# 24. Network Objects

We have studied how to build up communications from physical signals to a reliable message channel defined by the `Channel` spec in handout 21 on distributed systems. This channel delivers bytes from a sender to a receiver in order and without loss or duplication as long as there are no failures; if there are failures it may lose some messages.

Usually, however, a user or an application program doesn't want reliable messages to and from a fixed party. Instead, they want access to a named object. A user wants to name the object with a World Wide Web URL (perhaps implicitly, by clicking on a hypertext link), and perhaps to pass some parameters that are supplied as fields of a form; the user expects to get back a result that can be displayed, and perhaps to change the state of the object, for instance, by recording a reservation or an order. A program may want the same thing, or it may want to call a procedure or invoke a method of an object.

In both cases, the object name should have universal scope; that is:

> It should be able to refer to an object on any computer that you can communicate with.
>
> It should refer to the same object if it is copied to any computer that you can communicate with.

As we learned when we studied naming, it's possible to encode method names and arguments into the name. For example, the URL

```
http://altavista.digital.com/cgi-bin/query?&what=web&q=butler+lampson
```

could be written in Spec as `Altavista.Query("web", {"butler", "lampson"})`. So we can write a general procedure call as a path name. To do this we need a way to encode and decode the arguments; this is usually called 'marshaling' and 'unmarshaling' in this context, but it's the same mechanism we discussed in handout 7.

So the big picture is clear. We have a global name space for all the objects we could possibly talk about, and we find a particular object by simply looking up its name, one component at a time. This summary is good as far as it goes, but it omits a few important things.

- *Roots*. The global name space has to be rooted somewhere. A Web URL is rooted in the Internet's Domain Name Space (DNS).
- *Heterogeneity*. There may be a variety of communication protocols used to reach an object, hardware architectures and operating systems implementing it, and programming languages using it. Although we can abstract the process of name lookup as we did in handout 12, by viewing the directory or context at each point as a function `N -> (D + V)`, there may be very different code for this lookup operation at different points. In a URL, for example, the host name is looked up in DNS, the next part of the name is looked up by the HTML server on that host, and the rest is passed to some program on the server.





- *Efficiency*. If we anticipate lots of references to objects, we will be concerned about efficiency. There are various tricks that we can use to make things run faster:

  > Use specialized interfaces to look up a name. An important case of this is to pass a whole path name along to the lookup operation so that it can be swallowed in one gulp, rather than looking it up one simple name at a time.
  >
  > Cache the results of looking up prefixes of a name.
  >
  > Change the representation of an object name to make it efficient in a particular situation. This is called 'swizzling'. One example is to encode a name in a fixed size data structure. Another is to make it relative to a locally meaningful root, in particular, to make it a virtual address in the local address space.

- *Fault tolerance*. In general we need to deal with both volatile and stable (or persistent) objects. Volatile objects may disappear because of a crash, in which case there has to be a suitable error returned. Stable objects may be temporarily unreachable. Both kinds of objects may be replicated for availability, in which case we have to locate a suitable replica.
- *Location transparency*. Ideally, local and remote objects behave in exactly the same way. In fact, however, there are certainly performance differences, and methods of remote objects may fail because of communication failure or failure of the remote system.
- *Data types and encoding*. There may be restrictions on what types of values can be passed as parameters to methods, and the cost of encoding may vary greatly, depending on the encoding and on whether encoding is done by compiled code or by interpreting some description of the type.
- *Programming issues*. If the objects are typed, the type system must deal with evolution of the types, because in a big system it isn't practical to recompile everything whenever a type changes. If the objects are garbage collected, there must be a way to know when there are no longer any references to an object.

Another way of looking at this is that we want a system that is universal, that is, independent of the details of the code, in as many dimensions as possible.

| *Function* | *Independent of* | *How* |
|---|---|---|
| Transport bytes | Communication protocol | Reliable messages |
| Transport meaningful values | Architecture and language | Encode and decode<br>Stubs and pickles |
| Network references | Location, architecture, and language | Globally meaningful names |
| Request-response | Concurrency | Server: work queue<br>Client: waiting calls |
| Evolution | Version of an interface | Subtyping |
| Fault tolerance | Failures | Replication and failover |
| Storage allocation | Failures, client programs | Garbage collection |

There are lots of different kinds of network objects, and they address these issues in different ways and to different extents. We will look closely at two of them: Web URLs, and Modula-3 network objects. The former are intended for human consumption, the latter for programming, and indeed for fairly low level programming.

## Web URLs

Consider again the URL

```
http://altavista.digital.com/cgi-bin/query?&what=web&q=butler+lampson
```

It makes sense to view `http://altavista.digital.com` as a network object, and an HTTP `Get` operation on this URL as the invocation of a `query` method on that object with parameters `(what:="web", q:="butler+lampson")`. The name space of URL objects is rooted in the Internet DNS; in this example the object is just the host named by the DNS name plus the port (which defaults to `80` as usual). There is additional multiplexing for the RPC server `cgi-bin`. This server finds the procedure to run by looking up `query` in a directory of scripts and running it.

HTTP is a request-response protocol. Internet TCP is the transport. This works in the most straightforward way: there is a new TCP connection for each HTTP operation (although the latest version, HTTP 1.2, has provision for caching connections, which cuts the number of round trips and network packets by a factor of 3 when the response data is short). The number of instructions executed to do an invocation is not very important, because it takes a user action to cause an invocation.

In the invocation, all the names in the path name are strings, as are all the parameters. The data type of the response is always HTML. This, however, can contain other types. Initially GIF (for images) was the only widely supported type, but several others (for example, JPEG for images, Java and ActiveX for code) are now routinely supported. An arbitrary embedded type can be





handled by dispatching a ‘helper’ program such as a Postscript viewer, a word processor, or a spreadsheet.

It’s also possible to do a `Put` operation that takes an HTML value as a parameter. This is more convenient than coding everything into strings in a `Get`. Methods normally ignore parameters that they don’t understand, and both methods and clients ignore the parts of HTML that they don’t understand. These conventions provide a form of subtyping.

There is no explicit fault tolerance, though the Web inherits fault-tolerance for transport from IP and the ability to have multiple servers for an object from DNS. In addition, the user can retry a failed request. This behavior is consistent with the fact that the Web is used for casual browsing, so it doesn’t really have to work. This usage pattern is likely to evolve into one that demands much higher reliability, and a lot of the code will have to change as well to support it.

Normally objects are persistent (that is, stored on the disk) and read-only, and there is no notion of preserving state from one operation to the next, so there is no need for storage allocation. There is a way to store server state in the client, using a data structure called a ‘cookie’; the user is responsible for getting rid of these. Cookies are often used as pointers back to writeable state in the server, but there are no standard ways of doing this.

As everyone knows, the Web has been extremely successful. It owes much of its success to the fact that an operation is normally invoked by a human user and the response is read by the same user. When things go wrong, the user either gives up, makes the best of it, or tries something else. It’s extremely difficult to write programs that use HTTP, because there are so many things that can happen.

## Modula-3 network objects

We now look at the Module-3 network object system, which has an entirely different goal: to be used by programs. The things to be done are the same: name objects, encode parameters and responses, process request and response messages. However, most of the coding techniques are quite different. This system is described in the paper by Birrell et al. (handout 24). It addresses all of these issues in the table above except for fault-tolerance, and provides a framework for that as well. These network objects are closely integrated with Modula-3’s strongly typed objects, which are similar to the typed objects of C++, Java, and other ‘object-oriented’ programming languages.

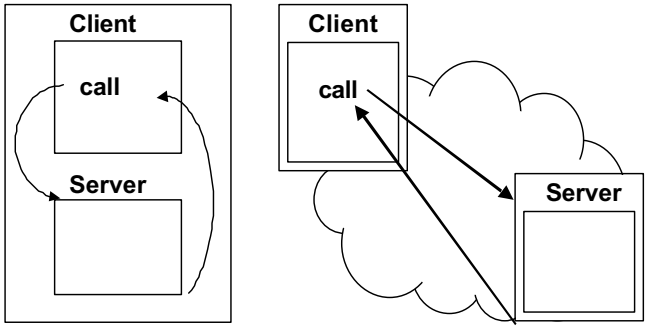
Why objects, rather than procedures? Because objects subsume the notions of procedure, interface, and reference/pointer. By an object we mean a collection of procedures that operate on some shared state; an object is just like a Spec module; indeed, its behavior can be defined by a Spec module. An essential property of an object is that there can be many codes for the same interface. This is often valuable in ordinary programming, but it’s essential in a distributed system, because it’s normal for different instances of the same kind of object to live on different machines. For example, two files may live on different file servers.

Although in principle every object can have its own procedures to implement its methods, normally there are lots of objects that share the same procedure code, each with its own state. A set of objects with the same code is often called a ‘class’. The standard code for an object is a

record that holds its state along with a pointer to a record of procedures for the class. Indeed, Spec classes work this way.

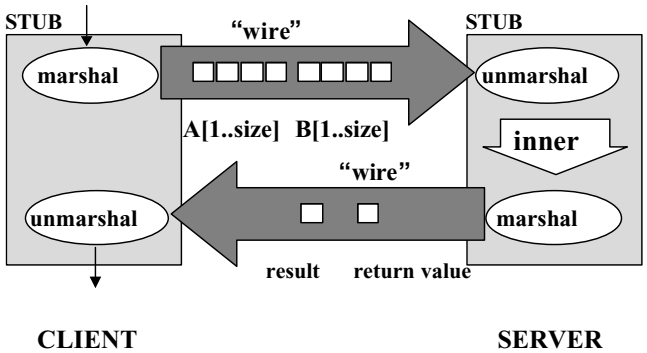
## The basic idea

We begin with a spec for network objects. The idea is that you can invoke a method of an object transparently, regardless of whether it is local or remote.




If it's local, the code just invokes the method directly; if it's remote, the code sends the arguments in a message and waits for a reply that contains the result. A real system does this by supplying a 'surrogate' object for a remote object. The surrogate has the same methods as the local object, but the code of each method is a 'stub' or 'proxy' that sends the arguments to the remote object and waits for the reply. The source code for the surrogate class is generated by a 'stub generator' from the declaration of the real class, and then compiled in the ordinary way.

**void** inner(**long** size, **long** A[], **long** B[], **long** *result)




We can't do this in a general way in Spec. Instead, we change the call interface for methods to a single procedure `Call`. You give this procedure the object, the method, and the arguments (with type `Any`), and it gives back the result. This gives us clumsy syntax for invoking a method,





`Call(o, "meth", args)` instead of `o.meth(args)`, and sacrifices static type checking, but it also gives us transparent invocation for local and remote objects.

An unrealistic form of this is very simple.

```
MODULE Object0 =

TYPE O          =  Method -> (PROC (Any) -> Any)      % Object
     Method     =  String                             % Method name

PROC Call(o, method, any) -> Any RAISES {failed} =
        RET o(method)(any)

END Object0
```

What's wrong with this is that it takes no account of what happens when there's a failure. The machine containing the remote object might fail, or the communication between that machine and the invoker might fail. Either way, it won't be possible to satisfy this spec. When there's a failure, we expect the caller to see a `failed` exception. But what about the method? It may not be invoked at all, or it may be invoked but no result returned to the caller, or it may still be running after the caller gets the exception. This third case can arise if the remote object gets the call message, but communication fails and the caller times out while the remote method is still running; such a call is called an 'orphan'. The following spec expresses all these possibilities; `Fork(p, a)` is a procedure that runs `p(a)` in a separate thread. We assume that the atomicity of the remote method when there's a failure (that is, how much of it gets executed) is already expressed in its definition, so we don't have to say anything about it here.

```
MODULE Object =

TYPE O          =  Method -> (PROC (Any) -> Any)      % Object
     Method     =  String                             % Method name

VAR  failure    :  Bool := false

PROC Call(o, method, any) -> Any RAISES {failed} =
     RET o(method)(any)
     [] failure =>
          BEGIN SKIP [] o(method)(any) [] Fork(o(method), any) END;
          RAISE failed

END Object
```

Now we examine basic code for this spec in terms of messages sent to and from the remote object. In the next two sections we will see how to optimize this code.

Our code is based on the idea of a `Space`, which you should think of as the global name of a process or address space. Each object and each thread is local to some space. An object's state is directly addressable there, and its methods can be directly invoked from a thread local to that space. We assume that we can send messages reliably between spaces using a channel `Ch` with the usual `Get` and `Put` procedures. Later on we discuss how to code this on top of standard networking.

For network objects to work transparently,

we must have a globally valid name for an object,

we must be able to find its space from its global name, and

we must be able to convert between local and global names.

We go from global to local in order to find the object in its own space to invoke a method; this is sometimes called 'swizzling'. We go from local to global to send (a reference to) the object from its own space to another space; this is sometimes called 'unswizzling'.

Looking at the spec, the most obvious approach is to simply encode the value of an `O` to make it remote. But this requires encoding procedures, which is fraught with difficulty. The whole point of a procedure is that it reads and changes state. Encoding a function, as long as it doesn't depend on global state, is just a matter of encoding its code, since given the code it can execute anywhere. Encoding a procedure is not so simple, since when it runs it has to read and change the same state regardless of where it is running. This means that the running procedure has to communicate with its state. It could do this with some low level remote read and write operations on state components such as bytes of memory. Systems that work this way are called 'distributed shared memory' systems,. The main challenge is to make the reads and writes efficient in spite of the fact that they involve network communication. We will study this problem in handout 30 when we discuss caching.

This is not what remote procedures or network objects are about, however. Instead of encoding the procedure code, we encode a *reference* to the object, and send it a message in order to invoke a method. This reference is a `Remote`; it is a path name that consists of the `Space` containing the object together with some local name for the object in that space. The local name could be just a `LocalObj`, the address of the object in the space. However, that would be rather fragile, since any mistake in the entire global system might result in treating an arbitrary memory address as the address of an object. It is prudent to name objects with meaningless object identifiers or `OId`'s, and add a level of indirection for exporting the name of a local object, `export: OId -> LocalObj`. We thus have `Remote = [space, oid]`.

We summarize the encoding and decoding of arguments and results in two procedures `Encode` and `Decode` that map between `Any` and `Data`, as described in handout 7. In the next section we discuss some of the details of this process.

```
MODULE NetObj =                                          % codes Object

TYPE O          =  (LocalObj + Remote)
     LocalObj   =  Object.O                              % A local object
     Remote     =  [space, oid]                          % Wire Rep for an object
     OId        =  Int                                   % Object Identifier
     Space      =  Int                                   % Address space

     Data       =  SEQ Byte
     CId        =  Int                                   % Call Identifier
     Req        =  [for: CId, remote, method, data]      % Request
     Resp       =  [for: CId,                 data]      % Response
     M          =  (Req + Resp)                          % Message
```





```
CONST r          :  Space := ...
     sendSR      := Ch.SR{s := r}

VAR  export      :  Space -> OId -> LocalObj          % One per space

PROC Call(o, method, any) -> Any RAISES {failed} =
     IF   o IS LocalObj => RET o(method)(any)
     [*]  VAR cid := NewCId(), to :=o.space |
          Ch.Put(sendSR{r := to}, Req{cid, remote, method, Encode(any)});
          VAR m |
               IF   << (to, m) := Ch.Get(r); m IS Resp /\ m.for = cid => SKIP >>;
                    RET Decode(m.data)
               []   Timeout() => RAISE failed
               FI
     FI
```

After sending the request, `Call` waits for a response, which is identified by the right `CId` in the `for` field. If it hasn't arrived by the time `Timeout()` is true, `Call` gives up and raises `failed`.

Note the Spec hack: an atomic command that gets from the channel only the response to the current `cid`. Other threads, of course, might assign other `cid`'s and extract their responses from the same space-to-space channel. Code has to have this demultiplexing in some form, since the channel is between spaces and we are using it for all the requests and responses between those two spaces. In a real system the calling thread registers its `CId` and wait on a condition. The code that receives messages looks up the `CId` to find out which condition to signal and where to queue the response.

```
THREAD Server() =
     DO VAR m, from: Space, remote, result: Any |
          << (from, m) := Ch.Get(r); m IS Req => SKIP >>;
          remote := m.remote;
          IF remote.space = r => VAR local := export(r)(remote.oid) |
               result := local(m.method)(Decode(m.data));
               Ch.Put(sendSR{r := from}, Resp{m.for, Encode(result)})
          [*] ...                                      % not local object; error
          FI
     OD
```

Note that the server thread runs the method. Of course, this might take a while, but we can have as many of these server threads as we like. A real system has a single receiving thread, interrupt routine, or whatever that finds an idle server thread and gives it a newly arrived request to work on.

```
FUNC Encode(any ) -> Data = ...
FUNC Decode(data) -> Any  = ...

END NetObj
```

We have not discussed how to encode exceptions. As we saw when we studied the atomic semantics of Spec, an exception raised by a routine is just a funny kind of result value, so it can be coded along with the ordinary result. The caller checks for an exceptional result and raises the proper exception.

This module uses a channel `Ch` that sends messages between spaces. It is a slight variation on the perfect channel described in handout 20. This version delivers all the messages directed to a particular address, providing the source address of each one. We give the spec here for completeness.

```
MODULE PerfectSR[
    M,                                          % Message
    A ] =                                       % Address

TYPE Q          =  SEQ M                        % Queue: channel state
    SR          =  [s: A, r: A]                 % Sender - Receiver

VAR  q          := (SR -> Q){* -> {}}           % all initially empty

APROC Put(sr, m)     = << q(sr) := q(sr) + {m} >>

APROC Get(r: A) -> (A, M) = << VAR sr, m | sr.r = r /\ m = q(sr).head =>
    q(sr) := q(sr).tail; RET (sr.s, m) >>

END PerfectSR

MODULE Ch = PerfectSR[NetObj.M, NetObj.Space]
```

Now we explain how types and references are handled, and then we discuss how the space-to-space channels are actually coded on top of a wide variety of existing communication mechanisms.

## Types and references

Like the Modula 3 system described in handout 25, most RPC and network object systems have static type systems. That is, they know the types of the remote procedures and methods, and take advantage of this information to make encoding and decoding more efficient. In `NetObj` the argument and result are of type `Any`, which means that `Encode` must produce a self-describing `Data` result so that `Decode` has enough information to recreate the original value. If you know the procedure type, however, then you know the types of the argument and result, and `Decode` can be type specific and take advantage of this information. In particular, values can simply be encoded one after another, a 32-bit integer as 4 bytes, a record as the sequence of its component values, etc., just as in handout 7. The `Server` thread reads the object `remote` from the message and converts it to a local object, just as in `NetObj`. Then it calls the local object's `disp` method, which decodes the method, usually as an integer, and switches to method-specific code that decodes the arguments, calls the local object's method, and encodes the result.

This is not the whole story, however. A network object system must respect the object types, decoding an encoded object into an object of the same type (or perhaps of a supertype, as we shall see). This means that we need global as well as local names for object types. In fact, there are in general *two* local types for each global type `G`, one which is the type of local objects of type `G`, and another which is the type of remote objects of type `G`. For example, suppose there is a network object type `File`. A space that implements some files will have a local type `MyFile` for its code. It may also need a surrogate type `SrgFile`, which is the type of surrogate objects that are implemented by a remote space but have been passed to this one. Both `MyFile` and `SrgFile`





are subtypes of `File`. As far as the runtime is concerned, these types come into existence in the usual way, because code that implements them is linked into the program. In Modula 3 the global name is the 'fingerprint' `FP` of the type, and the local name is the 'typecode' `TC`. The stub code for the type registers the local-global mapping with the runtime in tables `FPtoTC: FP -> TC` and `TCtoFP: TC -> FP`.[1]

When a network object `remote` arrives and is decoded, there are three possibilities:

- It corresponds to a local object, because `remote.space = r`. The `export` table maps `remote.oid` to the corresponding `LocalObj`.
- It corresponds to an existing surrogate. The `surrogates` table keeps track of these in `surrogates: Space -> Remote -> LocalObj.` In handout 24 the `export` and `surrogates` tables are combined into a single `ObjTbl`.
- A new surrogate has to be created for it. For this to work we have to know the local surrogate type. If we pass along the global type with the object, we can map the global type to a local (surrogate) type, and then use the ordinary `New` to create the new surrogate.

Almost every object system, including Modula 3, allows a supertype (more general type) to be 'narrowed' to a subtype (more specific type). We have to know the smallest (most specific) type for a value in order to decide whether the narrowing is legal, that is, whether the desired type is a supertype of the most specific type. So the global type for the object must be its most specific type, rather than some more general one. If the object is coming from a space other than its owner, that space may not even have any local type that corresponds to the objects most specific type. Hence the global type must include the sequence of global supertypes, so that we can search for the most specific local type of the object.

It is expensive to keep track of the object's sequence of global types in every space that refers to it, and pass this sequence along every time the object is sent in a message. To make this cheaper, in Modula 3 a space calls back to the owning space to learn the global type sequence the first time it sees a remote object. This call is rather expensive, but it also serves the purpose of registering the space with the garbage collector (making the object 'dirty').

This takes care of decoding. To encode a network object, it must be in `export` so that it has an `OId`. If it isn't, it must be added with a newly assigned `OId`.

Where there are objects, there must be storage allocation. A robust system must reclaim storage using garbage collection. This is especially important in a distributed system, where clients may fail instead of releasing objects. The basic idea for distributed garbage collection is to keep track for each exported object of all the spaces that might have a reference to the object. A space is supposed to register itself when it acquires a reference, and unregister itself when it gives up the reference (presumably as the result of a local garbage collection). The owner needs some way to detect that a space has failed, so that it can remove that space from all its objects. The details are somewhat subtle and beyond the scope of this discussion.

[1] There's a kludge that maps the local typecode to the surrogate typecode, instead of mapping the fingerprint to both.

## Practical communication

This section is about optimizing the space-to-space communication provided by `PerfectSR`. We'd like the efficiency to be reasonably close to what you could get by assembling messages by hand and delivering them directly to the underlying channel. Furthermore, we want to be able to use a variety of transports, since it's hard to predict what transports will be available or which ones will be most efficient. There are several scales at which we may want to work:

Bytes into or out of the channel.

Data blocks into or out of the channel.

Directly accessible channel buffers. Most channels will take bytes or blocks that you give them and buffer them up into suitable blocks (called packets) for transmission).

Transmitting and receiving buffers.

Setting up channels to spaces.

Passing references to spaces.

At the lowest level, we need efficient access to a transport's mechanism for transmitting bytes or messages. This often takes the form of a 'connection' that transfers a sequence of bytes or messages reliably and efficiently, but is expensive to keep around. A connection is usually tied to a particular address space and, unlike an address, cannot be passed around freely. So our grand strategy is to map `Space -> Connection` whenever we do a call, and then send the message over the connection. Because this mapping is done frequently, it must be efficient. In the most general case, however, when we have never talked to the space before, it's a lot of work to figure out what transports are available and set up a connection. Caching is therefore necessary.

The general mechanism we use is `Space -> SET Endpoint -> Location -> Connection`. The `Space` is globally unique, but has no other structure. It appears in every `Remote` and every surrogate object, so it must be optimized for space. An `Endpoint` is a transport-specific address; there is a set of them because a space may implement several transports. Because `Endpoint`'s are addresses, they are just bytes and can be passed freely in messages. A `Location` is an object; that is, it has methods that call the transport's code. Converting an `Endpoint` into a `Location` requires finding out whether the `Endpoint`'s transport is actually implemented here, and if it is, hooking up to the transport code. Finally, a `Location` object's `new` method yields a connection. The `Location` may cache idle connections or create new ones on demand, depending on the costs.

Consider the concrete example of TCP as the channel. An `Endpoint` is a DNS name or an IP address, a port number, and a UID for an address space that you can reach at that IP and port if it hasn't failed; this is just bits. The corresponding `Location` is an object whose `new` method generates a TCP connection to that space; it works either by giving you an existing TCP connection that it has cached, or by creating a new TCP connection to the space. A `Connection` is a TCP connection.

As we have seen, a `Space` is an abbreviation, translated by the `addrs` table. Thus `addrs: Space -> SET Endpoint`. We need to set up `addrs` for newly encountered `Space`'s,





and we do this by callback to the source of the `Space`, maintaining the invariant: have `remote ==> addrs!(remote.space)`. This ensures that we can always invoke a method of the remote, and that we can pass on the space's `Endpoint`'s when we pass on the remote. The callback returns the set of `Endpoint`'s that can be used to reach the space.

An `Endpoint` should ideally be an object with a `location` method, but since we have to transport them between spaces, this would lead to an undesirable recursion. Instead, an `Endpoint` is just a string (or some binary record value), and a transport can recognize its own endpoints. Thus instead of invoking `endpoint.location`, we invoke `tr.location` for each transport `tr` that is available, until one succeeds and returns a `Location`. If a `Transport` doesn't recognize the `Endpoint`, it returns `nil` instead. If there's no `Transport` that recognizes the `Endpoint`, then it's of no use.

A `Connection` is a bi-directional channel that has the `SR` built in and has `M = Byte`; it connects a caller and a server thread (actually the thread is assigned dynamically when a request arrives, as we saw in `NetObject`). Because there's only one sender and one receiver, it's possible to stuff the parts of a message into the channel one at a time, and the caller does not have to identify itself but can take anything that comes back as the response. Thus the connection replaces `NetObj.CId`. The idea is that a TCP connection could be used directly as a `Connection`. You can make a `Connection` from a `Location`. The reason for having both is that a `Location` is just a small data structure, while a `Connection` may be much more expensive to maintain. A caller acquires a `Connection` for each call, and releases it when the call is done. The code can choose between creating and destroying connections on the one hand, and caching them on the other, based on the cost of creating one versus the cost of maintaining an idle one.

The byte stream code should provide multi-byte `Put` and `Get` operations for efficiency. It may also provide access to the underlying buffers for the stream, which might make encoding and decoding more efficient; this must be done in a transport-independent way. Transmitting and receiving the buffers is handled by the transport. We have already discussed how to obtain a connection to a given space.

Actually, of course, the channels are usually not perfect but only reliable; that is, they can lose messages if there is a crash. And even if there isn't a crash, there might be an indefinite delay before a message gets through. If you have a transactional queuing system the channels might be perfect; in other words, if the sender doesn't fail it will be able to queue a message. However, the response might be long delayed, and in practice there has to be a timeout after which a call raises the exception `CallFailed`. At this point the caller doesn't know for sure whether the call completed or not, though it's likely that it didn't. In fact, it's possible that the call might still be running as an 'orphan'.

