Distributed Mobile Computation in

Obliq

Luca Cardelli

Digital Equipment Corporation
Systems Research Center

Abstract

Obliq is lexically-scoped, untyped, interpreted language that supports distributed object-oriented computation. Obliq objects have state and are local to a site. Obliq computations can roam over the network, while maintaining network connections. Distributed lexical scoping is the key mechanism for managing distributed computations.

Distributed Scripting

- Scripting languages are (normally) interpreted languages used to coordinate applications. They are particularly popular for connecting system services (programmed in “real languages”) to human interfaces (non-programmable). They are supposed to be used mostly by end-users and system managers.
- Recently, scripting languages have emerged for coordinating activities in distributed environments: for telecollaboration, telemarketing, navigation, etc.
- Scripting languages are often used to coordinate applications that are written in normal languages. However, there are situations in which a scripting language, with its libraries and user interface capabilities, can be used to program an entire application; even a distributed one.

What Makes a Good Distributed Scripting Language?

- The intended computation paradigm is based on distributed objects and roaming agents. This is qualitatively new, and exposes new problems.
- For example, a piece of code can run for a while on a machine and then migrate to, or exchange data with, another machine over the network (with a different processor, byte order, OS, etc.).
- The purpose of distributed scripting languages is then to overcome the difficulties arising from heterogeneity and distribution, to make the network as transparent as possible, and to facilitate the use of network resources.
- The goal is to obtain a language than can use distributed (say, Internet) resources as easily and securely as a conventional language can use local resources.

Obliq Language Overview

- Distributed computation ≈ *network objects* (i.e. network interfaces). Obliq supports objects in this spirit, relying for its implementation on Modula-3's network objects [5].
- Simple and powerful object primitives. Coherence between local and distributed semantics. Objects are collections of named fields, with four basic operations: selection/invocation, update/override, cloning, and redirection. (No class or delegation hierarchies, no complex method-lookup procedures).
- Every object is potentially and transparently a network object. An object may become accessible over the network either by the mediation of a name server, or by simply being used as the argument or result of a remote method.

- Obliq objects are local to a site and are never automatically copied over the network. In contrast, *network references* to objects can be transmitted from site to site without restrictions. Object migration can be coded from cloning and redirection.
- Obliq computations (*closures*, not source text) can be freely transmitted. Lexically scoped free identifiers retain their bindings to the originating sites. Through these free identifiers, migrating computations (*agents*) can maintain connections to objects and locations residing at various sites. Disconnected agents can be represented as procedures with no free identifiers.
- To concentrate on distributed computing issues, Obliq is designed as an *untyped* language. The language is in principle suitable for static typing.
- Obliq is *strongly typed* at run-time: erroneous computations produce clean errors and exceptions that are correctly propagated across sites.

Distributed Lexical Scoping

- Obliq [11] extends a familiar language feature, *lexical scoping*, to a distributed context. The situation is analogous to the extension of local procedure call to remote procedure call, but more general. In Obliq, procedures can be not only invoked, but also transmitted, over the network. Thanks to distributed lexical scoping, these roaming computations have a precise meaning which is site-independent (except for site-dependent data they receive as arguments), and therefore have a predictable behavior.
- The main technical issue is to find a meaning for *higher-order distributed computations* (e.g. as in a compute server receiving a procedure to execute): what happens to the free identifiers of network-transmitted procedures? Obliq takes the view that such identifiers are bound to their original locations and network sites, as prescribed by lexical scoping.
- Remote execution is nothing new. But, so far, all proposals either (A) restrict the ability to pass over the network procedures that have free identifiers or procedure parameters, (B) adopt dynamic scoping for those free identifiers, or (C) pass program text instead of active computations.

