Uniform Data Access Using GXD

Peter Vanderbilt*

September 22, 1999

Abstract

This paper gives an overview of GXD, a framework facilitating publication and use of data from diverse data sources. GXD defines an object-oriented data model designed to represent a wide range of things including data, its metadata, resources and query results. GXD also defines a data transport language, a dialect of XML, for representing instances of the data model. This language allows for a wide range of data source implementations by supporting both the direct incorporation of data and the specification of data by various rules. The GXD software library, prototyped in Java, includes client and server runtimes. The server runtime facilitates the generation of entities containing data encoded in the GXD transport language. The GXD client runtime interprets these entities (potentially from many data sources) to create an illusion of a globally interconnected data space, one that is independent of data source location and implementation.

1 Introduction

This paper gives an overview of GXD, Grid[1] eXtensible Data. GXD is a framework consisting of three main components: a ***data model*** definition, a ***data format*** definition and a library containing client and server APIs. Data sources logically map their data into the data model and use the corresponding server APIs to generate entities in the GXD format. Applications use the client APIs to access data at the logical, data model level. The *client runtime*, the implementation of the client APIs, obtains the GXD-encoded entities and translates them into the data model. All this will be described in more detail below.

The purpose of GXD is to facilitate the construction of applications that can handle data from diverse data sources, especially those data sources that are distributed and of heterogeneous implementation. Such applications should enhance the ability of scientists to form and explore new compositions of data sets.

*MRJ Technology Solutions, NASA Ames Research Center; email: pv@nas.nasa.gov.

[1] "Grid" refers to the Information Power Grid (IPG [3]).

The GXD data model, discussed in section 2, is designed to represent a wide range of things including data and its metadata, associations between data items, infrastructure resources (such as users, machines and networks) and GXD metamodel items (such as schemas and interface definitions). GXD is also designed to represent the structured results of complex queries.

One of the features of GXD is that it allows an application to view scientific data logically as a globally interconnected set of data nodes, called the ***data space***. The data space provides transparency over location and data source implementation. While the data space can vary over time, for the purposes of this paper we consider the data space to be immutable.

The data space is physically represented by a (large) collection of GXD *entities* as published by the various data sources. These entities may be files or they may be the outputs of programs (such as servlets or CGI programs) that access underlying data repositories, such as databases. These entities are encoded using GXD's data format (as described in section 3). At this level, GXD allows data to either be incorporated directly or be described by various rules. The GXD runtime provides the logical view by handling the task of locating and accessing the physical entities and by interpreting their content.

The GXD APIs include those used primarily by the client and those used by the server. The client APIs, discussed in subsection 2.3, provide access to the data space. The server APIs, discussed in subsection 3.3, are used to produce GXD entities. Currently, the only APIs are in Java (version 1.2).

GXD is designed to support a web-like, "organic" approach to data management. Scientists can publish their data without heavyweight, prior coordination (with other sources) and incrementally add new data features. Over time, sites can federate together by using common formats and publishing cross references. Additionally, other people can create virtual data sources that add value to one or more other data sources.

The GXD software is positioned below application code and above data transport code such as that for HTTP and IPG [3] data management services such as GASS [1] and SRB [4]. (Access to anything other than HTTP is not currently implemented).

GXD is an ongoing research project and its implementation is not complete. The current implementation is denoted "0.6" and this paper attempts to point out those features not implemented in 0.6.

The rest of the paper is organized as follows: Section 2 describes the GXD data model and corresponding APIs. Section 3 describes the data transport language. Section 4 discusses additional topics such as GXD polymorphism, interface evolution, why we didn't use XML directly and dataset dispatching.

2 The GXD Data Model

The GXD data model describes the universe of data spaces that can be represented by GXD. There are two key concepts: "nodes" and "interfaces". A *node* is a unit of (possibly structured) information. An *interface* can be thought of as a mathematical predicate over nodes that additionally imparts semantics. These are discussed in subsections 2.1 and 2.2 (respectively). Subsection 2.3 illustrates the client APIs used to access data model instances.

