#28

The importance of NVDMs is evident from the increasing interest in the topic [Cardenas 79], [Chi 79], [Esculier 79], [Fauser 79] and [Tsubaki 79]. We believe this interest reflects the requiremen for rapid incorporation of new and more sophisticated database products within an operationa environment.

NVDM design and implementation requires considering two major classes of issues: distributio management which has not been addressed in this paper and heterogeneity management whic has been the focus of the paper. Distribution management deals with issues related to supportin concurrent access to multiple DBMSs and has not yet been addressed in the context of XNDM However, the existing literature on distributed DBMSs provides substantial insight into suc problems [Rothnie 77].

In general, DBMS DMLs support both queries and updates. The preceding has only considered queries since this is required to establish addressability to existing data. Considering updates requires handling modification of existing data, insertion of new record instances and deletion of existing record instances. Since the network user may only see a limited portion of the data supported by a local DBMS the problem of updating views must also be considered. Although the general solution to this problem is very difficult [Dayal 79], partial solutions sufficient for the requirements of many network users may provide an alternative.

14.7. Parameter and data representation B.W. LAMPSON

In this section we consider in a broader context the problems of parameter and data representation in distributed systems. Our treatment will be somewhat abstract and, unfortunately, rather superficial.

This is really a programming language design problem, although in practice it is not usually addressed in that context. The reason is that a single bit has the same representation everywhere (except at low levels of abstraction which do not concern us here). An integer or a floating point number, to say nothing of a relational data base, may be represented by very different sets of bits on different machines, but it only makes sense to talk about the representation of data when its type is known. Thus our problem is to define a common notion of data types and suitable ways of transforming from one representation of a type to another. This kind of problem is customarily addressed in the context of language design, and we shall find it convenient to do so here.

In fact, we shall confine ourselves here to the problem of a single procedure call, possibly directed to a remote site. The type of the procedure is expressed by a declaration:

P: **procedure**(a_1: T_1, a_2: T_2, . . .) **returns** (r_1: U_1, r_2: U_2, . . .).

We shall sometimes write (a_1: T_1, a_2: T_2, . . .) → (r_1: U_1, r_2: U_2, . . .) for short. Of course, sending a message and receiving a reply can be described in the same way, as far as the representation of the data is concerned. The control flow aspects of orderly communication with a remote site are discussed in Chapter 7 and Section 14.8; here we are interested only in data representation. The function of the declaration is to make explicit what the argument and result types must be, so that the caller and callee can agree on this point, and so that there is enough information for an automatic mechanism to have a chance of making any necessary conversions. For this reason we insist that remote procedures must have complete declarations. We assert without proof that any data representation problem can be cast in this form without doing violence to its essence.

FROM DISTRIBUTED SYSTEMS: ARCHITECTURE AND IMPLEMENTATION, ED LAMPSON, PAUL, AND SIEGERT, SPRINGER, 1981.

There are three basic issues:

Binding: how to *locate* the remote procedure body which is supposed to be invoked when a particular procedure is invoked, and how to ensure that the body is *compatible* with the procedure, i.e., that it has the proper type.

Encoding: how to represent the arguments and results as self-contained collections of bits for transmission on the wire.

Conversion: if the representation of the data at the two ends is different, how to convert from one representation to the other.

The first two issues arise in any distributed system, and for that matter in conventional systems as well, where they are dealt with, often inadequately, by the linker/loader and by the parameter passing convention(s) respectively. The third is important only in heterogenous systems.

14.7.1. Types

In order to deal coherently with these problems, we need a suitable notion of data type. Following [Morris 73] loosely, we take the view that each object or value we deal with carries a set of marks which serve to identify the abstract types of which the object is a value. For example, integer is such a mark in most languages; an integer value which is also an index in an array of symbol table entries might serve to represent such an entry, and as such also carry the mark *STEntry*. A mark is simply a unique identifier, which can be attached by a marking procedure whose accessibility is suitably restricted, usually to the program responsible for implementing the corresponding type. For each mark there is a predicate which tells whether its argument carries that mark; access to these predicates is not restricted.

We can now define a *type* as a predicate on values, together with a set of operations. The terms of the predicate may include the mark predicates, as well as other expressions with Boolean values. For example, the Pascal type 0..10 corresponds to the predicate integer $\wedge\ 0 \leq x \leq 10$. The operations of this type include $+, -, *, /, =, <, :=$ and *New*.