Objects

An **object** is a collection of named fields:

$$\{l_1 \texttt{ => } a_1, \ \ldots\ , l_n \texttt{ => } a_n\}$$

Value fields contains values:

```
l => 3
```

Method fields contain methods of the form (where `x` is “self”):

$$l \texttt{ => } \textbf{meth}(x, x_1, \ \ldots\ , x_n)\ b\ \textbf{end}$$

Alias fields contain aliases that redirect operations to other objects:

```
l => alias l’ of o’ end
```

Object Operations

Select and Invoke

a.x	value selection
a.x(b_1, ... ,b_n)	method invocation

Update and Override

a.x := b	field update / method override

Cloning

clone(a)	“shallow copy” of a
clone(a_1, ... ,a_n)	concatenation (inheritance)

Redirection

redirect a_1 **to** a_2 **end**	Further operations on fields of a_1 are redirected to similar fields of a_2

Example

```
let o =
   {  x => 3,
      inc => meth(s,y) s.x := s.x+y; s end,
      next => meth(s) s.inc(1).x end }

o.x
o.x := 0
o.inc(1)
o.next (or o.next())
o.next := meth(s) clone(s).inc(1).x end
```

Network objects

An object can be initially shared through a name server:

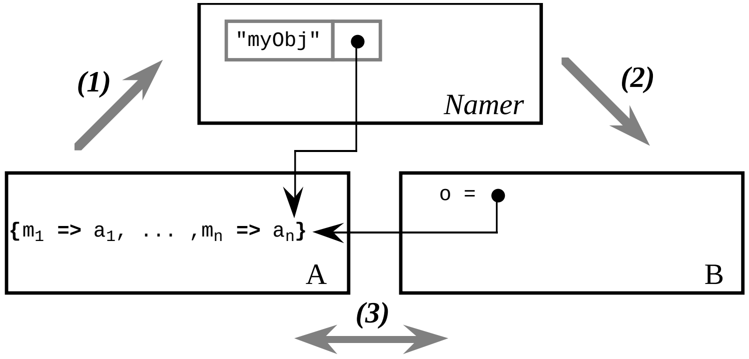
Site A:

(1) net_export(“myObj”,*Namer*,{m_1 => a_1, ... ,m_n => a_n});

Site B:

(2) **let** o = net_import(“myObj”,*Namer*);

(3) o.m_1(b);

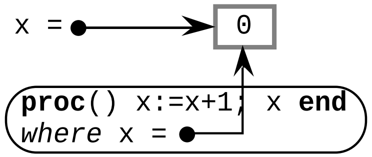


Lexical Scoping and Closures

```
var x = 0;
let f = proc() x := x+1; x end;
f(); (* = 1 *)
```

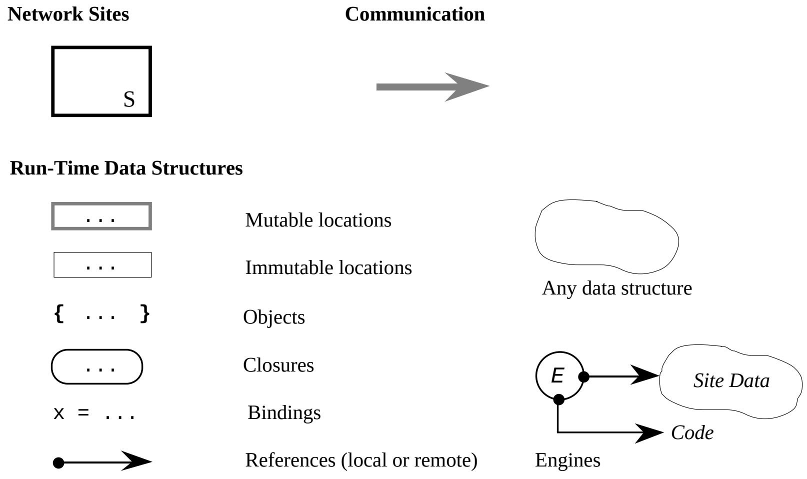
In a higher-order language, a procedure value (a *closure*) may escape the scope of its free identifiers. But the closure retains the appropriate identifier bindings.

```
let g = proc()
   var x = 0;
   proc() x := x+1; x end;
end;
let f = g();
f(); (* = 1 *)
```



In a distributed language a procedure value may escape not only the *scope* of its free identifiers, but even their *address space*.

