Solving the Data Inter-operability Problem Using a Universal Data Access Broker

Mike Higgs   Bruce Cottman
I-Kinetics, Inc.
{*mhiggs, bruce.cottman*}@*i-kinetics.com*

1 What is Data Inter-operability?

Stored data has been fundamental to human enterprise ever since the invention of clay tablets, or perhaps even as early as primitive cave paintings. All stored data requires the definition of persistent representations for that data, and corresponding interfaces to access and use that data. Once invented, each representation of persistent data must inevitably inter-operate with humans or mechanisms that use other persistent representations. None of this is new to the human experience, yet the Biblical Tower of Babel is as nothing compared to the bewildering data interoperability problem brought forth by the computer in just fifty years. It's perhaps hard to imagine Alexander halting his armies whilst critical data was translated by scribes from obsolete forms. But no great leap at all to imagine a modern day corporate Alexander stopped in his tracks because his CIO simply cannot get one enterprise system to inter-operate in the necessary manner with data produced by another enterprise system. Or perhaps because his CIO cannot achieve the required data inter-operability at the transaction speeds required by some new business campaign.

2 Historical Approaches to Data Inter-operability

Like much else we invent, a human tendency is that the solutions we form to the problems of data inter-operability have human parallels. Thus many of the most successful solutions parallel translation or interpretation of human languages. To achieve data inter-operability stored data is transformed from one representation to another. However like translation of human languages this solution falls short of the ultimate. We want the translator to be dynamically available. We want the translator not just to literally transpose but to deduce or infer semantics. We want the translator to understand all representations, including new variants or dialects as they are added. We want the translator to operate in real time (*on the fly*). Last but of course not least, we want to inter-operate with the translator in the representation of our choice, not one that the translator imposes on us because it will only support certain combinations of representations.

Another human tendency is to continually attempt to solve a problem by inventing the ultimate representation for persistent data. For the problem of data inter-operability this tendency manifests as a series of generations of database technology with corresponding persistent representations for stored data. In theory translating all required data into a new persistent representation will indeed solve the problem of data-interoperability. But unfortunately this flies in the face of both historical and economic reality. The bulk of existing stored data will

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

never be translated into the new persistent representation even if that persistent representation were viable in all problem domains, which it will not be. The truth is that each new persistent representation adds to the Tower of Babel, and the very success of a persistent representation in specific problem domains ensures that it enlarges the problem it is intended to solve rather than reduces it.

A more practical approach, which has been attempted from time to time, is not to invent the ultimate persistent representation into which all stored data must be transformed, but rather the ultimate interface by which all stored data must be manipulated. A recent incarnation of this approach is Microsoft's OLE-DB Universal Data Access [OLE97]. Of course this approach too is flawed, because all interfaces are to some extent problem domain specific, or else so pithy as to be useless. Yet this approach is also more useful, because from time to time different interfaces may become de-facto standards of great utility. This approach is also useful because it increases the value of a translator. Translating a persistent representation into a canonical representation can make the ultimate interface available for that persistent data. Up to this point however, all attempts to resolve the problem of data inter-operability using the ultimate interface approach have foundered on the limitations of the proposed interface, or upon non-technical constraints or characteristics of the proposed interface. Sometimes the ultimate interface approach turns out really to be a Trojan Horse designed to advance yet another ultimate representation approach.

So what is the real solution to the data inter-operability Tower of Babel? And why with all the inventiveness of the human race has the computer industry not thrown up such a solution in the past fifty years? We believe that the solution is neither an ultimate persistent representation for stored data, nor an ultimate interface for manipulating stored data. Both of these solutions are inherently static, and turn their face against entropy so that ultimately all they can do is build the data inter-operability Tower of Babel higher. Instead the solution is a representation independent Broker that is dynamically capable of supporting a variety of different interfaces against an even wider variety of persistent data representations. In theory a Broker is the universal interpreter caricatured by Lucas' 3CPIO, able not only to translate stored data between an almost infinite number of persistent representations, but also to infer and superimpose the necessary semantics and gestures. In practice a Broker provides an approach which can be used to define a product architecture where human ingenuity can be brought to bear on the problem of engineering usable solutions to theoretically intractable problems. Whatever the actual size of the data inter-operability Tower of Babel, a Broker can make it appear manageable.

