Andreas Paepcke, Chen-Chuan K. Chang,
Héctor García-Molina, and Terry Winograd

Interoperability for Digital Libraries Worldwide

How to achieve interoperability among the world's scattered digital libraries? Nobody knows exactly, though it's worth keeping several points in mind when creating the links.

Interoperability is a central concern when building digital libraries as collections of independently developed components that rely on each other to accomplish larger tasks. The ultimate goal for such systems is for the components to evolve independently yet be able to call on one another efficiently and conveniently. Digital libraries designed to scale to international dimensions need to be constructed from such interoperable pieces—not only for technical reasons but because information repositories and information processing services often need to be operated by independent organizations scattered around the world.

The terms "heterogeneous" and "federated" are often used to describe cooperating systems in which individual components are designed or operated autonomously. Such cooperation is in contrast to the more general term "distributed systems," which also includes collections of components deployed at different sites and 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.

	Information Management	Information Presentation	Communication	Operations	Protection
Long-Term Goals	Model/Formal/Language Independence			Anonymous Supply and Consumption of Services	Declarative Terms and Conditions
Current R&D	Mediation	Networked Documents, Distributed Animation	Naming, Knowledge Interchange	Coordination	Secure Communication, Contracts
Enabling Technology	Modeling	Distributed Display	Component Interconnect	Remote Computation	Access Control

Figure 1. Examples of system functions where interoperability issues arise

Interoperability has been a critical problem in the 1990s and will be for the foreseeable future, as the number of computer systems, information repositories, applications, and users multiplies at an explosive rate. It gets worse as system design and software production become global activities in which, for example, the politics of local regions may dictate which services a component may provide or what data can be exchanged. Interoperability is also, by nature, an extremely complex and evolving problem. Although researchers have been struggling with interoperability for more than 20 years, it is often not clear what principles have been established or key results have been obtained.

Here we present a broad introduction to the issues of interoperability, suggesting factors that may be used in evaluating related solutions and providing an overview of solution classes. Interoperability has been surveyed before, but mostly in the context of a specific domain (such as database systems and programming languages). The advantage of taking a broad systems approach is that it lets us identify common issues and solutions spanning domains and applications. We have also prepared an annotated bibliography [1] pointing to various in-depth examinations. Interoperability is not just an issue of intercomponent communication but needs to be considered in each system's many different functions. We offer an informal set of criteria by which interoperability solutions can be classified and evaluated. And our high-level survey of approaches for accomplishing interoperability shows there is no single magic bullet, that indeed, new approaches have to be used in conjunction with older types of solutions.

A Problem for Almost All Distributed Systems

One reason interoperability has been receiving broad attention is that the problem permeates almost all aspects of digital libraries implemented as distributed computing systems. Careful decisions around certain requirements that arise in interoperable systems can influence the cost of solutions significantly.

To identify at a glance the broad relevance of interoperability, Figure 1 lists in the columns five major functions of digital libraries. The first refers to the storage, organization, and retrieval of information; the second to its presentation to users; the third to communication among parts of the overall system; the fourth to initiation and control of a system's actions; and the fifth to protection for users and their property and information resources. The rows list respective examples of corresponding enabling technologies, current research thrusts, and some long-term goals.

For example, the modeling cell in the information management column represents basic technologies for organizing information so it can be shared with other parts of a system, even if they follow different information-structure conventions. Early federated database systems, for instance, created global database schemas to help smooth over syntactic differences [11]. 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, listing corporations involved in lawsuits, might call the same field `defendant`. A client could now issue queries searching over `companyName` in all directories at once. This approach could be used to answer such questions as "Find the earnings of corporate donors currently involved in lawsuits." Before being submitted to each database, the query would be modified to use the correct local field name.

We find a similar approach in a very different arena if we move to the component interconnection cell of Figure 1. This cell represents the networking technology of intercomponent communication. For example, if a library collection is available to one set of patrons through a proprietary 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 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 use the information-presentation column of Figure 1 to show the differences between the focus on earlier enabling technology and the currently more prevalent interoperability research thrusts. These thrusts seek to gain ever more independence from particular computer platforms and are often declarative in nature. 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 such facilities as the X Windows system. An alternative approach, exemplified by Display Postscript in Sun Microsystems' News system, provides a way to describe 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.