Legend



Network Sites

Communication

Run-Time Data Structures

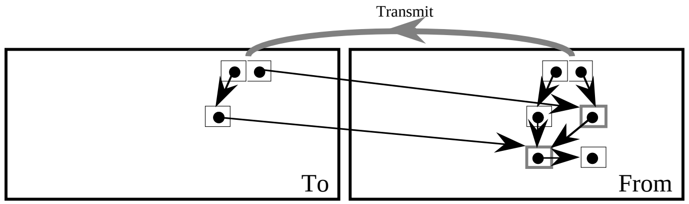
...	Mutable locations
...	Immutable locations
{ ... }	Objects
...	Closures
x = ...	Bindings
●→	References (local or remote)

Any data structure

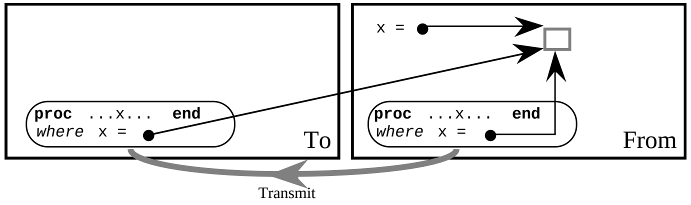
Engines

Value Transmission

On transmission, values are copied up to their embedded mutable locations; local pointers to mutable locations are converted to network pointers.



This rule applies, in particular, to the transmission of closures:



- Obliq data is network-transparent: immutable data may be duplicated, but state is never automatically duplicated. (Cloning allows explicit state duplication.)
- Obliq computations are network-transparent. Their effect on free variables is the same no matter where they execute. (However, procedures may receive different parameters at different sites.)
- Obliq programs are network-aware. Distribution is achieved by explicit acts, which give full control on allocations sites and execution sites. It is possible (in principle) to figure out what is happening where.
- Distributed lexical scoping makes it easy to distribute computations over multiple sites. Even when execution is carried out at the wrong place (by some measure) it behaves correctly.
- This flexibility in distribution has a down side: the resulting network traffic may be hard to predict or understand. Satisfactory distributed performance still requires care and planning.

Distributed Techniques

What follows is a consequence of the distributed semantics just explained.

Compute Servers / Remote Execution Engines.

Remote Agents.

Agent Migration.

Object Migration.

Safe Execution.

Application Partitioning.

Application Servers.

Application Migration.

Remote Execution Engines

A remote execution engine is a built-in compute server for Obliq programs. It accepts Obliq procedures (that is, procedure closures) from the network and executes them at the engine site. An engine can be exported from a site via the primitive:

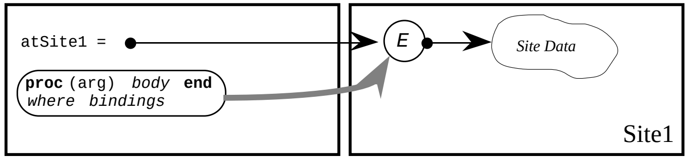
```
net_exportEngine("Engine1@Site1", Namer, arg);
```

The "arg" parameter is supplied to all the client procedures received by the engine. Multiple engines can be exported from the same site under different names.

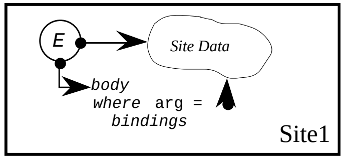
A client may import an engine and then specify a procedure to be execute remotely. An imported engine behaves like a procedure of one argument:

```
let atSite1 =
  net_importEngine("Engine1@Site1", Namer);
atSite1(proc(arg) 3+2 end);
```

Before transmission:



After transmission:



Remote Agents

Compute servers and execution engines can be used as general object servers. That is, as ways of allocating objects at remote sites. These objects can then act as *agents* of the initiating site.

Suppose, for example, that we have an engine at a database server site. The engine provides the database as an argument to client procedures:

```
(* DataBase Server Site *)
net_exportEngine("DBServer", dataBase, Namer);
```

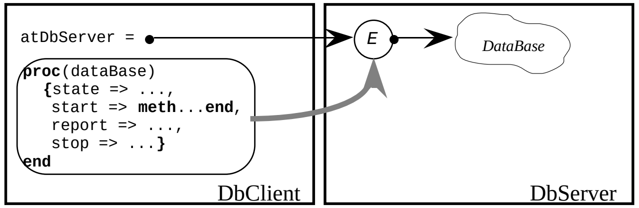
A database client could send over procedures performing queries on the database. However, for added flexibility, the client can instead create a remote object:

```
(* DataBase Client Site *)
let atDbServer =
  net_importEngine("DBServer", Namer);

let searchAgent =
  atDbServer(
    proc(dataBase)
      {state => ...,
       start => meth ... end,
       report => meth ... end,
       stop => meth ... end}
    end);
```

The execution of the client procedure causes the allocation of an object at the server site with methods "start", "report", and "stop", and with a "state" field. The server simply returns a network reference to this object, and is no longer engaged.

