

Rationale for the Data Access and Integration Architecture

Malcolm Atkinson¹, Simon Laws² and Greg Riccardi³

1

1	Introduction.....	2
2	DAI: Motivation and Scope.....	2
3	Application Programming Interfaces and Protocols.....	3
4	Conceptual Architecture.....	4
4.1	External Data Resources.....	5
4.2	Data Resources.....	6
4.3	Data Activity Sessions.....	8
4.4	Datasets.....	9
4.4.1	Dataset Contents.....	10
4.4.2	Specific dataset contents.....	11
4.4.3	Data Collection and Delivery Recipes.....	12
5	Greg’s Examples beyond here – to be absorbed.....	15
6	1 Processing request datasets.....	15
7	2 Processing SQL select queries.....	16
8	3 SQL update and DDL statements.....	17
9	4 Bulk load.....	18
10	4 Processing errors and response documents.....	18
11	5 Sample scenarios for performImmediate with delivery to/from 3 rd party.....	19
12	Appendix: Example of table schema and values in XML.....	20
13	21
	Processing request datasets.....	15
	2 Processing SQL select queries.....	16
	3 SQL update and DDL statements.....	17
	4 Bulk load.....	18
	4 Processing errors and response documents.....	18
	5 Sample scenarios for performImmediate with delivery to/from 3 rd party.....	19
	5.1 Sql query request dataset.....	20

¹ National e-Science Centre, 15 South College Street, Edinburgh EH8 9AA, Scotland, UK (mpa@nesc.ac.uk)

² IBM Hursley Services and Technology, Mail Point 137, Hursley Park, Winchester, Hampshire SO21 2JN, England (simon_laws@uk.ibm.com)

³ Department of Computer Science, Florida State University, Florida, USA (riccardi@cs.fsu.edu)

1 Introduction

The Global Grid Forum Data Access and Integration working group is developing a proposed specification for middleware that supports the use of multiple distributed data resources [1]. That document, as befits a specification, is developing definitions, but does not present the rationale for those decisions. There is also a primer under development [2] that presents examples and tutorials as to how to use the DAI services. It focuses on introduction to the facilities and suggests effective programming patterns.

The purpose of this document is to provide the rationale for decisions that lead to the architecture and high-level design of the DAI services. It summarizes the motivation for DAI services and discusses their current and eventual scope. It then presents the components of the conceptual model showing why they are included, their function and their relationship. Attention is drawn to the difference between requirements for application interfaces (APIs) and the requirements for DAI protocols. The paper concludes with examples that illustrate how the proposed model can serve in a number of typical scenarios.

2 DAI: Motivation and Scope

There are an ever increasing number of useful data resources that are independently managed and geographically distributed. Scientific discovery, engineering design, medical decisions and many other creative processes benefit from accessing and sometimes updating these diverse, dispersed and evolving data resources. Typically the creative step in a process is enabled by formulation of a hypothesis that is then tested by combining data from multiple sources using sophisticated models. The goal of Data Access and Integration (DAI) is to facilitate that step without constraining it. For further discussion of this motivation see [3], [4] and [5]. The data accessed may be primary data, metadata describing primary data, or administrative and system data used in the implementation and operation of distributed systems that enable the computations and data movement.

Guiding principles of DAI are:

- 1 To respect and comply with the policies and authorization mechanisms established by the providers of data resources;
- 2 To minimize the cost of data movement.

The first of these principles dominates where they are in conflict as failure to comply with conditions of use imposed by information providers would mean that access was denied.

The second principle is manifest in mechanisms for third-party transfers, for avoiding round-trips that accrue latency and for moving computation close to data. The last of these is currently based on using database-supported query languages, which in some cases will accommodate embedded procedures.

The requirement to move code to data is likely to increase as data volumes grow [6]. Data volumes grow much faster than code volumes as the speed, cost and sensitivity of data collectors and data generators grow according to Moore's law, but code is limited by human capacities. For example, biological sequence and protein databases are doubling approximately every nine months and astronomic data is doubling every 12 months. Some databases are already large [7].

The current work of the DAIS Working Group focuses on services that facilitate the interconnection of existing databases held in relational or XML form. This initial focus is motivated by the large investment already made in establishing, provisioning, operating, understanding and exploiting such databases. This existing knowledge and supporting software is expected to be the basis of many scientific and commercial applications.

It is anticipated that the boundaries between structured data and files will be eroded as more investment is made in metadata describing collections of files [8], [9] and [10]. Files are commonly used to hold raw and processed observational data and the inputs and outputs of simulations. They are therefore important in the classes of computation that DAI seeks to support. Technologies for their management (e.g. SRB [xxx]) and replication (e.g. RMS [yyy]), for their transfer (e.g. FTP and GridFTP [zzz]) and for extracting or deriving data from them (e.g. Data Cutter [aaa]), are well developed. Data Cutter has recently developed the capacity to accept appropriate Java jar files, dynamically load them, and run them against local files, thereby enabling a broad class of computations to be brought to the data. These trends mean that the architecture of data access and integration services must be prepared to accommodate file access. This goal is met by self-descriptive messages and by a dynamic binding model for activities (see below section ??).

The set of data models already in use includes object databases, semi-structured data [bbb], information retrieval systems and systems for extracting data from text documents [ccc]. There is certain to be progressive development of data models and query languages, and new data management technologies will be introduced. The architecture for DAIS must be open so that such advances can be accommodated and so that each stage of each model's or language's development can coexist within systems and applications. This goal is met by self-descriptive messages and by a dynamic binding model for activities (see below section ??).

The current standard focuses on combining data from a relatively static set of external data resources (see below for a definition). However, the use of dynamically created and non-persistent yet complex data resources occurs in several known scenarios. Such dynamic behavior will eventually need to be managed by DAIS though it will depend on other services. Similarly, DAIS will need to support co-optimization of data activities and computation activities in workflows and will raise issues where long-term optimization of data structure and placement will interact with those immediate work load optimizations. Other important interfaces for DAIS are with authorization services and accounting services, as data resources often have sophisticated existing versions of these functions.

