Techniques for high-throughput, reliable transfer systems: break-down of PhEDEx design

Tim Barrass, University of Bristol, UK; Daniele Bonacorsi, INFN-CNAF, Italy;
Jose Hernández, CIEMAT, Spain; Jens Rehn, CERN, Switzerland;
Lassi Tuura, Northeastern University, USA; Yujun Wu, FNAL, USA

Abstract

Distributed data management at LHC scales is a staggering task, accompanied by equally challenging practical management issues with storage systems and wide-area networks. CMS data transfer management system, PhEDEx, is designed to handle this task with minimum operator effort, automating the workflows from large scale distribution of HEP experiment datasets down to reliable and scalable transfers of individual files over frequently unreliable infrastructure. PhEDEx has been designed and proven to scale beyond the current CMS needs. Few of the techniques we have used are novel, but rarely documented in HEP. We describe many of the techniques we have used to make the system robust and able to deliver high performance. On schema and data organisation we describe our use of hierarchical data organisation, separation of active and inactive data, and tuning the database for the data and access patterns. Regarding monitoring we describe our use of optimised queries, moving queries away from hot tables, and using multi-level performance histograms to precalculate partial aggregated results. Robustness applies to both detecting and recovering from local errors, and robustness in the distributed environment. We describe the coding patterns we use for error-resilient and selfhealing agents for the former, and the breakdown of handshakes in file transfer, routing files to destinations, and in managing site presence for the latter.

WHAT IS PHEDEX?

PhEDEx is the data placement and transfer system for the CMS experiment [1] at the Large Hadron Collider at CERN [2]. It manages continuous high-load data transfers from CERN to several dozen computing centres around the world; transfers among those centres; and also transfers for individual physicists for their private analyses.

Building reliable high-performance distributed systems is hard; fortunately much prior art exists. The PhEDEx project [3, 4] has sought to apply the best practices known to us. We share here the techniques we have found useful.

DISTRIBUTED WORKFLOWS

Each site participating in PhEDEx transfers runs a suite of persistent processes called agents [5]. Each agent performs a specific small step of the overall transfer workflow.

Most PhEDEx agents communicate indirectly via a blackboard [6], or tuple space[7], implemented as a high availability database. Similar structures have been used to coordinate the concurrent processing of large quantities of data ([8], and derivative projects e.g. SETI@Home) with simple single-step workflows. PhEDEx however uses the blackboard to coordinate quite sophisticated workflows involving many steps.

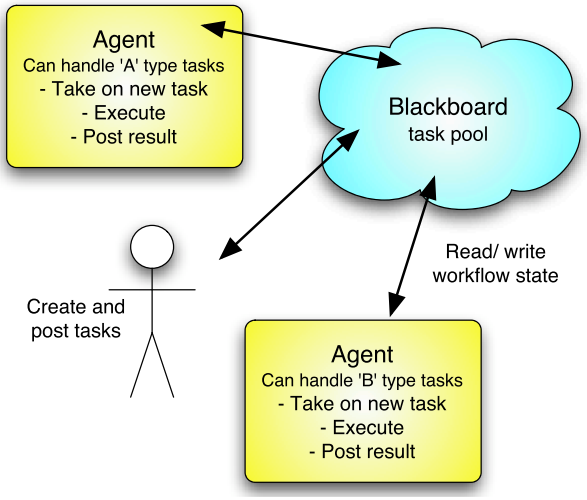

Figure 1: PhEDEx blackboard/tuple-space architecture.

Workflow tasks are created and posted on the blackboard, which acts as a task pool (Fig. 1). Imagine that task B can only be undertaken once task A has been completed. If the two tasks are handled by independent processes a handover of responsibility is required. In PhEDEx responsibility for tasks is generally pre- allocated to known instances of each agent, although they could be picked dynamically. When a task is complete, the agent posts status to the blackboard. Often this effectively creates a new task for another agent. The transfer workflow is defined by these state exchanges.

For complex local workflows some agents use the file system in a manner very similar to mail transport agents [9] to persist workflow state. We are currently investigating peer-to-peer information-smearing algorithms [10] to distribute partitions of the workflow state and limit PhEDEx reliance on a single central resource.

