# Interoperability for Digital Libraries: Problems and Directions

**Andreas Paepcke**
**Chen-Chuan K. Chang**
**Hector Garcia-Molina**
**Terry Winograd**
***Stanford University (paepcke/changcc/hector/winograd@cs.stanford.edu)***

## Introduction

Interoperability is a central concern whenever digital libraries are constructed as collections of independently developed components that rely on each other to accomplish a larger task. The ultimate goal for such a system is to have components evolve independently, yet to allow all components to call on each other efficiently and conveniently. For digital libraries to scale to an international level, they need to be constructed from such interoperable pieces. This is the case not only for technical reasons, but also because information repositories and information processing services for digital libraries often need to be operated by independent organizations.

Frequently, the terms “heterogeneous” or “federated” systems are used to describe cooperating systems where individual components are designed or operated autonomously. This is in contrast to the more general term “distributed systems” that also includes collections of components deployed at different sites that are carefully designed to work with each other. Our focus here is on heterogeneous or federated systems of information resources and services and how they can be made to interoperate.

Interoperability is one of the most critical problems in the 1990’s and beyond, as the number of computer systems, information repositories, applications, and users multiply at an explosive rate. It gets even worse as system design and software production becomes a global activity, where for example, the politics of each region may dictate what services a component may provide, or what data can be exchanged. Interoperability is also, by its very nature, an extremely complex and evolving problem. Although researchers have been struggling with interoperability for over 20 years, it

is often not clear what principles or key results have been established.

Our goal in this paper is to present a broad introduction to the issues of interoperability, suggesting factors that may be used in evaluating interoperability solutions, and providing an overview of solution classes. Interoperability has been surveyed before, but mostly in the context of a specific domain (e.g., database systems, or programming languages). We believe that the advantage of taking a broad "systems" approach is that it lets us identify common issues and solutions that span domains and applications. However, in this article, we cannot provide a broad overview of the literature, or explore any one issue in depth. To compensate, we have prepared an annotated bibliography [1] which points to in-depth examinations of more narrowly focused scope.

In the following section we show that interoperability is not just an issue of inter-component communication, but that it needs to be considered in many different functional parts of a system. We introduce an informal set of criteria by which interoperability solutions may be classified and evaluated. We follow this by a high-level survey of approaches for accomplishing interoperability. This survey shows that there is no single "magic bullet," and that indeed, new approaches have to be used in conjunction with older types of solutions.

## The Problem Space

One reason interoperability has been receiving broad attention is that the problem permeates almost all aspects of digital libraries that are implemented as distributed computing systems. In this section we illustrate the many places where interoperability issues enter into the constituent system functions. We also highlight some requirements that frequently arise in interoperable systems. Careful decisions around these requirements can impact the cost of solutions significantly. Finally, we informally study the impact and costs of those solutions to interoperability problems.

### Interoperability Issues Permeate Systems

In Figure 1, five large functions of digital libraries are listed along the horizontal dimension. The first function (column) refers to the storage, organization, and retrieval of information, the second to its presentation to users;

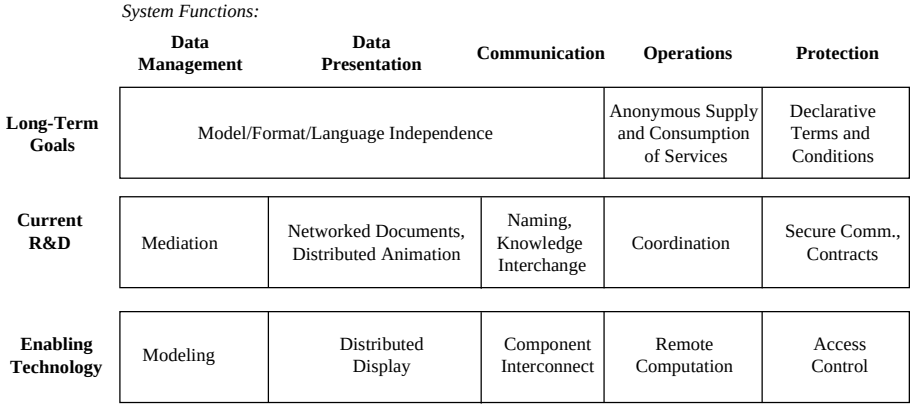


**Figure 1: Examples of system functions where interoperability issues arise**

the third function is concerned with the communication among parts of the overall system; the fourth covers the initiation and control over a system's actions, and the fifth provides protection for users, their property, and information resources.