3 Application Programming Interfaces and Protocols

A basic goal of DAIS is to facilitate the implementation of a wide range of applications. Such facilitation should mean that simple programming tasks are still simple and that familiar application patterns can be reused. On the other hand, there must be mechanisms that enable skilled application programmers to design, implement and precisely control data intensive workflows. The following architectural strategy:

- 1 The protocol used for activating DAI services, particularly data resources, is generic, extensible and extremely powerful – it is based on transferring data sets (see section ??), which are bundles of data requests (see section ??) as inputs and outputs to synchronous and asynchronous portTypes (performImmediate and performAsynchronous respectively).

- 2 There are high-level APIs proposed as libraries in popular hosting languages (Java, C, C# and Perl) that replicate well-established APIs, such as Java JDBC and C# ANO.NET DataSet and map them to the protocol using the low-level API⁴.
- 3 There is a low-level API used by servers and by application components, such as clients, analysts and consumers.

The low-level API will provide sufficient direct manipulation of datasets so that, in combination with direct use via SOAP of other portType operations, all forms of precise and sophisticated combinations of DAI services with other services can be achieved. The high-level APIs may be standard APIs with minor variations to match the DAI requirements. They must directly and conveniently support the frequently used programming patterns, such as “send query and receive results”. In both APIs there must be functions for constructing, populating and dispatching a dataset; we’ll call this the *construction API*. In both APIs there must be functions for obtaining, inspecting and reading data from a dataset; we’ll call this the *access API*. In the high-level APIs, much of the detail is hidden.

Figure 1 illustrates the relationships between the APIs and the protocol. The explicit introduction of the APIs to support application programming also has the advantage of insulating the protocol (portType and parameter) designs from the majority of application programmers. This should permit optimization of these protocols in response to issues exposed by real use while maintaining stability for most users. As always in data-intensive systems great care has to be taken in the design and implementation of these APIs to avoid multi-stage copying and multiple coexistent representations of the data, as these are performance killers.

Suggested features and behaviors of these APIs are presented in section ??.

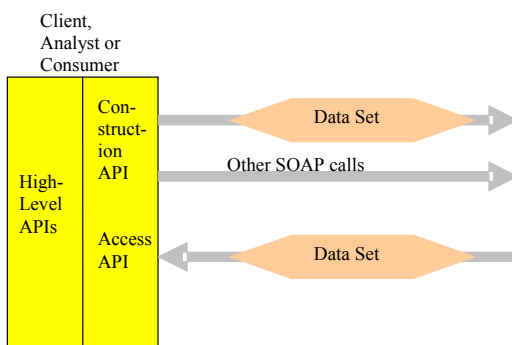


Figure 1: The structure of APIs shielding the dataset-based protocols

4 Conceptual Architecture

The concepts required for the architecture are described and their relationships, functions and *raison d’être* are presented. The concepts to be introduced are:

- 1 External Data Resources
- 2 Data Resources
- 3 Data Activity Sessions

⁴ The DAIS-WG should spawn a working group per hosting environment to develop the definition of these APIs. The output from these working groups will be a precise set of use-cases for the DAIS specification.

- 4 Datasets
- 5 Activities

These concepts are introduced to explain the architecture and are chosen to separate concerns. They are suggestive of an implementation, but different implementations will be developed that exhibit equivalent behavior. For example, in some implementations, one service may perform a number of these conceptual services.

4.1 External Data Resources

The term “external” is used to identify data resources that exist independently of data access and integration services. These will often be well-established systems, operated autonomously, known to the application developers or application users who will be aware of facilities, content policies and authorization mechanisms pertaining to these resources. At least for a transition period, these developers and users expect to explicitly identify such resources and to use their existing knowledge about them. An important property of many external data resources is that they are in use by many other mechanisms besides the DAIS mechanisms. These may result in independent updates, schema changes and service changes that are often of interest to the applications and users that access them via DAIS mechanisms.

The organizations that operate such *external data resources*, or provide the software and support these *external data resources* will not adapt to data access and integration requirements until these requirements are pervasive, stable and of proven value. Therefore, these facilities will frequently be used indirectly, exploiting their existing services, interfaces and functions.

An installed database management system instance, such as Xindice [ddd], Oracle 9i or DB2, is considered to be an *external data resource*. Such a system may manage zero or more other *external data resources*, each of which is a distinct database over which requests may be phrased. The repertoire of operations available may vary, e.g. between database management system instances and databases, and between databases that correspond to different data models. Figure 2 illustrates external data resources.

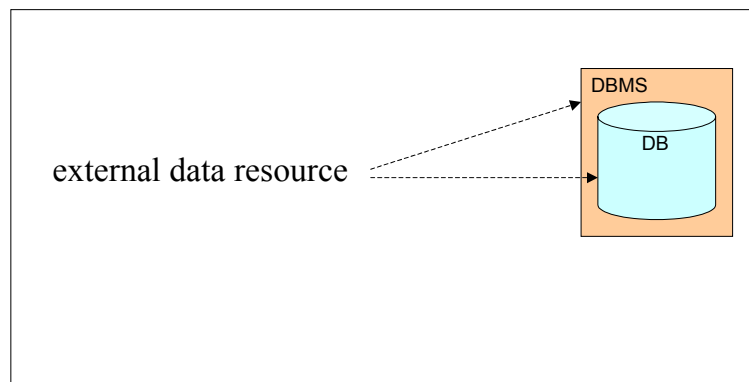


Figure 2 – External Data Resources

External data resources are not themselves services in the DAIS model. Instead they are represented by *data resources*. *Data resources* that represent a known *external data resource* will need to identify that *external data resource* and may present status information about that data resource, such as its name, data model, access policies, and DBMS version.

In summary: the *external data resource* concept is introduced to carry the concern for identifying and inspecting external, independent data systems that may be known to programmers and users. It provides access to the large investment in those systems but does not require them to change.

