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 implementation 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 implementation 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.

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 implemented on top of a wide variety of existing communication mechanisms.

Types and references

Like the Modula 3 system described in handout 24, 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 implementation. 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 = self`. 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.

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.

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

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          =  Meth -> (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 a basic implementation of 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 implement 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 29 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`. In the next section we discuss some of the details of this process.

MODULE NetObj = *% implements* 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

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

VAR  self       :  Space
     sendSR     := Ch.SR{s := self}

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(self); 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. An implementation 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(self); m IS Req => SKIP >>;
          remote := m.remote;
          IF remote.space = self => VAR local := export(self)(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

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 implementation 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. We now look at a network object system with 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 implementation techniques are quite different.

Modula-3 network objects

The network objects described in the paper by Birrell et al. (handout 24) address all of these issues except for fault-tolerance, and they provide a framework for that as well. They 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 implementations of 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 implementation of an object is a record that holds its state along with a pointer to a record of procedures for the class.

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.

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           =  Meth -> (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

Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 — Principles of Computer Systems

Handout 23 April 16, 1997

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 20 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 implementations of 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 implementation, 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 Stubs and pickles
Network references	Location, architecture, and language	Globally meaningful names
Request-response	Concurrency	Server: work queue 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