Network Objects

A. Birrell, G. Nelson, S. Owicki, E. Wobber, DEC SRC
SOSP 1994

Context and motivation

- A lot of work on distributed object systems Argus, Eden, Emerald, SOS, …
- Unclear what's important and what's not
- Goals
  - A new data point – an implementation of essential features
    - Just those features valuable to *all* distributed apps
  - A network object system implemented in Modula-3
    - It's design rational and implementation details
- Why do you care?
  - Most of the basic ideas behind Java RMI
  - A great design rational and implementation description

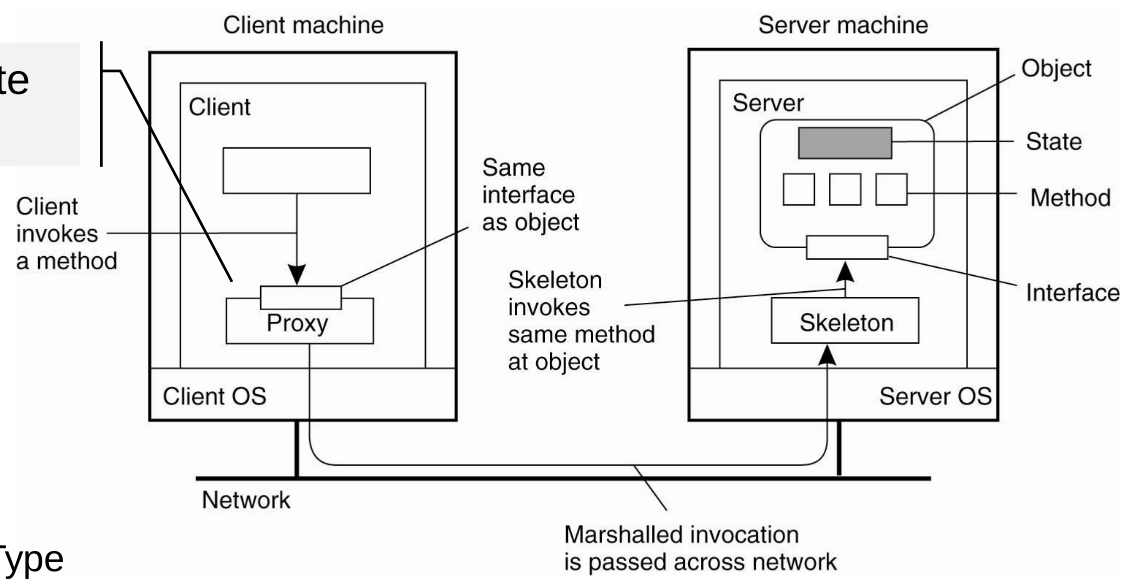
Basic concepts

- A Modula-3 object – a reference to data record + method suite
  - Method suite – a record of procedures that accept the object as first parameter
  - Includes a *typecode* that can be used to determine its type dynamically
- New object type can be defined as subtype of an existing one
  - New object has all methods of the original (single inheritance)
  - It can provide additional methods
  - … and new implementations of existing ones (overriding)

Basic concepts

- Network object – one that can be invoked by other programs
  - Reference in client program points to a surrogate object whose methods perform RPC to the owner of it

Surrogate object



T – Network object type

TSrg - Surrogate object type

TImpl – Type of the real object located in the owner

Basic concepts

- Third party transfer
  - If A has a ref to a network object owned by B,
  - A can pass the ref to C
  - C can then call the methods of the object – third party transfer
- When a client first receives a ref to a given network object, an appropriate surrogate is created by the unmarshalling code
  - The type of the surrogate must be determined
  - narrowest surrogate rule: 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 at both ends

Example – a trivial file service

Interface

```
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.
```

Two networked objects – File and Server; all network objects are subtype of NetObj.T

These types are pure, i.e. there are no data fields associated with them (they should go between OBJECT & METHODS)

The I/F defines two network object types, one for opening files and one for reading them.

If pointed to the interface FS, the stub generator will produce a module containing client & server stub for both types.

Example

Implementation

```
MODULE Server EXPORTS Main;
IMPORT NetObj, FS, Time;
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.
```

Exports net object Svr – it places a reference to it in a table under the name FS1; the table is contained in an agent process running in the same mach as the server.

Example

Client

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

VAR
  s: FS.Server:=
     NetObj.Import(“FS1”,
       NetObj.LocateHost(“server”));
  f := s.open(“/usr/dict/words”);
BEGIN
  WHILE NOT f.eof() DO
    IO.PutChar(f.getChar())
  END
END Client.
```

(LocateHost) Returns handle on the agent process running on the machine named “server”

(Import) Returns net object stored in the agent’s table under the name FS1 (i.e. ‘Svr’)

The net obj *s* is exported by name but *f* is anonymous, i.e. not present in any agent table

Implementation details - assumptions

- Single inheritance & some basic primitives wrt objects (testing its type at runtime, find the code for the direct supertype given the code for the type, …)
- Threads
- Some form of inter-address space (AS) communication
- Garbage collection
- A method for communicating object typecodes between AS
  - Typecodes are unique within an AS, M3 compiler generates a fingerprint for every object type (like a hash of the object structure)
  - Every AS contains two tables mapping typecodes to fingerprints
  - typecode→fingerprint before sending, fingerprint → typecode upon reception
- OO buffered streams
  - Object type for which the method for filling/flushing the buffer can be overridden differently in different subtypes (readers & writers in M3)

Implementation – Garbage collection

- Network-wide reference garbage collection (gc)
- For each exported object, runtime records set of clients containing surrogates (dirty set)
  - As long as set is not empty, it also retains a pointer to protect the object from local GC
  - Keeping list of clients allows detection of clients that exit/crash
  - When surrogate is created, it registers a procedure with the local GC to be called for cleanup (with a RPC call to owner)
- Can't GC cycles, that's a programmers problem
- There's a problem with passing references – if A sends B a reference to an object owned by C, A may call clean before B calls dirty – object's gone
  - Not a problem if ref is passed as an argument in a RPC since calling thread retains a reference to the object
  - … but a real one if the object is sent as a result – request an ACK (a simple solution at the cost of an additional message)

Implementation – basic representation

- Wire representation of a netobj: (sp, i)
  - sp: SpaceID – number that ID the owner of the object
  - i: ObjID – number that ID object among others by same owner
- Each AS keeps an object table with
  - All its surrogates
  - All its *exported* netobjs
- The concrete representation of a netobj includes
  - Wire representation
  - Object state – surrogate, exported, unexported
  - If state = surrogate, location to generate connections to owner's AS
  - If state = exported, dispatcher
- Dispatcher – dispatcher procedure for the object
  - Unmarshals a method number/index and argument from a connection c, calls the appropriate method of obj, and marshals and sends the results over c

Implementation – remote invocation

- The stub-generated surrogate declaration for FS.Server with a single method open

