[DigLib] Stanford Digital Library Project
Working Paper

Title: A proposal for basing our protocols on a general information exchange level
Author: Terry Winograd

Abstract: In order to build our protocols in a way that will provide for long term growth and extensibility, we should define a standard level on top of the CORBA/ILU level, to deal with the management of information across objects on different servers. It is based on a generalization of the protocols we have been working on.

Contents:

A proposal for basing our protocols on a general information exchange level

In order to build our protocols in a way that will provide for long term growth and extensibility, we should define a standard level on top of the CORBA/ILU level, to deal with the management of information across objects on different servers. It is based on a generalization of the protocols we have been working on. I don't know if such a level has already been proposed in the community developing object systems, and am sending this out in hopes that either we will find such a thing that is already well developed, or we can do so ourselves with a minimum of work. The mechanisms could be developed on top of ILU, by defining appropriate object types, with no change to the underlying system. They could be more efficient if supported directly within the ILU system, and we would want to discuss that potential further with Bill and others.

This proposal does not deal with access control. The assumption is that this can be dealt with at other levels (i.,e., in the programs written for objects using the primitives given here). If that isn't true, we need to think about how to incorporate it at this level. It does attempt to deal with issues of transaction integrity (thanks, Larry, for pointing out the need).

I. Introduction

The basic problem we need to deal with is that when objects are being used in a distributed way, there will often be cases where objects reside in multiple places that refer to the same thing but contain different information about it, and where it is not appropriate or efficient to simply replicate the information everywhere. This may be because of bandwidth limitations, or because the information is implicit in one object (can be computed when needed within its local context) and it would take substantial work to make it explicit so it could be replicated in an object at another site. This problem would not arise if we could realistically implement either of two alternatives:

1. Full availability
   There is never any need to replicate information, since any message that could be sent to an object at one site could simply be sent to the corresponding "home base" object that has all the data and context.
2. Full replication
   Information in an object at one site is always fully replicated in the corresponding objects at all sites, so that local access to any of these objects produces the same result.

There are a number of reasons why these idealized models can fail: limited bandwidth, implicit context that is costly to realize in a transportable form, the problem of multiple updates, the unreliability of servers, etc., Without going into these in detail I will make two claims. If you don't agree with these claims, the rest of the paper is probably useless. If you do, then we can work to get the details right.

Claim 1:
   It is not practical to base digital library protocols on idealized assumptions of availability or full replication, and it is therefore necessary to provide means for the developers of specific object classes to write explicit code that manages the distribution of information among objects across sites.

Claim 2:

It is possible to write some very general classes and methods that manage information flow, so that it does not need to be done from scratch on a class-by-class or object-by-object basis in the implementations of individual methods.

II. Basic Concepts

The proposal is based on four basic concepts: Counterpart objects, Attribute representation, Collections, and Flow-control objects. We will first give the concepts, then the more detailed definitions.

1. Counterpart objects

In the environment for interoperability in the digital library, we will find in general that a single conceptual "entity" (e.g., a document, a result set, a collection, a session, a translation process...) is represented by more than one ILU object. The simplest case is a multi-source search for documents, in which the same document is known by several sources, each of which has its own object for it.

There are two potential solutions to this problem. One is to have a privileged "object central" that assigns object identities in a way that guarantees unique mapping from its ILU objects to underlying entities. This is impractical for a variety of reasons (which I won't elaborate here). The other is to start from a fundamental assumption that there are two different levels of identity - identity of the underlying entities and identity of objects that represent them, with a many-to-one mapping (an entity can be represented by any number of objects). That is the path I propose.

For every entity in the world, there can be any number (one or more) of "counterpart objects" which are distinct in the OID world, but stand for the same entity. These will often (though not always) be at different sites, on different servers, etc. The proposal here is essentially for a way of managing information flow among the set of counterpart objects for an entity.

The counterpart relations could be stored by some centralized "counterpart database", which would have all the same problems as a centralized object catalog. Instead the proposal is to have each object maintain a list (which may be partial) of counterparts. These include actual ILU objects and "potential objects" that could be created on demand. There are no guarantees that the counterpart references are live (the objects may be gone or ungeneratable) or that they are symmetrical (object A may list B as a counterpart but not vice versa). This is managed by the programmer, using standard techniques for maintaining coherent references.