4.2 Data Resources

To present a uniform service model that allows these diverse external data resources to be used we introduce a *data resource* that provides standard DAIS portTypes and normally implements their functions by interacting with *external data resources*. Frequently a *data resource* will represent just one *external data resource*, but it may represent many, e.g. presenting a federation of their facilities, or it may provide its functions independently. The *data resources* are therefore the “work-horse” services for delivering data access and integration. This is illustrated in Figure 3. They may be viewed as proxies for *external data resources* or adapters or wrappers for those *external data resources*.

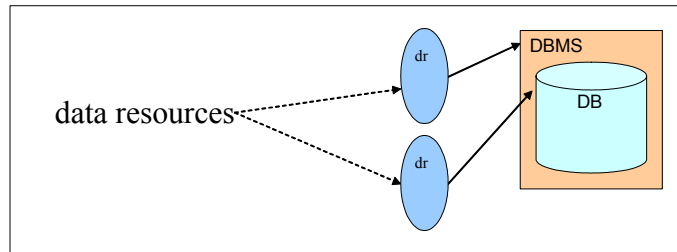


Figure 3 – Data Resources interfacing external data resources

The design of the *data resource* service has to meet two apparently conflicting goals:

- It must present a regular and simple structure of functions that can be readily understood by service implementers and application developers.
- It must make available the wide variety of models, languages and operations that are provided by all of the versions of all of the database management systems, file systems and information systems that occur in external data resources that DAIS users may wish to access.

These are jointly met by a `performImmediate` portType that provides an immediate synchronous response to a *data request* packaged in a *dataset* (see section ??). The *response* is similarly encoded in a *dataset*. *Datasets* are defined to be sufficiently general to accommodate all of the variations of data required for the full variety of operations that may be encountered. A *data request* will be formulated in terms of *activities* (see section ??). *Activities* are an extensible set of operations, that may carry any of the versions of the prevailing data manipulation and query languages, and thereby they enable the independent consideration of different data models, database management system versions and different categories of database and data transformation functions. This synchronous interaction with a data resource is illustrated in Figure 4.

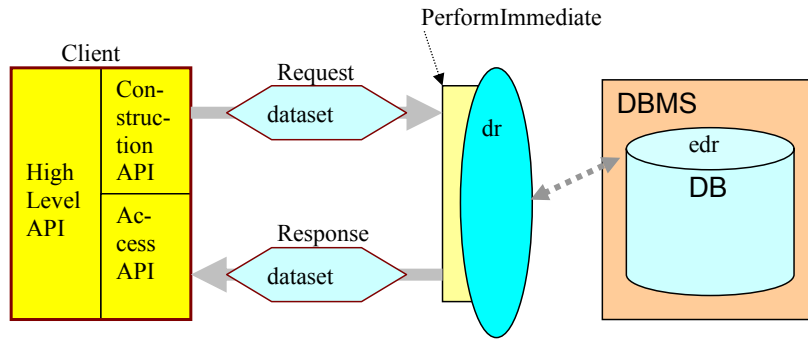


Figure 4 – Synchronous request-response pattern of *data resource* use

The communication between the *data resource* (dr) and the *external data resource* (edr) it is representing, shown as a dotted arrow, can be by any mechanism chosen by the implementers of the dr service. For example, if they implement in Java and the edr is a relational database they might communicate via JDBC.

The synchronous interaction pattern is suitable for many data operations that involve relatively small amounts of data, where as networks gain bandwidth, the issue of avoiding latency becomes paramount. For these sufficient information is passed to complete the requested activity (or activities), and the client has no further interaction with the *data resource* until the response is received. This has the additional advantage that the failure modes that must be considered by the client are simplified compared with those that may occur in asynchronous interactions made available via *data activity session* services (see section ??).

Data resources will present their capabilities and some state information, such as available resources, via the usual service data element (SDE) mechanisms [eee]. This will normally include information about what activities they are prepared to enact. One such activity may be schema enquiries that give fuller schema information than the SDEs provide. The SDEs will normally indicate which *external data resources* (if any) are being represented. As *data resources* only support synchronous behavior, they do not require SDEs that reflect progress with data activities.

Certain activities may create a new *data resource*, e.g. activities that request the creation of a database, a view or a materialized view. Some *data resources*, e.g. those representing an instance of a DBMS, may be able to reveal the existence of other *data resources* that they are able to make available. In each case, we can think of these data resources as being *derived data resources*. As a multiplicity of data models are considered, these *derived data resources* may offer all or many of the operations of their *primary data resource*, e.g. a file server may yield derived directories as *data resources*. Consequently, it does not make sense to introduce a different class of services for the *primary data resources*, as was suggested in the GGF8 version of the DAIS draft specification.

A data resource discovery mechanism, e.g. a registry of *data resources*, will need to hold a set of *primary data resources*, with which clients can initiate their interactions. The registry may also provide information about derived data resources.

Some of the possible relationships among data resources are illustrated in Figure 5.

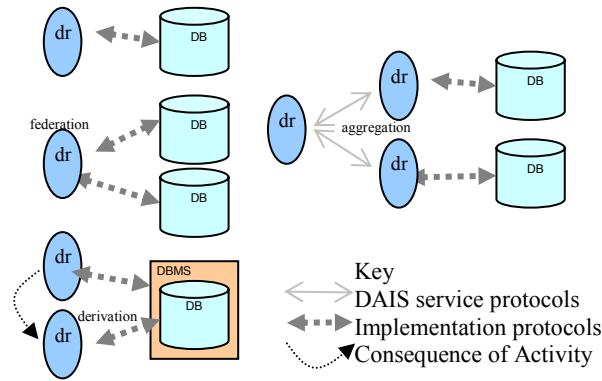


Figure 5 – Example configurations of data resources

In summary the conceptual role of a *data resource* is to support synchronous request-response patterns of use for a wide range of data models, data languages and co-existing versions of systems. They have a further role; that of creating *data activity session* services to support asynchronous activity.

