

Taking the Step From Meta-information to Communication Middleware in Computational Data Streams

Beth Plale, Patrick Widener, and Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332
{beth,pmw,schwan}@cc.gatech.edu

Abstract

It is our belief that network applications relying on globally distributed shared resources will increasingly adopt meta-level descriptions to describe the data streaming in the application at runtime. Our group has developed the notion of computational data streams to describe and act upon such data flows. Our work conceptualizes the data flows as database relations over which useful operations, such as querying, can be performed. This paper shows how one can make the step from a meta-level description of data flows to an actual implementation using CORBA-style event channels and binary I/O for data transport.

1 Introduction

The high-performance computing community is focusing effort on interoperability, particularly in the context of realizing the model of grid computation [5]. Emerging network applications in areas such as HPC, information analysis, and distributed collaboration involve use of multiple geographically distributed data resources having vastly different storage and access representations. The development of such applications can be significantly simplified by using meta-level descriptions of the data flow. Meta-level information, expressed by representations such as a relational schema or XML document, is an agreed upon description of the data that is independent of the underlying wire or storage format. Further, meta-descriptions can contain higher level abstractions (*e.g.*, molecules, atoms, electrons) in such a way that tools can operate on the abstraction. We believe that network applications relying on shared resources will increasingly adopt meta-level descriptions of data for data exchange, particularly when one considers that end-users of the systems are atmospheric scientists, physicists, etc.

At Georgia Tech, we have developed a system, dQUOB, that allows a user to extract and operate on data from a network application data stream. A strength of the work is that it allows a user to conceptualize the data streaming in a network application as a set of tables in a relational database, and extract relevant information from the data stream by specifying an SQL query. Specifically, a data stream can contain multiple event types, and a network application can have any number of data streams active at any moment. The relational view of data streams considers events of the same event type to be tuples of the same relation (*i.e.*, table). The user interface to dQUOB requires the user to provide 1.) a relational schema describing events flowing in the data streams, 2) SQL queries describing the data he/she needs, and 3) optional functions to be applied to the data extracted via the query.

The unique contribution of this paper is two-fold. First, we demonstrate the usefulness of a query language for extracting data from data streams through a series of examples. The examples show the expressiveness of the language and its applicability for scientific computing environments. Second, we show a framework for mapping from the meta-level relational schema to an underlying communication middleware layer, thus saving the user from specifying the low-level details. This involves two distinct tasks: first, a mapping from a relational schema to the specification and creation of specific event channels, and second, from the meta-representation of data to the low-level data formats.

2 dQUOB System Overview

dQUOB is a model of event data transfer, which I refer to simply as data streams and a software system, by which one can create computational entities that manipulate data streams. dQUOB is implemented as a peer-to-peer communication package; CORBA event service is a popular example. CORBA event service models data suppliers, data consumers, event channels as intervening objects, and events as the unit of data transfer. The model provides a publish-subscribe semantics that allow a supplier to publish events unbeknownst of the type and number of consumers.

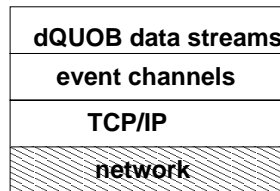


Figure 1: The dQUOB System.

dQUOB addresses application level issues in data transfer. That is, it assumes the underlying wire format resolves issues like byte ordering and word size inconsistencies that exist amongst heterogeneous platforms, and focus instead on broader application-level issues such as "user X needs a subset of fields from the union of events A, B, and C," or "user Y needs units in PPM where the supplier publishes them in PPB". The dQUOB data stream model, shown in Figure 1, defines a data stream in terms of a logical group of one or more event channels, one or more suppliers, usually a single consumer, intermediate computation called quoblets, and a relational data model of events. Quoblets are executable entities that physically reside at the supplier, consumer, or at points between. Quoblets serve to transform, aggregate, or filter the data. For example, to reduce downstream bandwidth needs, an aggregation algorithm might be employed to sum values over neighbor points in a 3D space. But quoblets can also manipulate the data stream itself. A data stream can be envisioned as a river. A river is fed by multiple tributaries upstream while downstream it flows into a delta region. With sediment buildup in the delta region, the river appears to split into multiple streams each of which terminate at the ocean. With this analogy, suppliers are located at the origin of the tributaries and consumers at the estuary edge. The role of quoblets is to form the river flow, that is, by join-style quoblets posted at the confluence of tributaries and view-style quoblets posted in the delta region to narrow the flow to a user's particular estuary.