The reason for treating types in this way can be seen by considering the role of the type system, as a primitive, but efficient program verifier. A declaration of the form

procedure *F*(*a*: *A*) **returns** (*r*: *R*) = S

means that the body S has the precondition that *a* satisfies the predicate for *A*, and in turn guarantees the postcondition that *r* will satisfy the predicate for *R*. Within S, in turn, the knowledge of the formal parameter types which is provided by the declaration can be used to check that the parameters are in turn being properly passed as arguments to other procedures.

Each object or value has a representation (a collection of bits) and a set of marks; in a statically typed language it is usually not necessary to store the marks with each object, since they can be dealt with completely before execution of the program is attempted.

A simple example of all this is types for complex numbers. Several representations are possible, for instance rectangular and polar. Since these are not interchangeable either we must choose one, which can then have the mark *Complex*, or each must have its own mark, *ComplexR* and *ComplexP* respectively. The representations are the same: a record of two reals (though the fields may be named differently). The operations have the same names and type structure (but of course different implementations):

$+, -, *, /$: (*Complex*, *Complex*) → *Complex*
Sin, *Sqrt*, . . .: *Complex* → *Complex*

14.7.2. Binding

A program may refer to procedures which are not defined within the program itself; this is an old idea which has been around since the earliest days of subroutine libraries. We can describe this situation as in the left side of figure 14-20: the program contains a set of procedure *variables*, which must eventually be *bound*, i.e., filled in with procedure *values* which are descriptors for compatible procedure bodies (on the right side of the figure) if the program is to complete successfully (at least those which are called must be filled in). The situation after binding is illustrated by the entire figure. These values can come from a variety of sources, which we classify as follows:

A *linker*, which statically examines a collection of programs, supplying procedure values by matching the name of a variable with the name of a procedure body; typically the names are taken from a single, global name space.

A *dynamic linker*, which gets control when an attempt is made to call an unbound procedure and finds a value to fill in by doing the same kind of name lookup.

The program itself, which obtains a suitable procedure value by some arbitrary computation and assigns it to the variable.

In the latter two cases it is also reasonable to consider *unbinding*, or assigning a null value to a variable. In any case, the abstract situation, as we have described it, is identical for local and remote procedures.

The need for compatibility is also identical, and it is desirable to state it precisely. Suppose we have a variable *localP*: $A_l \rightarrow R_l$ and a value *remoteP*: $A_r \rightarrow R_r$; the names of course are merely suggestive. The assignment

localP := *remoteP*

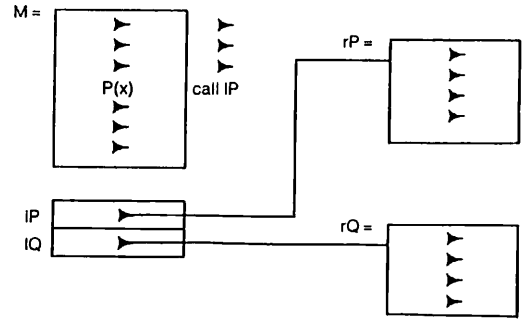

Figure 14-20: Binding a procedure identifier to a procedure body

is legal provided A_l implies A_r and R_r implies R_l. The implication of course refers to the predicates which define the types; strictly it should be

$$\forall x\text{: (Predicate}(A_l))(x) \Rightarrow (\text{Predicate}(A_r))(x)$$

This rule follows directly from the role of types in reasoning about the program. When called, *remoteP* can count on A_r, but a call of *localP* need only ensure A_l, hence A_l must imply A_r. Conversely, the caller of *localP* can count on R_l, but a return from *remoteP* need only ensure R_r, hence R_r must imply R_l; note that the roles of l and r are reversed for the result type. The common rule $A_l = A_r$ and $R_l = R_r$ is of course sufficient, but stronger than necessary. We reiterate that compatibility issues are identical for local and remote binding.

It is an obvious but sometimes overlooked fact that in order to talk about compability at all, it is necessary to have a common type system at the local and remote sites. This does *not* mean that there must be common representations. The difficulty in establishing agreement on the type system accounts for the popularity of ASCII strings as the common medium of exchange among machines, and indeed often among separately constructed programs on the same machine.

