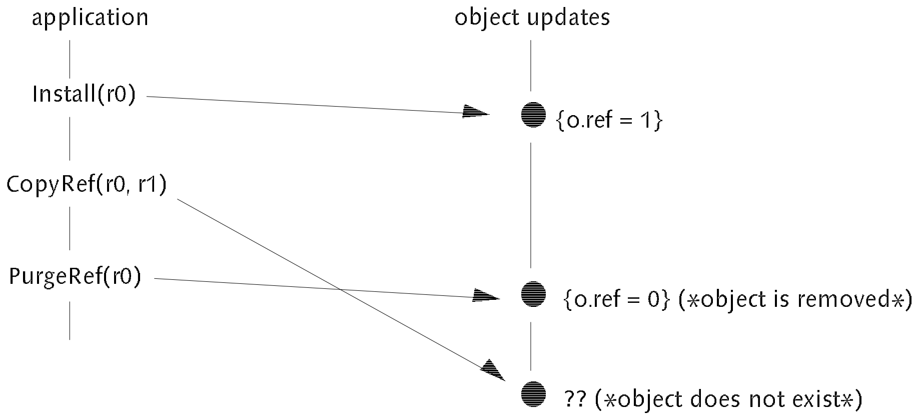

Figure 1 effects of a non–blocking CopyRef implementation

Hence, it would be possible for the message of the purge operation to reach the object before the message of the copy operation, because the causal ordering of events at the application level is not necessarily preserved over the network.

The ability to copy references, even if the cooperating activities reside on different machines, allows sharing to be introduced in a simple way. Of course, our approach is still less programmer friendly compared with automatic garbage collection one finds in non–distributed systems.

However, replicating this functionality in a distributed system seems unnecessary, because usually there is no complex sharing of objects in distributed applications. Even in highly parallel systems where computations consist of several activities sharing many fine–grained data items, the required resources are known at startup time, thereby making even a centralized resource management possible.

Distributed garbage collection is also undesirable due to the complexity and communication overhead of the algorithms needed to solve this problem in an incremental fashion [Schelvis89]. Needless to say, synchronous approaches adopted in non–distributed systems cannot be considered, since bringing a complete distributed system into a halt in order to identify garbage is not realistic. Simpler versions of asynchronous schemes have also been proposed [Birell93b], but they cannot handle cycles that span over several machines. As a consequence, cycles must be avoided, or have to be broken up explicitly by the programmer, a limitation that complicates, rather than simplifies programming.

Implementing Remote Objects

As already indicated, remote objects are abstract entities with a mailbox and message processing capability, implemented as a type–bound procedure that is invoked when messages arrive at the object's mailbox.

```
DEFINITION RObjs;

  TYPE
    Obj = POINTER TO ObjDesc;
    ObjDesc = RECORD
                  adr: Msgs.Adr;
                  PROCEDURE (o: Obj) Handle (msg: Msgs.Msg);
                END;
```

To support extensibility, the handler accepts the received message directly as a parameter, rather than being a parameterless procedure that must be programmed to retrieve a message from the object's mailbox. The latter approach would make it impossible for handlers to be called in a nested fashion to process the same message instance, because each handler invocation would retrieve (and process) a different message. On the contrary, our approach allows this, and consequently promotes an object–oriented development.

New remote object types with application specific state are defined as extensions of existing object implementations (including the base implementation) using the type extension facility of the Oberon language. Increased functionality may be introduced via new message types which are processed by the new object handler:

```
PROCEDURE (o: NewObj) Handle (msg: Hermes.Msg);
BEGIN
  IF msg IS NewMsgType0 THEN
    handle message of type NewMsgType0
   ELSIF msg IS NewMsgType1 THEN
```

```
          handle message of type NewMsgType1
        ELSE
          o.Handle↑(msg)    (*invoke code of base implementation*)
        END
      END Handle;
```

The actions to be executed can be selected depending on the actual type of the received message via type tests. Unknown messages, or messages defined by the base implementation can be processed via supercalls (indicated by a "↑" after the procedure name) to preserve its behaviour and invariants. The base type behaviour can be augmented by performing additional actions before and after a supercall; no supercall is required if the base type behaviour is to be overwritten. Notably, this event–oriented technique is similar to the way Oberon viewers (windows) are programmed to react to input events in an extensible way.

Since remote objects encapsulate complex components that can be active and contain data structures which must be set up explicitly when a remote object is created, it may be necessary to perform application specific initialization actions. Analogously, when a remote object is removed from the system, in general it is not sufficient to reclaim the memory occupied by it, but additional cleanup operations may be required depending on the type and the state of the object, such as destruction of threads performing internal tasks, or notification of other components.

```
DEFINITION RObjs;

  TYPE
    InitMsg = POINTER TO InitMsgDesc;
    InitMsgDesc = RECORD (Msgs.MsgDesc) END;

    RmvMsg = POINTER TO RmvDesc;
    RmvMsgDesc = RECORD (Msgs.MsgDesc) END;
```

For this reason, special initialization and removal messages (*InitMsg* and *RmvMsg*) are sent to remote objects when they are installed and collected, respectively. Remote object handlers can thus be programmed to execute the appropriate actions upon receipt of these messages. Base type processing for these messages can be refined but may not be overwritten.

An Example

As an example, a token that is acquired (and released) over the network to achieve synchronization between cooperating programs can be implemented as a remote object. Communication between clients and a token object is done via three messages. A request message containing the identification of the requester is used to issue an acquisition request. If the token is free or when it is relased by its current owner, a grant message is sent to the program that requested the token. Hence, clients requesting a token may either block until their request is granted, or continue execution and periodically check to see whether a corresponding grant message has arrived. Finally, the token is released by sending a release message.

The information that is required to decide whether to grant a token request can be stored in a FIFO queue containing the request messages sent by the individual client programs. When the token is first created, its request queue must be initialized and the owner of the token must be set appropriately to indicate that the token is free. A simple solution is to let the first entry of the queue denote the actual token holder so that no additional state is needed to record this information. Once the token object is initialized, it can start processing request and release messages (the corresponding definitions are omitted):

```
TYPE
   Token = TokenDesc;
   TokenDesc = RECORD (RObjs.ObjDesc)
                     reqQ: FIFO queue of Msgs.Adr
                  END;

PROCEDURE (t: Token) Handle (msg: Msgs.Msg);
BEGIN
   IF msg IS RObjs.InitMsg THEN
      initialize t.reqQ
   ELSIF msg IS ReqMsg THEN
      insert request in reqQ
      IF Head(t.reqQ).adr = msg.nfyadr THEN send grant message END
   ELSIF msg IS RlsMsg THEN
      remove head of t.reqQ
      IF NotEmpty(t.reqQ) THEN send grant message END
   END
END Handle;
```

When the object receives a request, it inserts it into the request queue, and if the queue was originally empty, it notifies the requester by sending it a grant message. On receipt of a release message, the object removes the head of the queue and if the queue still is non–empty a grant message is sent to the client program that issued the request in the head of the queue.

A possible extension of this simple implementation is a smarter token object that tolerates failures of its clients. Robustness can be achieved by monitoring the token holder so that the token object is notified via an error message if the program that holds the token fails prior to relasing the token (for simplicity we assume that the mailbox address of clients can be used for monitoring purposes):

```
TYPE
   RToken = RTokenDesc;
   RTokenDesc = RECORD (TokenDesc) END;
```

```
PROCEDURE (t: RToken) Handle (msg: Msgs.Msg);
BEGIN
  IF msg IS ReqMsg THEN
    t.Handle↑(msg);    (*supercall*)
    IF Head(t.reqQ).adr = msg.nfyadr THEN start monitoring owner END
  ELSIF msg IS RlsMsg THEN
    stop monitoring owner
    t.Handle↑(msg);    (*supercall*)
    IF NotEmpty(t.reqQ) THEN start monitoring owner END
  ELSIF msg IS Msgs.ErrMsg THEN
    IF NotEmpty(t.reqQ) & Head(t.reqQ).adr = msg.nfyadr THEN
      simulate reception of a matching release message
    END
  ELSE t.Handle↑(msg)
  END
END Handle;
```

This can be achieved by programming the handler of the extended token to monitor the actual token holder, and to switch monitoring targets when the token is passed to another program. No extra code is needed to manage the request queue since this is already done by the base implementation. It is, however, necessary to trigger base type processing via supercalls.