```
SrgSvr = FS.Server OBJECT
  OVERRIDES open:=SrgOpen END;
SrgOpen(ob: SrgSvr; n: TEXT): FS.File =
 VAR
   c:=ob.loc.new();
   res: FS.file;
 BEGIN
  MarshalNetObj(ob,c);
  MarshalInt(0,c);
  MarshalText(n,c);
  Flush(c.wr);
  res:=UnmarshalNetObj(c);
  ob.loc.free(c);
  RETURN res
 END;
```

Generate a connection to the owner AS

Identifies the method open on the wire

Writes to the connection c the wire representation of the reference obj. If the object has not been exported before, create its wire representation, and insert it in the object table with the associated dispatcher

Implementation – remote invocation

- Meanwhile, on the server side …

```
VAR
  ob:= UnmarshalNetObj(c);
BEGIN
  ob.disp(c,ob)
END;
```

The thread forked by the transport to service a connection c, runs something like this

```
   SvrDisp(c: Connection; o: FS.Server) =
     VAR
       methID:= UnmarshalInt(c);
     BEGIN
       IF methID = 0 THEN
         VAR
           n:= UnmarshalText(c);
           res: FS.File;
         BEGIN
           res := o.open(n);
           MarshalNetObj(res,c);
           Flush(c.wr);
         END
       ELSE
          <error, non-existent method>
       END
     END
```

The dispatcher procedure is typically written by the stub generator

Implementation – remote invocation

```
UnmarshalNetOb(c: Connection): NetObj.T =
  VAR
    sp := UnmarshalInt(c);
    i  := UnmarshalInt(c);
    wrep := (sp,i);
  BEGIN
    IF sp = -1 THEN RETURN NILL
    ELSIF objtbl[wrep] # NIL THEN
      RETURN objtbl[wrep]
    ELSE
      RETURN NewSurrogate(sp, i, c)
    END
  END;
```

```
NewSurrogte(sp: SpaceID, i: ObjID, c:
Connection): NetObj.T =
  VAR
    loc := Locate(sp,conn);
    tc  := ChooseTypeCode(loc,i);
    res := Allocate(tc);
  BEGIN
    res.wr := (sp,i);
    res.state := Surrogate;
    objtbl[(sp,i)] := res;
    RETURN res
  END
```

Returns a location that generates connections to sp

Allocates an object with type code tc

Implementation – remote invocation

Return the code for the calling AS's surrogate type for the object with ID i and whose owner is the AS to which loc generates connections
It also calls dirty

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

Implementation – remote invocation

```
Dirty(i: ObjID, sp: SpaceID): SEQ[Fingerprint] =
 VAR
   tc:= TYPE(objtbl[(SelfID(),i)]);
   res:= SEQ[Fingerprint]:= empty;
 BEGIN
   < add sp to 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
```

Extends sequence res with new element ‘argument’

Converts between equivalent typecodes and fingerprints

Light evaluation

- Implemented in 1 year by 4 people
- Size - ~10k SLOC
  - Runtime system 4k SLOC
  - Stub generator 3k SLOC
  - TCP transport 1,5K SLOC
  - Pickle package 750 SLOC
  - Network object agent 100 SLOC
- Numbers

Null call	3310 usecs/call	1600 usecs for a C-based TCP echo from user to user space, plus marshalling/unmarshalling, mas two user space context switches.
Ten integer call	3435 usecs/call	Every integer is +12usec
Same object argument	3895 usecs/call	Doesn’t lead to a dirty call
Same object return	4290 usecs/call	Extra call for ‘return’ is due to the ack needed
New object argument	9148 usecs/call	This requires a dirty call
New object return	10253 usecs/call	Same but also an ack
Reader test	2824 KB/sec	Throughput of a raw TCP stream using C ~3400 KB/sec, overhead comes from M3 user space thread switch

Experience

- System was under use by the group, by paper submission they had built
  - packagetool – a tool that allows software packages to be checked in and out of a repository
  - siphon - used to link repositories that are too far apart to be served by the same distributed file system
- Use of networked objects resulted on
  - Simpler interfaces
    - Transfer of objects simplifies implementation of siphon
  - Smaller, simpler implementations
    - Using a link structured of directory elements with one call rather than a set of RPC calls
  - More flexible implementations
    - Easy to plug a different transport