The PhEDEx system is primarily a driver of large-scale data distribution. It does not provide any low level tools for making transfers, interacting with local storage systems, or

cataloguing. Instead, well-defined points are provided for call-outs (via locally configured 'glue scripts' to whichever tools are locally preferred for these operations). Every attempt is made to make sure that local information stays local, to insulate the global system from local changes.

To limit the complexity of the system the software components are only indirectly dependent on each other– no component knows that any other component exists, or what they do. Agents are given increasing levels of autonomy to make decisions based on local conditions. To this end, agents are typically deployed at a site, rather than dealing with transfers remotely. That said, it is not yet possible to deploy and configure agents at every site, even though data can be created at Grid sites outside of the PhEDEx topology. To bring data created outside this topology to PhEDEx nodes we have an umbrella node that manages distant third-party transfers [11]. This sees use in CMS primarily to harvest Monte Carlo simulated data, that can be created on any Grid site that allows CMS access.

Robust agent design

PhEDEx agents do not maintain internal state, and can be stopped or started without ill consequences even after a system crash. Permanent workflow state is stored on the blackboard, and changes to this state are transaction-safe. Agents and sites are restricted to changing relevant partitions of the database using fine-grained database role grants, reducing damage from operational mishaps.

Each agent is responsible for one unreliable operation or workflow step, (e.g. file transfer, checking stager status or managing the overlay network partition for a site.) Each agent: finds pending work on the blackboard;picks and prioritises tasks, and executes them; and finally marks successful tasks complete, possibly indirectly assigning tasks to other agents in the process.

SQL operations, embedded verbatim in agent code, are used for all communication with the blackboard. No intermediate server tiers or encapsulated SQL code in separate libraries are used. Using a central high-availability Oracle database cluster with 24/7 support has been advantageous. The gains in system robustness, availability and flexibility have greatly outweighed the disadvantages, for example having to manage the redistribution and migration of client code on blackboard schema or SQL modifications.

Distributed handshakes

Tasks are joined to form a workflow using a handshake– a writing and reading of state information to and from the blackboard by a pair of agents. The definition of what state information is available at the start of a step, and what should be available at the end, is the basis for the design of agents, which act only on the appearance or change of state information.

This translates conveniently to inserting a row into a task table in the database. To take on a task an agent simply reads a row and acts on it. When a task has been successfully completed the row is updated, and potentially new rows are inserted. In complex workflows the state is check-pointed in the database, in which case processing continues from the last good check-point on agent restart, or when the agent finds itself idle, so that they recover from long-term skew of "lost" work.

At present tasks are implicitly assigned to specific agent instances at specific sites. However, forthcoming developments in some areas of PhEDEx mean that we are now creating a more dynamic environment, in which agents collaborate to provide services by claiming responsibility for tasks, whether when first advertised or when it becomes apparent that the original claimant cannot fulfill its obligation.

When two agents need to co-operate more closely, for example during a file transfer, we use a more fine-grained state machine to guide the evolution of each state row on the blackboard. This enables each party to operate on several entries in identical states. In certain such state machine exchanges the agents require the other party to actively refresh the state – if the other party is unable to regularly refresh the state, it's most likely also unable to do any useful work, and can be safely ignored.

DATABASE ACCESS

PhEDEx is written in object-oriented perl [12] and uses the DBI [13] Oracle [14] interface. Good advice is already widely available [15, 16] and will not be repeated here. Suffice to say the stated best practice guidelines are important to follow.

Robust common operations

We provide convenience functions for common database operations. While making the database programming easier, these functions allow us to handle problems in the distributed environment. In particular we detect stuck or bad database, and statement, handles and flag connections as unstable, enabling agents to recover and reconnect.

Maintain availability

Our procedures aim to keep the database available at all times, although it naturally some interventions are inevitable. We can remotely schedule agents to back off and to resume operations, allowing in-place tweaking without disturbing existing connections. Significant upgrades, still only requiring a shutdown of less than a day, are infrequent.

DEFENSIVE CODING

We explicitly assume every operation, however trivial, will fail. We provide safe operators for numerous tasks (e.g. writing temporary files and launching subprocess queues) so that the simplest operations are robust. We

check for internal errors to prevent the propagation of errors between parts of the system that should be independent.

Failure recovery tactics

Our failure recovery tactics are typically quite simple: on failure, we first log the issue, back out and retry after a cool-off period. To retry, we clear certain local state caches, and then rely on global system consistency to trigger the retry. The agents flag repeated problems as bad, alert an operator, and ignore them on subsequent iterations, unless the flag is removed.

