# LCG Persistency Project

## requirements and priorities
## component breakdown
## refs and navigation

Dirk Düllmann, IT-DB
*dirk.duellmann@cern.ch*

LCG Persistency Workshop, June 5th 2002

# Persistency Project – How to get started?

- Persistency RTAG has delivered its final report to SC2
  - Describing an agreed top level component model
  - Agreement has been achieved in some areas by leaving controversial issues open
- The Persistency project is a development project
  - Close release dates – will initially provide restricted functionality
  - Limited scope – won't not solve all data management problems
  - Output is real s/w - pass acceptance tests by compiler and end users
- Need an even more pragmatic approach than RTAG
  - Little time for philosophical arguments
  - But need to make sure that any initial simplifications
    - Are accepted by LCG Architect's Forum
    - Come with a clear plan when the real solution can be achieved

# Scope of the Project

- Limit the initial project scope to the core persistency problem
  - Inline with RTAG project proposal
  - Other important data management components are outside of the persistency project
    - We should at least know which outside components later to integrate with
- Some Examples
  - Application Cache Management
  - ConditionsDB
  - Production Bookkeeping
  - Grid Replica Management
- Later RTAGs may change that
  - but this project can't realistically address all problems at once

# Goals of this workshop

- Agree on a component breakdown
  - Refinement of the RTAG component model
- Agree on the basic principle of their interaction
  - Draft interfaces define the main protocols involved
  - Expect significant changes as we go along
- Agree on the priority of their implementation
  - Based on input from the experiments…
  - … and estimated effort from implementation studies
- Establish work packages and release dates
  - Refining the draft interfaces
  - Starting work on their implementation

# How to collaborate effectively?

- Non-trivial development project
  - All development will be open and will follow development process agreed for LCG applications area
  - Visible in many client areas - many people will
    - find problems with their particular application area
    - have concrete ideas about their favourite solution/implementation them
  - Effective communication is and will be a problem

- Proposal: Prefer new solutions above new requirements/problems
  - Use the experiment internal communication channels for maintaining a prioritised requirement lists – one(!) per experiment
    - If no agreement can be achieved within an experiment about a new requirement – this project probably can help either!
  - If you have a concrete solution for an accepted requirement – just contact me (and your experiment maybe…)

# Proposed Project Name - PooL

## Why ?

- Pool of persistent objects for LHC
  - Open projects have recursive acronyms
  - You can navigate in it
    - see later presentations for large volumes examples
  - It's a container for a large volume of matter which is somehow hard to move
  - And if you don't like water anyway...
    - there is still an indoor game with colliding particles

- Please note: PooL is pronounced without "h" after the "P"

# Experiment Deployment Models

- Summary of first round of discussions with the experiments
- Timescale for first release – September (CMS)
  - Others are interested but more relaxed about date
- A few numbers referring to minimal requirements for a first release (rather than to constraints imposed by the design/implementation)
- Volume – 10-50TB (CMS)
- Files – several 100k (ALICE)
- Distribution – O(10) sites (CMS)
- Recovery from jobs failures – (CMS, ATLAS)
  - CMS: Objy based setup allows to just re-issue same job
    - Append to existing files
    - Would like to see at least the same functionality
  - Less extend
    - LHCb (each file is written by exactly one job)
    - Alice (event spans several files)
- Number of population jobs – 10k (CMS)
- Use of REFs per Event (CMS/ATLAS/LHCb)
  - ALICE no plans
  - LHCb: O(100) refs per event; CMS (100-1000);

# Experiment Focus of Interest

(prioritised by # of votes)