Handler Execution

The handler procedure of a remote object is invoked as soon as a message arrives at its mailbox. When the handler executes to completion, the mailbox of the object is inspected again, and if it contains another message, the handler is called again to process it. Hence, message events are notified to remote objects one after the other by subsequent handler invocations.

Frequently, one wants to be able to continue to react to message events while a computation triggered by the previously received message continues. Although blocking can be avoided by creating a thread to host time consuming processing, we provide high level support for introducing intra–object concurrency, tailored for this situation.

```
DEFINITION RObjs;

  PROCEDURE Decouple (o: Obj);
  PROCEDURE Couple (o: Obj);
```

When message handling involves a time consuming computation, it is possible to achieve concurrency with a call to procedure *Decouple*. This operation informs the system that the object's handler should be invoked when a new message arrives at the object's mailbox, despite the fact that there is already an ongoing handler execution corresponding to this object. In other words, decoupling a handler invocation allows it to run concurrently to invocations triggered by message events occuring during its execution (figure 2).

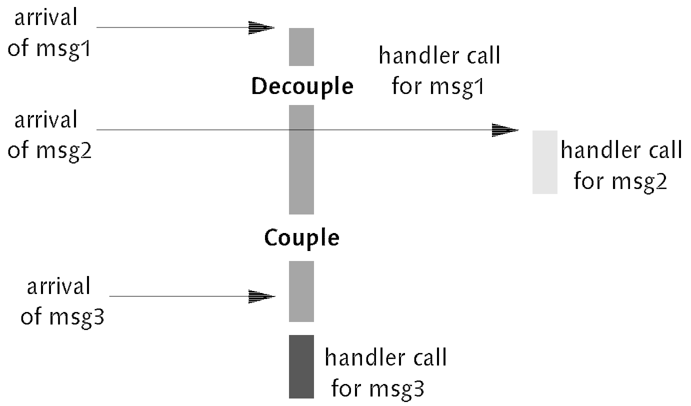

Figure 2 intra–object concurrency

The fact that a handler execution has been decoupled does not necessarily imply that it is desirable to let it run asynchronously to other handler calls until completion. On the contrary, decoupled executions may need to access the object's variables consistently, or it may be desired to revert back to the sequential execution mode after performing the time consuming part of message processing. Synchronization in this spirit is implemented by operation *Couple* which re–integrates a decoupled handler call in the sequential execution model (figure 2). After coupling completes and the caller is resumed, it is guaranteed that there are no other coupled handler calls associated to the corresponding object. A re–coupled call appears as if it had never been decoupled, i.e. no new handler calls are allowed –even if new messages arrive– until it terminates or is decoupled again. Consequently, a remote object can have several decoupled handler calls and at most one coupled handler call associated to it.

This execution model can be conceptually described by associating an object with a binary semaphore that is decreased before its handler is called and relased afterwards. Then, the Decouple and Couple operations are simply described as operations that increase and decrease this semaphore:

thread hosting call

```
P(s);
coupled := TRUE;
IF mailbox non–empty THEN
  remove msg from mailbox
  call handler
END;
IF coupled THEN V(s) END;
```

decouple

```
IF coupled THEN coupled := FALSE; V(s) END;
```

couple

```
IF ~coupled THEN coupled := TRUE; P(s) END;
```

In practice, thread creation is suppressed if a coupled execution is already associated with a remote object. To achieve this, each object has a thread variable, called *main*, containing the thread that is currently hosting the coupled handler call of this objects, if any. The value of this variable also indicates whether a new handler call should be made when a message arrives at the mailbox of the remote object. Obviously, main is initially set to NIL. Each remote object also has a FIFO queue containing the threads of handler executions waiting to be re–coupled. When the object is created, the coupling queue is empty. Also, a notification procedure is installed at the object's mailbox, which sets main equal to the currently executing thread and calls the object's handler if main is NIL, or simply returns otherwise. The decouple operation, depending on whether the coupling queue is empty, either resumes the first thread waiting to be re–coupled, or resets main to NIL and re–installs the notification procedure in the object's mailbox so that a new handler execution is created when messages arrive at the object's mailbox. Coupling of a thread is done in an analogous way. If main is equal to NIL, thereby indicating that there is no coupled execution associated to the object at the moment, main is set equal to the currently executing thread. Otherwise, the thread is placed in the coupling queue and is suspended. When a coupled handler call terminates, the effects of a decouple operation are simulated to update the object's state consistently.

Resource Control

Although the threads hosting the handler execution run with low priority, and thus the presence of active remote objects is transparent for the users of workstations, remote objects are not free of cost because they consume both processor time and memory of their host. Hence, a user may want to prohibit installation of remote objects on his/her workstation. What makes the need for control even more important, is that remote objects may be faulty. This is critical, because apart from the strong type checking of the Oberon language which guarantees that programs do not overwrite arbitrary memory locations, there is no other protection mechanism for containing errors. Specifically, due to the lack of a differentiated resource management, the amount of memory and disk space consumed by programs cannot be controlled, and thus a workstation hosting remote objects can potentially run out of these resources.

Since our environment consists of personal workstations and there is no central system administrator, the right to enable and to disable installation is given to the individual users via corresponding commands.

```
DEFINITION RObjs;

  PROCEDURE Protect;
  PROCEDURE Unprotect;

  PROCEDURE ShowObjs;
  PROCEDURE Destroy;
```

Allowing a workstation to be used as a host for remote objects is just a temporary concession, rather than a commitment. This not only means that an enabled workstation can be disabled again; it also implies that installed objects are not untouchable, but may be removed explicitly by the user of their host workstation. Access to remote objects is achieved by maintaining a directory of all installed objects, which can be inspected by the user to gain overview of the situation on the local machine and to selectively remove individual objects.

Enabling and disabling a workstation are local operations. As a consequence, finding an available workstation requires searching the network unless this information is provided in another way. It would also be possible to augment our system by introducing a registration service for available machines, as done in the Butler system [Nichols87]. Enabled workstations could then announce themselves to a server that would be contacted by programs to retrieve the addresses of machines where network objects can be installed.

Related Work

Our implementation adopts essentially the same event–oriented programming model as the XNet system [Lalis91] which was a result of previous work. However, in XNet, messages contain a request and a result part, and thus communication resembles that of an RPC based system where clients send a request and remain blocked until the corresponding reply arrives. While this allows to achieve the illusion of locality even stronger than in the current implementation, it also introduces the need for stub objects at the sender as well as at the remote machine, and has the disadvantages of procedure–oriented communication.

The Modula–3 network objects system [Birell93a] is closely related to our work, but introduces an additional layer on top of a message–oriented communication scheme, to achieve procedure–oriented communication. This allows network objects to be implemented and invoked in the same style as local objects. However, elaborate compiler support is needed to achieve this effect: for each user–defined type the compiler generates two stub–objects internally used by the system, a so called surrogate type and a server dispatcher. Instances of surrogate types serve as network references as well as representatives for the "real" objects, and transparently forward the procedure calls made to them over the network. The dispatcher is responsible for unpacking received request messages and invoking the corresponding object methods. It is placed on the same machine as the network object, and is invoked when request messages arrive over the network. Unlike our approach where references must be copied to achieve sharing, surrogate objects can be handled like ordinary pointers and are collected automatically. However, the implementation relies on the ability of the local garbage collector to provide notification when a local application object is collected. Network objects are automatically removed when all corresponding surrogate objects are collected by the local garbage collector, but the implementation fails to deal with cycles.

