Open Metadata Formats: Efficient XML-Based Communication for Heterogeneous Distributed Systems

Patrick Widener Karsten Schwan Greg Eisenhauer
Technical Report GIT-CC-00-21
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332, USA
{pmw,schwan,eisen}@cc.gatech.edu

Abstract

Definition and translation of metadata is incorporated in all systems that exchange structured data. We observe that the manipulation of this metadata can be decomposed into three separate steps: discovery of the metadata, binding of program objects to the message formats represented in the metadata, and marshaling of data to and from wire formats using the metadata. We have designed a method of representing message formats in XML, using datatypes available in the XML Schema specification. We have implemented a tool, xml2wire, that uses such metadata and exploits this decomposition in order to provide flexible metadata definition facilities for an efficient binary communications mechanism. We also observe that the use of xml2wire makes possible such flexibility without intolerable performance effects.

1 Introduction and Background

Today's distributed applications are highly diverse, ranging from E-commerce and B-to-B frameworks, to multimedia and distributed immersive systems, to collaborative design and manufacturing. Yet, most such applications interact via and, therefore, transfer structured rather than unstructured data. Data may describe stock quotes, parts being designed, graphical objects being displayed, or scientific data representing atmospheric volumes and chemical concentrations. Unfortunately, in many such cases, the needs for efficiency in data transfer have led to the creation of domain-specific or closed standards for data structure definition and manipulation. Conversely, where open standards for data structure definition have taken hold, such as CORBA/IIOP, performance considerations often limit their utility. Furthermore, most solutions to data structure definition require the use of specialized programming interfaces, where communicating programs must either agree to use a single communications mechanism or in some other way, reconcile the differences in how their data is defined.

Equally important to performance is the flexibility of data definition, so that applications may evolve, communicating via enhanced data structures. To date, most communication mechanisms perform data definition in a programmatic fashion, thereby embedding the metadata into the communication or application code itself. While this may result in good performance for such mechanisms, it can also result in substantial costs when applications evolve. This is because changes in metadata require consequent modification and recompilation of the codes using such metadata. Furthermore, embedding metadata also `hides' it from just the non-programmer end users to whom it is typically most useful: the engineers designing parts, the physicists studying atmospheric phenomena, and others, sharing such data in their distributed collaborative workspaces.

Our work is predicated on the belief that open metadata systems will become increasingly important and useful, especially for non-programmers using distributed computations that share substantial amounts of data. This belief is validated in part by the increasing popularity of metadata standards like XML. However, the wide acceptance of open metadata systems requires that their use does not strongly impede the performance of the dis-

tributed applications using them. This work seeks to reconcile openness and performance.

Efficient Binary Transmission of Structured Data. Our work addresses interoperability at levels 'below' those of RPC or CORBA, but with functionality exceeding that of common data exchange formats like XDR. Specifically, we are concerned with the efficient movement of the data structures that are defined at the 'system' level of distributed applications, middleware, and systems, typically using system implementation languages like C. Such structured data usually resides in main memory, and when it is moved across heterogeneous machines, one must deal with issues like byte-order, field alignment, and atomic type representation. Furthermore, at this level, data transmission in binary format is critical, due to the high communication bandwidths or low transmission latencies required, or because of the undue processing loads that would be imposed on systems if they were forced to transform information from end user readable formats, like text, to binary formats, for instance. Sample applications requiring binary data transmission include high performance codes moving scientific or engineering data and wide-area transfers of operational data, where scalability to many information clients and sources implies the need to reduce per-client or per-source processing and transmission requirements. They also include server-based applications in which single servers must provide information to large numbers of clients.

**Efficient and `Open' Specification of Data Structure.** In CORBA, the specification of data structure is achieved with IDL specifications. In RPC, procedure parameters are characterized by their types specified within `interface module' descriptions. Openness in metadata definition implies that data structure specifications are not linked to certain transmission mechanisms, such as RPC, or linked to specific protocols, such as those used in the transmission of manufacturing, parts, or design information in the automobile industry, or as those used by specific data storage facilities (e.g., database query languages). Instead, openness requires that data structure may be specified independently of data transmission and use, with translations of such structure to the efficient lower-level representations used for data tranmission or manipulation `hidden' from end users.

The open data structure specification created by our group is based on XML[11]. With XML, metadata or more simply, information about data structure, is moved across communication systems along with the actual data it describes. Unfortunately, existing implementations of communication systems that employ XML use text-based representations of such data. These representations are not acceptable when larger amounts of data must be moved, or when high scalability in terms of number of clients sending data to a single server is required, due to the undue bandwidth and end-system processing requirements of text-based data encodings. In contrast, data transmission in binary format has been shown to provide significant performance gains over transmission in text-based formats, as evident from the ubiquitous use of mechanisms like XDR and others in communication systems. Consequently, our approach combines the use of high level, easily readable and interpretable, and open data structure descriptions using XML, with the efficient transmission of such data using encodings in binary form. Our approach does not predicate the use of specific data delivery mechanisms, which may include standard communication protocols like TCP/IP, wide-area multicast methods like those offered by TIBCO, and high-performance communication methods like those offered for cluster systems (e.g., Myrinet, Giganet, etc.).

The approach to high performance, open metadata systems we have developed decomposes the transfer process for structured data into several components, ultimately resulting in efficient, binary (not text-based) low-level encodings of the data, while also maintaining open user-readable and -comprehensible data structure definitions. More concretely, we describe in this paper a tool that provides flexible metadata definition, using XML, while also supporting high-performance, binary data tranmission. This tool is called `xml2wire'.