A strength of the dQUOB data stream model is the relational data view it imposes on events which enables the user to conceptualize a data stream as a set of tables in a relational database. This enables information of interest to be extracted by specifying SQL queries. Specifically, a data stream can contain multiple event types. The relational view of data streams considers events of the same type as tuples of the same relation (*i.e.* table). As a consequence of the model, combining event channel flows can be conceptualized as a join over tables and splitting a data flow can be conceptualized as creating a "view", in which events of different types contribute to the creation of a new event type.

In the following section we highlight some of the advantages of taking a relational view to data streams.

2.1 Advantages of a Relational Data Model

By way of introduction, the relational data model is of data organized into tables (*i.e.* relations). A table consists of a set of attributes where an attribute corresponds to one column of the table. An attribute is associated with a domain indicating the set of values the attribute can take. Each row of a table is called a tuple and describes one real-world entity and/or relationship among several entities. A schema defines the tables, attributes, and domains for a given application [18].

The relational database model has a strong and well defined mathematical foundation in *relational algebra*, a collection of operations used to manipulate relations. Operations consist of selection (σ), projection (π), Cartesian product (\times), union (\cup), intersection (\cap), set difference ($-$), join (\bowtie), and division. SQL is the ANSI and ISO standard query language for relational databases. A recent proposal to ISO has petitioned for the addition of temporal operators to SQL. Temporal operators extend the expressiveness of SQL to include requests for data exhibiting time dependent relationships. A weakness of the relational data model is its inadequate support for complex data. That is, attributes

must be simple data types. We work around this by representing array data as a single attribute that is opaque to the query engine and are exploring object-relational data model features to extend query expressibility to complex data.

A significant number of the unique advantages can be accrued by a relational data model view of data streams. We discuss these below.

Declarative Query Language. SQL queries are specified declaratively, that is, users need only specify *what* data they want, not *how* the data is to be retrieved. The optimization potential is obvious: we can create a more efficient version of a query than the one given to us by the user. Procedural queries, on the other hand, embed an explicit order of execution. The latter type query is more common than one might think. A conditional request for data written in a procedural programming language such as C, C++, or Java is obviously procedural, but so are the query languages associated with today's hierarchical databases. In fact, the hierarchical data model¹, which is receiving renewed interest in the HPC community through applications like DNS and the LDAP/X.500 protocols, has a procedural query language that historically has had low-level and operational with COBOL-like operations such as “get leftmost” and “get next” [7]. A declarative query language is an important prerequisite to the non-trivial optimizations of queries.

Single Type. The relational data model supports a single type: the relation. That is, data is stored in relations (or tables), queries accept relations as input, and produce relations as output. This simplicity is not present in other data models. In the object data model, for instance, upon which object-oriented database management systems are built, the output of a query can be a set, bag, tuple, list, or array. The uniform view yields more efficient query processing and also simplifies the way in which one can think about streaming data and data stream processing.

Materialized View. A materialized view is the stored results of a query. dQUOB materialized views serve as a means to successively refine data by leveraging off ‘upstream’ queries. One can visualize this by analogy to a river. Drawing from the river analogy presented earlier, one can envision a client at each endpoint of the river. A materialized view can be viewed as a query inserted at each point in the delta region where the river splits. Materialized views, then, are a way by which a user can conceptualize specifying the data he/she needs from data streams. Further, the task can be made simpler by leveraging off existing upstream views.

Simplified Component Programming. Typical computational components perform action on data, respond to monitoring commands, and respond to steering commands from a user. For instance, a component might convert the units of atmospheric data from parts per million to parts per billion. Meanwhile, it could be responding to requests to modify itself in response to changes in network bandwidth, and to requests from users to modify the region of data in which the scientist is interested. Traditionally, each type of command is serviced separately, often via separate threads, making for a more complex programming task. By treating monitoring requests and steering requests as relations, however, the decision-making can be expressed within the framework of the query evaluator, thus reducing the complexity of a data stream component.