The Eden system [Almes85] also adopts a similar model for structuring distributed applications, via special active objects called ejects. Ejects cannot be extended, i.e. it is not possible to refine the behaviour of an existing eject implementation. A procedural interface is used for communicating with ejects, and a compiler automatically generates code for data conversion and transmission. Nonetheless, programmers are presented with a message–oriented model, since ejects must be explicitly instructed to receive and process messages using the compiler generated primitives. In our case, the programmer does not have to create processes explicitly, and messages can be processed directly from within the object handler without requiring any unpacking.

Our approach can also be compared to the concept of guardians which are used in Argus [Liskov87] for encapsulating co–located resources. The main difference is that guardians are invoked through procedures, rather than messages and are strongly transaction oriented, i.e. it is possible to preserve the state of the guardian despite machine crashes. A separate thread is created to run each call and a computation must be run as an action to guarantee that it will not be affected by other operations running concurrently to it. Synchronization is achieved via so called atomic objects [Weihl85] with write and read operations that are used to perform locking. No synchronization is required in our system because messages sent to the same network objects are processed sequentially, unless the programmer explicitly introduces intra–object concurrency.

The object model is also used in systems that focus especially on parallel programming, and thus support fine–grained encapsulation and mobility. A typical example is the Emerald system [Black86] where all entities are implemented as mobile network objects. Programs may move objects over the network to reduce communication costs. To enhance efficiency, the compiler chooses among different implementations for each object depending on the way it is accessed, and objects that are guaranteed to remain local are invoked directly via normal calls. Another approach is chosen in Amber [Chase89] where objects automatically move on the node of the activity that invokes them and remain there until another remote invocation occurs. The location of an object is determined through an entry in its reference descriptor which is replicated at the same memory location on all nodes as the object moves through the entire system. As an optimization, immutable objects are fully replicated and thus never need to migrate. An object can also be attached to other objects, thereby creating structures that move together in a single piece and remain co–located. Our programming model is less flexible, since we do not aim at supporting fine grained mobility of objects over the network.

Summary

We have introduced a facility that allows arbitrarily complex application components to be installed and invoked over the network. An object–oriented model was chosen to support implementation of such components, independently of whether they are passive or active entities. While we adopt an event driven execution model by letting remote objects react to messages, most other systems combine object–orientation with remote procedure calls. As a consequence, in these systems, objects residing on remote machines must be represented by stubs so that the clients can access them transparently. Since we promote a message based interaction between activities residing on different machines, the need for stubs is completely eliminated. This makes our system considerably simpler, because elaborate tools or special compiler support is not required. Moreover, since message passing is asynchronous, decoupling between remote objects and their clients can be achieved in a simple way. With remote procedures (or methods), clients must introduce parallelism by explicitly creating threads for each time consuming call, and server objects cannot communicate results to clients incrementally. Also, in other work, intra–object concurrency is supported by creating a separate thread for each call, and synchronization between concurrently executing invocations must be explicitly programmed via semaphores, or some other device. In our implementation, message handling activities can simply switch from sequential to concurrent execution and vice versa, a feature which can be used to achieve comparable effects yet with less programming effort. Finally, we support object sharing via reference counting primitives, rather than via attempting to simulate a non–distributed garbage collection environment. Given the limitations or the complexity of other approaches, we believe that this is a well balanced trade–off between simplicity and functionality.

Chapter 6 Applications

Test and prototype versions of our system have been used to develop non–trivial distributed programs in the Oberon environment. Different classes of applications were investigated to apply the implemented primitives in various situations, and to evaluate our system in a better way. It was indeed a pleasant surprise to see that once the interactions among program components and the corresponding algorithms were clearly understood, programming turned out to be straightforward. Often, applications were implemented in prototyping speed and –astonishingly– almost no debugging was required. This is especially encouraging since this work was partly done by students with little or no experience in distributed systems.

In this chapter, two applications are presented: a reliable broadcast protocol that guarantees total ordering, and a facility for distributing and replicating time consuming computations over idle machines. Since out intention is merely to give an overview of how this functionality was achieved, we do not elaborate with low–level details but focus on the most interesting parts of the implementations.

Reliable and Totally Ordered Broadcasting

Broadcasts, i.e. transmission of messages to a group of machines or programs is a special type of communication that is typically used in distributed systems to coordinate processing among a set of replicated resources or processes. Broadcasting is said to be *causal* if messages are delivered to the application while preserving causality of message events. Put in other words, a causal broadcasting facility guarantees that if a member receives message m1 and then broadcasts message m2 to the group, group members will receive m1 before m2. It also guarantees that if a member sends message m1 followed by m2, then these messages will be delivered to the group members in the order they were sent. An even stricter form of delivery is *total* ordering which preserves causality and where causally unrelated messages are also delivered in the same order. Letting all group members see the identical message history allows arbitration decisions to be taken by each member in a consistent way without any further communication. Moreover, group members can update their local state in exactly the same way, going through the same transitions, which simplifies programming if a group implements a replicated resource. Finally, *reliability* means that a message sent to a group is eventually delivered to all living group members.

A protocol that supports atomic and total broadcasting within a group of cooperating programs, is implemented on top of the message passing system of Hermes. Our implementation can be viewed as a simplification of more general schemes [Chang84]. It is also similar to the protocol used in the Orca system [Kaashoek89] to guarantee consistency among object replicas. In addition, we support group changes so that activities may dynamically join and leave a group. Members that fail, are automatically removed from the group, and a group disappears when it becomes empty.

A Sketch of the Algorithm

Identical ordering of messages within a group is achieved by relaying messages through a distinguished member, called the sequencer, that forwards them to all group members (figure 1). Since the Hermes message passing facility preserves FIFO ordering, messages are received by all members in the order messages are received and broadcasted by the sequencer.

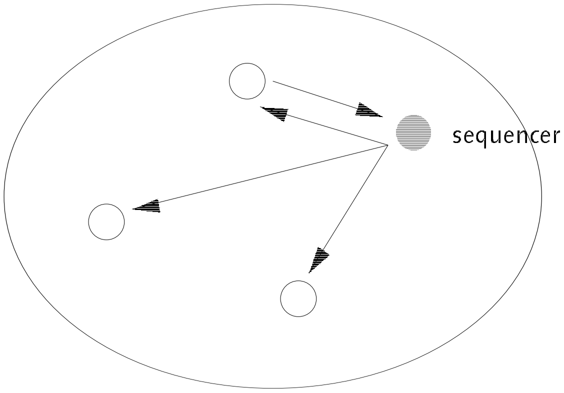

Figure 1 sequencer based broadcasting

This method also preserves causality, because if a member sends a message m' as a reaction to a delivered message m, it is impossible for m' to be broadcasted before m, because m has already been received (and broadcast) by the sequencer. Notably, messages generated by the same member cannot possibly overtake each other and are also broadcast in the correct order. In other words, the protocol can be described using three actions:

Application

send m: send message MbrMsg[m] to sequencer

Protocol

receive MbrMsg[m]: broadcast message SeqMsg[m] to all members

receive SeqMsg[m]: deliver m to application

This informal description not only captures the essence of this protocol, but also identifies its weakness with respect to failures. Namely, it shows that failure of a non–sequencer member is trivial to tolerate, because it does not affect operation of the protocol. However, if the sequencer fails, the protocol is blocked since messages sent to it will not be propagated to the group. While this problem can be solved by choosing another member as a new sequencer, this is not sufficient if the sequencer has failed in the midst of a broadcast action, because then some messages may have reached only a part of the group, or no members at all. Hence, the atomicity of incomplete broadcasts must be established before allowing the new sequencer to handle new messages.

