The Stanford InfoBus and Its Service Layers

Augmenting the Internet with Higher-Level Information Management Protocols

Martin Röscheisen, Michelle Baldonado, Kevin Chang, Luis Gravano, Steven Ketchpel, Andreas Paepcke

http://www-diglib.stanford.edu

Stanford Digital Libraries Project
Computer Science Department
Stanford University, CA 94305

The Stanford InfoBus is a prototype infrastructure developed as part of the Stanford Digital Libraries Project to extend the current Internet protocols with a suite of higher-level information management protocols. This paper surveys the five service layers provided by the Stanford InfoBus: protocols for managing items and collections (DLIOP), metadata (SMA), search (STARTS), payment (UPAI), and rights and obligations (FIRM).

Keywords: Internet protocols; middleware; digital libraries; interoperability; heterogeneous, networked environments; information access; metadata, payment, rights management; Java/CORBA.

1.0 Introduction

The Stanford Digital Libraries project is one of the six participants in the 4-year, $24 million US "Digital Library Initiative," started in 1994 and supported by NSF, DARPA, and NASA together with a set of industrial partners. Rather than addressing issues related to a specific content collection, the Stanford project focuses on developing a set of service protocols by which different information resources and services can be "glued" together.[1] This set of service protocols is referred to collectively as the "Stanford InfoBus." The InfoBus is a prototype infrastructure that is designed to provide a bottom-up way of extending the current Internet protocols with a suite of higher-level information management protocols that the Internet currently still lacks.

In this paper, we survey the design of the Stanford InfoBus and its five service layers: protocols for managing items and collections (DLIOP), metadata (SMA), search (STARTS), payment (UPAI), and rights and obligations (FIRM).

1. In other words, despite its possibly misleading name, the "Stanford Digital Libraries Project" is not about any specific "digital library." (The testbed of the project relies on content collections and services maintained by industrial partners and on the information and services on the Web.)

Also part of the Stanford project but not the focus of this paper is work that leverages the InfoBus middleware to experiment with novel user interfaces (DLITE [6], SenseMaker [2]) and services (GlOSS [8], SCAM [13], Fab [12], InterBib [11]).

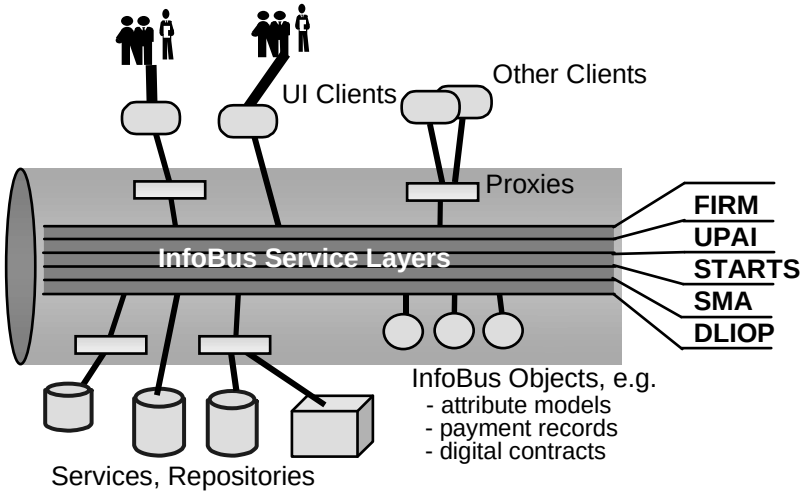
2.0 The Stanford InfoBus Architecture

The Stanford InfoBus is an architecture that gives clients uniform access to distributed, heterogeneous information resources and services. A collection or a service is "on" the InfoBus if it either uses one of the InfoBus protocols natively or if there exists an *InfoBus proxy* for this service.

InfoBus proxies are wrappers that can be operated either by a service itself or by some other service provider (e.g., a "digital library"). This flexibility makes it possible to enrich the information infrastructure and to achieve interoperability in a bottom-up way: It is not "imposed" (top-down) that services directly follow the InfoBus protocols; any third party can "plug" its favorite services into the InfoBus. For example, InfoBus clients can easily make use of the power of standard Web search engines such as Digital's AltaVista as soon as someone implements an InfoBus wrapper to AltaVista and publishes its availability. However, it is not required that this wrapper necessarily be run by Digital itself; it can be autonomously maintained by third parties (with the use of proper rights management).

Note that it is of course usually advantageous for a service provider to be also directly the operator of a corresponding InfoBus proxy. This allows them to make sure that the proxy implementation exploits the service's native features in the best possible way (making the service "look good" in the uniform InfoBus world); it allows them to guarantee quality of service, etc. In other words, there is a built-in incentive (a social push) for proxies to be moved out further to the sources–thus effectively leading to the adoption of the InfoBus protocols without the necessity to formally standardize on them.

FIGURE 1. The Stanford InfoBus and Its Service Layers: Clients and services access InfoBus services either directly or via InfoBus-compliant proxies. The InfoBus provides services for managing items and collections (DLIOP), metadata (SMA), search (STARTS), payment (UPAI), and rights and obligations (FIRM).



The Stanford InfoBus is designed to make it easy to provide such wrappers for an increasing number of clients and services. Ideally, clients and services would freely

interact, simply by plugging themselves into 'the system', just as hardware devices can be plugged into a bus. We therefore call our vision the *InfoBus.* (Cf. Figure 1.)

Clients use InfoBus resources to accomplish complex tasks such as writing papers, publishing newsletters, or tracking the development of industrial product lines. Information resources include information repositories such as online catalogs, stock feeds, census data, newspapers, and so on. Resources also include services that operate on information such as document translation, document summarization, remote indexing, or payment services and copyright clearance.

At an implementation level, the architecture of the InfoBus clearly melds well with technologies such as CORBA and DCOM that support object-oriented design in a distributed environment (although there is no reason why one could not implement the InfoBus protocols, say, on top of HTTP–our specifications are independent of any such implementation choices).

In our testbed, we have experimented with the use of CORBA as a prototyping environment for developing the InfoBus. Specifically, we have used the CORBA implementation of one of our industrial partners: Xerox PARC's ILU system [29]. Each participating entity (clients, information resources, documents, attribute models, contracts, etc.) is then either directly implemented as an object, or an object is created to act as a *proxy* to the entity. These objects can be placed on any machine on the network and can be accessed remotely from anywhere. The CORBA infrastructure makes it possible to implement proxies on different platforms and in different languages. In our case, we have used a mixture of Python, Java, and C++ to implement the various objects of the infrastructure.

InfoBus proxies communicate with the services they represent via the native service access protocols such as telnet, Z39.50, or HTTP. InfoBus clients that are interacting with the proxies do not need to be aware of these differences; they can simply use protocol calls to access the proxies. Examples of services that we have linked up to the InfoBus by wrapping them with proxy objects include Knight-Ridder's Dialog Information Service, Web search engines, automatic document summarizers, bibliography maintenance tools, OCR services, and others. InfoBus services also take care of any remaining translation tasks (e.g. translation of queries). For examples of other InfoBus services, see [3], [5]. More detail about the InfoBus and its implementation can also be found in [1].

In the process of designing the InfoBus protocols, we have considered a number of guidelines that have been broadly applicable to several different parts of the project:

- *Keep the common case fast and simple*. One of the potential pitfalls in designing higher-level protocols is that generality is achieved at the expense of making previously simple cases more complicated. In our designs, we have taken care to keep simple cases simple, and to use higher-level protocols to offer conceptual unification and a way of uniformly covering more sophisticated cases in addition to simpler cases.

- *Build in extensibility.* While designing a protocol, it is useful to think about how its evolution will be managed over time, and how this process can be facilitated at the outset. By explicitly considering sources of possible change and by modularizing the architecture in a way that reflects these forces, we can incorporate the evolution process into the technical infrastructure (and minimize the amount of coordination that change would entail).

- *Provide toolkits in addition to services.* A corollary of extensibility is that rather than building a solution, it is often more useful to build toolkits that allow qualified other people (e.g. librarians) to build solutions for specific circumstances.

- *Design for interoperability beyond the "least common denominator."* The commonalities among different native protocols are often not sufficiently rich to accommodate usages that users might care about in certain situations. We have generally ensured that it is possible for sophisticated clients to gain access to more advanced features in a structured way.

- *Design for resource unavailability.* In a widely distributed system, there will always be some part of the system that is unavailable or vastly slower. The protocols need to be able to handle such failures gracefully, even in cases where the missing resource (information source, printer, network) provides no helpful diagnostics or recovery.

- *Do not limit implementors unnecessarily.* Implementors of specific applications often have a better sense of the trade-offs that exist for their designs than it is possible for protocol designers to have. It is therefore useful to give implementors as much flexibility in their choices as possible given a protocol of a certain complexity.

In the following, we survey each of the five service layers of the Stanford InfoBus. In Section 3, we describe the DLIOP service layer for managing items and collections in a networked environment. In Section 4, we describe our metadata protocol. In Sections 5, 6, and 7, we then describe the Stanford InfoBus service layers for managing search (STARTS), payment (UPAI), and rights and obligations (FIRM), respectively.

3.0 DLIOP: Managing Items and Collections

The Digital Library Interoperability Protocol (DLIOP) is a protocol layer that provides basic services for managing *items* (e.g. documents) and *collections* in a networked environment. It allows InfoBus clients to communicate with information repositories, and to request and receive asynchronously information from InfoBus proxies. Proxies obtain this information natively from the information services they represent. Cf. Figure 2 for an example of a simple arrangement.

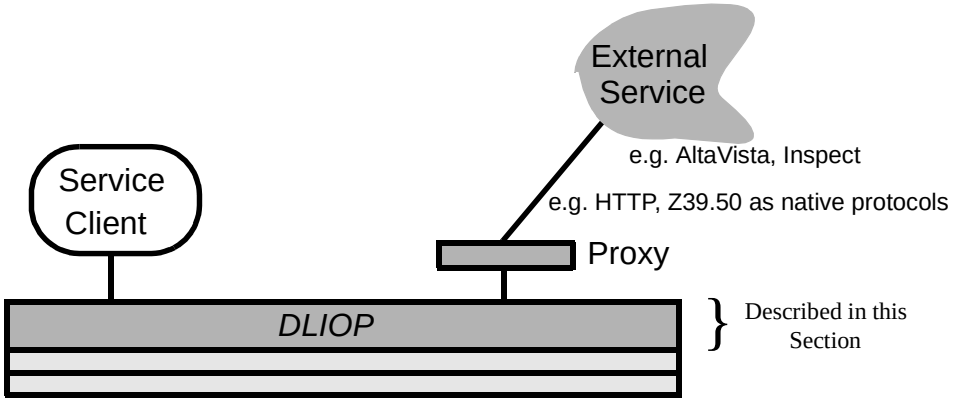
DLIOP combines advantages of stateless protocols such as HTTP (e.g., optimization for short-lived resource usage) with the advantages of session-based protocols such as Z39.50 (e.g. interaction efficiencies for multiple, related interactions) [27]. The protocol provides a testbed for experimenting with dynamic resource reallocation to achieve load balancing. It allows documents and computation to be moved among machines while interactions between clients and services are in progress. Furthermore, DLIOP enables experimentation with different caching strategies in contexts where clients sometimes request follow-up information quickly and at other times only after several days have elapsed.