Relational Algebra. Relational algebra defines a small number of operations. This is one of the reasons why relational query engines are efficient; and it is their efficiency that is a leading reason for the popularity of relational databases. Most object-based query languages, on the other hand, while providing a SQL-like syntax, have difficulties achieving efficient query evaluation, in part by the very features that give the object model its enhanced conceptualization; specifically complex objects and class hierarchy. *Complex objects* are objects having nested references to other objects. The resulting path expressions are a key feature in object queries, and a key difficulty in creating efficient queries. A *class hierarchy* is a set of objects related by a parent-child relationship. A query must access and return objects in some or all classes in a class hierarchy.

It is our experience that events are generally independent in the sense that they do not contain nested relationships to other events, nor are they inextricably tied to a class hierarchy. Thus the additional conceptual expressiveness of the object data model is not worth the additional complexity and loss of efficiency that accompany it.

3 Query Language Through Examples

The viability of the dQUOB approach is determined in large part by the expressiveness of its query language. In this section we demonstrate usefulness of the language with a series of examples drawn from a sample scientific application.

¹The hierarchical data model is defined by entities of one type which own or are parents of entities of a second type, forming a tree with the root at the top and leaves at the bottom.

The dQUOB language adopts the create-if-then rule construct of the Starburst [17] query language for active database systems. The *create*-clause creates a named rule, the *if*-clause contains an SQL query, and the *then*-clause is a set of actions to be executed when the query is satisfied. The SQL query has a select-from-where three-clause syntax. The *select*-clause specifies the attributes whose values are to be retrieved as well as the structure into which the retrieved values are to be organized. That is, it defines the format of the outbound event. The *from*-clause specifies the classes from which data are to be retrieved as well as the structure into which the retrieved values are to be organized; that is, it is a list of the inbound events. The *where*-clause specifies the conditions to be satisfied.

The data stream example we use is the visualization of 3D atmospheric data generated by a parallel and distributed global atmospheric model developed at Georgia Tech. The model consists of an atmospheric transport model that simulates the flow of chemical species, specifically ozone, through the stratosphere coupled with a chemical model that models the interaction of the ozone with short lived species (e.g., CH_4 , CO , HNO_3). Species data is pushed from the model each logical timestep (i.e., 2 hrs. of modeled time). A 3D gridpoint is defined by the tuple (level, latitude, and longitude). 'Level' corresponds to an atmospheric pressure. Through a series of small examples drawn from the atmospheric visualization we illustrate and discuss the query language.

Example 1: Rule construct, Select. The following rule, named C:1, is a simple query to retrieve data for the upper atmospheric levels of the Antarctic circle, stated by the conjunction of two select expressions. The data records that satisfy the query are passed to the function, ppm2ppb, which convert the grid points in the 3D slice from parts-per-million to parts-per-billion.

```
CREATE RULE C:1 ON Data_Ev
IF
  SELECT Data_New_Ev
  FROM Data_Ev as d
  WHERE
    d.latitude_min <= -63.68 and
    d.level_min >= 30 and
THEN
  FUNC ppm2ppb
```

Example 2: Boolean Operators, Join. The second example accepts two event types: data events from the atmospheric model as Data_Ev, and a user request for a particular region of data as Request_Ev. For illustration purposes, the user requested region is only one of two 3D points at selected latitudes. The query evaluates to true if the data event contains the data for the longitude and level in which one or the other requested latitudinal points appear. (The negation boolean operator, not shown, is supported as well.)

```
CREATE RULE C:2 ON Data_Ev, Request_Ev
IF
  SELECT Data_Ev
  FROM Data_Ev as d, Request_Ev as r
  WHERE
    ((r.lat_point1 >= d.lat_min and
     r.lat_point1 <= d.lat_max) or
     (r.lat_point2 >= d.lat_min and
     r.lat_point2 <= d.lat_max) or
     d.aid = r.aid
```

Though implicit, the join operation exists when a select is specified over multiple event types. A join is a Cartesian product over two tables coupled with a condition as follows; a pair of events satisfies the join if the time-based condition over the pair evaluates to true. Specifically, for two events α and β , the Cartesian product, $\alpha \times \beta$, is taken if the timestamp of α equals the timestamp of β plus or minus some small ϵ .

To perform joins over a data stream, one encounters the problem that at no time during execution is the entire 'table' available. Because of the limits of on-chip and L2 cache memory and the serious performance delays that one might incur by utilizing disk files for buffer storage, a 'table' can often not even contain every event received up to a particular moment. Thus, queries must be evaluated over partial data, where the data available at any moment for query evaluation can be viewed as a sliding window over the table.