We call this approach to solving the problem of data inter-operability the Universal Data Access Broker. Yet at first sight the Universal Data Access Broker still appears to be an unbounded engineering problem. Mathematically there are potentially even more interfaces to persistent representations for stored data than there are representations.

So how can a Universal Data Access Broker be produced? The answer lies in the fact that whilst there are an infinite variety of interfaces to persistent data representations, there are increasingly a bounded number of interfaces which be used "most of the time." This reflects the counter-balance to the human tendency to expand the data inter-operability Tower of Babel. This counter-balance is the equally powerful human tendency to take the easiest path. We observed that increasingly the tools for manipulating stored data have converged on a set of interfaces that are used as common plumbing, or middleware. A compelling example of this is the widespread use and abuse of Microsoft's ODBC [Gei94] interface. Middleware interfaces like ODBC that are required by widely used tools are often supported even if the competitive advantage of a particular persistent data is at odds with the interface. In the case of ODBC implementations of the interface are available even for database technology like object oriented databases [CBB97], which are in other ways fundamentally competing with the relational underpinnings of ODBC [SQL92]. The value of the middleware interface has transcended the contradictions and difficulties of applying the interface to the persistent data representation. This phenomenon, most likely an emerging paradigm shift, means that by both intent and in other cases accident certain middleware has become fundamental as the glue that binds together the tapestry of tools which we now use to develop systems. By leveraging this paradigm shift a Universal Data Access Broker can be produced which resolves a huge part of the data inter-operability Tower of Babel by supporting middleware interfaces.

3 What is Data Inter-Operability? Revisited

Before discussing the Universal Data Access Broker in more detail we believe it is important to distinguish between:

- *Syntactic Data Interoperability*. Implemented by a translator. Data is simply transformed from one persistent representation to another, typically in discrete (or batch) operations.
- *Semantic Data Interoperability*. This transforms data from one persistent representation into an implementation of an interface. This requires both transformation of the data into new representation and that "something" implements the semantic behavior corresponding to the new persistent form of the data. Rather than being transformed, data is "published" using a specific interface.

Syntactic data inter-operability can be enormously useful. Even so a typical phenomenon is that as soon as the user can access transformed data they will discover that they need to do more than "just" access it. In a typical business domain (where relational technology is pervasive) they may want to query it, update it in place and support all of the semantics for the data which will allow tools to be used. With syntactic data inter- operability this typically requires the use of syntactic inter-operability to transform the data into a canonical representation in a fixed persistent data form which can supply the missing semantics. For example the transformed data may actually be loaded into a relational database. Unfortunately this raises a series of complex new problems, of which the most obvious is the often intractable problem of synchronizing multiple representations of the same data.

Semantic data inter-operability is fundamentally more difficult because as part of the transformation process "something" must supply the semantics. Also semantic data inter-operability typically requires that data be transformed in ad-hoc ways, corresponding to the desired interface (often referred to as *on-the-fly* transformation). Whether the semantics can be inferred, or require human intervention to supply "hints" or even complete implementations depends on the mapping between the persistent data representation and the desired interface. For semantic data inter-operability to be useful we expect it to have to support a number of interfaces which the user will require. We further distinguish semantic data inter-operability based on exactly how those interfaces are supported resulting in two different kinds of semantic data inter-operability:

- Shallow – where the semantics of the desired interface are implemented against the persistent representation of the data.
- Deep – where the semantics of the desired interface are maintained inside a more complex weave of semantics. For example deep semantic data inter-operability allows for the implementation of a desired interface inside a transaction context which requires two-phase commit, or inside a secure context which requires that all communication be encrypted. These additional characteristics are not defined by the desired interface, but by other interfaces in the infrastructure. In effect the Broker is implementing not just one but a collection of interfaces against the persistent data representation:

A Universal Data Access Broker is capable of supporting both shallow and deep semantic data inter-operability. In fact it is likely that Brokers will support both kinds of semantic data inter-operability in order to allow users to make cost/benefit decisions. This is particularly true since we observe that achieving semantic data inter-operability is often described in other contexts as "legacy integration," or "legacy wrapping" and is a universal recurring problem which up to now has typically been solved by ad-hoc means. The Universal Data Access Broker provides a consistent approach and underpinning for attaining semantic data inter-operability in a wide variety of different domains.