The DLIOP protocol specifies methods invoked among two basic kinds of components: *Clients* and *collections* of *items*. "Clients" are any kind of application programs that want to access an information collection. "Items" are the most general form of an InfoBus information object; subclasses of items include documents, services, attribute models, rights, contracts, etc. "Collections" are objects (in the sense of object-oriented programming) that manage a set of items.

The basic goal of DLIOP is to provide a flexible transport mechanism by which collections of items at the server end can be made available at the client side without the overhead of having a client request each item individually across the network whenever an

item is needed. DLIOP facilitates the interaction between a client and an external service proxy by managing the way in which proxies to external services can return collections of items to an InfoBus client in response to a service request (e.g. a search query). In this section, we outline key aspects of the DLIOP service layer; additional motivation, comparisons to other protocols, and explanations can be found in [1].

FIGURE 2. The DLIOP Service Layer



3.1 Collections of Items

Collections are container objects with a simple interface. It includes methods such as ***`GetTotalItems()`***, which returns the number of items in the collection, or ***`AddItems()`***, which inserts new items into a collection if possible. An important subclass of collections is *constrainable collections*. Constrainable collections can be asked to produce a result collection that contains a subset of the items that they currently hold.

We use such constrainable collections to implement InfoBus proxies to information repositories: While behaving like a true collection to clients, such collections can be "virtual" in that they do not necessarily need to hold all of the items of a repository directly, but they only materialize them on an as-needed basis. When such a collection receives a constrain request, it issues a query to the external information source, and retrieves result information from it. This information from the native source will usually not be available in the InfoBus item format, but the proxy will make it available via this uniform interface. In Figure 3, the proxy representing the external service is thus implemented as a special kind of collection object, a constrainable collection which is a collection that may be queried.

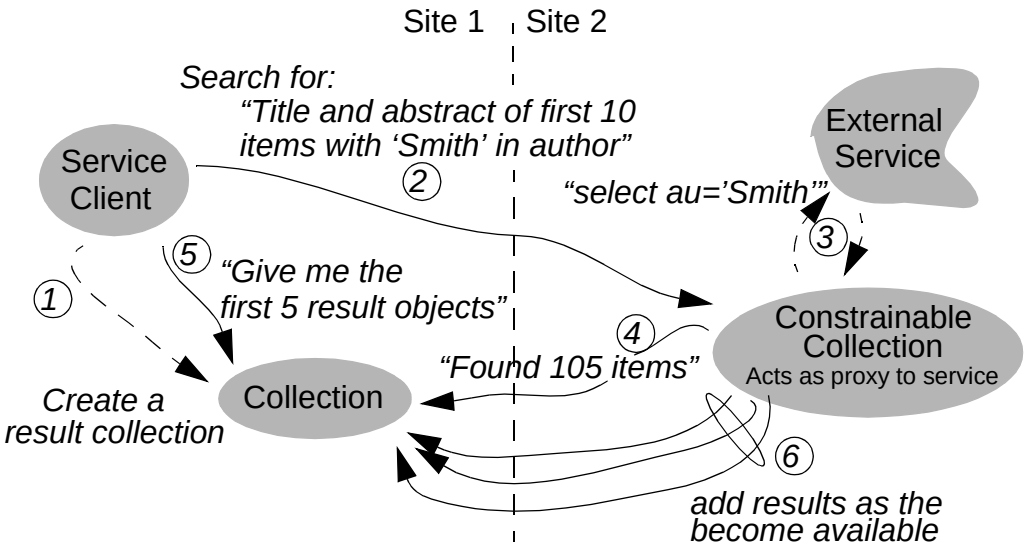
3.2 Basic Search Interaction

Figure 3 shows an example of a basic search interaction. The client wishes to search over some external service for which a constrainable collection object acts as a proxy. Note that while the steps in the figure are numbered for reference, most of them occur asynchronously. Arrows in Figure 3 represent method calls.

As a first step, the client creates a local collection object for holding the results. Then the client issues a query to the constrainable collection that acts as the external service's proxy. The query contains information on which objects are desired (authors must contain the word 'Smith'). It also contains instructions on how many results should be returned as soon as they become available (10), and on which parts of those results should be included (only abstract and title of each result). In addition, the query request

includes a pointer to the client's local result collection. The proxy collection will use this pointer to deliver results.

FIGURE 3. DLIOP: A Basic Search Interaction.



Once the query has been delivered, the client is free to perform other work while the query is processed: the query call to the proxy collection is asynchronous. Alternatively, the client may immediately ask its local result collection for the total number of results, and/or for some or all of the result objects. These calls could block until the required information is actually available.

Meanwhile, the proxy collection delivers the query to the external service by whatever means are appropriate: HTTP, Z39.50, a telnet connection, its local file system. As soon as the proxy collection knows how many hits to expect, it notifies the client's result collection. In (possibly) multiple calls to the client's collection object, the proxy collection subsequently delivers results. Each call (step 6) to the client's collection delivers some number of title/abstract values as lists, each list being the desired (title/abstract) excerpt from one result. The client collection creates a local object for each result, filling its title and abstract properties with the corresponding values. It is up to the proxy collection to decide whether to wait for all 10 results to arrive from the external service before delivering them to the client's result collection, or whether to collect a few and deliver them as early as possible.

3.3 Getting More Result Objects for the Same Query

At some point, the client will have received and examined the initial 10 results from its original request. Whenever the service client requests more results from its local result collection than are currently scheduled to arrive there, the result collection contacts the proxy collection at the server side for more of the hits. These are then delivered just like the original results. Notice that in order to do this, the proxy collection needs to let the client's result collection know the proxy collection's object ID. This is done as one of the parameters in each of the (partial) result deliveries.

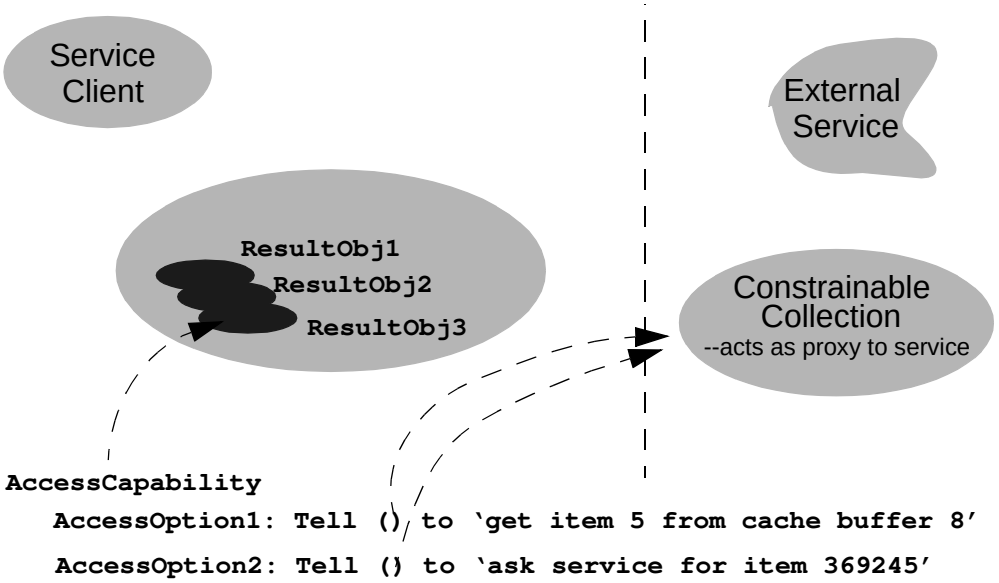
3.4 Freeing Proxy Collection Resources

The client may take a long time to ask its local result collection for more result objects, or for more properties of the result objects already partially retrieved. What if the proxy collection does not want to keep the resources associated with a given query indefinitely? It may want to use these resources for other requests, or the external service could be a for-pay service with high connect-time fees. The DLIOP allows the proxy collection to discard its resources at any time, although of course the intent is to service already pending requests before shutting down.

Allowing proxy collections to shut down the resources associated with a particular query raises two issues. What does a result object at the client result collection do if it is asked for properties it did not yet pull over from the proxy collection? And what does the client's result collection do if the client asks it for more hits for the same query? The first issue is handled by what we call *access capabilities*.

Each result object contains an access capability that contains the information needed to obtain values for the object's properties. The access capability for each result object is passed as a parameter from the proxy collection to the client's result collection when results are delivered. An access capability in turn may contain multiple *access options*, which each represent one way of getting the property values. When a result object needs to retrieve additional properties for itself, it initially tries to request them through the first access option in its access capability. If that fails, it tries the next one, and so on. Refer to Figure 4 for an informal illustration of access capabilities.

FIGURE 4. DLIOP: Access Capabilities Provide Flexibility for Resource Management



Each access option contains the object identifier (OID) of an object that can provide the additional properties, and a cookie to pass along with the request when contacting that object[2]. For example, the first access option may contain the OID of the proxy collection. The associated cookie is interpreted as the identifier for the in-memory results set from which the requested properties will come. If the proxy collection has already dis-

2. A cookie is a data structure that is passed uninterpreted to its final destination. Only that final destination knows how to interpret and use the data structure.