The encoding we have chosen for actual data transmission is a step beyond those of current commercial software, which uses XDR to translate data across heterogeneous systems. Instead, we transmit data in NDR (Natural Data Representation), which permits us to move data directly out of memory onto the transmission medium, and directly from the transmission medium into memory, thereby eliminating both computational and copy overheads occurring for XDR-based transmissions.

With this approach, we are able to demonstrate significant performance gains due to elimination of inappropriate `wire formats' for data transmission. Specifically, when transmitting XML data, our NDR-based approach to data transmission demonstrates performance an entire order of magnitude larger than existing, text-based XML transmission approaches. When transmitting structured binary data, we show substantial (often exceeding 50%) performance gains compared to commercial platforms

that use XDR-based data representations.

The reasons for our performance gains are simple: the performance of transmission systems strongly depends on the wire formats they use. Our approach is to entirely eliminate common wire formats like XDR and instead transmit data in the sender's native format (which we term *Natural Data Representation or NDR*), along with efficiently represented meta-information that identifies the precise formats of transmitted data. When data conversion is necessary on the receiving side, it is performed by custom routines created on-the-fly through dynamic code generation.We achieve these gains for XML by separating the discovery and binding tasks, and we posit that as a result, XML becomes a broadly useful, open mechanism for description of data structure.

2 An Application Scenario

As motivation for our work in this area, consider the following example of an airline operational information system. Such a system has several data capture points that provide structured information streams. These information streams might be "live" data on aircraft air or ground movement provided by the FAA or airport authorities, weather information being streamed from geographically remote sources such as other airports or centralized sources such as the NOAA, or they may be produced by data mining processes that periodically examine corporate data stores for trends or patterns of interest.

The communications infrastructure for these streams is a system-wide event backbone. Applications wishing access to the information streams subscribe using the services provided by the event backbone. Examples of applications that are consumers of these streams include data display points (such as real-time aircraft position and weather indicators) and data access points (such as the data terminals used by gate agents to process reservations information). Future data access points may include handheld devices which join the network when activated by their owners and leave the network when their work is done.

Since the data carried by streams has well-defined structure, the applications using such data must become dynamically aware of such structure. Furthermore, realistic models of modern deployed information systems must necessarily assume heterogeneity among the different machine architectures comprising the system. Efficiency concerns dictate that messages be exchanged as rapidly as possible, precluding architecture-neutral wire-format solutions such as ASCII. Also, it is desirable to allow the introduction of new data streams into the system at arbitrary times; this implies that applications should be able to discover the structure of messages sent over these streams without experiencing downtime due to recompilation.

This combination of factors is a difficult one to address using available technology. Efficient communication is possible, but at the cost of inflexibility with respect to admitting new message formats or changing existing ones. Flexibility in message structure is possible, but existing solutions use data transport mechanisms that are inappropriate for high-performance applications. Clearly, the resulting tradeoff between flexibility and performance must be reconciled.

3 The Use of Metadata

All binary communications mechanisms (BCMs) depend on a set of metadata. This metadata describes any messages to be sent in a level of detail that is sufficient for binary transport. These details include but are not limited to characteristics of the compiler in use, word and pointer sizes of the host architecture, and the set of data types available in the programming language being used. Preparing program data for transmission in a binary format requires significant manipulation of the corresponding metadata.

The `xml2wire' tool decomposes metadata usage and finally, the transfer of structured data described by metadata into three components, thereby separating the three actions necessary for the transmission of structured data: (1) discovery, (2) binding, and (3) marshaling. *Discovery* concerns finding the metadata that defines the structure of data being transmitted, whereas *binding* is the association of such metadata with specific data being transmitted. *Marshaling* refers to the actual conversion of data from a structured format to or from a wire format. One issue with existing XML-based data transmission methods, for instance, is that these three components are not separated, thereby unnecessarily forcing data to be transmitted in the same forms in which metadata is being set (i.e., as ASCII text). Instead, we maintain the association of metadata with data at the level of XML, but then use efficient encodings of both for actual data transmission.

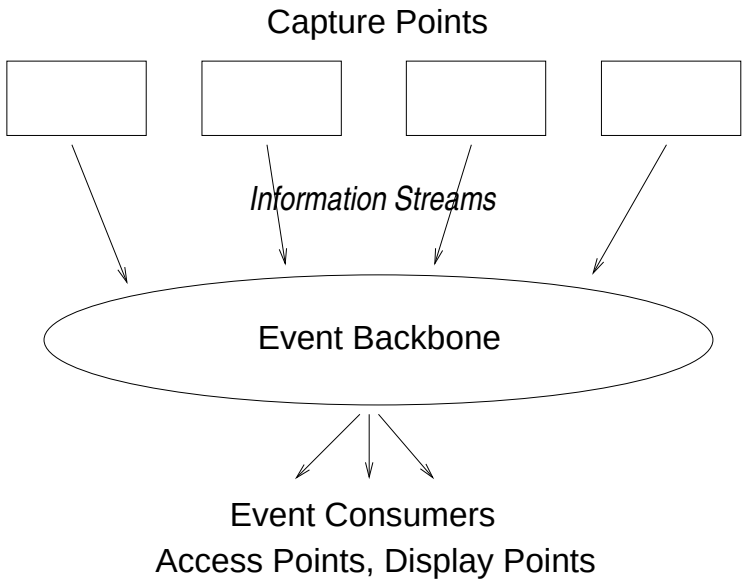

Figure 1. An Airline Operational Information System

3.1 Discovery, Binding, and Marshaling

Discovery Any BCM must somehow *discover* the metadata it will use for transmission of a given message. This discovery process can take several forms. It is possible for a BCM to be built into a program in such a way that the metadata for all possible messages is contained in the program code (for example, the communications stubs produced by an IDL compiler). In such a case, the metadata is directly available and the discovery process is obviated. Designs such as this, however, are difficult to implement correctly and still more difficult to maintain.