14.7.3. Encoding

In order to call a remote procedure, it is necessary to transmit the arguments to the remote machine, and later to transmit the results back to the caller. This means that the values of the arguments and results must be represented on the wire between the machines, presumably by a pile of bits which are not interpreted by the transmission mechanism. We note that the problem of obtaining such a representation is closely related to the problem of representing a value on the disk; thus a transmitted object and a permanent object stored in a file raise very similar problems. The permanent object is actually more difficult to handle in most respects, because its creator is no longer around and cannot answer questions about it when confusion arises.

To discuss a wire representation for a type T it is convenient to define another type T_w, and two procedures *Encode*: $T \rightarrow T_w$ and *Decode*: $T_w \rightarrow T$. We insist that *Decode*(*Encode*(x)) = x for all values of type T. The details of T_w are private to the implementation of T, but a value of T_w will occupy a contiguous region of storage and be assignable simply by copying bits, so that it can conveniently be sent over the wire. It is important that a T_w value still carries a mark, so that *Decode* can accept it safely. As on a single machine, if the static type checking is strong enough the mark need not occupy space in the representation. In most cases, however, such confidence in the multi-machine system is not warranted, and prudence dictates that the mark should appear explicitly, so that an error in transmission, or more likely in maintaining the compatibility of the procedure binding, can be detected before any serious damage is done.

If a value contains a pointer, it may be desirable to treat the object referenced by the pointer as a *sub-object* and include it in the encoded value. This is inappropriate if the sub-object is shared, i.e., referenced by other pointers. Frequently, however, the pointer is simply a convenient way to hold a complex object together, and transmitting the object as a whole is quite reasonable. To do this, *Encode* includes the sub-object in T_w, and *Decode* creates a new sub-object in order to obtain a new pointer which it stores in the new object. [Sollins 79] gives an extended treatment of this problem.

In some cases the representation of T_w may include a *context*. This is a fancy name for a large collection of bits which we don't want to transmit. For example, a type such as **pointer to** T is really a shorthand for **pointer to** T **relative to base** b, where in the simplest case **base** means the address space of a machine. Such a pointer can be a perfectly good value on a remote machine, but it cannot be dereferenced without access to the context provided by the base. Usually this means that it must be sent back to the source machine for dereferencing, although in some cases it may be practical to transmit the entire context.

14.7.4. Conversion

When a type T has different representations T_l and T_r in the procedure caller and the procedure body, the *Encode* and *Decode* operations of the previous section are insufficient, since they are private to a particular representation. Instead, we must have some chain of conversion procedures which can turn a T_l into a T_r through a sequence of intermediate types:

$$C_1\text{: } T_l \rightarrow IT_1\text{; } C_2\text{: } IT_1 \rightarrow IT_2\text{; } \ldots\ C_n\text{: } T_{n-1} \rightarrow T_r$$

The composition of two C's is also a conversion function. Again, we insist that the C_i be lossless: if $C_i \circ C_j$: $T \rightarrow T$, then it must be the identity. The simplest cases are pairwise conversion:

$$C_{lr}\text{: } T_l \rightarrow T_r$$

and conversion to and from a standard representation (T_w may be an attractive candidate):

$$C_{ls}\text{: } T_l \rightarrow T_s\text{; } C_{sr}\text{: } T_s \rightarrow r_s\text{; } \quad C_{rs}\text{: } T_r \rightarrow T_s\text{; } C_{sl}\text{: } T_s \rightarrow T_l$$

with their well-known nm and $n+m$ growth functions as the number of representations increases. Given a collection of types and conversion functions, it is straightforward to find the most economical conversion, and the lossless property guarantees that a different which will yield the same results. [Wallis 80] discusses these issues further.

One attractive approach to the conversion problem is decomposition. We say that a type R is *complete* if it has operations:

$$D_1\text{: } R \rightarrow T_1\text{; } D_2\text{: } R \rightarrow T_2\text{; } \ldots\ D_n\text{: } R \rightarrow T_n \text{ and}$$
$$C\text{: } (T_1, T_2, \ldots\ T_n) \rightarrow R$$

where the T_i are either *basic* types such as **integer**, or simpler than R in the obvious sense. A record type, for example, can easily be made complete: the D's yield the field values, and C constructs a record value from the fields. Now, given conversion functions for the basic types, it is easy to produce conversion functions for any complete type. This problem is discussed from a different point of view in Section 14.4.