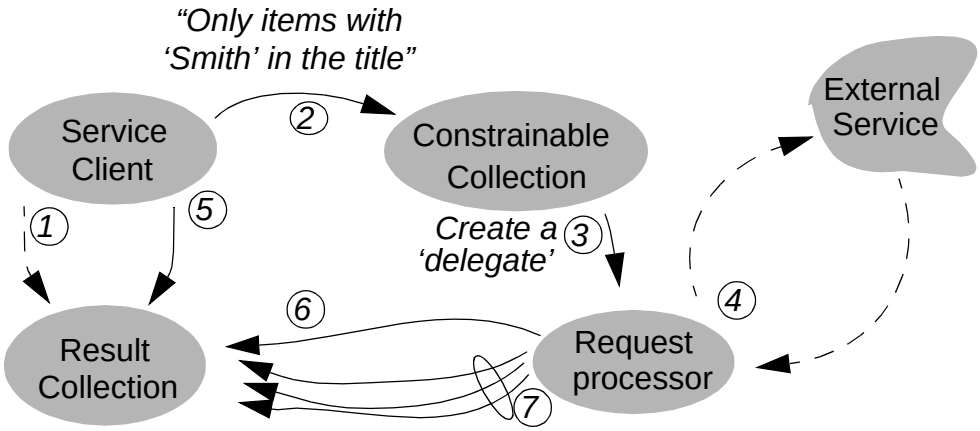
carded this result set, it will raise an error in response to the request for property values. The client's result object then attempts this operation again, this time with the second access option of its access capability. The OID may again be the proxy collection. But now the cookie will contain an indication to the proxy collection that a new query is to be performed, with the required information being retrieved (again) from the external service. This is, of course, more expensive than if the first access option had worked, but it allows the DLIOP to avoid sessions that tie up the server side indefinitely.

The second issue, of client collections being asked for more hits after the proxy collection has discarded its resources associated with the query, is solved similarly. Every time new results are added to the client's collection, a 'moreCookie' is passed along. Analogous to access capabilities for individual result objects, this data structure provides the client collection with one or more contacts for requesting additional hits.

3.5 Load Balancing at the Server Side

Sometimes it may be desirable to free the object that acts as proxy to the external service from handling new requests at the same time it is processing a pending request. Figure 5 shows how this requirement can be accommodated.

FIGURE 5. DLIOP: A Search Interaction with Server-side Off-Loading (Interactions 1, 2, 4, 5, 6, and 7 are as in Figure 2)



Instead of interacting with the external service itself, the constrainable collection immediately creates a 'delegate' object which takes over the remaining processing of the query request. It then returns to accepting more requests. Note that the request processor object created as the delegate may reside on a machine other than the one running the constrainable collection. The client is unaware of the origin of calls in steps 6 and 7.

3.6 Load Balancing During Result Delivery

The DLIOP allows server-side objects to perform load balancing even while results are being delivered to the client. This is easy because, as pointed out in Section 3.3, the OID of the proxy collection is passed to the client's result collection every time a set of results are delivered. Before delegating further work to a new object, the proxy collection can issue a call to the client's result collection, adding an empty set of results, and specifying a new OID as the target for future requests for additional result hits.

3.7 Discussion

The DLIOP protocol is very carefully designed to leave as many decisions as possible to implementors. If the external service is session-based, the proxy may maintain sessions for as long or short a time as the implementor deems appropriate. Similarly, the location and lifetime of caches are not prescribed by the protocol. For example, any given implementation may choose to cache large numbers of result objects at the proxy. Alternatively, the objects could be moved to the client-side result collection as quickly as possible. The correct strategy depends on usage patterns, resource considerations at the proxy, and the typical size of result objects. The implementor makes the decision, not the protocol.

The protocol is modular in that it allows the overall system to be partitioned into independently implemented and maintained units. Clients can be constructed independently from proxies. As proxies are added, clients can begin to take advantage of them right away. Conversely, proxies may be developed without knowledge of the current or future client population.

The protocol fares less well for the guideline of ‘making simple operations very simple to perform and implement’. We do pay overhead for the protocol’s ability to perform load balancing. This is best illustrated with the example of our ‘moreCookie’. In order for the proxy to have the freedom to delegate work at any time, the client does have to check the most recent ‘OID for future interactions’ before issuing follow-up requests to the proxy. Recall that the proxy passes this OID during each call to the client. While this is easy to implement, it is a complication that would not exist if the protocol did not have its advanced capabilities.

3.8 Implementation Status

A CORBA-based implementation was completed early in the project and has since then provided the foundation for many other protocols and services. DLIOP has also been used to interoperate with collections developed by other DLI digital library projects.

4.0 SMA: Managing Metadata

The Stanford Digital Library Metadata Architecture (SMA) defines a service layer for the uniform exchange and management of the metadata necessary for finding InfoBus services, for querying these services, and for interpreting the structured results returned by these services. Before this architecture was put in place, each InfoBus service handled metadata in its own ad hoc way. This led to incompatibilities at the user level (a query constructor might ask a user to fill in an “author” field, while a result analysis tool might refer to an author as a “creator”) and at the infrastructure level (query constructors and query translators must communicate with each other about field information).

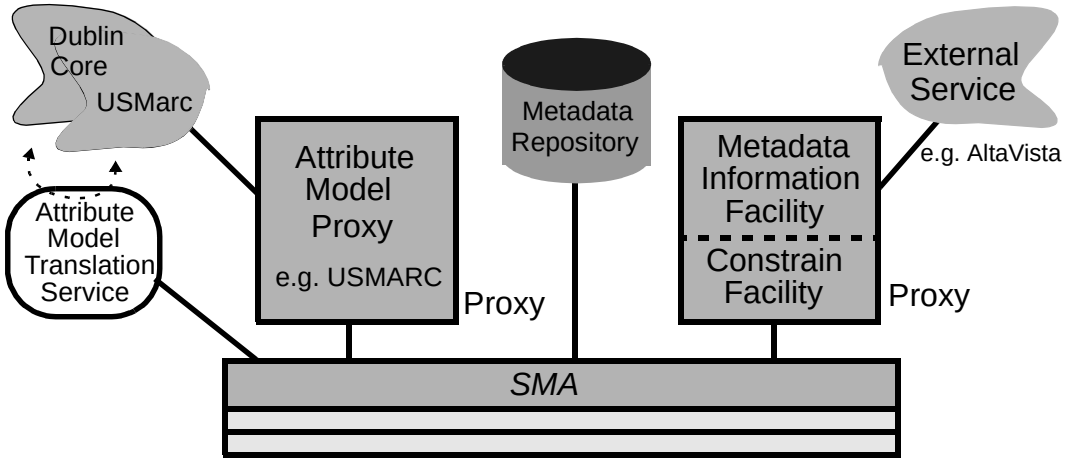
In particular, the SMA layer is concerned with the interoperability of two types of metadata: (1) metadata that describes the information objects (e.g., textual documents) available through search services, and (2) metadata that describes the services themselves. A more in-depth characterization of the metadata needs addressed by this architecture can be found in [3]; further discussion of the design rationale for this architecture can be found in [4]. Note that the following descriptions are geared towards search services but the design readily generalizes to other usages as well.

To facilitate metadata compatibility and interoperability, the Stanford Metadata Architecture (SMA) includes four basic component classes: attribute model proxies, attribute model translators, metadata facilities for search proxies, and metadata repositories.

Attribute model proxies elevate both attribute sets and the attributes they define to first-class objects. By this, we mean that attribute sets and attributes have computational representations through which they can describe themselves and interact with other components of the architecture. They also allow relationships among attributes to be captured. *Attribute model translators* map attributes and attribute values from one attribute model to another (where possible). *Metadata facilities* for search proxies provide structured descriptions both of the collections to which the search proxies provide access and of the search capabilities of the proxies. Finally, *metadata repositories* accumulate selected metadata from local instances of the other three component classes in order to facilitate global metadata queries and local metadata caching.

Figure 6 shows the current component classes, communication pathways, and InfoBus connections. We describe the specification of each component class and its associated communication pathways in turn.

FIGURE 6. The Stanford Metadata Service Layer



4.1 Attribute Model Proxies

Structured descriptions of resources and services are built out of *attributes* and *attribute values*. One level of aggregation beyond the individual attribute is the *attribute model*–a self-contained collection of attributes. Well-known attribute models include the USMARC set of bibliographic attributes (referred to as “fields” in the USMARC community) [22], the Dublin Core set of attributes [24], and so on.

In our metadata architecture, we reify both attributes and their encompassing attribute models as first-class objects. Attributes are instances of class `AttributeItem`. *Attribute model proxies* are implemented as InfoBus collections. Attribute model proxies represent real world attribute models, just as search proxies represent real world search services.

An `AttributeItem`’s properties include the following: model name, attribute name, aliases usable for queries, value type, documentation, various other information used by query translators, and so on.

The model name and attribute name are both strings that serve to identify the `AttributeItem` uniquely. As an example, an attribute might have the model name “Dublin Core’’ and the attribute name “Title.’’ The model name is repeated in all items

to make them self-contained. This is important when the items are passed around the system to components other than the "home'' attribute model proxy. When examining an `AttributeItem`, a client can always determine to which attribute model it belongs.

The attribute value type information in an `AttributeItem` dictates the data type that can be contained in fields described by the `AttributeItem`. We use the interface definition language (IDL) that is part of our CORBA implementation to specify these types. It is up to each search service proxy to ensure that the values it returns conform to these type specifications. If the external service that the proxy represents natively returns a different type, then the proxy is expected to transform the value into the specified type before returning it.

Attribute model proxies make attribute models first-class objects in our computational environment. They allow us to store and search over attribute-specific information that is independent of the capabilities possessed by any particular search proxy. Since an attribute model proxy is a collection, it is accessible via the same interface as all other search service proxies. In other words, the attribute model proxy has a search method that responds to a query by returning the appropriate subset of the included `AttributeItem`. Furthermore, attribute model proxies record what relationships hold among the included attributes. We currently record "is_a" and "part_of" relationships. This is important for some services, such as sophisticated user interface services, that require *structured* attribute models.

A search service proxy that supports such an attribute model could use this relationship information when processing queries. For example, a search service proxy might determine that items match the query `Creator contains Ullman` if they contain "Ullman'' in the value of the Creator attribute or if they contain "Ullman'' in the value of descendant attributes (e.g., in the value of Reporter or Author).

4.2 Attribute Model Translators

In heterogeneous environments, many different attribute models co-exist. This inevitably leads to mismatches when InfoBus components that support different attribute models attempt to communicate with each other. For example, consider a bibliographic database proxy and a client of that proxy. The bibliographic database proxy might support only the Dublin Core attribute model, while the client might support only the USMARC bibliographic data attribute model. In order for this client and this proxy to communicate with each other, they must be able to translate from USMARC attributes to Dublin Core attributes and vice versa. In other words, they require intermediate *attribute model translators*. Attribute model translators serve to mediate among the different metadata conventions that are represented by the attribute model proxies. These translation services, available via remote method calls, translate attributes and their values from one attribute model into attributes from a second attribute model.