Experience shows that many of the tools in our environment return misleading exit codes. We treat them with scepticism; each transfer is independently cross-checked for file existence and size. Checksums can also be checked, but this is a high-load task that is best handled in a parallel fashion, and not strictly coupled to the transfer operation.

Simplicity

We start with simple algorithms throughout. For example, transfer failure handling has evolved from a simple retry next time, through cooling-off processes to limited queue randomisation and prioritisation. Gradually subtler tactics are being implemented where necessary, making the system increasingly more autonomous [17]. For example, some of the more advanced agents detect pathological patterns and automatically throttle themselves. In these cases pathological patterns are identified by monitoring local rates – for example, the number of successful transfers per hour – and corrective behaviour is triggered when these rates pass high or low cutoff thresholds. Such behaviour is basically a set of higher-order corrections, suited to systems with a stable, reliable underlying fabric where response is linear.

ALGORITHMS

The routing overlay

An overlay network is used to describe a topology in which nodes represent storage resources, independent of the underlying network fabric [18]. This allows PhEDEx to cache data at regional centres for distribution to smaller sites.

The overlay network is maintained by a quite static shortest-path algorithm, with shortest paths calculated using Dijkstra's algorithm [19]. A neighbour-list containing static link-weight information is stored on the blackboard . Routing agents act at and on behalf of each node in the network, and use Dijkstra's algorithm to dynamically refresh a minimum spanning tree from their node to each other node in the network. This minimum spanning tree information is then stored in a routing table on the blackboard holding source, destination, gateway, hops information. We describe this as quite static as the topology is not subject to high churn, and the adjacency list is taken as the authoritative source of link state information rather than a dynamic exchange of state information between components. Thus the routing algorithm is effectively just a means of automating the changes in the routing table necessary when nodes leave or join the topology.

Routing files to destinations

The PhEDEx topology is a weighted, generally not fully-connected graph. Files may need to be temporarily replicated to a regional centre before final replication to a destination, to better manage the load on the central facility.

A file routing agent acts on behalf of each node in the network, and is responsible for triggering the set of replications that glue a transfer from source to destination together. The file router uses the routing table to determine shortest paths from source to destination, and triggers the first transfer in the chain by inserting a row into a transfer state table giving source and destination information. When that transfer is marked complete it reevaulates the closest replica for each file and triggers the next transfer in the chain.

Robust transfer handshake

PhEDEx developed during times of unrest in underlying storage and transfer technology. Much functionality desired of storage systems – stage-on-demand, intelligent grouping of stage requests, sophisticated space management – is still not in evidence. The PhEDEx transfer handshake/workflow is therefore sophisticated and incorporates much of the functionality desired of underlying systems (see Fig. 2). Note that the transfer operation is a sub-workflow, with pre-delete, bypass, transfer, verify and publish steps. The export step replicates functionality expected of underlying storage systems. The state transitions on the blackboard define the handovers of responsibility between distributed agents that together comprise the overall workflow.

PERFORMANCE

Basic schema design and tuning

Direct access to database resources and gurus is essential for database performance. We avoid using generic services that add an extra layer of indirection and processing over those provided by the database in favour of known, manual optimisations.

Client or database processing?

We've frequently changed the definition of a problem or an algorithm such that it can be executed as a small number of SQL statements rather than client-side logic. We use no stored procedures, and few triggers. We also divide responsibility between client and database engine intelligently –