4.3 Data Activity Sessions

The role of a *data activity session* service is to support longer-running activities including asynchronous requests, sequences of activities that are being orchestrated by a client, transactions, processes where progress monitoring is required and those where deferred delivery of results is required. A *data activity session* therefore accumulates state throughout the session.

Initially *data activity sessions* may be limited to supporting one client and to existing only as long as they can be maintained in memory. It is expected, however, that persistent and re-startable sessions will be needed, and there may be a case for allowing multiple clients, e.g. multiple participants in a transaction. The architecture should permit such variations.

A *data activity session* is created by a data resource by calling its `CreateDataActivitySession` portType. There will be an immediate response indicating whether the creation was successful and returning the identity, e.g. Grid Service Handle (gsh) [eee], of the newly created *data activity session*. The dataset supplied in the creation call may contain a data request that is sent directly to the new *data activity session*, so that computation on behalf of the client is initiated asynchronously.

A *data activity session* will provide SDEs that report on progress with requested activities. Many *data activity sessions* will be capable of deferred delivery. In this case, the SDEs will also indicate readiness of the constructed *dataset* for collection. Some *data activity sessions* may use other services for deferred delivery, or may offer a *dataset holding* service (see sections ?? and ?? respectively). The responses which provide information to enable deferred collection include a *data collection recipe* (see ??). A *data collection recipe* is a “collection note” which when delivered as specified, retrieves the identified *dataset*. This design avoids coupling the deferred delivery mechanics with clients that use the *data activity session* service, that is, the client does not need to know how deferred collection is arranged and a *data activity session* can use different mechanisms for different datasets.

Creation and use of a *data activity session* is illustrated in Figure 6.

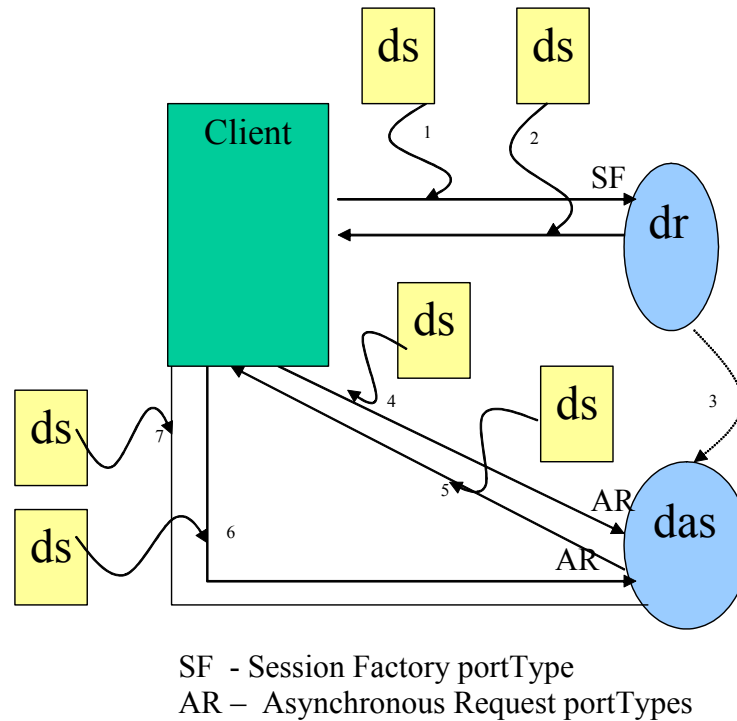


Figure 6 – The creation and use of a data activity session

In summary, the *data activity session* is introduced as a location in which to accumulate state pertaining to one or more activities and hence as a means of supporting asynchronous data operations. It is the locus for all aspects of data access and integration services that need to accumulate state about the progress of activities on behalf of client processes.

4.4 Datasets

The *dataset* is a packaging mechanism, so that various items of data, required as inputs and outputs to data intensive activities, can be collected together and then handled as a unit. Such data intensive activities are common in data access and integration, but also occur in computation intensive and data management applications. Therefore the definition of *datasets* is of wider interest.

It is helpful to think of the *dataset* as a padded envelope of arbitrary capacity into which an arbitrary sequence of separate data items of any form, table, XML document, DFDL file [9], DIME value [fff], etc. may be put during the creation of the *dataset*. That *dataset* is then sealed, and may be stored, moved or copied. Then another process may access the *dataset* and obtain the data items that were placed in it. There is a further discipline on *datasets*; that is, their contents are described, so that a recipient can inspect a *dataset* to discover what it contains and how to recover the information.

An additional property used by data access and integration services is that the *dataset* construct is recursive, that is, a data item in a *dataset* may itself be a *dataset*.

The motivations for *datasets* in this form are:

- To provide a convenient mechanism for sending and storing heterogeneous collections of values, so that developers of systems and applications are encouraged to group together values to be managed, stored or moved.

- This in turn, will reduce accumulated latency, handling and management overheads.
- To meet the requirement to deliver the same information as is transmitted for a variety and extensible set of possible values and mechanisms for describing values – the latter is helpful in permitting established practices to continue.
- To separate out the issues of data movement, storage, etc. from the issues of constructing or accessing a message.

The *dataset* concept is inspired by *datasets* in ADO .NET, but it differs in two respects:

1. The DAI version sets out to accommodate a wide variety of values and representations to respond efficiently to the long-term goal of accessing and integrating all forms of data resource.
2. The DAI version distributes its description of items as it may be impracticable with large, incrementally transferred messages, to defer sending the header until the complete contents are known.

It is an open issue whether the mechanisms for propagating updates in ADO .NET datasets should be assayed. They are likely to be expensive in the large scale distributed systems and require the maintenance of relationships. It is also difficult to find mechanisms that will work with the full range of data models. One possibility is to make these mechanisms available in selected case by higher-level services.