Of course, in some cases, translation may not be possible: consider translating from an attribute model designed for chemistry databases into an attribute model designed for ancient Greek texts. Similarly, Dublin Core is a much smaller set than USMARC, so some information that can be tagged in USMARC finds no equivalence in Dublin Core.

Even when translation is appropriate, the translation from one attribute model to another is often difficult and lossy. For example, the Dublin Core describes authorship using the single attribute Author. However, USMARC distinguishes among several different types of authors, including Corporate Author (recorded in the 100 attribute) vs. Individual Author (recorded in the 110 attribute). When translating an `AttributeItem` from Dublin Core to USMARC, a decision must be made whether to translate the Dublin Core

Author attribute value into a USMARC 100 attribute value or a USMARC 110 attribute value. This may be hard-coded, or the translation may be performed heuristically and may take into account the other attribute values present in the item being translated.

Translation services do more than map source attributes onto target attributes. They must also convert each attribute value from the data type specified for the source attribute into the data type specified for the target attribute. This conversion can be quite complex. For example, one attribute model might call for authors to be represented as lists of records, where each record contains fields for first name, last name, and author address. Another model might call for just a comma-separated string of authors in last-name plus initials format. When translating among these values, some information may again be lost if, for example, the address is simply discarded.

Our attribute model translation services may be accessed by a variety of other InfoBus components, including search service proxies. For example, a search service proxy might choose to use attribute model translators to be attractive to more clients, because it can then advertise that it deals in multiple attribute models. On the other hand, clients might use attribute model translators in order to ensure their ability to communicate with a wide variety of search services.

4.3 Metadata Facilities for Search Proxies

The *metadata facility* that we attach to each search service proxy is responsible for exporting metadata about the proxy as a whole, as well as for exporting metadata about the collections to which it provides access. Collection metadata includes descriptions of the collection, declarations as to what attribute models are supported, information about the collection's query facilities, and the statistical information necessary for resource discovery services like GlOSS [7] to predict the collection's relevance for a particular query. Clients can use the information to determine how best to access the collection maintained by the search service (i.e., what capabilities the search service supports).

We have decided to make the interface for accessing the metadata facility of search service proxies very simple in order to encourage proxy writers to provide this information. Search service proxy metadata is accessed via the `getMetadata()` method, which returns two metadata objects. Alternatively, each proxy may opt to "push" these metadata objects to its clients. The first metadata object contains the general service information, and it is based heavily on the metadata objects defined by STARTS (cf. Section 5). The general service information includes human-readable information about the collection, as well as information that is used by our query translation facility. Examples for the latter are the type of truncation that is applied to query terms, and the list of stopwords. Our current query translation engine is driven by local tables containing this kind of information about target sources.

An interesting attribute of our first metadata object is contentSummaryLinkage. The value for this attribute is a URL that points to a *content summary* of the collection. Content summaries are potentially large, hence our decision to make them retrievable using ftp, for example, instead of using our protocol. The content summary follows the STARTS content summaries, and consists of the information that a resource-discovery service like GlOSS [7] needs. Content summaries are formatted as Harvest SOIFs [24].

The second metadata object returned by the `getMetadata()` method contains attribute access characteristics. This is attribute-specific information for each attribute that the proxy supports. Recall that attribute model proxies contain only information that is independent from any particular search services. Attribute access characteristics complement this information in that they add the service-specific details for each attribute.

For example, some search services allow only phrase searching over their Author attributes, while others allow keyword searching. Similarly, some search services may index their Publication attributes, while others may not. The attribute access characteristics describe this information for each attribute supported by the collection specified in the `getMetadata()` call. The query translation services need this information to submit the right queries to the collections.

Notice that this design does not allow clients to query search service proxies directly for their metadata. Search service proxies only export all their metadata in one structured "blob.'' The reason for this is that the InfoBus includes many proxies, and more are being constructed as our testbed evolves. We therefore want proxies to be as lightweight as possible. Querying over collection-related metadata as exported by search service proxies is instead available through a special component, the metadata repository.

4.4 Metadata Repositories

Metadata repositories are local, possibly replicated databases that cache information from selected attribute model proxies, attribute model translators, metadata facilities for search proxies, and other InfoBus services in order to produce one-stop-shopping locations for locally valuable metadata. We allow for metadata repositories to pull metadata from the various facilities, as well as for the facilities to push their metadata to one or more repositories directly. The intent is for these repositories to be a local resource for finding answers to metadata-related questions and for finding specialized metadata resources. A metadata repository has a search-service-proxy interface, and includes:

- *The `AttributeItem` from locally relevant attribute model proxies*. This data is useful, for example, to search for attribute models that contain concepts of latitude and longitude.
- *Translator information for locally relevant attribute models*. This data is useful, for example, to search for translators to or from particular models. Searches return pointers to the translator components.
- *General service information for locally relevant search service proxies*. This data is obtained through `getMetadata()` calls on the proxies, as discussed in the previous section, and is useful, for example, to search for collections whose abstracts match a user's information need. It is also useful for more technical inquiries, such as for finding search proxies that support a given attribute model, or proxies that support proximity search.
- *The attribute access characteristics of the locally relevant search proxies.* This data is primarily useful to the query translators. Translators can, for example, find out which proxies support keyword-searchable Dublin-Core Author attributes.

4.5 Discussion

Defining what metadata should be managed in a generic architecture and how it should be managed is a complex task. Many variations on a metadata architecture are possible. In developing the SMA, we explicitly resolved a variety of design trade-off decisions by appealing to two factors: (1) the concrete metadata needs of existing InfoBus services [3], and (2) several of the protocol guidelines set forth in Section 2. We briefly describe here how some key aspects of our design were influenced by the protocol guidelines.

- *Design for interoperability beyond the "least common denominator."* Many of the attribute models that are in use today are simply flat attribute sets that do not record

information about attribute relationships. For example, many attribute sets contain both “author” and “corporate author” as distinct attributes. Whether or not a “corporate author” is interpreted as a kind of “author” is up to the writer of a metadata record or to conventional rules about how these fields should be used and interpreted. By moving beyond the “least common denominator” and providing for relationship information to be encoded explicitly in our attribute models, we have paved the way for more sophisticated forms of querying and result analysis.

- *Do not limit implementors unnecessarily.* In designing our metadata repositories, we ensured that the contents of repositories could be obtained either by pulling information from the local contributing services or by having the local services push their information to the repository. Depending on the situation, one approach might win out over the other. Push is often useful for cases in which a service makes infrequent major changes. Pull is often useful for cases in which services have exhibited unreliability in alerting a repository of important changes, or for cases in which outdated repository information causes difficulty for its clients. We realized that we could not anticipate which of these situations would prevail in use, and thus decided that this decision was best left to the implementors of repositories and contributing metadata services.

A second area in which this guideline was helpful was in the design of the metadata facility for search service proxies. Before the development of the SMA, there was no standard way of obtaining service-related metadata from a proxy. Adding a metadata facility to the proxy made this information uniformly available. However, we needed to decide how this information should be accessed. One possibility was to make this metadata accessible via the DLIOP. This would let us query a proxy for its information and for its metadata via the same protocol (similar to the approach taken by Z39.50’s Explain facility [27]). However, we realized that this approach would put an extra burden on proxy implementors in that they would need to write their own search facilities for locating this information. We decided that it was better instead to keep proxies as lightweight as possible and thus to give implementors more freedom and fewer restrictions in their proxy design. The SMA only requires that metadata be exported from the proxy in a single “blob” and leaves the provision of facilities for searching over metadata to the local metadata repository.

4.6 Implementation Status

We have implemented prototype instances of all four component classes of our metadata architecture: several attribute model proxies, two attribute model translators, a metadata repository, and a metadata facility for a search proxy. Our attribute model proxies include implementations of proxies for Z39.50’s Bib-1, Dublin Core, Refer, BibTeX, GILS, and a subset of USMARC. We can, for example, search over our USMARC proxy for all attributes containing the word Title in their description. This returns five entries, including attributes for Title Statement, Varying Form of Title, and Main Entry -- Uniform Title. This information will be used in our user interface to help users select proper attributes for search.

5.0 STARTS: Managing Search

The Stanford Protocol Proposal for Internet Retrieval and Search (STARTS) is an emerging protocol for Internet retrieval and search. In this section, we give a brief survey of the STARTS service layer; further information can be found in [9].

5.1 Objective

Users have many document sources available, both within their organizations and on the Internet (e.g., the CS-TR sources[3]). The source contents are often hidden behind search interfaces and models that vary from source to source. Even individual organizations use search engines from different vendors to index their internal document collections. These organizations can benefit from *metasearchers*, which are services that provide unified query interfaces to multiple search engines. These give users the illusion of a single combined document source.

The goal of STARTS is to facilitate the main three tasks that a metasearcher performs:

- Choosing the best sources to evaluate a query
- Evaluating the query at these sources
- Merging the query results from these sources

Also, as will become clear after the discussion below, STARTS makes it easier to build InfoBus proxies to external services.

5.2 Problems Addressed and Approach

Building metasearchers is currently a hard task because different search engines are largely incompatible and do not allow for interoperability. In effect, a metasearcher faces the following three problems:

- *The source-metadata problem:* A metasearcher might have thousands of sources available for querying. It then becomes crucial that the metasearcher just contact potentially useful sources. So, the metasearcher needs information about the sources' contents to choose the best sources for a given query.
- *The query-language problem:* A metasearcher submits queries over multiple sources. But the interfaces and capabilities of these sources may vary dramatically. Thus, the metasearcher has to translate the original query to adjust to each source's syntax and capabilities.
- *The rank-merging problem:* Most commercial sources rank documents according to how "similar" the documents and a given query are. Merging query results from different sources is hard, since a metasearcher might have little or no information on how the document ranks are computed at the sources.

In order to address the source-metadata problem, STARTS defines the information that sources should export about themselves. This information includes automatically generated content summaries to assist in choosing the best sources for a query. It also includes a description of the query capabilities available at the sources, since the STARTS query language has several optional parts that sources might decide not to support.

To address the query-language problem, STARTS defines a simple query language that sources should support. This language is based on a simple subset of Z39.50 [27]. A STARTS query contains two (optional) components: a Boolean expression that defines the documents that qualify in the answer to the query, and a ranking expression that associates a score with these documents and ranks them accordingly. STARTS also defines a suggested set of *fields* (e.g., author, title), and a suggested set of *modifiers* (e.g.,

3. The CS-TR sources constitute an emerging library of Computer Science Technical Reports. Cf. http://www.ncstrl.org/.

thesaurus, stemming) that should be available for searching at the sources. If sources follow these suggestions, then query translation becomes greatly facilitated.