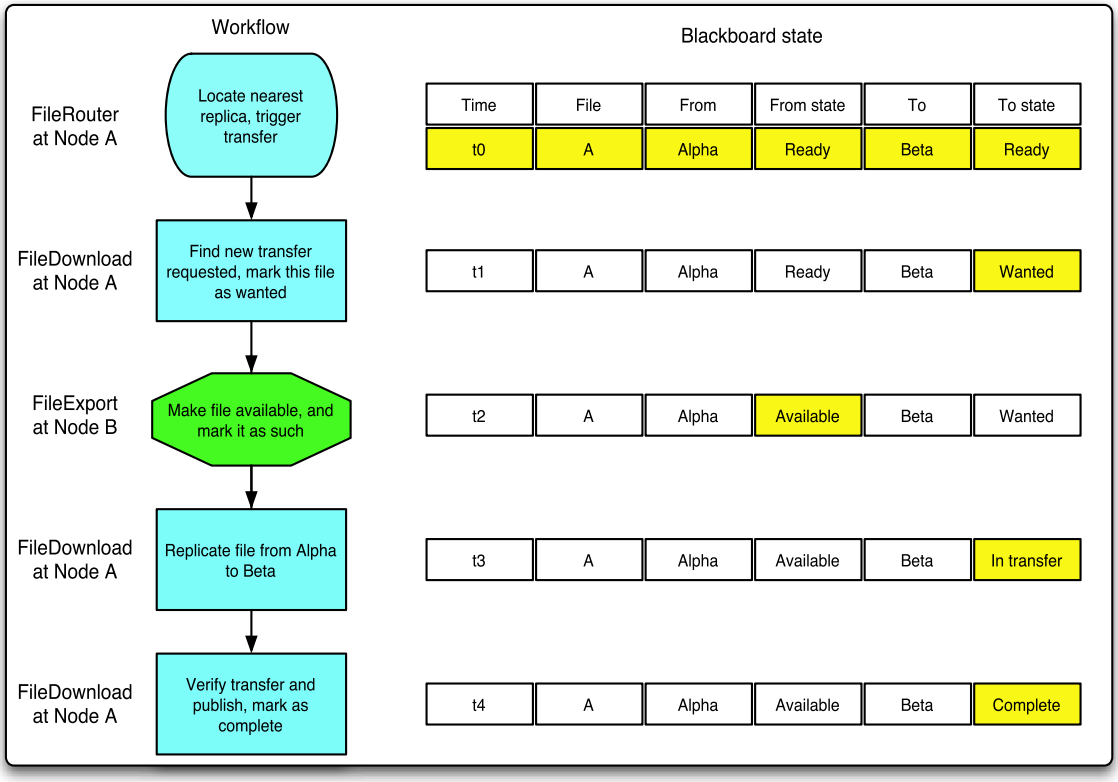

Figure 2: PhEDEx workflow state changes during a transfer handshake, with three agents involved.

Fundamental objects given primary keys	Constraints enforce data consistency at schema level, avoid error checking at client level
Foreign keys indexed, used to alias fundamental objects	
Suitable degree of denormalisation	Reduce number of joins required on hot tables
'Good' column ordering	Improves Oracle storage performance
Use bind variables in SQL	Reduce latency in query processing
Cost based query optimisation with regularly updated stats	Used to improve schema by identifying hot data and high-load queries, and inform storage decisions for administrators
Analyse important queries with oracle tools (explain plan..) and Tom Kyte's tkprof [13]	
Row movement enabled for busy tables	Improves concurrency for busy operations that require strict data consistency, and storage compaction for busy tables
Increase 'Initrans'	

Figure 3: Database design and access principles, and their benefits.

pulling data over WAN links is unwise, a big join handled by the database engine is typically more efficient.

Exploiting data relationships

All data either has internal relationships allowing it to be grouped hierarchically, or can have arbitrary relationships imposed upon it. The CMS experiment typically groups files by demand for certain physics phenomena expressed in the data, so a file group is naturally the unit at which nodes subscribe to the data. PhEDEx then divides whole of the data into "streams", those further into "blocks", which contain files. Operating on streams and even on blocks is extremely fast, since the tables describing them are compact.

Moreover, by operating on only "active" blocks– by expanding file information into hot tables only when they are actively being transferred– we massively reduce the number of operations that touch file-level data at any time (see Fig. 4).

Smart caching in stateless agents

Although the agents are stateless with regard to critical global workflow state, some agents build caches to improve throughput. Caches tag data with a validity of some hours after which the record is purged and reloaded from the database on next use. This makes the agents self-healing. Caches are used only when the agent is the sole authoritative source for the data so it only needs to shield itself against direct database changes, not changes by other agents. One example is the agent managing the stage space

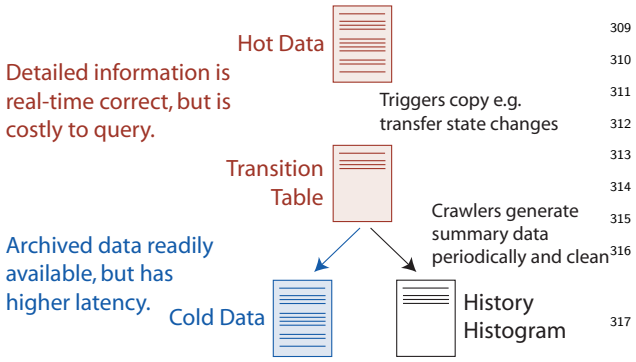

Figure 4: Hot data is kept as compact as possible for efficient access; cold data is effectively archived to make sure it's not processed unnecessarily.

of a given tape system at a site, which caches local filesystem structure information to enhance the efficiency with which it meets future stage-information requests.