It is expected that libraries or objects for efficiently creating and accessing *datasets* will be implemented (see 3) and Appendix 2. It is also expected that there will be efficient mechanisms for moving *datasets*. These are both concerns for implementers but are not explicitly addressed by the architecture. The architecture requires a `send` operation, that indicates a *dataset* is complete and can be handed to a transmission or storage service, and a `get` operation that obtains the *dataset* and allows the access operations to commence. Note that this pair will allow incremental transmission, e.g. once a *dataset* is partially constructed, those populated parts can be sent⁵, the `send` in this case, causes buffers to be flushed and an “end-of-dataset” transmitted. Once data has started arriving the `get` can return, and access operations can block if they request access beyond the data transmitted so far. The `get` can pull data or find it already delivered. The `send` can push data or arrange its storage for deferred collection.

4.4.1 Dataset Contents

A dataset will consist of a *header*, a *summary*, a *body* and a *tail*. The *header* will indicate it is a dataset, which version of dataset formats it complies with, and a hard-to-forge indication of where it originated⁶. The *summary* will normally say how many data items are in the *body* and give their identifiers. The exception may be *datasets* which are effectively endless streams of data items. The *body* holds the sequence of data items as described below. The *tail* holds sufficient information to indicate that the *dataset* is complete and has a low probability of having been corrupted.

A *data item* is a self-describing value. It has an *identifier*, unique within this dataset, a *type* and a *value*, which is used in the construction and access interfaces to operate on a specific data item. The *value* is often composite, containing for example a schema that describes the specific value, an

⁵ For this reason, the interfaces provided to datasets should not permit data already written to be updated.

⁶ This is where the dataset was created. It may be being delivered from some quite different service after being handled by intermediaries.

indication of how the value is encoded and then the value. Any of these which can be inferred from the type may be omitted. This permits separate and incremental definition of types and there formats by groups working contemporaneously.

Figure 7 illustrates the internal organization of a dataset.

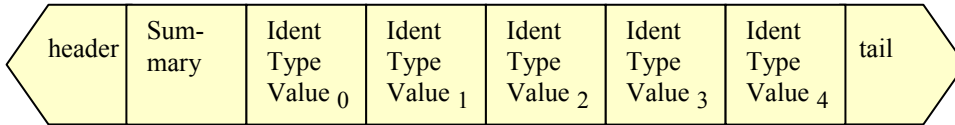


Figure 7 – the internal organization of a dataset

4.4.2 Specific dataset contents

Particular conventions prevail for the datasets used in DAI as inputs and outputs in portType operations, e.g. the request messages must contain at least one data request as their first element, their values must match the data model and activity requested, and so on. Similarly a response must contain a status response corresponding with each data request in the request message.

These two examples of specific datasets were introduced in Figures 1 and 4. Focusing only on the body in each case, we illustrate the specific contents in Figures 8 and 9 respectively. The text and XML is illustrative and not prescriptive, i.e. established nomenclature and schemas should be used.

The *request dataset* must begin with a data request element recognizable by being one of a set of specified types. In this case it requests an activity that performs an SQL query that uses a table as a value, e.g. in an IN clause. The table has been sent as a separate data item in some table representation, e.g. row set, perhaps because it was large or because it was convenient to copy it unchanged from some other dataset.

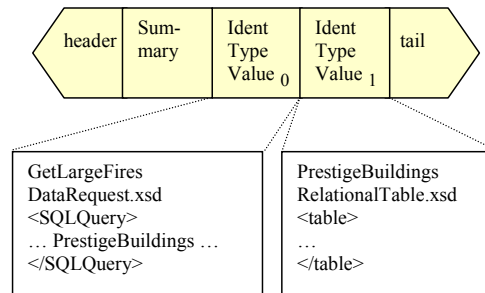


Figure 8 – An example request dataset

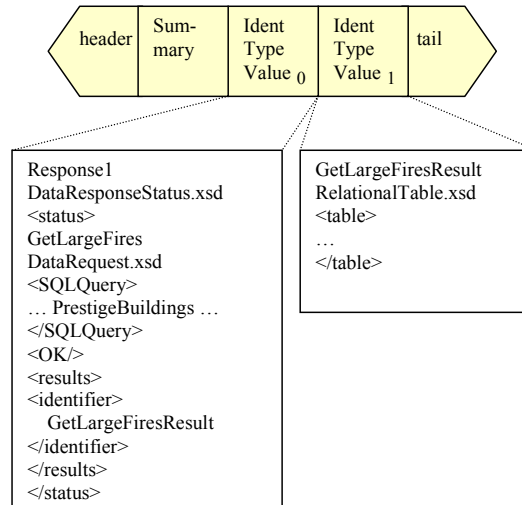


Figure 9 – An example of a corresponding response dataset

In the *response dataset* the first data item must be a response status data item, which (a) identifies the data request to which it responds, (b) indicates that the activity was completed OK or transmits exceptions indicating errors encountered, and (c) names the set of data items that are the result. In this case there is one represented as a table.

A *request dataset* may carry multiple *data requests*, in which case the destination *dr* or *das* is being asked to perform all of these requests. If every *data request* is performed successfully there will be a corresponding *data response* in the *response dataset*.

There are issues about whether the order in which multiple data requests are performed, about transactional behavior and its interaction with these data requests, and about processing after an error is detected that have yet to be explored.

4.4.3 Data Collection and Delivery Recipes

Data collection and *delivery recipes* are intended to allow an activity to specify third-party data collection and delivery, deferred delivery and so on. Indeed, they are a general mechanism that can cause a cascade of requests to obtain or generate input data and can specify any process being applied to output data. They are a special class of data items in a dataset that can be recognized by their specified types.

They will be illustrated using the example application process shown in Figure 10.