14.8. Debugging, testing and measurement

Although it is hard to argue that the problems of debugging and measurement differ qualitatively in centralized and distributed systems, in practice there do seem to be some major variations. Most of them arise from the fact that fault tolerance and partial failures are the norm for distributed systems, whereas a centralized system is typically designed to fail all at once. There is no property of distribution which makes it necessary for this to be true, but the contrast between reliable memory and unreliable communication, and between a single scheduler and autonomous

processors certainly makes it natural. This entire area is poorly understoood; neither theory nor experience are developed to any significant extent. We shall therefore confine ourselves to a catalog of some important issues and a collection of anecdotes.

Global state. Autonomy, high communication latency, and low bandwidth mean that the concept of a state for the entire system, very natural in a centralized system, must be approached with caution. Short of stopping all the processors, it is not possible to find out what the system state is, and stopping all the processors is usually either impossible or at least highly undesirable. As a consequence:

> A meaningful *definition* of the system state is not possible; at most we may be able to define sets of possible states.
>
> *Observation* of the global state is not possible.
>
> Observing the states of individual components is possible, but it requires resources to do this and to communicate the results. This resource consumption *interferes* with the normal operation of the system, unless additional resources are dedicated for this purpose.
>
> On the other hand, the fact that a distributed system is often readily expandable may make it easy to provide additional resources for observation without any redesign.

Control. Debugging techniques for centralized systems often require resetting, stopping and single-stepping the system. In a distributed system it may be fundamental to the design that such operations are not possible, because the system is designed for high reliability and must not have single buttons which can stop it; this is the case on the Pluribus, for example. Even if global control of the system is not excluded on principle, it may be very difficult to implement. In a system like the Arpanet, for example, with its variety of hosts running a variety of operating systems not designed for distributed computing, there is no way to exercise control over a distributed system consisting of a collection of processes running on various hosts (although in this situation monitoring all the communication between them would be fairly easy). On the other hand, if the system is designed from scratch and uses a high-bandwidth local net for communication, it should be possible to stop the system nearly as easily as a collection of processes within a single operating system can be stopped.

Time and ordering of events. It is possible to synchronize the local clocks of elements in a distributed system with an accuracy which is limited roughly by the sum of errors which accumulate because of different clock rates in each machine, and errors arising from uncertainty about the time for communication between machines [Chapter 12, Lamport 78]. Unfortunately, the errors are often large, especially if a network becomes partitioned, and hence the concept of global time, like the concept of global state, is of limited value. Instead of a total ordering of the events which agrees with the ordering imposed by real time, we must settle for a partial order which agrees, or a total order which does not, and unless considerable care is taken, we get neither. In addition to its other uses, knowledge of the times when events occur and of their relative ordering is often critical for measurement and debugging.

Localizing failures. Typically one attempts to make the communication mechanism a global one at a low level of abstraction, so that the details of the physical facilities used to send a message are concealed. With a local broadcast net like the Ethernet, this happens at the lowest physical level; with a store-and-forward net like the Arpanet it requires a good deal of software. Above this level, however, it is not easy to localize failures in the communication system, and unless great care is taken, a clumsy process of disabling components one at a time will be required.

In the Ethernet, for example, a shorted transceiver on the cable will prevent any communication, and examining the entire cable for the short is not attractive. Fortunately, there is a device called a time-domain reflectometer (TDR), which sends a signal down the cable and measures the time at which various reflections return. Any impedance discontinuity causes a reflection, and thus a short can be localized within a foot or two. Furthermore, an active transmitter also introduces discontinuities, so that the TDR can locate a malfunctioning transmitter, and during periods of normal operation can maintain an up to date map of the physical location of stations on the cable. The latter is not a trivial matter when there are a hundred stations, and one is added or removed about once a day.

Good physical isolation is not enogh to prevent these problems. On the Arpanet there was a famous "black hole" event when the Harvard IMP's memory failed in such a way that the region storing the routing table always returned zeros. The table was represented so that all zero was a legal value indicating that every other node was directly connected to Harvard. The adaptive algorithm proceeded to broadcast this false information throughout the network, and most of the packets immediately converged on Harvard. As each one arrived, of course, it would immediately be sent out again, but this made matters worse, since the neighboring IMP which received it would just send it back again.

Test environment. As with observation, so with stimulus. Any collection of concurrent programs is difficult to test systematically, and a distributed system has the additional problems of variable communication delay and interference of the testing procedure with normal operation.