Modular programming practice has led to the abstraction of BCMs into library code featuring well-defined interfaces, and consequently to the decoupling of metadata from BCM processing. This decoupling forces the discovery process to become explicit. Several alternatives have been devised to address metadata discovery. At one extreme is an approach only slightly removed from the approach described above, in which metadata is provided in the program code. In such systems, the metadata is provided in a structured manner defined by the BCM, and made programmatically available to the BCM library as part of program execution. This type of approach benefits from being able to make architectural decisions at compile time, which allows direct examination of the behavior of the compiler and examination of the host architecture. However, since the metadata is compiled into the system, it is not readily modifiable (although several systems have taken pains to increase the robustness of such metadata at runtime).

Other approaches rely on Interface Definition Languages (IDLs), which provide an abstract, architecture-independent set of message definitions that all parties to a communication can use. An IDL compiler then translates these definitions for each communicating party on each host platform into architecture-dependent definitions, and code generators provide predefined library code to translate messages at run-time. Still others eschew compile-time definition completely, relying on directory or other lookup services to retrieve metadata. These systems incur an added overhead due to the lookup, but become more flexible due to the fact that the actual metadata is not compiled into the program code.

Binding The discovery process results in the availability of metadata describing the message to be sent. Association of a particular metadata format with a particular piece of program data is a process we call *binding*. In compiled-metadata systems, an API call usually exists to associate a particular program variable with a metadata format. The programmer is responsible for ensuring the compatibility of the program data and the metadata in such systems. In IDL-based systems, format-specific subroutines are generated by the IDL compiler and serve the same purpose. Binding usually results in the construction of some type of message format descriptor or

token which the programmer can use during marshaling.

3.2 Marshaling

The purpose of BCMs is to translate messages from in-memory formats to a wire format. All BCMs define wire formats that define how any data will actually be transmitted over the communications transport layer. These wire formats vary among BCMs, although some standard ones do exist. A message is provided to a BCM as a region in the address space of a process whose contents must be translated to the wire format.

The marshaling process usually takes the following form. The message is considered as a record containing a set of fields. The metadata describes each of these fields: the field size, the layout of the field within the record, and the type of the field. This information is used by the BCM to generate wire format records that contain enough data for the receiver to properly reconstruct the message. Marshaling is not always a one-pass process; some BCMs are designed to convert metadata into an internal representation and only later use the internal representation to generate wire format transport-layer messages.

3.3 Orthogonality

It should be clear that how metadata is provided to a BCM does not in any way influence how that metadata is used for binding or marshaling. Where a BCM uses an internal representation of metadata as described above, for example, it is possible to construct a system that can switch from compiled-in metadata to that provided by a directory server, with each set of metadata being converted to internal representation. In such a system, the method of metadata discovery changes, but the method of metadata binding does not. Conversely, a BCM that relies on IDL descriptions of the messages it sends can make different binding decisions (choosing from among versions of message formats, for example) without affecting the method of metadata discovery.

All communications systems using structured data must perform these three steps. We have seen that there is a fair amount of flexibility in how a BCM performs each step, but the important point is that the steps are independent. We observe that changes in marshaling approaches are most likely to result in incremental performance gains, since marshaling is a repeated process and performance of the BCM is largely determined by the efficiency of the wire format representation is uses. Changes to the discovery process, on the other hand, are not likely to result in performance gains. Message formats change infrequently with respect to the number of messages sent by typical systems, and discovery of metadata is generally performed either only at program startup or else infrequently during program execution.

If changes to the discovery process do not result in significant performance gains, they can result in significant usability gains. Systems in which metadata is provided through compiled information suffer from inflexibility with respect to message format changes. Although infrequent, message format changes in such systems require source-code-level modification and recompilation of all affected installations, a tremendous penalty to pay even infrequently. As noted earlier, some BCMs have gone to great lengths to make compiled-format systems robust in the face of changing or inconsistent message formats, but such features are binding rather than discovery features (in order for the BCM to encounter two inconsistent message descriptions, each description has necessarily to have already been discovered), and in any case this robustness is achieved only through the cooperation of programmers at all endpoints of the communciation.

Introducing an extra layer of abstraction to the discovery process insulates programs from the effects of such changes. Metadata discovery is performed by consulting an extra-process entity, allowing message format changes to be either completely or partially centralized. Although this consultation carries the cost of a network round-trip, the infrequency with which message formats change works in favor of a system using remote discovery. Use of a remote discovery method also has fault-tolerance implications, as a broken network link or hardware failure could leave a remote discovery system without any way of finding the metadata it needs to function. However, a system that uses remote discovery as a primary discovery method and compiled-in information as a fault-tolerant discovery method can provide a useful, if degraded, level of functionality in such failure situations.

4 Implementation of an XML-Based Metadata Tool

In the following section, we will describe the *xml2wire* XML-based metadata tool that employs remote discov-

ery of metadata to provide a high level of flexibility in the definition of message formats. The architecture also allows for the use of compiled-in metadata to cope with failure modes that make remote discovery impossible. This architecture relies on metadata description in XML (Extensible Markup Language), and uses a well-exercised BCM to provide fast binary data transmission.

4.1 Underpinnings

Our system builds on two foundations, which we describe below: an open XML-based schema system for describing data types, and an efficient binary heterogeneous communication package.