The rows list respective examples of corresponding enabling technologies, of current research thrusts, and of some long term goals. The purpose of this figure is not to provide a complete overview of the space, but to illustrate at a glance the broad relevance of interoperability.

For example, the *modeling* cell in the information management column represents basic technologies for organizing information in such a way that it can be shared with other parts of a system, even if those follow different information structure conventions. Early federated database systems [11], for instance, created global database schemas to help smooth over syntactic differences. Translators would convert data field specifications to local conventions at the target components. For example, a global schema for a set of business directory databases might specify that the names of firms are stored in a field called **`companyName`**. A directory of corporate donors to charitable organizations might locally store this information in a field called **`corporation`**. Another directory, which lists corporations currently involved in law suits might call the same field **`defendant`**. A client could now issue queries searching over **`companyName`** in all directories at once. This could be used to answer questions like "find the earnings of corporate donors that

are currently involved in lawsuits". Before being submitted to each database, the query would be modified to use the correct local field name.

If we move to the *component interconnection* cell of Figure 1, we find a similar approach in a very different arena. This cell represents the networking technology of inter-component communication. For example, if a collection is available to one set of patrons through a propriety mainframe connection protocol, gateways can translate traffic between the mainframe's native protocol and the World-Wide Web.

In the *remote computation* cell (in the *operations* column) we find this translation approach represented yet again. This column of Figure 1 provides examples of interoperability for invoking operations in a target component. 'Heterogeneous computing' is a term sometimes used in this context [12]. CORBA and DCOM are both protocols that provide the ability for components to be written in different languages and for different computing platforms. The components remain interoperable in the sense that they can invoke operations on each other. Appropriate facilities translate among the mechanics of invoking an operation in each participating system.

We can take the information presentation column of Figure 1 as an example to illustrate the differences between the focus on enabling technology, and currently more prevalent research thrusts. Early interoperability for displaying information on disparate system components relied on some minimal, universal agreement, such as bitmap display technology. Later, the degree of interoperability was enriched by facilities such as the X Window system. An alternative approach, exemplified by Display Postscript in SUN Microsystem's News system, provides a way of describing exactly what is to be rendered, without relying on agreement at the level of a common window system. All receiving components are responsible for finding some way of rendering the descriptions locally.

An increase in focus on this declarative style of display interoperability is exemplified by the SGML/HTML standards. They attempt to communicate semantic intent by tags indicating that a portion of text is to assume the role of 'title', or of 'a bulleted list'. It is up to the receiving components to render these semantic notions appropriately.

Current research thrusts (middle row of Figure 1) are building on basic technologies, usually attempting to provide richer functionality while expanding platform independence. For example, research for the information presentation function of Figure 1 has begun to leverage the enabling remote computation technology provided by Java. Multivalent documents [8], for instance, are renderings of information in a Java virtual machine. These renderings can include behaviors that dynamically turn the image of a text document into ASCII, or that convert a single-spaced document image to be double-spaced. Other behaviors might include reaching through the Internet and requesting another service to summarize the document's contents. Interoperability in this example hinges on the common infrastructure provided by Java and its standard user interface elements. This infrastructure ensures that these richer documents can still move among components of a system.

Similarly, in the operations column of Figure 1, the remote computation facilities provided by CORBA, DCOM, and mobile code, like Java applets, are raising interoperability issues in the context of coordination among independently executing components. Full-scale distributed transaction approaches with guarantees for continuous information consistency can at times be too heavy-weight and too limiting. Some systems therefore attempt to allow heterogeneous components, such as the word processor of remotely collaborating authors of a document, to operate independently for periods of time. The programs then synchronize occasionally, so that over a long period of time, document consistency is assured (see, for example, the Bayou system in [1]).

As a long-term goal (see top row of Figure 1), systems would simply operate by allowing heterogeneous components to come online, advertise their capabilities, and engage in peer-to-peer interactions with other components. This vision is, of course, very difficult to realize, because it is not clear how to describe arbitrary functionality such that other components can inspect the description and 'decide' automatically that this functionality is appropriate for a given task, and what all the parameters are intended to convey.

Similarly, a long-term goal for the protection function is simply to declare terms and conditions for an interaction, and to have the system take care of the rest. For example, a document might travel among components with attached instructions stipulating that the document's contents may be read and passed on to another component, but that it must not be copied. Appropriate watermarking, or even preventative measures would cause these stipulations to be adhered to.

For the information management, information presentation, and communication functions, a major long-term goal is to achieve complete independence from data formats, document models, and languages. The vision is that each component would use, for example, its own way to represent documents, but that documents could still be freely exchanged and widely displayed on different computing platforms, and that maybe even human language barriers might be overcome. While complete human language translation continues to be an elusive goal, some progress is being made in the information management function. For example, Reference [10] describes a system in which queries can be issued using keywords from one human language, but which is able to identify relevant documents written in another language. Still, much more work is needed before the top row of Figure 1 describes reality.