4 Implementing a Universal Data Access Broker

The requirement for semantic data inter-operability is hardly new. The Broker approach is not obscure and consequently the construction of these kind of Universal Data Access Brokers has been attempted in the past. But it has always fallen short of the goal. Not so many years ago, if someone had proposed building a product which could support a standard API for data access which could be used from any platform they would have been considered deluded. So why can a Universal Data Access Broker finally be built now?

We believe this is because the appropriate technological underpinnings for a Universal Data Access Broker are finally available. We believe that the failure of previous attempts at constructing a Universal Data Access Broker can be traced not to inherent flaws in the approach, but instead to the lack of those underpinnings and the technological and economic barriers that this presented to those attempts. The fact that it took so long for these technological underpinnings to become available is at least in part because these underpinnings represent very fundamental paradigm shifts in the way that we approach the development of systems which manipulate persistent data.

We believe that the missing technological underpinnings for Universal Data Access Brokers were:

- Widely available and economically exploitable multi-threading and multi-processor systems.
- Universally available and economically exploitable infrastructures for distributed computing.
- The availability of Java, and Java based interfaces for data access.
- Widespread use and acceptance of object oriented approaches to building systems.

Let's address each of these in turn in terms of their impact on constructing a Universal Data Access Broker:

Multi-threading. Multi-threading is the ability of a single program (or process) context to establish a number of different execution contexts and allow the operating system to map those execution contexts onto a number of available processors. Only relatively recently has useful multi-threading become economically available on classes of systems other than mainframes. Multi-threading as a theoretical technique is well known and described [Kle95, Lea97], but without widespread availability (and corresponding cost) of Symmetric Multi-Processor (SMP) systems the benefits of using multi-threading were constrained. We believe that widely available multi-threading has the impact of a paradigm shift. This is because it allows the use of design and implementation techniques for massive scale-ability that had previously only been available to developers on mainframe class systems. For the purposes of a Broker multi-threading allows the construction of a Broker that scales to the necessary user configurations, sometimes thousands or tens of thousands of clients. Multi-threading also allows a Broker to provide effective returns on investment as more processors are added to an SMP system on which it runs.

Universally available distributed infrastructures. When we talk about data inter-operability we implicitly assume that there is some means of accessing a persistent data representation from the context in which we want to manipulate that data. Yet in the past the assumption of even this weak kind of inter-operability was impossible. Different types of systems were incompatible islands that could only be persuaded to talk to each other using low level network programming. Not long ago it would have been almost lunatic to conceive of an infrastructure that would allow programs on any system, in almost any language to inter-operate. Yet with CORBA 2.0 [CORBA], this is now a reality. This observation reinforces the fact that only part of the value of CORBA lies in the "distributed object" capabilities. Much of the key value of CORBA lies in the fact that it is practical to assume that interaction with a CORBA based system will always be possible at reasonable cost, no matter what platform or language the interaction is required in. This allows the construction of Brokers that do not have to solve the impossible combinatoric problem of supporting basic cross platform and cross language inter-operability. A universal distributed infrastructure is absolutely required for a practical Broker, not only to support appropriate de-facto drivers on each platform, but also to allow development in absolutely any desired configuration.

The availability of Java, and Java based interfaces for data access. Java as a technology is extensively documented elsewhere [GJS96] and does not need description here. The importance of Java to a Broker lies in the standard interfaces which JavaSoft have defined based on Java, in particular the JDBC interface for manipulating persistent data [HCF97]. JDBC is a Java based object oriented wrapper for a relational API, which has some functional similarities to ODBC. Whatever the imperfections of the first version of JDBC, for the purposes of a Broker in combination with Java it provides instant, out of the box connectivity from a client on any target platform to any persistent data representation supported by a Broker. Supporting JDBC is different from supporting CORBA. CORBA provides more capabilities than JDBC but requires development on the target platform using CORBA tools. Java and JDBC provide a universal interface that eliminates all barriers to initial use of a Broker.