Finally, to address the rank-merging problem, STARTS requires that sources return some statistics together with the results for a given query. This way, a metasearcher can ignore the scores that the sources compute for the documents, and compute its own scores from these statistics without having to retrieve the documents in the query results. The metasearcher then ranks the documents using the new scores. Statistics returned include the number of times that each of the query keywords occur in the documents in the query results, for example.

5.3 Discussion

Although STARTS intends to make the search-engine world more uniform, it leaves many aspects of the protocol open. As a notable example, STARTS does not specify how sources should answer queries. Instead, it just specifies a query syntax, with many options and significant room for future extensions.

A key goal in the STARTS specification is to keep the protocol requirements low and easy to implement. For example, the content summaries that sources should export are easily computable from the standard inverted-file indexes that search engines use to answer queries efficiently.

Throughout the STARTS specification, there are ways of extending the protocol with new, unforeseen features. For example, a new set of search fields may be defined in the future for some specific domain. A source may then support this new field set by simply specifying the set name in the source's metadata.

The goal of STARTS is to facilitate searching over *text* sources. No attempt is made to cover non-textual information, for example. As a result, the STARTS specification is much simpler and focused than what it would have been otherwise.

STARTS could have adopted a "least common denominator" approach. However, many interesting interactions would have been impossible under such a solution. Alternatively, STARTS could have incorporated the sophisticated features that the search engines provide, but that also would have challenged interoperability, and would have driven us away from simplicity. Consequently, we had to walk a very fine line, trying to find a solution that would be expressible enough, but not too complicated or impossible to quickly implement by the search engine vendors.

5.4 Implementation Status

STARTS has been developed in a unique way. It is not a standard, but a group effort involving over eleven companies and organizations. The Stanford Digital Libraries Project coordinated search engine vendors and other key players to informally design a protocol that would allow searching and retrieval of information from distributed and heterogeneous sources. The key participants in this effort are Infoseek, Fulcrum, PLS, Verity, WAIS, Microsoft Network, Excite, GILS, Harvest, Hewlett-Packard Laboratories, and Netscape.

The STARTS specification is completed. A reference implementation of the protocol has been built at Cornell University by Carl Lagoze. Also, the Z39.50 community is designing ZDSR, a profile of their Z39.50-1995 standard based on STARTS.

6.0 UPAI: Managing Payment

The Universal Payment Application Interface (UPAI) protocol is the Stanford InfoBus layer that provides services related to payment. UPAI, described in detail in [10], makes it easy to have client applications (such as a merchant's online storefront or a user's browser) that include payment transactions without requiring the client itself to know the details of about a specific payment mechanisms.

6.1 Objective

A number of vendors have offered solutions for secure, digital payments over the Internet. First Virtual's system[4] and e-cash from DigiCash[5] are examples of early payment systems. These and others payment mechanisms all share a few basic conceptual states and transactions; yet they differ greatly in protocol content, format, and order of execution. UPAI steps in at this point, and provides conceptual unification of the different mechanisms as well as ways by which these concepts can be mapped into the native protocols.

A traditional application developer who wants to support both the First Virtual and the DigiCash payment mechanism would add the logics for both protocols to his or her application–as well as a mechanism to select the protocol to use for a particular transaction. As new payment systems gain users, such an application developer would have to add new sets of subroutines to legacy applications.

In contrast, UPAI permits existing applications to remain unchanged. New kinds of payment systems are supported by simply adding new proxies. The existing applications would simply be able to communicate with any new payment protocol since newly introduced objects would respect the pre-defined interface of the application as specified by UPAI.

6.2 Architecture

Figure 7 depicts key components of the UPAI service layer.

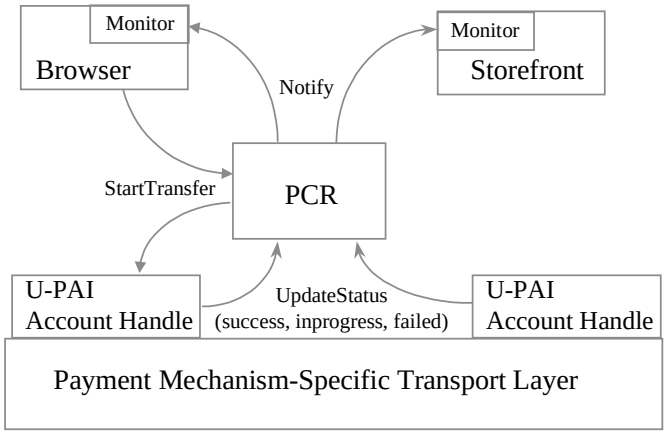
The rectangle at the bottom of the figure represents the native payment mechanism protocol. Notice that the application code (both the browser on the customer's side and the storefront on the merchant's side) never makes direct calls on the native protocols. Instead, they always make the UPAI defined calls through intermediate objects. First, the *Payment Control Record*, represented by the central square labelled PCR, encapsulates all of the necessary information about one payment, such as the source and destination account (represented by `AccountHandles`), and the amount. `AccountHandles` are proxies that translate from UPAI calls such as `StartTransfer` to the corresponding native calls of the payment mechanism, perhaps generating an e-mail message of a particular format or invoking a UNIX command.

Just as calls from the application code to the payment mechanism code are mediated by UPAI proxies, so the return values generated by the native payment mechanism protocol are converted by proxies into appropriate UPAI responses. The status updates are composed of two pieces: a *major status* taking one of three values (`Complete`, `InProgress`, or `Failed`); and a *minor status* (which may be of any type). The `PaymentControlRecord` collects this status information, and rebroadcasts it to each of the *Monitor* objects. Monitor objects have basic interfaces defined by UPAI, but may be subclassed to add additional functionality. The monitor objects are integrated into the

4. Cf. http://www.fv.com/
5. Cf. http://www.digicash.com/

user or merchant's application. They register an interest in tracking the status of the transaction, and receive all updates. If a failure is reported, the user application may, for example, select a different payment mechanism and try again.

FIGURE 7. The UPAI Service Layer: Components.



6.3 Discussion

The design considerations described in Section 2.0 permeate the design of the UPAI protocol. For instance, a standard payment is a common operation, and may be done with a single call in the interface to `StartTransfer.` (Of course, the PCR must be first created with the requisite control information.) This simplicity is in contrast to the complexity of the native payment mechanisms, which often require multiple calls to the operating system for a single payment. Note that UPAI does not provide an *operational* efficiency gain–the proxies still make whatever calls are dictated by the native payment mechanism protocol. UPAI instead provides a gain in *implementation* efficiency; it simplifies the coding process for the application developers. The proxy itself could be developed by the payment system provider or a third party.

The UPAI protocol is by its nature adaptable to new payment mechanisms. As described above, the applications need not be aware of the particular details of any specific payment mechanism. They merely make calls to the UPAI proxies which are identical regardless of the underlying native system. Moreover, the limited set of feasible return values also disguises the underlying details of the native system. Therefore, a browser implemented for a (hypothetical) ‘‘CyberBucks’’ system knows that it should expect the goods that it has ordered if the `StartTransfer` invocation on the CyberBucks `AccountHandle` proxy results in a `Complete` status update. On the other hand, payment mechanism-specific information may also be conveyed in the “minor” status field, if the application's `Monitor` object was designed with awareness of CyberBucks. For instance, if the `MajorStatus` is `Failure`, and the `MinorStatus` reports that the public key server is temporarily unavailable, the `Monitor` object may reinitiate the transfer after an appropriate delay.

The UPAI protocol's scope is limited to only the operational aspects of payment. It does not, for example, provide any primitives related to the user interface for wallet management. If desired, a higher level protocol could be built on top of UPAI to interact with the user to set spending limits and provide expense reporting.

The UPAI protocol is also designed as an asynchronous protocol. Callbacks are used to convey return values; the methods themselves return as soon as they are invoked. Although this introduces some additional complications in process management, it provides a number of benefits. If a payment system is unavailable (or if it is simply taking longer than expected), the user can recover quickly, returning to an application task, making a second payment, or aborting the initial payment and selecting a different mechanism.

6.4 Implementation Status

A prototype implementation of UPAI has been completed, with a proxy supporting Stanford Bucks. UPAI is used as part of the Stanford Digital Library testbed to deal with the fulfillment processing for payment obligations that are part of digital contracts defined by the InfoBus FIRM rights management service layer. Proxies for the First Virtual and DigiCash protocols were developed for an earlier version of the UPAI protocol.

7.0 FIRM: Managing Rights, Obligations, and Contracts

The Stanford Framework for Interoperable Rights Management (FIRM) defines a rights management service layer on top of existing Internet protocols, supporting a host of usages including, but not limited to, digital contracting, privacy negotiations, and network security.

In FIRM, “form designers” develop standard digital contract forms that are then made available as the “stationery” that anyone can easily take, customize, and instantiate into computational contract objects. These “smart contracts,” also called “commpacts,” articulate and enforce the terms and conditions of a given rights relationship (subscriptions, site licenses, etc.). Commpacts are first-class control objects that are managed independently of the objects they control (by “commpact manager” services), accommodating a wide variety of transaction scenarios.

In this section, we give a basic survey; further detail can be found in [14] and in [15].

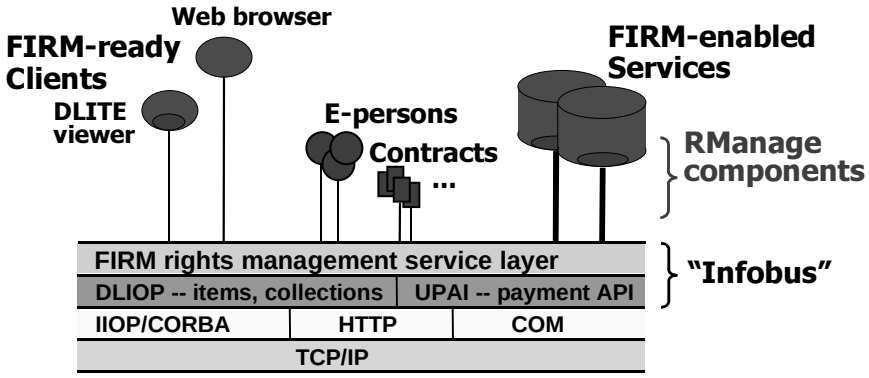
7.1 Towards an “Open Standards” Rights Management Service Layer

FIRM is based on the assumption that the rights management landscape will continue to be heterogeneous, with each of the systems making different design trade-offs to best accommodate the constraints from a specific area of application. FIRM therefore defines a two-level standard, consisting both of a generic specification that defines a domain-independent common rights language object model, and a standard format that allows anyone to contribute media-specific or domain-specific rights definitions.