For maximum efficiency you may want to use a specialized transport rather than a general one like RPC. Handout 11 described one such transport and analyzes its efficiency in detail.

## Bootstrapping

So far we have explained how to invoke a method on a remote object, and how to pass references to remote objects from one space to another. To get started, however, we have to obtain some remote objects. If we have a single remote directory object that maps names to objects, we can

look up the names of lots of other objects there and obtain references to them. To get started, we can adopt the convention that each space has a special object with `OId 0` that is a directory. Given a `space`, we can forge `Remote{space, 0)` to get a reference to this object.

Actually we need not a `Space` but a `Location` that we can use to get a `Connection` for invoking a method. To get the `Location` we need an `Endpoint`, that is, a network address plus a well-known port number plus a standard unique identifier for the space. So given an address, say `www.microsoft.com`, we can construct a `Location` and invoke the lookup method of the standard directory object. If a server thread is listening on the well-known port at that address, this will work.

A directory object can act as a 'broker', choosing a suitable representative object for a given name. Several attempts have been made to invent general mechanisms for doing this, but usually they need to be application-specific. For example, you may want the closest printer to your workstation that has B-size paper. A generic broker won't handle this well.





.

# 25. Paper: Network Objects

The attached paper on network objects by Birrell, Nelson, Owicki, and Wobber is a fairly complete description of a working system. The main simplification is that it supports a single language, Modula 3, which is similar to Java. The paper explains most of the detail required to make the system reliable and efficient, and it gives the internal interfaces of the implementation.

February 28, 1994
Revised December 4, 1995

SRC Research Report

115

# Network Objects

Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber

digital

**Systems Research Center**
130 Lytton Avenue
Palo Alto, California 94301

**Affiliations**

Susan Owicki is an independent consultant. Her current address is 956 N. California Ave, Palo Alto, CA 94303, U.S.A. (e-mail: owicki@mojave.stanford.edu).

**Publication History**

An earlier version of this report appeared in the Proceedings of the Fourteenth ACM Symposium on Operating System Principles, Asheville, North Carolina, December 5-8, 1993.

This report is to appear in Software – Practice & Experience, John Wiley & Sons, Ltd.

## Authors' Abstract

A network object is an object whose methods can be invoked over a network. The Modula-3 network objects system is novel for its overall simplicity. It provides distributed type safety through the narrowest surrogate rule, which allows programmers to export new versions of distributed services as subtypes of previous versions. This report describes the design and implementation of the system, including a thorough description of realistic marshaling algorithms for network objects, precise informal specifications of the major system interfaces, lessons learned from using the system, and performance results.


## Contents

# 1 Overview

In pure object-oriented programming, clients cannot access the concrete state of an object directly, but only via the object's methods. This methodology applies beautifully to distributed computing, since the method calls are a convenient place to insert the communication required by the distributed system. Systems based on this observation began to appear about a decade ago, including Argus[16], Eden[1], and early work of Shapiro[24], and more keep arriving every day. It seems to be the destiny of distributed programming to become object-oriented, but the details of the transformation are hazy. Should objects be mobile or stationary? Should they be communicated by copying or by reference? Should they be active? Persistent? Replicated? Is the typical object a menu button or an X server? Is there any difference between inter-program typechecking and intra-program typechecking?

We believe that the way to make progress in these issues is to discuss them in the context of real implementations. To this end, this report describes a distributed programming system for Modula-3, that we call *network objects*.

Network objects provide functionality similar to remote procedure call (RPC), but they are more general and easier to use. Our network objects are not mobile, but we make it easy to communicate objects either by copying or by reference. Our objects are passive: they have no implicitly associated thread of control, nor is there any implicit synchronization associated with calling their methods. Our objects are not persistent or replicated. They are sufficiently lightweight that it would be perfectly practical to use one per menu button. We provide strong inter-program typechecking.

The primary distinguishing aspect of our system is its simplicity. We restricted our feature set to those features that we believe are valuable to all distributed applications (distributed type-checking, transparent invocation, powerful marshaling, efficient and convenient access to streams, and distributed garbage collection), and we omitted more complex or speculative features (object mobility, transactions).

We organized the implementation around a small number of quite simple interfaces, each of which is described in this report. The report also describes a number of implementation details that have been omitted from previously published work, including simple algorithms for marshaling and unmarshaling network objects in a heterogeneous network. All of this material makes the report longer than we would like, but to make progress in the complicated design space of distributed object systems it seems necessary to describe real systems in more detail than is customary in the research literature.

We now briefly introduce some of the central aspects of our design.

*Distributed typechecking.* Our system provides strong typechecking via the *narrowest surrogate rule*, which will be described in detail below. In a distributed environment it can be very difficult to release a new version of a service, since old clients must be supported as well as new ones. The narrowest surrogate rule allows a programmer to release a new version of the service as a subtype of the old version, which supports both old and new clients and ensures type safety. The narrowest surrogate rule could also be useful in other situations where separately compiled programs need to communicate in a type-safe way; for example, it could be used to solve the problem of version skew in shared libraries.

*Transparent remote invocation.* In our system, remote invocations are syntactically identical to local ones; their method signatures are identical. A client invoking a method of an object



need not know whether the object is local or remote.

*Powerful marshaling.* As in any distributed programming system, argument values and results are communicated by *marshaling* them into a sequence of bytes, transmitting the bytes from one program to the other, and then unmarshaling them into values in the receiving program. The marshaling code is contained in stub modules that are generated from the object type declaration by a stub generator. Our marshaling code relies heavily on a general-purpose mechanism called *pickles*. Pickles use the same runtime-type data structures used by the local garbage collector to perform efficient and compact marshaling of arbitrarily complicated data types. Our stub generator produces in-line code for simple types, but calls the pickle package for complicated types. This combination strategy makes simple calls fast, handles arbitrary data structures, and guarantees small stub modules.

We believe it is better to provide powerful marshaling than object mobility. The two facilities are similar, since both of them allow the programmer the option of communicating objects by reference or by copying. Either facility can be used to distribute data and computation as needed by applications. Object mobility offers slightly more flexibility, since the same object can be either sent by reference or moved; while with our system, network objects are always sent by reference and other objects are always sent by copying. However, this extra flexibility doesn't seem to us to be worth the substantial increase in complexity of mobile objects. For example, a system like Hermes[5], though designed for mobile objects, could be implemented straightforwardly with our mechanisms.

*Leveraging general-purpose streams.* Our whole design makes heavy use of object-oriented *buffered streams*. These are abstract types representing buffered streams in which the method for filling the buffer (in the case of input streams) or flushing the buffer (in the case of output streams) can be overridden differently in different subtypes. The representation of the buffer and the protocol for invoking the flushing and filling methods are common to all subtypes, so that generic facilities can deal with buffered streams efficiently, independently of where the bytes are coming from or going to. To our knowledge these streams were first invented by the designers of the OS6 operating system[26]. In Modula-3 they are called *readers* and *writers*[18].

Because we use readers and writers, our interface between stubs and protocol-specific communication code (which we call *transports*) is quite simple. This choice was initially controversial, and viewed as a likely source of performance problems. However, since readers and writers are *buffered* streams, it is still possible for the stubs to operate directly on the buffer when marshaling and unmarshaling simple types, so there is not much loss of efficiency for simple calls. And for arguments that need to be pickled, it is a further simplification that the streams used by the transport interface are of the same type as those assumed by the pickle package.

*Marshaling support for streams.* Inter-process byte streams are more convenient and efficient than RPC for transferring large amounts of unstructured data, as critics have often pointed out. We have therefore provided special marshaling support for Modula-3's standard stream types (readers and writers). We marshal readers and writers by defining surrogate readers and writers as subtypes of the abstract stream types. To communicate a stream from one program to another, a surrogate stream is created in the receiving program. Data is copied over the network between the buffers of the real stream and the surrogate stream.

Here again the prevalence of readers and writers is important: the stream types that the marshaling code supports are not some new kind of stream invented for marshaling purposes, but

exactly the readers and writers used by the existing public interfaces in the Modula-3 library.

*Distributed garbage collection.* Garbage collection is a valuable tool for programming distributed systems, for all the reasons that apply to programs that run in a single address space. Our distributed collector allows network objects to be marshaled freely between processes without fear of memory leakage. We employ a fault-tolerant and efficient algorithm for distributed garbage collection that is a generalization of reference counting; it maintains a set of identifiers for processes with references to an object. Distributed collection is driven by the local collectors in each process; there is no need for global synchronization. Our algorithm, however, does not collect circular structures that span more than one address space.

### 1.1 Related work

We have built closely on the ideas of Emerald[14] and SOS[25]; our main contribution has been to select and simplify the essential features of these systems. One important simplification is that our network objects are not mobile. Systems like Orca[2] and Amber[7] aim at using objects to obtain performance improvements on a multiprocessor. We hope that our design can be used in this way, but our main goal was to provide reliable distributed services, and consequently our system is quite different. For example, the implementations of Orca and Amber described in the literature require more homogeneity than we can assume. (Rustan Leino has implemented a version of Modula-3 network objects on the Caltech Mosaic, a fine-grained mesh multiprocessor[15]. But we will not describe his work here.)

Systems like Argus[16, 17] and Arjuna[8] are like network objects in that they aim to support the programming of reliable distributed services. However, they differ from network objects by providing larger building blocks, such as stable state and multi-machine atomic transactions, and are oriented to objects that are implemented by whole address spaces. Our network objects are more primitive and fine-grained.

The Spring *subcontract* is an intermediary between a distributed application and the underlying object runtime[11]. For example, switching the subcontract can control whether objects are replicated. A derivative of this idea has been incorporated into the *object adaptor* of the Common Object Request Broker Architecture[19]. We haven't aimed at such a flexible structure, although our highly modular structure allows playing some similar tricks, for example by building custom transports.

### 1.2 Definitions

A Modula-3 *object* is a reference to a data record paired with a method suite. The method suite is a record of procedures that accept the object itself as a first parameter. A new object type can be defined as a *subtype* of an existing type, in which case objects of the new type have all the methods of the old type, and possibly new ones as well (inheritance). The subtype can also provide new implementations for selected methods of the supertype (overriding). Modula-3 objects are always references, and multiple inheritance is not supported. A Modula-3 object includes a typecode that can be tested to determine its type dynamically[18].

A *network object* is an object whose methods can be invoked by other programs, in addition to the program that allocated the object. The program invoking the method is called the *client* and the program containing the network object is called the *owner*. The client and owner can be running on different machines or in different address spaces on the same machine.



A remote reference in a client program actually points to a local object whose methods perform remote procedure calls to the owner, where the corresponding method of the owner's object is invoked. The client program need not know whether the method invocation is local or remote. The local object is called a *surrogate* (also known as a proxy).

The surrogate object's type will be declared by a stub generator rather than written by hand. This type declaration includes the method overrides that are analogous to a conventional client stub module. There are three object types to keep in mind: the network object type `T` at which the stub generator is pointed; the surrogate type `TSrg` produced by the stub generator, which is a subtype of `T` with method overrides that perform RPC calls; and the type `TImpl` of the real object allocated in the owner, also a subtype of `T`. The type `T` is required to be a *pure* object type; that is, it declares methods only, no data fields. The type `TImpl` generally extends `T` with appropriate data fields.

If program `A` has a reference to a network object owned by program `B`, then `A` can pass the reference to a third program `C`, after which `C` can call the methods of the object, just as if it had obtained the reference directly from the owner `B`. This is called a *third party transfer* . In most conventional RPC systems, third party transfers are problematical; with network objects they work transparently, as we shall see.

For example, if a network node offers many services, instead of running all the servers it may run a daemon that accepts a request and starts the appropriate server. Some RPC systems have special semantics to support this arrangement, but third-party transfers are all that is needed: the daemon can return to the client an object owned by the server it has started; subsequent calls by the client will be executed in the server.

When a client first receives a reference to a given network object, either from the owner or from a third party, an appropriate surrogate is created by the unmarshaling code. Care is required on several counts.

First, different nodes in the network may use different underlying communications methods (transports). To create the surrogate, the code in the client must select a transport that is shared by the client and owner—and this selection must be made in the client before it has communicated with the owner.

Second, the type of the surrogate must be selected. That is, we must determine the type `TSrg` corresponding to the type `TImpl` of the real object in the owner. But there can be more than one possible surrogate type available in the client, since `TSrg` is not uniquely determined by `TImpl`. As we shall see, this situation arises quite commonly when new versions of network interfaces are released. The ambiguity is resolved by the *narrowest surrogate rule*: the surrogate will have the most specific type of all surrogate types that are consistent with the type of the object in the owner and for which stubs are available in the client and in the owner. This rule is unambiguous because Modula-3 has single inheritance only.

Since the type of the surrogate depends on what stubs have been registered in the owner as well as in the client, it can't be determined statically. A runtime type test will almost always be necessary after the surrogate is created.

### 1.3 Examples

The narrowest surrogate rule is useful when network interfaces change over time, as they always do. This section presents some examples to illustrate this utility. The examples also show

how network objects generalize and simplify features of conventional RPC. The examples are based on the following trivial interface to a file service:

```
INTERFACE FS;

IMPORT NetObj;

TYPE
  File = NetObj.T OBJECT METHODS
    getChar(): CHAR;
    eof(): BOOLEAN
  END;
  Server = NetObj.T OBJECT METHODS
    open(name: TEXT): File
  END;

END FS.
```

The interface above is written in Modula-3. It declares object types `FS.File`, a subtype of `NetObj.T` extended with two methods, and `FS.Server`, a subtype of `NetObj.T` with one extra method. Any data fields would go between `OBJECT` and `METHODS`, but these types are pure. It is conventional to name the principal type in an interface `T`; thus `NetObj.T` is the principal type in the `NetObj` interface.

In our design, all network objects are subtypes of the type `NetObj.T`. Thus the interface above defines two network object types, one for opening files, the other for reading them. If the stub generator is pointed at the interface `FS`, it produces a module containing client and server stubs for both types.

Here is a sketch of an implementation of the `FS` interface:

```
MODULE Server EXPORTS Main;

IMPORT NetObj, FS;

TYPE
  File = FS.File OBJECT
    <buffers, etc.>
  OVERRIDES
    getChar := GetChar;
    eof := Eof
  END;
  Svr = FS.Server OBJECT
    <directory cache, etc.>
  OVERRIDES
    open := Open
  END;

<code for GetChar, Eof, and Open>;

BEGIN
  NetObj.Export(NEW(Svr), "FS1");
  <pause indefinitely>
END Server.
```



The call `NetObj.Export(obj, nm)` *exports* the network object `obj`; that is, it places a reference to it in a table under the name `nm`, whence clients can retrieve it. The table is typically contained in an *agent* process running on the same machine as the server.

Here is a client, which assumes that the server is running on a machine named `server`:

```
MODULE Client EXPORTS Main;
  IMPORT NetObj, FS, IO;

VAR
  s: FS.Server := NetObj.Import("FS1", NetObj.Locate("server"));
  f := s.open("/usr/dict/words");

BEGIN
  WHILE NOT f.eof() DO IO.PutChar(f.getChar()); END
END Client.
```

The call `NetObj.Locate(nm)` returns a handle on the agent process running on the machine named `nm`. The call to `NetObj.Import` returns the network object stored in the agent's table under the name `FS1`; in our example this will be the `Svr` object exported by the server. `Import`, `Export`, and `Locate` are described further in the section below on bootstrapping.

The client program invokes the remote methods `s.open`, `f.getChar`, and `f.eof`. The network object `s` was exported by name, using the agent running on the machine `server`. But the object `f` is anonymous; that is, it is not present in any agent table. The vast majority of network objects are anonymous; only those representing major services are named.

For comparison, here is the same functionality as it would be implemented with an RPC that is not object-oriented, such as DCE RPC[20]. The interface would define a file as an *opaque type*:

```
INTERFACE FS;

TYPE T;

PROC Open(n: TEXT): T;
PROC GetChar(f: T): CHAR;
PROC Eof(f: T): BOOL;

END FS.
```

A conventional RPC stub generator would transform this interface into a client stub, a server stub, and a modified client interface containing explicit binding handles:

```
INTERFACE FSClient;

IMPORT FS;

TYPE Binding;

PROCEDURE Import(hostName: TEXT): Binding;
PROCEDURE Open(b: Binding, n: TEXT): FS.T;
PROCEDURE GetChar(b: Binding, f: FS.T): CHAR;
PROCEDURE Eof(b: Binding, f: FS.T): BOOL;

END FSClient.
```

The server would implement the FS interface and the client would use the FSClient interface. In FSClient, the type Binding represents a handle on a server exporting the FS interface, and the type T represents a so-called *context handle* on an open file in one of these servers. Here is the same client computation coded using the conventional version:

```
MODULE Client;

IMPORT FSClient, IO;

VAR
  b := FSClient.Import("server");
  f := FSClient.Open(b, "/usr/dict/words");
BEGIN
  WHILE NOT FSClient.Eof(b, f) DO
    IO.PutChar(FSClient.GetChar(b, f))
  END
END Client.
```

Comparing the two versions, we see that the network object s plays the role of the binding b, and the network object f plays the role of the context handle f. Network objects subsume the two notions of binding and context handle.

In the conventional version, the signatures of the procedures in FSClient differ from those in FS, because the binding must be passed. Thus the signature is different for local and remote calls. (In this example, DCE RPC could infer the binding from the context handle, allowing the signatures to be preserved; but the DCE programmer must be aware of both notions.) Moreover, although conventional systems tend to allow bindings to be communicated freely, they don't do the same for context handles: It is an error (which the system must detect) to pass a context handle to any server but the one that created it.

The conventional version becomes even more awkward when the same address space is both a client and a server of the same interface. In our FS example, for example, a server address space must instantiate the opaque type FS.T to a concrete type containing the buffers and other data representing an open file. On the other hand, a client address space must instantiate the opaque type FS.T to a concrete type representing a context handle. (This type is declared in the client stub module.) These conflicting requirements make it difficult for a single address space to be both a client and a server of the same interface. This problem is called *type clash*. It can be finessed by compromising on type safety; but the network object solution avoids the problem neatly and safely.

Object subtyping together with the narrowest surrogate rule make it easy to ship a new version of the server that supports both old and new clients, at least in the common case in which the only changes are to add additional methods. For example, suppose that we want to ship a new file server in which the files have a new method called close. First, we define the new type as an extension of the old type:

```
TYPE
  NewFS.File = FS.File OBJECT METHODS
    close()
  END;
```

Since an object of type NewFS.File includes all the methods of an FS.File, the stub for a NewFS.File is also a stub for an FS.File. When a new client—that is, a client linked



with stubs for the new type—opens a file, it will get a surrogate of type NewFS.File, and be able to invoke its close method. When an old client opens a file, it will get a surrogate of type FS.File, and will be able to invoke only its getChar and eof methods. A new client dealing with an old server must do a runtime type test to check the type of its surrogate.

The extreme case of the narrowest surrogate rule occurs when a network object is imported into a program that has no stubs linked into it at all. In this case the surrogate will have type NetObj.T, since every program automatically gets (empty) stubs for this type. You might think that a surrogate of type NetObj.T is useless, since it has no methods. But the surrogate can be passed on to another program, where its type can become more specific. For example, the agent process that implements NetObj.Import and NetObj.Export is a trivial one-page program containing a table of objects of type NetObj.T. The agent needs no information about the actual subtypes of these objects, since it doesn't call their methods, it only passes them to third parties.

## 1.4 Failure semantics and alerts

An ordinary procedure call has no special provision for notifying the caller that the callee has crashed, since the caller and the callee are the same program. But a remote procedure call mechanism must define some *failure semantics* that cover this situation, in order to make it possible to program reliable applications.

In theory, distributed computations can be more reliable than centralized ones, since if a machine crashes, the program can shift the computation to use other machines that are still working. But it isn't easy to put this theory into practice. Many distributed systems end up being *less* reliable than their centralized equivalents, because they are vulnerable to the failure of many machines instead of just one. Leslie Lamport, prominent both as a theorist and as a suffering user, has facetiously defined a distributed system as one in which "the failure of a computer you didn't even know existed can render your own computer unusable".

Many methodologies and tools have been proposed to aid in programming replicated distributed services that survive the failures of individual replicas. Our network object system is intended to provide a more fundamental communications primitive: replicated services can be built out of network objects, but so can non-replicated services.

The failure semantics of network objects are similar to those of many conventional RPC systems. The runtime raises the exception NetObj.Error in the client if the owner crashes while the method call is in progress. Therefore, in serious applications, all methods of network objects should include this exception in their RAISES set. (Failure to include the exception would cause the client to crash in this situation, which is usually not what you want a serious application to do.)

Unfortunately, there is no absolutely reliable way that one machine can tell if another has crashed, since the communication network can fail, and a live machine can't distinguish itself from a dead machine if it cannot communicate. Therefore, the exception NetObj.Error doesn't guarantee that the owner has crashed: possibly communication has failed. In the latter case, the method call in the owner may continue to execute, even while the client runtime raises NetObj.Error. The abandoned computation in the owner is called an *orphan*. To build an application that is robust in the presence of communication failures, the programmer must ensure that the computation meets its specification even in the presence of orphaned computations.

Modula-3 provides a mechanism for *alerting* a thread. This is not an interrupt, but a polite request for the thread to stop at the next convenient point and raise a pre-defined exception. When programming a lengthy computation that might for any reason be subject to cancellation, it is good style to check periodically to see if the thread has been alerted.

If a thread engaged in a remote call is alerted, the runtime raises `NetObj.Error` in the calling thread and simultaneously attempts to notify and alert the server thread executing the call. The reason that `NetObj.Error` is raised is that there is no guarantee that the attempt to alert the server thread will succeed; therefore, an orphan may have been created.

The network object system also uses alerts to handle the situation in which a client crashes while it has an outstanding remote method call. In this case, the network object runtime alerts the thread that is executing the method call in the owner. Therefore, most methods of network objects should include `Thread.Alerted` in their `RAISES` sets.

# 2 Implementation

This subsection describes the structure of our implementation. Much of the lower levels of our system are similar to that of conventional RPC, as described by Birrell and Nelson[4]. We will concentrate on the implementation aspects that are new in network objects.

## 2.1 Assumptions

We implemented our system with Modula-3 and Unix, but our design would work on any system that provides threads, garbage collection, and object types with single inheritance. At the next level of detail, we need the following capabilities of the underlying system:

1. Object types with single inheritance.
2. Threads (lightweight processes).
3. Some form of reliable, inter-address-space communication.
4. Garbage collection, together with a hook for registering a cleanup routine for selected objects to be called when they are collected (or explicitly freed).
5. Object-oriented buffered streams (readers and writers).
6. Runtime type support as follows. Given an object, to determine its type; given a type: to determine its supertype, to allocate and object of the type, and to enumerate the sizes and types of the fields of the type.
7. A method of communicating object types from one address space to another.

We will elaborate on the last item.

The Modula-3 compiler and linker generate numerical typecodes that are unique within a given address space. But they are not unique across address spaces and therefore cannot be used to communicate types between address spaces. Therefore, the Modula-3 compiler computes a *fingerprint* for every object type appearing in the program being compiled. A fingerprint is a sixty-four bit checksum with the property that (with overwhelming probability) two



types have the same fingerprint only if they are structurally identical. Thus a fingerprint denotes a type independently of any address space. Every address space contains two tables mapping between its typecodes and the equivalent fingerprint. To communicate a typecode from one address space to another, the typecode is converted into the corresponding fingerprint in the sending address space and the fingerprint is converted into the corresponding typecode in the receiving address space. If the receiving program does not contain a code for the type being sent, then the second table lookup will fail.

## 2.2 Pickles

We use a mechanism known as *pickles* to handle the more complex cases of marshaling, specifically those that involve references types as arguments or results. Our pickles package is similar to the value transmission mechanism described by Herlihy and Liskov[12], who seem to be the first to have described the problem in detail. However, our package is more general because it handles subtyping and dynamic types.

For simple usage, our pickle package provides a simple interface. `Pickle.Write(ref, wr)` writes a flattened representation of the dynamically typed value `ref` to the writer `wr`. `Pickle.Read(rd)` reads the representation of a value from the reader `rd` and returns a dynamically typed reference to a copy of the original value.

The pickle package relies on the compile-time and runtime type support described earlier, and in particular on the existence of type fingerprints. Given this support, the basic method for writing a pickle is quite simple. It writes the fingerprint of the given value's type on the byte stream, followed by the referent's data fields. The method recurses on any constituent values that are themselves references types. Reading is the inverse operation: read a fingerprint, allocate a value, examine the type, read data fields and recursively read reference fields.

One minor complication is the problem of keeping the values independent of machine architecture (for example, byte order or word length). We do this by encoding the salient properties of the architecture in a small number of bytes at the start of the pickle, then writing the pickle in the sender's native form. This approach is efficient in homogeneous cases, and no more costly than anything else in heterogeneous cases. We assume that all architectures can be described by our header. If there were an aberrant architecture, its pickle package would be required to map to and from a standard one on sending and receiving.

A slightly more significant complication is detecting and dealing with multiple occurrences of the same reference within a single pickled value. This happens in cyclic structures and also in graph-like structures that are not trees. (We make no attempt to preserve sharing between separate pickles.)

When writing a pickle, the sender maintains a hash table keyed by references. The values in this table are small integers, allocated sequentially within each particular pickle. When a reference is first encountered in writing a pickle, it is entered in the table and allocated a small integer. This integer is written on the byte stream after the reference's fingerprint, as the defining occurrence. Then the pickle package writes the referent by recursing. If a reference is encountered for a second or subsequent time in a single pickle, the reference's small integer is found in the hash table and written on the byte stream as a subsequent occurrence; in this case there is no need to examine or write the referent.

When reading a pickle, the receiver maintains an array indexed by these small integers. When it encounters the first occurrence of a reference's small integer, it allocates the storage

and records the new reference in the appropriate entry of the array, and proceeds to read the referent from the byte stream. When it encounters a subsequent occurrence of the reference's small integer, it just uses the reference obtained by indexing the table.

The default behavior of the pickle package isn't satisfactory for all types. Problems can arise if the concrete representation of an abstract type isn't an appropriate way to communicate the value between address spaces. To deal with this, the pickle package permits clients to specify custom procedures for pickling (and therefore for marshaling) particular data types. Typically the implementer of an abstract data type specifies such a custom procedure if the type's values aren't transferable by straightforward copying. We use this facility to marshal network objects, readers, and writers that are embedded in structures that would ordinarily be marshaled by value.