Widespread use and acceptance of object oriented distributed systems. Perhaps the Internet should be entirely separate from the problem of data inter-operability. Yet in practice is it not. The Internet has forever changed the playing field on which we must deploy systems. Many systems must now meet scale-ability requirements that previously only a small number of systems had to contend with. Not only that but almost every major corporation has been plunged into a competitive environment in which the deployment of these systems is critical. The search for solutions to building highly scalable systems has accelerated the acceptance and deployment of object oriented distributed systems. In an object oriented distributed system it is typical for specialized services to be delegated by business objects to discrete implementations, which will provide critical capabilities such as data access, fault-tolerance, load balancing and so forth. This is in sharp contrast to the way in which earlier distributed systems were built. Earlier distributed systems were more monolithic and essentially required Broker technology, if used, to be platform specific and embeddable. A Broker solution to data inter-operability fits perfectly with the object oriented architectures of these distributed systems. In addition a Broker solution has the ability to scale when other solutions to data inter-operability cannot.

5 DataBroker – A Universal Data Access Broker

Identifying these paradigm shifts in technological underpinnings motivated us to design and implement a Universal Data Access Broker, DataBroker, which is a practical solution to the problem of data inter-operability.

For practical and economic reasons the shipping versions of DataBroker are constrained in terms of the interfaces and the persistent data representations that are supported. For economic reasons we chose to prove the viability of the Broker concept by producing a Broker based implementation of the de-facto standard JDBC interface for Java. Again for economic reasons we chose initially to support persistent representations which were in widespread use and for which there was demand for a high quality JDBC driver. These included both relational databases and mainframe persistent data representations such as DB2 and IMS. However neither of these choices precludes the subsequent use of DataBroker to support other interfaces, or to implement those interfaces against less widely used persistent data representations. In fact I-Kinetics has already experimented both with the development of other interfaces and the development of prototype tools such as a "Batch Plugin" which allow almost any persistent data representation to be "Brokered" into the interfaces supported by DataBroker.

From our experiences with DataBroker we believe that there are several key lessons:

- DataBroker both confirms and leverages the fundamental paradigm shifts we had identified, as outlined in more detail below. In particular we believe we have fully vindicated our belief that CORBA was the perfect choice as an infrastructure for a Broker because of the fundamental emphasis on platform and language inter-operability.

- DataBroker demonstrates that a Broker approach will allow the practical and relatively rapid construction of support for specific interfaces against almost any kind of persistent data representation.
- DataBroker vindicates our belief that not only is a Broker based architecture viable for high performance manipulation of persistent data, in fact a Broker based architecture is able to support the construction not just of competitive implementations of interfaces but of leading implementations. This perhaps deserves a longer discussion, since it reflects on a wider debate on whether two tier or n-tier architectures are more effective.

DataBroker was designed to support data inter-operability, which meets the following practical requirements:

- Robust. A Broker must be capable of 7x24 operation.
- High performance. A Broker must both deliver high performance to an individual client and sustain that performance when hundreds, thousands and perhaps tens of thousands of clients are being supported.
- Scalable. A Broker must be able to meet the demands of deployment on a development Intranet, with a few users, and a production Internet with hundreds, thousands or even tens of thousands of users.
- Adaptable. A Broker must be able to deal with data in predictable formats, like relational database tables, as well as data in less friendly formats, in both files (whether application specific or generic) and non-relational databases.
- Secure. A Broker must be able to provide appropriate security to meet specific business objectives at a cost that matches the benefit and without imposing unwanted overheads on all usage of the Broker.
- Manageable. A Broker must be capable of being deployed and managed across a widely distributed system, thus it should be manageable (in appropriate circumstances) across the Internet and a Broker must integrate with existing standards for managing distributed systems such as SNMP.
- Standard. A Broker must follow both formal and de-facto standards in a way which maximizes the return on investment for users and which as far as possible allows the user to make decisions about technology and tools which are not solely based on the constraints of the underlying persistent data representation.