Current research thrusts (middle row of Figure 1) build on basic technologies, usually attempting to provide richer functionality while expanding platform independence. For example, research into the information presentation function in Figure 1 has begun to leverage the remote computation enabling technology provided by Java. Multivalent documents [8], for instance, are renderings of information in a client's Java virtual machine. These renderings can include behaviors that dynamically turn the image of a text document into ASCII or convert a single-spaced document image into a double-space one. Other behaviors might include reaching through the Internet to request 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 a system's components.

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 limiting or computationally expensive. Therefore, some systems attempt to allow heterogeneous components, such as the word processors of remotely collaborating authors of a document, to operate independently for random periods of time. The programs then synchronize occasionally, so that over a long period of time, document consistency is ensured (see, for example, the Bayou system in [1]).

As a long-term goal (see top row of Figure 1), systems would operate by allowing heterogeneous components to come online, advertise their capabilities, and engage in peer-to-peer interaction with other components. This vision is very difficult to realize, because it is not clear how to describe arbitrary functionality so 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 for detecting violations or even preventive measures would ensure adherence to these stipulations.

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

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

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.

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 subcollections 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 highly desired, a designer might decide to not make any functionality available that may be obtained at only some of the participating collections. This approach ensures that all collections appear maximally homogeneous in functionality, although it 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 different user interfaces as users interact with the various collections. On the other hand, if a common look and feel is considered crucial to a system's success, an interoperability solution may need to include a complete user interface that bypasses the collections' native interfaces.

Finally, usage requirements may demand transparency of physical distribution of data and operations. This requirement makes designing for interoperability more complex, because it implies that access time differences among the collections need to be eliminated or minimized. Achieving it may involve precomputation, data caching, precise scheduling, or even the artificial slowing of the faster collections. On the other hand, if transparency of distribution is less important, appropriate indicators, such as a cursor turning to an hourglass for some operations, may be acceptable. Alternatively in this case, a human user may be asked to decide whether or not an expensive operation is to be performed. 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 significantly affects the complexity of solutions.

Syntax vs. Semantics

The degree to which component differences are to be bridged at a syntactic vs. a semantic level is frequently stressed when describing interoperability projects. The implication is often that semantic interoperability is more important or sophisticated than syntactic approaches. But the differences are not always clear.

As a first approximation, a simple example can illustrate the difference between syntactic and semantic interoperability: Consider a component publishing the fact that anyone can 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 also published the fact that it will print at 600dpi on the printer in Hall A, that the parameters are supposed to specify a book to 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. But 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 the syntax is Lisp-like, but the

A long-term goal for information management, presentation, and communication functions is complete independence from data formats, document models, and languages.

implied semantics are 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 have to do with what `Knows` means in some axiom system. Or one might instead characterize the whole formal axiom system as syntactic and conclude that the real semantics are in the axiom system 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 inference system. But are they really interoperable if `Knows` in one system has a different shade of meaning 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—whether, for instance, it is playing chess or balancing a checkbook. 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—and the more difficult it is to implement.

Measuring Success

One of the biggest problems with interoperability is that comparing solutions is very difficult. 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 trade-offs among them.

We can, however, isolate criteria for evaluating interoperability solutions. There are many such criteria, but the following six stand out:

- High degree of component autonomy
- Low cost of infrastructure
- Ease of contributing components
- Ease of using components
- Breadth of task complexity supported by the solution
- Scalability in the number of components

These are not quantitative measures but provide useful guidelines for understanding distributed and interoperable digital libraries. Sometimes, trade-offs that optimize one criterion can negatively affect another. For example, a system that minimizes the cost of infrastructure may be usable only for simple tasks or may be difficult to use. Because the resulting simple facilities require more programming for each participating component, we limit the following discussion to the first four criteria.

