Report of the LHC Computing Grid Project

Persistency Management RTAG

Rene Brun, Dirk Duellmann, Vincenzo Innocente, David Malon (convenor), Pere Mato, Fons Rademakers

CERN

5 April 2002

Table of Contents

1 Convenor's Summary 3

2 SC2 mandate to the RTAG 4

2.1 Guidance from the SC2 4

2.2 Response of the RTAG to mandate and guidance 4

2.3 RTAG timetable 5

3 Scope and Requirements 6

3.1 Scope 6

3.2 Principal use cases 6

3.3 Requirements 6

4 Architecture Design 9

4.1 Design Criteria 9

4.2 Service model 9

4.3 Interface model 10

4.4 Other Issues 10

4.5 Component breakdown 11

5 Specific recommendations to the SC2 18

5.1 Recommendations for near-term work 18

5.2 Recommendations regarding potential common components not addressed in detail in this report 18

5.3 Longer-term R&D recommendations 18

6 Product Specification for a near-term common project 19

6.1 Charge to a first common project on persistence 19

6.2 Implementation Technologies 19

6.3 Components 19

6.4 Resource Estimates 21

1 Convenor's Summary

This document is the final report of the LHC Computing Grid Project's Requirements Technical Assessment Group (RTAG) on persistency.

The mandate to this RTAG, reproduced in the next section of this document, was quite ambitious. We did not manage to satisfy the mandate in its entirety, and those issues that we have managed to consider have been treated at uneven levels of detail. We do hope, however, that we have succeeded in our principal objective, which has been to provide a clear and solid foundation for an initial collaborative project to develop a common persistence infrastructure.

There are several matters, some related to the architecture generally and some involving more detailed specification of particular components, regarding which we have not, admittedly, reached full consensus. We have chosen not simply to omit all such material from this document, but rather, to include some of this material as a record of our thinking. We expect that a common project will preserve the spirit if not the details of our recommendations, resolving issues that we have not managed to resolve in the limited lifetime of this RTAG, and addressing our many acknowledged omissions.

With the SC2's concurrence, we propose to conclude the RTAG effort at this point in the interest of timely commissioning of a common project. We realize that we have not resolved many genuinely important issues, but we are confident that we will have the opportunity to continue these discussions in a future forum, for we view this RTAG as merely the first step in an ongoing and fruitful collaboration in which we all look forward to participating.

David Malon

5 April 2002

2 SC2 mandate to the RTAG

- Write the product specification for the Persistency Framework for Physics Applications at LHC.
- Construct a component breakdown for the management of all types of LHC data
- Identify the responsibilities of Experiment Frameworks, existing products (such as ROOT) and as yet to be developed products.
- Develop requirements/use cases to specify (at least) the metadata /navigation component(s)
- Estimate resources (manpower) needed to prototype missing components

2.1 Guidance from the SC2

The RTAG may decide to address all types of data, or may decide to postpone some topics for other RTAGS, once the components have been identified. The RTAG should develop a detailed description at least for the event data management. Issues of schema evolution, dictionary construction/storage, and object and data models should be addressed.

2.2 Response of the RTAG to mandate and guidance

It is the intent of this RTAG to assume an optimistic posture regarding the potential for commonality among the LHC experiments in all areas related to data management. The RTAG hopes to come to a consensus regarding identification of the principal components of a data management architecture, and, where possible, to come to agreement on the roles and responsibilities of those components. The RTAG plans further to identify which components might be candidates for shared implementation, and which are likely to be experiment-specific. From among the components that could in principle be common to the experiments, the RTAG proposes to identify those that should be given the highest priority as the LCG seeks to initiate common development efforts, as well as to indicate which components are expected to come from (other) grid projects.

The limited time available to the RTAG precludes treatment of all components of a data management architecture at equal depth. The RTAG will propose areas in which further work, and perhaps additional RTAGs, will be needed. Consonant with the guidance from the SC2, the RTAG has chosen to focus its initial discussions on the architecture of a persistence management service based upon a common streaming layer, and on the associated services needed to support it.

While our aim is to define components and their interactions in terms of abstract interfaces that any implementation may support and must respect, it is not our intention to produce a design that requires a clean-slate implementation. For the streaming layer and related services, we plan to provide a foundation for an initial common project that can be based upon the capabilities of existing implementations, and upon ROOT's streaming capabilities in particular. While the extent of new capabilities required of an initial implementation should not be daunting, we do not wish at this point to underestimate the amount of repackaging and refactoring work required to support common project requirements. Resource estimates are expected in the final RTAG report.