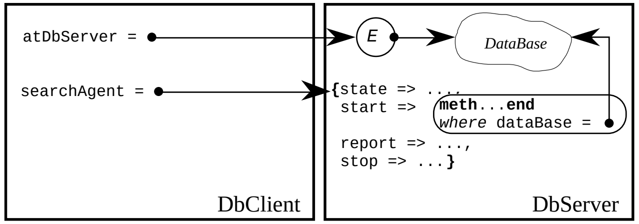
Before the invocation:



```
atDbServer =

proc(dataBase)
  {state => ...,
   start => meth...end,
   report => ...,
   stop => ...}
end
```

After its completion:



```
atDbServer =

searchAgent =

{state => ...,
 start => meth...end
          where dataBase =
 report => ...,
 stop => ...}
```

Agent Hopping

An agent is a computation that may hop from site to site over the network.

- A *suitcase* is a piece of data that an agent carries with it as it moves.
- A *briefing* is data that an agent receives at each site, as it enters the site.
- An *agent server*, for a given site, is a program that accepts code over the network, executes the code, and provides it with a local briefing.
- A *hop instruction* is used by agents to move from one site to the next. This instruction has as parameters an agent server, the code of an agent, and a suitcase. The agent and the suitcase are sent to the agent server for execution.
- Finally, an *agent* is a user-defined piece of code parameterized by a suitcase and a briefing. All the data needs of the agents should be satisfied by what it finds in either the suitcase or the briefing parameters.

If an agent has a user interface, it takes a snapshot of the interface, stores it in the suitcase during the hop, and rebuilds the interface from the snapshot at the destination.

In Obliq, agents, suitcases, briefings, and hop instructions are not primitive notions. They can be fully understood in terms of the Obliq network semantics.

Agent are just procedures of two parameters. Suitcases and briefings are arbitrary pieces of data, such as objects.

```
let rec agent =
   proc(suitcase, briefing)
      (* work at the current site *)
      (* decide where to go next *)
            ...
      hop(nextSite, agent, suitcase);
      (* run agent at nextSite with suitcase *)
   end;
```

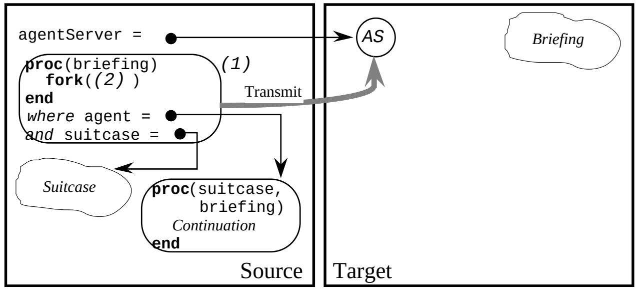
Each agent is responsible for the contents of its suitcase, and each agent server is responsible for the contents of the briefing. Agent servers are simple compute servers whose main task it to run agents and supply them with appropriate briefings (and maybe check the agent's credentials).

The hop instruction can be programmed in Obliq as follows:

```
let hop =
   proc(agentServer, agent, suitcase)
      agentServer(
(1)   proc(briefing)
            fork(
(2)         proc()
               agent(copy(suitcase), briefing);
               end);
            ok
         end);
   end;
```

- Fork: fork a thread.
- Copy: make a complete local copy of a possibly distributed structure.

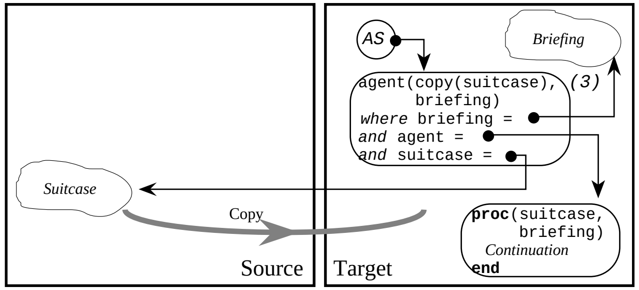
Suppose a call hop(agentServer, agent, suitcase) is executed at a source site. Here, agentServer is (a network reference to) a remote compute server at a target site.