The narrowest surrogate rule places a serious constraint on the pickles design: since pickling and unpickling an object can make its type less specific, the unpickler must check that an unpickled object is legal at the position in which it occurs. It is possible for the unpickler to check this because the runtime provides the ability to enumerate the types of the fields of an object.

There are many subtleties in the design of the pickles package. The ability to register custom pickling procedures for selected types has a tricky interaction with subtyping. It is also tricky to define and efficiently compute type fingerprints for recursive and opaque types. But the details are beyond the scope of the present work.

## 2.3 Garbage collection

Our system includes network-wide, reference-counting garbage collection. For each network object, the runtime records the set of clients containing surrogates for the object (the *dirty set*). As long as this set is non-empty, the runtime retains a pointer to the object. The retained pointer protects the object from the owner's garbage collector, even if no local references to it remain. When a surrogate is created, the client runtime adds its address space to the dirty set for the concrete object by making a "dirty call" to the owner. When a surrogate is reclaimed, the client runtime deletes its address space from the dirty set by making a "clean call". When the dirty set becomes empty, the owner's runtime discards the retained pointer, allowing the owner's local garbage collector to reclaim the object if no local references remain. To trigger the clean call, the client runtime relies on the assumed ability to register cleanup hooks for surrogates with the local collector.

This scheme will not garbage-collect cycles that span address spaces. To avoid this storage leak, programmers are responsible for explicitly breaking such cycles.

If program `A` sends program `B` a reference to an object owned by a third program `C`, and `A` then drops its reference to the object, we must ensure that the dirty call from `B` precedes the clean call from `A`, to avoid the danger that the object at `C` will be prematurely collected. This is not a problem if the object is sent as an argument to a remote method call, since in this case the calling thread retains a reference to the object on its stack while it blocks waiting for the return message, which cannot precede the unmarshaling of the argument. But if the object is sent as a result rather than an argument, the danger is real. Our solution is to require an acknowledgement to any result message that contains a network object: the procedure that executes the call in the owner blocks waiting for the acknowledgement, with the reference to the object on its stack. The stack reference protects the object from the garbage collector. This



solution increases the message count for method calls that return network objects, but it doesn't greatly increase the latency of such calls, since the thread waiting for the acknowledgement is not on the critical path.

By maintaining the set of clients containing surrogates rather than a simple count, we are able to remove clients from the dirty set when they exit or crash. The mechanism for detecting that clients have crashed is transport-specific, but for all reasonable transports there is some danger that a network partition that prevents communication between the owner and client will be misinterpreted as a client crash. In this case, the owner's object might be garbage collected prematurely. Because communication is unreliable, the risk of premature collection is inherent in any strategy that avoids storage leaks in long-running servers. Since we never reuse object IDs, we can detect premature collection if it occurs.

Dirty calls are synchronous with surrogate creation, but clean calls are performed in the background and can be batched. If a clean call fails, it will be attempted again. If a dirty call fails, the client schedules the surrogate to be cleaned (since the dirty call might have added the client to the dirty set before failing) and raises the exception `NetObj.Error`. Clean and dirty calls carry sequence numbers that increase monotonically with respect to any given client: the owner ignores any clean or dirty call that is out of sequence. This requires the owner to store a sequence number for each entry in the dirty set, as well as a sequence number for each client for which a call has failed. The sequence numbers for clients that have successfully removed themselves from the dirty set can be discarded.

A companion paper[3] presents the details of the collection algorithm and a proof of its correctness.

## 2.4 Transports

There are many protocols for communicating between address spaces (for example, shared memory, TCP, and UDP), and many irksome differences between them. We insulate the main part of the network object runtime from these differences via the abstract type `Transport.T`.

A `Transport.T` object generates and manages connections between address spaces. Different subtypes are implemented using different communication mechanisms. For example, a `TCPTransport.T` is a subtype that uses TCP.

Each subtype is required to provide a way of naming address spaces. A transport-specific name for an address space is called an *endpoint*. Endpoints are not expected to be human-sensible. Naming conventions ensure that an endpoint generated by one transport subtype will be meaningful only to other instances of the same subtype. (Some use the term "endpoint" in a weaker sense, meaning little more than a port number. For us, different instances of a program are identified by different endpoints.)

The `fromEndpoint` method of a `Transport.T` enables creation of connections to recognized endpoints. If `tr` is a `Transport.T` and `ep` is an endpoint recognized by `tr`, then `tr.fromEndpoint(ep)` returns a `Location` (described in the next paragraph) that generates connections to the address space named by `ep`. If `tr` doesn't recognize `ep`, then such an invocation returns `NIL`.

A `Location` is an object whose `new` method generates connections to a particular address space. When a client has finished using a connection, it should pass the connection to the `free` method of the location that generated it. This allows transports to manage the allocation and

deallocation of connections so as to amortize the overhead of connection establishment and maintenance.

The system uses the type `StubLib.Conn` to represent connections. It is perfectly possible to implement a class of connection that communicates with datagrams according to a protocol that makes idle connections essentially free[4]. That is, in spite of its name, implementations of the type `StubLib.Conn` need not be connection-oriented in the standard sense of the term.

A connection `c` contains a reader `c.rd` and a writer `c.wr`. Connections come in pairs; if `c` and `d` are paired, whatever is written to `c.wr` can be read from `d.rd`, and vice-versa. Ordinarily `c` and `d` will be in different address spaces. Values are marshaled into a connection's writer and unmarshaled from a connection's reader. Since readers and writers are buffered, the marshaling code can treat them either as streams of bytes (most convenient) or as streams of datagrams (most efficient).

One of the two connections in a pair is the *client* side and the other is the *server* side. Transports are required to provide a thread that listens to the server side of a connection and calls into the network object runtime when a message arrives indicating the beginning of a remote call. This is called *dispatching*, and is described further below.

A connection is required to provide a way of generating a "back connection": the location `c.loc` must generate connections to the address space at the other side of `c`. If `c` is a server-side connection, the connections generated by `c.loc` have the opposite direction as `c`; if `c` is a client-side connection, they have the same direction as `c`.

A transport is responsible for monitoring the liveness of address spaces for which it has locations or connections. This is discussed in more detail later when we specify the interface to the transport system.

## 2.5 Basic representations

We will now describe the wire representation of network objects, the client and server stubs involved in remote invocation, and the algorithms we use to marshal and unmarshal network objects. In these descriptions we will use Modula-like pseudocode.

The wire representation for a network object is a pair `(sp, i)` where `sp` is a `SpaceID` (a number that identifies the owner of the object) and `i` is an `ObjID` (a number that distinguishes different objects with the same owner):

```
TYPE WireRep = RECORD sp: SpaceID; i: ObjID END;
```

Each address space maintains an *object table* `objtbl` that contains all its surrogates and all its network objects for which any other space holds a surrogate:

```
VAR objtbl: WireRep -> NetObj.T;
```

We use the notation `A -> B` to name the type of a table with domain type `A` and element type `B`. We will use array notation for accessing the table, even though it is implemented as a hash table. We write `domain(tbl)` to denote the set of elements of the domain of `tbl`.

We now specify the representation of the opaque type `NetObj.T`. In Modula-3, the `REVEAL` statement permits such a specification to be visible within a bounded scope.

```
REVEAL
```



```
  NetObj.T = OBJECT
    srgt, inTbl: BOOLEAN;
    wrep: WireRep;
    loc: Location;
    disp: Dispatcher
  END;

TYPE Dispatcher = PROC(c: StubLib.Conn; obj: NetObj.T);
```

The field `obj.srgt` indicates whether `obj` is a surrogate. The field `obj.inTbl` indicates whether `obj` is present in `objTbl`, and this is guaranteed to be the case if `obj` is a surrogate or if another address space holds a surrogate for it. If `obj.inTbl` is `TRUE`, then `obj.wrep` is the wire representation of the object.

If `obj` is a surrogate is then `obj.loc` is a `Location` that generates connections to the owner's address space at `obj.wrep.sp`, and `obj.disp` is unused. Otherwise, if `obj.inTbl` is `TRUE`, then `obj.disp` is the dispatcher procedure for the object, and `obj.loc` is unused. The call `obj.disp(c, obj)` unmarshals a method number and arguments from `c`, calls the appropriate method of `obj`, and marshals the result to `c`.

## 2.6 Remote invocation

To illustrate the steps in a remote method invocation we continue with our example of a simple file service. In that example, we defined the type `FS.Server` with a single method `open`. The corresponding stub-generated surrogate type declaration looks like this:

```
TYPE
  SrgSvr = FS.Server OBJECT
    OVERRIDES
      open := SrgOpen
    END;

PROCEDURE SrgOpen(obj: SrgSvr; n: TEXT): FS.File =
  VAR
    c := obj.loc.new();
    res: FS.File;
  BEGIN
    OutNetObj(c, obj);
    OutInteger(c, 0);
    OutText(c, n);
    <flush buffers to network>;
    res := InNetObj(c);
    obj.loc.free(c);
    RETURN res
  END SrgOpen;
```

The procedures `OutNetObj` and `InNetObj` are described in the next subsection. Procedures for marshaling basic types (like `OutInteger`) are in the `StubLib` interface which we specify later in the report. (Actually, `StubLib.OutRef` subsumes both `OutNetObj` and `OutText`, but we ignore that here.)

The method being invoked is identified on the wire by its index; the `open` method has index zero. The code presented would crash with a narrow fault if the network object returned by

InNetObj were not of type FS.File. For example, this would happen if appropriate stubs had not been linked into the client or owner. The actual system would raise an exception instead of crashing.

On the server side, the thread forked by the transport to service a connection c calls into the network object runtime when it detects an incoming RPC call. The procedure it calls executes code something like this:

```
VAR obj := InNetObj(c); BEGIN obj.disp(c, obj) END;
```

The dispatcher procedures are typically written by the stub generator. The dispatcher for the type FS.Server would look something like this:

```
PROCEDURE SvrDisp(c: StubLib.Conn; obj: FS.Server) =
  VAR methID := InInteger(c); BEGIN
    IF methID = 0 THEN
      VAR
        n := InText(c);
        res := obj.open(n);
      BEGIN
        OutNetObj(c, res);
        <flush buffers to network>
      END
    ELSE
      <error, non-existent method>
    END
  END SvrDisp;
```

The stubs have a narrow interface to the rest of the system: they call the new and free methods of Location objects to obtain and release connections, and they register their surrogate types and dispatcher procedures where the runtime can find them, in the global table stubs:

```
VAR stubs: Typecode -> StubRec;

TYPE StubRec = RECORD srgType: TypeCode; disp: Dispatcher END;
```

An address space has stubs for tc if and only if tc is in the domain of stubs. If tc is in the domain of stubs, then stubs[tc].srgType is the typecode for the surrogate type for tc, and stubs[tc].disp is the owner dispatcher procedure for handling calls to objects of type tc.

A stub module that declares a surrogate type srgTC and dispatcher disp for a network object type tc also sets stubs[tc] := (srgTC, disp). The network object runtime automatically registers a surrogate type and null dispatcher for the type NetObj.T.

In the actual system the stubs table is indexed by stub protocol version as well as type code, to make it easy for a program to support multiple protocol versions. The actual system also includes code for relaying exceptions raised in the owner to the client, and for relaying thread alerts from the client to the owner.

## 2.7 Marshaling network objects

The call OutNetObj(c, obj) writes the wire representation of obj to the connection c:



```
PROCEDURE OutNetObj(c: StubLib.Conn; obj: NetObj.T) =
  BEGIN
    IF obj = NIL THEN
      OutWireRep(c, (-1,-1));
    ELSE
      IF NOT obj.inTbl THEN
        VAR i := NewObjID(); BEGIN
          obj.wrep := (SelfID(), i);
          objtbl[obj.wrep] := obj;
          obj.inTbl := TRUE;
          obj.srgt := FALSE;
          obj.disp := GetDisp(TYPECODE(obj))
        END
      END;
      OutWireRep(c, obj.wrep)
    END
  END OutNetObj;

PROCEDURE GetDisp(tc: INTEGER): Dispatcher =
  BEGIN
    WHILE NOT tc IN domain(stubs) DO tc := Supertype(tc) END;
    RETURN stubs[tc].disp
  END GetDisp;
```

In the above we assume that NewObjID() returns an unused object ID, that SelfID() returns the SpaceID of the caller, and that Supertype(tc) returns the code for the supertype of the type whose code is tc.

The corresponding call InNetObj(c) reads a wire representation from the connection c and returns the corresponding network object reference:

```
PROCEDURE InNetObj(c: StubLib.Conn): NetObj.T =
  VAR wrep := InWireRep(c); BEGIN
    IF wrep.i = -1 THEN
      RETURN NIL
    ELSIF wrep IN domain(objtbl) THEN
      RETURN objtbl[wrep]
    ELSE
      RETURN NewSrgt(wrep, c)
    END
  END InNetObj;
```

The call NewSrgt(wrep, c) creates a surrogate for the network object whose wire representation is wrep, assuming that c is a connection to an address space that knows wrep.sp. (We say that an address space sp1 *knows* an address space sp2 if sp1=sp2 or if sp1 contains some surrogate owned by sp2.)

NewSrgt locates the owner, determines the typecode of the surrogate, and enters it in the object table:

```
PROCEDURE NewSrgt(wrep: WireRep; c: StubLib.Conn): NetObj.T =
  VAR
    loc := FindSpace(wrep.sp, conn);
    tc := ChooseTC(loc, wrep.i);
```

```
    res := Allocate(tc);
  BEGIN
    res.wrep := wrep;
    res.srgt := TRUE;
    res.inTbl := TRUE;
    objtbl[wrep] := res;
    RETURN res
  END NewSrgt;
```

The call `FindSpace(sp, c)` returns a `Location` that generates connections to `sp`, or raises `NetObj.Error` if this is impossible. It requires that `c` be a connection to an address space that knows about `sp`. The call `ChooseTC(loc, i)` implements the narrowest surrogate rule. It returns the local code for the local surrogate type for the object whose ID is `i` and whose owner is the address space to which `loc` generates connections. The call `Allocate(tc)` allocates an object with type code `tc`.

To implement `FindSpace` without resorting to broadcast, each address space maintains information about its own transports and the endpoints of the address spaces it knows about. This information is maintained in the variables `tr` and `names`:

```
VAR tr: SEQ[Transport.T];
VAR names: SpaceID -> SEQ[Endpoint];
```

The sequence `tr` lists the transports available in this space, in decreasing order of desirability. Typically, it is initialized by the network object runtime and is constant thereafter. For any space `sp`, the sequence `names[sp]` contains the endpoints for `sp` recognized by `sp`'s transports. We write `SEQ[T]` to denote the type of sequences of elements of type `T`.

The fast path through `FindSpace` finds an entry for `sp` in `names`; this entry is the list of names for `sp` recognized by `sp`'s transports. These names are presented to the transports `tr` available in this space; if one is recognized, a common transport has been found; if none is recognized, there is no common transport.

The first time an address space receives a reference to an object owned by `sp`, there will be no entry for `sp` in the space's name table. In this case, `FindSpace` obtains the name sequence for `sp` by making an RPC call to the address space from which it received the reference into `sp`. This is our first example of an RPC call that is nested inside an unmarshaling routine; we will use the notation `RPC(loc, P(args))` to indicate an RPC call to `P(args)` directed at the address space identified by the location `loc`. Here is the implementation of `FindSpace`:

```
PROCEDURE FindSpace(sp: SpaceID; c: StubLib.Conn): Location =
  BEGIN
    IF NOT sp IN domain(names) THEN
      names[sp] := RPC(c.loc, GetNames(sp));
    END;
    VAR nm := names[sp]; BEGIN
      FOR i := 0 TO LAST(tr) DO
        FOR j := 0 TO LAST(nm) DO
          VAR loc := tr[i].fromEndpoint(nm[j]); BEGIN
            IF loc # NIL THEN RETURN loc END
          END
        END
      END;
```



```
      RAISE NetObj.Error
    END
  END FindSpace;

PROCEDURE GetNames(sp) = BEGIN RETURN names[sp] END GetNames;
```

Placing the `i` loop outside the `j` loop gives priority to the client's transport preference over the owner's transport preference. The choice is arbitrary: usually the only point of transport preference is to obtain a shared memory transport if one is available, and this will happen whichever loop is outside.

The only remaining procedure is `ChooseTC`, which must implement the narrowest surrogate rule. According to this rule, the surrogate type depends on which stubs have been registered in the client and in the owner: it must determine the narrowest supertype for which both client and owner have a registered stub. This requires a call to the owner at surrogate creation time, which we combine with the call required by the garbage collector: the call `Dirty(i, sp)` adds `sp` to the dirty set for object number `i` and returns the supertypes of the object's type for which stubs are registered in the owner.

```
PROCEDURE Dirty(i: ObjID; sp: SpaceID): SEQ[Fingerprint] =
  VAR
    tc := TYPE(objtbl[(SelfID(), i)]);
    res: SEQ[Fingerprint] := <empty sequence>;
  BEGIN
    <add sp to object i's dirty set>;
    WHILE NOT tc IN domain(stubs) DO tc := Supertype(tc) END
    LOOP
      res.addhi(TCToFP(tc));
      IF tc = TYPECODE(NetObj.T) THEN EXIT END;
      tc := Supertype(tc)
    END;
    RETURN res
  END Dirty;

PROCEDURE ChooseTC(loc: Location; i: ObjID): INTEGER =
  VAR fp: SEQ[Fingerprint]; BEGIN
    fp := RPC(c.loc, Dirty(i, SelfID()));
    FOR j := 0 TO LAST(fp) DO
      IF FPToTC(fp[j]) IN domain(stubs) THEN
        RETURN stubs(FPToTC(fp[j])).srgType
      END
    END
  END ChooseTC;
```

The loops in `Dirty` are guaranteed to terminate, because stubs are automatically registered for `NetObj.T`. In `ChooseTC` we assume that `TCToFP` and `FPToTC` convert between equivalent typecodes and fingerprints (if there is no local typecode for `fp`, we assume that `FPToTC(fp)` returns some illegal typecode never present in `stubs`). We also assume that `s.addhi(x)` extends the sequence `s` with the new element `x`.

This concludes our description of the algorithms for marshaling network objects. We have omitted a number of details. For example, to avoid cluttering up the program, we have ignored synchronization; the real program must protect the various global tables with locks. Some

care is required to avoid deadlock; for example, it is not attractive to hold a lock all the way through a call to `NewSrgt`. Instead, we make an entry in the surrogate table at the beginning of the procedure, recording that a surrogate is under construction, and do not reacquire the table lock until the end of the sequence, when the surrogate has been fully constructed. A thread that encounters a surrogate under construction simply waits for it to be constructed.

### 2.8 Marshaling streams

An important feature of our treatment of streams is that data is not communicated via RPC, but by the underlying transport-specific communication. This facility is analogous to the remote pipes of DCE RPC, but with a critical difference: the streams we pass are not limited in scope to the duration of the RPC call. When we marshal a stream from process A to process B, process B acquires a surrogate stream attached to the same data as the original stream. In process B the surrogate stream can be used at will, long after the call that passed it is finished. In contrast, in a scheme such as the pipes provided in DCE, the data in the pipe must be communicated in its entirety at the time of the RPC call (and at a particular point in the call too). Our facility is also analogous to the remote pipes of Gifford and Glasser[10], but is simpler and more transparent.

Readers and writers are marshaled very similarly; to be definite, consider a reader `rd`. The sending process has a concrete reader `rd` in hand. The marshaling code must create a surrogate reader `rdsrg` in the receiving process, such that `rdsrg` delivers the contents of `rd`. The general strategy is to allocate a connection between the sender and receiver, allocate `rdsrg` in the receiver so that it reads from the connection, and fork a thread in the sender that reads buffers from `rd` and sends them over the connection. (The thread could be avoided by doing an RPC to fill the buffer of `rdsrg` whenever it is empty, but this would increase the per-buffer overhead of the cross-address space stream.) For the detailed strategy we explored two designs.

In the first design, the sender uses the connection `c`, over which `rd` is to be marshaled, to create a new connection `nc := c.loc.new()`. The sender then chooses a unique ID, sends the ID over `nc`, sends the ID over `c` as the wire representation of `rd`, and forks a thread that copies data from `rd` into `nc`. In the receiving process, two threads are involved. The thread servicing the connection `nc` reads the ID (distinguishing it from an incoming call message) and places the connection in a table with the ID as key. The thread unmarshaling the reader looks up the connection in the table and allocates the surrogate reader `rdsrg` using that connection. This seems simple, but the details became rather complicated, for example because of the difficulty of freeing connections in the table when calls fail at inopportune times.

The second design employs a network object called a `Voucher` with a method `claim` that returns a reader. Vouchers have nonstandard surrogates and dispatchers registered for them, but are otherwise ordinary network objects.

To marshal `rd`, the sending process allocates a voucher `v` with a data field `v.rd` of type reader, sets `v.rd := rd`, and calls `OutNetObj(v)`. When the receiving process unmarshals a network object and finds it is a surrogate reader voucher `vs`, it calls `vs.claim()` and returns the resulting reader.

The `claim` method of a surrogate voucher `vs` invokes `vs.loc.new()` to obtain a new connection `nc`. It then marshals `vs` to `nc` (just like an ordinary surrogate method call). But then,



instead of sending arguments and waiting for a result, it allocates and returns the surrogate reader `rdsrg`, giving it the connection `nc` as a source of data.

On the server side, the voucher dispatcher is called by a transport-supplied thread, just as for an ordinary incoming call. The arguments to the dispatcher are the server side of the connection `nc` and the voucher `vs` containing the original reader `vs.rd`. The dispatcher procedure plays the role of the forked thread in the first design: it reads buffers from `vs.rd` and sends them over `nc`.

The second design relies on the transport to provide the required connection and thread, and relies on the ordinary network object marshaling machinery to connect the surrogate voucher with the original reader. This makes the protocol simple, but it costs three messages (a round trip for the dirty call for the voucher; then another message to launch the voucher dispatcher). It would be easy enough to avoid the all-but-useless dirty call by dedicating a bit in the wire representation to identify vouchers, but perhaps not so easy to stomach the change. On the other hand, the first design uses only one message (to communicate the `ID` from the sender to the receiver), and this message could perhaps be piggybacked with the first buffer of data.

We chose the second design, because: (1) it is trivial to implement; (2) given that cross-address space streams are intended for bulk data transfer, it is not clear how important the extra messages are; and (3) if experience leads us to get rid of the extra messages, it is not obvious whether to choose the first design or to optimize the second.

### 2.9 Bootstrapping

The mechanisms described so far produce surrogate network objects only as a result of method calls on other surrogate network objects. We have as yet no way to forge an original surrogate. To do this we need the ingredients of a surrogate object: a `Location`, an object ID, and a surrogate type. To make it possible to forge the object ID and type, we adopt the following convention: every program into which network objects are linked owns a *special object* with an ID of zero, of a known type. The methods of the special object implement the operations required by the network object runtime (reporting in clean and dirty, `GetNames`, etc.). The special object also has `get` and `put` methods implementing a table of named network objects. At initialization time the network object runtime allocates a special object with ID 0 and places it in `objtbl`.

All that remains to forge an original surrogate is to obtain a `Location` valid for some other program into which network objects have been linked. Fundamentally, the only way to obtain a `Location` is to call some transport's `fromEndpoint` method—that is, the program forging the surrogate must know an address where something is listening. For this step the application has two choices. We provide a network object agent that listens at a well-known TCP port; thus a surrogate for the agent's special object can be forged given the IP name of the node on which it is running. If every node runs the agent from its start-up script, then no other well-known ports are needed: applications can export their objects by putting them in the table managed by the agent's special object, and their clients can get them from the same table. If the application writer prefers not to rely on the agent, he can choose his own transport and well-known port and configure his program to listen at that port and to forge surrogates for the special objects at that port.

The procedures `NetObj.Import` and `NetObj.Export`, which appeared in our file server example, implement object bootstrapping. These procedures simply forge a surrogate for the

special object of some agent process, and then invoke the `get` or `put` method on that surrogate.

# 3 Public Interfaces

In this section and the next, we present the major system interfaces exactly as they appear in the network objects programmers library. This section describes the `NetObj`, `NetStream`, and `NetObjNotifier` interfaces, as well as the stub generator. These interfaces are sufficient for most network objects clients. The following section presents important internal interfaces that are not used by most clients.

The interfaces in this section depend on a few local-level facilities from the SRC Modula-3 runtime library[18, 13]. We summarize these dependencies here:

- An `Atom.T` is a unique representation for a set of equal texts (like a Lisp atomic symbol). Atoms are often used to parameterize exceptions.
- An `AtomList.T` is a linked list of atoms. Atom lists are used for propagating lists of nested exception parameters up the call stack.
- A `Rd.T` represents an abstract data source. The `Rd` interface provides operations for reading data and re-positioning the stream.
- A `Wr.T` represents an abstract data sink. The `Wr` interface provides operations for writing data and re-positioning the stream.
- `Thread.Alerted` is the exception to be raised by an alerted thread.
- The `WeakRef` interface allows clients to register garbage collection finalization procedures for objects in the traced heap. In other words, a client can obtain notification just prior to collection of heap storage.

## 3.1 NetObj interface

This is the primary public interface for using network objects. Before listing the interface, here are a few definitions.

A *program instance* is an activation of a program. The same program can have many instances running concurrently or consecutively. A program instance can be thought of as an address space, although the design does not preclude the implementation of a program instance by a suite of address spaces.

Recall that an agent is a program that provides a table that maps names to network objects. Any program can be an agent, but every machine has a particular default agent. Owners typically make network objects available to clients by inserting them into an agent's table, using the procedure `NetObj.Export`. Clients typically use `NetObj.Import` to retrieve network objects from the table.

```
INTERFACE NetObj;
IMPORT Atom, AtomList, Thread;
```



```
TYPE
  T <: ROOT;
  Address <: REFANY;
```

`NetObj.T` is the root type of all network objects. A `NetObj.Address` designates a program instance.

```
PROCEDURE Locate (host: TEXT): Address
    RAISES {Invalid, Error, Thread.Alerted};
```

*Return an address for the default agent at the machine whose human-sensible name is `host`.*

The naming convention used by `Locate` is system-dependent. For example, in an Internet environment, `Locate("decsrc.pa.dec.com")` returns the address of the default agent on the machine `decsrc` in the DEC Palo Alto Internet domain.