4.1.1 XML and XML Schema

XML has been a focus of recent activity in scientific and other computing communities due to its expressiveness in describing and representing arbitrarily complex data structures in a machine-independent manner. XML achieves this platform-independence in a lowest-common-denominator manner, by representing all messages using ASCII text. Entities wishing to use XML to communicate must be willing to convert any transmitted data to and from ASCII. This conversion carries a significant performance cost of its own, and the resulting increase in size of the message to be physically transmitted imposes a further penalty. There are several areas in which the descriptive power available to systems using XML in this manner outweighs the poor performance of such systems, or where small data quantities remove performance as an issue. However, empirical evidence mitigates against the use of XML as a data transport language for high-performance applications [1].

While using XML for data transport is not recommended in high-performance applications, it is possible to apply XML as a metadata description language. Such an XML description would name the structures to be transmitted and their formats, while the actual data transmission would be performed in some other manner. XML is also better suited as a data description language rather than a data content language. For instance, when selecting from among several data streams, the nature (format) of the information being sent along that stream is of primary importance in the selection process rather than the content of the data itself.

The ability to use XML as a metadata definition language for a fast binary transport mechanism provides several advantages. First, it adds a great deal of flexibility in the definition of message structures. Instead of compiling a particular structure definition into a program, a Uniform Resource Locator (URL) can be used instead. The XML description of a message structure (or multiple definitions contained in the same document) can be retrieved and used to set up binary-format messaging. Furthermore, a change in the message structure now becomes a change to the document indicated by the URL, insulating the program from changes in the message format. Second, with the message format represented in XML, the rapidly increasing set of XML document display and manipulation tools can be used. In particular, since the structure of a message will be represented using XML, schema-checking tools will be applicable to live messages received from other parties. This ability could be used to determine which of a set of structure definitions a message most closely fits. Third, the abstraction process inherent in the use of XML for metadata removes the need to consider some platform-dependent features (for example, structure padding). XML-based metadata allows application developers to concentrate more on the structure and content of the message and less on the details of the message transport.

The benefits of using XML for the metadata definition process arise from the ability to define arbitrary types and compose such types into more complex ones. These types are defined by associating Document Type Definition (DTD) documents with XML documents. DTDs provide basic information about the structure of XML documents, but not enough information to support the type of structured data exchange in which we are interested. An additional disadvantage to using DTDs is that they are not themselves XML documents, although it is possible to parse them using commonly available XML parsing software in a relatively straightforward manner.

Application programs draw from a well-understood set of primitive types that can be used for composition: integer, float, character, string (even if their specific definitions vary according to architecture and platform). In order to use XML as a metadata language, a common set of XML definitions must be used to describe the data structures to be transmitted. It is possible to derive a set of arbitrary definitions to be shared by all communicating parties in a system. However, in the interests of interoperability, we prefer to appeal to ongoing open standards defintion efforts. One such effort is the XML Schema[13] specification, under development by the World Wide Web Consortium. XML Schema provides primitive types such as integer, string, and enu-

meration types that can be used to compose new abstract data types. Using these annotations, enough information can be obtained from an XML description of a data structure to drive a fast binary data transport mechanism.

We believe that the use of XML Schema to define communication message formats is a novel contribution, and so we present some details of our approach below.

Composing Message Formats from Primitive Types The basic XML Schema datatypes are referenced by using the XML namespace convention[12]. This makes the entire set of data types in a namespace available to the programmer.

The most "basic" message formats compose a set of data items, each of primitive type, in a particular order to form a message. The XML Schema tag `complexType` is used to denote the definition of a new type[1]. The message forms a type, which is given a name in XML Schema using the `name` attribute in the `complexType` tag.

Composing Message Formats from User-defined Types Inclusion of data elements of a previously-defined type in a new type is accomplished by referencing the name of an already-defined type in a new XML Schema `element`tag. Each `element`tag includes a `type` attribute, which is a character string. In order to use an element of a previously defined type, the `type` attribute is set to the name of the previously defined `complexType`.

Array Types Array types are also specified using attributes of the `element` tag. The type attribute is used to specify the base data type for the array. The maxOccurs attribute is used to specify the array bounds, and may have either a numeric or string value. If the value is numeric, the value will be used as the absolute size of the array. If the value is the wildcard character "*", the array is treated as dynamically-allocated. Lastly, if the value is a string, an element of type `xsd:integer` with an identical name attribute must be present in the structure definition; the value of this variable will be used at runtime to indicate the size of the array. This functionality allows dynamically-allocated arrays to be specified.

[1] XML Schema does allow the definition of new simple types by extension or restriction of primitive types, and these types can be used in the definition of message formats, but we will not illustrate their use here.

4.1.2 PBIO

The wire-format package we use is PBIO [3], developed here at Georgia Tech. PBIO provides facilities for encoding application data structures so that they may be transmitted in binary form over computer networks or written to data files in a heterogeneous computing environment. PBIO relies on compiled knowledge of structure metadata, embedded as language-level constructs in the application program. While PBIO does have the ability to select from among different metadata formats for a specific encoded record, providing flexibility in the face of small structure changes, we sought to remove compiled-in structure formats as a requirement of its operation. However, this ability allows us to retain robustness in situations where remote metadata is unavailable due to network, application, or machine failure. Using a small set of compiled-in message formats, applications can still conduct a basic level of communication (for example, such formats could allow communication with a configuration server or broker in order to determine a new location from which to obtain metadata).

4.2 *xml2wire*

Our realization of the principles we've discussed in this paper is a software tool called *xml2wire*. In this section, we describe the implementation of this tool and how it is used with an efficient binary communication package.