Component autonomy. The degree of component autonomy refers to the amount of compliance with global rules 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 insist on any interaction protocol or data format. These protocols and formats 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, and, for example, to organize all their information following organizational schemes established by the Library of Congress.

Limiting autonomy may affect many aspects of a component. There may be limitations on how a component schedules its activities; it may be required to react right away to interrupts, or it may be allowed to accept requests asynchronously and return results via callbacks. A component may have 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 may, for example, require all participating components to guarantee certain behaviors, while a higher level of autonomy may not make any a priori rules but curb security transgressions dynamically at run time.

While desirable in principle, high autonomy can lead to solutions that allow interoperation over only the lowest common denominator of functionality or that require very expensive construction of component descriptions or translation facilities.

This limitation in turn can undermine other desirable characteristics, such as the ease of using the components.

Practical interoperable systems lie between these extremes. For example, federated databases with global schemas provide very high autonomy for participating local DBMSs. In contrast, consider the use of blackboard architectures for coordinating large tasks. In such an architecture, all components of an interoperable system coordinate their work by posting tasks and results to a centrally accessible location. This approach provides less autonomy to the components, because they must all agree to use the blackboard and 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. The cost of the infrastructure needed to support a solution can be very difficult to assess even after construction, because costs are shared among many users, or even nonusers if funds are derived from taxes. Examples include development of such widely available "free" software as SGML parsers and the Internet's own development and maintenance. These costs are paid for by entities beyond the scope of a single organization. If the infrastructure costs are local, such as installation of fiberoptic wiring in a building, they are easier to assess.

Ease of contributing components. This criterion, unlike the previous criterion, refers to the incremental cost of enabling interoperability when building a new component. This incremental cost could involve hardware investment necessitated by the approach or could be in the form of software complexity required to ensure interoperability.

A good example is in the coordination area. If the operations of interoperating components are coordinated by transactions that initially lock access to 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 in which 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 own actions.

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 concurrency-control example mentioned earlier, such an advantage potentially arises for the optimistic concurrency solution. If there are few conflicts, the overall system runs faster under the solution, 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 refers to both the complexity of creating client components and the complexity of interacting with the component at run time. For example, an information service that provides only a very simple query interface might make creation of clients easy, but everyday use might be more complex.

The ease of using existing components in an interoperable system needs to be considered separately from the cost of creating service components, because construction of a service component occurs only 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.

Consider, for example, a remote client/server communication mechanism modeled on Unix pipes; clients and the server produce output by writing to a standard output port, and other components consume this output by reading from a standard input port. This design makes client components easy to build if the interoperation consists of components producing single data types, such as ASCII-encoded words or a few predefined types, that are then processed by another component. The Common Gateway Interface used on the Web is a slightly more involved version of this 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 it is 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 syntactically allows components to be viewed as if they were local objects.

These 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, the system's designer must weigh the importance of each goal against how well each strategy meets the goal. Because it is difficult to quantify the whole evaluation process, one must rely on experience and intuition.

Blending Solutions

Over the years, system designers have developed many very different approaches to achieving interoperability. Curiously, these solutions are beginning to blend into each other (see Figure 2). Each point on the circle in Figure 2 represents one cluster of approaches.

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. Such standards as the ISO 802 for network connections and Z39.50 for information retrieval were created by committees that convened because a large and diverse enough community agreed 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 help a standard gain wide acceptance, as happened with USMARC, an important method for organizing metainformation about books.

A de facto standard occasionally arises spontaneously because a small group of people developed an approach that is compelling and easy to deploy and fills an important need at the right time. Examples include the initial versions of the document markup language HTML, the Web's communication protocol HTTP, and MIME, a set of facilities to enhance Internet mail.

The success or failure of standards and the design philosophies underlying standardization efforts are often determined more by social and business considerations than by technical merit. Companies sometimes resist official standardization processes because they believe they are strong enough to establish a de facto standard earlier than an official standard would evolve. Such a de facto standard would give them a lead over competitors, because once the de facto standard is 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 influence standards exert on interoperability [6], although such exploration is beyond the scope of this article.