As is evident from Figure 1, interoperability may concern information, operations, and protection functions. Beyond that, there are differences in interoperability requirements that need to be considered when designing a digital library consisting of collaborating components.

Among the requirements for interoperable systems, two will be outlined here: The degree to which component heterogeneity needs to be hidden, and the degree to which components must be bridged at a syntactic versus a semantic level.

**How Much to Hide Heterogeneity?**

Alternative degrees of hiding heterogeneity can be illustrated by examining transparency for three aspects of distributed digital libraries: differing levels of functionality in participating components, heterogeneity among user interfaces, and the effects of data and functionality distribution on the use

of components in the system. We look at each in turn.

Ideally, all components of an interoperable system would be made to appear equally fast, equally rich in functionality, and equally expressive in modeling data. For example, a digital library of independently maintained collections would appear to the user as one big resource whose sucollections all behaved identically. In practice, this is usually not possible. Instead, a series of design choices must be made, depending on how much homogeneity is required. For example, if homogeneity of functionality across all collections is very highly desired, a designer might decide not to make any functionality available that may be obtained at only some of the participating collections. This will ensure that all collections appear maximally homogeneous in functionality, but this approach also sacrifices functionality that would be available if some heterogeneity in the functionality of the digital library's collections were deemed tolerable.

Similarly at the user interface level: If differences in interaction styles are tolerable, it is permissible to display multiple, different user interfaces as users interact with the various collections. On the other hand, if a common look and feel is considered crucial to the success of a system, an interoperability solution may need to include a complete user interface that bypasses the collections' native interfaces.

Finally, usage requirements may demand that the physical distribution of data and operations be transparent. This makes design for interoperability more complex, because it implies that access time differences among the collections need to be eliminated or minimized. This may involve pre-computation, data caching, precise scheduling, or even the artificial slowing of the faster collections. On the other hand, if the transparency of distribution is of less importance, appropriate indicators, such as a cursor turning to an hour glass for some operations, may be acceptable. Alternatively in this case, a human user may be asked to decide whether an expensive operation is to be performed or not. However, this approach assumes the ability to predict system behavior, which is frequently not possible.

The degree to which all aspects of an interoperable digital library are to look homogeneous therefore significantly impacts the complexity of solu-

tions.

**Syntax vs. Semantics**

The degree to which component differences are to be bridged at a syntactic versus a semantic level is the second range of requirements we consider. This difference is frequently stressed when describing recent interoperability projects. Often, the implication is that semantic interoperability is more important or sophisticated than syntactic approaches. In fact, the differences are not always clear-cut.

To a first approximation, a simple example can illustrate the difference between syntactic and semantic interoperability: Consider a component publishing the fact that anyone may remotely call its function **`print(String:author, String:pubData, Float:price, String:address)`**. Assuming appropriate remote invocation technology, this publication provides syntactic interoperability. Anyone can call this function without causing an invocation error. Semantic interoperability would be improved if this component would in addition publish the fact that it will print in 600dpi on the printer in Hall A, that the parameters are supposed to specify a book to be be paid for in Japanese Yen, and that the printed output will be an order form as required by standard company procedure.

This kind of simple example is generally used when describing the difference between syntactic and semantic interoperability. In fact, the difference is actually more complex, in that it recurs at multiple layers. For example, looking at the formula **`(Forall x (Exists y (Knows y x)))`** one might say that the syntax is Lisp-like, but the implied semantics is first-order logic. On the other hand, one might say that its being a statement in first-order logic is really just syntactic, and the semantics has to do with what **`Knows`** means in some axiom system. Or one might in turn characterize the whole formal axiom system as syntactic, and conclude that the real semantics is in its mapping onto some domain of interest in the world. Two representations might therefore be said to be “semantically interoperable” if they can be used with a common inferential system. But are they really interoperable, if **`Knows`** in one system has a different shade of meaning than in the other?

Similar complexity arises in programming languages. What most people

refer to as the “semantics” of a program, is really the syntax of its execution, with no reference to what the program is about, for instance whether it is playing chess or balancing a bank account.

The difference between syntactic and semantic interoperability is thus not clear-cut. We can say loosely that the more ambitious a system becomes in considering semantic interoperability, the more flexibility we have in options for interacting with it, but the more difficult it is to implement.