`Locate` raises `Invalid` if it determines that `host` is not a valid name. It raises `Error` if it is unable to interpret the name or determine its validity, typically because it is unable to contact the naming authority, or if there is no standard agent running on the specified host.

```
PROCEDURE Export(name: TEXT; obj: T; where: Address := NIL)
    RAISES {Error, Thread.Alerted};
```

*Set `table[name] := obj` where `table` is the table provided by the agent whose address is `where`, or by the default agent for the local machine if `where=NIL`. This can be used with `obj=NIL` to remove an entry from the table.*

```
PROCEDURE Import(name: TEXT; where: Address := NIL): T
    RAISES {Error, Thread.Alerted};
```

*Return `table[name]` where `table` is the table provided by the agent whose address is `where`, or by the default agent for the local machine if `where=NIL`. `Import` returns `NIL` if `table` contains no entry for `name`.*

```
EXCEPTION Error(AtomList.T), Invalid;

VAR (*CONST*)
  CommFailure, MissingObject, NoResources,
  NoTransport, UnsupportedDataRep, Alerted: Atom.T;

END NetObj.
```

The exception `NetObj.Error` indicates that a failure occured during a remote method invocation. Every remote method should therefore include this exception in its raises clause. If `NetObj.Error` is not raised, then the invocation completed successfully. If it is raised, it may or may not have completed successfully. It is possible that an orphaned remote invocation continued to execute at the owner, while the client raised `NetObj.Error`.

The first atom in the argument to `NetObj.Error` explains the reason for the failure; any subsequents atoms provide implementation-specific detail. The atom `CommFailure` indicates communication failure, which might be network failure or a crash on a remote machine. The atom `MissingObject` indicates that some network object, either the one whose method is

invoked or an argument to that method, has been garbage-collected by its owner. (This indicates that the owner mistakenly determined that one of its clients was dead.) `NoResources` indicates that the call failed because of a lack of resources, for example Unix file descriptors. `NoTransport` indicates that an attempt to unmarshal an object failed because the client and owner shared no common transport protocol implementation and were therefore unable to communicate. `UnsupportedDataRep` indicates a mismatch between the network representation of data and the ability of a receiver to handle it, for example a 64-bit `INTEGER` with non-zero high-order bits is not meaningful as an `INTEGER` on a 32-bit machine. `Alerted` indicates that a client thread was alerted in the middle of a remote call and that an orphaned remote computation might still be in progress. (Threads alerted in remote calls might also raise `Thread.Alerted`; in which case it is guaranteed that no orphans remain.) If the first atom in the argument list does not appear in this interface, a network object runtime error is indicated.

### 3.2 NetStream interface

The `NetStream` interface describes the marshaling of readers and writers, and provides procedures that you will need to use if you plan to reuse a stream after marshaling it.

The network object runtime allows subtypes of `Rd.T` and `Wr.T` to be marshaled as parameters and as results of remote method invocation. To communicate a reader or writer from one program to another, a surrogate stream is created in the receiving program. We call the original reader or writer the concrete stream. Data is copied over the network between the concrete stream and the surrogate stream. Surrogate streams are free-standing entities, valid beyond the scope of the remote call that produced them. Data can be transmitted on a surrogate stream at close to the bandwidth supported by the underlying transport.

The initial position of the surrogate reader or writer equals the position of the corresponding concrete stream at the time it was marshaled. All surrogate readers and writers are unseekable. Data is transferred between surrogates and concrete streams in background. Therefore, undefined behaviour will result if you 1) perform local operations on the concrete stream while a surrogate for it exists, or 2) create two surrogates for the same stream by marshaling it twice. There is a mechanism, described below, for shutting down a surrogate stream so that the underlying stream can be remarshaled.

Calling `Wr.Flush` on a surrogate writer flushes all outstanding data to the concrete writer and flushes the concrete writer. Calling `Wr.Close` flushes and then closes both the surrogate and the concrete writer. Similarly, a call on `Rd.Close` on a surrogate closes both readers.

Clients who marshal streams retain responsibility for closing them. For example, `Rd.Close` on a surrogate can fail due to the network, leaving the owner responsible for closing the concrete reader. The `WeakRef` interface can be used to register a GC cleanup procedure for this purpose.

The `ReleaseWr` procedure is used to shut down a surrogate writer so that the underlying writer can be reused. It flushes any buffered data, closes the surrogate, and frees any network resources associated with the surrogate. It leaves the concrete writer in a state where it can be reused locally or remarshaled.

Similarly. the `ReleaseRd` procedure is used to shut down a surrogate reader so that the underlying reader can be reused. It closes the surrogate, frees any network resources associated with the surrogate, and leaves the concrete reader in a state where it can be reused locally or remarshaled. There is an important difference between releasing readers and writers:



`ReleaseRd` discards any data buffered in the surrogate or in transit.

```
INTERFACE NetStream;

IMPORT Rd, Wr, Thread;

PROCEDURE ReleaseRd(rd: Rd.T)
    RAISES {Rd.Failure, Thread.Alerted};
```

*If `rd` is a surrogate reader, release all network resources associated with `rd`, discard all buffered data, close `rd`, but do not close the concrete reader for `rd`. This procedure is a no-op if `rd` is not a surrogate.*

```
PROCEDURE ReleaseWr(wr: Wr.T)
    RAISES {Wr.Failure, Thread.Alerted};
```

*If `wr` is a surrogate writer, flush `wr`, release all network resources associated with `wr`, close `wr`, but do not close the concrete writer for `wr`. This procedure is a no-op if `wr` is not a surrogate.*

```
END NetStream.
```

### 3.3 NetObjNotifier interface

The `NetObjNotifier` interface allows the holder of a surrogate object to request notification of when the object's owner becomes inaccessible. This can be useful, for example, if it is necessary to remove surrogates from a table upon termination of the programs holding their corresponding concrete objects.

```
INTERFACE NetObjNotifier;

IMPORT NetObj;

TYPE
  OwnerState = {Dead, Failed};

  NotifierClosure = OBJECT METHODS
    notify(obj: NetObj.T; st: OwnerState);
  END;

PROCEDURE AddNotifier(obj: NetObj.T; cl: NotifierClosure);
```

*Arrange that a call to `cl.notify` will be scheduled when `obj` becomes inaccessible. If `obj` is not a surrogate object then `AddNotifier` has no effect. If `obj` is already inaccessible at the time `AddNotifier` is called, then a call to `cl.notify` is scheduled immediately.*

```
END NetObjNotifier.
```

The `notify` method of a `NotifierClosure` object is invoked when the concrete object corresponding to the surrogate `obj` becomes inaccessible. The procedure `AddNotifier` must have been called to enable this notification. There may be more than one `NotifierClosure` for the same surrogate. At notification time, the `st` argument is `Dead` if and only if the object

owner is known to be permanently inaccessible. Otherwise `st` is `Failed`. It is possible for `notify` to be called multiple times on the same object. Any invocations on `obj` are guaranteed to fail in a timely fashion subsequent to a closure notification with `st = Dead`.

In general, a surrogate object can still be collected if a notifier closure is registered for it. However, if the closure object contains a reference to the surrogate, then its registration might delay or prevent collection. Therefore this should be avoided.

Although this interface is organized to enable notification of owner death on a per object basis, in practice this is achieved by monitoring the state of the owner's address space. This means that death notification will be more or less simultaneous for all surrogates whose concrete objects have the same owner.

### 3.4 The stub generator

The stub generator is a program that generates stubs for Modula-3 network object types. There are restrictions on the subtypes of `NetObj.T` for which the stub generator can produce stubs; a network object type that obeys them is said to be *valid*. Here is a list of these restrictions:

1. A valid network object type must be pure, that is it cannot contain data fields, either in its declaration or in a revelation.
2. To generate stubs for a network object `I.T`, the stub generator must be able to determine a complete revelation for all opaque supertypes of `I.T` (including `I.T` itself, if it is opaque) up to `NetObj.T`.
3. A method argument may not be of type `PROCEDURE` or have a component that is of type `PROCEDURE`. (A network object with an appropriate method can always be sent instead of a procedure.)
4. A Modula-3 method declaration specifies the set of exceptions that the method can raise. It is possible to specify (via `RAISES ANY`) that any exception can be raised, but this is not allowed for a valid network object type.
5. The methods of the type and its supertypes must have distinct names.

Given a valid network object type, the stub generator lays down code that implements parameter marshaling and remote invocation for that type's methods. For both arguments and results, subtypes of `NetObj.T` are marshaled as network references, subtypes of `Rd.T` and `Wr.T` are marshaled as surrogate streams, and all other parameters are marshaled by copying. The copying is performed by the pickles package if the parameter is a reference.

`VALUE` and `READONLY` parameters are copied only once, from the caller to the owner of the object. `VAR` parameters are normally copied from caller to owner on the call, and from owner to caller when the call returns. The pragma `<*OUT*>` on a `VAR` parameter in a method declaration indicates that the parameter may be given an arbitrary legal value when the method is invoked. The stub generator may use this information to optimize method invocation by not copying the parameter's value from caller to owner. At present, the stub generator does not make this optimization.

Any change in marshaling protocol that would make stubs incompatible is implemented as a new version of the stub generator. Typically, the previous version will continue to be supported



for some time after the release of a new one. Thus, multiple versions of the stub generator may sometimes exist at the same time.

Stubs for multiple versions may be linked into the same program. Method invocation between two programs is possible so long as the owner and the caller have at least one common version of the stubs for the network object in question. The network object runtime will use the most recent version of the protocol that is available in both programs. This allows gradual migration of applications from the old to the new protocol.

## 4 Internal Interfaces

In this section we present the main internal systems interfaces. The typical programmer using network objects has no need to read them, but we present them here in order to document the structure of the system. These are the interfaces you would use to write a new stub generator, hand-code stubs for some particular network object type, or add a new transport to the system.

### 4.1 StubLib interface

This interface contains procedures to be used by stub code for invoking remote object methods and servicing remote invocations. Each stub module provides type-dependent network support for marshaling and unmarshaling method calls for a specific subtype of `NetObj.T`. Usually, stubs are built automatically. For each `NetObj.T` subtype `T` intended to support remote method invocation there must be both a client and a server stub. The client stub defines a subtype of `T` in which every method is overridden by a procedure implementing remote method invocation. Such a surrogate object is constructed by the network object runtime whenever a reference to a non-local object is encountered. The server stub consists of a single procedure of type `Dispatcher` that is called to unmarshal and dispatch remote invocations. A surrogate type and null dispatcher for `NetObj.T` are defined and registered by the network object system itself.

```
INTERFACE StubLib;

IMPORT Atom, AtomList, NetObj, Rd, Wr, Thread;

TYPE Conn <: ROOT;
```

A remote object invocation can be viewed as an exchange of messages between client and server. The messages are exchanged via an object of type `Conn`, which is opaque in this interface. The `StubConn` interface reveals more of this type's structure to clients who wish to hand-code stubs for efficiency. A `Conn` is unmonitored: clients must not access it from two threads concurrently.

```
TYPE
  Byte8 = BITS 8 FOR [0..255];
  DataRep = RECORD
    private, intFmt, floatFmt, charSet: Byte8;
  END;

VAR (*CONST*) NativeRep: DataRep;
```

The type DataRep describes the format used to encode characters, integers, and floating point numbers in network data. Data is always marshaled in the sender's native format. NativeRep is a runtime constant that describes the native format of the current environment.

Stubs may optimize in-line unmarshaling by first checking that the incoming representation is the same as the native one for all data types relevant to the call. If it is not, then the generic data unmarshaling routines at the end of this interface should be used.

Automatic conversion between the data representations is performed wherever possible. If conversion is impossible, NetObj.Error is raised with NetObj.UnsupportedDataRep in the argument atom list.

Concrete values for the elements of DataRep are not defined here as it is sufficient to compare against NativeRep and invoke the marshaling procedures defined below if the encoding is non-native.

```
TYPE
  Int32 =  BITS 32 FOR [-16_7FFFFFFF-1..16_7FFFFFFF];
  StubProtocol = Int32;

CONST
  NullStubProtocol = -1;
  SystemStubProtocol = 0;
```

The type StubProtocol indicates the version of the stub compiler used to generate a particular stub. Multiple stubs for the same network object can coexist within the same program (for example, the outputs of different stub compilers). During surrogate creation, the network object runtime negotiates the stub protocol version with the object owner.

NullStubProtocol is a placeholder to indicate the absence of a stub protocol value. The value SystemStubProtocol indicates the fixed stub encoding used by the runtime to implement primitives that operate prior to any version negotiation.

```
VAR (*CONST*) UnmarshalFailure: Atom.T;
```

UnmarshalFailure should be used as an argument to NetObj.Error whenever stubs encounter a network datum that is incompatible with the target type. For example, the stub code might encounter a CARDINAL value greater than LAST(CARDINAL) or an unrecognized remote method specification.

```
TYPE Typecode = CARDINAL;
```

Typecode is the type of those values returned by the Modula-3 TYPECODE operator.

```
PROCEDURE Register(
    pureTC: Typecode; stubProt: StubProtocol;
    surrTC: Typecode; disp: Dispatcher);
```

*Let T be the type whose typecode is pureTC, and let srgT be the type whose typecode is surrTC. Set the client surrogate type and dispatch procedure for T to be srgT and disp, respectively. The stubProt parameter indicates the stub compiler version that generated the stub being registered.*

The following constraint applies to stub registration. If stubs are registered for types A and B, where B is a supertype of A, then the protocol versions registered for B must be a superset of



the versions registered for A. If this rule is violated, attempts to invoke remote methods may raise NetObj.Error.

Note that a concrete object of type A will receive method invocations only for stub versions for which A is registered. This is true even if a supertype of A is registered with additional stub versions.

Register must be called before any object of type T is marshaled or unmarshaled.

### Client stub procedures

Here is a simplified sketch of the procedure calls performed by a client to make a remote call to a method of obj:

```
VAR
  c := StartCall(obj, stubProt);
  resDataRep: DataRep;
BEGIN
  <marshal to "c" the number of this method>
  <marshal to "c" the method arguments>
  resDataRep := AwaitResult(conn);
  <unmarshal from "c" the method results>
  <results will be in wire format "resDataRep">
  EndCall(c, TRUE)
END;
```

For both arguments and results, the sender always marshals values in its native format; the receiver performs any conversions that may be needed. The procedure result typically begins with an integer specifying either a normal return or an exceptional return. If a protocol error occurs, the client should call EndCall(c, FALSE) instead of EndCall(c, TRUE). This requires TRY FINALLY instead of the simple straight-line code above; a more complete example is presented in the next section.

Here are the specifications of the client protocol procedures:

```
PROCEDURE StartCall(obj: NetObj.T; stubProt: StubProtocol): Conn
    RAISES {NetObj.Error, Wr.Failure, Thread.Alerted};
```

*Return a connection to the owner of obj, write to the connection a protocol request to perform a remote method call to obj, using the data representation NativeRep. The value stubProt is the stub protocol version under which the arguments and results will be encoded.*

Upon return from StartCall, the client stub should marshal a specification of the method being invoked followed by any arguments.

```
PROCEDURE AwaitResult(c: Conn): DataRep
    RAISES {NetObj.Error, Rd.Failure, Wr.Failure,
            Thread.Alerted};
```

*AwaitResult indicates the end of the arguments for the current method invocation, and blocks waiting for a reply message containing the result of the invocation. It returns the data representation used to encode the result message.*

Upon return from `AwaitResult` the client stub should unmarshal any results.

```
PROCEDURE EndCall(c: Conn; reUse: BOOLEAN)
    RAISES {NetObj.Error, Rd.Failure, Wr.Failure,
            Thread.Alerted};
```

*`EndCall` must be called at the end of processing a remote invocation, whether or not the invocation raised an exception. The argument `reUse` must be `FALSE` if the client has been unable, for any reason, to unmarshal either a normal or exceptional result. It is always safe to call `EndCall` with `reUse` set to `FALSE`, but performance will be improved if `reUse` is `TRUE` whenever possible.*

`EndCall` determines, by examining `c`, whether the result message requires acknowledgement, that is, whether the result contained any network objects. If an acknowledgement is required, it is sent. `EndCall` then releases `c`. After `EndCall` returns, `c` should not be used.

### Server dispatcher procedures

Next we consider the server-side stub, which consists of a registered dispatcher procedure.

```
TYPE Dispatcher = PROCEDURE(
  c: Conn; obj: NetObj.T; rep: DataRep; stubProt: StubProtocol)
  RAISES {NetObj.Error, Rd.Failure, Wr.Failure, Thread.Alerted};
```

A procedure of type `Dispatcher` is registered for each network object type `T` for which stubs exist. The dispatcher is called by the network object runtime when it receives a remote object invocation for an object of type `T`. The `rep` argument indicates the data representation used to encode the arguments of the invocation. The `stubProt` argument indicates the version of stub protocol used to encode the call arguments. The same protocol should be used to encode any results.

The dispatcher procedure is responsible for unmarshaling the method number and any arguments, invoking the concrete object's method, and marshaling any results.

Here is a simplified sketch of a typical dispatcher:

```
PROCEDURE Dispatch(c, obj, rep) =
  BEGIN
    <unmarshal from "c" the method number>
    <unmarshal from "c" the method arguments>
    <arguments will be in the wire format "rep">
    <call the appropriate method of "obj">
    StartResult(c);
    <marshal to "c" the method result or exception>
  END Dispatch;
```

Here is the specification of `StartResult`:

```
PROCEDURE StartResult(c: Conn)
    RAISES {Wr.Failure, Thread.Alerted};
```

*`StartResult` must be called by the server stub to initiate return from a remote invocation before marshaling any results.*



Upon return from `StartResult` the stub code should marshal any results or error indications.

### Marshaling of reference types

The following procedures are made available for marshaling of subtypes of `REFANY`.

```
PROCEDURE OutRef(c: Conn; r: REFANY)
    RAISES {Wr.Failure, Thread.Alerted};
```

*Marshal the data structure reachable from `r`. Certain datatypes are handled specially: subtypes of `NetObj.T` are marshaled as network references. Subtypes of `Rd.T` and `Wr.T` are marshaled as surrogate streams. The types `TEXT` and `REF ARRAY OF TEXT` are marshaled by copying via custom code for speed. All others are marshaled by copying as pickles. Subtypes of `NetObj.T`, `Rd.T`, and `Wr.T` which are embedded within other datatypes are also marshaled by reference.*

```
PROCEDURE InRef(c: Conn; rep: DataRep; tc:=-1): REFANY
    RAISES {NetObj.Error, Rd.Failure, Thread.Alerted};
```

*Unmarshal a marshaled subtype of `REFANY` as pickled by `OutRef`. If `tc` is non-negative, it is the typecode for the intended type of the reference. The exception `NetObj.Error(UnmarshalFailure)` is raised if the unpickled result is not a subtype of this type. If `tc` is negative, no type checking is performed.*

`OutRef` and `InRef` use pickles and therefore are affected by any custom pickling procedures that have been registered. The network objects runtime itself registers procedures for pickling network objects and streams. Therefore, for any network objects or streams that are reachable from the reference `r` are pickled by reference as described elsewhere in this report.

### Marshaling of generic data

The `StubLib` interface also provides a suite of procedures to facilitate the marshaling and unmarshaling of primitive data types. For the sake of brevity, we use the `INTEGER` datatype as an example. The actual interface provides routines to handle all other types that are primitive in Modula-3 such as `CARDINAL`, `REAL`, and `LONGREAL`.

```
PROCEDURE OutInteger(c: Conn; i: INTEGER)
    RAISES {Wr.Failure, Thread.Alerted};


PROCEDURE InInteger(c: Conn; rep: DataRep;
    min := FIRST(INTEGER); max := LAST(INTEGER)): INTEGER
    RAISES {NetObj.Error, Rd.Failure, Thread.Alerted};
```

Since all marshaling procedures output their parameters in the native representation of the sender, they can be trivially replaced by inline code that manipulates the writer buffer directly. All unmarshaling procedures decode the incoming wire representation as indicated by `rep` and return their results in native format. These procedures can be replaced by inline unmarshaling code whenever the relevant elements of `rep` match the corresponding elements of `NativeRep`.

Finally, the StubLib interface provides two procedures for raising NetObj exceptions conveniently:

```
PROCEDURE RaiseUnmarshalFailure() RAISES {NetObj.Error};
```

*Raise `NetObj.Error` with UnmarshalFailure in the argument list.*

```
PROCEDURE RaiseCommFailure(e: AtomList.T) RAISES {NetObj.Error};
```

*Raise `NetObj.Error` with the result of prepending `NetObj.CommFailure` to `e`.*

```
END StubLib.
```

### 4.2 An example stub

This subsection illustrates the use of the StubLib interface by presenting hand-generated stub code for a simple network object type, Example.T:

```
INTERFACE Example;

IMPORT NetObj, Thread;

EXCEPTION Invalid;

TYPE
  T = NetObj.T OBJECT METHODS
    get(key: TEXT) : TEXT
        RAISES {Invalid, NetObj.Error, Thread.Alerted};
  END;

END Example.
```

Notice that the object methods must raise NetObj.Error or else communications failures will be treated as checked runtime errors. Also notice that Thread.Alerted is present in the RAISES clause of all methods. This is not required, but is strongly advised. If an object method does not propagate the Thread.Alerted exception, then not only is it impossible to alert remote invocations, but the server implementation must guarantee that Alerted will never be raised. This guarantee must hold even though the network object runtime uses Thread.Alert to recover server threads when the client address space dies.

The following module defines and registers both client and server stubs for Example.T:

```
MODULE Example;

IMPORT NetObj, StubLib, Thread, Rd, Wr;

TYPE P = { Get };  R = { OK, Invalid };
```

The enumerated types P and R define values to be associated with the methods of T and with the various results (normal return or exception) of these methods.

```
TYPE StubT = T OBJECT OVERRIDES get := SurrogateGet; END;
```

The type StubT is the surrogate object type for T. It provides method overrides that perform remote invocation.



```
CONST StubVersion = StubLib.SystemStubProtocol;
```

This constant will be set by the stub generator to denote the stub generator version that created a given stub.

```
PROCEDURE SurrogateGet (t: StubT; key: TEXT) : TEXT
    RAISES {Invalid, NetObj.Error, Thread.Alerted} =
  VAR reuse := FALSE;
      rep: StubLib.DataRep;
      c: StubLib.Conn;
      res: TEXT;
  BEGIN
    TRY
      c := StubLib.StartCall(t, StubVersion);
      TRY
        StubLib.OutInt32(c, ORD(P.Get));
        StubLib.OutRef(c, key);
        rep := StubLib.AwaitResult(c);
        CASE StubLib.InInt32(c, rep) OF
        | ORD(R.OK) =>
            res := StubLib.InRef(c, rep, TYPECODE(TEXT));
            reuse := TRUE;
        | ORD(R.Invalid) =>
            reuse := TRUE;
            RAISE Invalid;
        ELSE
            StubLib.RaiseUnmarshalFailure();
        END;
      FINALLY
        StubLib.EndCall(c, reuse);
      END;
    EXCEPT
    | Rd.Failure(ec) => StubLib.RaiseCommFailure(ec);
    | Wr.Failure(ec) => StubLib.RaiseCommFailure(ec);
    END;
    RETURN res;
  END SurrogateGet;
```

Invoke is the server stub dispatcher for T. It is called when the network object runtime receives a method invocation for an object of type T.

```
PROCEDURE Invoke(
    c: StubLib.Conn; obj: NetObj.T; rep: StubLib.DataRep;
    <*UNUSED*> stubProt: StubLib.StubProtocol)
    RAISES {NetObj.Error, Rd.Failure, Wr.Failure,
            Thread.Alerted} =
  VAR t := NARROW(obj, T);
  BEGIN
    TRY
      CASE StubLib.InInt32(c, rep) OF
      | ORD(P.Get) => GetStub(c, t, rep);
      ELSE StubLib.RaiseUnmarshalFailure();
```

```
      END;
    EXCEPT
    | Invalid =>
        StubLib.StartResult(c);
        StubLib.OutInt32(c, ORD(R.Invalid));
    END;
  END Invoke;
```

There is one server side stub procedure for each method of `T`.

```
PROCEDURE GetStub (c: StubLib.Conn; t: T; rep: StubLib.DataRep)
    RAISES {Invalid, NetObj.Error, Rd.Failure,
            Wr.Failure, Thread.Alerted} =
  VAR key, res: TEXT;
  BEGIN
    key := StubLib.InRef(c, rep, TYPECODE(TEXT));
    res := t.get(key);
    StubLib.StartResult(c);
    StubLib.OutInt32(c, ORD(R.OK));
    StubLib.OutRef(c, res);
  END GetStub;
```

All stub code is registered with the network object runtime by the main body of the stub module. The protocol number is set to the stub protocol constant defined above.

```
BEGIN
  StubLib.Register(
    TYPECODE(T), StubVersion, TYPECODE(StubT), Invoke);
END Example.
```

### 4.3 StubConn interface

A `StubLib.Conn` represents a bidirectional connection used to invoke remote methods by the network objects runtime. Here we reveal that a connection `c` consists of a message reader `c.rd` and a message writer `c.wr`.

Connections come in matching pairs; the two elements of the pair are typically in different address spaces. If `c1` and `c2` are paired, the target of `c1.wr` is equal to the source of `c2.rd`, and vice versa. Thus the messages written to `c1.wr` can be read from `c2.rd`, and vice versa.

```
INTERFACE StubConn;

IMPORT MsgRd, MsgWr, StubLib;

REVEAL StubLib.Conn <: Public;

TYPE Public = OBJECT rd: MsgRd.T; wr: MsgWr.T END;

END StubConn.
```

The types `MsgWr.T` and `MsgRd.T` are subtypes of the standard Modula-3 stream types `Wr.T` and `Rd.T`; they are described in detail in the next subsection. Since they are subtypes, any of



the standard stream operations can be used on them. For example, in a hand-coded stub you could replace the pair

```
StubLib.OutByte(c, byte)
b := StubLib.InByte(c)
```

with the pair

```
Wr.PutChar(c.wr, VAL(byte, CHAR))
b := ORD(Rd.GetChar(c.rd)).
```

The gain in speed from this change will be very modest. To make optimization worthwhile, you will want to make direct access to the buffers in the reader and writer. To do this, import the `RdClass` and `WrClass` interfaces[18]. Importing these interfaces will allow you to write stubs that operate directly on the reader and writer buffers.

If you use this optimization, you will have to be careful about locks. All readers and writers contain an internal lock used to serialize operations. It is a requirement of the `StubLib` interface that all parameters of type `Conn` be passed with both streams unlocked. It is a further requirement that no client thread operate on the streams while an activation of a `StubLib` procedure is in progress.