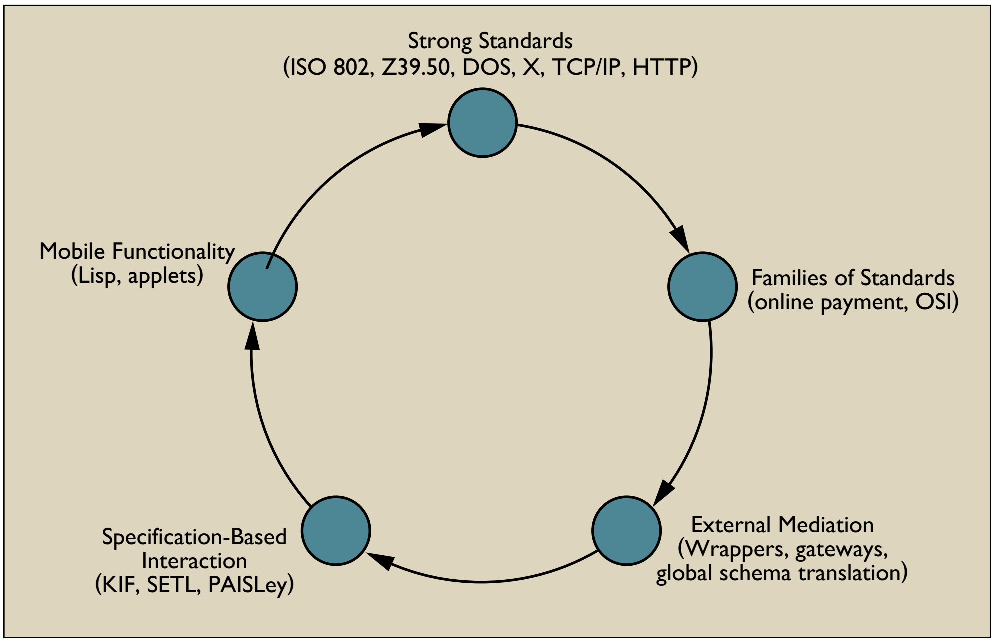

Figure 2. Families of interoperability solutions

An appropriate standard that is widely adhered to provides a powerful interoperability tool. For example, a strong, well-designed standard makes it worthwhile for vendors and freelance programmers to create easy-to-use modules that implement the standard. Wide availability of such modules enhances the ease of contributing new services that use the standard as a foundation (as per the ease-of-contributing-components criterion). 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 therefore often end up being complex combinations of features reflecting the interests of many disparate parties. A more fundamental difficulty is that a standard by nature infringes on site autonomy (as per the high-degree-of-component-autonomy criterion). With a single standard, component providers are no longer free to introduce local optimizations or satisfy the preferences of different customer groups.

One solution is to include optional portions in the standard, but this can quickly lead to increased complexity and risks, 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 the families-of-standards approach, component designers 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 to share. Electronic commerce systems typically operate this way. Any given vendor or customer may implement payment through a variety of payment schemes, such as First Virtual, DigiCash, or one of several credit cards.

The International Organization for Standardization (ISO) standard for interconnecting systems called Open Systems Interconnection (OSI) created an interoperability framework based on the family-of-standards approach. OSI conceptually partitions interconnection tasks into seven layers, each containing a family of standards concerned with a given set of interoperability issues in the area of interconnection. For example, the bottom (first) layer contains a set of standards concerned with the physical interconnection of components, such as transmission speeds and voltage levels. One of the middle layers is concerned with packaging information for transport, such as partitioning large bodies of data into packets. Layers near the top are concerned with such issues as establishing sessions. Each layer is designed to function without knowledge of choices made at other layers. For example, the session layer is intended to operate without regard to whether the relevant lower layer is using token ring or Carrier Sense Multiple Access (CSMA) facilities. Interaction negotiations take place only among the corresponding layers in communicating components.