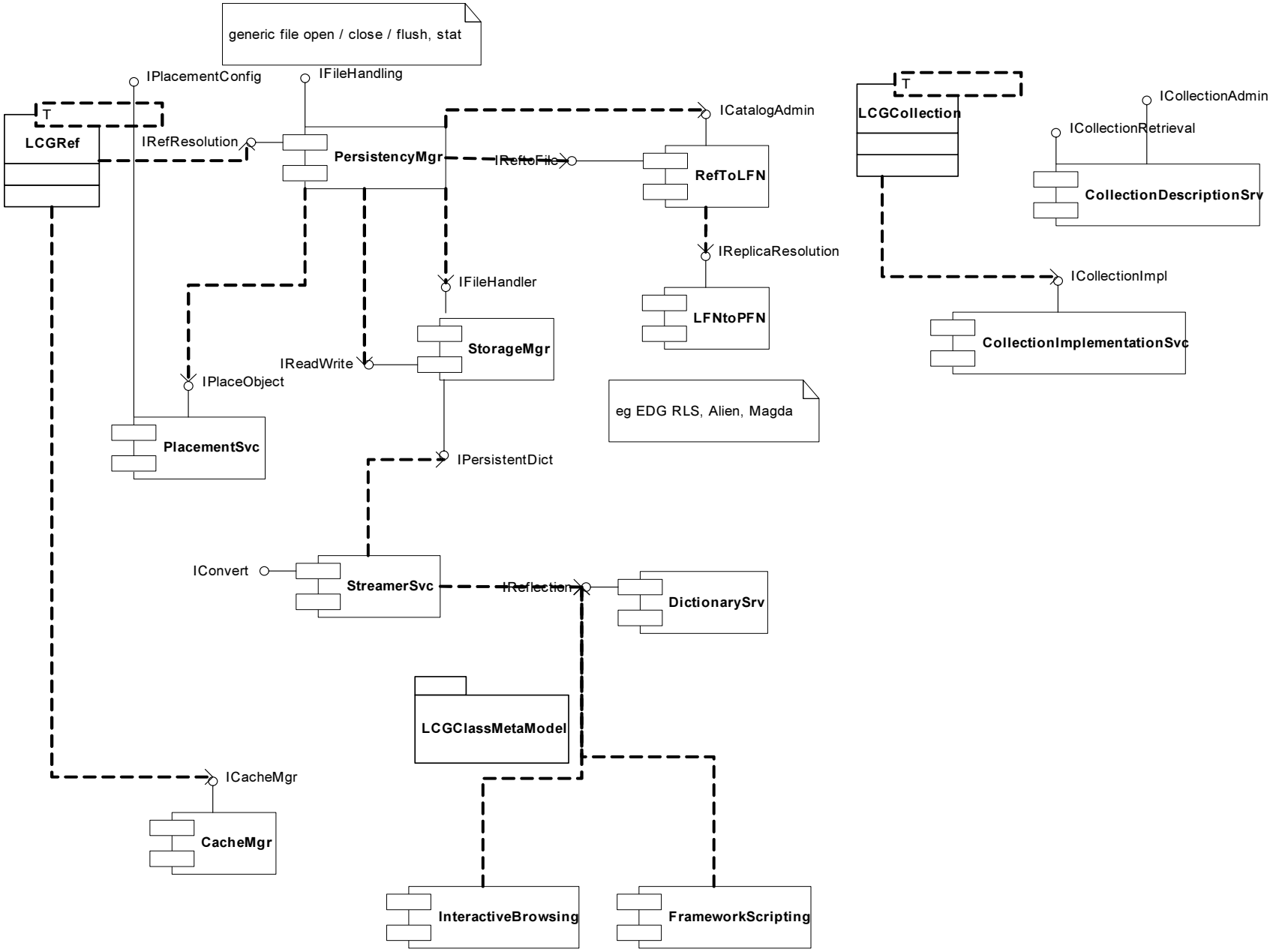
- RootI/O - Catalog integration (ALL)
  - Transparent Navigation (ATLAS/CMS/LHCb)
    - ALICE(maybe, but only in some places)
  - EDG (ALL), Alien (ALICE), Magda (ATLAS),
- Consistency between streaming data and meta-data (CMS/ATLAS)
  - At least at application defined checkpoints during a job
- Early separation of persistent and transient dictionary (ATLAS/LHCb)
  - see eg Pere's and Stefan's presentation
- Support for shallow (catalog-only) data copies (CMS)
  - formerly known as cloned Federations
- Support for deep (extracted part-of-object hierarchy) copies (CMS)
  - Both still needs more discussion to define concrete requirements

# RDBMS choice for initial Prototyping

- Some comments about my understanding
  - MySQL has been chosen on as implementation technology for the first project prototypes by the LCG Architect's Forum
  - Several people (including myself) have expressed their concern about using MySQL to build very large scale, reliable production environments
  - Still for the September timescale we a pragmatic choice to start with
- The question about the most suitable RDBMS back end will stay open until we real have production experience
  - Production requirements (scalability, concurrency, reliability) are not yet equal priority to all experiments
  - We can not assume only one technology
    - at one point in time
    - over the duration of the project
- We need to insulate against changes in the RDBMS as much as against changes in the streaming technology

# RDBMS interface

- Need RDBMS C++ binding - from a quick web survey
  - MySQL C++
    - Used for initial prototyping
    - free, but limited compiler support, bound to MySQL
  - Proposal: either replace asap, or accept for september release
- SQLAPI++
    - shareware - but essentially free
    - comes with full source code
    - multi platform(Win/Linux/Solaris), multi backend (10 db vendors)
- Root RDBMS interface
    - Two flavours: one based on ODBC/JDBC
    - More limited functionality, somewhat coupled to framework
    - Free, multi platform, multi backend (even though fewer DBs)


generic file open / close / flush, stat
IPlacementConfig
IFileHandling
T
LCGRef
IRefResolution
PersistencyMgr
ICatalogAdmin
IRefToFile
RefToLFN
IReplicaResolution
LFNtoPFN
IFileHandler
StorageMgr
IReadWrite
IPlaceObject
PlacementSvc
eg EDG RLS, Alien, Magda
IPersistentDict
IConvert
StreamerSvc
IReflection
DictionarySrv
LCGClassMetaModel
ICacheMgr
CacheMgr
InteractiveBrowsing
FrameworkScripting
T
LCGCollection
ICollectionAdmin
ICollectionRetrieval
CollectionDescriptionSrv
ICollectionImpl
CollectionImplementationSvc