Performance bugs. An outstanding characteristic of any fault-tolerant system is that bugs which would prevent an ordinary system from working at all, and hence would be found and fixed, instead degrade the performance without affecting the functionality (hence the term, performance bugs). Since most distributed systems have a high degree of fault-tolerance, this phenomenon affects them strongly. Poorly chosen timeouts are a standard example of a performance bug, but there are many others. Situations have been observed on the Ethernet in which packets sent between a particular pair of machines were in error 80% of the time. It was not until systematic observations were made of the error characteristics of the net [Shoch 80a] that this was discovered; a 20% success rate was sufficient to provide adequate, if sluggish service. Such problems can be detected only by systematic and continued monitoring of performance at each level of abstraction. The cost of such monitoring is low if planned for initially.

14.8.1. A remote debugger

A powerful technique for debugging in a distributed system is to put the debugging software on another machine. This can be viewed as a variation on the old "world-swap" technique, in which the entire memory is written out to a reserved disk area and reloaded with the debugger; the process is reversed when it is time to run the user program again. It is simpler in that all the elements of the program state need not be sought out, preserved and restored, a process which can become quite messy when, for example, the microcode which interprets instructions is part of the "world." It is more complex in that a complete set of primitives must be provided by which the remote debugger can access the program being debugged.

Such a debugger has been implemented for the Alto (see Section 19.1). It requires about 400 bytes in the target machine to hold the *nub* which implements the primitives and the internet communication with the remote debugger, which of course is of the most primitive variety, but

nonetheless is capable of interacting with a debugger anywhere in the internet (see section 19.2), not simply on the same Ethernet. The primitives required are:

read and write a single memory location;

start the program at a specified location, and stop the program;

determine whether the program is stopped;

proceed from a breakpoint.

By comparison with other ways of interfacing a debugger, this approach has only one serious drawback: since the debugger and the program have a different set of I/O devices, the user located remotely cannot observe the program's output or supply any input unless special precautions have been taken. Of course, if the terminal of the debugger is physically next to the terminal of the program's machine this is not a problem, but often it is not. Even when it is, a debugging methodology which depends on program-supplied procedures to display and modify the program's data structures is inconvenient, if not entirely impractical. This problem can be solved only by requiring the user program to do its I/O through the debugger nub.

It might be thought that the nub would be vulnerable to corruption by the program, and indeed this is true, but most other debugging interfaces have similar problems. The difficulty is often overcome by putting the interface into read-only memory, microcode, or some other inaccessible place, and this can certainly be done with the remote debugging nub also.

14.8.2. Monitoring communication

A very powerful technique for observing the behavior of a distributed system is to tap its communication. This is especially easy on a local net which uses a bus (e.g., the Ethernet) or a ring (e.g., the Cambridge ring), since all the packets pass every point, and it is easy to make provision for reading them as they go by. Although this method does require additional resources (a machine to monitor the wire), it has the charm of generating no interference at all with normal operation. Software in the monitoring machine can be written to filter the input stream (e.g., to look only at packets from a particular machine) and to arrange the data for convenient display. It must be recognized that this method is not foolproof; any errors in reception by the monitoring machine cannot be corrected by retransmission in the normal way, because the sender is not aware of the fact that there is more than one receiver. The procedures which interpret the data must take this into account by tolerating some amount of bad or missing data. This method has been applied to the Ethernet in a program called PeekPup. The most common use is simply to list the basic facts about each packet (source, destination, size, type) on the screen or in a file as it goes by; manual examination of this data yields a great deal of insight into what is going on at several different levels of abstraction. More elaborate automated analysis is of course also possible.

The same idea has been applied in the Arpanet, in the form of a trace facility which is triggered by a bit in each packet. As the traced packet passes through each IMP, an information packet is constructed and sent to a monitor machine. This method does, of course, interfere with the normal operation of the network.

14.8.3. Event logs

Perhaps the most powerful single technique for understanding the behavior of distributed systems (or for that matter, of most complex systems) is an event log. To use this technique, the program is instrumented with calls on a logging procedure at suitable points, in some way which allows the call to be bypassed at very low cost. This procedure writes a log entry which contains the time, a type code, and some data. By setting switches in the program even while it is running, the amount of information to be logged can be controlled, and perhaps the log action can be modified to collect counts or histograms instead.

This arrangement has two important properties:

Instrumentation is always installed and can be activated at any time; hence it can be turned on at low cost, perhaps in response to an observed malfunction or dgradation in performance. Events which occur infrequently can be logged at all times, so that a record is available in case of disaster. Similarly, basic statistics of bandwidth, latency and load can be continuously accumulated.