4.2.1 xml2wire Architecture

xml2wire consists of two main modules. The first module is responsible for parsing XML documents containing format definitions and constructing an internal representation of each format. Currently, XML documents are processed by specifying their location in the local file system; however, the architecture of the tool is designed to accept documents indicated by URLs of remote network locations. The second module converts the internal representation of the discovered message formats into a "native" metadata of the underlying BCM. Each module is designed to accept a different compatible parsing engines or BCM, respectively, with minimal integration effort.

4.2.2 Generating and Registering PBIO Metadata

The parser module builds a list of information about the fields as they are encountered in turn. For each message field, a internal *Field* structure containing all the necessary information to register that data element with PBIO is generated. The Field structure is populated as follows:

Field Type The field type is determined from the `type` attribute of the `element` tag. PBIO defines a set of primitive data types such as "integer", "float", "char", and "string". A straightforward mapping is performed between the `type` attribute (which denotes one of the XML Schema data types) and a corresponding PBIO type. One feature of PBIO is that it separates the notion of field type from that of field size; the data type specifies a marshaling/unmarshaling technique to be applied irrespective of the size.

For data types that are built by composition of other previously defined data types, a *Catalog* is kept of known format definitions. The `type` attribute for such a format contains a string which is the name of the previously defined type, and this name is used to retrieve size information from the Catalog.

Field Size Field size is determined by using the C `sizeof` operator on the native data type resulting from the Field Type mapping performed in the previous step. Note that there is no size information specified in the XML format definition. This provides a measure of architecture independence, as "integer" may be a 2-word type on some machines and a 4-word type on others. By providing xml2wire as a run-time toolkit, we are able to record the same size in a format as is used in the application structures that format represents.

Field Offset A naive calculation of field offset simply computes the offset of field n as the sum of the sizes of fields 1 through $n - 1$. However, this approach fails due to padding inserted by compilers to generate more efficient code by ensuring that structures align on word boundaries. PBIO provides facilities to determine these offsets at compile time through a macro which computes the offset according to the structure layout produced by the compiler. xml2wire uses the type parameterization features of C++ to accomplish this task with minimal coding effort. A C++ template function is used to produce this offset value for each data type considered by the tool.

Note that since the platform-dependent calculations are carried out in the same manner and on the same machine which will actually perform the PBIO calls, they are necessarily correct for that machine.

After the entire metadata document has been parsed, a list of Field structures exists that is a functional equivalent to the XML metadata. This list is then used to generate and register PBIO-style metadata. As xml2wire does not perform marshaling, the PBIO objects that represent the newly-registered format are made available to the programmer for later use. Figure 2 illustrates this process.

4.3 xml2wire vs. Generated Metadata

Systems that use IDLs (such as CORBA-based systems and Sun RPC[8] to define data structures for all interested parties employ stub generators to convert the interface definitions into code that performs marshaling and unmarshaling. It would be straightforward to use XML as a foundation for an IDL, using the XML Schema datatypes and composition methods similar to those outlined here, and then convert that IDL into marshaling stubs. However, it is unclear what such an XML-based IDL would have to offer over existing IDL specifications, aside from simply being defined in XML. It is true that IDL compilers could be written to retrieve interface definition modules from remote locations, but as long as compiled-code stubs are generated, applications will still pay a penalty in the form of recompilation when formats change. In contrast, metadata discovery systems that operate at runtime (like xml2wire) allow applications to dynamically react to message format changes.

4.4 xml2wire in the Application Scenario

xml2wire can be used to address problems in the airline operational information system depicted above in Figure 1. xml2wire is used as a run-time tool to discover and register new message formats for use by the underlying BCM. Each participant in communication can discover metadata for any information stream and gain the ability to subscribe to the stream using the event backbone. Figure 3 shows a potential system construction relying on xml2wire for metadata discovery.

Newly created streams can make their metadata available as XML Schema documents on a publicly known intranet server. The server can also be extended to

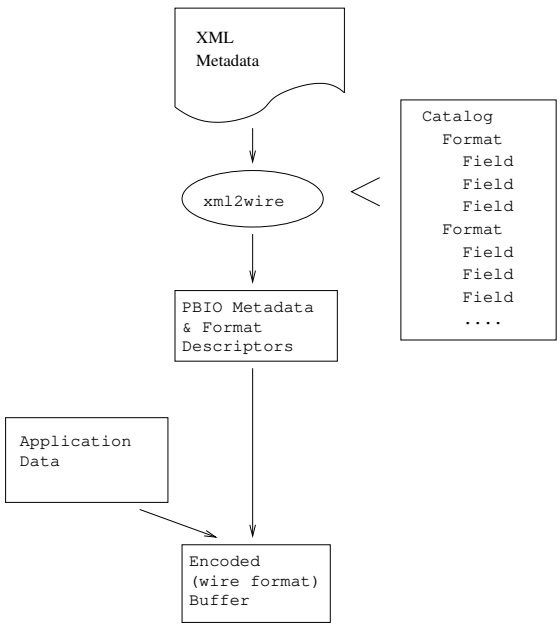

Figure 2. The xml2wire process. Metadata descriptions in XML are retrieved and converted to a *Catalog* of *Format* and *Field* structures. PBIO (or possibly other BCM) metadata is then generated from the *Catalog*.

dynamically generate metadata (recall that each piece of metadata is an XML document) based on information such as requestor location or authentication credentials. With sufficient support from the BCM, this ability can introduce "format-scoping" behaviors where certain "slices" of each information stream are exposed or hidden based on attributes of each subscribing application.