The call agentServer(...) has the effect of shipping the procedure (1) to the remote agent server for execution. At the target site, the agent server executes the closure for procedure (1) by supplying it with a local briefing.

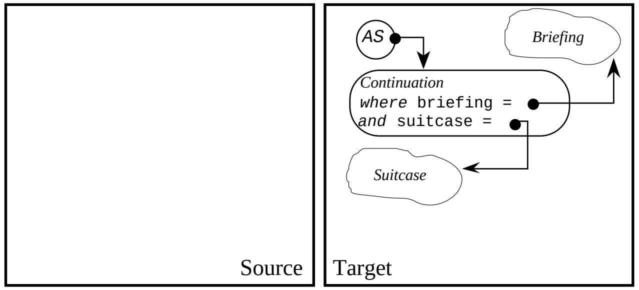
Next, at the target site, the execution of the body of (1) causes procedure (2) to be executed by a forked thread. Immediately after the fork instruction, procedure (1) returns a dummy value (ok), thereby completing the call to hop that originated at the source site.

The source site is now disengaged, while the agent computation carries on at the target site. At the target site, the forked procedure (2) executes copy(suitcase). The suitcase, at this point of the computation, is usually a network pointer to the former suitcase that the agent had at the source site.



The **copy** instruction (an Obliq primitive) makes a complete local copy of any local or distributed data.

Thus, **copy**(suitcase) is a suitcase whose state is local to the target site, suitable for local use by the agent.



After the copying of the suitcase, the agent migration is complete. The source site could now terminate or crash without affecting the migrated agent.

Finally (3), the agent is invoked with the local suitcase and the local briefing as parameters. The program text of the agent was copied over as part of the closure of procedure (1). Since the agent has no free variables, it can execute locally.

In the special case when the suitcase contains the entire application state, we have a migratory application.

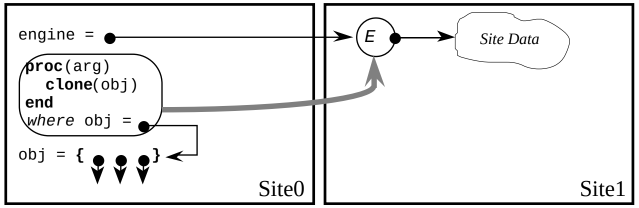
Object Migration

This example uses a remote execution engine to migrate an object between two sites. First we define a procedure that, given an object, the name of an engine, and a name server, migrates the object to the engine's site. Migration is achieved in two phases: (1) by causing the engine to remotely clone the object, and (2) by redirecting the original object to its clone.

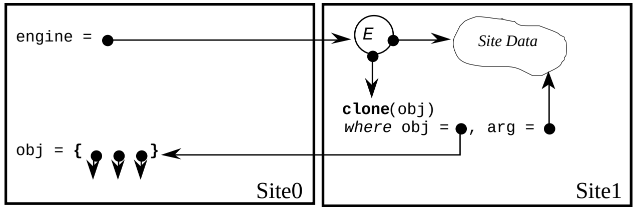
```
let migrateProc =
  proc(obj, engineName)
    let engine = net_importEngine(engineName, NS);
    let remoteObj = engine(proc(arg) clone(obj) end);
    redirect obj to remoteObj end;
    remoteObj;
  end;
```

After migration, all operations on the original object are redirected to the remote site, and executed there.

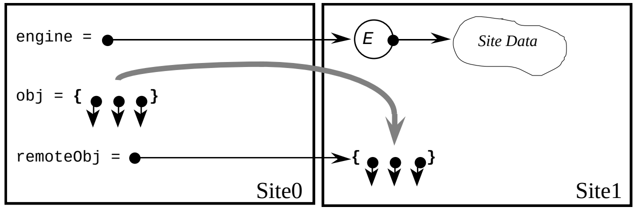
Before the engine invocation:



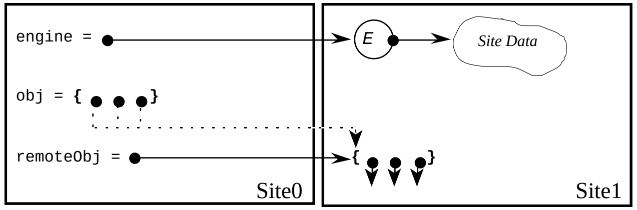
Before cloning:



After cloning:



After redirection:



It is critical that the two phases of migration be executed *atomically*, to preserve the integrity of the object state. This can be achieved by serializing the migrating object, and by invoking the "`migrateProc`" procedure from a method of that object, where it is applied to self:

```
let obj1 =
  { serialized, protected,
    ...
    migrate =>
      meth(self, engineName)
        migrateProc(self, engineName);
      end};

let remoteObj1 = obj1.migrate("Engine1@Site1")
```

Because of serialization, the object state cannot change during a call to "`migrate`". The call returns a network reference to the remote clone that can be used in place of "`obj1`" (which, anyway has been redirected to the clone).