The perhaps complex task of analyzing the data is not done in the running system, but is deferred to a more convenient time and place. The log has enough raw data to permit many different kinds of analysis to be done.

In a distributed system, the network provides a nice place to write the log. Packets can be put on the net and addressed to a logging machine; if no such machine is present, the packets can be dropped on the floor at small cost. This scheme has had a number of applications on the Ethernet. Several hundred machines routinely run diagnostics when idle, and report the results every few minutes to a maintenance center. Another application is a general logging package called Metric [McDaniel 77], which provides procedures for putting log packets on the net, and a program for collecting the packets and doing some analysis.

14.9. Remote procedure calls

One of the major problems in constructing distributed programs is to abstract out the complications which arise from the transmission errors, concurrency and partial failures inherent in a distributed system. If these are allowed to appear in their full glory at the applications level, life becomes so complicated that there is little chance of getting anything right. A powerful tool for this purpose is the idea of a *remote procedure call.* If it is possible to call procedures on remote machines with the *same* semantics as ordinary local calls, the application can be written without concern for most of the complications This goal cannot be fully achieved without the transaction mechanism of Chapter 11, but we show in this section how to obtain the same semantics, except that the action of the remote call may occur more than once. Our treatment uses the definitions of Sections 11.3 and 11.4 for processor and communication failures and stable storage. Following [Lamport 78], we write $a \rightarrow b$ to indicate that the event a precedes the event b; this can happen because they are both in the same process, or because they are both references to the same datum, or as the transitive closure of these immediate relations. Recall that physical communication is modeled by a datum called the message; transmitting a message from one process to another involves two transitions in the medium, the *Send* and the *Receive.*

A remote procedure call consists of several events. There is the *main call*, which consists of the following events:

the call *c*;
the start *s*;
the work *w*;
the end *e*;
the return *r*.

The ones on the left occur in the calling machine, the ones on the right in the machine being called. The events which detail the message transmission have been absorbed into the ones listed. These events occur in the order indicated (i.e., each precedes the next). In addition, there may be *orphan* events *o* in the receiver, consisting of any prefix of the transitions indicated. These occur because of duplicated call messages, which can arise from failures in the communication medium, or from timeouts or crashes in the caller which are followed by a retry. The orphans all follow the call and precede the rest of the main call. Difficulties arise, however, in guaranteeing the order of the orphans relative to the rest of the main call.

14.9.1. The no-crash case

What we want is $o \rightarrow s$: all the orphans precede the start of the main call. This gives us semantics as close as we can expect to an ordinary procedure call without the machinery of transactions; the only peculiarity is that the work may be partly done several times before being completed. A straightforward sequence numbering scheme, together with explicit queuing of successive messages from the same process, will give us this property in the absence of crashes. The scheme is detailed in figure 14-21. Note that there are two separate things going on here:

- Ensure that messages are accepted in the order they are sent and duplicates rejected, to eliminate the errors of the communication medium (the sequence numbering does this);
- Ensure that for a given calling process the work is strictly serial, to give the proper semantics if the caller times out and resends the message.

It is possible to make one mechanism do both jobs, either by sequence numbering the messages separately for each process, or by serializing all the work for a processor. The reason for two mechanisms is performance:

- The sequence numbering requires some kind of stable storage, and a record of each "connection." It is expensive to maintain this information for each process, but cheap to maintain it for each processor, both because there are many fewer and because it needs to be updated only when the processor crashes.
- The serialization is logically required only for work within a single process. To apply it more broadly reduces the amount of concurrency, perhaps drastically.