### 4.4 Message readers and writers

The byte streams of the readers and writers in a `StubLib.Conn` are divided into segments called *messages*. Messages are convenient for delineating call and return packets, and seem essential for sending both data and control information for surrogate streams.

We define the types `MsgRd.T` and `MsgWr.T` to present the abstraction of a stream of messages. A message is a sequence of bytes terminated by an end-of-message marker. The initial position is at the start of the first message. Messages can be of zero length.

If the end-of-message marker is encountered while reading from a `MsgRd.T`, it is represented by `EndOfFile` on the reader. The `nextMsg` method can be used to advance to the next message in the stream. This method waits for the next message and returns `TRUE` when it becomes available. A return value of `FALSE` indicates that there are (and will be) no further messages. The reader's current position is set to zero on return from `nextMsg`, and the reader no longer reports `EndOfFile` (unless of course the next message is zero length).

If `nextMsg` is invoked when the reader is not at `EndOfFile`, the remaining bytes in the current message are skipped.

As for all readers, calling `Rd.Close` on a `MsgRd.T` releases all associated resources.

Here is a listing of the interface:

```
INTERFACE MsgRd;

IMPORT Thread, Rd;

TYPE
  T = Rd.T OBJECT METHODS
    nextMsg(): BOOLEAN RAISES {Rd.Failure, Thread.Alerted};
  END;

END MsgRd.
```

As with a MsgRd.T, the nextMsg method of a MsgWr.T can be used to end the current message and position the writer at the start of the next message. The writer's current position is reset to zero on return from nextMsg.

Invoking Wr.Flush on a MsgWr.T flushes the current buffer to the abstract writer target, but does not end the current message.

As for all writers, calling Wr.Close on a MsgWr.T releases all associated resources. Close also flushes and terminates the current message. This means that a zero-length message is sent at close time if no data has been written into the current message (for example, directly after nextMsg or writer initialization).

Here is a listing of the interface:

```
INTERFACE MsgWr;

IMPORT Thread, Wr;

TYPE
  T = Wr.T OBJECT METHODS
    nextMsg() RAISES {Wr.Failure, Thread.Alerted};
  END;

END MsgWr.
```

There are two final clauses in the specification of message readers and message writers. First, their buffers must be word-aligned in memory. More precisely, if byte i in the data stream is stored in the buffer at memory address j, then i and j must be equal modulo the machine word size. This requirement allows optimized stubs to read and write scalar word values from the buffer efficiently. Second, their buffers must not be too small. More precisely, when the nextMsg method of a writer returns, there must be at least 24 bytes of free space in the writer buffer, and when the nextMsg method of a reader returns, there must be at least 24 bytes of message data in the reader buffer. This requirement allows the runtime to efficiently read and write the headers required by the network object protocol.

### 4.5 Transport interface

The Transport interface separates the main part of the network object runtime system from the parts that deal with low-level communication. It is the interface that must be implemented to extend the system to use new communication protocols. The interface is reasonably narrow:

```
INTERFACE Transport;

IMPORT NetObj, NetObjNotifier, StubLib, StubConn, Thread;

TYPE
  T <: Public;
  Endpoint = TEXT;
  Public = OBJECT METHODS
    fromEndpoint(e: Endpoint): Location;
    toEndpoint(): Endpoint;
    serviceCall(t: StubLib.Conn): (*reUse*) BOOLEAN
      RAISES {Thread.Alerted};
  END;
```



```
  Location <: LocationP;
  LocationP = OBJECT METHODS
    new(): StubLib.Conn RAISES {NetObj.Error, Thread.Alerted};
    free(c: StubLib.Conn; reUse: BOOLEAN);
    dead(st: NetObjNotifier.OwnerState);
  END;

  Conn = StubConn.Public BRANDED OBJECT
    loc: Location
  END;

REVEAL
  NetObj.Address = BRANDED REF ARRAY OF Endpoint;
  StubLib.Conn <: Conn;

END Transport.
```

The main ideas in the interface were described earlier. To summarize these briefly:

- A Transport.T is an object that manages connections of some particular class (*e.g.*, TCP).
- A Transport.Location is an object that creates connections of some particular class to some particular address space.
- A Transport.Endpoint is a transport-specific name for an address space (*e.g.*, an IP address plus a port number plus a non-reusable process ID).
- The fromEndpoint method of a transport converts an endpoint into a location, or into NIL if the endpoint and transport are of different classes.

Here are specifications for the methods of a Transport.T:

- The toEndpoint method returns an endpoint for the address space itself. The resulting endpoint should be recognized by the fromEndpoint method of transports of the same class anywhere in the network. That is, if program instance P calls tr.toEndpoint(), producing an endpoint ep, then the call tr1.fromEndpoint(ep) executed in any program instance either returns NIL (if tr and tr1 are of different classes) or returns a location that generates connections to P.
- Transports are required to provide the threads that listen to the server sides of connections. When a message arrives on the connection indicating the beginning of a remote call, the threads are required to call the serviceCall method of their transport. The default value of this method locates and calls the dispatcher procedure. Ordinarily a transport implementation will not need to override the serviceCall method. If conn is the argument to serviceCall, then at entry conn.rd is positioned at the start of the incoming message. The serviceCall method processes the incoming remote invocation and sends the result on conn.wr. If it returns TRUE, then the remote invocation was processed without error and the transport can cache the connection. If it returns FALSE, a protocol error occurred during the call, and the transport implementation should destroy the connection.

And here are the specifications for the methods of a `Transport.Location`:

- The `new` method of a location returns a connection to the address space for which it is a location. The call `loc.new()` returns a connection whose server side is that address space and whose client side is the program instance making the call. The caller must pass the resulting connection to `loc.free` when it is finished with it.

- The call `loc.free(c, reUse)` frees the connection `c`, which must have been generated by `loc.new()`. If `reUse` is `TRUE`, the client asserts that the connection is in a suitable state for executing another remote method call. In particular, `c.wr` must be positioned at the beginning of a message.

- A transport is responsible for monitoring the liveness of program instances for which it has locations or connections. The method of monitoring depends on the transport. For example, the transport might periodically ping the other program instances. A program is considered dead if it exits, crashes, or if the underlying communication network cannot reach it for an appreciable amount of time. Suppose that `loc` is a location that generates connections to some program instance `P`. If `P` dies, the transport that provided `loc` is responsible for calling the method `loc.dead(st)`. (The network object runtime implements this method; the transport should not override it.) The argument `st` indicates whether the transport has detected a permanent failure, or one that is potentially transient. In addition to calling `loc.dead`, the transport is responsible for alerting all threads it has spawned to handle method invocations on behalf of `P`.

A transport is expected to manage the connections it creates. If creating connections is expensive, then the transport's locations should cache them. If maintaining idle connections is expensive, then the transport's locations should free them. Often connections are time-consuming to create, but then tie up scarce kernel resources when idle. Therefore transports typically cache idle connections for a limited amount of time.

The `Transport` interface reveals the representation of `NetObj.Address`: an address is simply an array of endpoints for the program instance designated by the address. The endpoints are generally of different transport classes; they provide alternative ways of communicating with the program instance. The modules of the network object runtime that require this revelation are exactly the modules that import the transport interface, so this is a convenient place to put it.

The `Transport` interface also reveals more information about the type `StubLib.Conn`. If `t` is a `StubLib.Conn`, then `t.loc` is a `Location` that generates connections to the program instance at the other end of `t`. The connections generated by `t.loc` connect the same pair of program instances that `t` connects, but if `t` is a handle on the server side of the connection, then the connections generated by `t.loc` will reverse the direction of `t`: their client side will be `t`'s server side, and vice versa (so-called back connections). On the other hand, if `t` is a handle on the client side of the connection, then the connections generated by `t.loc` will be in the same direction as `t`. A transport must ensure that the `loc` field is defined in all connections returned by any of its locations.



# 5 Performance

Our system was designed and implemented in a year by the four authors. The network object runtime is 4000 lines, the stub generator 3000 lines, the TCP transport 1500 lines, the pickle package 750 lines, and the network object agent 100 lines. All the code is in Modula-3.

We haven't attempted extensive performance optimization, but we have measured the times for some basic operations. The numbers given in Table 1 were taken using Digital workstations equipped with DECchip 21064 processors (at 175 MHz) running OSF/1 and communicating over a 100 megabit/sec AN1 network[22]. The numbers include the cost of Modula-3 runtime checks.

Table 1: Sample remote invocation timings

| *Call parameters* | *Elapsed time/call* |
|---|---|
| Null call | 960 usec |
| Ten integer arguments | 1010 usec |
| `REF CHAR` argument | 1280 usec |
| Linked list argument | 5200 usec |
| Network object argument $(s)$ | 1030 usec |
| Network object argument $(c)$ | 1050 usec |
| Network object argument $(c, d)$ | 2560 usec |
| Network object result $(s)$ | 1180 usec |
| Network object result $(c)$ | 1190 usec |
| Network object result $(c, d)$ | 2680 usec |

$(c)$ concrete object marshaled

$(s)$ surrogate object marshaled

$(d)$ dirty call required

On our test configuration, it takes 660 microseconds for a C program to echo a TCP packet from user space to user space. A null network object method invocation takes an additional 300 microseconds. The difference is primarily due to the cost of two Modula-3 user space context switches (64 microseconds), the cost of marshaling and unmarshaling the object whose null method is being invoked, and the cost of the wire protocol used to frame invocation and result packets.

The ten integer argument test shows that the incremental cost of an integer argument is about 5 microseconds. The `REF CHAR` test measures the cost of marshaling a small data structure by pickling. This minimal use of the pickle machinery adds an additional 220 microseconds to the null call. The linked list test measures the cost of marshaling a complex data structure, in this case a doubly-linked list with 25 elements. The additional cost per element is roughly 80 microseconds.

The next six tests show the total cost of various calls involving a single network object argument or result (in addition to the object whose method is being invoked). An "(s)" indicates that the argument or result is marshaled as a surrogate and unmarshaled as a concrete object. A "(c)" indicates that the argument or result is marshaled as a concrete object and unmarshaled

as a surrogate. A "(d)" indicates that a dirty call is required.

The incremental cost of a surrogate network object argument that does not lead to a dirty call is roughly 70 microseconds. A concrete network object argument is somewhat more expensive. If a dirty call is required, there is an additional cost of about 1500 microseconds.

Network object results are more expensive than arguments, because of the acknowledgement that must be sent when the result message contains a network object. In the tests that do not involve dirty calls, this cost shows up as a difference of approximately 150 microseconds, but in the tests that do involve dirty calls the cost seems to be lost in the noise.

We also measured the performance of marshaled readers and writers. Since there is only a minimal layer of protocol between marshaled data streams and the underlying network transport, there is little difference in bandwidth. In the test configuration described above, our implementation delivers over 95 percent of the full network bandwidth (100 MBits/sec). We attribute our failure to achieve full network bandwidth to the cost of user-space thread emulation, Unix non-blocking I/O, and TCP protocol overhead.

The purpose of our project was to find an attractive design, not to optimize performance, and our numbers reflect this. Nevertheless, the performance of our system is adequate for many purposes. It is competitive with the performance of commercially available RPC systems[20], and we believe that our design does not preclude the sort of performance optimizations reported in the literature[23, 27]. Furthermore, our use of buffered streams for marshaling permits careful hand-tuning of stubs while still offering the flexibility of a general purpose stream abstraction.

# 6 Experience

Our network objects system has been working for almost two years. Several projects have built on the system, including the Siphon distributed software repository[21], the Argo teleconferencing system[9], and the Obliq distributed scripting language[6]. We report on experience gained from these projects here.

## 6.1 Siphon

The Siphon system consists of two major components. The *packagetool* allows software packages to be checked in and out from a repository implemented as a directory in a distributed file system. The repository is replicated for availability. When a new version of a package is checked in, it is immediately visible to all programmers using the local area network. All the files in the new version become visible simultaneously.

The *siphon* component is used to link repositories that are too far apart to be served by the same distributed file system. (In our case, the two repositories of interest are 6000 miles apart.) When a new version of a package is checked in at one repository, the siphon copies it to the other repository within a few hours. Again, all new files in a single package become visible simultaneously.

An earlier version of this system was coded with conventional RPC. The current version coded with network objects is distinctly simpler, for several reasons.

First, pickles and network streams simplified the interfaces. For example, to fetch a package, the old siphon enumerated the elements of the directory by repeated RPC calls; the new siphon



obtains a linked structure of directory elements in one call. Also, the old siphon used multiple threads copying large buffers of data to send large files; the new siphon uses a network stream.

Second, third-party transfers eliminated an interface. The previous version of the siphon would pull a new version of a package from one of the source replicas, push it over the wide area network to a partner siphon at the other site, which would cache it on its disk and then push it to each of the destination replicas. Thus both a pull and a push interface were required. The new siphon transfers the object implementing the pull interface to its remote partner, which then pulls the files from the source replica directly. Thus the push interface was eliminated.

Third, we can take advantage of the ability to plug new transports into the system. Data compression is known to significantly increase bandwidth over wide area networks. Although we have not had need to do so, we could easily provide a subtype of `Transport.T` that automatically compresses and decompresses data. This would move the compression code out of the application and into a library where it could easily be reused.

In writing the Siphon system we deliberately stressed distributed garbage collection by performing no explicit deallocations. The results of this strategy were mixed. There were no serious memory leaks and garbage collection overhead was not a problem, but automatic reclamation was not as timely as we would have liked. The fundamental problem is that performance tradeoffs made in the local collector may not be appropriate for the distributed case. For example, it may be perfectly acceptable for the collector to delay reclaiming some small object, but if the object is a surrogate this can prevent the reclamation of an object that holds some important resource. We also found that the Modula-3 local collector occasionally fails to free unreachable objects because of its conservative strategy, and that this is more of a problem for distributed computations than for local ones. We conclude that for an application like the Siphon system that holds important resources like Unix file handles, it is necessary either to rewrite the local collector or to code the application to free resources explicitly.

## 6.2 Argo

Argo is a desktop telecollaboration system using audio, video, a shared whiteboard, and shared application windows to facilitate cooperation among multiple users who may be separated across long distances. A central function of Argo is conference control, which coordinates the sharing of the various media and tools to provide a coherent model of group collaboration. This shared state is held in a small special-purpose database that is implemented in terms of network objects.

The conference control server's database defines three types of objects: *users*, *conferences*, and *members*. A user object represents a human user of the system; a conference object represents an collaboration, such as a virtual conference room. The basic event is that users join and leave conferences. A member represents a $\{\text{user, conf}\}$ pair and is created automatically when a user joins a conference. Each object in the database has a list of properties whose meaning is defined by client programs. General property lists were chosen instead of a predefined hierarchy of subtypes to increase independence among client programs.

The primary function of the server is to notify clients of events that occur in its database. This is done via callbacks. A client program registers a *handler* object and an *event filter*. When an event passes the filter, an appropriate callback method of the client's handler object is invoked, and is passed the relevant database object(s) that were involved in the event. For

example, when a user joins a conference, the client programs involved in the conference are notified and can obtain the properties of the new user's object.

Callbacks like those employed in Argo are commonplace in non-distributed applications, and network objects extend this style to distributed programming. Nonetheless, transparent distributed invocation is not a panacea; distributed programs are inherently more complex than centralized ones. For example, callbacks from the Argo server to clients cannot be treated like local callbacks: the server must protect itself against clients that crash or are too slow, especially when locks are involved. Although good tools can hide many of the tiresome details of distributed programming, they do not yet eliminate the fundamental issues that must be faced in designing a robust distributed system.

The ease of defining, debugging and modifying the Argo conference control system and its protocol via network objects has been quite striking. Because of the leverage provided by Modula-3 and network objects, the entire conference control server implementation contains only 1400 lines of source code.

### 6.3 Obliq

Obliq is a lexically-scoped, untyped, interpreted language that supports distributed object-oriented computation. Obliq objects have state and are local to a site, but computations can roam over the network.

The characteristics of Modula-3 network objects had a major influence on Obliq, not just in the implementation, but also in the language design. All Obliq objects are implemented as network objects, so there is no artificial separation between local Obliq objects and those that may be remotely accessed. Also, all Obliq program variables are network objects, including global variables. Therefore, a remote computation can still access global state at the site at which it originated. Two elements of the network objects system were particularly useful in simplifying the Obliq implementation. Distributed garbage collection relieved concerns about space reclamation, and marshaling via pickles made it easy to transmit complex data structures such as the runtime representation of Obliq values.

### 6.4 Other work

Network objects are also in use in a system for continuous performance monitoring. The system provides a *telemonitoring* server which can be directed by the user to retrieve and process event logs from remote programs. The logs are communicated to the telemonitor via remote readers. Because a monitored program may disconnect from one telemonitor and reconnect to another, this application requires the ability to disconnect a network reader and reconnect the underlying data stream to a different process. This works, but requires extra logic in the application. In order to guarantee that no data is lost, the telemonitor must call the monitored program to indicate that it is disconnecting.

Our experience with the continuous monitoring project showed us that the design of reader and writer marshaling is trickier than it would at first appear. The semantics of marshaled streams are surprisingly difficult to specify and there are many design tradeoffs involved. We chose to give rather weak semantics, barring third-party marshaling of streams and specifying little about the state of a concrete stream after marshaling. Providing stronger guarantees would have had a high cost in either throughput or complexity.



We have also implemented secure network objects. A secure network object method can authenticate its caller, which allows security based on access control lists. A secure network object can also be passed to third parties and cannot be forged, which allows security based on capabilities. A client can use secure and ordinary insecure network objects together in the same application, incurring a performance penalty only for secure invocations or third party transfers of secure objects. Leendert van Doorn has implemented secure network objects[28], but the implementation has not yet been released or extensively used.

We learned three things from our secure network object implementation. First, the design of the secure system followed naturally from that of the insecure system. A large-scale redesign was not necessary. Second, because we use readers and writers for marshaling, it was easy to insert reader and writer subtypes that perform the cryptographic functions necessary for network security. Finally, because of our desire to make remote invocations transparent, we did not identify the caller via an implicit argument to the owner's method. Instead, we require callers to pass an explicit extra argument if they want to identify themselves. This argument is of a distinguished type that is marshaled specially.

Several users of network objects have noted our lack of support for object persistence. We note that Carsten Weich has recently added support for stable objects to the Modula-3 runtime system. He captures a stable snapshots of an object's state and then writes the arguments of update methods into a redo log using techniques quite similar to marshaling.

Providing support for stable concrete objects is not the whole story, however. In a distributed system, it can be valuable for a surrogate object to remain valid after restart of the owner's address space. We have not implemented this facility, although we believe that it would be straightforward to do so with a library built on top of the existing network object system.

## 7 Conclusion

The narrowest surrogate rule is flexible, but the associated type checking is dynamic rather than static, which has all the usual disadvantages. Because programs can be relinked and re-run at any time, it seems impossible to provide purely static type checking in a distributed environment. Dynamic checking imposes a burden of discipline upon programmers. The most common failure during the early stages of debugging a network objects application is a narrow fault (failure of a runtime type-check). For example, if a programmer forgets to link in stubs for a subtype `A` of `NetObj.T`, an import of an `A` object will succeed, but the resultant surrogate will have type `NetObj.T` and any attempt to `NARROW` it to an `A` object will fail.

Even an application that has been in service for a long time can crash with a narrow fault if some programmer carelessly changes a low-level interface and rebuilds another program with which the application communicates. Because of this danger, programmers should minimize external dependencies when defining a network object subtype. It is also important that the implementation of type fingerprinting not introduce spurious dependencies.

Programmers appreciate the narrowest surrogate rule, and more than one has asked for comparable flexibility in the case of ordinary objects. (If an attempt is made to unpickle an ordinary object into a program that does not contain the type of the object, an exception is raised.) But in this case liberality seems unsound. Suppose type `AB` is derived from `A`, and that we contrive to send a copy (rather than a reference) of an object of type `AB` into a program that knows the type `A` but not the type `AB`. One can imagine doing this either by ignoring the `B` data fields and

methods, or by somehow holding them in reserve. In either case, the new program can operate on the A part of the state, for example by reading or writing data fields or by calling methods of A. However, there is no guarantee that these operations will be valid, since the original type AB may have overridden some of the methods of A; for example in order to accommodate a change in the meaning of the representation of the A fields. The narrowest surrogate rule seems sound only when objects are transmitted by reference.

The programmers that have used network objects have found the abstractions it offers to be simple yet powerful. By providing transparent remote invocation through Modula-3 objects, we eliminate many of the fussy details that make RPC programming tedious. Through the use of pickles, and by implementing third party network object transfers and marshaled abstract streams, we remove many restrictions about what can be marshaled, and we do so without increasing the complexity of generated stubs. The strength of our system comes not from proliferating features, but from carefully analyzing the requirements of distributed programming and designing a small set of general features to meet them.

### Acknowledgements

We are grateful for help and suggestions from Bob Ayers, Andrew Black, John Ellis, David Evers, Rivka Ladin, Butler Lampson, Mark Manasse, and Garret Swart. We thank Luca Cardelli and Dave Redell for providing words describing the Obliq and Argo projects. Paul Leach, Sharon Perl, Allan Heydon, and Cynthia Hibbard made helpful comments on the presentation of this report.

## References

[1] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden system: A technical review. *IEEE Trans. Software Engineering*, 11(1):43–59, 1985.

[2] Henri E. Bal, M. Frans Kaashoek, and Andy S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Software Engineering*, 18(3):190–205, 1992.

[3] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber. Distributed garbage collection for Network Objects. Research report 116, Systems Research Center, Digital Equipment Corp., 1993.

[4] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure call. *ACM Trans. Computer Systems*, 2(1):39–59, 1984.

[5] Andrew P. Black and Yeshayahu Artsy. Implementing location independent invocation. *IEEE Trans. Parallel and Distributed Systems*, 1(1):107–119, 1990.

[6] Luca Cardelli. Obliq: A language with distributed scope. Research report 122, Systems Research Center, Digital Equipment Corp., 1994.



[7] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proc. 12th ACM Symposium on Operating System Principles*, pages 147–158, 1989.

[8] Graeme N. Dixon, Santosh K. Shrivastava, and Graham D. Parrington. Managing persistent objects in Arjuna: a system for reliable distribed computing. Technical report, Computing Laboratory, University of Newcastle upon Tyne, undated.

[9] Hania Gajewska, Jay Kistler, Mark S. Manasse, and David D. Redell. Argo: A system for distributed collaboration. In *Proc. ACM Multimedia '94*, pages 433–440, 1994.

[10] David K. Gifford and Nathan Glasser. Remote pipes and procedures for efficient distributed communication. *ACM Trans. Computer Systems*, 6(3):258–283, 1988.

[11] Graham Hamilton, Michael L. Powell, and James G. Mitchell. Subcontract: A flexible base for distributed programming. Technical Report SMLI TR-93-13, Sun Microsystems Laboratories, 1993.

[12] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Trans. Programming Languages and Systems*, 4(4):527–551, 1982.

[13] Jim Horning, Bill Kalsow, Paul McJones, and Greg Nelson. Some useful Modula-3 interfaces. Research report 113, Systems Research Center, Digital Equipment Corp., 1993.

[14] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Trans. Computer Systems*, 6(1):109–133, 1988.

[15] K. Rustan M. Leino. Extensions to an object-oriented programming language for programming fine-grain multicomputers. Technical report CS-TR-92-26, California Institute of Technology, 1992.

[16] Barbara Liskov. Distributed programming in Argus. *CACM*, 31(3):300–312, 1988.

[17] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proc. 11th ACM Symposium on Operating System Principles*, pages 111–122, 1987.

[18] Greg Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, Englewood Cliffs, New Jersey, 1991. ISBN 0-13-590464-1.

[19] Object Management Group. *Common Object Request Broker: Architecture and Specification*, 1991. OMG Document 91.12.1.

[20] Open Software Foundation. *DCE Application Development Reference, volume 1, revision 1.0*, 1991.

[21] Francis Prusker and Edward Wobber. The Siphon: Managing distant replicated repositories. Research report 7, Paris Research Laboratory, Digital Equipment Corp., 1991.

[22] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. Research report 59, Systems Research Center, Digital Equipment Corp., 1990.

[23] Michael D. Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Trans. Computer Systems*, 8(1):1–17, 1990.

[24] Marc Shapiro. Structure and encapsulation in distributed systems: the proxy principle. In *IEEE International Conference on Distributed Computer Systems*, pages 198–204, 1986.

[25] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Celine Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–337, 1989.

[26] J. E. Stoy and C. Strachey. OS6—an experimental operating system for a small computer. Part 2: input/output and filing system. *The Computer Journal*, 15(3):195–203, 1972.

[27] Andrew S. Tannenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *CACM*, 33(12):46–63, 1990.

[28] Leendert van Doorn, Martín Abadi, Mike Burrows, and Edward Wobber. Secure Network Objects. Research report IR-385, Vrije Universiteit, Amsterdam, 1995.

# 26. Reliable Messages

The attached paper on reliable messages is Chapter 10 from the book *Distributed Systems: Architecture and Implementation*, edited by Sape Mullender, Addison-Wesley, 1993. It contains a careful and complete treatment of protocols for ensuring that a message is delivered at most once, and that if there are no serious failures it is delivered exactly once and its delivery is properly acknowledged.





# Reliable Messages and Connection Establishment

**Butler W. Lampson**

## 1 Introduction

Given an unreliable network, we would like to reliably deliver messages from a sender to a receiver. This is the function of the transport layer of the ISO seven-layer cake. It uses the network layer, which provides unreliable message delivery, as a channel for communication between the sender and the receiver.

Ideally we would like to ensure that

- messages are delivered in the order they are sent,
- every message sent is delivered exactly once, and
- an acknowledgement is returned for each delivered message.

Unfortunately, it's expensive to achieve the second and third goals in spite of crashes and an unreliable network. In particular, it's not possible to achieve them without making some change to stable state (state that survives a crash) every time a message is received. Why? When we receive a message after a crash, we have to be able to tell whether it has already been delivered. But if delivering the message doesn't change any state that survives the crash, then we can't tell.

So if we want a cheap deliver operation that doesn't require writing stable state, we have to choose between delivering some messages more than once and losing some messages entirely when the receiver crashes. If the effect of a message is idempotent, of course, then duplications are harmless and we will choose the first alternative. But this is rare, and the latter choice is usually the lesser of two evils. It is called 'at-most-once' message delivery. Usually the sender also wants an acknowledgement that the message has been delivered, or in case the receiver crashes, an indication that it might have been lost. At-most-once messages with acknowledgements are called 'reliable' messages.