Migration permanently modifies the original object, redirecting all operations to the remote clone. In particular, if `"obj1"` is asked to migrate again, the remote clone will properly migrate.

We can avoid accumulating aliasing indirections if the migrating object "`obj1`" is publicly available through a name server. The "`migrate`" method can then register the migrated object with the name server under the old name:

```
let obj1 =
  net_export("obj1", NS,
    { serialized, protected,
      ...
      migrate =>
        meth(self, engineName)
          net_export("obj1", NS,
            migrateProc(self, engineName));
        end});
```

This way, old clients of `"obj1"` go through aliasing indirections, but new clients acquiring `"obj1"` from the name server operate directly on the migrated object.

Safe Execution

Operations that may affect critical resources require capabilities, which are bound to global lexically-scoped identifiers at each site:

```
rd_open(fileSys, “/etc/passwd”);
process_new(processor, [“rm”, “*”], true);
```

Because of lexical scoping, capabilities at other sites are not visible to migrating procedures. Those capabilities can be obtained only with server cooperation.

```
atEngineSite(
  proc(arg)
    let file1 = rd_open(fileSysReader, “to-do-list”);
    let file2 = rd_open(arg.publicFileSys, “test”);
    ...
  end);
```

Restricted access privileges can be modeled by different capabilities. Above, `fileSysReader` refers remotely back to the client file system in read-only mode (it does not refer to the compute server file system). A client-accessible file system may be provided by the server through the `arg` parameter.

Modula-3 Network Objects

- The Obliq implementation is based on the Modula-3 Network Objects library (A.D.Birrell, G.Nelson, S.Owicki, E.Wobber [5]). This library supports remote method calls (RPC for objects) by providing simple name services, transport for Modula-3 data tapes (except procedures), stub generation, and distributed garbage collection.
- A language like Obliq is easy to implement on top of such a library. In fact, Obliq would never have been conceived without it.
- Vice versa, a network-objects library should make it easy to implement a language like Obliq, as a test of completeness. This not the case for most or all other network-object libraries.

Distributed Runtime

These are the features that one should come to expect from a good distributed runtime. Given these features, it is straighforward to implement a language like Obliq:

- Network objects.
- Marshalling/pickling of *arbitrary* data structures.
- Distributed garbage collection.
- Correct error propagation.
- Unforgeable network pointers (secure capabilities).
- Network identities (for access control lists).

Modula-3 Network Objects provide these features (the last two are part of an unreleased version).

Uses of Obliq

- As a scripting language for a user-interface toolkit (M.H.Brown, J.R.Meehan [7]) including digital video (S.M.G.Freeman, M.S.Manasse [10]).
- As a scripting language for algorithm animation (M.H.Brown [6]).
- As a scripting language for 3D graphics (M.Najork, M.H.Brown [11,12]).
- The underlying language of Visual Obliq, a distributed-application and user-interface builder (K.Bharat, M.H.Brown [2]).
- As a language enabling dynamic application migration (K.Bharat, L.Cardelli [3,4]).
- As an experimental client of secure network objects (M.Abadi, L.vanDoorn, T.Wobber), for secure scripting (future work).
- As distributed applets for a Web browser (M.Najork, M.H.Brown [1]).

Binaries / Documentation

http://www.research.digital.com/SRC/personal/Luca_Cardelli/home.html

References

[1] M.Najork and M.H.Brown, **Distributed Active Objects**. Report, Digital Equipment Corporation, Systems Research Center. 1996.

[2] Bharat, K. and M.H. Brown, **Building distributed applications by direct manipulation**. *Proc. UIST'94*. 1994.

[3] Bharat, K. and L. Cardelli: **Migratory applications**, *Proc. of the ACM Symposium on User Interface Software and Technology '95*. 133-142. 1995.