2.1 Nodes

A GXD node is a logical unit of information. Nodes are either atomic or structured. The two kinds of atomic nodes are values and externals. A *value node* contains a string (which possibly contains numeric or structured information). An *external node* is one that contains a pointer to data in the underlying programming language: externals result from reading datasets as discussed in subsection 4.4.

The structured nodes are lists, dictionaries, methods and objects. These contain or produce other nodes (or, actually, references to other nodes). Note that the definition given here is recursive and, so, allows arbitrarily structured nodes.

A *list node* contains an ordered collection of other nodes. A list has a fixed length and elements are indexed by position. A *dictionary node* contains a collection of nodes indexed by key. The keys are strings and the set of keys can be obtained from a dictionary. A *method node* is logically an algorithm that, given zero or more arguments, returns a node. (Methods are not implemented in 0.6).

An *object node* contains an unordered collection of nodes indexed by *property names*. A property name is an identifier possibly qualified by an interface identifier. The set of property names can be obtained from an object. Each node named by a property name is called a *property*. Some properties act as *attributes* of the object, providing information about the object, and some properties act as *associations*, providing one or more links to other nodes.

As an example of some GXD nodes, consider figure 1. Circles are used to

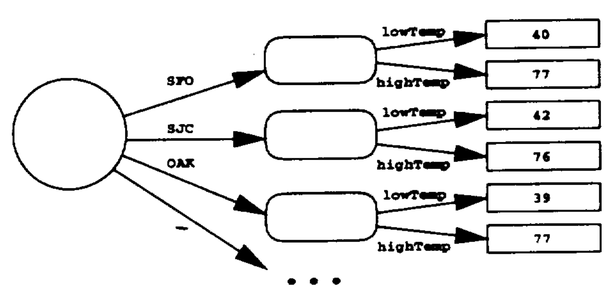

Figure 1: Example Weather Data Nodes

indicate collections (dictionaries if its arrows are labeled, lists if not), rounded boxes for objects and rectangles for values. An arrow indicates that a node references another and its label gives the corresponding key or property name. The content of a value node is displayed within its rectangle. The ellipsis indicates that more nodes are referenced by the dictionary node.

Thus figure 1 shows a dictionary node referring to object subnodes, the first three of which are keyed by the strings "SFO", "SJC" and "OAK". Each object subnode has two value subnodes, with property names "lowTemp" and "highTemp". The content of the "lowTemp" value node of the "SFO" object node is "40".

Every GXD node is addressed by one or more URLs. For example, the URLs

```
http://baweather.com/wd/sfo.gxd
http://baweather.com/wd/sfo.gxd#SFO
http://baweather.com/wd/sfo.gxd#SFO&lowTemp
```

could address, respectively, the dictionary node, the topmost object node and the topmost value node of figure 1. It is planned that every node should also have a unique *name*, but this is not implemented in 0.6.

2.2 Interfaces

Taken alone, any given node is pretty much meaningless. Generally, in order for an application to use data, it needs to know the potential set of data values and how to interpret each value.

In GXD, this interpretation is specified by an *interface* which imposes constraints on GXD nodes and imparts semantics to them. A node is said to *implement* an interface if that node meets the constraints and semantics imposed by the interface. A node can implement more than one interface, in which case it must meet the union of the constraints and semantics of all its interfaces. Each interface is named by a globally unique *interface identifier* which is a string similar to a Java class name.

An interface can *inherit* from (or *extend*) another, in which case it imposes additional constraints and stricter semantics than its "superinterface". Actually GXD allows *multiple inheritance*. GXD inheritance supports *polymorphism*, whereby a node implementing an interface also implements all of its superinterfaces.

Interfaces apply to all kinds of GXD nodes (not just object nodes) and all such interfaces can be extended. Since an interface can constrain the interfaces of its components, it can define a substructure consisting of a number of nodes. For example, review figure 1 and consider the following interface descriptions:

WeatherDataCollection: A node of interface WeatherDataCollection contains information about the previous day's temperature range for various airports around the country. In particular, a WeatherDataCollection node is a dictionary node whose keys are airport codes and whose contents are WeatherData nodes.