Handling Sequencer Failures

In order to be able to retransmit messages in case of a sequencer failure, at each group member, messages received from the sequencer are kept in a queue until it is known that they have been received by all members. A second queue is maintained for messages originating from the members themselves, since they must also be retransmitted in case the sequencer failed before initiating the corresponding broadcast actions. When the sequencer fails, an election protocol is run to select as new sequencer the machine which has received the most messages from the old sequencer, with ties broken by using the identification of group members.

An election is started when a member detects failure of the sequencer. The member creates an election message containing its own identification and its local sequence number denoting the last message received from the old sequencer, and sends it traveling around a virtual ring formed by the operational group members. Upon receipt of an election message, a member switches to election mode, if it has not already done this, and forwards the message to its "next" neighbour only if the originator of the message is better qualified to become sequencer. To tolerate failures that occur during an election phase, each member monitors its neighbour (according to the ring formation), and retransmits its election message to the next neighbour if a crash is detected.

A member becomes sequencer if it receives its own election message. In this case, it retransmits the messages the old sequencer failed to broadcast to all members (if any) and broadcasts a message with its identification announcing termination of the election phase. Upon receipt of this message, all members retransmit their own messages which have been sent to the old sequencer but have not been broadcasted at all, and resume normal operation.

Joining and Leaving Groups

Members that wish to join a group, send a corresponding request to a group member and wait for a confirmation. The member receiving the request forwards it to the sequencer, just like any ordinary message, which in turn broadcasts it to the group. However, the confirmation is sent to the new member after all existing members have explicitly acknowledged receipt of the join message. This guarantees that the new member will not cause any inconsistencies if an election is started concurrently to the joining process. Any messages other than the confirmation received by the joining member are actively rejected and the member retries its join attempt. Rejections are treated as failures, i.e. the corresponding entry is removed from the group membership directory.

Leaving a group simply requires sending a leave message to the sequencer. If the sequencer is to leave the group, a fast election is simulated by randomly selecting a group member and sending it a leave message. As a result, the group member broadcasts an election termination message bearing its identification and becomes sequencer. Failures occuring during leave phases do not require special handling since they are adequately handled by the conventional failure detection and correction mechanisms.

Implementation

The reactive character of this highly asynchronous protocol naturally fits into the message based communication of Hermes, since protocol events arriving from the network can be naturally implemented as messages that trigger protocol processing. Protocol state at each member is encapsulated within a descriptor that is updated each time events occur. Events are protocol messages arriving from the network, some of which may contain application messages that are to be forwarded to the application.

Each protocol descriptor contains a mailbox that is used for the network communication with the group; the mailbox address serves as an identification for the member holding the descriptor. Protocol handling is implemented as a message handling procedure which takes actions depending on the type of the received messages:

```
PROCEDURE Handle (p: Protocol; msg: Msgs.Msg);
BEGIN
  IF msg IS SeqMsg THEN    (*for all members*)
```

```
        put message into queue and deliver corresponding application message
    ELSIF msg IS MbrMsg THEN    (*only for sequencer*)
        broadcast message to group members
    ELSIF msg IS ElectMsg THEN    (*all members except sequencer*)
        IF new election THEN start new election END;
        IF msg "better" than own election message THEN forward msg END
    ELSIF msg IS Msgs.ErrMsg THEN
        IF sequencer failure THEN start new election
        ELSIF election & neighbour failure THEN send election message to new neighbour
        ELSIF ~election THEN broadcast failure message
        END
    ...
    END
END Handle;
```

The protocol handler is invoked from within a notification procedure installed in the descriptor's mailbox so that reaction to protocol events occurs as soon as the corresponding messages arrive from the network and asynchronously to the application using the protocol.

Messages to be broadcast to the group must be passed to the corresponding protocol descriptor with calls to procedure *Queue* that forwards them to the sequencer.

```
DEFINITION Groups;

    TYPE
        Msg = POINTER TO MsgDesc;
        MsgDesc = RECORD (Hermes.MsgDesc) END;

        JoinMsg = POINTER TO JoinMsgDesc;
        JoinMsgDesc = RECORD (MsgDesc)
                            who: Msgs.Adr    (*address of member*)
                        END;

         LeaveMsg = POINTER TO LeaveMsgDesc;
         LeaveMsgDesc = RECORD (MsgDesc)
                            who: Msgs.Adr    (*address of member*)
                        END;

         Protocol = POINTER TO ProtocolDesc;
         ProtocolDesc = RECORD
                            adr: Msgs.Adr;
                            PROCEDURE (p: Protocol) Deliver (msg: Msg);
                            PROCEDURE (p: Protocol) TransferState (l: IO.Linearizer);
                        END;

    PROCEDURE Init (p: Protocol);
    PROCEDURE Join (p: Protocol; contact: Msgs.Adr);
    PROCEDURE Leave (p: Protocol);

    PROCEDURE Queue (p: Protocol; msg: Msg);
```

Conversely, messages that are broadcast by the sequencer are passed from the protocol handler to the application by invoking the type–bound procedure *Deliver* which must be overwritten according to the application requirements to integrate arriving messages into the application program.

Groups can change dynamically as new members are introduced and existing members leave the group. Applications can instruct their protocol instances to initiate the actions required to join or to leave a group via calls to procedures *Join* and *Leave*, respectively. When joining, an address of a group member is required to establish communication with the group. Joining succeeds if this member can be contacted successfully and does not fail during the process. If the specified address is equal to the mailbox address of the own protocol object, then a new group is created with the initiator as a single member. The protocol communicates join and leave events to the application by generating special messages *JoinMsg* and *LeaveMsg* bearing the identity of the corresponding member. For symmetry, join and leave messages are also generated for the own events, and signal the success (or failure) of the corresponding join and leave actions. Join and leave messages are totally ordered, i.e. are delivered in the same order to all group members, thereby allowing all members to maintain a consistent view of the group.

Once a member has received its own join message, it can start sending and receiving messages within the group, and is a candidate for assuming the role of the sequencer. However, if initialization of newly added members depends on the actual state of the group, then further actions must be performed at the application level before a new member becomes fully operational. To relieve the application from having to employ an additional protocol, this task is supported by introducing a state transfer mechanism in the form of a type–bound procedure *TransferState*. State transfer is invoked from within the protocol handler and occurs between the sequencer and the new member before the new member is sent its join message. Hence, when a new member receives its own join message it is guaranteed that its state has already been appropriately initialized.

An Example

This protocol has been used to implement a distributed editor that organizes the users editing a shared document in a group. Each participant maintains a local copy of the document that is used to read data without performing any communication over the network. Synchronization of write operations is achieved using a token based scheme. Only the user holding the token has the right to modify the document and changes made to the document are asynchronously communicated to the local copies. The token must be requested and released explicitly via corresponding operations, hence a request to acquire the token is granted only when the token is released by its current holder.