We bound the window size for performance reasons, but as window size decreases, the risk of false negatives increases. A *false negative* is a match that should have been made that is missed [9]. A reasonable window size depends strongly on the temporal ordering of the two input streams used in the join and on their relative synchrony [14]. For two event streams, each with a guaranteed partial order and containing an event for every logical timestep, minimal

storage would be required. But if ordering or relative synchrony are absent, the window size is potentially unlimited. In ongoing work, we are investigating the impact of false negatives on the types of queries a user may issue.

Example 3: Project. dQUOB supports *materialized views* defined as the stored results of a query. Put in context, this means that the result of a query can be a new relation that can serve as input to a downstream query. In the following example, a user wishes to be alerted when a particular condition becomes true. The alert message, `Notif_Ev`, is a relation whose source for tuples is the query consisting of three attributes: model timestep and timestamp from the atmospheric model event, `Data_Ev`, and user name from the user request, `Request_Ev`. The project operator creates a notification event, `Notif_Ev`, having these three attributes when the user has issued a request and the model has advanced to the stated logical timestep (e.g. 4270).

```
CREATE RULE C:3 ON Data_Ev, Request_Ev
IF
  SELECT Notify_Ev as d.timestep, d.timestamp,
           r.userName
  FROM Data_Ev as d, Request_Ev as r
  WHERE
    d.timestep = 4270 and d.aid = r.aid
```

Example 4: Temporal and Time-based Operators. dQUOB supports the `'meets'` and `'precedes'` temporal operators of the temporal relational database query language, ATSQL [13]. The expression α MEETS β evaluates to true if there exists two events α and β such that the start time of β is less than or equal to the end time of α but not less than the start time of α . The expression α PRECEDES β evaluates to true when for an event β there exists an event α in the history buffer such that the timestamp of α is earlier than the timestamp of β . As with the join, the `'precedes'` operator potentially requires infinite storage.

The example considers a stream of frames, each marked with a start and end time. Suppose the user wishes to perform a difference over a pair of sequential frames. The query generates a pair of sequential frames that are passed to the difference function, `diff()`. The resulting difference frame is generated as `Frame_Diff_Ev`.

```
CREATE RULE C:4 ON Frame_Ev
IF
  SELECT Frame_Diff_Ev
  FROM Frame_Ev as f1, Frame_Ev as f2
  WHERE
    f1 meets f2
THEN
  diff
```

4 From Relational Schema to Event Channel Configuration

At Georgia Tech our work [11] and the work of others in our group [6] has established the viability of the computational data stream. A *computational data stream* is a data stream with computation inserted at the source, destination, or at intermediate points between. The computations serve to transform, aggregate, or filter the data. For instance, aggregation might be used to sum values over neighbor points in a 3D space to reduce downstream bandwidth needs. Additionally, our group has developed a CORBA-style publish, subscribe event middleware layer, called ECho [2] which creates event channels as the transport vehicle. ECho relies upon a reader-makes-right data format, P BIO [3] for binary I/O. Computational data streams are one of the underlying mechanisms of the Infosphere project [12]. Their viability has been considered by others in [4], [1], and [8].

It is common to start up a network application by first explicitly creating each event channel, passing the channel ID for the created channel as an input parameter to each of the processes at startup. The programmer must also explicitly code the data format descriptions in the form that the binary library demands. The need for using binary I/O to stream data has been established by earlier measurements by our group [2] comparing the cost of binary I/O to an XML encoded data stream transmitted as ASCII.

We have approached the problem of explicitly having to create event channels and declaring binary formats by leveraging the meta-information available to us by our relational view of data streams. Specifically, for a user to issue a query to a database management system, she needs to know the SQL query language, the relations in the database, and the attributes of the relations in which she is interested. A description of relations, attributes, and the data type of the attributes together form the *schema*; the schema is frequently referred to as *metadata*. Establishing a query over

a data stream also requires a query and a schema. For example, the simple query shown in Example 1 in Section 3 requires a single relation, `Data_Ev`. `Data_Ev` has at least two attributes, `latitude_min` of type `real`, and `level_min`, of type `integer`. `'ppm2ppb'` is a user supplied function that is executed whenever the query evaluates to true. But whereas in a traditional database management system the schema maps to a storage representation internal to the DBMS software, then eventually to the low level file system of the O/S, a schema for a data stream, on the other hand, maps to one or more binary I/O formats and event channels. This paper shows ongoing work in mapping from a meta-level relational schema and set of query/action rules to the underlying “storage medium”, which in our case is data passing through event channels.