# A few preliminary implementation choices…

- ANSI C++
  - incl. STD libs
- Develop on RH 7.2, gcc 2.95.2
  - test frequently on gcc 3.1
- Use C++ namespace
  - but maybe foresee to disable use of namespace via #define
- Use exceptions
  - Only consistent way to reliably trap problems eg during navigation
- One shared library per component
  - Provides well know factory method for component implementation
  - Sufficient to get started – but need still to agree on a real physical component model (larger scope than just persistency)

# Ref Interface and Implementation

- Assume smart-pointer based approach
- A few proposals
  - Refs are implemented as concrete class templates
    - Using a technology specific strategy rather than implementing an abstract interface
  - Start from an existing interface/implementation
    - Gaudi and Espresso Refs
  - Could use Refs not only for persistent user objects but also persistent framework objects
    - eg Files (via the File Catalog), Event Collections (via the Collection registry)
    - This would reduce the number of required interfaces in the system and simplify component reused
- May need subscriber-like interface to experiment specific cache managers
  - Register all refs which are created in a cache
  - Invalidate all refs once a cache is purged (eg end of event)

# Transparent Navigation & Root I/O TRef

- Current Root I/O implementation of TRef does
  - not open files
  - not provide information to uniquely identify which file needs to be opened
  - not bring any objects into memory
  - not provide information on how to find the destination object

- The TRef functionality is currently not sufficient to implement a StorageManager in the RTAG sense
  - "write object" is supposed to produce a "token" which can be used (eg in a different process) to directly read the same object again
  - read could be triggered from Root (eg using the EXEC comment in class definition)

# Transparent Navigation for Root I/O

- Several possibilities to extend the basic Root I/O services exist
  - LCG Persistency Framework maintains complete information about files and object location in its own Ref – see Markus Frank's talk
    - keeps track of file, tree and branch info
    - may need hooks into RootI/O to gather required information as data gets written
  - LCG Persistency Framework uses another mechanism to uniquely identify objects in Root files
    - eg store objects under a unique key
    - less Root I/O specific…
    - …but also giving up some of Root features (tree access?)
  - Root TLongRef / TGridFile
    - See Rene's talks tomorrow
- Navigation is an essential part of the RTAG model
  - we need to converge on a possible implementation soon!

# Checkpointing/Transactions

- Model for how to integrate several independent technologies to achieve application defined points of consistency (checkpoints)

- Proposal:
  Components which modify persistent state need to implement the ITransactionHandler/ITransactionContext interface
  - ITransactionContext *ctx = comp->createContext("context name");
  - comp->destroyContext(ctx);

- ITransactionContext interface provides:
  - bool start()  // activate context
  - bool prepare()  // check if commit would be possible so far
  - void commit()  // commit current state to disk now
  - void abort()  // (optional) undo all changes since last start
- Not full ACID transactions – but at least ACD is required

# How to attach “Meta Data”?

- Proposal: go for a minimal (but consistent) approach
  - allow to attach extensible list of named attributes
  - Scope/Container provides IAttributeList interface
    - eg files (scope: file catalog)
    - eg event collections (scope: collection registry)
    - eg individual events (scope: event collection)
  - queriable eg by SQL or higher level interface

- IAttributeList
  - Creation and deletion of attributes
    - scope->defineAttribute(“name”, type); // allow C++ basic types + string
    - scope->removeAttribute(“name”);
  - Defining and retrieving individual values
    - setAttribute(ref, “name”, value);
    - getAttribute(ref, “name”, &value);
  - Query
    - itr = scope->makeIterator(“a query string”); // or query predicate object?

# Summary

- Deployment model and focus of different experiments are different
  - Simplifies split of tasks ☺
  - Complicates agreement on priorities ☹
  - Some danger of ending up with a shopping bag of disjoint features rather than a common project
- Still in a phase of discovering more problems/requirements than solutions
  - we'll need to freeze (not the code but) the requirement list soon/now
  - transparent object navigation for Root I/O needs a real solution asap
    - otherwise we may have to delay the September release
- Prototyping work is starting
  - First order component breakdown underway
  - First tests of catalog prototypes look promising