2.3 RTAG timetable

The RTAG convened in ten three-hour sessions during the weeks of 28 January, 18 February, and 11 March, and delivered an interim report to the SC2 on 8 March. An additional report was provided during the LCG Launch Workshop on 12 March. A final report to the SC2 is expected on 5 April 2002.

3 Scope and Requirements

3.1 Scope

The intention of the RTAG is to address all types of LHC data used in offline event processing—simulation, reconstruction, and analysis—including associated metadata. While the persistence infrastructure we propose must suffice to support construction and operation of LHC-scale event stores, we do not propose an infrastructure specialized to event data—it should be suitable for event and non-event data (geometry, detector description, conditions) alike.

3.2 Principal use cases

This RTAG has, admittedly, considered use cases principally from specialists' viewpoints, e.g., from the point of view of a developer of an experiment's event processing framework, so that, for example, a use case might be "store the event currently in my (experiment-specific) transient object cache, saving only the constituent data objects that I have designated."

For high-level use cases, we refer the reader to the large number of examples gathered for the Hoffmann review, in experiment-specific computing technical proposals, and in many grid projects (e.g., by WP8 for the EU DataGrid).

3.3 Requirements

3.3.1 Language

Transient event models for all LHC experiments are currently implemented in C++, and it is a requirement that any near-term implementation support saving and restoring the states of C++ objects efficiently. Multilanguage environments are, however, a reality (e.g., FORTRAN for simulation, Java for visualization and analysis), and multiple languages may infiltrate event data models as well. Designs must not assume that either the implementation language or the type of an object being restored is the same as that of the object whose state was saved.

3.3.2 Scalability

The data scale of LHC experiments—hundreds of megabytes per second, petabytes per year, thousands of processors processing and contributing data, globally distributed analysis by thousands of physicists—has been well documented elsewhere. An LHC persistence infrastructure must support data storage and retrieval at these scales.

3.3.3 Schema evolution

The term "schema evolution" is used rather loosely in HEP circles, meaning, variously,

– The ability to change the definition of a transient class Ti (without changing its name), and to build such a revised Ti from the same persistent state representation Pj that was previously used;

- The ability to change the persistent representation Pj of the state of a transient object type Ti, and to build objects of type Ti from both the old and new state representations;
- The ability to update persistent data, changing from an old representation Pj to a newer one, without compromising the integrity of any persistent pointers to such data;
- Various combinations of the above, particularly of the first two.

The design of a persistence infrastructure must recognize and support the potential for evolution of both transient models and persistent representations.

When the state of an object of transient type Ti is saved, a particular persistent representation Pi is chosen. That representation may be automatically generated from the definition of Ti, and a "converter"—the code to save and restore state using this representation--may be generated as well. This model, in which there is a one-to-one correspondence between transient types and persistent representations (at least within a single storage technology), is expected to be a typical use of persistence services.

Over time, better representations Pj may be found and implemented, and the definition of Ti may change as well. It is in fact possible that transient models may advance sufficiently that the object to be initialized from a state representation Pi may be of a type altogether different from Ti.

These considerations imply the following requirements.

- The architecture should assume that a single transient type Ti may be restorable from many different persistent representations {Pj}.
- The architecture should assume that a particular persistent representation Pj may be used to initialize transient objects of various types {Ti}.

The consequence of the previous is that on reading, the persistence infrastructure should not presume that the type of the transient object that will be initialized can be deduced simply by inspecting the persistent representation that will be used as input.

3.3.4 Multi-technology persistence

While we propose that a near-term common project focus on a single persistence technology for event data streaming, the design must not preclude use of multiple storage technologies, alternatively or in tandem, nor should components make assumptions about the implementation technologies used by the components with which they interact.

The use of relational database technologies to manage access to streaming layer data provides an opportunity to develop an approach that supports coherent management of other experiment data as well—XML geometry files and output of legacy FORTRAN simulations are two examples.

Even in a single-streaming-technology/single-relational-implementation environment, certain data may appear in either or both layers, and transparency of access, independent of storage technology, must be provided.

3.3.5 Control of physical placement

The persistence framework must support writing different events to different physical locations (e.g., to different files). This requirement is motivated principally by the need to support multiple physics streams.