The basic idea behind FIRM is that its core needs to be limited to represent only the shared structure of the various kinds of rights relationships that people might want to establish. FIRM specifically identifies the basic principles of contract law as this shared structure—since contract law is essentially nothing else than a body of concepts and principles that describe the shared structure of rights relationships in a generic way.

In other words, FIRM uses a computational reification of contract law as its “interoperability core.” Any kind of information that reflects domain-specific conceptualizations is then kept outside of the type system of the core specification itself, while FIRM provides a way of linking in such rights vocabularies into an integrated rights management layer.

FIGURE 8. The FIRM Service Layer: FIRM makes use of the other InfoBus service layers. RManage is a prototype implementation of the FIRM framework.



Rather than having a single-level rights management standard, we therefore choose an extensible, two-level framework (much like the MIME [28] framework):

- a generic (domain-independent) specification that defines a common rights language object model that represents the shared structure of different kinds of rights relationships that one might have, and that provides the transactional infrastructure for negotiating contracts, exercising contract rights, etc.; and
- a format for declaring domain-specific rights. This format allows specific rights language vocabularies to be contributed as "plug-ins" to the basic rights management infrastructure. Sample rights languages that can be expressed in this way include the Unix file access rights language (cf. [14] for details about this case), Xerox' DPRL [20], the EDI standard message types [17], or specific privacy rights languages [16].

In the following two sections, we highlight components of each of these two levels that complement each other to make up FIRM:

- *the FIRM Common Rights Language Object Model:* an interface specification that describes how generic concepts and principles from contract law are reified digitally and how to interact with them.
- *the FIRM Object Attribute Models (FOAMs):* a standard format for defining media-specific or domain-specific rights vocabularies. FOAMs are based on first-class attribute models that define with which attributes FIRM objects represent their state (for use by "agents," etc.).

7.2 FIRM's Common Rights Object Model

The FIRM common object model reifies (generic) principles and concepts from contract law. In FIRM, contracts, contract parties, and even rights and obligations are represented as first-class objects, each with an appropriate set of transactions that can be performed on them.

Shifting the Perspective to Relationships

FIRM takes a relationship-based approach. It represents relationships (contracts) explicitly as first-class objects (called "commpact"/"communication pacts") and encapsulates the boundary conditions of a relationship in these objects.

Note that having such first-class "relationship objects" is different from traditional "client-based" control, which is usually centered around a notion of (access-control) "capabilities." Capabilities are (opaque) tokens that give its possessor (hopefully—but not necessarily—its owner) access to objects that accept them. Capabilities are user-conceptually like the "tickets" that we know from everyday usage—with the same set of associated problems: when they are lost, they are gone; trying to revoke any of them guarantees to be a major enterprise, etc.[6]

However, unlike full-fledged contracts, "tickets" only provide limited information about the context within which their use came about—which is why, in the "real world," they are used almost exclusively in situations which are either characterized by low stakes and a strong imbalance of trust/bargaining power (e.g. Joe Individual vs. National Railway Operations) or low stakes and immediate fulfillment (e.g. movie theater tickets). For such special cases, tickets are a low-cost variant of externalizing contract information.

Some technical systems take this as a reason to only reify tickets and abstain from reifying contracts themselves. While certainly a plausible trade-off for many applications, this clearly limits the affordances available for users of such an environment. For example, while they might be able to purchase a ticket and use it to gain access to online content, there would be no structured ways of cancelling the contract and returning the ticket, or just obtaining information about the warranties a good has, etc. We are interested here in coming up with a more complete infrastructure and investigating its properties. In particular, we would like to provide a way to establish and modulate relationships directly. We will therefore reify contracts themselves. (It turns out that "tickets" will then effectively show up as the way in which contract objects are referenced.)

Note that commpacts also generalize conventional server-based control as well as third-party control in that they accommodate these usages as the special cases where the commpact is interpreted at the server or at a third party, respectively. The difference is that rather than only having idiosyncratic protocols for special purposes (e.g. the PhotoShop group license server protocol) we have the ability to tie in such protocols into an "open standards" framework.

FIRM's Object Reifications

The following is a summary of the some of the major object reifications in FIRM. Note that some of the terms either are new words or they are phrases used with a certain special intended meaning. New words such as 'commpact' and 'epers' were introduced to distinguish between real-world objects (e.g. a legal contract or a real person) and their electronic representation (the digital objects), allowing their interface to be clearly negotiated. In other cases, we use phrases with a somewhat different meaning following the new concepts and institutions introduced by the technical architecture.

Epers (also: e-person): An epers is a software agent that is the persistent digital representation of (a role of) a person with a structured request interface. When acting online, users are identified by a (possibly opaque) handle to their e-person, allowing any communication partners to get back this structured representation and negotiate access conditions in detail—only in some cases involving the user (e.g. when certain interactions are not covered by the default preferences that a user set up for his or her e-person). One

6. The "electronic tickets" that airlines have recently introduced are still called "tickets"; but in terms of the control architecture, they clearly implement more of a form of a commpact in our terminology here.

person can have more than one epers. A Unix account can be seen as a current form of a limited version of an e-person.

Home Provider: A home provider is the service that provides an "online home" for persons in the form of an e-person. Current online services and ISPs can be home providers, although in principle people with a permanently connected machine can also run their own home provider.

Commpact (also: smart contract): A commpact is the computational object that is the digital representation of an agreement between two or more parties, be it a legal contract or a more light-weight "communication pact" (e.g. one related to privacy). Commpacts are "smart contracts" in that they have a structured (FIRM) interface, code that implements behavior, state (e.g. the validity status, the number of times a right was exercised, etc.), and a set of textual descriptions. In other words, commpacts represent a mixture of informal textual descriptions and implementation code (where the fact that both have the same semantics is the responsibility of the designer of the underlying commpact form). Commpacts are effectively a network-centric form of an authorization monitor. They authorize actions, enforce prerequisites ("student status required"), and provide a way to live up to obligations (e.g. initiate a payment transfer when fulfilling an obligation). The piece of text by which we generally know legal contracts is just the result of one of the many methods that can be called on commpact objects–but there are also others, including negotiation methods (e.g. 'terminate'), structural messages ('get me the set of promise objects'), and, last but not least, authorization interactions ('exercise this right'). Cf. Figure 9.

FIGURE 9. Commpacts as "Smart Contract" Objects: FIRM Interface + Code + State + Texts.

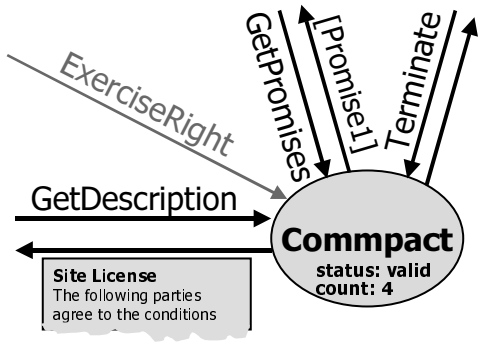
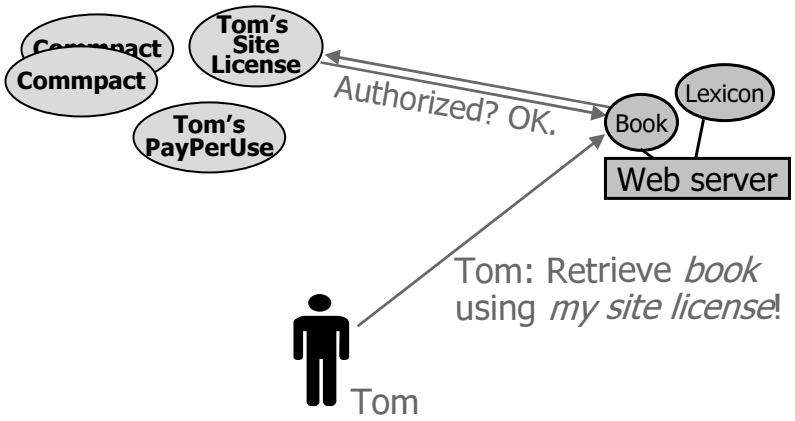


FIGURE 10.

Commpacts as Authorization Monitors: Commpacts are a network-centric variant of an authorization monitor. Actions are authorized using the commpact that the actor designated as the base line with respect to which a certain action is to be performed. The choice of the appropriate commpact will usually be determined by preference rules in a way that is transparent to the user.

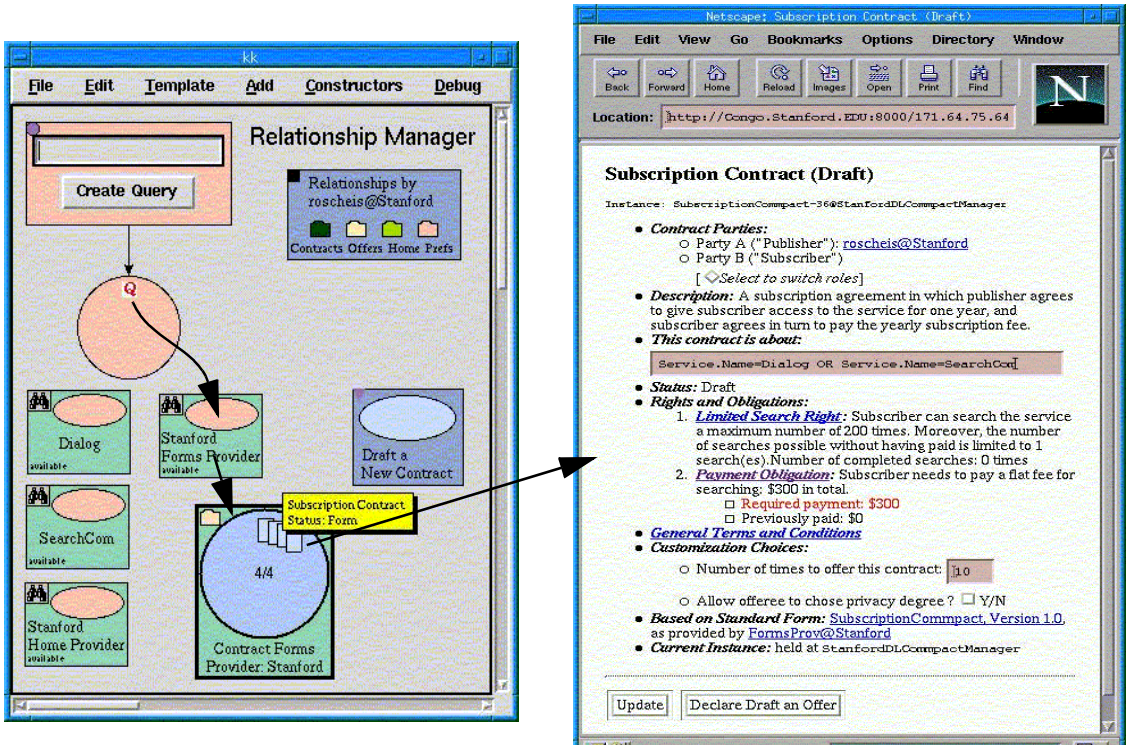


Commpact Manager: A commpact manager is the service that keeps, manages, and interprets commpact objects that have been assigned to it. Commpact managers can be co-located with clients (e.g. for usage control, for the mobile case), servers (conventional access control), or trusted third parties (e.g. rights clearing houses); the protocol remains the same in each case. Every epers has a default commpact manager.