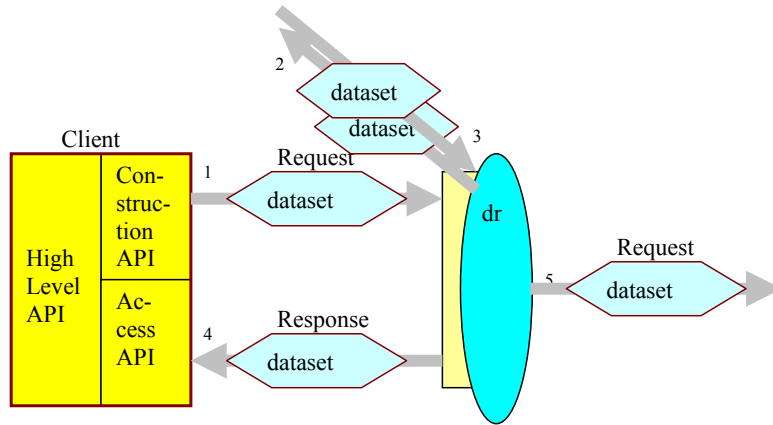


Figure 10 – An application requiring third-party data collection and delivery

Step 1 sends the initial *data request* containing two recipes, one stating how to collect some data, the other stating how to deliver a result. In step 2 dr sends a *dataset* that was in the *data collection recipe* to the `performImmediate portType` (or `performAsynchronous` if it chooses) of the supplier specified in the *data collection recipe*. In step 3, that supplier responds with the requested data. In step 4, the dr reports on the processing of the activities in the data request and delivers any results that were not specified for third-party or deferred delivery. In step 5, the dr assembles a *dataset*, which is that supplied in the *data delivery recipe* with additional data items appended that contain the data to be delivered. This *dataset* is now sent to the consumer specified in the *data delivery recipe*. The *data request(s)* in the *dataset* supplied in the *data delivery recipe* will state how to handle the incoming data items, e.g., it might create a table to hold each one and fill it with a bulk load, or create files, etc.

This mechanism is very general, for example, it can be used to pick up a result from one dr and then send it to another dr, with a *data request* that uses it in a further query, and so on. It can also be used to provide deferred delivery. The deferred items can be stored by the das, or by some other service on behalf of the das, and then the response will contain the *data collection recipe* that will allow the client or its delegate to perform the collection.

The form of *data collection and delivery recipes*, they are data items, are related to the activity that uses or generates them via the data item identifier. They are recognizable by being of type `CollectionRecipe.xsd` and `DeliveryRecipe.xsd` respectively. They then contain the identity of the supplier or the identity of the consumer, e.g. the gsh. The fourth item is the identity they will use, in the response to the collection request in the *data collection recipe* case, and to identify the data in the case of a *data delivery recipe*. Finally, there follows the complete *dataset* in the case of collection and the incomplete *dataset* in the case of delivery. This is illustrated in Figure 11, which partially expands the dataset transmitted in step 1 of Figure 10.

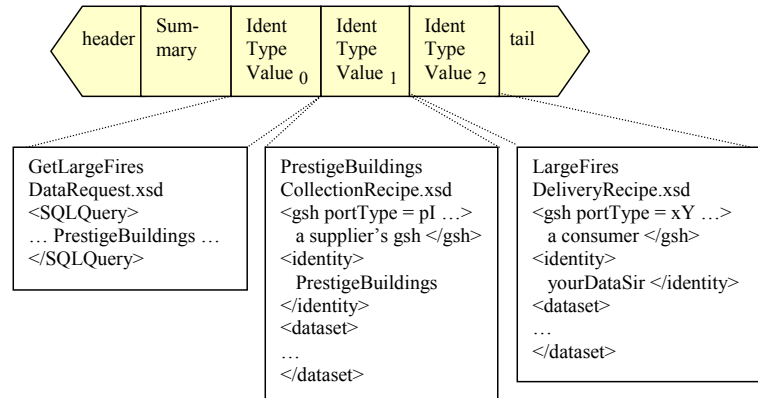


Figure 11 – Examples of data collection and delivery recipes

In the example the possibility of annotating the gsh to indicate which portType and which function to use is illustrated.

In summary, the data collection and delivery recipes are powerful and general purpose and could be used by a wide range of services. In the DAI context they enable third-party transfers, deferred delivery and multi-service cascading operations. As they are constructed by the party arranging for the data or for its collection, they allow a service to be asked to transfer data without that service having to know about the service with which it is making transfers.

4.4.4 Dataset Summary

The dataset is used to group a heterogeneous sequence of data items together so that they can be handled and moved as one unit. The dataset is not a service and may be manifest as an object in many hosting environments. It provides for very flexible and extensible functionality in a framework that is self-describing and has regular properties.

4.5 Activities

Activities are introduced to name classes of functional capability supported by data resources. This serves three purposes:

1. It provides a mechanism for incremental definition. As it is required and implementation effort is available a new activity can be defined and added to the repertoire of various data resources and data activity sessions. In some implementations this activity extension may be supported by dynamic binding.
2. It provides a capability description mechanism; a service can advertise the activities it is prepared to perform, e.g. by listing their names in a SDE.
3. It provides a succinct notation for discriminating classes of functionality, when this is useful for system and application implementers.

There is a case for allowing limited forms of activity composition in order to permit several activities to be pipelined without intermediate data storage or data transfer. This must be limited to those cases where the composition behaves as a transaction, so that clients are not involved in recovery from partially computed compositions of activities. More complex compositions will be achieved by coordination engines or work-flow enactment engines behaving as the clients of DAI services.

5 Greg's Examples beyond here – to be absorbed

This document demonstrates how the most recently proposed DAIS conceptual model can support simple query and update processing and third party data delivery. In addition, the document uses an XML document schema that expresses the capabilities of the conceptual model. In many ways the current conceptual model is very close to the GGF7 specification document.

According to the conceptual model, a dataset is a collection of elements, each of which is a triplet (name, type, value). To interact with a data service, a client will create a dataset document containing requests for services and submit it to a perform method of the service. The service will respond with a dataset document containing the responses to the requests. Thus datasets are the values that are moved around between clients and services. A data activity session is able to retain state in the form of named datasets.

A dataset in XML looks like

```
<dataset>
  <element>
    <name> ... </name>
    <type> ... </type>
    <value> ... </value>
  </element>
  ...
</dataset>
```

6 Processing request datasets

A request document is a dataset that includes one or more request elements. That is, elements whose type is derived from request. E.g. `Request.Query.SqlQuery`. Similarly, a response document is one that includes one or more response elements. In the language of DAIS, each request element represents an instance of an activity type. Activities in DAIS are the fundamental operations that can be performed on data services.

