[DigLib] Stanford Digital Library Project
Working Paper

Title: Lightweight objects for the digital library
Author: Terry Winograd
Abstract:

We have been looking at the potentials for integrating Xerox's GAIA system into the INFObus architecture. Ramana suggested we look at Xerox/Novell's Document Enhanced Networking (DEN) specification which incorporates some of the GAIA ideas in a product. I went through the spec, and had some realizations about what we are trying to do with the INFObus that I thought would be generally useful. The latter part of this message is a proposal for our own architecture.

Contents:

Lightweight objects for the digital library

We have been looking at the potentials for integrating Xerox's GAIA system into the INFObus architecture, but it still hasn't completed the external clearance process at Xerox. In the meantime, Ramana suggested we look at Xerox/Novell's Document Enhanced Networking (DEN) specification which incorporates some of the GAIA ideas in a product. (I have a copy I'll be glad to loan out, and you can get your own from Xerox, but I've lost the address - I'm sure Ramana can supply it).

I went through the spec, and had some realizations about what we are trying to do with the INFObus that I thought would be generally useful. The latter part of this message is a proposal for our own architecture, on which I would like your feedback. Since I don't know how close GAIA is to DEN, I'm not sure how much will be the same, and will look forward to finding out more about it and seeing how it fits these general considerations.

EXECUTIVE SUMMARY

1. DEN is a high-level distributed file system, with "heavyweight" objects and some useful generalizations.
2. We need a lightweight object system that builds and uses objects on the fly.
3. We should have as our rallying cry "UNIVERSAL LIGHTWEIGHT OBJECTIFICATION."
4. We should use three complementary development languages (all with ILU): Python, CommonLisp, and C.

1. WHAT IS DEN?

DEN has a number of good features, and is perhaps best thought of as a high-level distributed file system. It is high level in a number of useful ways:

1. The notion of a "document" can include an information content that brings together would now be several different files in different formats (e.g., an HTML page along with it's associated inline GIFs could be a single document)
2. The set of attributes that the file system maintains about a document is not limited to a small built in set ("name", "owner,", "creation-date", etc.) but is extensible in an open-ended way by declaration of arbitrary new attributes for use at an administrative site (which is called a "document store") in the distributed file system.
3. The standard system calls for locating files include a generalized Boolean search on the values of any of the attributes, with scope over one or more of the document stores.
4. The API for making file system calls is object-oriented (in spirit, though it is implemented in C) and cleanly structured.
5. There is support for a uniform and general set of transaction and locking primitives for shared documents
6. There is a general framework for specifying translators and having multiple conversions of documents available.

2. WHAT DO WE WANT IN AN OBJECT SYSTEM?

What I realized as I went through it was that there was a major difference in where they were positioning DEN in the tradeoff space, and what I thought the INFObus would be. In particular, creating or using a DEN document object is an operation of the same weight as the creation and use of a file in a current operating system (probably more, since it is doing more useful work than current file systems do, such as indexing the attributes). I had been envisioning scenarios that depended on very lightweight objects, e.g.:

1. A search interface sends a query to a PM (protocol machine) using ILU and specifying a source such as a WAIS source or database on Dialog (pick your favorite external search service here).
2. The PM queries the external service and returns to the search interface a list of 100 objects, each corresponding to one hit in the database. What it actually returns might be a pair consisting of the object ID plus document title for each.

The interface selectively uses methods associated with those objects to do its interesting work (e.g.., getting the author, abstract, full contents, or whatever, as it is needed) by sending ILU calls to the PM. In some cases the message can be responded to using data that was cached in the PM when the initial query results came back. In others it will have to make an additional call to the external service to get the information (e.g., an abstract or full contents.)

So the PM has created 100 objects for one query, has used almost no information from most of them, and throws them away when the next query comes along (or when the session is over). If we think of the equivalent of creating 100 files to do this, the overhead becomes substantial enough to worry about. My sense was that the overhead per-object should be more like what it takes to create a new object in an Object-oriented programming language such as Smalltalk or C++. I think this requires a somewhat different architecture and a lot of different design tradeoffs.

3. UNIVERSAL LIGHTWEIGHT OBJECTIFICATION

I summarize with a slogan about what our architecture needs to support:

* UNIVERSAL LIGHTWEIGHT OBJECTIFICATION

LIGHTWEIGHT means that the amount of work to create an object and access it should be data-structure-like, not file-like (this needs some more precise quantification).