The persistence framework must support writing different data objects within an event to different physical locations, e.g., to allow physical separation of raw muon data from raw calorimeter data.

Typically the control of the placement is not with the creator of the object. An algorithm creating an event object should not have to decide where it should be placed and not even if the object will be saved. Physical placement control should be done elsewhere and be configurable at run time.

3.3.6 Navigability

The experiment framework must provide transparent navigation to end-user physicists through object relationships, and must support load on demand. Therefore, the role of the persistence service is to provide persistency for (experiment-specific) transient data models allowing this transparent navigation. In addition it is desirable to be able to record associations among transient objects in the persistence layer in a way that allows navigation from one persistent object to another without reference to the experiment framework.

Transient data models will have associations between objects embedded in larger persistence-capable objects, e.g., between the tracks of a TrackCollection and the hits of a HitCollection, or between reconstructed particles and TruthParticles embedded in a generator event. It must be possible to create and persistify a reference to such an embedded object. From such a reference, it must be possible to navigate to the containing object in a manner independent of the experiment-specific framework. It is understood, however, that navigation to the embedded object itself may not be possible in certain cases without application-specific knowledge, as, for example, in the case that an embedded particle is indexed in the transient data model by a map or other nontrivial data structure.

3.3.7 Data Modeling

Experiments at this stage in their software development tend not to think about data models for their data stores. The viewpoint is that the persistent event representation simply echoes the transient one, with the addition of placement control to support physical clustering of objects likely to be requested together. This approach is well suited to projects in their development stages, and is well matched to automatic generation of converters and persistent representations. The architecture should, however, support the possibility of explicit persistent data modeling distinct from any specific transient views of data.

3.3.8 Other functional requirements

T - transactional consistency
C - crash recovery
P - data partitioning in time
R - support for read-only replication
L - support read only (e.g. laptop) sub setting/caching

4 Architecture Design

The architecture of the persistency system is described in terms of the components we have identified and their interactions. A software component is a part of the system, which performs a single function and has a well-defined interface. Components interact with other components through their interfaces.

4.1 Design Criteria

Here is a list of the main design criteria we have discussed in the RTAG so far:

- Abstract Interfaces. Components of the Persistency Framework should implement abstract interfaces and be as technology neutral as possible. Several implementations of a single component are not encouraged but they should be possible.
- The interaction between components should happen exclusively through the public and agreed interfaces. This is to avoid private communication between components (e.g. static storage) that will make impossible a later replacement of an implementation with another one.
- Typically the "end-user" (physicist writing a piece of code) does not interact directly with the framework abstract interfaces. A thin layer to hide the technicalities of such interfaces should be envisaged.
- Re-use existing implementations. If implementations already exists providing the required functionality they should be used to provide the initial implementation.
- We target C++ as the main programming language, however we should avoid constructions, which will make impossible the migration to existing or future new languages.
- Noninvasiveness. Transient objects whose states will be saved/restored will be compiled and linked without knowledge of any specific persistence technology.

4.2 Service model

Consider as an example a logical filename to POSIX filename mapping service. For an on-demand request, this may involve consulting a replica catalog to get a list of physical filenames, choosing the best instance, initiating a transfer if necessary, and finally providing a local filename. For a logical file that has been provided as part of the job description using EDG WP1 tools, the same service may be provided by simply consulting the BrokerInfo file through an EDG-provided interface to find the physical instance that has already been chosen for this job—no grid catalog lookups are required. Both approaches should provide the same service interface to the user—indeed, these may be implemented as a single service, with the choice of how to deliver the service made by the implementation.

In our vision of a hybrid architecture, the same may be true, for example, of a service to find a logical file that can resolve a specific Ref. In principle, the Ref→LFN translation may require querying a relational layer, but if the required resources have been marshaled and delivered in advance (e.g., to a laptop), we intend that the same service interface (the same service!) should be able to do the translation without consulting the relational layer.

4.3 Interface model

One essential ingredient when designing the architecture of a system is to agree upon the interface model. The interfaces of the different software components that collaborate to provide the high-level functionality are specified according to this interface model. Current design practice at the architecture level is to favor abstract interfaces when building collaborations of various classes. This is the way we can best decouple the client of a class from its concrete implementation. An abstract interface in C++ is a class where all the methods are pure

virtual.