This would also work in the presence of crashes if each crashed process always reattempted its call right after the crash. This could be accomplished by doing a *Save* just before each call (see 11.4.2), an idea which we reject because we are unwilling to pay two *StablePut* operations per remote call. Furthermore, this design would make it impossible to time out a call and do something else without the danger of orphans. Clearly, a realistic system must be able to do this. Other methods must therefore be adopted to deal with crashes and timeouts which do not repeat; they are considered in Section 14.9.2.


```
{ Remote procedures, using Send and Receive for messages, and UniqueId for unique identifiers.. }
type ID = 0..2^64; const timeout=...;
type Message = record
  state: (call, return); source, dest: Processor; id, request: ID; action: procedure; val: Value end;

{ The one process which receives and distributes messages }
var m: Message; var s: Status; while true do begin (s, m) := Receive(); if s=good then
    if m.state=call and OKtoAccept(m) then StartCall(m)
    else if m.state=return then DoReturn(m) end;

{ Make calls }
monitor RemoteCall = begin
type CallOut = record m: Message; received: Condition end; var callsOut: set of ↑CallOut := ( );
entry function DoCall(d: Processor, a: procedure, args: Value): Value = var c: ↑CallOut; begin
    New(c); with c↑ do with m do begin
        source := ThisMachine(); request := UniqueID(); dest := d; action := a; val := args;
        state := call; callsOut := callsOut + c { add c to the callsOut set };
        repeat id := UniqueID(); Send(dest, m); Wait(received, timeout) until state=return;
        DoCall := val; Free(c) end end;
entry procedure DoReturn(m: Message) =
  var c: ↑CallOut; for c in callsOut do if c↑.m.id=m.id then begin
    c↑.m := m; callsOut := callsOut - c { Remove c from callsOut }; Signal(c↑.received) end;
end RemoteCall

{ Serialize calls from each process, and assign work to worker processes }
type CallIn = record m: Message; work: Condition end
monitor CallServer = begin var callsIn, pool: set of ↑CallIn := ( );
entry procedure StartCall(m: Message) = var w, c: ↑CallIn; begin
    w := ChooseOne(pool) {waits if the pool is empty};
    for c in callsIn do if c↑.m.request=m.request then begin c↑.m.id := id; return; end
    pool := pool - w; callsIn := callsIn + w; w↑.m := m; Signal(w↑.work) end;
entry procedure EndCall(w: ↑CallIn) = begin
    Send(w↑.m.source, w↑.m); callsIn := callsIn - w; pool := pool + w; Wait(w↑.work) end;
end CallServer

{ The worker processes which execute remotely called procedures }
var c: ↑CallIn; New(c); c↑.m.source := nil; EndCall(c); with c↑.m do
while true do begin val := action(val); state := return; EndCall(c) end;

{ Suppress duplicate messages. Needn't be a monitor, since it's called only from the receive loop. }
type Connection=record from: Processor; lastID: ID end; var connections: set of Connection:=( );
function OKtoAccept(m: Message): Boolean =var c: ↑Connection; with m do begin
    for c in connections do if c↑.from=source then begin
        if id≤ c↑.lastID then return false; c↑.lastID := id; return true end;
    { No record of this processor. Establish connection. }
    if action=UniqueID then return true { Avoid an infinite loop; OK to duplicate this call. };
    { For good performance the next two lines should be done in a separate process. }
    New(c); c↑.from := source; c↑.lastID := DoCall(source, UniqueID, nil);
    connections := connections + c; return false { Suppress the first message seen. } end
```

Figure 14-21: Algorithm for remote procedure calls

14.9.2. The crash case

The algorithm of figure 14-21 ensures $c \rightarrow o \rightarrow s \rightarrow w \rightarrow e \rightarrow r$ for any remote call. If the caller crashes (or gives up), however, and starts doing something else, it provides no guarantee that $o \rightarrow n$, where n is part of the something else. This guarantee can be provided in two ways.

Deadline with refreshing. We associate with a *root* process (one which is not working on a remote call) a deadline d, and ensure that no event in that process (including events which are part of remote calls initiated by the process) can continue past d. Now $o \rightarrow n$ can be guaranteed by waiting until after d before embarking on n. If we never set any d more than d_{max} in the future, then it is sufficient to wait for $d_{max}+e_{max}$, where e_{max} is the maximum synchronization error between clocks; we don't have to consult any recorded d to know how long to wait.

For this to work, we need

a way to abort a process;

some variation of clocks synchronized throughout the system.

It is sufficient to have local clocks plus an upper bound on the time a received message spent in the communication medium, but this is tantamount to having synchronized clocks [Lamport 78].

If a process has passed the deadline, it can just be aborted and the guarantee will be met. Alternatively, the deadline can be *refreshed* by consulting with the caller; this chain of consultation will run back to the root process, so that all machines involved in the current computation of the process will be refreshed. Refreshing only requires checking that there is a chain of *callsOut* entries leading back to the root, so it requires little computation. It does add some complication, but has the advantage that an expensive computation will not be abandoned because it is unlucky enough to pass its deadline.