## Measuring Success

One of the biggest problems with interoperability is that solutions are very difficult to compare. Different approaches operate under differing assumptions, and design goals frequently conflict with one another. It is therefore important to articulate the potentially relevant goals, and to understand tradeoffs among them.

Given the problem space sketched above, and considering associated solutions, we can isolate criteria for evaluating the tradeoffs made by any given approach. There are many such criteria, but the following six stand out:

1. High degree of component autonomy
2. Low cost of infrastructure
3. Ease of contributing components
4. Ease of using components
5. Breadth of task complexity supported by the approach
6. Scalability in the number of components

These are not quantitative measures, but they do provide useful guidelines for understanding distributed and interoperable digital libraries. Sometimes, tradeoffs that optimize one criterion can negatively impact another. For example, a system that minimizes the cost of infrastructure may then only be usable for simple tasks, or may be very difficult to use. We limit the following discussion to the first four criteria.

### Component Autonomy

The degree of component autonomy (Criterion 1) refers to the amount of

compliance to global rules that is required of each participating component. Not considering interactions with other goals, higher autonomy is better, because it provides more local control over implementation and operation of components, and because it makes it easier to include legacy systems as participating components. At one extreme, complete autonomy would make no assumptions of components complying with any global rules. Components could present arbitrary interfaces, and could insist on any interaction protocol or data format they chose. These could be freely changed without notice. At the other extreme, components participating in the system might be required to engage in global procedures such as transactions or information store-and-forward, or to organize all their information by Library of Congress organizational schemes.

Limitations in autonomy may affect many aspects of a component. There may be limitations on how a component may schedule its activities: it may be required to react right away to interrupts, or it may instead be allowed to accept requests asynchronously and return results via callbacks. Another limitation on autonomy may require a component to make all its capabilities available at startup time, at a particular address or port, and in a particular form. Less autonomy limitation in this area may instead allow late binding of functionality. Yet another aspect of autonomy concerns security: Limited autonomy in this area may require all participating components to guarantee certain behaviors, while a higher level of autonomy may instead not make any a-priori rules, but may curb security transgressions dynamically at runtime.

While desirable in principle, high autonomy can lead to solutions that only allow interoperation over the lowest common denominator of functionality, or that require very expensive construction of component descriptions or translation facilities. This in turn can negatively impact other desirable characteristics, such as the ease of using the components.

Practical interoperable systems therefore lie between these extremes. For example, the translation scheme of federated databases with global schemas provides very high autonomy for participating local DBMSs. In contrast, consider the use of blackboard architectures for coordinating large tasks. In this scheme, all components of an interoperable system coordi-

nate their work by posting tasks and results to a centrally accessible location. This approach would provide less autonomy to the components, because they must all agree to use the blackboard, and to adhere to the respective data exchange formats. On the other hand, the system might be easy to use and implement.

**Cost of Infrastructure and Entry**

The cost of a solution is another aspect to consider in any evaluation. Criterion 2 refers to the cost of the infrastructure that is needed to support a solution. These costs can be very difficult to assess because they are shared among many users, or even non-users if funds are raised from taxes. Examples include the development of widely available 'free' software, such as SGML parsers, and the development and maintenance of the Internet. These are costs born by entities beyond the scope of a single organization. If the infrastructure costs are local, such as the installation of fiberoptic wiring in a building, they are easier to assess. We do not further consider these more obvious costs here.

Criterion 3, in contrast, refers to the incremental cost of enabling interoperability when building a new component. This could involve hardware investment necessitated by the approach, or the cost could be in the form of software complexity required to ensure interoperability.

A good example arises in the *coordination* area. If the operations of interoperating components are coordinated by transactions that initially lock all resources needed by a component, then any individual component can be assured that once it is finished and commits its transaction, it will not need to undo what it has done. On the other hand, if coordination is achieved by optimistic concurrency control where all actions are performed even in the face of possible interoperation conflicts with other components, then all components must be much more sophisticated and ready to undo their actions.

Obviously, a low cost of entry is highly desirable, but a higher cost of providing new components may well be justified if it provides other engineering advantages. In the example above, such an advantage potentially arises in the second solution: If there are few conflicts, the overall system will run

faster, because components do not need to wait for resources as often. Another reason for choosing to accept a higher cost of contributing new components is to make them easier to use.

**Ease of Use**

A component's ease of use (Criterion 4) refers both to the complexity of creating client components and to the complexity of interacting with the component at runtime. For example, an information service that provided only a very simple query interface might make the creation of clients easy, but everyday usage might be more complex.