[4] Bharat, K. and L. Cardelli, **Distributed applications in a multimedia setting**, *Proc. of the First International Workshop on Hypermedia Design*. 185-192. Montpelier France, 1995.

[5] Birrell, A.D., G. Nelson, S. Owicki, and E. Wobber, **Network objects**. *Proc. 14th Symposium on Operating Systems Principles*. 1993.

[6] Brown, M.H., **Report on the 1993 SRC algorithm animation festival**. Report n.126. Digital Equipment Corporation, Systems Research Center. 1994.

[7] Brown, M.H. and J.R. Meehan, **The FormsVBT Reference Manual**. Unpublished. Digital Equipment Corporation, Systems Research Center. 1994.

[8] Brown, M.H. and M.A. Najork, **Distributed active objects**. Report n.141. Digital Equipment Corporation, Systems Research Center. 1996

[9] Cardelli, L., **A language with distributed scope**. *Computing Systems*, **8**(1), 27-59. MIT Press. 1995. Also: *Proc. 22nd Annual ACM Symposium on Principles of Programming Languages*, 286-297. 1995.

[10] Freeman, S.M.G. and M.S. Manasse, **Adding digital video to an object-oriented user interface toolkit**. *Proc. ECOOP'94*. Springer-Verlag. 1994.

[11] Najork, M, **Obliq-3D tutorial and reference manual**. Report 129. Digital Equipment Corporation, Systems Research Center. 1994.

[12] Najork, M. and M.H. Brown, **A library for visualizing combinatorial structures**. *Proc. IEEE Visualization'94*. 1994.

A Compute Server

A

```
var q = proc() end;
net_export("computeServer", Namer,
   { rexec => meth(s,p) q:=p; p() end }
```

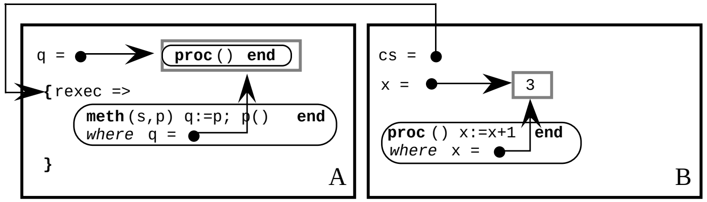
B

```
let cs = net_import("computeServer", Namer);
var x = 3;
cs.rexec(proc() x:=x+1 end);
x;   (* is now 4 *)
```

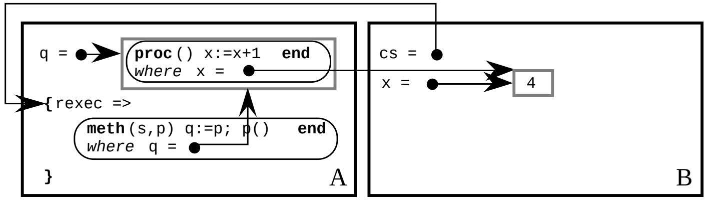
```
q;   (* is now proc() x:=x+1 end *)
q();
```

```
x;   (* is now 5 *)
```

Before the invocation:



After its completion:



CONCURRENCY IN OBLIQ

We need to deal with concurrency because it is implied by distributed objects.

Concurrency is based on the Modula-3 thread/mutex model.

It is enhanced with object-based “self serialization” (a way to avoid trivial deadlocks between sibling methods).

It is also enhanced with object-based conditional synchronization.

Self-inflicted Operations

Let *op*(o) be either a select, update, clone, or redirection operation. Then:

> "*op*(`o`)" is *self-inflicted*
> iff "`o`" is the same object as the self of the *current method* (if any).

Here, the *current method* is the last method that was invoked in the current thread of control and has not yet returned. Procedure calls do not change or mask the current method, even when they have not yet returned.

> Moreover, "*op*(`o`)" is *external* iff it is not self-inflicted.

Ex.: `meth(s) [s.x, s.x.y] end`

Here `s.x` is self inflicted, and `(s.x).y` is self-inflicted if `s.x` returns self.

N.B. it is possible to detect this condition at run-time with a simple test.

Protected Objects

It is useful to protect objects against certain external operations, to safeguard their internal invariants. Protection is particularly important, for example, to prevent clients from overriding methods of network services, or from cloning servers. Still, protected objects should be allowed to modify their own state.