WeatherData: A WeatherData node is an object node with two (required) properties, "lowTemp" and "highTemp", each a Temperature node giving the low and high temperatures (respectively) of the previous day's weather.

Temperature: A temperature node is a value node containing a decimal number giving a temperature in Fahrenheit.

It is possible for the data publisher to assert that the dictionary of figure 1 implements the WeatherDataCollection interface. This, in turn, implies that all the object nodes implement WeatherData and all the value nodes implement Temperature.

It is planned that an interface will be formally described by a GXD (meta) node called an *interface definition* but this has not been implemented in 0.6.

2.3 Client APIs

The GXD client APIs follow the data model, with a Java interface for each GXD node type. For example, figure 2 shows the code used to print a simple table of low temperatures for airports.

```
String url = "http://faa.gov/weather_data.gxd";
GxdRuntime gxdRuntime = GxdImpl.getDefaultRuntime();
GxdDict collection = (GxdDict) gxdRuntime.getGxdNode(url);
Iterator iter = collection.getKeys().iterator();

system.out.println("Low temperatures for US airports:");
while (iter.hasNext()) {
    String airport = (String) iter.next();
    GxdObject weatherData = (GxdObject) collection.get(airport);
    GxdValue temp = (GxdValue) weatherData.getProperty("lowTemp");
    system.out.println("   " + airport + " " + temp.getValue());
}
```

Figure 2: Example code for accessing data

The first line sets the URL, which must address a WeatherDataCollection node such as the dictionary node of figure 1. The next line obtains a `GxdRuntime` object which encapsulates the state of the GXD runtime. The third line gets a reference to the dictionary node, given its URL. `getGxdNode()`, like many GXD operations, returns an object of (Java) interface `GxdAny`, the root of the GXD node inheritance hierarchy, and so the result must be "narrowed" to `GxdDict`, the interface for dictionary nodes. From the dictionary node, an iterator over the keys is obtained and, for each key, the `get()` operation is used to yield the corresponding object node. For each such object node, `getProperty("lowTemp")` is used to yield the corresponding value node, on which `getValue()` yields the temperature (as a string).

3 The GXD Data Transport Language

Recall that, at any point in time, the global set of GXD nodes is called the data space. The data space is virtual in that it is a fiction created by the GXD runtime from data published by the various data source providers. The GXD data transport language is the language used to publish this data. An instance of the transport language, called an *entity*, is used to download some region of the data space to a client. An entity may be a file or it may be generated on demand.

The data language uses XML (the eXtensible Markup Language [6]) to provide the basic structure for GXD entities. XML consists of tagged *elements* which can have *attributes* and can contain other elements, text or nothing.

The data language can encode GXD nodes directly or indirectly. Direct encoding is used when a copy of the node's state is transferred. Indirect encoding is used when some sort of rule is transferred to the client and the application of that rule yields the node's state. Each of these encoding methods is discussed in subsequent subsections.

3.1 Direct Encoding

Directly encoded nodes are represented using XML elements with tags corresponding to the node type; these tags are VALUE, OBJECT, LIST, DICT. For example, figure 3 shows an entity encoding the "SFO" object node of figure 1. For the convenience of the reader, the first line of each entity example contains the entity's URL (as an XML comment).

```
<!-- URL: http://baweather.com/wd/sfo.gxd -->
<?xml version="1.0"?>
<GXDFile>
    <OBJECT intf="gov.faa.WeatherData">
        <VALUE id="lowTemp" val="40"/>
        <VALUE id="highTemp" val="77"/>
    </OBJECT>
</GXDFile>
```

Figure 3: Example of directly encoding 3 nodes

The second line (in this example) is the standard XML declaration element. The GXDFile element is the top-level GXD element and exists to allow entity metadata to be specified; the logical content starts with its only subelement. For the remainder of this paper, the XML declarations and the GXDFile elements will not be shown.