UNIVERSAL means several things:

1. It should apply to all kinds of information objects -- anything we can encounter on the net. It will use hierarchies of class descriptions (our pool of models), some very specific to a type of object (e.g., a

geographic map), with lots of useful attributes and methods, and others extremely general, which can apply to anything that can be accessed at all.

2. It should apply at all levels of granularity. For example it should be possible on the fly to create objects for each labeled section of an SGML document, or each paragraph, drawing, etc. of a word-processing document. The DEN model assumes that a document is a fairly big deal, and its protocol has a special hack ("snippets") for returning the location of search hits in text without having to create a lot of little objects. We should do this in a more uniform way.
3. It should allow quick objectification of information items using their "native" model. For example, an INFObus interface to a raw unix file system should be able to produce with a minimum of work an object for each file or directory, based on a class definition whose methods and attributes correspond to the concepts in unix ("owner", "group", "symbolic link,"....). We need to support the use of a large number of pre-existing models like this, and then do the translation into common-denominator models on an as-needed basis using object-to-object translation with "lazy evaluation" (just-in-time conversion?).
4. It should enable quick communication with non-object based systems (e.g., translation from record-based to object-based entities as discussed in previous memos).
5. It should make as much use as possible of existing standards in widespread use (e.g., MIME, SQL, Z39.50) or at least match the conceptual models of those standards where appropriate.

OBJECTIFICATION includes the ability to produce an object from external data (e.g., a bibliographic record, raw file...) and produce an ILU interface to services that want to use it. There may also be persistent object-based stores, either for items fully in the object-based system, as with DEN, or for object-based headers for information stored in other forms (e.g., a bibliographic record for something stored on paper or in an external image base). But the default assumption is that objects are created as needed and not assumed to persist in an object form (i.e., the conceptual object, such as a document or map persists, but no object-system object stays around)

If we take this as our base, the "glue" has two components (hmm. just like epoxy...). One is the use of ILU as a communication protocol. The other is the shared use of a set of schemas for the object classes.

4. IMPLEMENTATION PLATFORMS

I propose that we support three different implementation vehicles for doing this, each with particular strengths:

1. A lightweight easy-development object system for doing quick sketches, special-case translators, etc. From my reading (no direct experience), Python sounds like the best bet here. It would be used for the equivalent of what people on the web now do in PERL or TCL, but its object-based style and an already-existing ILU implementation would make it fit in well. For example the unix-file-to-object translator I mentioned above might well be done in Python as an experimental test. I understand there is a Python/TK, which makes it good for doing "mini-interface" experiments as well.
2. A full-strength symbolic processing language. Given our background of experience and the work at Xerox, CommonLisp/CLOS seems the best bet. This would be used for services that had to do complex processing (I resist calling it "intelligent" but you know what I mean). For example a search processor that used heuristics to parcel search out to the most cost-effective sources and deal with all their different protocols might well use this (JANUS is an example). This vehicle would be specifically well suited to dealing with "dynamic schemas" where the program doesn't have specific classes wired in, but has to adapt to new classes, reading and use their definitions dynamically.
3. A common highly-portable, widely-familiar, efficiency-oriented language (is there more than one candidate at this point in computing history?) By using C (or C++?) we could produce components that were particularly efficient for schemas that are relatively fixed, and easily embed them in other programs (e.g., as part of some other interface or to modify an existing search service to produce its results in object form).

Since both Python and C cross all of the relevant platforms, this would leave us highly platform independent, except for the heavy-duty services. But since these are servers, not interface clients, they don't need to be as portable and distributable. I would expect things from both the Lisp and Python strands to sometimes get converted to C when we want to make them easily exportable for other people to use and test.

It isn't clear just where GAIA fits into all of this - to the extent that it provides a valuable service with an object-oriented face, we can make good use of it. To the extent that it has developed specific object classes and schemas for them, we can adopt those, not as THE object definitions in the INFObus, but as a particularly useful set for intertranslating. To the extent it has protocols for specific services (e.g., resource-bounded search across a set of sources) we could well use those as our "preferred full-octane" ones, while supporting smaller and simpler ones as well (we need "lightweight services" as well as "lightweight objects").

I'd be interested in reactions to all this and will be eager to learn more.

--t

Change history:
Converted to HTML by Terry Winograd June 24, 1995
Original version January 2, 1995 by Terry Winograd