4.1 Creating Queries

A query, for instance Example 1 in Section 3, consists of the query action rule, which is the text shown in courier font in the example, and a user supplied function written in C or C++ (named `ppm2ppb`). To create a query, the query and a relational schema is provided to the query compiler, as shown in Figure 2. The compiler compiles and optimizes queries, then generates for each query a Tcl script. The query script and user function are stored to a data repository that has an HTTP interface.

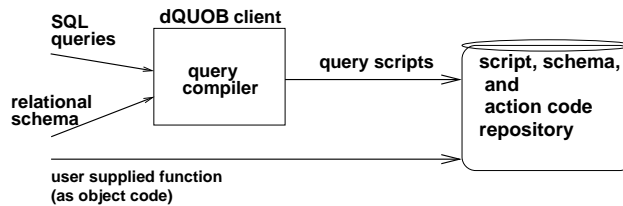


Figure 2: Query compiler.

4.2 Establishing Event Channels

The mapping from a relational schema to a particular event channel configuration is accomplished by creating one event channel for every database relation (*i.e.* table). Since a relation naturally corresponds to an event type, and all events of a particular type are considered tuples of the relation (or rows of the table), in essence, we are creating one event channel per event type. This notion is reasonable; ECho endorses the notion with its built-in support for what it calls “typed event channels”, or event channels carrying events of a single type.

To establish an event channel configuration, the dQUOB server, shown at the top right of Figure 3, obtains the relational schema from the repository via an HTTP GET request, extracts the names of the relations from the relational schema, and for each relation, creates an event channel having the same name as the relation. The names are guaranteed to be distinct because relation names are unique.

But how do stream components, such as the atmospheric model or visualization client, know to which channels to subscribe and whether to subscribe as a source or a sink (the latter required by the ECho API)? For quoblets, the information can be extracted from the query scripts that the quoblet receives. As a SQL query explicitly names the input relations and output relations, the quoblet can extract these names during query instantiation and connect as a sink to its input relations, and as a source to its output relations. Non-quoblet components, not having the advantage of receiving query scripts, must obtain relation names from the dQUOB server, or from a name server, for the event types they export and in that way can subscribe to a channel based on relation name.

In practice, we have not found the scheme of one event channel per relation to be limiting. Event channel creation is done with low overhead and the middleware does not suffer from the maximum open sockets limitation that socket-based solutions do. Further, the scheme scales well to accommodate queries added at runtime. Since a quoblet keeps track of the relations it has subscribed to, it can quickly identify and subscribe to new relations present in the query. For instance, if a newly received query needed to accept observational data in addition to the model generated data already being received at the quoblet, the quoblet could subscribe as a sink to the observational data event channel. Extensions to the framework will consider integration of non-quoblet based embedded stream components into relation-based computational data stream.

4.3 Configuring a Quoblet

A quoblet is created through an *rsh* command executed by the dQUOB server as shown in Figure 3. A quoblet at startup contains no queries and subscribes to one single event channel that allows it to receive configuration information from the dQUOB server.

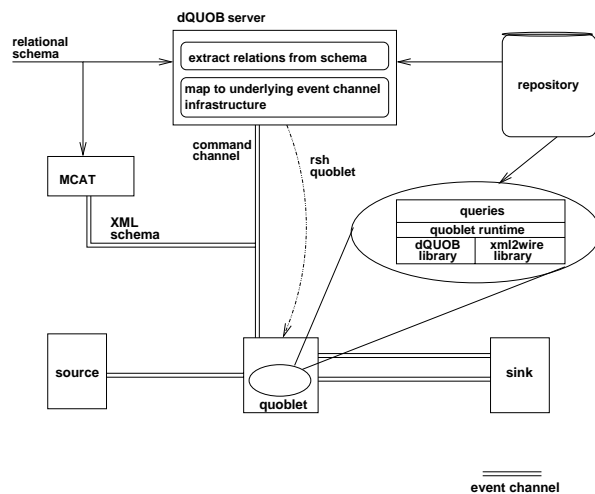


Figure 3: Framework.