Token management is implemented in a decentralized manner, by maintaining a list of token acquisition requests at each group member. A member holds the token if its request is the first element in its own local queue. To allow each individual member to make this check in isolation and in a consistent way, token requests and relases are sent to the group as total broadcasts, and the protocol descriptor to which the request queue is bound is updated correspondingly as messages arrive from the network:

```
TYPE
   MyProtocol = POINTER TO MyProtocolDesc;
   MyProtocolDesc = RECORD (Groups.ProtocolDesc)
                        reqQ: FIFO queue of requests
                    END;

PROCEDURE Request (p: MyProtocol);
   VAR req: ReqMsg;
BEGIN NEW(req); req.adr := p.adr; Groups.Queue(p, req)
END Request;

PROCEDURE Release (VAR p: MyProtocol);
   VAR rls: RlsMsg;
BEGIN NEW(rls); rls.adr := p.adr; Groups.Queue(p, rls)
END Release;

PROCEDURE (p: MyProtocol) Deliver (msg: Groups.Msg);
BEGIN
   IF msg IS ReqMsg THEN
      insert entry [msg(ReqMsg).adr] in p.reqQ
      IF Head(p.req).adr = msg(ReqMsg).adr THEN start monitoring holder END
   ELSIF msg IS RlsMsg THEN
      remove entry msg(RlsMsg).adr] from p.reqQ
      IF holder changed THEN stop monitoring holder
         IF NotEmpty(p.reqQ) THEN start monitoring holder END
      END
   ELSIF msg IS Msgs.ErrMsg THEN
      simulate release operation
     ...
   END
END Deliver;
```

Since the group protocol guarantees total ordering, a release message will never overtake a corresponding request message, and concurrent requests for the token will be communicated to the application in the same order at all members. Even though copies may be updated with some delay, all list copies will eventually go through the same state changes, which suffices to guarantee that token ownership will be determined correctly. When a member leaves the group or fails, the list is searched to remove corresponding requests. If this member happens to be the actual token holder, then the token is automatically released and the next request is granted. The state transfer mechanism is exploited to initialize the request list of new members as they join a group.

Support for Remote Computations

Sometimes time consuming computations can be divided into sub–tasks that can be executed in parallel. In this case, the network can be used to perform each of these tasks using separate machines to accelerate processing. But even if it is not possible to distribute a computation, execution time can be decreased if it is shipped to a machine that is idle and has a faster processor than the local machine. In the latter type of computation, it may also be desirable to make a computation fault–tolerant so that it does not need to be restarted from scratch after a failure.

As a small step towards supporting these tasks, we developed a scheduler that distributes a list of jobs over a group of virtual processing elements. Processors may be dynamically added and removed without halting the computation process. For simplicity, it is assumed that no communication between the processing elements is required. This service is similar to the the Marionette system [Sullivan89] developed to support execution of parallel programs in a master–slave fashion. While fault tolerance can be achieved by cutting a computation into sub–computations, or by replicating the computation on multiple machines [Cooper85] hoping that one will carry the computation to its end, robustness against failures is further supported via checkpointing. In other words, the intermediate state of each processor computing a job can be periodically saved and used to restart the job if a failure indeed occurs.

Abstract Processing Elements

The processors used to perform the computations are modeled as abstract objects that accept a job and produce the corresponding results. Since it must be possible to dynamically instantiate processors on remote machines when the computation is started and destroy them after the computation terminates, processors are implemented as remote objects.

```
DEFINITION RComps;

  TYPE
    JobMsg = POINTER TO JobMsgDesc;
    JobMsgDesc = RECORD (Msgs.MsgDesc)
                        sadr: Msgs.Adr    (*address of scheduler*)
                      END;

    ResultMsg = POINTER TO ResultMsgDesc;
    ResultMsgDesc = RECORD  (Msgs.MsgDesc)
                            padr: Msgs.Adr;    (*address of processor*)
                            j: Job             (*state used to restart in case of failure*)
                          END;

    Processor = POINTER TO ProcessorDesc;
    ProcessorDesc = RECORD (RObjs.ObjDesc) END;
```

Jobs are defined as abstract messages of type *JobMsg* which serves as a base type for defining the job messages of the application hosting computation–specific data. Similarly, a type *ResultMsg* is used to introduce messages containing the intermediate results of a processor computing a job. Result messages also contain a job message serving as a point from which the original job can be restarted in case the processor fails. Hence, for a processor that receives a job, there is no difference between starting a job from scratch and using a checkpoint as a starting point. Job processing is presumed to be completed when a result with no job message is received.

Notably, unlike in systems that associate a separate address space with each activity and can therefore take a snapshot of the entire address space including stack and processor registers, checkpoints must be explicitly created by the processor elements. However, our approach supports customized, and thus potentially very efficient checkpointing.

A Simple Scheduling Model

Provided that a computation can be divided into jobs that can be executed in isolation, the process of performing such a computation in parallel can be formulated as a sequence of five main steps: (0) install and initialize the processors, (1) divide the computation into jobs, (2) distribute the jobs over the available processors, (3) collect the results, and (4) combine them to obtain the final result. But although this already achieves better parallelism compared to a purely sequential job execution, arranging these steps according to this strict order may be too restrictive.

For example, it may be desirable to commence processing before the computation has been fully subdivided into jobs, or to support installation of new processors even in the midst of a computation, so that additional machines can be involved in the computation as they become available. Moreover, for certain applications it may be worth handling results as soon as they arrive from the network to perform part of the merging process in parallel to job processing. This flexibility is easily achieved if scheduling is described as a set of isolated actions that can be executed in any interleaved way:

```
AddJob(j):               RegisterJob(j);
                         IF IdleProcFound(p) THEN Send(p, j) END;

AddProcessor(p):         RegisterProc(p);
                         IF IdleJobFound(j) THEN Send(p, j) END;

receive ResultMsg[id]:   FindProcessor&Job(id, p, j); RemoveJob(j);
                         IF IdleJobFound(j) THEN Send(p, j) END;

receive ErrMsg[id]:      FindProcessor&Job(id, p, j); RemoveProc(p);
                         IF IdleProcFound(p) THEN Send(p, j) END;
```

Failure handling is also easy to integrate in such an action–oriented approach, in fact, receiving an error message indicating that a processor failed is the exact inverse of receiving a result, i.e. the failed processor is removed, and a new processor is sought for to which the job can be sent.

The Scheduler

The state of the scheduling process is encapsulated into a scheduler object that has a mailbox and communicates with the processors to coordinates distribution. Messages arriving from the network are handled according to the above scheme from within a notification procedure that is installed in the scheduler's mailbox.

```
DEFINITION RComps;

   TYPE
      Scheduler = POINTER TO SchedulerDesc;
      SchedulerDesc = RECORD
               PROCEDURE (s: Scheduler) JobSent (netAdr: SHORTINT; jid: LONGINT);
               PROCEDURE (s: Scheduler) GotResult (netAdr: SHORTINT; jid: LONGINT;
                                                                     r: ResultMsg);
               PROCEDURE (s: Scheduler) Failed (netAdr: SHORTINT);
      END;

   PROCEDURE AddProcessor (s: Scheduler; netAdr: SHORTINT);
   PROCEDURE RemoveProcessor (s: Scheduler; netAdr: SHORTINT);
   PROCEDURE AddJob (s: Scheduler; job: JobMsg);

   PROCEDURE Initialize (s: Scheduler; mod, type: ARRAY 32 OF CHAR);
   PROCEDURE Destroy (s: Scheduler);
```

In order for the application to follow the scheduling process, internal events such as job assignments, receipt of checkpoints, and failures are communicated to the application by invoking corresponding type–bound procedures. These procedures can be used to display monitoring output about the computation, perhaps even to take correcting actions. For example, failing servers could be automatically replaced with new ones. In particular, the application must implement the *GotResult* procedure to process results as they are being produced by the processors.

Scheduler objects must be initialized with a call to procedure *Initialize*. At initialization, the type of the application processor objects must be specified so that this information need not to be passed each time the application installs a new processor. When a computation completes, the scheduler must be destroyed via *Destroy* to remove the installed processor objects. Actions such as adding a processor, or a job are triggered by the application via conventional procedure calls *AddProcessor* and *AddJob*, respectively. It is also possible to remove a processor with a call to procedure *RemoveProcessor*. Even though the application may add and receive processors at will, the existence of multiple processors is transparent, because coordination and data exchange with the processor group is implemented completely within the scheduler object (figure 2).