From a `Request.Update.SqlUpdate` element, the response is an instance of type `Response.Update.SqlUpdate`. The response element includes status information plus references to the dataset elements that were included in the response dataset as a result of satisfying the request.

A data resource (dr) supports method `performImmediate` which executes the request elements of a single request dataset. A data activity session (das) allows the retention of state within the service and supports method `performAsynchronous` in addition to `performImmediate`. Additional examples and explanations including service interactions and the use of data activity sessions can be found in the document “DAIS Data Service Interactions and the SkyQuery Portal” [<http://www.nesc.ac.uk/~greg/skyquery.pdf>].

The example request datasets that follow each contain one or more request elements. The response to a multi-element request is a dataset that combines the response elements of the requests. No change in meaning results from combining requests.

7 Processing SQL select queries

The document of Figure 1 is a request dataset that includes a single request element, of type `Request.Query.SqlQuery`. The value tag of the element (lines 5-7) contains an SQL select query. This document can be sent to the `performImmediate` method of a data resource for processing.

```
1      <DataSet>
2        <element>
3          <name>req1</name>
4          <type> Request.Query.SqlQuery </type>
5          <value>
6            <SqlQuery>select * from customer</SqlQuery>
7          </value>
8        </element>
9      </DataSet>
```

Figure 1 Request dataset for SQL select statement

Figure 2 shows the response of an appropriate data resource to the request of Figure 1. The response document has two elements. The first element, in lines 2-10, is a direct response to the request element. It repeats the request details in line 6 and gives the name of the dataset element (line 6) that contains the table produced by the select statement.

```
1      <DataSet>
2        <element>
3          <name>req1</name>
4          <type>Response.Query.SqlQuery</type>
5          <value>
6            <request><SqlQuery>select * from customer</SqlQuery></request>
7            <Result name="req1.resp1"/>
8            <status>complete</status>
9          </value>
10       </element>
11       <element>
12         <name>req1.resp1</name><!-- contains the result table -->
13         <type>
14           <xs:schema>--table schema here in XML--</xs:schema>
15         </type>
16         <value><Table>...</Table></value>
17       </element>
18     </DataSet>
```

Figure 2 Part of the response dataset for the request of Figure 1

The second element of the response dataset (lines 11-17) contains the query result table. The type is given by including an XML schema for the table. Line 14 is a place holder for the schema, which is not included. Finally, line 16 is where the rows of the table, in their XML form, will be.

Several XML formats are appropriate for representing relational tables in the result document. The Sun Java WebRowSet format and the Microsoft ADO.Net DataSet format are useful candidates.

8 SQL update and DDL statements

Update statements and data definition language (DDL) statements do not return tables as results, but rather return either an update count (for update statements) or no result (for DDL statements). Figures 3 and 4 show the request and response documents for a delete request and a drop table request.

```
1      <DataSet>
2        <element>
3          <name>upd1</name>
4          <type> Request.Update.SqlUpdate </type>
5          <value>
6            <SqlUpdate>delete from Customer where lastName='Doe'
7            </SqlUpdate>
8          </value>
9        </element>
10       <element>
11         <name>ddl1</name>
12         <type> Request.Update.SqlUpdate </type>
13         <value>
14           <SqlUpdate>drop table Movie</SqlUpdate>
15         </value>
16       </element>
17     </DataSet>
```

Figure 3 An update request dataset:

```
1      <DataSet>
2        <element>
3          <name>upd1</name>
4          <type>Response.Update.SqlUpdate</type>
5          <value>
6            <request><SqlUpdate>delete from Customer where lastName="Doe"
7            </SqlUpdate></request>
8            <RowCount>2</RowCount>
9            <status>complete</status>
10         </value>
11       </element>
12       <element>
13         <name>ddl1</name>
14         <type>Response.Update.SqlUpdate</type>
15         <value>
16           <request><SqlUpdate>drop table Customer</SqlUpdate></request>
17           <status>complete</status>
18         </value>
19       </element>
20     </DataSet>
```

Figure 4 Response dataset for request of Figure 3

9 Bulk load

A create table request, as shown in Figure 5, is an instance of the more general bulk load operation. Figure 5 includes a request to create a table from a dataset element. That is, to interpret the type of the element as the schema of the table and the `CreateTable` tag in line 6 specifies the name of the new table as an attribute. The schema and values for the new table are contained in the dataset element named `table1`, shown in lines 9-15.

```
1    <DataSet>
2      <element>
3        <name>req1</name>
4        <type> Request.Update.SqlCreateTable </type>
5        <value>
6          <CreateTable name="movie"><Input element="table1"/></CreateTable>
7        </value>
8      </element>
9      <element><!-- example of this element given in appendix -->
10     <name>table1</name>
11     <type>
12       <xs:schema>--table schema here in XML--</xs:schema>
13     </type>
14     <value><Table>...</Table></value>
15   </element>
16 </DataSet>
```

Figure 5 SQL create table request dataset

```
1    <DataSet>
2      <element>
3        <name>req1</name>
4        <type>Response.Update.SqlCreateTable </type>
5        <value>
6          <request>
7            <CreateTable name="customer2"><Input element="table1"/>
8            </CreateTable>
9          </request>
10         <RowCount>15</RowCount>
11       </value>
12     </element>
13 </DataSet>
```

Figure 6 Response dataset for create table request of Figure 5

10 Processing errors and response documents

Each request element has specific meaning to the dr. If the dr is incapable of performing the request, the response dataset includes an exception tag. Figure 7 shows an example

```
1    <element>
2      <name>req1</name>
3      <type>Response.Query.SqlQuery</type>
4      <value>
```

```

5         <SqlQuery>select * from cxstomer</SqlQuery>
6         <exception><type>SqlQueryError</type>
7           <message>No table named cxstomer in database</message>
8         </exception>
9       </value>
10    </element>

```

Figure 7 Example of response document with exception

The processing of other requests within the same document may or may not be effected by the error. The DAIS standard will allow implementers to enforce various strategies for exception handling. Examples of possible interpretations follow.