The more serious objection to this scheme that it isn't clear how to choose the timeout interval. If it is too long, crash recovery is slowed down. If it is too short, there is too much refreshing. However, the following calculation is encouraging. Suppose that we have a system in which a disk access takes 50 ms of elapsed time and 3 ms of computing; the latter number is quite small. Also suppose that the cost of a refresh is three times the cost of a disk access (10 ms). Then to keep the cost of refreshing down to 1%, we must do 300 disk accesses per refresh; this takes 15 seconds. Delaying a crash recovery by 15 seconds does not seem too bad. Of course if communication is very slow, the real time for the refresh may be unacceptable, but even several seconds for communication time does not seem intolerable.

It is also interesting to note that the cost of timeouts is related to the reality of the possibility that pre-crash computation will continue after a crash. If all remote calls are short, they will never time out.

Extermination. An alternative approach is that after a crash, no action is taken until all outstanding activity has been explicitly stopped. Activity on the crashed machine is obviously no problem, but all remote calls must also be exterminated. Since information about which machines these are on (the *hit list*) must be kept in stable storage, it is not practical to keep it completely accurate. Instead, it must be updated periodically. When a call to a machine not on the list is made, the list must be updated before the call is started; for this reason, it is probably a good idea to use an aging scheme to delete machines from the list.

When machine A tells machine B to exterminate A's calls, B must be able to tell which of its processes are working on A's calls. Furthermore, B is responsible for any remote calls made by these workers. There are two cases:

If B is alive, the *callsOut* list contains all its outstanding calls; it is easy to tell which ones are on behalf of A, and a list of the machines involved is returned to A and added to A's hit list. When A hits these machines, they must exterminate B's calls made on behalf of A; each call must therefore carry the whole set of machines in the call stack at the time the call is made.

If B is recovering from a crash itself, it returns its whole hit list (or at least the unfulfilled portion remaining) to A. This will result in a certain amount of waste motion, but only when two machines recover at about the same time, and it avoids any possibility of deadlock.

Keeping the call history is unattractive, since it is an extra cost imposed even when there are no crashes. It can be avoided by making B do the exterminations for which it is responsible, rather than passing them back to A. Some care is then needed to avoid deadlock if B is recovering.

The charm of extermination is that it costs nothing unless there is a crash. The trouble with it is that if some machine on the hit list is down, or if the communication network is partitioned into two subnets which temporarily cannot communicate, it will be impossible for crash recovery to complete. This seems like an overwhelming drawback which makes it impossible to rely on extermination alone.

The other possibility is to relax the guarantee: instead of ensuring $o \rightarrow n$, we will promise only $n \rightsquigarrow o$. This means that computation may continue after a crash in real time, but there will always be some serialization in which it completes before any new activity. To implement this we use the idea of infection. Every object carries a level of infection i. Whenever a process interacts with a datum, the i of each is set to the maximum of the two. If this requires increasing the level of a process, it must be aborted or refreshed. Thus we have

$$p \rightarrow d \Rightarrow i_p \leq i_d$$
$$d \rightarrow p \Rightarrow i_d \leq i_p$$

and in general

$$p \rightarrow q \Rightarrow i_p \leq i_q$$

After a process crashes, its level is set larger than it was before the crash, and hence larger than any orphan from before the crash: $i_n > i_o$. Since $n \rightarrow o$ implies $i_n \leq i_o$, it is ruled out as desired.

Keeping a level for each datum is unattractive, especially since it has to be in stable storage. More attractive is to aggregate all the data in a single processor and keep a single level for the whole processor (and hence for every process running on that processor). The main problem with this scheme is that if there is any communication at all between two parts of a system, both will quickly rise to the same level. Since the level increases after each crash, a given processor will tend to see an increase in level every time any other processor crashes. Each such increase requires a stable storage write, as well as refreshing of some number of processes.

It may be possible to avoid this non-linearity by keeping more information in the level. Another simple scheme would keep the level as an array, one integer for each processor in the system. Now the cost of a crash is minimal, but storing and updating all the levels is non-linear. An intermediate scheme would organize the processors in some hierarchy, and keep a fixed amount of level information, in detail for processors which are changing frequently, and in large aggregates for those which are static or never communicated with. All this seems rather complicated.

The most attractive scheme is a combination of extermination with deadline or infection, since extermination is clearly preferable except when some targets cannot be reached. Extermination plus deadline allows a large d_{max} to be used; this will seldom force a long wait on crash recovery, because extermination will usually work. Extermination plus infection means that the level only needs to be increased when extermination fails; since this happens seldom, a single level will be acceptable even for a very large system.