The ease of using existing components in an interoperable system needs to be considered separately from the cost of service component creation (Criterion 3) because the construction of a service component only occurs once and might warrant higher costs, and because it may be desirable to ensure that the creators of client components need not be as well trained as the creators of service components.

For example, consider a remote client/server communication mechanism that is modeled on Unix pipes: clients and the server produce output by writing to a 'standard' output port. Other components consume this output by reading from a 'standard' input port. This design makes client components very easy to build, if the interoperation consists of components producing single data types, such as ASCII encoded words, or a few predefined types, which are then processed by another component. The Common Gateway Interface (CGI) used in the World-Wide Web is an only slightly more involved version of the piping mechanism.

In contrast, the CORBA/DCOM approach requires the programmer to acquire and process a special file that uses a specification language to describe the interface of the service component at a syntactic level. The client program must then faithfully adhere to the conventions laid out in that interface. For simple tasks, this approach makes it more complicated to write client components than in the piping solution. On the other hand, if complex data structures and multiple component methods are involved, a CORBA-like approach is much easier to use, because it takes care of packaging parameters appropriately for travel over the communication link

(parameter marshalling), and it syntactically allows components to be viewed as if they were local objects.

Our examples show that evaluation criteria tend to be interrelated in complex ways. Evaluation also depends on the complexity of tasks the system in question is to be used for. In general, to select particular strategies for a given scenario, one must weigh the importance of each goal by how well each strategy meets the goal. It is of course hard to quantify this whole evaluation process, and one must rely on experience and intuition.

## The Solution Space

Over the years, many, very different approaches to achieving interoperability have been developed. Curiously, as we will see, these solutions are beginning to blend into each other. Figure 2 shows a layout of the solution

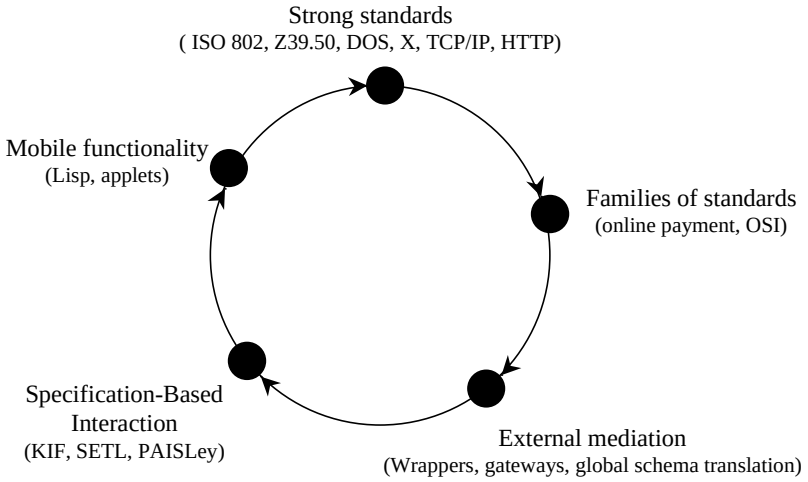


**Figure 2: Families of interoperability solutions**

space. Each point on the circle represents one cluster of approaches. We will sketch each cluster in turn.

### Strong Standards

One of the oldest approaches to achieving interoperability among heterogeneous components is to agree on a standard that achieves a limited amount of homogeneity among them. These standards come about in different ways. Standards such as ISO 802 for network connections and the Z39.50 standard for information retrieval were created by committees that

convened because a large and diverse enough community agreed that a standard was needed. Sometimes one product gains enough market share that it becomes a de facto standard by virtue of its broad deployment, as happened with DOS, and later Windows in the area of desktop operating systems. Other times, government organizations can help a standard gain wide acceptance, as happened with USMARC, one important method for organizing metainformation about books.

Occasionally, a de facto standard will arise spontaneously because a small group of people has developed an approach that is compelling, easy to deploy, and that fills an important need at the right time. The initial versions of the document markup language HTML, the World-Wide Web's communication protocol HTTP, and MIME are examples.

The success or failure of standards, and the design philosophies underlying standardization efforts are very often determined more by social and business decisions than by technical merits. Sometimes companies may resist an official standardization process, because they believe that they are strong enough to establish a de facto standard earlier than an official standard would evolve. This gives them a lead over competitors, because once the de facto standard is then ratified and elevated to official status, their products and technologies have the advantage of deep market penetration. A careful exploration of these connections is important for understanding the impact of standards on interoperability, but it is beyond the scope of this paper. Reference [6] provides a broader treatment of these issues.