DataBroker meets these requirements by a combination of high quality engineering and by the explicit leverage of the four paradigm shifts we have previously identified.

The impact of Multi-threading. DataBroker is implemented as a multi-threaded CORBA based server that allows the mapping of multi-threading onto the semantics of the desired interface. For example for JDBC DataBroker uses a session based threading strategy where each thread corresponds to a specific JDBC Connection which is being mapped to a session against the persistent data representation. In addition DataBroker uses multi-threading as an optimization technique at multiple levels. In the implementation of the interface, for example by separate threads for pre-fetch of data. In the implementation of the Broker, by separate threads for pre-initialization of required objects and overheads such as garbage collection. In interacting with the persistent data representation, for example by separate threads for pre-fetch of data and pre-initialization of required objects. In toto we have found that these optimization techniques helped us build a much faster implementation of JDBC than would be possible with a non multi-threaded approach.

The impact of using CORBA. The use of CORBA throughout DataBroker is pervasive. Internally DataBroker uses only CORBA IDL to define system interfaces and IIOP as the communication protocol. Because of this DataBroker integrates seamlessly with capabilities such as Security, Naming and Transactions provided by our Orb vendor [OTM97]. In addition the use of CORBA has allowed us to continuously develop and refine the internal interfaces which we use in the search for performance. Because of this continuous refinement DataBroker now defines an internal protocol which is tightly optimized for the support of interfaces such as OPENjdbc. This allows the combination of DataBroker and OPENjdbc to exceed the performance of JDBC drivers that are implemented using a simple two-tier approach against general-purpose relational database interfaces. This violates conventional wisdom, that introducing an extra tier into a distributed system will slow down performance. In fact using a Broker can actually improve performance.

The impact of Java & JDBC. Initially DataBroker supports both JDBC and a CORBA IDL interface. For pragmatic reasons the initial CORBA IDL interface is itself based on JDBC functionality, with modifications to support usage from "thin" CORBA clients. Overall JDBC has had a pervasive impact on DataBroker. In some ways the first version of JDBC (JDBC 1.22) still presents a low-level object oriented interface to persistent data. But the limitations of the first version of JDBC have been compensated for by the enormous acceptance of the interface by tool vendors and the consequent rapid maturity of the market. For the most part the implementation of the JDBC interface against DataBroker currently supports shallow semantic data inter-operability. We anticipate that JDBC 2.0 will have an equally significant impact on DataBroker in a different way, by emphasizing the importance of a Broker to support multiple interfaces for deep semantic data inter-operability.

Developing DataBroker as a CORBA based component. We chose to develop a Broker that has a component based architecture, anticipating deployment in multi-platform distributed systems. The result of this is that DataBroker can be engineered using the CORBA Orb to provide capabilities such as distributed management and "thin" clients. This also allows us to allow for deployment of DataBroker capabilities as discrete components (plug-ins).

6 A Universal Data Access Broker in Use

In these two examples the problems and system architectures are real, but the company names have been left out.

The first example is a credit card transaction system. This system takes in credit card transactions from various vendors on the Internet and processes them in a central relational database. Volume is high and 7x24 availability is fundamental. The transaction implementation uses JDBC and SQL directly and is functionally simple. A credit card transaction request is fielded, passed to a processing database which checks and flags it and finally a transaction approved or failed response is issued. However the architecture of the system cannot be simple because of the volume of transactions which it must support, there may be as many as 1000 concurrent users. This imposes a high processing load on all of the system components, the web server, the Java JVM and the database. During deployment of the initial version of the system both performance and reliability were unacceptable, the system was taking perhaps ten minutes to respond to incoming transaction requests, connections to the system were failing and the system would crash under stress. None of these problems had been present or obvious from prototype testing.

The system architecture used a single host on which the web server, Java JVM and database were all located. A simple type-4 JDBC driver was used to transmit SQL based operations to the database. Profiling tests demonstrated that:

- The web server, JDBC driver and database server together were consuming up to 6MB of virtual memory for each connection.

- Connections could become blocked in the Java JVM, continuing to consume virtual memory even though the connection was dead.
- The web server was a single point of failure.