5 Experimental Results

In this section, we discuss performance implications of using XML as a metadata definition language (as opposed to a wire format). Previous research[1] has indicated that the use of XML as a wire format is inappropriate for high-performance systems. This is one of the motivating factors in the development of xml2wire:

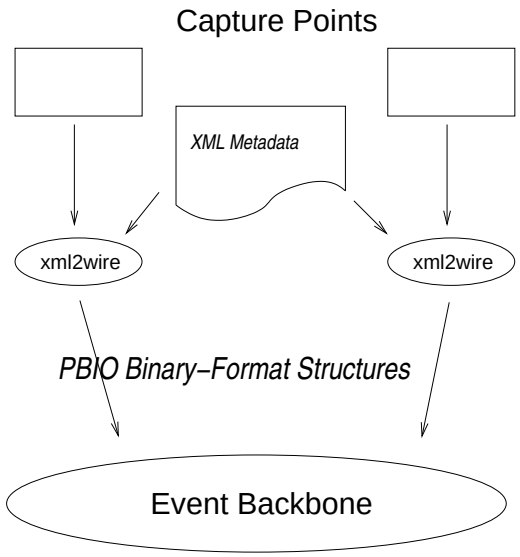

Figure 3. An Airline Operational Information System Using xml2wire

realizing that there are phases in the data communications process where choosing each side of the tradeoff between flexibility and performance is appropriate. We show that the metadata discovery phase as implemented in xml2wire results in a large gain in usability with tolerable impact on performance.

Since the introduction of XML metadata into a system whose communication is performed using PBIO doesn't add any additional overhead to data transport, it isn't meaningful to compare communications times of systems with and without such metadata. Any metadata-related overhead occurs only at program startup; since the approach we describe uses PBIO format registrations derived from the provided XML format descriptions, PBIO-based communications can continue as if normal PBIO metadata were being used.

Table 1 characterizes the time required to parse and register metadata for different structure sizes. Format registration time for xml2wire includes the time necessary to parse the XML description of the format and register the format with PBIO. Structure definitions and the equivalent XML Schema representations can be found in Appendix A.

Table 1 characterizes the time required to parse and register metadata for different structure sizes. Format registration time for xml2wire includes the time necessary to parse the XML description of the format and register the format with PBIO. Structure definitions and the equivalent XML Schema representations can be found in Appendix A. **Structure Size** is the size of the language-level structure in bytes. **Encoded Size** represents the size of a buffer resulting from a marshal operation in PBIO for each metadata approach.

Note that the time required to parse metadata grows proportionally to the structure size. This indicates that the raw overhead of xml2wire does not impose unduly on the metadata discovery and registration process. This experiment was conducted with the structure definitions retrieved from an XML document available on the local file system; network latencies for HTTP retrieval of the document would increase the absolute time required for discovery and retrieval. However, the increase should still remain proportional to the size of the XML document itself.

The final version of this paper will contain measurements of end-to-end latency of communication between two endpoints. These comparisons will illustrate that the overhead introduced by using XML-based metadata is negligible in the context of the total transmission time.

It is important to remember that metadata discovery and registration only occurs at stream subscription time or when metadata changes, and that the associated costs do not recur with each message exchange in an xml2wire-based system. This allows the increased cost of discovery and registration to be amortized across the entire set of messages sent using a particular metadata format. As the number of messages sent in a particular format can reasonably be expected to dominate the number of format discoveries and changes, this implies that the overall effect on performance will be tolerable[2].

6 Related Work

In a sense, most research on high-performance computing that involves the exchange of structured data is to some degree related to our work. All such packages have some facility for describing the structure of messages that will be exchanged and governing the translation of messages into a specific wire format. Their differences with respect to this research lie in how they define and manage this metadata, and to what degree they can make use of efficient communication mechanisms.

Packages such as PBIO, PVM[6], and Nexus[5] support language-level features for defining messages in which the communicating applications “pack” and “unpack” messages, building and decoding them field by field.. In such high performance communication packages, the operational norm is for all parties to a communication to have an *a priori* agreement on the format of messages exchanged. PBIO does support a form of restricted evolution in message formats in which elements may be added to message formats without causing receivers of previous versions of the message to fail. Other packages, such as MPI[4], allow the creation of user-defined datatypes for messages and fields and provide some marshalling and unmarshalling support for them internally.

Another general class of communication systems use IDLs to define the structure of messages. This class of systems, including Sun RPC[8], and CORBA[7], provides an implementation-language-independent set of datatypes that can be arbitrarily composed. The resulting structure definitions are processed by a code generator that produces implementation-language code that is included in the application program. Our use of XML Schema serves much the same purpose, with the added benefits of interoperability and the active state of development on analysis and verification tools. We note that there is no available specification for automated exchange of IDL definitions, even though CORBA IDL is

[2]Note also that in both compiled-metadata and IDL-metadata systems, format changes require human intervention at every source and sink point (in the form of recompilation of systems that cannot cope with the format changes). The resulting effect on end-to-end latency is, at best, non-trivial.

Structure Size (bytes)	Encoded Size		Format Registration Time (ms)	
	PBIO	*xml2wire*	*PBIO*	*xml2wire*
32	72	72	.102	.191
52	104	104	.110	.225
180	268	268	.158	.304

Table 1. Format registration costs using xml2wire and PBIO

relatively mature; by contrast, exchanging metadata defined in XML leverages (nearly) ubiquitous HTTP transport services.

A third class of systems uses object systems technology, providing for some amount of plug-and-play interoperability through subclassing and reflection. This is a significant advantage in building loosely coupled systems, but they tend to suffer when it comes to communication efficiency. For example, CORBA-based object systems use IIOP as a wire format. IIOP attempts to reduce marshalling overhead by adopting a "reader-makes-right" approach with respect to byte order (the actual byte order used in a message is specified by a header field). This additional flexibility in the wire format allows CORBA to avoid unnecessary byte-swapping in message exchanges between homogeneous systems but is not sufficient to allow such message exchanges without copying of data at both sender and receiver. xml2wire compares favorably to this class of systems. Both commercially [14] and freely available XML-parsing packages [2, 9] allow run-time investigation of message formats, providing functionality equivalent to reflection. Also, our integration with an efficient wire-format provides high-performance communication.