A *protected* object is an object that is protected against external update, cloning, and redirection, but not against self-inflicted update, cloning, and redirection.

> **{ protected**, x_1 => a_1, ... , x_n => a_n **}**

Methods of a protected object can update sibling fields through self, but external operations cannot modify such fields.

Note that a solution to the protection problem based on “private” fields would not address protection against cloning and redirection.

Serialized Objects

- An Obliq server object can be accessed concurrently by multiple remote client threads. Moreover, local concurrent threads may be created explicitly. To prevent race conditions, it must be possible to control concurrent access to objects and other entities with state.
- We say that an object is *serialized* when
  - ~ (1) in presence of multiple threads, at most one method of the object can be executing at any given time, but we want to ensure that:
  - ~ (2) a method may call a sibling through self without deadlock.

  Note that requirement (2) does not contradict invariant (1), because an invocation through self suspends a method before activating a sibling.
- Solution: serialized objects have a hidden associated mutex, called the object mutex. An object mutex serializes the execution of field selection, method invocation, update, cloning, and redirection operations on its host object.

> External operations always acquire the mutex, and release it on completion.
> Self-inflicted operations never acquire the mutex of their object.

- Conditional synchronization can be applied to the implicit object mutexes. A new *condition* `c` can be created by `condition()`" and signaled by `signal(c)`. A special **`watch`** statement allows waiting on a condition in conjunction with the hidden mutex of an object. This statement must be used inside the methods of a serialized object; hence, it is always evaluated with the object mutex locked:

  watch *c* **until** *guard* **end**

- The **`watch`** statement evaluates the condition, and, if the guard evaluates to true, terminates leaving the mutex locked. If the guard is false, the object mutex is unlocked (so that other methods of the object can execute) and the thread waits for the condition to be signaled. When the condition is signaled, the object mutex is locked and the boolean guard is evaluated again, repeating the process.
- There is no automatic serialization for variables or arrays.
- The full Modula-3 thread interface is available for hand-crafting other synchronization mechanisms.

Obliq Concurrency Primitives

mutex()
lock a_1 **do** a_2 **end**
fork(a_1,a_2)
join(a)

condition()
signal(a)
broadcast(a)
wait(a_1,a_2)

watch a_1 **until** a_2 **end**

Here, "a_1" is a condition and "a_2" is a boolean expression. This statement waits for "a_2" to become true, and then terminates. Whenever "a_2" is found to be false, the statement waits for "a_1" to be signaled before trying again. The statement is equivalent to:

```
let x=$a_1$;
loop
   if $a_2$ then exit else wait(mu,x) end
end
```

where "*x*" does not occur in "a_2", and "*mu*" is the hidden mutex of the self "*s*" of the textually enclosing method. The "`watch`" operation must be *self-inflicted* with respect to such "*s*", if any, or an error is reported.

A Serialized Queue

```
let queue =
  (let nonEmpty = condition();
   var q = [];            (* the (hidden) queue data *)

   {protected, serialized,
      write =>
        meth(s, elem)
          q := q @ [elem];   (* append elem to tail *)
          signal(nonEmpty);  (* wake up readers *)
        end,
      read =>
        meth(s)
          watch nonEmpty     (* wait for writers *)
          until #(q)>0 end;  (* check size of queue *)
          let q0 = q[0];       (* get first elem *)
          q := q[1 for #(q)-1];(* remove from queue *)
          q0;                  (* return first elem *)
        end; });
```

Let us see how this queue can be used. Suppose a reader is activated first when the queue is still empty. To avoid an immediate deadlock, we fork a thread running a procedure that reads from the queue; this thread blocks on the **`watch`** statement. The reader thread is returned by the `fork` primitive, and bound to the identifier `t`:

```
let t =               (* fork a reader t, which blocks *)
  fork(proc() queue.read() end, 0);
```

Next we add an element to the queue, using the current thread as the writer thread. A non-empty condition is immediately signaled and, shortly thereafter, the reader thread returns the queue element.

```
queue.write(3);       (* cause t to read 3 *)
```

The reader thread has now finished running, but is not completely dead because it has not delivered its result. To obtain the result, the current thread is joined with the reader thread:

```
let result = join(t); (* get 3 from t *)
```

In general, **`join`** waits until the completion of a thread and returns its result.