The OBJECT element encodes the object node; it starts on the fourth line and ends on the seventh. It contains an intf attribute as will be discussed shortly. Within the element are two VALUE subelements encoding the object's value properties. Each of these elements has an `id` attribute, which contains its property name, and a `val` attribute, which contains the node's value.

In general, the `id` attribute of an element indicates its relation to its parent. Each subelement of an `OBJECT` element has an `id` attribute giving its property name. Each subelement of an `DICT` has an `id` specifying its key. Subelements of an `LIST` element have no `id` attribute as they are implicitly indexed by numbers (0 to size-1).

An element can also have an `intf` attribute specifying the set of interface identifiers for those interfaces implemented by the corresponding node. The above example has such an attribute on the `OBJECT` element to indicate that the corresponding node implements the WeatherData interface (which is assumed to have been defined in the `gov.faa` namespace). In the current version of GXD, interface specification is optional.

3.2 Indirect Encoding

Indirect encoding in GXD involves encoding some sort of rule in place of the actual data: the GXD runtime applies the rule to yield the specified nodes. One form of indirect encoding is the `LINK` element which uses a `ref` attribute containing a URL to indicate a redirection. For example, figure 4 shows a possible encoding of the dictionary node of figure 1.

```
<!-- URL: http://faa.gov/weather_data.gxd -->
<DICT>
    <LINK id="SFO" ref="http://baweather.com/wd/sfo.gxd"/>
    <LINK id="SJC" ref="http://baweather.com/wd/sjc.gxd"/>
    <LINK id="OAK" ref="http://oak.faa.gov/wd.gxd"/>
    ...
    <LINK id="JFK" ref="http://ny.faa.gov/wd.cgi#Kennedy"/>
    <LINK id="LGA" ref="http://ny.faa.gov/wd.cgi#LaGuardia"/>
    <LINK id="EWR" ref="http://ny.faa.gov/wd.cgi#Newark"/>
    ...
</DICT>
```

Figure 4: Example of LINK elements

Assume that the URL in the first `LINK` element refers to the entity of figure 3. This `LINK` element specifies that the dictionary's "SFO" subnode is obtained by (recursively) decoding the entity of figure 3. The GXD runtime takes responsibility for following these `LINK` elements; it uses a "lazy evaluation" implementation so an entity is not read unless it is referenced. (The GXD API also includes methods that cause the runtime to read entities at that time.)

Note that the `LINK` elements of figure 4 reference a distributed collection of weather data nodes. Since the GXD handles following links, code like that of figure 2 can simply loop through these nodes, without concern for data location.

Also note that GXD can leverage the flexibility of URLs and standard web servers, allowing for a wide range of data source implementations. For example, the URLs of the first three LINK elements might reference files. Or they might reference *wrappers*, which are CGI programs or servlets that yield GXD entities by accessing some underlying database or service. These are all server-side options and may, over time, be changed without affecting the client.

The last three LINKs demonstrate the ability for a link to reference an element within a GXD entity. Assume that `http://ny.faa.gov/wd.cgi` is the URL for the entity of figure 5. Then `http://ny.faa.gov/wd.cgi#Kennedy` refers to the

```
<!-- URL: http://ny.faa.gov/wd.cgi -->
<DICT>
    <OBJECT id="Kennedy">
        <VALUE id="lowTemp" val="20"/>
        <VALUE id="highTemp" val="35"/>
    </OBJECT>
    <OBJECT id="Newark">
        <VALUE id="lowTemp" val="23"/>
        <VALUE id="highTemp" val="33"/>
    </OBJECT>
    <OBJECT id="LaGuardia">
        <VALUE id="lowTemp" val="27"/>
        <VALUE id="highTemp" val="35"/>
    </OBJECT>
</DICT>
```

Figure 5: Example data referenced by figure 4

OBJECT element with id attribute of "Kennedy". Thus the dictionary's "JFK" subnode is obtained by decoding the entity of figure 5 and returning a reference to the "Kennedy" subnode. The GXD runtime takes care to read an entity only once, so there may be a pause while obtaining the "JFK" node but none while subsequently obtaining the "LGA" and "EWR" nodes.