While all of the communication systems above rely on some form of a fixed wire format for communication, systems using XML as a wire format[10] take a different approach to communication flexibility. Rather than transmitting data in binary form, their wire format is ASCII text, with each record represented in textual form with header and trailer information identifying each field. This allows applications to communicate with no a priori knowledge of each others. However, XML encoding and decoding costs are substantially higher that those of other formats due to the conversion of data from binary, im-memory format to ASCII and vice-versa. In addition, XML has substantially higher network transmission costs because the ASCII-encoded record is larger, often substantially larger, than the binary original (an expansion factor of 6-8 is not unusual [1].

7 Summary, Conclusions and Future Work

This paper has presented the motivation for and details of the implementation of xml2wire, a tool for run-time discovery of metadata for high-performance communications. In addition, we have detailed a original method of constructing metadata based on data types taken from the XML Schema specification. We have shown that definition of metadata using our approach is conceptually simpler than relying on language-level metadata mechanisms, and that the overhead of using xml2wire is tolerable, especially when used in systems with large amounts of message traffic.

Currently the system we have built relies on the existence of XML description documents on a file system directly accessible by the programs using them. It is our intention to add a format registration mechanism on top of PBIO that incorporates the HTTP protocol so that the XML descriptions of PBIO formats can be retrieved from remote locations in the same manner that web browsers retrieve other XML documents. We also intend to explore dynamic incorporation of new message formats into applications at run-time, as well as generation of language-level message object representations in both the C++ and a planned Java version of xml2wire.

References

[1] Fabian Bustamante, Greg Eisenhauer, Karsten Schwan, and Patrick Widener. Efficient wire formats for high performance computing. In *Proceedings of Supercomputing 2000*, November 2000. To appear.

[2] James Clark. expat - xml parser toolkit. http://www.jclark.com/xml/expat.html.

[3] Greg Eisenhauer and Lynn K. Daley. Fast heterogenous binary data interchange. In

Proceedings of the Heterogeneous Computing Workshop (HCW2000), May 3-5 2000. http://www.cc.gatech.edu/systems/papers/Eisenhauer00FHB.pdf.

[4] Message Passing Interface (MPI) Forum. MPI: A message passing interface standard. Technical report, University of Tennessee, 1995.

[5] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, pages 70–82, 1996.

[6] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 Users Guide and Reference manual*. Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 94.

[7] Object Management Group. Common object request broker architecture. http://www.omg.org/corba2/.

[8] Sun Microsystems. Xdr: External data representation standard. IETF RFC 1014.

[9] The Apache XML Project. Xerces xml parser. http://xml.apache.org/xerces-c/index.html.

[10] Inc. UserLand Software. Xml-rpc specification. http://www.xmlrpc.com/spec/.

[11] The World Wide Web Consortium (W3C). The extensible markup language. http://www.w3.org/TR/1998/REC-xml-19980210, 1998.

[12] World Wide Web Consortium (W3C). Namespaces in xml. http://www.w3.org/TR/1999/REC-xml-names/.

[13] World Wide Web Consortium (W3C). Xml schema. http://www.w3.org/XML/Schema/.

[14] www.xml.com Buyer's Guide. Xml.com - buyer's guide. http://www.xml.com/pub/buyersguide/.

A Structure, PBIO Metadata, and XML Schema Definitions

```
typedef struct asdOff_s
{
  char* cntrId;
  char* arln;
  int fltNum;
  char* equip;
  char* org;
  char* dest;
  unsigned long off;
  unsigned long eta;
} asdOff;
```

Figure 4. Structure A with no arrays and no nesting.

```
IOField asdOffFields[] =
{
  { "cntrID", "string", sizeof (char*), IOOffset (asdOffptr, cntrId) },
  { "arln", "string", sizeof (char*), IOOffset (asdOffptr, arln) },
  { "fltNum", "integer", sizeof (int), IOOffset (asdOffptr, fltNum) },
  { "equip", "string", sizeof (char*), IOOffset (asdOffptr, equip) },
  { "org", "string", sizeof (char*), IOOffset (asdOffptr, org) },
  { "dest", "string", sizeof (char*), IOOffset (asdOffptr, dest) },
  { "off", "integer", sizeof (unsigned long), IOOffset (asdOffptr, off) },
  { "eta", "integer", sizeof (unsigned long), IOOffset (asdOffptr, eta) },
  { NULL, NULL, 0, 0 }
};
```

Figure 5. PBIO metadata for Structure A.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema"
targetNamespace="http://www.cc.gatech.edu/ pmw/schemas">

<xsd:annotation>
<xsd:documentation>
ASDOff
</xsd:documentation>
</xsd:annotation>

<xsd:complexType name="ASDOffEvent">
  <xsd:element name="cntrID" type="xsd:string" />
  <xsd:element name="arln" type="xsd:string" />
  <xsd:element name="fltNum" type="xsd:integer" />
  <xsd:element name="equip" type="xsd:string" />
  <xsd:element name="org" type="xsd:string" />
  <xsd:element name="dest" type="xsd:string" />
  <xsd:element name="off" type="xsd:unsigned-long" />
  <xsd:element name="eta" type="xsd:unsigned-long" />
</xsd:complexType>

</xsd:schema>
```

Figure 6. XML Schema-based metadata for Structure A.