There are various ways to implement reliable messages. An implementation is called a 'protocol', and we will look at several of them. All are based on the idea of tagging a message with an identifier and transmitting it repeatedly to overcome the unreliability of the channel. The receiver keeps a stock of *good* identifiers that it has never accepted before; when it sees a message tagged with a good identifier, it accepts it, delivers it, and removes that identifier from the good set. Otherwise, the receiver just discards the message, perhaps after acknowledging it. In order for the sender to be sure that its message will be delivered rather than discarded, it must

This paper originally appeared as chapter 10 in *Distributed Systems*, ed. S. Mullender, Addison-Wesley, 1993, pp 251-281. It is the result of joint work with Nancy Lynch and Jørgen Søgaard-Andersen.

tag the message with a good identifier.

What makes the implementations tricky is that we expect to lose some state when there is a crash. In particular, the receiver will be keeping track of at least some of its good identifiers in volatile variables, so these identifiers will become bad at the crash. But the sender doesn't know about the crash, so it will go on using the bad identifiers and thus send messages that the receiver will reject. Different protocols use different methods to keep the sender and the receiver more or less in sync about what identifiers to use.

In practice reliable messages are most often implemented in the form of 'connections'. The idea is that a connection is 'established', any amount of information is sent on the connection, and then the connection is 'closed'. You can think of this as the sending of a single large message, or as sending the first message using one of the protocols we discuss, and then sending later messages with increasing sequence numbers. Usually connections are full-duplex, so that either end can send independently, and it is often cheaper to establish both directions at the same time. We ignore all these complications in order to concentrate on the essential logic of the protocols.

What we mean by a crash is not simply a failure and restart of a node. In practice, protocols for reliable messages have limits, called 'timeouts', on the length of time for which they will wait to deliver a message or get an ack. We model the expiration of a timeout as a crash: the protocol abandons its normal operation and reports failure, even though in general it's possible that the message in fact has been or will be delivered.

We begin by writing a careful specification S for reliable messages. Then we present a 'lower-level' spec D in which the non-determinism associated with losing messages when there is a crash is moved to a place that is more convenient for implementations. We explain why D implements S but don't give a proof, since that requires techniques beyond the scope of this chapter. With this groundwork, we present a generic protocol G and a proof that it implements D. Then we describe two protocols that are used in practice, the handshake protocol H and the clock-based protocol C, and show how both implement G. Finally, we explain how to modify our protocols to work with finite sets of message identifiers, and summarize our results.

The goals of this chapter are to:

- Give a simple, clear, and precise specification of reliable message delivery in the presence of crashes.
- Explain the standard handshake protocol for reliable messages that is used in TCP, ISO TP4, and many other widespread communication systems, as well as a newer clock-based protocol.
- Show that both protocols can be best understood as special cases of a simpler, more general protocol for using identifiers to tag messages and acknowledgements for reliable delivery.
- Use the method of abstraction functions and invariants to help in understanding these three subtle concurrent and fault-tolerant algorithms, and in the process present all the hard parts of correctness proofs for all of them.
- Take advantage of the generic protocol to simplify the analysis and the arguments.





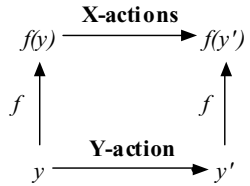
### *1.1 Methods*

We use the definition of 'implements' and the abstraction function proof method explained in Chapter 3. Here is a brief summary of this material.

Suppose that X and Y are state machines with named transitions called *actions*; think of X as a specification and Y as an implementation. We partition the actions of X and Y into *external* and *internal* actions. A *behavior* of a machine M is a sequence of actions that M can take starting in an initial state, and an *external behavior* of M is the subsequence of a behavior that contains only the external actions. We say Y *implements* X iff every external behavior of Y is an external behavior of X.[1] This expresses the idea that what it means for Y to implement X is that from the outside you don't see Y doing anything that X couldn't do.

The set of all external behaviors is a rather complicated object and difficult to reason about. Fortunately, there is a general method for proving that Y implements X without reasoning explicitly about behaviors in each case. It works as follows. First, define an *abstraction function f* from the state of Y to the state of X. Then show that Y *simulates* X:

1. *f* maps an initial state of Y to an initial state of X.
2. For each Y-action and each reachable state *y* there is a sequence of X-actions (perhaps empty) that is the same externally, such that the following diagram commutes.




A sequence of X-actions is the same externally as a Y-action if they are the same after all internal actions are discarded. So if the Y-action is internal, all the X-actions must be internal (perhaps none at all). If the Y-action is external, all the X-actions must be internal except one, which must be the same as the Y-action.

A straightforward induction shows that Y implements X: For any Y-behavior we can construct an X-behavior that is the same externally, by using (2) to map each Y-action into a sequence of X-actions that is the same externally. Then the sequence of X-actions will be the same externally as the original sequence of Y-actions.

In order to prove that Y simulates X we usually need to know what the reachable states of Y are, because it won't be true that every action of Y from an arbitrary state of Y simulates a sequence of X-actions; in fact, the abstraction function might not even be defined on an arbitrary state of Y. The most convenient way to characterize the reachable states of Y is by an *invariant*,

[1] Actually this definition only deals with the implementation of *safety* properties. Roughly speaking, a safety property is an assertion that nothing bad happens; it is a generalization of the notion of partial correctness for sequential programs. A system that does nothing implements any safety property. Specifications may also include *liveness* properties, which roughly assert that something good eventually happens; these generalize the notion of termination for sequential programs. A full treatment of liveness is beyond the scope of this chapter, but we do explain informally why the protocols make progress.

a predicate that is true of every reachable state. Often it's helpful to write the invariant as a conjunction, and to call each conjunct an invariant. It's common to need a stronger invariant than the simulation requires; the extra strength is a stronger induction hypothesis that makes it possible to establish what the simulation does require.

So the structure of a proof goes like this:

- Establish invariants to characterize the reachable states, by showing that each action maintains the invariants.
- Define an abstraction function.
- Establish the simulation, by showing that each Y-action simulates a sequence of X-actions that is the same externally.

This method works only with actions and does not require any reasoning about behaviors. Furthermore, it deals with each action independently. Only the invariants connect the actions. So if we change (or add) an action of Y, we only need to verify that the new action maintains the invariants and simulates a sequence of X-actions that is the same externally. We exploit this remarkable fact in Section 9 to extend our protocols so that they use finite, rather than infinite, sets of identifiers.

In what follows we give abstraction functions and invariants for each protocol. The actual proofs that the invariants hold and that each Y-action simulates a suitable sequence of X-actions are routine, so we give proofs only for a few sample actions.

### *1.2 Types and notation*

We use a type *M* for the messages being delivered. We assume nothing about *M*.

All the protocols except S and D use a type *I* of identifiers for messages. In general we assume only that *I*s can be compared for equality; C assumes a total ordering. If $x$ is a multiset whose elements have a first *I* component, we write ids($x$) for the multiset of *I*s that appear first in the elements of $x$.

We write $\langle\cdots\rangle$ for a sequence with the indicated elements and + for concatenation of sequences. We view a sequence as a multiset in the obvious way. We write $x = (y, *)$ to mean that $x$ is a pair whose first component is $y$ and whose second component can be anything, and similarly for $x = (*, y)$.

We define an action by giving its name, a *guard* that must be true for the action to occur, and an *effect* described by a set of assignments to state variables. We encode parameters by defining a whole family of actions with related names; for instance, *get*(*m*) is a different action for each possible *m*. Actions are atomic; each action completes before the next one is started.

To express concurrency we introduce more actions. Some of these actions may be internal, that is, they may not involve any interaction with the client of the protocol. Internal actions usually make the state machine non-deterministic, since they can happen whenever their guards are satisfied, not just when there is an interaction with the environment. We mark external actions with *s, two for an input action and one for an output action. Actions without *s are internal.

It's convenient to present the sender actions on the left and the receiver actions on the right. Some actions are not so easy to categorize, and we usually put them on the left.





## 2 The specification S

The specification S for reliable messages is a slight extension of the spec for a FIFO queue. Figure 1 shows the external actions and some examples of its transitions. The basic state of S is the FIFO queue $q$ of messages, with *put*(*m*) and *get*(*m*) actions. In addition, the *status* variable records whether the most recently sent message has been delivered. The sender can use *getAck*(*a*) to get this information; after that it may be forgotten by setting *status* to *lost*, so that the sender doesn't have to remember it forever. Both sender and receiver can crash and recover. In the absence of crashes, every message put is delivered by *get* in the same order and is positively acknowledged. If there is a crash, any message still in the queue may be lost at any time between the crash and the recovery, and its ack may be lost as well.

The *getAck*(*a*) action reports on the message most recently put, as follows. If there has been no crash since it was put there are two possibilities:

- the message is still in $q$ and *getAck* cannot occur;
- the message was delivered by *get*(*m*) and *getAck*(*OK*) occurs.

If there have been crashes, there are two additional possibilities:

- the message was lost and *getAck*(*lost*) occurs;
- the message was delivered or is still in $q$ but *getAck*(*lost*) occurs anyway.

The ack makes the most sense when the sender alternates *put*(*m*) and *getAck*(*a*) actions. Note that what is being acknowledged is delivery of the message to the client, not its receipt by some part of the implementation, so this is an end-to-end ack. In other words, the *get* should be thought of as including client processing of the message, and the ack might include some result returned by the client such as the result of a remote procedure call. This could be expressed precisely by adding an *ack* action for the client. We won't do that because it would clutter up the presentation without improving our understanding of how reliable messages work.

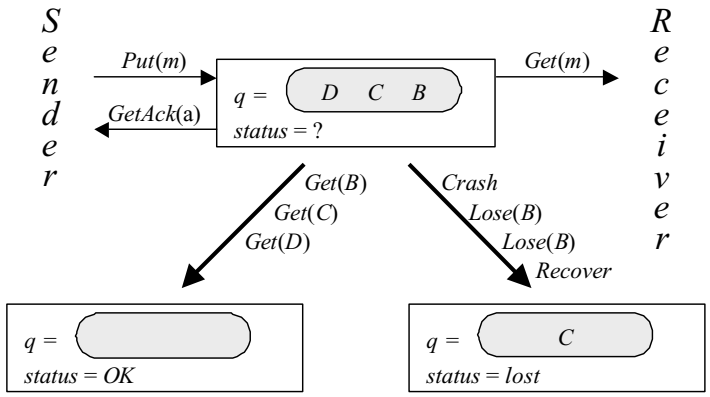



*Figure 1.* Some states and transitions for S

| Sender | | | Receiver | | |
|---|---|---|---|---|---|
| **Name** | **Guard** | **Effect** | **Name** | **Guard** | **Effect** |
| ***put*(*m*) | $rec_s$ = *false* | append *m* to *q*, *status* := ? | **get*(*m*) | $rec_r$ = *false*, *m* is first on *q* | remove head of *q*, if *q* = *empty* and *status* = ? then *status* := *OK* |
| **getAck*(*a*) | $rec_s$ = *false*, *status* = *a* | optionally *status* := *lost* | | | |
| **$crash_s$ | | $rec_s$ := *true* | **$crash_r$ | | $rec_r$ := *true* |
| *$recover_s$ | $rec_s$ | $rec_s$ := *false* | *$recover_r$ | $rec_r$ | $rec_r$ := *false* |
| *lose* | $rec_s$ or $rec_r$ | delete some element from *q*; if it's the last then *status* := *lost*, or *status* := *lost* | | | |

| | | |
|---|---|---|
| *q* | : sequence[*M*] | := ⟨ ⟩ |
| *status* | : *Status* | := *lost* |
| $rec_s$ | : Boolean | := *false* (*rec* is short for 'recovering') |
| $rec_r$ | : Boolean | := *false* |

*Table 1*. State and actions of S

To define S we introduce the types *A* (for acknowledgement) with values in {*OK, lost*} and *Status* with values in {*OK, lost,* ?}. Table 1 gives the state and actions of S. Note that it says nothing about channels; they are part of the implementation and have nothing to do with the spec.

Why do we have both *crash* and *recover* actions, as opposed to just a *crash* action? A spec that only allows messages to be lost at the time of a *crash* is not implemented by a protocol like C in which the sender accepts a message with *put* and sends it without verifying that the receiver is running normally. In this case the message is lost even though it wasn't in the system at the time of the crash. This is why we have a separate $recover_r$ action that allows the receiver to declare the point after a crash when messages are again guaranteed not to be lost. There seems to be no need for a $recover_s$ action, but we have one for symmetry.

A spec which only allows messages to be lost at the time of a *recover* is not implemented by any protocol that can have two messages in the network at the same time, because after a $crash_s$ and before the following $recover_s$ it's possible for the second message in the network to be delivered, which means that the first one must be lost to preserve the FIFO property.

The simplest spec that covers both these cases can lose a message at any time between a *crash* and its following *recover*, and we have adopted this alternative.

## 3 The delayed-decision specification D

Next we introduce an implementation of S, called the delayed-decision specification D, that is more non-deterministic about when messages are lost. The reason for D is to simplify the proofs of the protocols: with more freedom in D, it's easier to prove that a protocol simulates D than to prove that it simulates S. A typical protocol transmits messages from the sender to the receiver over some kind of channel that can lose messages; to compensate for these losses, the sender retransmits. If the sender crashes with a message in the channel it stops retransmitting, but whether the receiver gets the message depends on whether the channel loses it. This may not be





decided until after the sender has recovered. So the protocol doesn't decide whether the message is lost until after the sender has recovered. D has this freedom, but S does not.

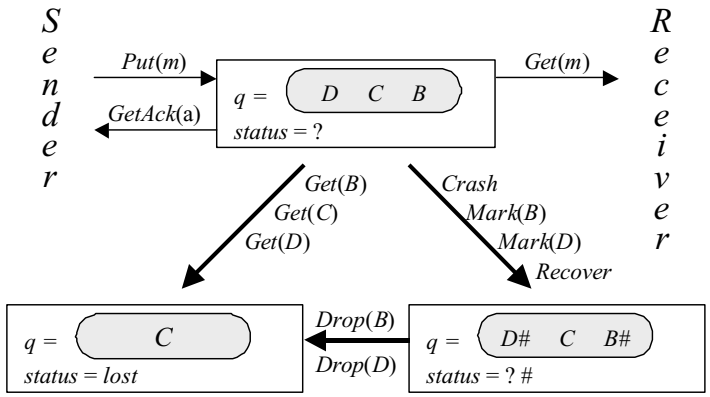



*Figure 2*. Some states and transitions of D

| Sender | | | Receiver | | |
|---|---|---|---|---|---|
| **Name** | **Guard** | **Effect** | **Name** | **Guard** | **Effect** |
| ***put*(*m*) | $rec_S$ = *false* | append (*m*, +) to *q*, *status* := (?, +) | **get*(*m*) | $rec_R$ = *false*, (*m*, *) first on *q* | remove head of *q*, if *q* = *empty* and *status* = (?, *x*) then *status*:=(*OK*,*x*) |
| **getAck*(*a*) | $rec_S$ = *false*, *status* = (*a*, *) | *status* := (*a*, +) or *status* := (*lost*, +) | | | |
| **$crash_S$ | | $rec_S$ := *true* | **$crash_R$ | | $rec_R$ := *true* |
| *$recover_S$ | $rec_S$ | $rec_S$ := *false* | *$recover_R$ | $rec_R$ | $rec_R$ := *false* |
| *mark* | $rec_S$ or $rec_R$ | for some element of *q* or for *status*, *mark* := # | *unmark* | | for some element of *q* or for *status*, *mark* := + |

*drop* delete an element of *q* with *mark* = #; if it was the last element, *status* := (*lost,* +) or if *status* = (*, #), *status* := (*lost,* +)

| | | |
|---|---|---|
| *q* | : sequence[(*M*, *Mark*)] | := ⟨ ⟩ |
| *status* | : (*Status, Mark*) | := (*lost*, +) |
| $rec_S$ | : Boolean | := *false* |
| $rec_R$ | : Boolean | := *false* |

*Table 2*. State and actions of D

D is the same as S except that the decisions about which messages to lose at recovery, and whether to lose the ack, are made by asynchronous *drop* actions that can occur after recovery. Each message in $q$, as well as the *status* variable, is augmented by an extra component of type *Mark* which is normally + but may become # between crash and recovery because of a *mark* action. At any time an *unmark* action can change a mark from # back to +, a message marked # can be lost by *drop*, or a *status* marked # can be set to *lost* by *drop*. Figure 2 gives an example of the transitions of D; the + marks are omitted.

To define D we introduce the type *Mark* that has values in the set {+, #}. Table 2 gives the state and actions of D.

### *3.1 Proof that D implements S*

We do not give this proof, since to do it using abstraction functions we would have to introduce 'prophecy variables', also known as 'multi-valued mappings' or 'backward simulations' (Abadi and Lamport [1991], Lynch and Vaandrager [1993]). If you work out some examples, however, you will probably see why the two specs S and D have the same external behavior.

## 4 Channels

All our protocols use the same *channel* abstraction to transfer information between the sender and the receiver. We use the name 'packet' for the messages sent over a channel, to distinguish them from reliable messages. A channel can freely drop and reorder packets, and it can duplicate a packet any finite number of times when it's sent;[2] the only thing it isn't allowed to do is deliver a packet that wasn't sent. The reason for using such a weak specification is to ensure that the reliable message protocol will work over any bit-moving mechanism that happens to be available. With a stronger channel spec, for instance one that doesn't reorder packets, it's possible to have somewhat simpler or more efficient implementations.

There are two channels *sr* and *rs*, one from sender to receiver and one from receiver to sender, each a multiset of packets initially empty. The nature of a packet varies from one protocol to another. Table 3 gives the channel actions.

Protocols interact with the channels through the external actions *send*(...) and *rcv*(...) which have the same names in the channel and in the protocol. One of these actions occurs if both its pre-conditions are true, and the effect is both the effects. This always makes sense because the states are disjoint.

| Name | Guard | Effect | Name | Guard | Effect |
|---|---|---|---|---|---|
| $**send_{sr}(p)$ | | add some number of copies of $p$ to $sr$ | $**send_{rs}(p)$ | | add some number of copies of $p$ to $rs$ |
| $*rcv_{sr}(p)$ | $p \in sr$ | remove one $p$ from $sr$ | $rcv_{rs}(p)$ | $p \in rs$ | remove one $p$ from $rs$ |
| $lose_{sr}(p)$ | $p \in sr$ | remove one $p$ from $sr$ | $lose_{rs}(p)$ | $p \in rs$ | remove one $p$ from $rs$ |

*Table 3*. Actions of the channels

[2] You might think it would be more natural and closer to the actual implementation of a channel to allow a packet already in the channel to be duplicated. Unfortunately, if a packet can be duplicated any number of times it's possible that a protocol like H (see section 8) will not make any progress.





## 5 The generic protocol G

The generic protocol G generalizes two practical protocols described later, H and C; in other words, both of them implement G. This protocol can't be implemented directly because it has some 'magic' actions that use state from both sender and receiver. But both real protocols implement these actions, each in its own way.

The basic idea is derived from the simplest possible distributed implementation of S, which we call the stable protocol SB. In SB all the state is stable (that is, nothing is lost when there is a crash), and each end keeps a set $g_s$ or $g_r$ of good identifiers, that is, identifiers that have not yet been used. Initially $g_s \subseteq g_r$, and the protocol maintains this as an invariant. To send a message the sender chooses a good identifier $i$ from $g_s$, attaches $i$ to the message, moves $i$ from $g_s$ to a $last_s$ variable, and repeatedly sends the message. When the receiver gets a message with a good identifier it accepts the message, moves the identifier from $g_r$ to a $last_r$ variable, and returns an ack packet for the identifier after the message has been delivered by *get*. When the receiver gets a message with an identifier that isn't good, it returns a positive ack if the identifier equals $last_r$ and the message has been delivered. The sender waits to receive an ack for $last_s$ before doing *getAck*(*OK*). There are never any negative acks, since nothing is ever lost.

This protocol satisfies the requirements of S; indeed, it does better since it never loses anything.

1. It provides at-most-once delivery because the sender never uses the same identifier for more than one message, and the receiver accepts an identifier and its message only once.
2. It provides FIFO ordering because at most one message is in transit at a time.
3. It delivers all the messages because the sender's good set is a subset of the receiver's.
4. It acks every message because the sender keeps retransmitting until it gets the ack.

The SB protocol is widely used in practice, under names that resemble 'queuing system'. It isn't used to establish connections because the cost of a stable storage write for each message is too great.

In G we have the same structure of good sets and *last* variables. However, they are not stable in G because we have to update them for every message, and we don't want to do a stable write for every message. Instead, there are operations to grow and shrink the good sets; these operations maintain the invariant $g_s \subseteq g_r$ as long as there is no receiver crash. When there is a crash, messages and acks can be lost, but S and D allow this. Figure 3 shows the state and some possible transitions of G in simplified form. The names in outline font are state variables of D, and the corresponding values are the values of the abstraction function in that state.

Figure 4 shows the state of G, the most important actions, and the S-shaped flow of information. The *new* variables in the figure are the complement of the *used* variables in the code. The heavy lines show the flow of a new identifier from the receiver to the sender, back to the receiver along with the message, and then back again to the sender along with the acknowledgement.

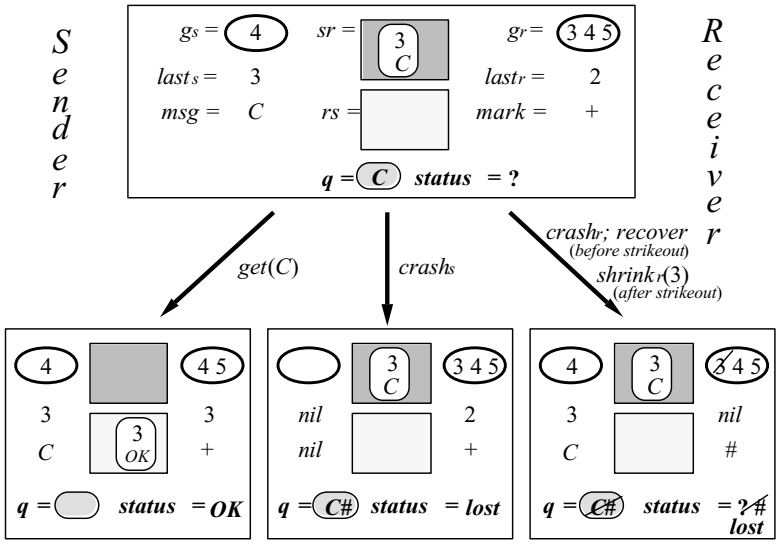



*Figure 3*. Some states and transitions of G

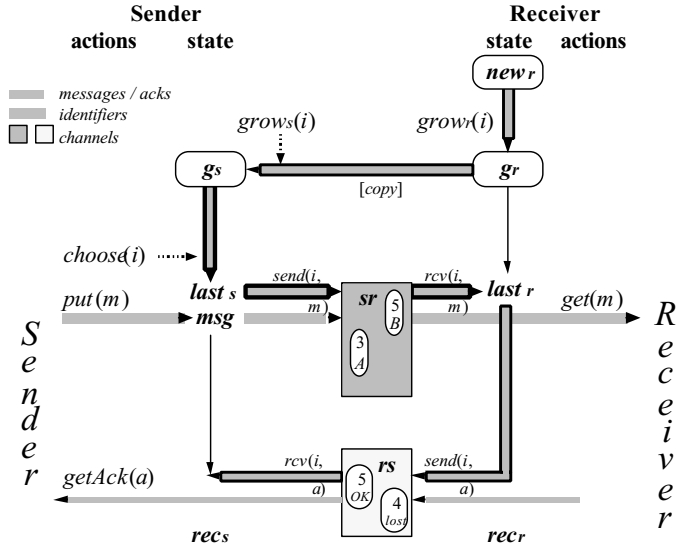



*Figure 4*. State, main actions, and information flow of G





G also satisfies the requirements of S, but not quite in the same way as SB.

1. At-most-once delivery is the same as in SB.
2. The sender may send a message after a crash without checking that a previous outstanding message has actually been received. Thus more than one message can be in transit at a time, so there must be a total ordering on the identifiers in transit to maintain FIFO ordering of the messages. In G this ordering is defined by the order in which the sender chooses identifiers.
3. Complete delivery is the same as in SB as long as there is no receiver crash. When the receiver crashes $g_s \subseteq g_r$ may cease to hold, with the effect that messages that the sender handles during the receiver crash may be assigned identifiers that are not in $g_r$ and hence may be lost. The protocol ensures that this can't happen to messages whose *put* happens after the receiver has recovered. When the sender crashes, it stops retransmitting the current message, which may be lost as a result.
4. As in SB, the sender keeps retransmitting until it gets an ack, but since messages can be lost, there must be negative as well as positive acks. When the receiver sees a message with an identifier that is not in $g_r$ and not equal to $last_r$ it optionally returns a negative ack. There is no point in doing this for a message with $i < last_r$ because the sender only cares about the ack for $last_s$, and the protocol maintains the invariant $last_r \le last_s$. If $i > last_r$, however, the receiver must sometimes send a negative ack in response so that the sender can find out that the message may have been lost.

G is organized into a set of implementable actions that also appear, with very minor variations, in both H and C, plus the magic *grow*, *shrink*, and *cleanup* actions that are simulated quite differently in H and in C.

When there are no crashes, the sender and receiver each go through a cycle of modes, the sender perhaps one mode ahead. In one cycle one message is sent and acknowledged. For the sender, the modes are *idle*, [*needI*], *send*; for the receiver, they are *idle* and *ack*. An agent that is not idle is busy. The bracketed mode is 'internal': it's possible to advance to the next mode without receiving another message. The modes are not explicit state variables, but instead are derived from the values of the *msg* and *last* variables, as follows:

$mode_s = idle$ iff $msg = nil$ $\qquad$ $mode_r = idle$ iff $last_r = nil$
$mode_s = needI$ iff $msg \ne nil$ and $last_s = nil$
$mode_s = send$ iff $msg \ne nil$ and $last_s \ne nil$ $\qquad$ $mode_r = ack$ iff $last_r \ne nil$

To define G we introduce the types:

$I$, an infinite set of identifiers.

$P$ (packet), a pair ($I$, $M$ or $A$).