The interface model should define some practical guidelines for defining interfaces. The following are issues to take into account:

- Interface naming. It is a good practice to have a convention for naming interfaces. In this way, developers will identify immediately what C++ classes are interfaces.
- Interface hierarchy. For example it is convenient to derived form a common base class interface.
- Concrete classes implementing multiple interfaces as it is the case for most modern languages.
- Interface identification and versioning. In theory, an interface is a contract between a class providing a service and the clients using the service. The interface should never change, but in practice this is very difficult to achieve since we do not know at the beginning the required functionality. To allow the detection of changes and eventually take corrective actions, it is convenient to version interfaces that can be queried at run-time.
- Reference counting. Since the services provided through abstract interfaces may be used by several clients, the interface model should define a mechanism for discarding services that are no longer needed. A possible mechanism could be based on reference counting.

Note: There is less than complete agreement on some of the points raised in the above section on an Interface model, particularly with regard to reconciling this approach with the efficient use of existing software. Some of the points, moreover, apply to essentially all components developed under the aegis of LCG common projects, not simply to a persistence infrastructure. We propose that these issues be taken up in the PEB's proposed "architects' forum."

4.4 Other Issues

4.4.1 Object Navigation

The framework should provide transparent navigation through object relationships with load on demand. In the transient object model, this relationship is provided by the transient type *Ref<T>* that is able to reference any persistent-capable object (or any object in general).

This *Ref* can be specialized and optimized for cases such as that T inherits from some sort of Storable-Object (d_*obj, PObject, TObjec*t) or if the referenced object is allocated using some ad-hoc memory manager. A *Ref<T>* should be buildable from a memory-pointer or from a *Token* returned by the persistent manager as a result of a streaming operation.

Ref<T> should be a persistent capable type by itself, i.e., mappable to a persistent layout *PREF* that may be technology specific.

A *Ref<T>* should be able to reference an object embedded into a container (such as an element of vector<T>) A collection of *Ref<T>* may be optimized to elide any common part.

A mechanism should be provided to ensure that:

- an object embedding a Ref<T> can be streamed before the object the Ref<T> references
- the use of Ref<T> does not imply noticeable performance penalties in case the embedding object and/or the referenced object are not eventually streamed
- an object not yet present in memory is properly retrieved when a corresponding Ref<T> is dereferenced
- the Ref<T> should not directly depend on any meta-data stored on files (storage-units?) different than those where the embedding object or the reference object are stored (no recompilation if the file catalog implementation changes)
- there should exist a technology-independent persistent representation of a Ref<T> (PREF); however, a proprietary, optimized, representation is not only allowed but encouraged.

The ability to generate a unique logical identifier, robust against relocation and cloning of the target object, is required.

Relocation or a cloning operation may be requested to generate a different PREF. The project is invited to evaluate the pro and cons of a PREF based on the physical location of the referenced object. Such an implementation may also be required as benchmark against other, more flexible, solutions. A collection of PREF may be optimized to elide (store once) any common part.

4.5 Component breakdown

Based on the discussions in the numerous meetings of the RTAG, we have identified a number of components in the persistency framework. The idea in this section is to enumerate them and describe the expected functionality of each one and interactions with the other components. Figure 1 shows the main components of the framework.