A quoblet is configured, that is, populated with queries and subscribed to the proper event channels, by a command from the dQUOB server communicated via the event channel. To instantiate a query at a quoblet, the dQUOB server passes the quoblet a Universal Resource Locator (URL) that “points” to the location of the initialization information. The dQUOB server can then issue an HTTP GET request to retrieve the information. We intend to provide the information as XML documents, which will be parsed by the dQUOB server to extract the query scripts and relational schema needed to initialize the quoblet.

The primary advantage of retrieving the configuration information from a remote server is flexibility. Query scripts and schema can be centrally located and managed, and all servers become immediately aware of any updates (since they always retrieve the most recent copy).

Once the configuration information is available to the quoblet, it interprets the script by invoking a resident Tcl interpreter. Interpretation results in a series of calls to the dQUOB library and a list of input and output relations. The calls to the library cause the query to be built as a set of instantiated objects linked as a directed graph. The list of input and output relations is used to subscribe to event channels; the quoblet subscribes as a sink for the input relations and as a source for the output relations. The user object code is retrieved from the repository as well and dynamically linked into the quoblet. The visualization example without an intermediate quoblet could be built using a single event channel, *EC_Data*, as shown in Figure 4. The addition of a quoblet having the query from Example 1 in Section 3 causes a new event channel to be created, *EC_New_Data*, to which the visualization client, 'viz', now subscribes.

The computational data stream shown at the bottom of the figure consists of a source, sink, and quoblet connected with three event channels. The data flows through the event channels from source to sink through the quoblet, the latter of which performs intermediate computation. An additional feedback channel exists from sink to quoblet to influence decision making at the quoblet. This configuration, as simple as it is, has useful application to scientific visualization. For instance, a common computational data stream used in our work establishes as the source a global atmospheric transport model emitting chemical species data every logical timestep to a visualization client. The quoblet might filter the data not of immediate relevance to the user, while converting the non-filtered data from the spectral form of the model to the grid form needed by visualization tools; the sink visualizes the results while occasionally feeding back to the quoblet a request for a region of data of current interest.

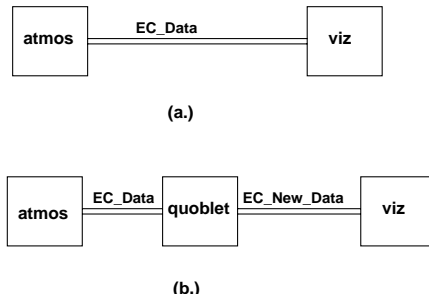


Figure 4: Event channel configuration before (a.) and after (b.) insertion of quoblet.

4.4 Mapping to Low-level Data Formats

The ECho event channel system requires the definition of program-level metadata that describes the structured data being transported. Exactly what this metadata should describe, however, is an important issue. The structure of the events that each channel will transport is fully described by the relational schema. What is needed is a mechanism to translate the relational schema into a program-level format that ECho can use. Our approach to this problem leverages the interoperability advantages presented by XML.

We have implemented a tool, `xml2wire` [16], that provides the benefits of open metadata definition while facilitating fast binary data transmission. `xml2wire` is a run-time library that converts metadata definitions expressed using XML Schema [15], a World Wide Consortium work-in-progress that provides a vocabulary for describing the structure and content of XML documents. XML Schema descriptions are themselves XML documents, and `xml2wire` uses a XML parser to convert them to an internal metadata format used by PBIO. These PBIO formats can then be used when subscribing to an event channel. `Xml2wire` is based on the principle of separation between the steps of metadata discovery and metadata binding. Native PBIO metadata is defined in and compiled into application programs and used directly to encode and decode data buffers. `Xml2wire` makes it possible to perform metadata definition outside the scope of the application program, promoting flexibility and reusability of metadata information.

What remains is to define some method of translating a relational schema into an XML Schema-compatible format. While we have not yet attempted to build a tool that performs this translation, we believe its implementation would be straightforward. There are also significant efforts being undertaken by other researchers to achieve similar goals. One such effort is MCAT (Meta Information Catalog) [10], part of the Data-Intensive Computing project at NPACI. MCAT is a metadata repository that supports storage, retrieval, and query operations. While it currently only supports these operations through a proprietary interface, the goals for the project state that eventually SQL and XML Schema interfaces will be supported for both input and output. We believe it reasonable to expect that MCAT (or systems like it) will be available to perform the relational schema-to-XML Schema mapping we require.