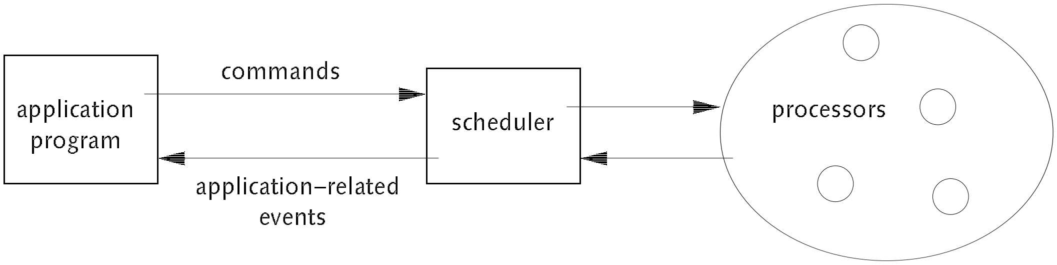

Figure 2 interaction between scheduler and application

Hence, the scheduler can be viewed as a super server whose performance and fault–tolerance can be adjusted dynamically. Robustness against failures can be achieved in three different ways, by dividing a computation into smaller parts that are performed in parallel, by replicating the same computation on several processors, and finally by using a single active processor with checkpointing. The provided operations can also be used to implement migration, i.e. to move a computation over to another machine even if there is no failure. To achieve this, it is merely required to remove the processor currently performing the computation and add a new processor on another machine.

An Example

An application that profits from this support is a Mandelbrot program that reads a set of coordinates and draws the corresponding Mandelbrot set on a viewer. Modification of the program was particularly simple, because a Mandelbrot computation can be divided into smaller sub–computations that can be executed fully in parallel. Since the time required for this computations cannot be determined simply by looking at the values of the input parameters, load balancing is achieved by producing more jobs than available machines. Hence, as soon as a processor finishes its computation and sends its result back, another job is sent to it, and thus a computation is hardly delayed by machines that are either slow or receive a computation intensive job. Failures are also handled automatically, since jobs of failed processors are re–scheduled using available ones.

Only a minor programing effort was required to obtain the processor objects that actually perform the computation. The original program was formulated as a procedure ComputeMandel so that it can be invoked from within the object handler, and was slightly altered to store the calculated values as a file, instead of drawing pixels on the screen:

```
TYPE
  MandelProc = POINTER TO MandelProcDesc;
  MandelProcDesc = RECORD (RComps.ProcessorDesc) END;

  PROCEDURE (mp: MandelProc) Handle (msg: Msgs.Msg);
    VAR F: Files.File;
  BEGIN
    IF msg IS JobMsg THEN
      RObjs.Decouple(mp);
      ComputeMandel(msg(JobMsg).pars, F);
      send F back to scheduler
      RObjs.Couple(mp)
    ELSE mp.Handle↑(msg)
    END
  END Handle;
```

To avoid blocking when computing a job, the processor object decouples the corresponding handler call before commencing computation. When processing terminates, the file is put into a result message and sent back to the scheduler. The *GotResult* procedure of the scheduler object is programmed to directly output incoming data at the appropriate location on the screen. The rest of the type–bound procedures are used to print short messages on a log viewer associated to the scheduler, thereby informing the user about the progress of the computation.

Summary

A few non–trivial applications that were developed using the Hermes system have been described. They serve as characteristic examples of how services can be introduced on top of our system in a straightforward way. Compared to services realized in other environments, we believe that we have achieved satisfactory functionality at only a modest cost. Although these tools are designed for completely different application areas, their implementation profited significantly from the asynchronous and event–oriented design of Hermes.

Chapter 7 **Conclusions**

We described a system developed to support distributed programming in the Oberon environment. Based on a general framework for externalizing and internalizing complex data, a communication facility was built that allows applications to exchange entire data structures over the network as if they were indivisible data items using asynchronous message passing primitives. An additional component is implemented to support implementation of application components which are to be installed over the network at run time, using an event–oriented approach that allows application components to be made extensible.

With these few services we believe that we have succeeded in providing powerful and intuitive abstractions that promote the development of distributed programs. The ease with which several non–trivial applications were implemented using the available primitives strongly supports our thesis, especially since programming has partly been done by students with little experience in distributed programming.

With object–oriented techniques it was possible to isolate important problems of distribution and address them in a structured and generic way. Services could thus be enriched at a later time by adding refined processing elements without any modification or recompilation. This approach not only proved to be convenient for introducing application specific behaviour but also enhanced the internal structuring of our system. In fact, extensibility turned out to be the key for addressing the problem of complex data transfer in a simple and most flexible way without requiring elaborate automatic support, thereby setting the corner–stone of our communication facility.

The decision to adopt a message–oriented model for supporting communication between activities affected the design of our system and its applications in a significant way. Using messages, rather than procedures, as a communication unit obviates the employment of stub–objects for achieving transparency, and consequently makes stub generators superfluous. Furthermore, asynchronous message passing is indeed an attractive alternative to synchronous models, because it naturally achieves decoupling between caller and callee which is often desired in distributed applications, and promotes event–oriented structuring of software, which we have found to be appropriate for a wide range of applications. It has also been demonstrated that message passing can be combined with strong support for marshalling data so that the application programs are completely relieved from data conversion and transfer problems. On the contrary, applications can use messages as conventional data types and new messages can be defined as extensions of existing message types to achieve program extensibility.

Through the compactness of our implementation (less than 20K in all) we have shown that support for distributed programming can be built at only a moderate software cost so that it can be installed on small computers, traditionally used only for single–user tasks. Even in the Oberon system which excels by its efficiency and economic design, powerful and flexible services were implemented without boosting complexity. As networks of personal workstations become a popular way of organizing computing environments, we strongly believe that support along these lines will be important for future software development.

The Oberon programming environment served as a good and robust foundation for our project. Strong typing and run–time checking allowed programming errors to be reliably detected, thereby eliminating cumbersome debugging. Also, it was this features that made it practically feasible to build a multi–threaded system that is robust against index and pointer errors yet without containing each activity in a separate address space as this is done in systems with unsafe languages. Moreover, the advanced support for dynamic loading and meta–programming considerably simplified our implementation; in other platforms this functionality would have had to be introduced explicitly, presumably at a substantial software cost.

In our opinion, the only important drawback of the Oberon system is the lack of support for true background processing and timely response to events which unnecessarily limits its potential. While we do not consider our attempt to introduce concurrency in Oberon as the ultimate solution to these problems, it is an indication that the desired functionality can be achieved without making the system complicated; we sincerely hope that this work will serve as a starting point for further development in this direction. In the course of our project, we found object finilization and code fingerprinting to be important. Both services can be implemented in Oberon without much effort. Our implementation would also have benefited from a mechanism for controlling the use of heap and disk in the system. Namely, with such a support one can guarantee that the resources needed to operate the local applications of a workstation will not be used by remote activities, which would further encourage users to have installation of remote objects enabled even if they are currently using their machines.

blank

Appendix System Size and Performance Figures

To give the reader a rough idea about the complexity of our implementation, a few data about its size and performance is shown in the following. For clarity, the Hermes system is considered separately from the Concurrent Oberon system. We also have attempted to conduct our measurements so that the performance and overhead of each component can be identified in a simple way. Performance figures are given for Ceres–2 workstations with a NS32000 processors.

The Concurrent Oberon System

Although the Oberon system has conceived as a strictly sequential system, the amount of code that was added to it to implement multitasking is relatively small (table 1).