Commpact Form: A commpact form is the basic "template" of a commpact. Commpact forms are much like a standard rental agreements in that they have been carefully designed by someone once, but then they are readily available as "stationery" to everyone else; general users can just "take" such a form, customize it, fill in some parameters (such as the actual price offered, etc.), and then declare it an offer. Commpact forms are assumed to be designed by what we call a "commpact forms designer." They are made accessible through "commpact forms providers."

Commpact Forms Provider: A forms provider is the service that actually operates an online server that carries a collection of commpact forms—such that people can search or browse for the ones that they are interested in. Figure 11 shows a user searching for commpact forms that might be useful to draft a new subscription contract, using the direct-manipulation affordances in the RManage/DLITE [6] interface (a viewer running side-by-side the Web browser).

FIGURE 11. Using Commpact Forms to Make it Easy to Offer New Relationships: In the RManage prototype, a user creates a query for 'subscription' (upper left) and employs the 'Stanford Forms Provider' service (middle) to search for digital contract forms that might be useful in drafting a subscription offer. A result collection is returned with four different commpact forms. Users can inspect each form and then take one and use it to draft an offer (by customizing it and by filling in form-specific parameters).



Reifying Contract Law

FIRM closely follows contract law principles in defining the structure and behavior of its objects. A FIRM interface specification is available in CORBA's Interface Definition Language [18] (although nothing in FIRM intrinsically relies on the availability of a CORBA distributed object infrastructure).

FIRM's top-level encapsulation of control information is that of a contract object ("commpact"). Following the contract law thinking, commpacts are primarily a set of promises between two or more parties that have been mutually agreed upon. Each contract party participates in the contract as part of a certain contract role. Contract roles are characterized by a constraint on who can fill the role. There can be two or more contract roles for one contract, and each role can possibly be filled by more than one party (instance), except for the 'offeror' role. A commpact has an AboutItems constraint that characterizes which items the contract is about. (In other words, the control object quantifies over the objects it controls–rather than being attached to these objects.)

Two major subtypes of promise objects are rights and obligations: rights allow their holders to do something, obligations require them to do something. Promises have components such as a promissory condition (ConditionPrecedent) that specifies which conditions need to be in place before the promise becomes effective, a ConditionSubsequent, etc. Promises can be waived, transferred, etc.

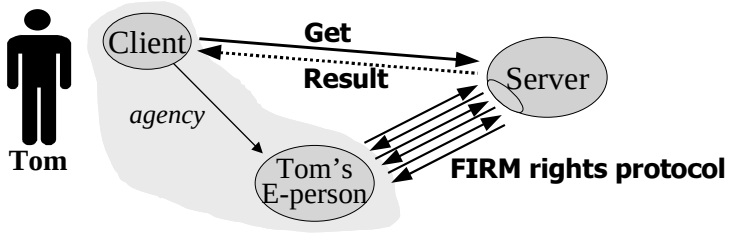
Most importantly, of course, rights can be declared to be exercised, and obligations can be declared to be fulfilled. Moreover, there can be additional "trigger" methods that provide further automatic integration. For example, fulfilling a payment obligation might trigger an actual payment transfer. When this transfer completes successfully, the obligation will be declared to be fulfilled. Declaring fulfillment and initiating fulfillment are two separate actions on an obligation (with the analogous case applying for rights objects). Obligations can register a time with the manager of the commpact that they are part of that describes the circumstance under which they want to receive a 'wakeup' call. For example, a payment obligation that is due on a certain date could register this time, and then, when woken up, it can decide to fulfill a payment obligation right before it is due.

Each FIRM object has a method by which one can obtain a generic description and a generic reference to it that contains both a handle to its current address as well as a persistent name (a "URN") to it (that is valid in the long term, but that might have to undergo some name lookup mechanism to resolve it into an address). Furthermore, there is a protocol by which rights objects can declare what their options are to manipulate their state (e.g., a payment obligation might offer users to conducts payments anonymously; if non-anonymous payment is chosen, then there might be a list of payment means, etc.). In the simplest case, this protocol is just based on HTTP using the standard HTML forms/URL-encoded parameters convention. In this way, people can use a standard Web browser to directly interact with FIRM objects for such purposes as customizing parameters, accepting offers, etc.

FIRM defines a comprehensive protocol for negotiating new relationships and for authorizing actions based on existing relationships (by exercising rights of a previously negotiated contract). This protocol mirrors contract law practices.

FIGURE 12.

Transactions in FIRM: A user's e-person agent executes FIRM protocol actions on behalf of their user. For example, in the RManage prototype implementation of FIRM, users can use an 'e-person control panel' to articulate basic preferences such as which offers an e-person should automatically accept or which obligations it should automatically fulfill. Note that while the client-server transactions can use conventional protocols (e.g. HTTP), the FIRM protocol can be based on more sophisticated mechanisms (e.g. a distributed object infrastructure such as IIOP/CORBA [18]).



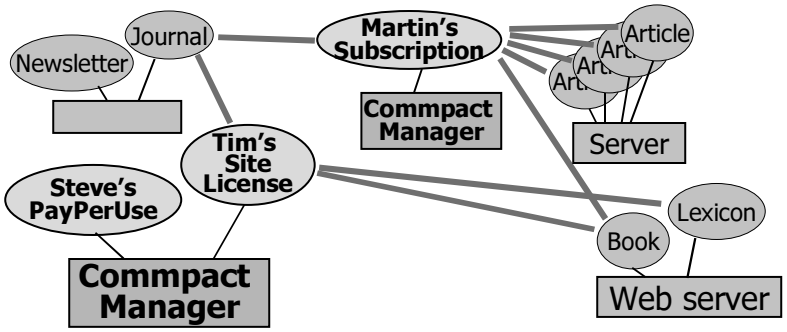
Network-Centric Architecture for Managing Control Objects

We have seen that we would like to accommodate the flexibility of managing first-class control objects independently of any (content/service) objects that they might control. This creates the need to have an underlying architecture that is rich enough to enable such behavior.

FIRM is therefore architecturally based on a "network-centric" design in which control information is encapsulated in first-class (relationship) objects ("commpacts") that can live in principle anywhere on the network and that stand in a *m:n*-relation with the objects and services that they control. Cryptographic means are then used to determine which control objects are eligible to control which other objects, and, vice versa, which objects may be controlled by which control objects.[7] Note that the fact that control and controlled objects are independently managed does of course not exclude the possibility that in some cases commpact managers and content servers are realized in the same address space on one and the same machine (reflecting a traditional access control model).

In other words, rather than attaching control information to controlled information, we have first-class control objects that *designate* the objects that they control (using a constraint). In this way, we keep independent two dimensions that are orthogonal: the question of which controls apply and the question of which objects control is applied to. Note that, among other things, this gives us the flexibility of being able to independently deliver and modify objects of each of the types (e.g. providing a pay-per-view contract in addition to a subscription contract for one and the same online article; making available a new article under a previous subscription contract, etc.).[8]

FIGURE 13. Network-Centric Architecture for Managing Control Information.



FIRM puts an emphasis on reifying relationships/contracts as much as it reifies ways of referencing contracts. Note that this is different from traditional server-based control, where we generally have control behavior embedded in the server mechanism with no principled way of exchanging, moving, and interacting with control information.

7.3 FIRM Object Attribute Models (FOAMs)

Complementing the common rights object model outlined in the previous section, we have domain-specific vocabularies for terms and conditions of specific types of rights objects. These are not taken into the type system of the object model, but they are specified independently.

7. Currently, the only system besides FIRM that accommodates full-fledged control objects in its design is the one recently developed by InterTrust [21] (called 'control sets' there).
8. Note that, strictly speaking, it is therefore incorrect to say that "rights languages *attach* control information to objects"—since they just need to set it in relation with each other ("associate it"). Attachment is primarily a matter of the delivery mechanism (NNTP, etc.), not a matter of the control design.

FIRM uses standard SMA attribute models, called FOAMs here, to "publish" the attributes with which the corresponding objects represent their state—in a way that is useful for other programs (such as "agents"). Consider a payment obligation object that has two state variables: the amount that is due and the amount that has been paid already. This state information is made accessible to other objects in the following way: The payment obligation has a method by which clients can obtain a reference to the attribute model that defines these two attributes. Clients can then use the definition of this attribute model to obtain information about the supported attributes, and request relevant state information then directly from the obligation object.

FOAMs give the flexibility of adding rights vocabularies for specific domains on top of an already established basic FIRM infrastructure. In particular, this flexibility is achieved in a way that does not assume any centralized infrastructure (standards institution), and that is extensible over time without the need to modify the basic FIRM specification.

FIRM's attribute models provide a format by which other rights languages can be tied into the framework. This includes specific mechanisms such as the Unix file access rights language, but also more extensive standards such as the domain-specific record formats of EDI's standard message types.

7.4 Discussion

FIRM provides the programmable framework and the protocols and mechanisms in support of different rights languages/systems. Note that by choosing contract law as the delimitation for what will be part of the "core" standard and what will be part of "plug-ins," we leverage a long history of work done in the legal community that established a way of conceptualizing boundary conditions of social relationships.

Using a two-level standard that separates specific from generic elements has many advantages in terms of simplicity, extensibility, and distribution of authority. In the FIRM framework, participants can continue to use legacy rights management system, combined with wrappers to open up their proprietary rights management protocols for interoperation with third parties.

Moreover, the framework is inherently extensible: Any party can contribute new rights vocabularies (say, for new domains or new media) by simply "publishing" corresponding rights attribute models. Note that such extensibility reduces complexity in that components for additional domains/usages only need to be "loaded" on an as-needed basis.