2. Attribute representation

One of the things that distinguishes an object from a record in a traditional programming language is that objects are active -- a method can trigger arbitrary processing, while a record-field access or assignment simply deals with the association of data to a field. As a classical case, an object can support a method "random" which simply returns a random result each time it is called. The "meaning" of the method is really the process for generating that result, not a field containing a value.

Without giving up this generality, we can note that many of the things we care about in many objects do behave more like traditional fields in records, and we can think of them as "attributes" that we can "get" and "set", rather than as methods that trigger computation. This enables us to replicate them without dealing with the generality of replicating execution context (which can be arbitrary) in order to get consistency across counterpart objects.

So the basic idea is to have standard operations that can access and set arbitrary attributes of an object, which are then mapped by the individual class definitions onto internal variables, methods, etc. From the point of view of this mechanism, objects are being treated as records with fields -- fields whose values can change over time, but which can be thought of as containing a representable value, not as doing an operation. The objects retain their full capability for arbitrary methods, but that is just not the part being dealt with by the information exchange management. Programmers need to understand the interactions between attributes and context in order to program the class implementations properly.

3. Collections

Many objects in a digital library environment will stand for collections of entities (e.g., a result set which stands for a collection of documents, a user group which stands for a collection of users, etc.). This could be represented in a straightforward way by having the collection object contain a sequence with an object for each element in it. However, for reasons of efficiency, this straightforward implementation may not be effective. If we generate a result set with 10,000 items, and ask for the authors and titles of the first 5, it will add a significant amount of overhead to generate 10,000 individual objects for the items, so we can represent the collection.

The members of a collection and information about them are often held implicitly in a context, and although they could be generated as needed, they are not realized initially.

The basic idea of the collection object is to provide a means for exchanging information about members of a collection without realizing objects for all of them. This is done by allowing the attribute communication methods work both in the individual case (they are the attributes for the objects exchanging messages) and the collection case (they are attributes for specified items in the collections represented by the objects exchanging messages).

As a side note, this use of the word "collection" has nothing to do with "garbage collection" and the "collectible" property of ILU objects. If there is too much potential for confusion we'll have to change the name.

4. Flow-control objects

In order to provide the needed generality for digital library protocols, the exchange of attribute information needs to be done in both synchronous and asynchronous fashion, with explicit flow control when needed. The protocol proposed here allows for a spectrum from simple synchronous transfer to arbitrarily designed flow control regimes. This is done in the protocol by providing for basic flow control primitives (transaction identities, time stamps, and sequence markers), and having an arbitrary object that serves as the flow control manager. By using "well-known" objects to stand for simple standard flow regimes (e.g., all-at-once and send-whenever-you-want), the simple cases can be managed without undue overhead.

III. Counterpart objects

We start by having a general class "DL-entity" for entities in the digital library. [Note: In this and the following, refer to the definitions at the end of the document]. It is a root of the class hierarchy for all the things we build, both information objects and activities. Each object can maintain a list of known counterparts. The first three methods in the class definitions are the obvious ones for managing a list.

Direct and indirect counterparts

The most straightforward kind of counterpart is another ILU object. The intended meaning is that the object in which it is listed refers to the same entity. The other kind of counterpart, an "indirect counterpart," is essentially a locator for a counterpart object. It specifies some object which is the "base object" and further data (the "cookie") that the base object can use to identify or generate an appropriate counterpart object. The "scheme" determines what scheme is intended for using the cookie. In some cases the cookie may be a simple identifier (e.g., the scheme indicates "ISBN" and the cookie is an ISBN number). In others it may be highly context dependent (the scheme is a special indicator known to the base object, and the cookie is a bundling up of previous context, so the context can be restored when the object is requested). No restrictions are placed on the number and type of schemes, or the contents of cookies.

The "Get-counterpart" method returns a real OID for the object referred to by an indirect counterpart. But this is often not necessary, since the attribute information operations are designed to let attribute data be transferred to and from indirect counterparts without actually instantiating them. This is done by having an additional argument in the data-exchange methods, specifying the object whose data is being accessed (which may not be the same as the one to which the message is sent).