Module	*old version*	*new version*	*increase abs.*	*increase rel.*
Kernel	3200	5100	1900	60%
Files	6700	7200	500	8%
Modules	3500	3700	200	6%
Threads	–	3800	3800	
Input	900	1100	200	22%
Oberon	5500	5600	100	2%
System	10800	13400	2600	24%
Oberon (raw)	110800	120100	9300	8%

Table 1 software complexity of the Concurrent Oberon system

The total software cost amounts about 10 kilobytes. Notably, the implementation of threads and the scheduler amounts only for about 1/3 of the total costs. A substantial part of the newly introduced code is used to modify the garbage collector to trace pointers deposited on stacks (module *Kernel*) and to provide the user with a set of auxiliary commands for monitoring the threads available in the system (module *System*). All in all, the code size of the system increased by 8% of the original Oberon system which amounts about 110 kilobytes in its simplest form (i.e. including only its default text editor).

The run time cost of the introduced multitasking facility is analyzed below. Table 2 shows the overhead of a null call to the scheduler and a context switch as experienced from a user program. Times are given in microseconds.

operation	μs
scheduler call (no switch)	26–46
scheduler call (context switch)	40–60
raw context switch	14
saving registers	+ 15

Table 2 context switching costs

By comparing these figures it can be seem that actual switching (i.e. coroutine transfer) requires about 14 microseconds. An additional penalty of approximately 15 microseconds must be paid if the general purpose and floating point registers are saved, which is the case when the scheduler is invoked from within an interrupt routine. The displayed values include an overhead of about 10 microseconds for accessing the current stack variable of the kernel via a supervisor call; if the interface of module *Kernel* were changed to export this variable, it would be possible to completely eliminate this cost.

The variance of about 20 microseconds in the data given above was obtained by inserting dummy instructions in the scheduler code, or by repositioning of the scheduling procedure within the Threads module. Since none of these changes affected the code that was actually executed during our measurements, we assume that these differences in performance are due to caching and code alignment effects. Our code in its original form (as it is used in Concurrent Oberon) consumes around 40 and 54 microseconds for a null scheduler call and a context switch, respectively.

To put the cost of multitasking into a better perspective, we conducted the following experiment: we let one thread increment a local integer variable for a given time, and then compared it to the sum of the integer values generated by 10 threads executing in parallel for the same amount of time. According to the observed results, the 10 competing threads together achieved 99% of the value produced by the single thread, which indicates that timesharing is relatively cost efficient in our system.

The Hermes System

As already explained, Hermes consists of several modules that gradually introduce new functionality in the system. The size of the individual modules and the total size of the Hermes system are displayed in table 3.

component	*size*	*increase abs.*	*increase rel.*
Concurrent Oberon (raw)	120100		
IO	3600		
NetCarriers	4000		
Msgs	4200		
RObjs	6000		
Hermes Total		17800	15%
NOP2 Compiler		122700	100%
Kepler Graphics		68700	57%

Table 3 size of the Hermes modules

The third column shows the relative increase with respect to the Concurrent Oberon system. For comparison, we also give the size of the Oberon–2 compiler and the size of a graphics editor.

To analyze the costs of message transmission, which is the main communication operation in our system, we have measured the time spent to transfer a message for the case where the destination mailbox is located on a remote machine. The message transfer time as viewed by the application was obtained by sending several messages one after the other and dividing the total delay by the number of messages sent. Notably, this is only a conservative estimation of the time needed to send a message, since when sending the next message the application is blocked until the last one has been acknowledged, which is typically not the case in a single message transmission. Measurements were made between two Ceres–2 workstations connected by a phone–net with a raw transfer rate of 250 kilobits per second. At the time we conducted our experiments, the network was lightly loaded.

Table 4 displays the corresponding results in microseconds both for an empty message and a message containing 10 four–byte integer numbers. To visualize how the total transfer cost is distributed among the various components of our system, we give the delay due to message externalization, and the time required to send data over the network. The former is approximated by measuring the time needed to transfer a message in a memory buffer. The network overhead is estimated as the sum of the time for sending a data packet and an acknowledgement over the wire. We include the acknowledgement processing in our calculation, because program execution at the sender is effectively delayed by the time it takes the network interrupt handler to read the incoming acknowledgement into a packet buffer (we assume that the time needed to read data from the network is comparable to the time needed to send it).

	empty msg	*10 int msg*	*contribution*
message externalization	540	740	12%
network	3100	4500	65–75%
application delay	4700	6000	

Table 4 time spent for sending a message

According to our results, the network time amounts for almost 70% of the message transfer cost, while approximately 12% of the total time is spent to externalize the message into a packet buffer. The remaining 18% of the costs are due to protocol processing, synchronization, and context switching.

For a rough comparison, a recently developed remote object system [Birell93a] running on Digital workstations equiped with MIPS 3000 processors and a 100 megabit network is about 10000 lines of Modula–3 code. Its remote procedure call facility which features comparable marshalling support to our implementation requires around 3300 microseconds for a null call and 3400 microseconds for a ten–integer call. Admitedly, a remote procedure call is more expensive than a message transfer, because the acknowledgment confirming receipt of the data packet is not sent as soon as the packet is received, but is generated –with some delay– from within the server program to which the request is sent. In our system, such a request–reply cooperation requires a complete exchange, and thus should come at approximately the double cost of a single message transfer (i.e. around 10000 and 12000 microseconds, respectively). Nevertheless, given the small size of our system (approximately 1300 lines of Oberon code), the moderate speed of our hardware and the fact that our system has not been designed especially for supporting synchronous communication, we believe that the achieved performance is acceptable.

References

Accetta M, Baron R, Bolosky W, Golub D, Rashid R, Tevanian A, and Young M. (1986) Mach: A New Kernel Foundation for UNIX Development, *Proc. Summer USENIX Conference*, pp. 93–112.

Almes G, Black A, Lazowska E, and Noe J. (1985) The Eden System: A Technical Review, *IEEE Trans. on Software Engineering* 11(1), pp. 43–58.

Andrews G. (1991) Concurrent Programming: Principles and Practice, Benjamin/Cummings Publishing Company.

Arbenz P, Lüthi H–P, Sprenger C, and Vogel S. (1994) SCIDDLE: A Tool for Large Scale Distributed Computing, Technical Report No. ???, Departement Informatik, ETH Zürich.

Bershad B, Ching D, Lazowska E, Sanislo J, and Schwartz M. (1987) A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems", *IEEE Trans. on Software Engineering* 13(8), pp. 880–894.

Birell A and Nelson B. (1984) Implementing Remote Procedure Calls, *ACM Trans. on Computer Systems* 2(1), 1984, pp. 39–59.

Birell A, Nelson G, Owicki S, and Wobber E. (1993) Network Objects, *Proc. ACM Symposium on Operating Systems Principle*s, pp. 217–230.

Birell A, Evers D, Nelson G, Owicki S, and Wobber E. (1993) Distributed Garbage Collection for Network Objects, Technical Report No. 116, DEC SRC.

Black A, Hutchinson N, Jul E, and Levy H. (1986) Object Structure in the Emerald System, *Proc. OOPSLA Conference*, pp. 78–86.

Bougnion E. (1994) Object–Oriented Distributed Programming with UNIX–Oberon, Diploma Thesis, Departement Informatik, ETH Zürich.

Brinch Hansen P. (1970) The Nucleus of a Mutliprogramming System, *Communications of the ACM* 13(4), pp. 238–241.

Cerf V and Kahn R. (1974) A Protocol for Packet Network Intercommunication, *IEEE Trans. on Communications* 22(5), pp. 637–648.

Chang J–M and Maxemchuk N. (1984) Reliable Broadcast Protocols, *ACM Transactions on Computer Systems* 2(3), pp. 251–273.

Chase J, Amador F, Lazowska E, Levy H, and Littlefield R. (1989) The Amber System: Parallel Programming on a Network of Mutliprocessors, *Proc. ACM Symposium on Operating Systems Principles*, pp. 147–158.