We have now established the mapping from relational schema to the low-level PBIO data formats required by the ECho library. As depicted in Figure 3, relational schema definitions are provided to a repository or general translation service (MCAT in this example). The quoblet retrieves the metadata definitions in their XML Schema form and uses the services of the `xml2wire` run-time library to convert them into the PBIO structure definitions required to establish typed ECho event channels.

5 Conclusion

We have shown in this paper that the user can benefit in numerous ways from thinking about computational data streams using a relational view taken from relational database technology. Conceptualizing the events flowing in a network application as belonging to relations enables operations such as querying, performed over the event flows to extract information of interest. Additionally, data description at the meta-level facilitates inclusion of diverse data sources into the network application by being independent of any low-level description of the data.

This paper focuses on a framework for mapping from the high-level conceptualization of relational schemas and SQL queries to low-level middleware domain of event channels and binary I/O data formats. The mapping to event channels capitalizes on the unique name space of relation names. The mapping to the binary I/O package currently

relies on a translation from a relational schema to an XML representation. Though perhaps a direct relational schema to binary I/O mapping would be desirable, we intend to eventually incorporate support for XML to relational schema translation into our system to facilitate interoperability with the computational grid, in which case the XML specification of the network application could be directly used by xml2wire.

References

- [1] Randall Bramley, Kenneth Chiu, Shridhar Diwan, Dennis Gannon, Madhusudhan Govindaraju, Nirmal Mukhi, Benjamin Temko, and Madhuri Yechuri. A component based services architecture for building distributed applications. In *IEEE International Symposium on High Performance Distributed Computing (HPDC)*, August 2000.
- [2] Greg Eisenhauer, Fabian Bustamante, and Karsten Schwan. Event services for high performance computing. In *IEEE International High Performance Distributed Computing (HPDC)*, 2000.
- [3] Greg Eisenhauer and Lynn K. Daley. Fast heterogeneous binary data interchange. In *Heterogeneous Computing Workshop (HCW)*, 2000.
- [4] Renato Ferreira, Tahsin Kurc, Michael Beynon, Chialin Chang, and Joel Saltz. Object-relational queries into multidimensional databases with the Active Data Repository. *Journal of Supercomputer Applications and High Performance Computing (IJSA)*, 1999.
- [5] Ian Foster and eds. Carl Kesselman. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann, 1999.
- [6] Carsten Isert and Karsten Schwan. ACDS: Adapting computational data streams for high performance. In *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2000.
- [7] H. V. Jagadish, Laks V. S. Lakshmanan, and Divesh Srivastava. Revisiting the hierarchical data model. *IEICE Transactions on Information Systems*, January 1999.
- [8] Fabio Kon, Roy Campbell, Marshall Mickunas, Klara Nahrstedt, and Francisco Ballesteros. 2k: A distributed operating system for dynamic heterogeneous environments. In *IEEE International High Performance Distributed Computing (HPDC)*, 2000.
- [9] Ling Liu, Calton Pu, Roger Barga, and Tong Zhou. Differential evaluation of continual queries. Technical Report TR95-17, Department of Computer Science, University of Alberta, 1996.
- [10] Data-Intensive Computing Thrust National Partnership for Advanced Computational Infrastructure (NPACI). Mcat - a meta-information catalog. <http://www.npaci.edu/dice/SRB/mcat.html>.
- [11] Beth Plale and Karsten Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *IEEE International High Performance Distributed Computing (HPDC)*, August 2000.
- [12] Calton Pu, Jonathan Walpole, Karsten Schwan, Ling Liu, and Gregory Abowd. Infosphere. <http://www.cc.gatech.edu/projects/infosphere/>, 2000.
- [13] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Transitioning temporal support in TSQL2 to SQL3. In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases: Research and Practice*. Springer-Verlag, 1998.
- [14] Richard Snodgrass. A relational approach to monitoring complex systems. *IEEE Transactions on Computers*, 6(2):156–196, May 1988.
- [15] World Wide Web Consortium (W3C). Xml schema. <http://www.w3.org/XML/Schema/>.
- [16] Patrick Widener, Karsten Schwan, and Greg Eisenhauer. Open metadata formats for fast communication. Technical Report GIT-CC-00-21, College of Computing, Georgia Institute of Technology, 2000.

- [17] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems*. Morgan Kaufmann, 1996.
- [18] C. Yu and W. Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann, 1998.