The family-of-standards approach alleviates the problem of autonomy infringement somewhat, while maintaining the benefits of standards. But the approach breaks down when standards are not available or are not adhered to for technical or business reasons. This lack of standards can occur when, for instance, applications or infrastructure are poorly developed and multiple organizations are attempting to gain market dominance. In such cases, an infrastructure explicitly constructed to provide interoperability among highly autonomous components can be put into place.

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

However, translation in the sense of simple mapping is not always sufficient for full interoperability. Components sometimes completely lack certain data types or operations and therefore cannot interoperate with some clients without further work. For example, consider two collections of documents 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 in an equally convenient way, a mediation facility could provide a separate ranking facility that would 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 in [9], which is placed between information clients and servers and converts data attributes of queries and the corresponding result values.

Even more of a 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 a mismatch.

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

The drawbacks of the mediation approach lie mostly in the area of ease of contributing a new component; whenever a new component is added, a corresponding mediation facility (such as a wrapper or a schema augmentor) needs to be built as well. Notice that for cases in which family-of-standards solutions are used by some of the components, this drawback is much less severe.

Mediation technology then reaps the benefit of standardization just as any regular client would. For example, in an external mediation system providing interoperability for highly autonomous search components, a single mediation facility covers all 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, the system designer needs to construct n × m mediation facilities. One way out of this complexity is to design the mediation facility so it uses one common standard internally (for, say, sets of operations and data structures). 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 attempt first to translate to USMARC and then to translate from there to the desired target set. Some systems apply the family-of-standards approach in this context, translating to one of a small number of intermediate standards and from there to the final target. This method is appropriate if translation to a single common standard is too lossy, because no single standard is sufficiently similar to all components. For example, the Networked Digital Library of Theses and Dissertations uses PDF (a page-description de facto standard) and SGML (an international markup standard).

An important tool for mediation technology is metadata for describing and translating among components. Metadata is information describing the elements the mediation technology deals with, such as components, or data items to be passed among the components. Examples are the global schemas of some federated databases, routing tables for gateways, catalogs for document repositories, "semantic values" in [9], and tags in document formats like SGML. Due to the current increased emphasis on component autonomy and the consequent interest in interoperability solutions 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—specification-based interaction—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 such an approach 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 so the components can interact with each other after inspecting and reasoning about each others' specifications. Various enabling technologies have been developed toward this goal. For example, the Agent Communication Language (ACL), a knowledge-sharing facility for software agents, includes a Knowledge Interchange Format (KIF) that 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]. The specification-based interaction 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 is also interested in methods for describing component functionality as succinctly and completely as possible. Very high-level languages (VHLLs), such as SETL and PAISLey [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 the best component is selected for each job. The same descriptions can also be used to further interoperability in the tradition of specification-based approaches.

Specification-based solutions rate high in autonomy, because of the strict separation of their functionality/data description from their implementation. The general lack of nonreplicable centralized facilities ensures good scalability. These approaches suffer most from the complexity—and sometimes impossibility—of completely describing components, giving them a low ranking on the ease-of-component-contribution criterion.

Mobile functionality. At least since the introduction of Lisp in the late 1950s, which made programs and data share the same representation, the movement of functionality implementation has been considered from time to time. An example is General Magic's Magic Cap mobile agent system, which has software agents travel through the network to sites where they access the services they need. The agents move even after starting to execute code. They then report back to their original site with the results of their work.

More recently, Java applet facilities have enabled approaches using mobile functionality to deliver new capabilities to client components at run time, 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, and the functionality delivered to a component can be the ability to communicate successfully with another component. Instead of complex component descriptions, third-party mediation, or standardization, this approach accomplishes interoperability by exchanging code that "does the right thing" to communicate successfully among components. However, it is still true that the one interface of the applet itself must be well known or hand adapted. For example, a new kind of search engine might supply clients with an applet allowing sophisticated interactions with that search engine. This approach 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 with the user and that the user interface standards that are part of Java and Java-enabled browsers are widely available. In that sense, Java relies heavily on a standards approach. If applets were also used to implement mobile functionality invoked by programs on 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 in the service of interoperability can be found in [2], in which 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 arrangement again solves some of the client/applet interoperability problems through a standards approach, namely CORBA, except that the standards implementation itself is delivered through mobile functionality.