In the future, the GXD data language may have other mechanisms for indirectly specifying nodes. One mechanism under consideration is a SCRIPT element which would allow the inclusion of (or reference to) a script together with some parameters; the script would be executed (with the parameters) and the resulting nodes returned in its place. For example, a script element might be used to generate a list of links whose references are created by interpolating the index into a URL template.

Another mechanism under consideration is to allow a *template* to be associated with a set of elements. A template is a GXD node (of a GXD-defined interface) that contains information logically included in all elements of the set. For example, templates might contain constant values, links (which are relative to the target element), methods and scripts.

3.3 Server APIs

The server APIs are used to generate GXD entities. The APIs include a Java class corresponding to each of the element types discussed in subsection 3.1 and to LINKs as discussed in subsection 3.2. A data source implementation uses these classes to create a tree corresponding to the region of the data space to be represented and calls `writeXml(stream)` on the root: the GXD server runtime writes the entity content on the stream `stream`.

The server classes are designed to allow for flexibility of implementation. For instance, the constructor for the dictionary class takes any instance of the Java interface `java.util.Map`, allowing the implementor to pick the most suitable class implementing `Map`.

4 Advanced Topics

4.1 Polymorphism

In subsection 2.2, it was mentioned that GXD supports polymorphism, whereby a node implementing an interface also implements all of its superinterfaces. For example, assume that there is an interface, AirportConditions, that inherits from WeatherData (of subsection 2.2) and adds properties for wind and precipitation. The data publisher might upgrade the entity of figure 3 to have the content of figure 6. The code of figure 2 (which indirectly references this

```
<!-- URL: http://baweather.com/wd/sfo.gxd -->
<OBJECT intf="AirportConditions,WeatherData">
    <VALUE id="windDirection" val="NNE"/>
    <VALUE id="windSpeed" val="6"/>
    <VALUE id="lowTemp" val="40"/>
    <VALUE id="highTemp" val="77"/>
    <VALUE id="precip" val="0"/>
</OBJECT>
```

Figure 6: Example of GXD polymorphism

entity) would work as before, by simply ignoring the extra properties. At the same time, other code (developed to the AirportConditions interface) could take advantage of the additional fields to provide advanced functionality.

4.2 Interface Evolution

It is expected that the definition of interfaces will be an evolutionary endeavor. Initially, independent groups will define interfaces. As understanding matures, these groups will need to modify interfaces (or create new ones). Also different groups will collaborate and wish to have common interfaces, thus creating interfaces with a union of the capabilities of the independently-created ones.

The scheme planned for GXD (after 0.6) is to allow an interface to have a collection of *revisions*, each denoted by a *revision number*, and to allow nodes to implement a range of revisions. Thus a data source publisher can publish data in a range of revisions allowing it to support both older and newer applications. Similarly, an application can be implemented to handle a range of revisions, so that it can handle both older and newer data sources.

4.3 Why not use XML directly?

GXD puts a layer over XML. To see this, compare figure 3 to figure 7 which is one way to encode the same information directly in XML. In figure 7, "WeatherData" is the tag of an element enclosing two subelements, respectively tagged "lowTemp" and "highTemp" and containing the values "40" and "77".

```
<!-- URL: http://baweather.com/wd/sfo.xml -->
<?xml version="1.0"?>
<WeatherData>
    <lowTemp>40</lowTemp>
    <highTemp>77</highTemp>
</WeatherData>
```

Figure 7: Direct encoding of figure 3 in XML

The problem with the GXD encoding, as illustrated by figure 3, is that GXD's constrained use of XML inhibits leveraging some of the technology associated with XML. But there are several advantages to the GXD encoding, as discussed in the remainder of this subsection, which (we believe) outweigh the disadvantages.

One advantage is that putting a layer over XML allows the kinds of indirections discussed in subsection 3.2. When encoding directly in XML, any kind of indirection must be handled by the client. By having the GXD runtime handle it instead, the client is greatly simplified. We believe that the combination of templates, links and scripts will allow encoding techniques far beyond what one would get using direct XML encoding. This allows for a wide range of data source implementations and allows them to evolve without affecting existing clients.