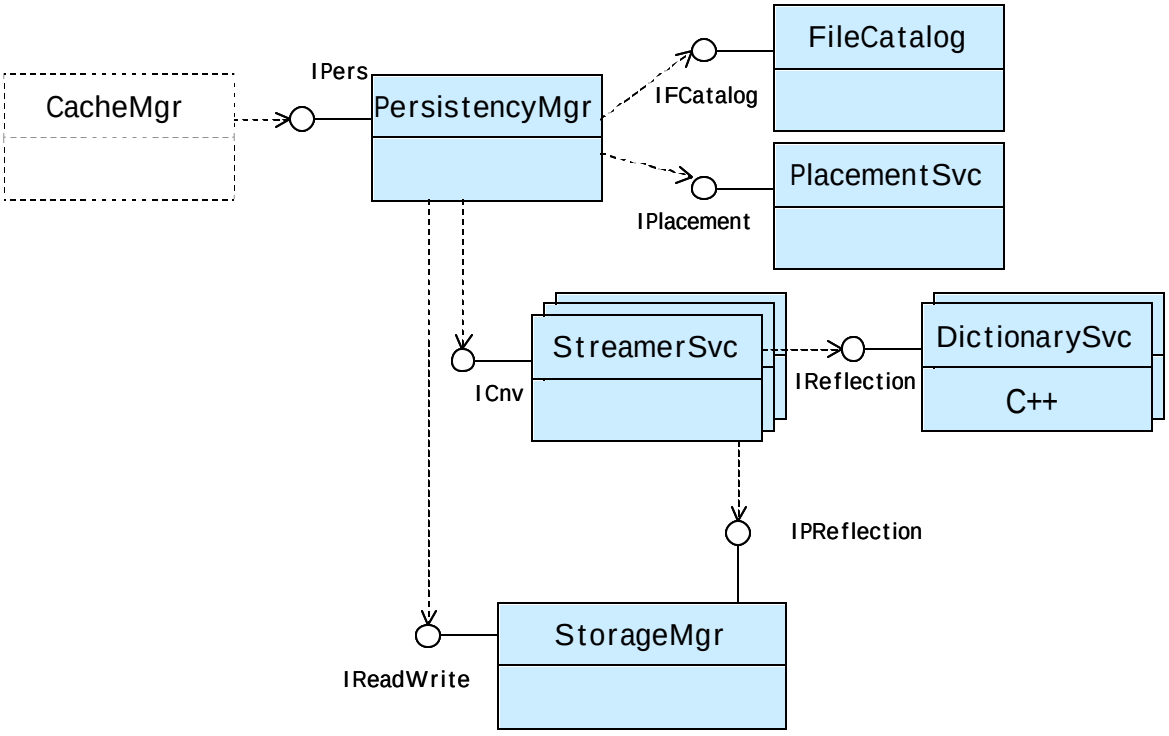

Figure 1 Component Diagram. This shows a three-layer organization of services.

4.5.1 Persistency Manager

The *Persistence Manager* is the principal point of contact between experiment-specific frameworks and persistence services. It is this service through which requests to store the state of an object, or to retrieve the state of an object, are made.

When an object O (of type T) is streamed, the persistence manager may be asked to return a **Token** that can be used at a later time to restore the state of that object. This token should be externalizable, i.e., representable and storable by a persistence mechanism different from the one that produced it (including human means such as writing it down on a piece of paper). Here "token" can be thought of as the persistent address of an object. We have chosen another word to avoid discussions of whether this is a physical address, a pseudo-physical address (like an Objectivity ooRef), or a name or key understood by the underlying technology.

The *Persistence Manager* should provide mechanisms to allow assignment of user-defined identifiers to a streamed object, and should provide optimized mechanisms to retrieve objects matching requests defined in terms of such identifiers.

4.5.2 Streaming or Conversion Service

A *Conversion Service* consults the transient and persistent representation dictionaries (see below) to produce a corresponding persistent (or "persistence-ready") representation of a transient data object, or to produce a transient object from such a representation. When the "converter" is a streamer, the persistence-ready representation is a variable-length stream of bytes.

Because of our implementation technology choices, we expect that automatic generation of streamers, like automatic generation of persistent layouts, will be possible with the components delivered by this initial project, with transient object descriptions as a starting point.

4.5.3 Dictionary Service

We expect (at least) two dictionaries, one describing transient classes to be made persistent, and the other describing persistent representations. The transient class dictionaries will be language-specific, and we expect a dictionary facility for C++ class descriptions as a starting point. It should be a component capable of producing persistent representation descriptions from transient ones, so that persistent dictionary entries may be generated automatically.

The transient data dictionary service, which is programming language specific, should provide an interface to obtain reflective information about classes and objects.

- The API will typically consists of a number of interfaces and [meta]classes that describe the object model. These classes should be as neutral as possible and not be dependent of the storage implementation technology. The expected functionality should be similar to the reflective capabilities that are provided by modern languages (e.g. Java, C#).
- The API should primarily be optimized for the retrieval of the reflective information by the "streamer service". We expect that the dictionaries will not be specific to a particular streaming technology. Two streaming technologies may use the same dictionaries, a non-streaming persistency technology may use the same transient class dictionary but a different persistent one; other services (e.g. interactive services, browsers, remote invocations, etc.) may also be clients.
- We expect no assumptions about how dictionary entries are generated--this may be automatic, or entries may be hand-coded. An API for filling the reflective information should be available. We expect that dictionary entries will be neutral with respect to the input format used by the tools that generate them. This would allow experiments to choose their own mechanism for describing their object models (e.g. using C++ header files, IDL, XML, debug information, etc.). This part of the reflective API is not provided typically by Java and C# since it is the compiler that does the filling work directly.
- The C++ dictionary API should support the standard C++ constructs (e.g., inheritance, methods, data members, accessibility, templates, etc.)