High performance monitoring

Web monitoring pages showing current live state [20] can inhibit the performance of sizeable and active databases, let alone presenting historical plots and statistics.

- Auxiliary monitoring tables are filled by background processes at regular intervals; web pages query these tables.
- Update frequency depends on source data; most just captures overall state with a large 'select ... group by ...'
- Compromise between query cost and user requirements for observation– varies between 40 seconds and 15 minutes.
- Visible updates guaranteed every 4-5 minutes using multiple layers of aggregation, independent of database load.
- Fine-grained partial histograms with 5-10 minute bins represent historical data (e.g. see RRD [21] and MonALISA [22].)
- Histograms updated on movement from hot to cold tables via holding tables.
- Agents can access this historical information to help adjust their own behaviour, bringing us closer to developing an autonomic system [17]. Adding dynamic behaviour is, however, complex.

SUMMARY

PhEDEx provides a robust and reliable infrastructure for driving large-scale dataset transfers. To make it reliable and robust, even when overlying an unstable fabric, it uses well-established principles of asynchronous systems design, some of which we've summarised here. The use of these principles enables PhEDEx to be effective in a production environment, in which it currently manages nearly 0.5PB of globally-distributed data, across heterogeneous storage systems. By continuing to incorporate existing and cutting-edge experience in asynchronous distributed systems design we are confident that PhEDEx will be able scale to CMS' required data volumes of order 10 PetaBytes a year.

REFERENCES

[1] CMS Collaboration, "The Compact Muon Solenoid Computing Technical Proposal", CERN/LHCC 1996-045, 1996.

[2] European Centre for Nuclear Research (CERN), http://www.cern.ch.

[3] Rehn et al, "PhEDEx high-throughput data transfer management system", CHEP06, Mumbai, 2006.

[4] Barrass et al, "Software agents in data and workflow management", CHEP04, Interlaken, 2004.

[5] The Foundation for Intelligent Physical Agents, http://www.fipa.org.

[6] Corkill, "Collaborating software: Blackboard and multi-agent systems and the future", Proceedings of the International Lisp Conference, New York, 2003

[7] Gelernter, "Mirrorworlds", Oxford University Press, 1992.

[8] Anderson, Kerpela and Watson, "High-performance task distribution for volunteer computing", First IEEE International Conference on e-Science and Grid Technologies, 2005.

[9] Venema, Postfix, http://www.postfix.org.

[10] Maymounkov and Mazieres, "Kademlia: A peer-to-peer information system based on the XOR metric", 2nd International Workshop on Peer-to-Peer Systems, 2003.

[11] Caballero, Garcia-Abia and Hernández, "Monte Carlo production on the LHC Computing Grid", CHEP06, Mumbai, 2006.

[12] Cozens, Object Oriented Perl, http://www.perl.com/pub/a/2001/11/07/ooperl.html.

[13] Perl DataBase Interface (Perl DBI), http://dbi.perl.org.

[14] Oracle, http://otn.oracle.com.

[15] Bunce, "Advanced DBI Tutorial", http://search.cpan.org/ timb/DBI_AdvancedTalk.

[16] Kyte, "Effective Oracle by Design", Osborne ORACLE Press series.

[17] IBM, "Autonomic Computing", http://www.research.ibm.com/autonomic.

[18] Wikipedia, "Overlay Network", http://en.wikipedia.org..

[19] Cormen, Leiserson, Rivest and Stein, "Introduction to algorithms", 2nd Ed, MIT Press and McGraw-Hill, 2001.

[20] PhEDEx live monitoring, http://cms-project-phedex.web.cern.ch/cms-project-phedex/cgi-bin/browser.

[21] RRD, http://people.ee.ethz.ch/ oetiker/webtools/rrdtool.

[22] MonALISA, http://monalisa.cacr.caltech.edu/monalisa.htm.