IV. Attributes

Every object type can be implicitly understood as having an associated set of attribute tags. In this proposal there is no official mechanism for declaring those tags (though it might be good to add one). In that sense the attribute mechanism here is more like the "property list" mechanisms of LISP than like the record fields in a typed language.

The basis for conveying attribute data is the idea that a named attribute of a type can be encoded into a byte-sequence with a known encoding-scheme associated with that attribute name. If there are multiple encoding byte-sequences that might be used with a given conceptual attribute, these can be handled by having multiple attribute names (e.g., "Date-time-as-seconds" and "Date-time-as-standard-format-string"). It is up to the implementations of the relevant types to deal with the correspondences between these. The choice of mechanisms is based on several assumptions:

1. There is a relatively high overhead on individual object creation and cross-network object calls. Therefore, in the interests of efficiency it is necessary to be able to "batch transfer" multiple attributes for multiple objects in a single call, without necessarily instantiating all the objects
2. A substantial part of the data in attributes we deal with in the

digital library will be strings, or things easily convertible to and from byte-sequences without complex parsing.

3. For simplicity of mechanism it is better to have a uniform data-type for transfer and not get into complex type unions, discriminators (brands?), etc.

Attribute transmission

There are three sets of operations for attributes, each with a single-attribute version and a multiple-attribute version. All of them have a first argument which is a counterpart reference, indicating the holder of the attributes. In the case where the counterpart reference is direct, this will generally be the same object the message is being sent to (although there may be some other interesting cases). In the case where it is indirect, the base-object field will generally be the object the message is sent to, and the cookie and scheme are used by that basic object to get or set the attributes for the appropriate counterpart object.

Get-attribute(s)

The get-attribute and get-attributes methods specify the attribute(s) desired, which are returned synchronously by the call. In the case of multiple attributes, the attribute value list is in the same sequence order as the list of attribute tags in the call. Missing attributes will be indicated with explicit empty values, not by simply leaving out the pair. The definitions need to be extended with exceptions to handle the error cases (no such attribute, etc.), but I haven't attempted to do that yet.

Set-attribute(s)

The set-attribute and set-attributes methods specify the attribute(s) desired, the mode in which setting is done (either replace or add-to-sequence) and the new value(s). These are passed in parallel sequences with corresponding elements in the same position. Again, there is no strong type enforcement (e.g., you could pass lists of different lengths and ILU would allow it even though it is meaningless based on the semantics of attribute setting, and would produce an exception). The "source" argument will be discussed below when we describe flow control.

Attribute-request

This is a request to the object receiving it to send one or more corresponding set-attribute(s) message back to the original sender. This provides for asynchronous attribute changing. Rather than doing "get..." and dealing with the values returned, the originator does a request and goes on about its business. The recipient at some later point does one or more set-attribute(s) to provide the data. The details of flow control, transaction completion, etc. are discussed below, after dealing with collections.

A note on terminology: One interesting aspect of the natural language naming of object methods is that you can take the point of view of either the sender or receiver. For example "give-money" might be an appropriate name for two different methods. "Give-money 5" might mean "I'm giving you $5" (the sender is telling the receiver that a payment is being made) or "Give me $5" (a request to the receiver to make a payment). To be technical, this is the difference between a performative and an imperative reading of the verb.

In the case of "set" and "get" this isn't too bad, since there is a way in which we can think of the receiver as a "helper" in doing what the originator wants -- just like going up to a service window at a store and saying "get-shoes". This means "I want to get shoes" and "will you get some shoes for me", since the service-provider can be thought of as getting them from some third party (a storage place). So for those it doesn't come up. But when the action is explicitly something going on between the two parties, the perspectives are opposite.

The method for requesting that attribute values be sent asynchronously was originally called "send-attribute" since that's what the originator was asking the receiver to do. It was then changed to "request-attribute" since that's what the originator was doing, which seemed clearer. Finally, I decided to use a third-party nominalization (the message itself can be described as an "attribute-request") to avoid the confusion that results when someone reading the definitions makes the opposite assumption about perspectives. Has anyone dealt with this in other discussions of object-oriented programming?

In order to know whether a requested set of data has appeared, there needs to be some mechanism for keeping track of the overall transaction. ..to be filled in....