4.5.4 Store Manager

The *Store Manager* is responsible for storing a variable-length stream of bytes into the persistent store (file?) and for later being able to retrieve the same variable-length stream of bytes. It is responsible for managing the buffers currently in use and for storing/retrieving them in/from the proper file.

The *Store Manager* should provide asynchronous writing capability like any standard streaming technology. It should also provide a "flush" (seek) functionality which writes all open buffers and all required metadata on file. It should ensure the ACIDity of the operation. In particular, once flushed, the data should be available in read-only mode to any other process.

The *Store Manager* should be able to store more than one "object" per buffer. The details of the collaboration among *Persistency Manager*, the *Streamer* and the *Store Manager* itself required to achieve such a functionality is left to the design and implementation of the final product.

4.5.4.1 Files

In the following we call *file* a "collection of bytes suitable to be stored on disk" that looks to the user (in this case the store manager) as a sequential, randomly addressable, stream of bytes.

In C++ terms a file should be managed by a "iostream" class and addressable using iostream-iterators. Our current reference is C++ std::ios, even if one can limit oneself to files such as those present in current file systems (UNIX, Windows).

Besides the raw file format, that we assume to be completely transparent even to the *Store Manager* itself, the file will have a physical format determined by the store manager in use. Our current reference is the ROOT file format. We assume such a file format to be fully described and documented. Any change should be approved by all parties involved.

The default logical structure of the file may be a "name-tree" that mimics the Unix file system (in its looks and feel). The leaves will be "logical-records" (in the following also called "buffers"). Again the reference here is ROOT.

The metadata describing the actual logical structure of the file, (including persistent dictionary and any eventual object location table based on OID) should be stored in the corresponding file. It should also be possible to store and use this very metadata in other contexts. In particular one should be able to extract them and store them in a "metadata database" (object catalog?) implemented using a different technology.

4.5.4.2 Client/server architecture (on the grid)

The *Store Manager* should support client/server architectures. In particular it should be ready to support clever servers able to manage and resolve all metadata information locally avoiding the network and data-server load caused by multiple clients loading, for each "transaction," catalog, file, and object metadata.

4.5.4.3 Other Store Manager issues

- Storage organization adapted to data nature. The organization could be different for event data and detector conditions data.
- Different implementation adapted to data nature (event vs. detector vs. catalog), running environment (laptop, datagrid, etc.), activity (official production, software development, etc.)

4.5.5 Cache Manager

The *Cache Manager* is responsible for the access to for the lifetimes of transient objects whose states may be saved and restored. It is the *Cache Manager* (or an associated service) that communicates with the *Persistence Manager* to accomplish transient-to-persistent and persistent-to-transient conversions, controlling the timing and granularity of conversion requests, as well as determination of which transient objects are persistence-eligible.

The services associated herein with the *Cache Manager* are today provided by experiment-specific frameworks; hence, the scope of *Cache Management* services also varies by experiment.

4.5.6 Placement Service

The role of the *Placement Service* is to support control of physical clustering of data on output, by providing the Persistence Service with guidance regarding where to write an object (e.g., which file or which database or which container), equivalent to the "&where" hint provided to persistent "new" operators in object databases.

The *Placement Service* addresses the requirement that it must be possible to write different events to different physical locations (e.g., corresponding to different physics streams), and to write different data objects within in event to different physical locations as well.

4.5.7 File Catalogue

The service must provide the mapping for resolving any externalizable reference to a physical file name. It defines the list of all files which participate in one data store and their logical attributes (owner, read-only, last update....). It is used during the process of dereferencing a smart pointer. This component can be broken into two steps: reference to logical file identifier and from logical file identifier to the physical file (Posix file name) .

4.5.7.1 Reference→LFN Service

A service must be provided to map an externalizable Reference to a logical filename (LFN), in support of a file-based storage management layer in particular. We do not anticipate that an implementation of a Reference will necessarily contain a complete logical filename.

4.5.7.2 LFN→{PFN→}→PosixName Service (replica catalog)

This service maintains, for a list of (grid exposed?) files, the mapping to their physical counterparts. The main use case is to translate logical file name into a physical file name of a host close to the client. It should possible to query/browse all available replicas. This is very closely related to grid services, but may not be an identical instance (e.g., files which are not exposed to grid, but accessed in production).