A linear model of request processing would require that each subsequent request also have an error response. In this context, "subsequent" means in the order processed. An atomic transaction model would require that all updates in the request be cancelled. We probably want the response to include all of the information generated by the successful execution, together with a rollback tag.

11 Sample scenarios for performImmediate with delivery to/from 3rd party

[The following has yet to be explained]

The request document of Figure 8 specifies that the result of the query is to be delivered by ftp and not returned within the response document. The SqlQuery tag includes an output tag (lines 7-9) that identifies the element named delivery1 is to be used as the data delivery recipe.

```

1     <DataSet>
2       <element>
3         <name>req1</name>
4         <type> Request.Query.SqlQuery </type>
5         <value>
6           <SqlQuery>select * from customer
7             <output>
8               <DataDeliveryRecipe element="delivery1"/>
9             </output>
10          </SqlQuery>
11        </value>
12      </element>
13      <element>
14        <name>delivery1</name>
15        <type>DeliveryRecipe</type>
16        <value>
17          <recipe>
18            <ftp>
19              <server>ftp.nesc.ac.uk</server>
20              <file>/pub/customer.xml</file>
21              <usr>ftp</usr>
22            </ftp>
23          </recipe>
24        </value>
25      </element>
26    </DataSet>

```

Figure 8 Sql query request dataset with 3rd party delivery

The delivery recipe (lines 13-25) specifies an ftp delivery. Lines 18-22 list the server name, file name and user id of the ftp transfer.

Figure 9 shows the response to the request. No data is returned to the client.

```
1      <DataSet>
2        <element>
3          <name>req1</name><!-- repeats the query and identifies the result -->
4          <type>Response.Query.SqlQuery</type>
5          <value>
6            <request>
7              <SqlQuery>select * from customer
8                <output>
9                  <DataDeliveryRecipe element="delivery1"/>
10               </output>
11             </SqlQuery>
12           </request>
13           <status>complete</status>
14         </value>
15       </element>
16     </DataSet>
```

Figure 9 Response dataset from the request of Figure 8

12 Appendix: Example of table schema and values in XML

This example is derived from the XML format generated by the Microsoft ADO.Net DataSet class. The document could be embedded in the document of Figure 5 as the input to a bulkload operation.

```
1      <DataSet xmlns="http://tempuri.org/">
2        <element>
3          <name>table1</name>
4          <type>
5            <xs:schema id="WebServiceData" xmlns=""
6              xmlns:xs="http://www.w3.org/2001/XMLSchema"
7              xmlns:mldata="urn:schemasmicrosoftcom:xmlmldata">
8              <xs:element name="movie">
9                <xs:complexType>
10                 <xs:sequence>
11                   <xs:element name="movieId" type="xs:int" minOccurs="0" />
12                   <xs:element name="title" type="xs:string" minOccurs="0" />
13                 </xs:sequence>
14               </xs:complexType>
15             </xs:element>
16           </xs:schema>
17         </type>
18       <value>
19         <movie>
20           <movieId>101</movieId>
21           <title>The Thirtynine Steps</title>
22         </movie>
23         <movie>
24           <movieId>123</movieId>
```

```

25     <title>Annie Hall</title>
26 </movie>
27 <movie>
28     <movieId>145</movieId>
29     <title>Lady and the Tramp</title>
30 </movie>
31 </value>
32 </element>
33 </DataSet>

```

13

Bibliography

1. Antonioletti, M., Atkinson, M., Chue Hong, N.P., Krause, A., Malaika, S., McCance, G., Laws, S., Magowan, J., Paton, N.W. and Riccardi, G., *Grid Data Service Specification*, in *D.AIS-WG Papers*, N.W. Pearson D. and Paton, Editor. 2003, Global Grid Forum.
2. Riccardi, G., *A Primer for Data Access and Integration*. in preparation, 2003.
3. Hey, A.J.G.a.T., A., *The Data Deluge: an e-Science Perspective*, in *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, Fox, G.C. and Hey, A.J.G., Editor. 2003, Wiley: Chichester. p. 809-824.
4. Atkinson, M.P., Dialani, V., Guy, L., Narang, I., Paton, N.W., Pearson, D. and Watson, P., *Grid Database Access and Integration: Requirements and Functionalities*, in *GGF7 Papers*. 2003, Global Grid Forum, Data Access and Integration Services Working Group.
5. Atkinson, M.P., Chervenak, A.L., Kunszt, P., Narang, I., Paton, N.W., Pearson, D., Shoshani, A. and Watson, P., *Data Access, integration and Management*, in *The GRID (second Edition)*, I.a.K. Foster, C., Editor. 2003, Morgan Kaufmann: San Francisco.
6. Gray, J., *The Economics of Distributed Computing*. 2003, Microsoft Research. p. 6.
7. Adesanya, A., Azemoun, T., Becla, J., Hanushevsky, A., Hasan, A., Kroeger, W., Trunov, A., Wang, D., Gaponenko, I., Patton, S. and Quarrie, D., *On the Verge of One Petabyte - the Story Behind the BaBar Database System*. Proceedings of 2003 Computing in High Energy and Nuclear Physics (CHEP03), La Jolla, Ca, USA, 2003.
8. Westhead, M.a.B., M., *Representing Scientific Data on the Grid with BinX*. 2003, EPCC, University of Edinburgh.
9. Westhead, M.D., Wen, Q. and Carroll, R., *Describing Data on the Grid*. accepted for 4th International Workshop on Grid Computing (Grid 2003), 2003.
10. Kleese van Dam, K., Sufi, S., Drinkwater, G., Blanshard, L., Manandhar, A., Tyer, R., Allan, R., O'Neill, K., Doherty, M., Williams, M., Woolf, A. and Sastry, L., *An integrated e-Science environment for environmental science*. submitted to World Scientific, 2003.