V. Collections

The additional methods for a collection exactly correspond to those for an object, except that they specify a range of objects in the collection and a corresponding sequence of data. For simplicity, there is only the multi-attribute version. This means that to get or set a single attribute for multiple objects, you use the multi-attribute form with an attribute list containing a single element. Note that the implementing object may choose to implement these by simply storing a table and retrieving from it, rather than parceling out the data to the objects. Underlying operations like adding items to a collection are not included since they may or may not be possible for various kinds of collections. All that is common is the ability to transfer attributes and find out the size.

All multi-object and multi-attribute sets are all-or-none -- either the whole things gets done, or nothing is done and an exception is raised.

VI. Flow control

The mechanisms support a wide variety of protocols for control of the data transmission:

1. Simple synchronous
   By using the "get" forms, the results are passed back using the standard result return from a method call.
2. Ask and get batch
   The originator sends a "request" to the counterpart, putting a unique identifier into the control argument (see below) and indicating that the results should come in a single chunk. It can then proceed asynchronously, maintaining in storage the identity of the request. When the counterpart does the corresponding "set", which includes all the requested data, it provides the identifier as part of the "source", so the originator can match it up with the request.
3. Ask and get incremental
   The originator sends a "request" to the counterpart, putting a unique identifier into the control argument (see below) and indicating that the results should come in any number of chunks. It can then proceed asynchronously, maintaining in storage the identity of the request. When the counterpart does each corresponding "set", which includes some part of the requested data, it provides the identifier as part of the "source", along with sequencing information, so the originator can match the incoming data up with the request and be sure it is all there. The implementation could choose to make internal changes on a bit-by-bit-basis, or stash away all the pieces until they are done and then enter them, in order to have a coherent transaction.
4. Fire and forget
   The originator does not keep a record of the request, and simply accepts incoming sets without transaction management.
5. Active flow control
   The flow control object is used in an active way to manage flow control

in traditional ways (buffer management, timeouts, liveness checks, etc.) This is all done by the definitions of the flow control objects, so is not specified at this level of the protocol.

The control and source objects

The additional arguments in a set and request use the indirect-counterpart structure to specify an object that could be used for flow control. In many cases the form of this structure provides the necessary information without actually instantiating an object. Here are some ways the different protocols above could be represented with conventions for using these objects:

1. Ask and get batch

   The control argument in the request specifies some "well-known" object (its OID is stable and known), with the scheme being "batch" and the cookie being any unique identifier the sender makes up (a timestamp would be a likely candidate).

   The "source" argument in the corresponding "set" (there is only one in this case) is the same structure, so the receiver can match it up with the request.
2. Ask and get incremental

   The control argument in the request specifies some "well-known" object (its OID is stable and known), with the scheme being "as-you-please" and the cookie being any unique identifier the sender makes up (a timestamp would be a likely candidate).

   The "source" argument in each of the corresponding "set" messages (there may be any number) replaces the cookie with a byte-sequence representing four elements: the request ID, an ID for this message (presumably another timestamp), a sequence number (to deal with lost message, restarts, etc.), and a Boolean indicating whether it is the last.

   Note that by encoding all this into a byte-sequence, we keep it in the conventions for a particular control type, rather than building it into the basic protocol (the definitions given here). It might be cleaner to add more arguments to the Set methods, if these are really the ones we want.
3. Fire and forget

   The control argument in the request specifies some "well-known" object (its OID is stable and known), with the scheme being "forget" and an empty cookie. The corresponding set message(s) also have empty cookies, and no control is done.

4. Active flow control

The object in the control argument is a real object, generated by the sender, with its own methods for communicating flow control in an active way. The "scheme" in the counterpart structure that is sent as an argument tells the receiver what kind of flow control object it is, and the code for responding is based on knowing and using its definition.