We expect that such a component will be provided by other projects. Such a service is already provided in the EDG Testbed1 toolkit via the BrokerInfo service, which allows applications to determine the particular local files that have been identified by the WP1 scheduler on the basis of input LFNs listed in the job description. For more general access, one can query replica catalogs, choose appropriate replicas, transfer files, and so on.

4.5.8 Event Collection Manager

The Event Collection Manager is responsible for maintaining a collection of persistent event references and provide sequential (forward, backward, random?) iterators to access the contained objects (events). Collection by containment e.g. within a file (or group/chain of files) should be supported as well (e.g., via proxy objects). Transparent support of hierarchical collections (collections of collections) should also be supported. This components can be generalized in

many ways, e.g., collections for any kind of persistent object or polymorphic collections of objects.

4.5.9 Run and Event Collection Catalogue

This component maintains the metadata (mainly physics characterization) for runs or event collections. It maintains a catalogue of available runs or event collections together with their metadata (perhaps as (a table of) name/value pairs). The catalogue do not have to implemented using the same storage technology as the event data. This catalogue is used by end-user physicists to formulate a high level data selection based on the meta data information. Example: get event collection where name = "xyz" where creator = me where ...

4.5.10 Production Workflow Manager

This component maintains a consistent view of the status of a production activity at one or more data production sites. It keeps track of used job templates, job configuration and status, set of in/output files, the success or failure of machine job and used resources (time, cpu time, memory, disk space). The service allows one to query/browse jobs by type (sim, evgen,...), status (completed, failed,...) or other job related metadata (produced files, originating host, processing host, etc.). Some grid projects propose related components.

4.5.11 Detector Geometry and Conditions Service

This service maintains consistent sets of geometry representations and detector conditions (calibration and environmental parameters) used for simulation, reconstruction, etc. The kind of data this service maintains has a time interval validity and is versioned. Example use cases are:

- get quantity X valid at time T as obtained by processing version V
- may need to support hierarchical structured naming space for X
- may need to support consistent versioning across many X

4.5.12 Job Configuration Service

This service maintains consistent versions of the central configuration parameters of production and user jobs. Examples of these parameters are: event generator parameters, simulation parameters, used geometry version, reconstruction parameters and version, etc.

4.5.13 Hierarchical Mass Storage System

This component maintains bulk data files (independent of *Storage Manager*), and is implemented typically as multi-level storage. It should support media migration and file level import/export.

4.5.14 Data Analysis Warehouse

This component maintains a (read-only) view of all data which is relevant for end user analysis. It should support reclustered and indexed data representations for specific needs of fast/interactive analysis, parallel query evaluation and many concurrent users.

5 Specific recommendations to the SC2

5.1 Recommendations for near-term work

We recommend the commissioning of an initial common project to develop a common object streaming layer and associated persistence infrastructure. The project should deliver the components of a common data management architecture. These components include a common object streaming layer and several related components to support it, including a (currently lightweight) relational database layer. Specifications for such a product appear in the following section.

The authors acknowledge at the outset that the scope of the proposed project is only one part of a complete data management environment, one with enormous potential for commonality among the experiments. This limited scope for a first common project in data management is intentional.

Note that dictionary services are included in the near-term project specification. The RTAG recognizes that while the persistence infrastructure imposes many requirements upon dictionary services, it is not the only potential client of such services, and that another RTAG to consider the requirements of those additional clients and their implications for the design of dictionary services may be appropriate.

5.2 Recommendations regarding potential common components not addressed in detail in this report

Conditions databases:

We agreed that an interval-of-validity-based retrieval infrastructure is necessary, and that a common service interface should be possible. One could imagine the service itself being implemented in a relational technology, managing access to objects whose persistence may or may not be provided by the proposed common streaming infrastructure. We did not reach consensus on a potential common project, though the ATLAS and LHCb are already involved in joint work in this area, and hope to see it continue.

5.3 Longer-term R&D recommendations

The principal focus of our work has of necessity been on near-term projects, but the RTAG strongly recommends that the LCG include longer-term R&D as an important component of its effort profile and task portfolio. Specific recommendations here include tracking of emerging trends and technologies, and of potential alternatives and fallback solutions to our proposed common approaches, including continued explorations at a low level of effort with Oracle9i. To support the LHC experiments' ability to contribute to such prototyping, we recommend that development of a procedurally lightweight installation of Oracle9i for the standalone laptop—no more difficult than installing an experiment's own software—be investigated.