Cheriton D. (1984) The V Kernel: A Software Base for Distributed Systems, *IEEE Software* 1(2), pp. 19–42

Cheriton D. (1986) VMTP: A Transport Protocol for the Next Generation of Communication Systems, *Proc. ACM Symposium on Communication Architectures and Protocols*, pp. 406–415.

Clark D. (1985) The Structuring of Systems Using Upcalls, *Proc. ACM Symposium on Operating Systems Concepts*, pp. 171–180.

Cooper E. (1985) Replicated Distributed Programs, *Proc. ACM Symposium on Operating System Principles*, pp. 63–78.

Dijkstra E. (1968) Cooperating Sequential Processes, Programming Languages, Academic Press.

Dijkstra E. (1972) Hierarchical Ordering of Sequential Processes, *Acta Informatica* 1, pp. 115–138.

DoD. (1980) Ada Reference Manual, US Dept. of Defence.

Eberle H. (1987) Development and Analysis of a Workstation Computer, Dissertation No. 8431, Departement Informatik, ETH Zurich.

Eicken von T, Culler D, Goldstein S, and Schauser S. (1992) Active Messages: A Mechanism for Integrated Computation, *Proc. ACM SIGARCH Annual International Symposium on Computer Architecture*, pp. 256–266.

Fenart J–M, Fievet M, Huitena C, Martin B, Remille A, and Vaysseix G. (1986) OSI and TCP/IP Protocols on a UNIX System V, *Proc. Winter USENIX Conference*, pp. 46–58.

Fitzgerald R and Rashid R. (1986) The Integration of Virtual Memory Management and Interprocess Communication in Accent, *ACM Transactions on Computer Systems* 4(2), pp. 147–177.

Fleisch B and Popek G. (1989) Mirage: A Coherent Distributed Shared Memory Design, *Proc. ACM Symposium on Operating System Principles*, pp. 211–223.

Gehani N and Roome W. (1986) Concurrent C, *Software –Practice* *and Experience* 16(9), pp. 821–844.

Gifford D and Glasser N. (1988) Remote Pipes and Procedures for Efficient Distributed Communication, *ACM Trans. on Computer Systems* 6(3), pp. 258–283.

Gutknecht J. (1993) Oberon System 3 –A Realm of Persistent Objects, Internal Report, Institut für Computersysteme, ETH Zürich.

Heeb B and Noack I. (1991) Hardware Description of the Workstation Ceres–3, Technical Report No. 168, Departement Informatik ETH Zurich.

Herlihy M and Liskov B. (1982) A Value Transmission Method for Abstract Data Types, *ACM Trans. on Programing Languages and Systems* 4(4), pp. 527–551.

Hoare C. (1974) Monitors: An Operating System Structuring Concept, *Communications of the ACM* 17(10), pp. 549–557.

Jones M and Rashid R. (1986) Mach and Matchmaker: Kernel and Language Support of Object–Oriented Distributed Systems, *Proc. OOPSLA Conference*, pp. 67–77.

Kaashoek M, Tanenbaum A, Hummel S, and Bal H. (1989) An Efficient Reliable Broadcast Protocol, *ACM Operating Systems Review* 23(4), pp. 5–19.

Kalsow B and Cavalli–Sforza V. (1988) Pickles: An Implementation of Persistent Data Structures, Internal Report, DEC SRC.

Lalis S. (1991) XNet – Supporting Distributed Programming in the Oberon Environment, Technical Report No. 161, Departement Informatik, ETH Zürich.

Lalis S and Sanders B. (1994) Adding Concurrency to the Oberon System, *Proc. International Conference on Programming Languages and System Architectures*, pp. 328–344.

Li K. (1988) A Shared Virtual Memory System for Parallel Computing, *Proc. IEEE International Conference on Parallel Processing*, pp. 94–101.

Linton A and Panzieri F. (1986) A Communication System Supporting Large Datagrams on a Local Area Network, *Software –Practice and Experience* 16(3), pp. 277–289.

Liskov B, Herlihy M, Gilbert L. (1986) Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing", *Proc. ACM Symposium on Principles of Programming Languages*, pp. 150–159.

Liskov B, Curtis D, Johnson P, and Schleifer R. (1987) Implementation of Argus, *Proc. ACM Symposium on Operating Systems Principles*, pp. 111–122.

Liskov B and Shrira L. (1988) Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems, *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, pp. 260–267.

Marlin C. (1980) Coroutines, Lectures in Computer Science 95, Springer.

McQuillan J and Walden D. (1997) The ARPA Network Design Decisions, *Computing Networks* 1, pp. 243–289.

Moessenboeck H and Wirth N. (1991) The Programming Language Oberon–2, *Structured Programming* 12(4), pp. 179–195.

Nelson G, Levin R, Kalsov B, Horning J, Guttag J, Donahue J, Cardelli L, Brown M, and Birell A. (1991) Systems Programming in Modula–3, Prentice Hall.

Nichols D. (1987) Using Idle Workstations in a Shared Computing Environment, *Proc. ACM Symposium on Operating System Principles*, pp. 5–12.

Presotto D and Ritchie D. (1985) Interprocess Communication in the Eigth Edition Unix System, *Proc. Summer USENIX Conference*, pp. 309–316.

Rashid F and Robertson G. (1981) Accent: A Communication Oriented Network Operating System Kernel", *Proc. ACM Symposium on Operating Systems Principles*, pp. 64–75.

Requa J. (1985) UNIX Kernel Networking Support and the Lincs Communications Architecture, *Proc. Winther USENIX Conference*, pp. 98–103.

Schelvis M. (1989) Incremental distribution of Timestamp Packets: A New Approach to Distributed Garbage Collection, *Proc. OPPSLA Conference*, pp. 37–48.

Schmidt D and Bauknecht K. (1989) DB++ –Persistent Objects for C++, *Proc. GI/SI Fachtagung*, pp. 277–281.

Sullivan M and Anderson D. (1989) Marionette: A System for Parallel Distributed Programming Using a Master/Slave Model", *Proc. IEEE International Conference on Distributed Computing Systems*, pp. 181–188.

Sun (1990) External Data Representation, Sun Technical Notes, Sun Microsystems.

Sunshine C. (1976) Factors in Interprocess Communication Protocol Efficiency for Computer Networks, *Proc. AFIPS Conference*, pp. 571–576.

Sunshine C and Dalal Y. (1978) Connection Management in Transport Protocols, *Computer Networks* 2, pp. 454–473.

Szyperski C. (1990) Network Communication in the Oberon Environment, Technical Report No. 126, Departement für Informatik, ETH Zürich.

Szyperski C. (1992) Insight ETHOS: On Object–Orientation in Operating Systems, Dissertation No. 9884, Departement für Informatik, ETH Zürich.

Tanenbaum A. (1981) Computer Networks, Prentice Hall.

Tomlinson R. (1975) Selecting Sequence Numbers, *Proc. ACM SIGCOM/SIGOPS Interprocess Communication Workshop*, pp. 11–23.

Weck W, (1990) Support for Remote Procedure Calls in Oberon, Diploma Thesis, Departement für Informatik, ETH Zürich.

Weihl W and Liskov B. (1985) Implementation of Resilient, Atomic Data Types, *ACM Transactions on Programming Languages and Systems* 7(2), pp. 244–269.

Wirth N. (1988) The Programming Language Oberon, *Software –Practice and Exprerience* 18(7), pp. 671–690.

Wirth N and Gutknecht J. (1989) The Oberon System, *Software –Practice and Experience* 19(9), pp. 857–893.

Wirth N. (1989) Ceres–Net: A Low Cost Computer Network, *Software –Practice and Experience* 20(1), pp. 13–14.

Zimmermann H. (1980) OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection, *IEEE Trans. on Communications* 28(4), pp. 425–432.