The sender sends ($I$, $M$) packets to the receiver, which sends ($I$, $A$) packets back. The $I$ is there to identify the packet for the destination. We define a partial order on $I$ by the rule that $i < i'$ iff $i$ precedes $i'$ in the sequence $used_s$.

The G we give is a somewhat simplified version, because the actions are not as atomic as they should be. In particular, some actions have two external interactions, sometimes one with a channel and one with the client, sometimes two with channels. However, the simplified version differs from one with the proper atomicity only in unimportant details. The appendix gives a version of G with all the fussy details in place. We don't give these details for the C and H

protocols that follow, but content ourselves with the simplified versions in order to emphasize the important features of the protocols.

Figure 5 is a more detailed version of Figure 4, which shows all the actions and the flow of information between the sender and the receiver. State variables are given in bold, and the black guards on the transitions give the pre-conditions. The *mark* variable can be # when the receiver has recovered since a message was put; it reflects the fact that the message may be dropped.

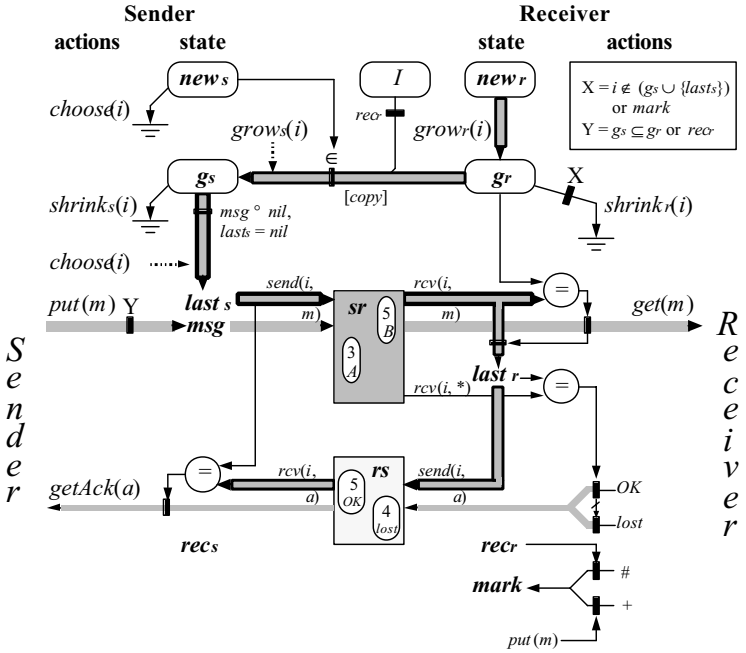



*Figure 5*. Details of actions and information flow in G

Table 4 gives the state and actions of G. The magic parts, that is, those that touch non-local state, are boxed. The conjunct $\neg\, rec_s$ has been omitted from the guards of all the sender actions except $recover_s$, and likewise for $\neg\, rec_r$ and the receiver actions.

In addition to meeting the spec S, this protocol has some other important properties:

- It makes progress: regardless of prior crashes, provided both ends stay up and the channels don't always lose messages, then if there's a message to send it is eventually sent, and otherwise both parties eventually become idle, the sender because it gets an ack, the receiver because eventually *cleanup* makes *mode = idle*. Progress depends on doing enough *grow* actions, and in particular on completing the sequence $grow_r(i)$, $grow_s(i)$, *choose*(*i*).





| Name | Guard | Effect | Name | Guard | Effect |
|---|---|---|---|---|---|
| $**put(m)$ | $msg = nil$, $\boxed{g_s \subseteq g_r \text{ or } rec_r}$ | $msg := m$, $\boxed{mark := +}$ | | | |
| $choose(i)$ | $msg \neq nil$, $last_s = nil$, $i \in g_s$ | $g_s -:= \{j \mid j \leq i\}$, $last_s := i$, $used_s +:= \langle i \rangle$ | $*get(m)$ | exists $i$ such that $rcv_{sr}(i,m)$, $i \in g_r$ | $g_r -:= \{j \mid j \leq i\}$, $last_r := i$, $send_{rs}(i, OK)$ |
| $send$ | $last_s \neq nil$ | $send_{sr}(last_s, msg)$ | | | |
| $*getAck(a)$ | $rcv_{rs}(last_s, a)$ | $last_s := nil$, $msg := nil$ | $sendAck$ | exists $i$ such that $rcv_{sr}(i, *)$, $i \notin g_r$ | optionally $send_{rs}$ ($i$, if $i = last_r$ then $OK$ else $lost$) |
| $**crash_s$ | | $rec_s := true$ | $**crash_r$ | | $rec_r := true$ |
| $*recover_s$ | $rec_s$ | $last_s := nil$, $msg := nil$, $rec_s := false$ | $*recover_r$ | $rec_r$, $\boxed{used_r \supseteq g_s \cup used_s}$ | $last_r := nil$, $mark := \#$, $rec_r := false$ |
| $shrink_s(i)$ | | $g_s -:= \{i\}$ | $shrink_r(i)$ | $\boxed{i \notin g_s, i \neq last_s}$ or $mark = \#$ | $g_r -:= \{i\}$ |
| $grow_s(i)$ | $i \notin used_s$, $\boxed{i \in g_r \text{ or } rec_r}$ | $g_s +:= \{i\}$ | $grow_r(i)$ | $i \notin used_r$ | $g_r +:= \{i\}$, $used_r +:= \{i\}$ |
| $grow\text{-}used_s(i)$ | $i \notin used_s \cup g_s$, $\boxed{i \in used_r \text{ or } rec_r}$ | $used_s +:= \{i\}$ | $cleanup$ | $\boxed{last_r \neq last_s}$ | $last_r := nil$ |
| | | | $unmark$ | $\boxed{g_s \subseteq g_r, last_s \in g_r \cup \{last_r, nil\}}$ | $mark := +$ |

| | | | | | |
|---|---|---|---|---|---|
| $used_s$ | : sequence[*I*] | := ⟨ ⟩ (stable) | $used_r$ | : set[*I*] | := { } (stable) |
| $g_s$ | : set[*I*] | := { } | $g_r$ | : set[*I*] | := { } |
| $last_s$ | : *I* or *nil* | := *nil* | $last_r$ | : *I* or *nil* | := *nil* |
| $msg$ | : *M* or *nil* | := *nil* | $mark$ | : *Mark* | := # |
| $rec_s$ | : Boolean | := *false* | $rec_r$ | : Boolean | := *false* |

*Table 4*. State and actions of G

- It's not necessary to do a stable storage operation for each message. Instead, the cost of a stable storage operation can be amortized over as many messages as you like. G has only two stable variables: $used_s$ and $used_r$. Different implementations of G handle $used_s$ differently. To reduce the number of stable updates to $used_r$, refine G to divide $used_r$ into the union of a stable $used_r$-*s* and a volatile $used_r$-*v*. Move a set of *I*s from $used_r$-*s* to $used_r$-*v* with a single stable update. The $used_r$-*v* becomes empty in $recover_r$; simulate this with $grow_r(i)$ followed immediately by $shrink_r(i)$ for every $i$ in $used_r$-*v*.
- The only state required for an idle agent is the stable variable *used*. All the other (volatile) state is the same at the end of a message transmission as at the beginning. The sender forgets its state in *getAck*, the receiver in *cleanup*, and both in *recover*. The *shrink* actions

make it possible for both parties to forget the good sets. This is important because agents may need to communicate with many other agents between crashes, and it isn't practical to require that an agent maintain some state for everyone with whom it has ever communicated.

- An idle sender doesn't send any packets. An idle receiver doesn't send any packets unless it receives one, because it sends an acknowledgement only in response to a packet. This is important because the channel resources shouldn't be wasted.

We have constructed G with as much non-determinism as possible in order to make it easy to prove that different practical protocols implement G. We could have simplified it, for instance by eliminating *unmark*, but then it would be more difficult to construct an abstraction function from some other protocol to G, since the abstraction function would have to account for the fact that after a $recover_r$ the *mark* variable is # until the next *put*. With *unmark*, an implementation of G is free to set *mark* back to + whenever the guard is true.

### *5.1 Abstraction function to D*

The abstraction function is an essential tool for proving that the protocol implements the spec. But it is also an important aid to understanding what is going on. By studying what happens to the value of the abstraction function during each action of G, we can learn what the actions are doing and why they work.

*Definitions*

| | | | |
|---|---|---|---|
| *cur-q* | = | $\{(msg, mark)\}$ | if $msg \neq nil$ and ($last_s = nil$ or $last_s \in g_r$) |
| | | $\{\ \}$ | otherwise |
| $inflight_{sr}$ | = | $\{(i, m) \in \text{ids}(sr) \mid i \in g_r$ and $i \neq last_s\}$, sorted by $i$ to make a sequence | |
| *old-q* | = | the sequence of (*M, Mark*)'s gotten by turning each ($i, m$) in $inflight_{sr}$ into ($m$, #) | |
| $inflight_{rs}$ | = | $\{last_s\}$ | if ($last_s$, *OK*) $\in$ *rs* and $last_s \neq last_r$ |
| | | $\{\ \}$ | otherwise. |

Note that the *inflight*s exclude elements that might still be retransmitted as well as elements that are not of interest to the destination. This is so the abstraction function can pair them with the # mark.

*Abstraction function*

| | | | |
|---|---|---|---|
| *q* | *old-q* + *cur-q* | | |
| *status* | (?, *mark*) | if *cur-q* $\neq \{\ \}$ | (a) |
| | (*OK*, +) | if $mode_s = send$ and $last_s = last_r$ | (b) |
| | (*OK*, #) | if $mode_s = send$ and $last_s \in inflight_{rs}$ | (c) |
| | (*lost*, +) | if $mode_s = send$ and $last_s \notin (g_r \cup \{last_r\} \cup inflight_{rs})$ | (d) |
| | (*lost*, +) | if $mode_s = idle$ | (e) |
| $rec_{s/r}$ | $rec_{s/r}$ | | |

The cases of *status* are exhaustive. Note that we do *not* want (*msg*, +) in *q* if $mode_s = send$ and $lasts_s \notin g_r$, because in this case *msg* has been delivered or lost.





We see that G simulates the *q* of D using *old-q* + *cur-q*, and that *old-q* is the leftover messages in the channel that are still good but haven't been delivered, while *cur-q* is the message the sender is currently working on, as long as its identifier is not yet assigned or still good. Similarly, *status* has a different value for each step in the delivery process: still sending the message (a), normal ack (b), ack after a receiver crash (c), lost ack (d), or delivered ack (e).

### *5.2 Invariants*

Like the abstraction function, the invariants are both essential to the proof and an important aid to understanding. They express a great deal of information about how the protocol is supposed to work. It's especially instructive to see how the parts of the state that have to do with crashes ($rec_{s/r}$ and *mark*) affect them.

The first few invariants establish some simple facts about the *used* sets and their relation to other variables. (G2) reflects that fact that identifiers move from $g_s$ to $used_s$ one by one, (G3) the fact that unless the receiver is recovering, identifiers must enter $used_r$ before they can appear anywhere else (G4) the fact that they must enter $used_s$ before they can appear in *last* variables or channels.

If $msg = nil$ then $last_s = nil$ (G1)

$g_s \cap used_s = \{\ \}$ (G2a)

All elements of $used_s$ are distinct. (G2b)

$used_r \supseteq g_r$ (G3a)

If $\neg\, rec_r$ then $used_r \supseteq g_s \cup used_s$ (G3b)

$used_s \supseteq \{last_s, last_r\} - \{nil\} \cup \text{ids}(sr) \cup \text{ids}(rs)$ (G4)

The next invariants deal with the flow of identifiers during delivery. (G5) says that each identifier tags at most one message. (G6) says that if all is well, $g_s$ and $last_s$ are such that a message will be delivered and acknowledged properly. (G7) says that an identifier for a message being acknowledged can't be good.

$\{m \mid (i = last_s$ and $m = msg)$ or $(i, m) \in sr\}$ has 0 or 1 elements (G5)

If $mark$ = + and $\neg\, rec_s$ and $\neg\, rec_r$ then $g_s \subseteq g_r$ and $last_s \in g_r \cup \{last_r, nil\}$ (G6)

$g_r \cap (\{last_r\} \cup \text{ids}(rs)) = \{\ \}$ (G7)

Finally, some facts about the identifier $last_s$ for the message the sender is trying to deliver. It comes later in the identifier ordering than any other identifier in *sr* (G8a). If it's been delivered and is getting a positive ack, then neither it nor any other identifier in *sr* is in $g_r$, but they are all in $used_r$ (G8b). If it's getting a negative ack then it won't get a later positive one (G8c).

If $last_s \neq nil$ then

$\text{ids}(sr) \leq last_s$ (G8a)

and if $last_s = last_r$ or $(last_s, OK) \in rs$ then $(\{last_s\} \cup \text{ids}(sr)) \cap g_r = \{\ \}$ and $(\{last_s\} \cup \text{ids}(sr)) \subseteq used_r$ (G8b)

and if $(last_s, lost) \in \text{ids}(rs)$ then $last_s \neq last_r$ (G8c)

*5.3 Proof that G implements D*

This requires showing that every action of G simulates some sequence of actions of D which is the same externally. Since G has quite a few actions, the proof is somewhat tedious. A few examples give the flavor.

—$recover_s$: Mark *msg* and drop it unless it moves to *old-q*; mark and drop *status*.

—*get*(*m*): For the change to *q*, first drop everything in *old-q* less than *i*. Then *m* is first on *q* since either *i* is the smallest *I* in *old-q*, or $i = last_s$ and *old-q* is empty by (G8a). So D's *get*(*m*) does the rest of what G's does. Everything in *old-q* + *cur-q* that was ≤ *i* is gone, so the corresponding *M*'s are gone from *q* as required.

We do *status* by the abstraction function's cases on its old value. D says it should change to (*OK, x*) iff *q* becomes empty and it was (?, *x*). In cases (c-e) *status* isn't (?, *x*) and it doesn't change. In case (b) the guard $i \in g_r$ of *get* is false by (G8b). In case (a) either $i = last_s$ or not. If not, then *cur-q* remains unchanged by (G8a), so *status* does also and *q* remains non-empty. If so, then *cur-q* and *q* both become empty and *status* changes to case (b). Simulate this by umarking *status* if necessary; then D's *get*(*m*) does the rest.

—*getAck*(*a*): The *q* is unchanged because $last_s = i \in$ ids(*rs*), so $last_s \notin g_r$ by (G7) and hence *cur-q* is empty, so changing *msg* to *nil* keeps it empty. Because *old-q* doesn't change, *q* doesn't either. We end up with *status* = (*lost,* +) according to case (e), as required by D. Finally, we must show that *a* agrees with the old value of *status*. We do this by the cases of *status* as we did for *get*:

(a) Impossible, because it requires $last_s \in g_r$, but we know $last_s \in$ ids(*rs*), which excludes $last_s \in g_r$ by (G7).

(b) In this case $last_s = last_r$, so (G8c) ensures $a \neq lost$, so $a = OK$.

(c) If $a = OK$ we are fine. If $a = lost$ drop *status* first.

(d) Since $last_s \notin inflight_{rs}$, only ($last_s$, *lost*) ∈ *rs* is possible, so $a = lost$.

(e) Impossible because $last_s \neq nil$.

—$shrink_r$: If $rec_r$ then *msg* may be lost from *q*; simulate this by marking and dropping it, and likewise for *status*. If *mark* = # then *msg* may be lost from *q*, but it is marked, so simulate this by dropping it, and likewise for *status*. Otherwise the precondition ensures that $last_s \in g_r$ doesn't change, so *cur-q* and *status* don't. $Inflight_{sr}$, and hence *old-q*, can lose an element; simulate this by dropping the corresponding element of *q*, which is possible since it is marked #.

## 6 How C and H implement G

We now proceed to give two practical protocols, the clock-based protocol C and the handshake protocol H. Each implements G, but they handle the good sets quite differently.

In C the good sets are maintained using time; to make this possible the sender and receiver clocks must be roughly synchronized, and there must be an upper bound on the time required to transmit a packet. The sender's current time $time_s$ is the only member of $g_s$; if the sender has already used $time_s$ then $g_s$ is empty. The receiver accepts any message with an identifier in the range ($time_r - 2\varepsilon - \delta$, $time_r + 2\varepsilon$), where ε is the maximum clock skew from real time and δ the maximum packet transmission time, as long as it hasn't already accepted a message with a later identifier.





In H the sender asks the receiver for a good identifier; the receiver's obligation is to keep the identifier good until it crashes or receives the message, or learns from the sender that the identifier will never be equal to $last_s$.

We begin by giving the abstraction functions from C and H to G, and a sketch of how each implements the magic actions of G, to help the reader in comparing the protocols. Careful study of these should make it clear exactly how each protocol implements G's magic actions in a properly distributed fashion.

Then for each protocol we give a figure that shows the flow of packets, followed by a formal description of the state and the actions. The portion of the figures that shows messages being sent and acks returned is exactly the same as the bottom half of Figure 4 for G; all three protocols handle messages and acks identically. They differ in how the sender obtains good identifiers, shown in the top of the figures, and in how the receiver cleans up its state. In the figures for C and H we show the abstraction function to G in outline font.

Note that G allows either good set to grow or shrink by any number of *I*s through repeated *grow* or *shrink* actions as long as the invariants $g_s \subseteq g_r$ and $last_s \in g_r \cup \{last_r\}$ are maintained in the absence of crashes. For C the *increase* actions simulate occurrences of several $grow_r$ and $shrink_r$ actions, one for each *i* in the set defined in the table. Likewise $rcv_{rs}(j_s, i)$ in H may simulate several $shrink_s$ actions.

*Abstraction functions to G*

| **G** | **C** | **H** |
|---|---|---|
| $used_s$ | $\{i \mid 0 \le i < time_s\} \cup \{sent\} - \{nil\}$ | $used_s$ (history) |
| $used_r$ | $\{i \mid 0 \le i < low\}$ | $used_r$ |
| $g_s$ | $\{time_s\} - \{sent\}$ | $\{i \mid (j_s, i) \in rs\}$ |
| $g_r$ | $\{i \mid low < i$ and $i < high\}$ | $\{i_r\} - \{nil\}$ |
| *mark* | # if $last_s \in g_r$ and *deadline* = *nil*<br>+ otherwise | # if $mode_s$ = *needI* and $g_s \not\subseteq g_r$<br>+ otherwise |
| *msg,* $last_{s/r}$, and $rec_{s/r}$ are the same in G, C, and H | | |
| *sr* | *sr* | the (*I, M*) messages in *sr* |
| *rs* | *rs* | the (*I, A*) messages in *rs* |

*Sketch of implementations*

| G | C | H |
|---|---|---|
| $grow_s(i)$ | *tick*(*i*) | $send_{rs}(j_s, i)$ |
| $shrink_s(i)$ | *tick*(*i'*), $i \in \{time_s\} - \{sent\}$ | $lose_{rs}(j_s, i)$ if the last copy is lost or $rcv_{rs}(j_s, i')$, for each $i \in g_s - \{i'\}$ |
| $grow_r(i)$ | *increase-high*(*i'*), for each $i \in \{i \mid high < i < i'\}$ | *mode* = *idle* and $rcv_{sr}$(needI, *) |
| $shrink_r(i)$ | *increase-low*(*i'*), for each $i \in \{i \mid low < i \le i'\}$ | $rcv_{sr}(i_r,$ done) |
| *cleanup* | *cleanup* | $rcv_{sr}(last_r,$ done) |

## 7 The clock-based protocol C

This protocol is due to Liskov, Shrira, and Wroclawski [1991]. Figure 6 shows the state and the flow of information. Compare it with Figure 4 for G, and note that there is no flow of new identifiers from receiver to sender. In C the passage of time supplies the sender with new identifiers, and is also allows the receiver to clean up its state.

The idea behind C is to use loosely synchronized clocks to provide the identifiers for messages. The sender uses its current *time* for the next identifier. The receiver keeps track of *low*, the biggest clock value for which it has accepted a message: bigger values than this are good. The receiver also keeps a stable bound *high* on the biggest value it will accept, chosen to be larger than the receiver's clock plus the maximum clock skew. After a crash the receiver sets *low* := *high*; this ensures that no messages are accepted twice.

The sender's clock advances, which ensures that it will get new identifiers and also ensures that it will eventually get past *low* and start sending messages that will be accepted after a receiver crash.

It's also possible for the receiver to advance *low* spontaneously (by *increase-low*) if it hasn't received a message for a long time, as long as *low* stays smaller than the current time – 2ε – δ, where ε is the maximum clock skew from real time and δ is the maximum packet transmission time. This is good because it gives the receiver a chance to run several copies of the protocol (one for each of several senders), and make the values of *low* the same for all the idle senders. Then the receiver only needs to keep track of a single *low* for all the idle senders, plus one for each active sender. Together with C's *cleanup* action this ensures that the receiver needs no storage for idle senders.

If the assumptions about clock skew and maximum packet transmission time are violated, C still provides at-most-once delivery, but it may lose messages (because *low* is advanced too soon or the sender's clock is later than *high*) or acknowledgements (because *cleanup* happens too soon).





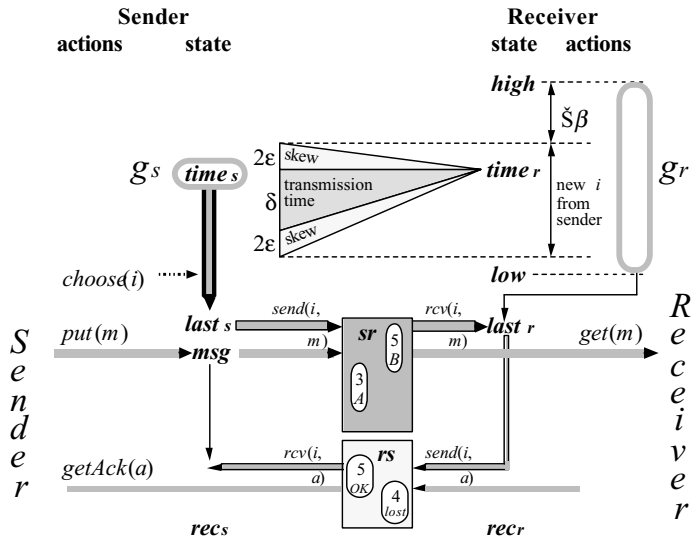



*Figure 6*. The flow of information in C

Modes, types, packets, and the pattern of messages are the same as in G, except that the *I* set has a total ordering. The *deadline* variable expresses the assumption about maximum packet delivery time: real time doesn't advance (by *progress*) past the deadline for delivering a packet. In a real implementation, of course, there will be some other properties of the channel from which the constraint imposed by *deadline* can be deduced. These are usually probabilistic; we deal with this by declaring a *crash* whenever the channel fails to meet its deadline.

Table 5 gives the state and actions of C. The conjunct ¬ $rec_s$ has been omitted from the guards of all the sender actions except $recover_s$, and likewise for ¬ $rec_r$ and the receiver actions.

Note that like G, this version of C sends an ack only in response to a message. This is unlike H, which has continuous transmission of the ack and pays the price of a *done* message to stop it. Another possibility is to make timing assumptions about *rs* and time out the ack; some assumptions are needed anyway to make *cleanup* possible. This would be less practical but more like H.

Note that $time_s$ and $time_r$ differ from real time (*now*) by at most ε, and hence $time_s$ and $time_r$ can differ from each other by as much as 2ε. Note also that the deadline is enforced by the *progress* action, which doesn't allow real time to advance past the deadline unless someone is recovering. Both $crash_s$ and $crash_r$ cancel the deadline.

*About the parameters of C*

The protocol is parameterized by three constants:

- $\delta$ = maximum time to deliver a packet
- $\beta$ = amount beyond $time_r + 2\varepsilon$ to increase *high*
- $\varepsilon$ = maximum of $|now - time_{r/s}|$

These parameters must satisfy two constraints:

- $\delta > \varepsilon$ so that $mode_s = send$ implies $last_s < deadline$.
- $\beta > 0$ so *increase-high* can be enabled. Aside from this constraint the choice of $\beta$ is just a tradeoff between the frequency of stable storage writes (at least one every $\beta$, so a bigger $\beta$ means fewer writes) and the delay imposed on $recover_r$ to ensure that messages put after $recover_r$ don't get dropped (as much as $4\varepsilon + \beta$, because *high* can be as big as $time_r + 2\varepsilon + \beta$ at the time of the crash because of (e), and $time_r - 2\varepsilon$ has to get past this via $tick_r$ before $recover_r$ can happen, so a bigger $\beta$ means a longer delay).

### *7.1 Invariants*

Mostly these are facts about the ordering of various time variables; a lot of $x \neq nil$ conjuncts have been omitted. Nothing being sent is later than $time_s$ (C1). Nothing being acknowledged is later than *low*, which is no later than *high*, which in turn is big enough (C2). Nothing being sent or acknowledged is later than $last_s$ (C3). The sender's time is later than *low*, hence good unless equal to *sent* (C4).

$last_s \le time_s$ (C1)

$last_r \le low \le high$ (C2a)

ids($rs$) $\le low$ (C2b)

If $\neg\, rec_r$ then $time_r + 2\varepsilon \le high$ (C2c)

ids($sr$) $\le last_s$ (C3a)

$last_r \le last_s$ (C3b)

$\{i \mid (i, OK) \in rs\} \le last_s$ (C3c)

$low \le time_s$ (C4)

$low < time_s$ if $last_s \neq time_s$

If a message is being sent but hasn't been delivered, and there hasn't been a crash, then *deadline* gives the deadline for delivering the packet containing the message (based on the maximum time for a packet that is being retransmitted to get through *sr*), and it isn't too late for it to be accepted (C5).

If $deadline \neq nil$ then

$now < last_s + \varepsilon + \delta$ (C5a)

$low < last_s$ (C5b)

An identifier getting a positive ack is no later than *low*, hence no longer good (C6). If it's getting a negative ack, it must be later than the last one accepted (C7).