Based on these results and other analysis the system architecture was changed to a multi-tier architecture so that:

- The web server would be on a separate dedicated host.
- The database server would be on a separate dedicated host.
- There would be a number of separate hosts each of which would support an instance of a Broker, the JDBC driver for the Broker, the Java JVM and the actual transaction implementation. The initial number of hosts selected was two, but this could be scaled upwards as necessary.

This new system architecture also allowed the Broker to offload connection management from the web server and database server. Because the Broker uses a multi-threaded architecture the virtual memory requirements for each connection dropped dramatically. In addition connections were load balanced from the web server across the hosts running an instance of the Broker. The Broker also supplied a fail-over capability which meant that an attempt to connect to a Broker instance which was somehow unavailable would be automatically routed to a different Broker instance. Dropping the Broker into the architecture did not just replace the JDBC driver with a faster implementation. It also filled in a number of key requirements for scalability which had previously been missing without modifying the transaction implementation.

The second example is a claim processing system. This system already uses a three-tier approach. The middle tier implements a web server and the Java based business logic. Another tier supports the database against which the business logic is implemented using JDBC. Finally the clients represent the third tier, interacting with the business logic tier via HTML and the web server. The system is used by both claims processing agents and 3rd parties, at any one time there may be as many as 3000 users of the system. To support this the development team implemented the above architecture using a high performance Java servlet that exploited multi-threading, connection pooling and caching of data in the business logic. They were confident that the three-tier architecture in combination with this sophisticated implementation of the servlet would scale to meet the requirements.

But it did not scale. The cause was hidden inside the sophisticated servlet. The developers had used a simple type-4 JDBC driver to implement the SQL calls against the database. This turned out to be the weak link in the system architecture. Despite the sophisticated approach used for the servlet the performance of the deployed system foundered because:

- The driver did not stream and cache data efficiently and could not deliver the necessary performance under load.
- The driver did not manage connections efficiently and over time would eventually cause the server to fail.
- The driver was thread-safe, but not multi-threaded. It was not capable of exploiting resources in the same way as the servlet.

To resolve this problem the development team added a Broker and the JDBC driver for the Broker to the middle tier of the system architecture. The multi-threaded architecture of the Broker reinforced the sophisticated implementation of the servlet, rather than adding a weak link to the chain of components that had to scale to meet the requirements.

7 Is a Universal Data Access Broker a viable solution to the problem of data inter-operability?

We believe the answer is an emphatic yes. We believe DataBroker demonstrates the viability of the Broker approach as well as less conventionally accepted conclusions, such as the ability of the Broker approach to deliver the highest performance for the interfaces that it supports.

Of course there will still be problem domains where specific requirements mandate the use of a specific persistent data representation with appropriate characteristics, such as an object oriented database in one case or a relational database in others. However whichever technology is selected for the development of a system in a specific problem domain the problem of data inter-operability with systems in other domains remains. It is guaranteed that these systems will not use the same persistent data representation. We believe that a representation independent Broker represents the only viable approach to delivering practical and usable data inter-operability which will scale to the levels required by the next generation of distributed systems.

References

[CBB97] Cattell, Barry, and Bartels. The Object Database Standard. ODMG 2.0. Morgan Kaufmann, 1997.

[GJS96] Gosling, Joy, and Steele. The Java Language Specification. Addison Wesley, 1996.

[Gei94] K. Geiger. Inside ODBC. Microsoft Press, 1994.

[HCF97] Hamilton, Cattell, and Fisher. JDBC Database Access with Java. Addison Wesley, 1997.

[Kle95] Kleiman. Programming with Threads. Prentice Hall, 1995.

[Lea97] Lea. Concurrent Programming in Java : Design Principles and Patterns. Addison Wesley 1997.

[OLE97] Microsoft OLE DB 1.1 Programmers Reference and Software Development Kit. Microsoft Press, 1997.

[OMG92] Object Management Group. The Common Object Request Broker: Architecture and Specification. 1992.

[OTM97] Orbix OTM Users Guide, IONA, 1997.

[SQL92] Data Management: Structured Query Language (SQL), Version 2. The Open Group, 1996.