VII. Interface Definitions

I have not attempted to be complete in thinking about exceptions, garbage collection, brands, etc. This is the basis sketch to be filled out by someone who knows much more about ILU than I do.

```
INTERFACE Info-exchange

   TYPE DL-entity = OBJECT

       (* This is the root of the inheritance hierarchy for all objects
that have the distribution properties *)

    METHODS
         (* The first four methods deal directly with the counterparts.
* )
         Counterparts () : counterpart-ref-list,

          (* Returns a list of the counterparts.  This could be
elaborated with a second argument to allow returning only those
counterparts satisfying some filter, to avoid having to then go through
the list looking for appropriate ones.  For simplicity this version just
returns the whole list. *)
         Add-counterpart (ref: counterpart-ref),
         Remove-counterpart(ref: counterpart-ref),
        Get-counterpart(scheme: scheme-ref, cookie: cookie) : OBJECT,
         (* The remaining methods deal with the attributes *)
          Get-attribute (ref: counterpart-ref, attr: attribute-name) :
byte-sequence,
          Get-attributes (ref: counterpart-ref, attrs: attribute-name-
list)  :  byte-sequence-list,
          Set-attribute (ref: counterpart-ref, attr: attribute-name,
replace: BOOLEAN, value: byte-sequence, source: counterpart-ref),
```

Set-attributes (ref: counterpart-ref, attrs: attribute-name-list, replace: boolean-list, values: byte-sequence-list, source: counterpart-ref),

(* It is up to the object implementation to do the right thing for each entry in the list, which could involve invoking methods corresponding to each attribute, or could simply mean storing the list as is. *)

Attribute-request (ref: counterpart-ref, attr: attribute-name, replace: BOOLEAN, target: OBJECT, control: counterpart-ref),

Attributes-request (ref: counterpart-ref, attrs: attribute-name-list, replace: boolean-list, target: OBJECT, control: counterpart-ref)

END,

END;

TYPE counterpart-ref = UNION OBJECT, indirect-counterpart END;

TYPE indirect-counterpart = RECORD

base-object : OBJECT,

scheme : scheme-ref,

cookie : cookie-ref

END;

TYPE scheme-ref = byte-sequence

(* For the sake of simplicity the scheme is indicated by a byte-sequence. It could be handled by something more interesting such as a singleton object class (if I understand it properly) or enumerated type, or even an articulated structure (like the multilevel MIME-type structure). These would have implications for the distribution of the appropriate definitions, which I don't fully understand, so I have simply included the type scheme-ref as a place-holder currently defined trivially, but which might be defined differently. *)

TYPE cookie-ref = byte-sequence

(* The type of "cookie" really should be ANY since different schemes can make use of different data types here. I don't think ILU supports this, and it's easier to think about encoding arbitrary junk into byte-sequences rather than into integers or some kind of structure. As with the scheme, this type is a placeholder for doing it right. *)

TYPE counterpart-ref-list = SEQUENCE of counterpart-ref

TYPE attribute-name = byte-sequence

(* as with the others above, we may want to add structure here. *)

TYPE attribute-name-list : SEQUENCE of attribute-name

TYPE byte-sequence-list : SEQUENCE of byte-sequence

TYPE byte-sequence-list-list : SEQUENCE of SEQUENCE of byte-sequence

(* this might be better defined using the array types. I didn't know what the issues were with unspecified-length arrays, so left them as nested sequences *)

TYPE boolean-list : SEQUENCE of BOOLEAN

TYPE Collection = OBJECT

SUPERTYPES DL-entity

METHODS
Size () : integer
Multi-get-attributes (ref: counterpart-ref, first: integer, number-of-items: integer, attrs: attribute-name-list) : byte-sequence-list-list

(* The result is a sequence (one for each intended object) of sequences (one value for each attribute) of byte-sequences. *)
Multi-set-attributes (ref: counterpart-ref, first: integer, number-of-items: integer, attrs: attribute-name-list, replace: boolean-list, values: byte-sequence-list-list, source: counterpart-ref),

(* It is up to the object implementation to do the right thing for each entry in the list, which could involve invoking methods corresponding to each attribute, or could simply mean storing the list as is. *)
Multi-send-attributes (ref: counterpart-ref, first: integer,

number-of-items: integer, target: OBJECT attrs: attribute-name-list, replace: boolean-list, values: byte-sequence-list-list, source: counterpart-ref)

Total-items () : Integer

END;

Change history:
Modified June 27, 1995 by Terry Winograd
Changes:

* Translation to HTML
* Got rid of single-attribute forms to simplify the protocol. Once you're going through this overhead, the additional cost of having a singleton list instead of a single attribute is minimal.
* Replaced "String" by "byte-sequence" throughout
* Generalized the "base-object" to include counterpart refs, to allow multiple levels of indirection.

Original text version June 5, 1995 by Terry Winograd