```
typedef struct asdOff_s
{
  char* cntrId;
  char* arln;
  int fltNum;
  char* equip;
  char* org;
  char* dest;
  unsigned long off[5];
  unsigned long *eta;
  int eta_count;
} asdOff;
```

Figure 7. Structure B with static and dynamically-allocated arrays.

```
IOField ASDOffFields[] =
{
  { "cntrID", "string", sizeof (char*), IOOffset (asdOffptr, cntrId) },
  { "arln", "string", sizeof (char*), IOOffset (asdOffptr, arln) },
  { "fltNum", "integer", sizeof (int), IOOffset (asdOffptr, fltNum) },
  { "equip", "string", sizeof (char*), IOOffset (asdOffptr, equip) },
  { "org", "string", sizeof (char*), IOOffset (asdOffptr, org) },
  { "dest", "string", sizeof (char*), IOOffset (asdOffptr, dest) },
  { "off", "integer[5]", sizeof (unsigned long), IOOffset (asdOffptr, off) },
  { "eta", "integer[eta_count]", sizeof (unsigned long), IOOffset (asdOffptr, eta) },
  { "eta_count", "integer", sizeof (unsigned long), IOOffset (asdOffptr, eta_count) },
  { NULL, NULL, 0, 0 }
};
```

Figure 8. PBIO metadata for Structure B.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema"
targetNamespace="http://www.cc.gatech.edu/ pmw/schemas">

<xsd:annotation>
<xsd:documentation>
ASDOff
</xsd:documentation>
</xsd:annotation>

<xsd:complexType name="ASDOffEvent">
  <xsd:element name="cntrID" type="xsd:string" />
  <xsd:element name="arln" type="xsd:string" />
  <xsd:element name="fltNum" type="xsd:integer" />
  <xsd:element name="equip" type="xsd:string" />
  <xsd:element name="org" type="xsd:string" />
  <xsd:element name="dest" type="xsd:string" />
  <xsd:element name="off" type="xsd:unsigned-long" minOccurs="5" maxOccurs="5" />
  <xsd:element name="eta" type="xsd:unsigned-long" minOccurs="0" maxOccurs="*" />
</xsd:complexType>

</xsd:schema>
```

Figure 9. XML Schema-based metadata for Structure B.

```
typedef struct asdOff_s
{
  char* cntrId;
  char* arln;
  int fltNum;
  char* equip;
  char* org;
  char* dest;
  unsigned long off[5];
  unsigned long *eta;
  int eta_count;
} asdOff;

typedef struct threeAsdOff_s
{
  asdOff one;
  double bart;
  asdOff two;
  double lisa;
  asdOff three;
} threeAsdOffs;
```

Figure 10. Structures C and D with arrays and composition by nesting.

```
IOField ASDOffFields[] =
{
  { "cntrID", "string", sizeof (char*), IOOffset (asdOffptr, cntrId) },
  { "arln", "string", sizeof (char*), IOOffset (asdOffptr, arln) },
  { "fltNum", "integer", sizeof (int), IOOffset (asdOffptr, fltNum) },
  { "equip", "string", sizeof (char*), IOOffset (asdOffptr, equip) },
  { "org", "string", sizeof (char*), IOOffset (asdOffptr, org) },
  { "dest", "string", sizeof (char*), IOOffset (asdOffptr, dest) },
  { "off", "integer[5]", sizeof (unsigned long), IOOffset (asdOffptr, off) },
  { "eta", "integer[eta_count]", sizeof (unsigned long), IOOffset (asdOffptr, eta) },
  { "eta_count", "integer", sizeof (unsigned long), IOOffset (asdOffptr, eta_count) },
  { NULL, NULL, 0, 0 }
};


IOField threeASDOffFields[] =
{
  { "one", "ASDOffEvent", sizeof (asdOff), IOOffset (twoAsdOffsPtr, one) },
  { "bart", "double", sizeof (double), IOOffset (twoAsdOffsPtr, bart) },
  { "two", "ASDOffEvent", sizeof (asdOff), IOOffset (twoAsdOffsPtr, two) },
  { "lisa", "double", sizeof (double), IOOffset (twoAsdOffsPtr, lisa) },
  { "three", "ASDOffEvent", sizeof (asdOff), IOOffset (twoAsdOffsPtr, three) },
  { NULL, NULL, 0, 0 }
};
```

Figure 11. PBIO metadata for Structures C and D.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema"
targetNamespace="http://www.cc.gatech.edu/ pmw/schemas">

<xsd:annotation>
<xsd:documentation>
ASDOff
</xsd:documentation>
</xsd:annotation>

<xsd:complexType name="ASDOffEvent">
  <xsd:element name="cntrID" type="xsd:string" />
  <xsd:element name="arln" type="xsd:string" />
  <xsd:element name="fltNum" type="xsd:integer" />
  <xsd:element name="equip" type="xsd:string" />
  <xsd:element name="org" type="xsd:string" />
  <xsd:element name="dest" type="xsd:string" />
  <xsd:element name="off" type="xsd:unsigned-long" minOccurs="5" maxOccurs="5" />
  <xsd:element name="eta" type="xsd:unsigned-long" minOccurs="1" maxOccurs="*" />
</xsd:complexType>

<xsd:complexType name="threeASDOffs">
  <xsd:element name="one" type="ASDOffEvent" />
  <xsd:element name="bart" type="xsd:double" />
  <xsd:element name="two" type="ASDOffEvent" />
  <xsd:element name="lisa" type="xsd:double" />
  <xsd:element name="three" type="ASDOffEvent" />
</xsd:complexType>

</xsd:schema>
```

Figure 12. XML Schema-based metadata for Structures C and D.