6 Product Specification for a near-term common project

6.1 Charge to a first common project on persistence

The charge to this initial common project is to deliver the components of a common file-based streaming layer sufficient to support persistence for all four experiments' event models, with management of the resulting files hosted in a relational layer.

While the design we propose is intended to be quite general, respecting abstract interfaces, allowing multiple persistence technologies, and permitting multiple implementations, we propose delivery of a single implementation based upon specific technology choices listed in the next section.

Persistence support means storage and retrieval of events currently defined in C++ without intrusion into experiments' current event models, and without requiring run-time or link-time dependence between those models and this project's persistence technology choices. Event persistence support, in addition to simple storage and retrieval, includes placement control (physical clustering). Since physics applications rarely process exactly one event, the project must also deliver support for creation of and iteration over event collections.

"Management" of the resulting files includes maintaining queryable associations between event collections and the logical files that hold event data, and provision of services to resolve references to and between objects in terms of the logical files in which the corresponding objects' persistence states reside.

Definitions of specific components, and explicit requirements, appear in the following sections.

6.2 Implementation Technologies

The initial streaming layer should be implemented using the ROOT framework's I/O services.

Components with relational database implementations should make no deep assumptions about the underlying relational technology at this point. The RTAG has chosen not to recommend a specific relational technology for an initial implementation, but has not intentionally proposed anything that precludes implementation using such open source products as MySQL.

6.3 Components

6.3.1 Persistence Manager

The Persistence Manager is the principal point of contact between experiment-specific frameworks and persistence services. It is this service through which requests to store the state of an object, or to retrieve the state of an object, are made. The functionality required is described in 4.5.1.

6.3.2 Streaming or Conversion Service

A streaming conversion service consults the transient and persistent representation dictionaries to produce a corresponding persistent (or "persistence-ready") representation of a transient data object, or to produce a transient object from such a representation. The functionality required is described in 4.5.2.

6.3.3 Dictionary Service

The dictionary service should provide an interface to obtain reflective information about classes and objects. It should describe transient classes to be made persistent and should also describe their persistent representations. The functionality required and the design guidelines are described in 4.5.3. This component is included in the near-term project specification but the RTAG recognizes that the streaming service is not the only potential client of such service, and that another RTAG to consider the requirements of those additional clients and their implications for the design of dictionary services may be appropriate.

6.3.4 Store Manager

The store manager is responsible for storing a variable-length stream of bytes into the persistent store and later for retrieving the same variable-length stream of bytes. The functionality required is described in 4.5.4.

6.3.5 Cache Manager

While we recognize the potential benefits of a common cache management component, we do **not** recommend delivery of such a component in this initial project. We wish to ensure that our initial deployment supports all four experiments' current frameworks and event models, and to ensure conformance to the project's abstract interfaces; for these reasons, we propose to defer potential common work in the area of transient object cache management.

6.3.6 Placement Service

The role of this component is to support control of physical clustering of data on output, by providing the Persistence Service with guidance regarding where to write an object. We recognize that we have not done a sufficient job of describing how the service should work. A preliminary description is in 4.5.6.

6.3.7 Event Collection Services

The project should provide support for explicit event collections—not simply collections by containment (e.g., "the events in this file"), but rather, collections at least equivalent in functionality to lists of References to events. The functionality required is described in 4.5.8.

6.3.8 References and Reference Services

The framework must provide transparent navigation through object relationships, and must support load on demand. The functionality required is described in 4.4.1.

6.3.9 File Catalogue Services

6.3.9.1 Reference→LFN Service

The framework must provide a service to translate object references (or part of) to logical file name (LFN) implemented using a relational database. Details of the required functionality are described in 4.5.7.1. We propose that an implementation of this service interface be provided in such a way that runtime access to the relational layer is not, however, a requirement—a "pre-query," for example, may resolve Reference→LFN mappings, akin to what BrokerInfo services provide in EDG, so that runtime relational queries are not required.

6.3.9.2 LFN→{PFN→}→PosixName Service

An LFN→PosixName service is required. We expect that such a component will be provided by other projects (e.g. EDG). This project requires a standard service interface to such information. Details of the required functionality are described in 4.5.7.2.

6.4 Resource Estimates

We have elected to omit resource estimates, since we did not have sufficient time in RTAG sessions to discuss them.