Mobile functionality scores lower on the autonomy criterion than some of the other solutions, because all the components share the same execution environment (such as the Java run time). On the other hand, contributing a new component is easier in this approach than, for example, in the specification-based approach, because achieving interoperability through mobile functionality involves 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 risk 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 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 equivalence 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 (in addition to any related data) needs to travel across the network again and again. Mobile functionality is also expensive in terms of risk management, because authenticity and safety of code have to be checked wherever the functionality travels.

Consequently, in the case of Java, interoperability efforts are now beginning to move through the circle in Figure 2 again. For example, facilities delivered frequently have been migrating into Web browsers as standard components. One example is the recent addition of Java-based CORBA facilities to Netscape browsers. As soon as functionality is assumed by component providers to be resident at all client components, interoperability is standards-based. Thus there is natural movement of interoperability solutions along the circle in Figure 2. In the future, we could imagine other developments based on combinations of the solution families in Figure 2.

Conclusions

Interoperability is gaining in importance as the Internet unites digital libraries of different types 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. The result is an urgent need to solve the problems hindering true interoperability on national and international scales.

Our discussion has been informal, because interoperability is a complex topic for which there are no good metrics. Nevertheless, we hope to have provided a feel for how issues of interoperability across different domains are interrelated, for the spectrum of solutions, and for the primary criteria for comparing them. C

References

1. An Annotated Bibliography of Interoperability Literature. Stanford University; see www-diglib.stanford.edu/diglib/pub/interopbib.html.
2. Cousins, S. Reification and affordances in a user interface for interacting with heterogeneous distributed applications. Ph.D. dissertation, Stanford University, 1997.
3. Genesereth, M., and Ketchpel, S. Software agents. *Commun. ACM 37,* 7 (July 1994), 48–53.
4. Krueger, C. Software reuse. *ACM Comput. Surv. 24,* 2 (June 1992), 131–183.
5. Lagoze, C., Lynch, C., and Daniel, R., Jr. The Warwick Framework: A container architecture for aggregating sets of metadata. Tech. Rep. TR96-1593, Cornell Univ., 1996.
6. Olle, T. Impact of standardization work on the future of information technology. In *IFIP World Conference on IT Tools*, N. Terashima and E. Altman, Eds. (Canberra, Australia, Sept. 2–6), Chapman & Hall, New York, 1996, pp. 97–105.
7. Paepcke, A., Cousins, S., García-Molina, H., Hassan, S., Ketchpel, S., Röscheisen, M., and Winograd, T. Using distributed objects for digital library interoperability. *IEEE Comput. 29,* 5 (May 1996), 61–68.
8. Phelps, T., and Wilensky, R. Toward active, extensible, networked documents: Multivalent architecture and applications. In *Proceedings of DL'96* (Bethesda, Md., Mar. 20–23), ACM Press, New York, 1996, pp. 100–108.
9. Sciore, E., Siegel, M., and Rosenthal, A. Using semantic values to facilitate interoperability among heterogeneous information systems. *Trans. Database Syst. 19*, 2 (June 1994), 254–290.
10. Sheridan, P., and Ballerini, J. Experiments in multilingual information retrieval using the SPIDER system. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (Zurich, Aug. 18–22), ACM Press, New York, 1996, pp. 58–65.
11. Sheth, A., and Larson, J. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv. 22*, 3 (Sept. 1990), 183–236.
12. Siegel, H., Dietz, H., and Antonio, J. Software support for heterogeneous computing. *ACM Comput. Surv. 28*, 1 (Mar. 1996), 237–239.

Andreas Paepcke (paepcke@cs.stanford.edu) is a senior researcher at Stanford University.
Chen-Chuan K. Chang (changcc@cs.stanford.edu) is a graduate student of computer science at Stanford University.
Héctor García-Molina (hector@cs.stanford.edu) is a professor of computer science at Stanford University.
Terry Winograd (winograd@cs.stanford.edu) is a professor of computer science at Stanford University.