If an appropriate standard can be created and is widely adhered to, it provides a powerful interoperability tool. For example, a well-designed, strong standard will make it worthwhile for vendors or free-lance programmers to create easy-to-use modules that implement the standard. When such modules are widely available, the ease of contributing new services that use the standard as a foundation (Criterion 3) is enhanced. A reliable standard also helps encourage infrastructure investment, even when infrastructure costs are high.

One drawback of standards is that they are difficult to agree on, and there-

fore often end up being complex combinations of features that reflect the interests of many disparate parties. A more fundamental reason is that a standard by its very nature infringes on site autonomy (Criterion 1). With a single standard, component providers are no longer free to introduce local optimizations, or to satisfy the preferences of different customer groups. One solution is to include optional portions to the standard. This can quickly lead to increased complexity, and it bears the danger of diluting the standard. An alternative approach to increasing site autonomy without completely losing the benefit of standards is to have more than one standard.

**Families of Standards**

In this approach, components have the choice of implementing one or more of several standards. When two components begin to communicate, an initial automatic or human-mediated negotiation process determines which standards they share. Mirroring everyday life, this approach is increasingly encountered in the area of online payment. Any given vendor or customer may implement payment by a variety of payment schemes, such as First Virtual, DigiCash, or one of several credit cards.

The ISO standard for interconnecting systems (OSI) created an interoperability framework based on the family of standards approach. The OSI conceptually partitions interconnection tasks into seven layers. Each layer contains a family of standards concerned with a given set of interoperability issues in the area of interconnection. For example, the bottom layer contains a set of standards concerned with the physical interconnection of components, such as transmission speeds, or voltage levels. One of the layers above is concerned with packaging information for transport, such as partitioning large bodies of data into packets. Layers near the top are concerned with issues such as establishing sessions. Each layer is to function without knowledge of choices made at the layers below. For example, the session layer is intended to operate without regards to whether the relevant lower layer is using token ring or CSMA access facilities. Interaction negotiations take place only among corresponding layers in communicating components.

The family of standards approach does somewhat alleviate the problem of

autonomy infringement, while maintaining the benefits of standards. But it breaks down when standards are not available, or are not adhered to for technical or business reasons. This can happen, for instance, when applications or infrastructure are still poorly developed, and multiple organizations are attempting to gain market dominance. For these cases, an infrastructure explicitly constructed to provide interoperability among highly autonomous components can be put in place.

**External Mediation**

The only way to provide very high levels of autonomy for components is to locate interoperability machinery outside of the participating local systems. This machinery mediates between the components. One primary function of such mediation machinery is the translation of data formats and interaction modes. For example, in the area of interconnection, network gateways play such a mediation role. Similarly, facilities that map global schemas to local ones are examples of this approach.

However, translation in the sense of simple mapping, is not always sufficient for full interoperability. Sometimes, components completely lack certain data types or operations, and can therefore not interoperate with some clients without further work. For example, consider two collections of documents being provided by different digital library search services. The first provides a ranking feature that sorts search results by estimated relevance, the second does not. In order for a client to interact with both collections equally conveniently, a mediation facility could provide a separate ranking facility. It would be used to augment the less sophisticated collection's functionality. When dealing with the first collection, clients can simply call the `search` operation. Instead of interacting with the second component directly, clients would always interact with the mediation facility which would rank the results. Such mediation facilities are sometimes called wrappers, or proxies. An extensive example is described in [7], where proxy objects play a major mediation role in a digital library environment. Another example is the context mediator component of [9], which is a module that is placed between information clients and servers, and that converts data attributes of queries, and the corresponding result values.

An even more severe mismatch occurs when components differ in their

interaction models. For example, if some components expect to establish long-lasting interaction sessions, while others are stateless, then mediation technology may need to simulate session-based interactions for the stateless components. Connecting HTTP and Z39.50 based components is an example of such mismatches.

Mediation approaches to interoperability are particularly strong along the criteria of autonomy, ease of use, and scalability. They require no compliance from the components, and to the extent that mediation can be successful, clients have the illusion of a highly integrated system. All mediation facilities can be replicated, so scalability tends not to be a problem.

The drawbacks of this approach lie mostly in the area of ease of contributing a new component: whenever a new component is added, a corresponding mediation facility (e.g., wrapper, schema augmentor, etc.) needs to be built as well. Notice that for cases where family of standards solutions are in use by some of the components, this drawback is much less severe. Mediation technology then reaps the benefit of standardization just like any regular client would. For example, in an external mediation system which provides interoperability for highly autonomous search components, a single mediation facility will cover all of the Z39.50 sources at once. Different facilities still need to be constructed for the non-Z39.50 sources.