The FIRM transaction protocol has been designed to have the following characteristics:

- *Simple cases are simple in the FIRM protocol.* In particular, a typical specialization of a simple case turns out to be essentially the same as standard HTTP authorization. In fact, due to the flexibility of the architecture, FIRM is likely to be faster than this in many cases.

- *Complex cases are uniformly possible.* The main point is then that the FIRM protocol uniformly extends to cases that are not possible in existing protocols (e.g. negotiating new relationships). It accommodates sophisticated negotiation and control behavior, although the number of message exchanges will of course scale with the complexity of what is trying to be accomplished.

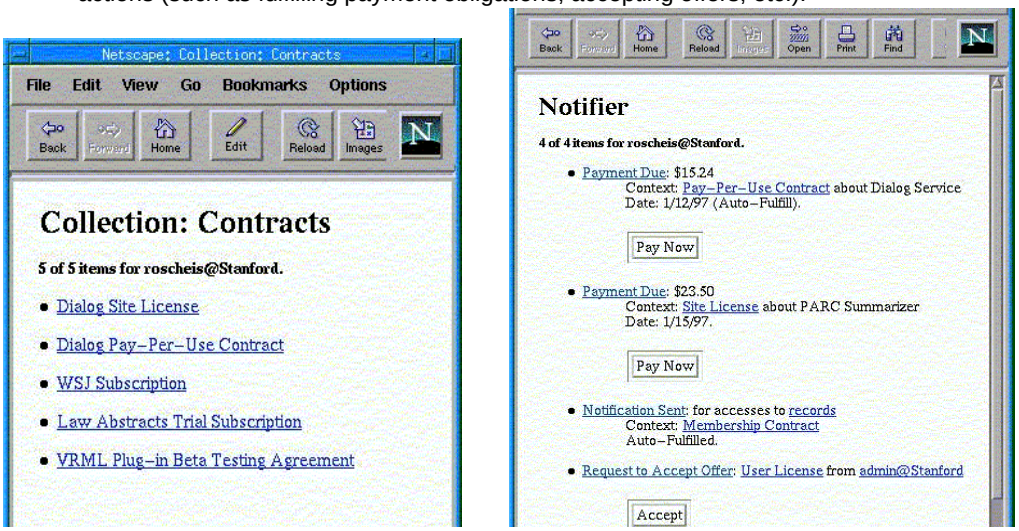
7.5 Implementation Status

FIRM has been prototyped in a system called “RManage.” RManage augments services such as Web servers with plug-ins that allow these services to make use of the FIRM infrastructure. RManage also provides implementations of the e-person and commpact objects that FIRM assumes. Furthermore, RManage can be used either with a plain Web browser or integrated into DLITE [6], an interface viewer developed as part of the Stanford Digital Libraries project.

RManage enables services to make available information governed by (FIRM-compatible) digital contracts. The sample contracts currently used include various forms of subscriptions, site licenses, and pay-per-view contracts, each with different forms of search rights, approval rights, notification obligations, and payment obligations. These digital contracts are “smart contracts” in that they integrate behavior related to what the contract is about (authorization, payment, privacy protection, etc.). For example, RManage provides fulfillment processing for a whole range of payment obligations by making use of the UPAI payment application interface (see Section 6), also prototyped as part of the Stanford project, that provides an abstraction layer to integrate native payment protocols from a variety of providers such as First Virtual, DigiCash, VISA, etc. Figure 14 gives a sample view that shows how users can uniformly manage their relationships and initiate associated actions such as payment transfers in the Web-based RManage client.

Services that are currently using digital contracts as part of our experimental InfoBus testbed include Knight-Ridder’s Dialog databases, the Xerox PARC document summarizer (running behind the company’s firewall), and others. Web-based services that have been augmented by FIRM plug-ins include a site with weather information and a site with a Yahoo-like guide to online legal information. Sample contracts used here deal mainly with the controlled use of personal information (e.g. for content personalization, advertising demographics, etc.).

FIGURE 14. Unified Relationship Management and Notification in RManage: RManage gives users a unified view on their relationships. Each relationship can be manipulated with generic and type-specific actions. Events from relationships are brought to a user’s attention in a uniform way in the notifier structure. The notifier also allows people to initiate certain actions (such as fulfilling payment obligations, accepting offers, etc.).



8.0 Conclusion

We have outlined the architecture of the Stanford InfoBus and its five service layers for managing items and collections (DLIOP), metadata (SMA), search (STARTS), payment (UPAI), and rights management (FIRM). The Stanford "Infobus" extends the current Internet protocols with a suite of higher-level information management protocols that define the technical concepts for a sophisticated future information infrastructure.

Acknowledgments

Scott Hassan developed significant parts of the testbed infrastructure of the project, whose principal investigators are Hector Garcia-Molina, Terry Winograd, and Daphne Koller. Other contributors include Marko Balabanovic, Steve Cousins, Arturo Crespo, Rebecca Wesley, Frankie James, Larry Page, Vicky Reich, Mehran Sahami, Tom Schirmer, Narayanan Shivakumar, and Alan Steremberg.

This material is based upon work supported by the National Science Foundation under Cooperative Agreement IRI-9411306. Funding for this cooperative agreement is also provided by ARPA, NASA, and the industrial partners of the Stanford Digital Libraries Project. Any opinions, finding, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the other sponsors.

9.0 References

[1] Paepcke, A., S. Cousins, H. Garcia-Molina, S. Ketchpel, M. Röscheisen, and T. Winograd (1996). Towards Interoperability in Digital Libraries. *IEEE Computer,* 29 (5).

[2] Baldonado, M. (1997). SenseMaker: An Information-Exploration Interface Supporting the Contextual Evolution of a User's Interest. *Computer-Human Interaction Conference CHI'97*, Atlanta.

[3] Baldonado, M., K. Chang, L. Gravano, and A. Paepcke (1997). The Stanford Digital Library Metadata Architecture. *International Journal of Digital Libraries*, **1**(2).

[4] Baldonado, M., K. Chang, L. Gravano, and A. Paepcke (1997). Metadata for Digital Libraries: Architecture and Design Rationale. *Proceedings of DL'97*.

[5] Chang, C.-C., K., H. Garcia-Molina, and Andreas Paepcke (1996). Boolean Query Mapping Across Heterogeneous Information Sources. *IEEE Transactions on Knowledge and Data Engineering*, **8**(4):515-521, August.

[6] Cousins, S., A. Paepcke, T. Winograd, E.A. Bier, and K. Pier (1996). The Digital Library Integrated Task Environment (DLITE). *Proceedings of DL'97*.

[7] Gravano, L., K. Chen-Chuan Chang, H. Garcia-Molina, and A. Paepcke (1996). STARTS: Stanford Protocol Proposal for Internet Retrieval and Search. Accessible at http://www-db.stanford.edu/~gravano/starts.html

[8] Gravano, L., H. Garcia-Molina, and A. Tomasic (1994). The effectiveness of GLOSS for the text-database discovery problem. *Proceedings of SIGMOD'94*.

[9] Gravano, L., K. Chen-Chuan Chang, Hector Garcia-Molina, and Andreas Paepcke (1996). STARTS: Stanford Proposal for Internet Meta-Searching. *Proceedings of SIGMOD'97*.

[10] Ketchpel, S., *et al.* (1996). U-PAI: The Stanford Universal Payment Application Interface. Economics Subgroup, Stanford Digital Libraries Project. In *USENIX 96--Electronic Commerce.*

[11] Paepcke, A. (1996). InterBib. Cf. http://www-db.stanford.edu/~testbed/.

[12] Balabanovic, M. and Y. Shoham (1997). Combining Content-Based and Collaborative Recommendation. *Communications of the ACM*, 40(3), March.

[13] Shivakumar, N., and Hector Garcia-Molina (1995). SCAM: A Copy Detection Mechanism for Digital Documents. *Proceedings of DL'95.*

[14] Röscheisen, M. (1997). *A Network-Centric Design for Relationship-based Rights Management.* Ph.D. Dissertation, Computer Science Department, Stanford University.

[15] Röscheisen, M., and T. Winograd (1997). The FIRM Framework for Interoperable Rights Management: Defining a Rights Management Service Layer for the Internet. *Forum on Technology-*

based Intellectual Property Management, Washington, DC. Interactive Media Association, White House Economic Council, and White House Office of Science and Technology.

[16] P3 (1997). The Platform for Privacy Preferences. Working group to be announced. Cf. *Release 1.0*, February.

[17] Wright, B. (1995). *The Law of Electronic Commerce. EDI, E-Mail, and Internet: Technology, Proof, and Liability.* Little, Brown and Co.

[18] Object Management Group (1993). *The Common Object Request Broker: Architecture and Specification.* Accessible at ftp://omg.org/pub/CORBA/.

[19] Object Management Group (1995). *Object Property Service*. IBM, SunSoft, Taligent. OMG TC Document 96.6.1.

[20] Stefik, M. (1996). The Digital Property Rights Language. Manual and Tutorial. Version 1.02, September 18th. Xerox Palo Alto Research Center, Palo Alto, CA.

[21] InterTrust (1995). InterTrust Electronic Rights System. InterTrust, Incorporated. Web: http://www.intertrust.com/.

[22] USMARC (1994). Format for Bibliographic Data: Including Guidelines for Content Designation. Cataloging Distribution Service, Library of Congress, Washington, D.C.

[23] GILS (1996). Government Information Locator Service. Accessible at http://info.er.usgs.gov:80/gils/.

[24] Hardy D.R., M.F. Schwartz, and D. Wessels (1996). Harvest User's Manual. Accessible at http://harvest.transarc.com/-afs/-transarc.com/-public/-trg/-Harvest/-user-manual/.

[25] Lagoze, C., and D. Ely (1995). Implementation Issues in an Open Architectural Framework for Digital Object Services. TR95-1590, Cornell University.

[26] Lagoze, C., and C.A. Lynch and Ron Daniel Jr. (1996). The Warwick Framework: A Container Architecture for Aggregating Sets of Metadata. TR96-1593, Cornell University.

[27] Z3950 (1995). Information Retrieval: Application Service Definition and Protocol Specification. ANSI/NISO. April.

[28] Borenstein, N., and N. Freed (1993). MIME: Multipurpose Internet Mail Extensions: Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies. Internet RFC 1521.

[29] Cutting, D., B. Janssen, M. Spreitzer, F. Wymore (1993). ILU Reference Manual. Xerox Palo Alto Research Center. Accessible at ftp://ftp.parc.xerox.com/pub/ilu/ilu.html