If $(last_s, OK) \in rs$ then $last_s \le low$ (C6)

If $(last_s, lost) \in rs$ then $last_r < last_s$ (C7)





| Name | Guard | Effect | Name | Guard | Effect |
|---|---|---|---|---|---|
| ***put*(*m*) | $msg = nil$ | $msg := m$ | | | |
| *choose*(*i*) | $msg \neq nil$, $last_s = nil$, $i = time_s$, $i \neq sent$ | $sent := i$, $last_s := i$, $deadline := now + \delta$ | **get*(*m*) | exists *i* such that $rcv_{sr}(i, m)$, $i \in (low..high)$ | $low := i$, $last_r := i$, $deadline := nil$, $send_{rs}(i, OK)$ |
| *send* | $last_s \neq nil$ | $send_{sr}(last_s, msg)$ | | | |
| **getAck*(*a*) | $rcv_{rs}(last_s, a)$ | $last_s := nil$, $msg := nil$ | *sendAck* | exists *i* such that $rcv_{sr}(i, *)$, $i \notin (low..high)$ | $low := \max(low, i)$, $send_{rs}(i,$ if $i = last_r$ then *OK* else *lost* ) if $i = last_s$ then $deadline := nil$ |
| ***crash*$_s$ | | $rec_s := true$, $deadline := nil$ | ***crash*$_r$ | | $rec_r := true$, $deadline := nil$ |
| **recover*$_s$ | $rec_s$ | $last_s := nil$, $msg := nil$, $rec_s := false$ | **recover*$_r$ | $rec_r$, $high < time_r - 2\varepsilon$ | $last_r := nil$, $low := high$, $high := time_r + 2\varepsilon + \beta$, $rec_r := false$ |
| | | | *increase-low*(*i*) | $low < i \le time_r - 2\varepsilon - \delta$ | $low := i$ |
| | | | *increase-high*(*i*) | $high < i \le time_r + 2\varepsilon + \beta$ | $high := i$ |
| *cleanup* | $sent \neq time_s$ | $sent := nil$ | *cleanup* | $last_r < time_r - 2\varepsilon - 2\delta$ | $last_r := nil$ |
| *tick*(*i*) | $time_s < i$, $\|now - i\| < \varepsilon$ | $time_s := i$ | *tick*(*i*) | $time_r < i$, $\|now - i\| < \varepsilon$, $i + 2\varepsilon < high$ or $rec_r$ | $time_r := i$ |
| *progress*(*i*) | $now < i$, $\|i - time_{s/r}\| < \varepsilon$, $i < deadline$ or $deadline = nil$ | $now := i$ | | | |

| | | | | | |
|---|---|---|---|---|---|
| $time_s$ | : *I* | := 0 (stable) | $time_r$ | : *I* | := 0 (stable) |
| *sent* | : *I* or *nil* | := *nil* | *low* | : *I* | := 0 |
| | | | *high* | : *I* | := $\beta$ (stable) |
| $last_s$ | : *I* or *nil* | := *nil* | $last_r$ | : *I* or *nil* | := *nil* |
| *msg* | : *M* or *nil* | := *nil* | | | |
| $rec_s$ | : Boolean | := *false* | $rec_r$ | : Boolean | := *false* |
| | | | *deadline* | : *I* or *nil* | := *nil* |
| | | | *now* | : *I* | := 0 |

*Table 5*. State and actions of C. Actions below the thick line handle the passage of time.

## 8 The handshake protocol H

This is the standard protocol for setting up network connections, used in TCP, ISO TP-4, and many other transport protocols. It is usually called three-way handshake, because only three packets are needed to get the data delivered, but five packets are required to get it acknowledged and all the state cleaned up (Belsnes [1976]).

As in the generic protocol, when there are no crashes the sender and receiver each go through a cycle of modes, the sender perhaps one ahead. For the sender, the modes are *idle*, *needI*, *send*; for the receiver, they are *idle*, *accept*, and *ack*. In one cycle one message is sent and acknowledged by sending three packets from sender to receiver and two from receiver to sender, for a total of five packets. Table 6 summarizes the modes and the packets that are sent.

The modes are derived from the values of the state variables $j$ and $last$:

| | | | |
|---|---|---|---|
| $mode_s = idle$ | iff $j_s = last_s = nil$ | $mode_r = idle$ | iff $j_r = last_r = nil$ |
| $mode_s = needI$ | iff $j_s \neq nil$ | $mode_r = accept$ | iff $j_r \neq nil$ |
| $mode_s = send$ | iff $last_s \neq nil$ | $mode_r = ack$ | iff $last_r \neq nil$ |

| Sender | | | | Receiver | | |
|---|---|---|---|---|---|---|
| **mode** | **send** | **advance on** | **packet** | **advance on** | **send** | **mode** |
| *idle* | see *idle* below | *put*, to *needI* | | | ($i$, *lost*) when ($i$, $m$) arrives[3] | *idle* |
| *needI* | (needI, $j_s$) repeatedly | | (needI, $j$) → | (needI, $j$) arrives, to *accept* | | |
| | | ($j_s$, $i$) arrives, to *send* | ($j$, $i$) ← | | ($j_r$, $i_r$) repeatedly | *accept* |
| *send* | ($last_s$, $m$) repeatedly | | ($i$, $m$) → | ($i_r$, $m$) arrives, to *ack* ($i_r$, done) arrives, to *idle* | | |
| | | ($last_s$, $a$ ) arrives, to *idle* | ($i$, $a$) ← | | ($last_r$, *OK*) repeatedly[4] | *ack* |
| *idle* | ($i$, done) when ($i$, $a$) arrives | | ($i$, done) → | ($last_r$, done) arrives, to *idle* | | |
| *needI* or *send* | ($i$, done) when ($j \neq j_s$, $i$) or ($i$, *OK)* arrives, to force receiver to *idle* | | | | | |

*Table 6*. Exchange of messages in H

[3] ($i$, *lost*) is a negative acknowledgement; it means that one of two things has happened:

— The receiver has forgotten about $i$ because it has learned that the sender has gotten a positive ack for $i$, but then the receiver has gotten a duplicate ($i$, $m$), to which it responds with the negative ack, which the sender will ignore.

— The receiver has crashed since it assigned $i$, and $i$'s message may have been delivered to *get* or may have been lost.

[4] ($i$, *OK*) is a positive acknowledgement; it means $i$'s message was delivered to *get*.





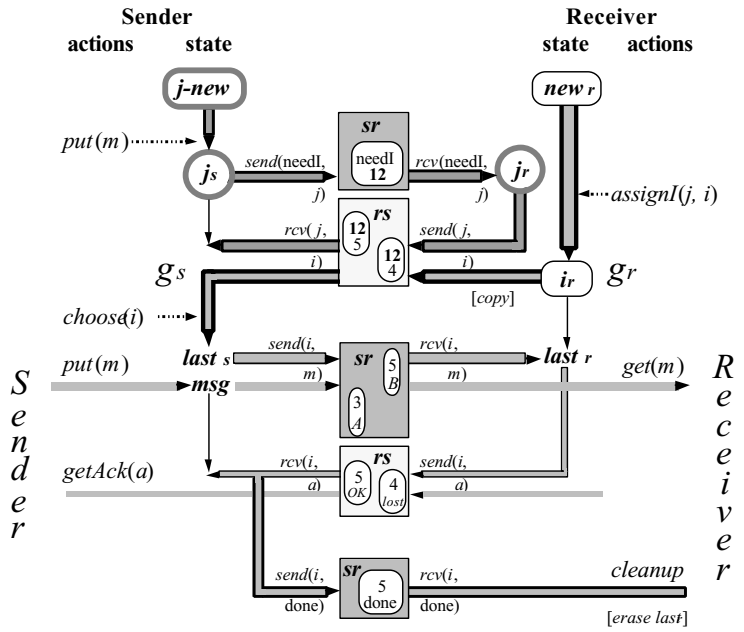



*Figure 7*. The flow of information in H

Figure 7 shows the state, the flow of identifiers from the receiver to the sender at the top, and the flow of *done* information back to the receiver at the bottom so that it can clean up. These are sandwiched between the standard exchange of message and ack, which is the same as in G (see Figure 4).

Intuitively, the reason there are five packets is that:

- One round-trip (two packets) is needed for the sender to get from the receiver an $I$ (namely $i_r$) that both know has not been used.
- One round-trip (two packets) is then needed to send and ack the message.
- A final *done* packet from the sender informs the receiver that the sender has gotten the ack. The receiver needs this information in order to stop retransmitting the ack and discard its state. If the receiver discards its "I got the message" state before it knows that the sender got the ack, then if the channel loses the ack the sender won't be able to find out that the message was actually received, even though there was no crash. This is contrary to the spec S. The *done* packet itself needs no ack, because the sender will also send it when idle and hence can become *idle* as soon as it sees the *ack*.

We introduce a new type:

> $J$, an infinite set of identifiers that can be compared for equality.

The sender and receiver send packets to each other. An *I* or *J* in the first component is there to identify the packet for the destination. Some packets also have an *I* or *J* as the second component, but it does not identify anything; rather it is being communicated to the destination for later use. The (*i, a*) and (*i,* done) packets are both often called 'close' packets in the literature.

The H protocol has the same progress and efficiency properties as G, and in addition, although the protocol as given does assume an infinite supply of *I*s, it does not assume anything about clocks.

It's necessary for a busy agent to send something repeatedly, because the other end might be idle and therefore not sending anything that would get the busy agent back to idle. An agent also has a set of expected packets, and it wants to receive one of these in order to advance normally to the next mode. To ensure that the protocol is self-stabilizing after a crash, both ends respond to an unexpected packet containing the identifier *i* by sending an acknowledgement: (*i, lost*) or (*i,* done). Whenever the receiver gets *done* for its current *I,* it becomes idle. Once the receiver is idle, the sender advances normally until it too becomes idle.

Table 7 gives the state and actions of H. The conjunct $\neg\, rec_s$ has been omitted from the guards of all the sender actions except $recover_s$, and likewise for $\neg\, rec_r$ and the receiver actions.

### 8.1 Invariants

Recall that ids(*c*) is $\{i \mid (i, *) \in c\}$. We also define jds(*c*) = $\{j \mid (j, *) \in c$ or $(*, j) \in c\}$.

Most of H's invariants are boring facts about the progress of *I*'s and *J*'s from *used* sets through $i/j_{s/r}$ to $last_{s/r}$. We need the history variables $used_s$ and *seen* to express some of them. (H6) says that there's at most one *J* (from a needI packet) that gets assigned a given *I*. (H8) says that as long as the sender is still in mode *needI*, nothing involving $i_r$ has made it into the channels.

$j\text{-}used \supseteq \{j_s, j_r\} - \{nil\} \cup \text{jds}(sr) \cup \text{jds}(rs)$ (H1)

$used_r \supseteq \{i_r, last_r\} - \{nil\} \cup used_s \cup \{i \mid (*, i) \in rs\} \cup \text{ids}(sr) \cup \text{ids}(rs)$ (H2)

$used_s \supseteq \{last_s, last_r\} - \{nil\} \cup \text{ids}(sr) \cup \text{ids}(rs)$ (H3)

If $(i, done) \in sr$ then $i \neq last_s$ (H4)

If $i_r \neq nil$ then $(j_r, i_r) \in seen$ (H5)

If $(j, i) \in seen$ and $(j', i) \in seen$ then $j = j'$ (H6)

If $(j, i) \in rs$ then $(j, i) \in seen$ (H7)

If $j_s = j_r \neq nil$ then $(i_r, *) \notin sr$ and $(i_r, done) \notin rs$ (H8)

### 8.2 Progress

We consider first what happens without failures, and then how the protocol recovers from failures.

If neither partner fails, then both advance in sync through the cycle of modes. The only thing that derails progress is for some party to change *mode* without advancing through the full cycle of modes that transmits a message. This can only happen when the receiver is in *accept* mode and gets ($i_r$, done), as you can see from Table 6. This can only happen if the sender got a packet containing $i_r$. But if the receiver is in *accept*, the sender must be in *needI* or *send*, and the only thing that's been sent with $i_r$ is $(j_s, i_r)$. The sender goes to or stays in *send* and doesn't make *done* when it gets $(j_s, i_r)$ in either of these modes, so the cycling through the modes is never disrupted as long as there's no crash.





| Name | Guard | Effect | Name | Guard | Effect |
|---|---|---|---|---|---|
| ***put*(*m*) | $msg = nil$, | $msg := m$, | | | |
| | exists *j* such that $j \notin j\text{-}used$ | $j_s := j$, $j\text{-}used$ +:= $\{j\}$ | | | |
| *requestI* | $j_s \neq nil$, $last_s = nil$ | $send_{sr}$(needI, $j_s$) | *assignI*(*j*,*i*) | $rcv_{sr}$(needI, *j*), $i_r = last_r = nil$, $i \notin used_r$ | $j_r := j$, $i_r := i$, $used_r$ +:= *i*, *seen* +:= {(*j, i*)} |
| *choose*(*i*) | $last_s = nil$, $rcv_{rs}(j_s, i)$ | $j_s := nil$, $last_s := i$, $used_s$ +:= ⟨*i*⟩ | *sendI* | $j_r \neq nil$ | $send_{rs}(j_r, i_r)$ |
| *send* | $last_s \neq nil$ | $send_{sr}(last_s, msg)$ | **get*(*m*) | exists *i* such that $rcv_{sr}(i, m)$, $i = i_r$ | $j_r := i_r := nil$, $last_r := i$, $send_{rs}(i, OK)$ |
| | | | *sendAck* | $last_r \neq nil$ | $send_{rs}(last_r, OK)$ |
| **getAck*(*a*) | $rcv_{rs}(last_s, a)$ | if $a = OK$ then $send_{sr}(last_s, done)$ $msg := last_s := nil$ | *bounce* | exists *i* such that $rcv_{sr}(i, *)$, $i \neq i_r$, $i \neq last_r$ | $send_{rs}(i, lost)$ |
| *bounce* (*j, i*) | $rcv_{rs}(j, i)$, $j \neq j_s, i \neq last_s$ or $rcv_{rs}(i, OK)$ | $send_{sr}(i,$ done) | *cleanup*(*i*) | $rcv_{sr}(i,$ done), $i = i_r$ or $i = last_r$ | $j_r := i_r := nil$, $last_r := nil$ |
| ***crash*$_s$ | | $rec_s := true$ | ***crash*$_r$ | | $rec_r := true$ |
| **recover*$_s$ | $rec_s$ | $msg := nil$, $j_s := last_s := nil$, $rec_s := false$ | **recover*$_r$ | $rec_r$ | $j_r := i_r := nil$, $last_r := nil$, $rec_r := false$ |
| *grow-j-used*(*j*) | | $j\text{-}used$ +:= $\{j\}$ | *grow-used*(*i*) | | $used_r$ +:= $\{i\}$ |

| | | | | | |
|---|---|---|---|---|---|
| $used_s$ | : sequence[*I*] | := ⟨ ⟩ (history) | $used_r$ | : set[*I*] | := { } (stable) |
| $j\text{-}used$ | : set[*J*] | := { } (stable) | *seen* | : set[(*J, I*)] | := { } (history) |
| $j_s$ | : *J* or *nil* | := *nil* | $j_r$ | : *J* or *nil* | := *nil* |
| *msg* | : *M* or *nil* | := *nil* | $i_r$ | : *I* or *nil* | := *nil* |
| $last_s$ | : *I* or *nil* | := *nil* | $last_r$ | : *I* or *nil* | := *nil* |
| $rec_s$ | : Boolean | := *false* | $rec_r$ | : Boolean | := *false* |

*Table 7.* State and actions of H. Heavy black lines outline additions to G

If either partner fails and then recovers, the other becomes idle rather than getting stuck; in other words, the protocol is self-stabilizing. Why? When the receiver isn't idle it always sends something, and if that isn't what the sender wants, the sender responds *done*, which forces the receiver to become idle. When the sender isn't idle it's either in *needI*, in which case it will eventually get what it wants, or it's in *send* and will get a negative ack and become idle. In more detail:

The receiver bails out when the sender crashes because

- the sender forgets $i_s$ and $j_s$ when it crashes,
- if the receiver isn't idle, it keeps sending $(j_r, i_r)$ or $(last_r, OK)$,

- the sender responds with ($i_r$/$last_r$, done) when it sees either of these, and
- the receiver ends up in *idle* whenever it receives this.

The sender bails out or makes progress when the receiver crashes because

- If the sender is in *needI*, either
  —it gets ($j_s$, $i \neq i_r$) from the pre-crash receiver, advances to *send*, and bails out as below, or
  —it gets ($j_s$, $i_r$) from the post-crash receiver and proceeds normally.
- If the sender is in *send* it keeps sending ($last_s$, *msg*),
  —the receiver has $last_r = nil \neq last_s$, so it responds ($last_s$, *lost*), and
  —when the sender gets this it becomes *idle*.

An idle receiver might see an old (needI, $j$) with $j \neq j_s$ and go into *accept* with $j_r \neq j_s$, but the sender will respond to the resulting ($j_r$, $i_r$) packets with ($i_r$, done), which will force the receiver back to *idle*. Eventually all the old *needI* packets will drain out. This is the reason that it's necessary to prevent a channel from delivering an unbounded number of copies of a packet.

## 9 Finite identifiers

So far we have assumed that the identifier sets $I$ and $J$ are infinite. Practical protocols use sets that are finite and often quite small. We can easily extend G to use finite sets by adding a new action *recycle*($i$) that removes an identifier from $used_s$ and $used_r$ so that it can be added to $g_r$ again. As we saw in Section 1, when we add a new action the only change we need in the proof is to show that it maintains the invariants and simulates something in the spec. The latter is simple: *recycle* simulates no change in the spec. The former is also simple: we put a strong enough guard on *recycle* to ensure that all the invariants still hold. To find out what this guard is we need only find all the invariants that mention $used_s$ or $used_r$, since those are the only variables that *recycle* changes. Intuitively, the result is that an identifier can be recycled if it doesn't appear anywhere else in the variables or channels.

Similar observations apply to H, with some minor complications to keep the history variable *seen* up to date, and a similar *recycle-j* action. Table 8 gives the *recycle* actions for G and H.

| Name | Guard | Effect |
|---|---|---|
| *recycle*($i$) for G | $i \notin g_s \cup g_r \cup \{last_s, last_r\}$ $\cup$ ids($sr$) $\cup$ ids($rs$) | $used_s$ –:= $\{i\}$, $used_r$ –:= $\{i\}$ |
| *recycle*($i$) for H | $i \notin \{last_s, i_r, last_r\}$ $\cup \{i \mid (*, i) \in rs\} \cup$ ids($sr$) $\cup$ ids($rs$) | $used_s$ –:= $\{i\}$, $used_r$ –:= $\{i\}$, $seen$ –:= $\{j \mid (j, i) \in seen \mid (j, i)\}$ |
| *recycle-j*($j$) for H | $j \notin \{j_s, j_r\} \cup$ jds($sr$) $\cup$ jds($rs$) | $used$-$j$ –:= $\{j\}$, $seen$ –:= $\{i \mid (j, i) \in seen \mid (j, i)\}$ |

*Table 8*. Actions to recycle identifiers

How can we implement the guards on the *recycle* actions? The tricky part is ensuring that $i$ is not still in a channel, since standard methods can ensure that it isn't in a variable at the other end. There are three schemes that are used in practice:





- Use a FIFO channel. Then a simple convention ensures that if you don't send any $i_1$'s after you send $i_2$, then when you get back the ack for $i_2$ there aren't any $i_1$'s left in either channel.
- Assume that packets in the channel have a maximum lifetime once they have been sent, and wait longer than that time after you stop sending packets containing $i$.
- Encrypt packets on the channel, and change the encryption key. Once the receiver acknowledges the change it will no longer accept packets encrypted with the old key, so these packets are in effect no longer in the channel.

For C we can recycle identifiers by using time modulo some period as the identifier, rather than unadorned time. Similar ideas apply; we omit the details.

## 10 Conclusions

We have given a precise specification S of reliable at-most-once message delivery with acknowledgements. We have also presented precise descriptions of two practical protocols (C and H) that implement S, and the essential elements of proofs that they do so; the handshake protocol H is used for connection establishment in most computer networking. Our proofs are organized into three levels: we refine S first into another specification D that delays some of the decisions of S and then into a generic implementation G, and finally we show that C and H both implement G. Most of the work is in the proof that G implements D.

In addition to complete expositions of the protocols and their correctness, we have also given an extended example of how to use abstraction functions and invariants to understand and verify subtle distributed algorithms of some practical importance. The example shows that the proofs are not too difficult and that the invariants, and especially the abstraction functions, give a great deal of insight into how the implementations work and why they satisfy the specifications. It also illustrates how to divide a complicated problem into parts that make sense individually and can be attacked one at a time.

## References

Abadi, M. and Lamport, L. (1991), The Existence of Refinement Mappings, *Theoretical Computer Science* **82** (2), 253-284.

Belsnes, D. (1976), Single Message Communication, *IEEE Trans. Communications* **COM-24**, 2.

Lampson, B., Lynch, N., and Søgaard-Andersen, J. (1993), Reliable At-Most-Once Message Delivery Protocols, Technical Report, MIT Laboratory for Computer Science, to appear.

Liskov, B., Shrira, L., and Wroclawski, J. (1991), Efficient At-Most-Once Messages Based on Synchronized Clocks, *ACM Trans. Computer Systems* **9** (2), 125-142.

Lynch, N. and Vaandrager, F. (1993), Forward and Backward Simulations, Part I: Untimed Systems, Technical Report, MIT Laboratory for Computer Science, to appear.

## Appendix

For reference we give the complete protocol for G, with every action as atomic as it should be. This requires separating the getting and putting of messages, the sending and receiving of packets, the sending and receiving of acks, and the getting of acks. As a result, we have to add buffer queues $buf_{s/r}$ for messages at both ends, a buffer variable *ack* for the ack at the sender, and a *send-ack* flag for positive acks and a buffer *nack-buf* for negative acks at the receiver.

The state of the full G is:

| | | | | | |
|---|---|---|---|---|---|
| $used_s$ | : sequence[*I*] | := ⟨ ⟩ (stable) | $used_r$ | : set[*I*] | := { } (stable) |
| $g_s$ | : set[*I*] | := { } | $g_r$ | : set[*I*] | := { } |
| $last_s$ | : *I* or *nil* | := *nil* | $last_r$ | : *I* or *nil* | := *nil* |
| $buf_s$ | : sequence[*M*] | := ⟨ ⟩ | $buf_r$ | : sequence[*M*] | := ⟨ ⟩ |
| *msg* | : *M* or *nil* | := *nil* | *mark* | : + or # | := + |
| *ack* | : *A* | := *lost* | *send-ack* | : Boolean | := *false* |
| | | | *nack-buf* | : sequence[*I*] | := ⟨ ⟩ |
| $rec_s$ | : Boolean | := *false* | $rec_r$ | : Boolean | := *false* |

The abstraction function to D is:

| | | | |
|---|---|---|---|
| *q* | the elements of $buf_r$ paired with + | | |
| | + *old-q* + *cur-q* | | |
| | + the elements of $buf_s$ paired with + | | |
| *status* | (?, +) | if $buf_s \neq empty$ | |
| | else (?, *mark*) | if *cur-q* ≠ { } | (a) |
| | (?, +) | if $mode_s = send$, $last_s = last_r$, $buf_r \neq$ { } | (b) |
| | (*OK*, +) | if $mode_s = send$, $last_s = last_r$, $buf_r =$ { } | (c) |
| | (*OK*, #) | if $mode_s = send$ and $last_s \in inflight_{rs}$ | (d) |
| | (*lost*, +) | if $mode_s = send$ and $last_s \notin (g_r \cup \{last_r\} \cup inflight_{rs})$ | (e) |
| | (*ack*, +) | if $mode_s = idle$ | (f) |
| $rec_{s/r}$ | $rec_{s/r}$ | | |





| Name | Guard | Effect |
|---|---|---|
| ***put*(*m*) | | append *m* to $buf_s$ |
| *prepare*(*m*) | $msg = nil$, *m* first on $buf_s$, $g_s \subseteq g_r$ or $rec_r$ | $buf_s$:=tail ($buf_s$), $msg := m$, *mark* := + |
| *choose*(*i*) | $msg \neq nil$, $last_s = nil$, $i \in g_s$ | $g_s$ –:= $\{j \mid j \leq i\}$, $last_s := i$, $used_s$ +:= ⟨*i*⟩ |
| $send_{sr}(i, m)$ | $i = last_s \neq nil$, $m = msg$ | |
| $rcv_{rs}(i, a)$ | if $i = last_s$ then | $ack := a$, $msg := nil$, $last_s := nil$ |
| **getAck*(*a*) | $msg = nil$, $buf_s = empty$, $ack = a$ | $ack := lost$ |
| **$crash_s$ | | $rec_s := true$ |
| **recover*$_s$ | $rec_s$ | $last_s := nil$, $msg := nil$, $buf_s := \langle\ \rangle$, $ack := lost$, $rec_s := false$ |
| $shrink_s(i)$ | | $g_s$ –:= {*i*} |
| $grow_s(i)$ | $i \notin used$, $i \in g_r$ or $rec_r$ | $g_s$ +:= {*i*} |
| $grow\text{-}used_s(i)$ | $i \notin used \cup g_s$, $i \in used_r$ or $rec_r$ | $used$ +:= {*i*} |

| Name | Guard | Effect |
|---|---|---|
| $rcv_{sr}(i, m)$ | if $i \in g_r$ then append *m* to $buf_r$, *sendAck* := *false*, $g_r$ –:= $\{j \mid j \leq i\}$, $last_r := i$, else if $i \notin g_r \cup \{last_r\}$ then optionally *nack-buf* +:= ⟨*i*⟩ else if $i = last_r$ then *sendAck* := *true* | |
| **get*(*m*) | *m* first on $buf_r$ | if $buf_r = \langle m \rangle$ then *sendAck* := *true*, $buf_r$ :=tail ($buf_r$) |
| $send_{rs}$ (*i, OK*) | $i = last_r$, *sendAck* | optionally *sendAck* := *false* |
| $send_{rs}$ (*i, lost*) | *i* first on *nack-buf* | *nack-buf* := tail (*nack-buf*) |
| **$crash_r$ | | $rec_r := true$ |
| **recover*$_r$ | $rec_r$, $used_r \supseteq g_s \cup used_s$ | $last_r := nil$, *mark* := #, $buf_r$:=⟨ ⟩, *nack-buf*:=⟨ ⟩, $rec_r$:=*false* |
| $shrink_r(i)$ | $i \notin g_s$, $i \neq last_s$ or *mark* = # | $g_r$ –:= {*i*} |
| $grow_r(i)$ | $i \notin used_r$ | $g_r$ +:= {*i*}, $used$ +:= {*i*} |
| *cleanup* | $last_r \neq last_s$ | $last_r := nil$ |
| *unmark* | $g_s \subseteq g_r$, $last_s \in g_r \cup \{last_r, nil\}$ | *mark* := + |

*Table 9*. G with honest atomic actions