More generally, if mediation technologies are used to make *n* kinds of components interoperate with *m* other kinds, one needs to construct *n* x *m* mediation facilities. One way out of this complexity is to design the mediation facility such that it uses one common standard (set of operations, data structures, etc.) internally. Then mediation is provided between that internal standard and all the components that are to interoperate. For example, a mediation facility translating among *n* metadata attribute sets might first attempt to translate to USMARC, and then translate from there to the desired target set. Some systems apply the family of standards approach in this context: they translate to one of a small number of intermediate standards, and from there to the final target. This is appropriate if translation to one single common standard is too lossy, because no single standard is sufficiently similar to all components.

An important tool for mediation technology is the use of metadata to describe and translate among components. Metadata is information that describes the elements that the mediation technology deals with, such as components, or data items to be passed among these components. Examples are the global schemas of some federated databases, routing tables for gateways, catalogs for document repositories, the ‘semantic values’ of [9], or tags in document formats like SGML. Due to a current increase in emphasis on component autonomy and consequent interest in interoperability solutions that are strong in this criterion, representation and acquisition of metadata are being widely explored [5].

Metadata plays an even more important role for another approach to interoperability which attempts to avoid the additional infrastructure required by mediation approaches.

**Specification-Based Interaction**

When interoperability is achieved by thoroughly describing the semantics and structure of all data and operations, we speak of a specification-based approach. The vision of these approaches is to allow the use of components without prior arrangement, and without the help of mediators. The goal is to describe each component’s requirements, assumptions, and services such that components can interact with each other after inspecting, and reasoning about each others’ specifications. Various enabling technologies have been developed towards this end goal. For example, an Agent Communication Language (ACL), a knowledge sharing facility for software agents, has included a ‘Knowledge Interchange Format’ (KIF) which is an extension of first order predicate calculus. Also included is a ‘Knowledge Query and Manipulation Language’ (KQML) for passing constraints and instructions among agents [3]. This approach assumes that all components use the same knowledge exchange facilities, although the use of different ontologies to cover varying application domains is anticipated.

The software reuse community also has been interested in methods for describing component functionality succinctly, and as completely as possible. Very High-Level Languages (VHLLs), such as SETL and PAISLey (summarized in [4]), attempt to describe the semantics of a component’s functionality in purely declarative form. That is, the procedural means by

which the functionality is achieved is not the subject of VHLL specifications. The goal in the context of software reuse is to describe component functionality so that the best component can be selected for each job. The same descriptions could also be used to further interoperability in the tradition of specification-based approaches.

Specification-based solutions rate high in autonomy, because of their strict separation of functionality/data description from their implementation. The general lack of non-replicable centralized facilities ensures good scalability. The approaches suffer most from the complexity, and sometimes impossibility, of completely describing components. This drawback makes them rank low in the ease of component contribution criterion.

**Mobile Functionality**

At least beginning with the introduction of Lisp, which made programs and data share the same representation, the movement of functionality implementation has been considered from time to time. General Magic's Magic Cap mobile agent system is one example. Its system had software agents travel through the network to the sites where they could get access to the services they needed. The agents could move even after they had started to execute code. They would then report back to their origin with the results of their work.

More recently, Java applet facilities have enabled approaches that use mobile functionality to deliver new capabilities to client components at runtime, rather than relying on service components to provide functionality remotely, or making all components fully functional from the outset. Interoperability is an important application of mobility: the functionality delivered to a component could be the ability to communicate successfully with another component. Instead of complex component descriptions, third-party mediation, or the use of standardization, this approach accomplishes interoperability by exchanging code that 'does the right thing' to communicate successfully among components. Of course, it is still true that one interface of the applet must be well-known, or hand-adapted. For example, a new kind of search engine might supply clients with an applet that allows sophisticated interactions with that search engine. This makes the search

engine component very autonomous in that its interface can be arbitrary, as long as it supports the request for the applet. On the other hand, the client still needs to know how to communicate with the applet once it arrives. Today, this problem is solved by the fact that most Java applets interface directly to the user, and the user interface standards that are part of Java and Java-enabled browsers are wide-spread. In that sense, Java relies heavily on a standards approach. If applets were used also to implement mobile functionality that is invoked by programs at the client side, then the client-side interaction with the applet would be subject to the same interoperability issues as the original client/service component interaction.

An example of mobile functionality used in the service of interoperability can be found in [2] where a Java applet is used to deliver a small CORBA-based distributed digital library interface. After the applet is received, its sender and the receiving component can communicate via remote method calls. This again solves some of the client/applet interoperability problem through a standards approach, namely CORBA, except that the standards implementation itself is delivered through mobile functionality.