A second advantage has to do with inheritance and polymorphism, as discussed in subsection 4.1. In particular, GXD allows a node of some subinterface to be used wherever a node of a certain interface is expected. To do this in "valid" XML, where instances must conform to a type definition (known as a "DTD"), the direct XML encoding corresponding to figure 6 would include an element for "windDirection" which would fail the validity tests for the DTD corresponding to WeatherData.

Even with "well-formed", DTD-less XML, there is a related problem: what should be the tag for the direct XML encoding of figure 6, "WeatherData" or "AirportConditions"? If "AirportConditions" is used, code understanding only the base type will not recognize the element. But if "WeatherData" is used, code understanding the derived type gets no clue that the element has additional (derived type) data. GXD gets around this by moving interface specification into an attribute that contains a list of interface ids.

There are similar issues with direct XML encodings of nodes implementing multiple interfaces and implementing multiple revisions of interfaces.

Finally, it is hoped that translation between a direct XML encoding and the corresponding GXD encoding will be easy. In particular, a SCRIPT element should allow one to represent a region of the data space directly in XML by specifying a script that reads XML and yields GXD nodes. Similarly, it should be straightforward to write GXD-to-XML translators for various interfaces.

4.4 Dataset Handling

While GXD is good for representing metadata and some data, its encoding is too inefficient for large quantities of data. Instead GXD facilitates access to scientific data in its native format (such as HDF [2] or PLOT3D [5]). In 0.6, the GXD transport language includes an indirection element, tagged DATASET, which, when encountered by the runtime, causes an application-specified *dataset handler* to be invoked. This dataset handler is given the contents of the DATASET element and returns a GXD node. Typically, this result node is the root of some substructure containing external nodes. The external nodes point to in-memory structures resulting from reading the specified data.

However, experience has shown that this upcall mechanism is too limiting and not helpful enough. Instead, post-0.6 there will be set of interfaces for dataset descriptors. A *dataset descriptor* is an object node that contains metadata for accessing a scientific dataset in its native format. The object's interface specifies the format of the data (such as HDF or PLOT3D) and its content typically contains a URL plus whatever metadata is needed to interpret the data referenced by the URL. An application would read the data itself, using tools provided by GXD such as those for handling URLs or for dispatching based on a dataset descriptor's interface.

5 Conclusion

In summary, GXD is middleware that allows data from diverse data sources to be published in a common data format, using common protocols. This potentially allows applications to synthesize data from several distributed, heterogeneous data sources. The GXD data model allows for standardized interpretation of data and the GXD client APIs allow applications to access the data without regard for data source location or implementation. The GXD transport language provides a flexible way to download regions of the data space to a client.

The current implementation of GXD (0.6) has server-side APIs, for creating instances of the GXD transport language, and client-side APIs, for handling these entities and providing the virtual data model. Implemented are value, external, list, dictionary and object nodes and `LINK` and `DATASET` elements. This software is of prototype quality.

Plans for GXD include implementing the mechanisms mentioned previously such as methods, templates, interface revisioning, formal interface definitions, unique node names and script elements. We are also working with users to try to determine a useful set of common base interfaces.

For more information about GXD, contact the author.

References

[1] The Globus Project (www.globus.org). *Global Access to Secondary Storage (GASS).* http://www.globus.org/gass/.

[2] The National Center for Supercomputing Applications. *Hierarchical Data Format (HDF).* http://hdf.ncsa.uiuc.edu/.

[3] Numerical Aerospace Simulation Systems (NAS) Division of NASA. *Information Power Grid.* http://www.nas.nasa.gov/IPG/.

[4] San Diego Supercomputing Center (SDSC). *Storage Resource Broker (SRB).* http://www.npaci.edu/DICE/SRB/.

[5] P. Walatka, P. Buning, L. Pierce, and P. Elson. *PLOT3D User's Manual.* National Aeronautics and Space Administration, July 1992. NASA Technical Memorandum 101067.

[6] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0 (REC-xml-19980210).* http://www.w3.org/TR/REC-xml.