Mobile functionality scores lower in the autonomy criterion than some of the other solutions, because all the components must share the same execution environment (e.g., the Java runtime). On the other hand, contributing a new component is easier in this approach than, for example, in the specification-based approach, because attaining interoperability through mobile functionality involves the creation of concrete programs, rather than a sophisticated, often mathematical, abstraction of functionality. Ease of use tends to be good, except that the client component bears all the risk of importing another component's programs. Until proper security safeguards are worked out, this will continue to represent a significant cost.

If we think of incompatible components as discontinuities within an overall system, then mobile functionality is a technique for smoothing these discontinuities dynamically, whenever the need arises. Note that in this sense, a system based on standards is perfectly smooth at all times: All components can interoperate from the outset. Observed over time, a system whose interoperability is implemented through mobile functionality is therefore equivalent to an interoperable system based on standards. This is why

the solutions in Figure 2 are arranged in a circle.

However, implementing all of a system's interoperability through mobile functionality is expensive in terms of latency and bandwidth consumption, because in the absence of long-term client-side caching, the same code needs to travel across the network again and again (in addition to any related data). As mentioned, mobile functionality is also expensive in terms of risk management, because authenticity and safety of code needs to be checked wherever the functionality travels. Consequently, in the case of Java, interoperability efforts are now beginning to move through the circle of Figure 2 again. For example, facilities delivered frequently have been migrating into World-Wide Web browsers as standard components. One example is the recent addition of Java-based CORBA facilities into Netscape browsers. Of course, as soon as functionality is assumed by component providers to be resident at all client components, interoperability is standards-based. Thus the natural movement of interoperability solutions along the circle of Figure 2. In the future, we could imagine other developments that combine multiple solution families of Figure 2.

## Conclusion

Interoperability is gaining in importance as the Internet brings together digital libraries of different types that are run by separate organizations in different countries. At the same time, the increasing power of desktop computers, the increasing bandwidth of networks, and the popularity of mobile code is changing the interoperability landscape. This results in an increasing urgency for solving the many problems which remain in the way of true interoperability on a national, and international level.

In this paper we have provided an overview of the interoperability space and its solutions. Our discussion has been necessarily informal because interoperability is a complex topic for which no good metrics exist. Nevertheless, we hope to have given the reader a feel for how interoperability issues across different domains are interrelated, for the spectrum of solutions, and for the primary criteria in comparing them.

The following references are a small selection of past work in this area. A

more comprehensive bibliography is available at [1].

## References


[1] An Annotated Bibliography of Interoperability Literature. Stanford University. http://www-diglib.stanford.edu/diglib/pub/interopbib.html.

[2] Steve Cousins. *Reification and Affordances in a User Interface for Interacting with Heterogeneous Distributed Applications*. PhD thesis, Stanford University, 1997.

[3] Michael R. Genesereth and Steven P. Ketchpel. Software Agent. *Communications of the ACM*, 37(7), July, 1994.

[4] Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, June, 1992.

[5] Carl Lagoze, Clifford A. Lynch, and Ron Daniel, Jr. *The Warwick Framework: A Container Architecture for Aggregating Sets of Metadata*. Number TR96-1593. Cornell University, June, 1996.

[6] T.W. Olle. Impact of Standardization Work on the Future of Information Technology. In Nobuyoshi Terashima and Edward Altman, editors, *Advanced IT Tools: IFIP World Conference on IT Tools*, pp. 97–105. Chapman & Hall, September, 1996.

[7] Andreas Paepcke, Steve B. Cousins, Héctor García-Molina, Scott W. Hassan, Steven K. Ketchpel, Martin Röscheisen, and Terry Winograd. Towards Interoperability in Digital Libraries: Overview and Selected Highlights of the Stanford Digital Library Project. *IEEE Computer Magazine*, May, 1996.

[8] Thomas A. Phelps and Robert Wilensky. Toward Active, Extensible, Networked Documents: Multivalent Architecture and Applications. In *Proceedings of DL'96*, 1996.

[9] Edward Sciore, Michael Siegel, and Arnon Rosenthal. Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems. *Transactions on Database Systems*, 19(2):254–290, June, 1994.

[10] Paraic Sheridan and Jean Paul Ballerini. Experiments in Multilingual Information Retrieval Using the SPIDER system. In *Proceedings of the Nineteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 1996.

[11] Amit P. Sheth and James A. Larson. Federated Database Systems for

Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, September, 1990.

[12] Howard Jay Siegel, Henry G. Dietz, and John K. Antonio. Software Support for Heterogeneous Computing. *ACM Computing Surveys*, 28(1):237–239, March, 1996.