CoralServer in ATLAS

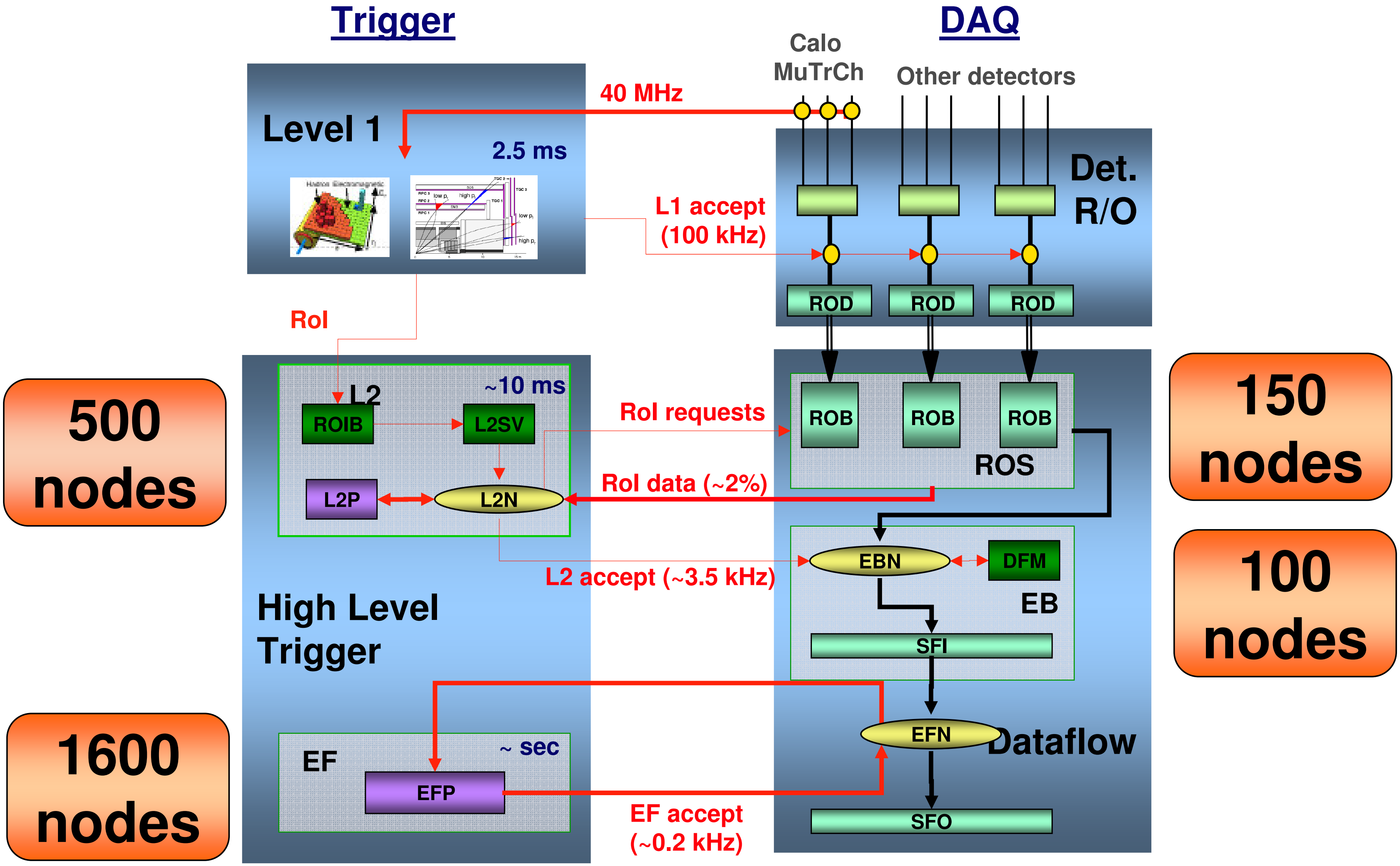
Applications Area Meeting
2009-11-04

Andy Salnikov
for SLAC TDAQ group

Outline

- ATLAS HLT farm
- HLT configuration problem
- Using database proxies for configuration
  - MySQL proxy
  - MySQL-to-ORACLE bridge
  - CORAL Server / CORAL proxy

DAQ/HLT Architecture (2006)



HLT Farm at Point1

- TDR assumptions (2003):
  - 500 LVL2 “processors”
  - 1600 EF “processors”
  - (assuming 8GHz clock speed)
  - 20+80 racks
- Current setup
  - 27 “XPU” racks
  - 31 nodes per rack
  - dual quad-core CPU
  - ~820 nodes, ~6500 processes
  - 10 more EF racks early next year
- Final setup
  - depends on first beam experience
  - farm will certainly grow

HLT Configuration

- Every HLT application (LVL2/EF) has to read configuration data to be able to process event data
  - Trigger configuration: trigger menus and prescales
  - Geometry data, complete detector description
  - Conditions data: calibrations, alignment, cabling maps, etc.
- Cannot spend much time on reading these
  - some configurations are loaded during stable beams
- The configuration data comes from several database instances
  - ATLAS C++ code uses CORAL library for database access
  - MySQL, SQLite: used during commissioning in 2006-2007
  - production will use ORACLE exclusively

Configuration Problem

- HLT configuration moves a lot of data
  - around 2000 nodes, up to 8 processes per node, tens to hundred MB of configuration data

    2000 × 8 × 10MB = 160GB

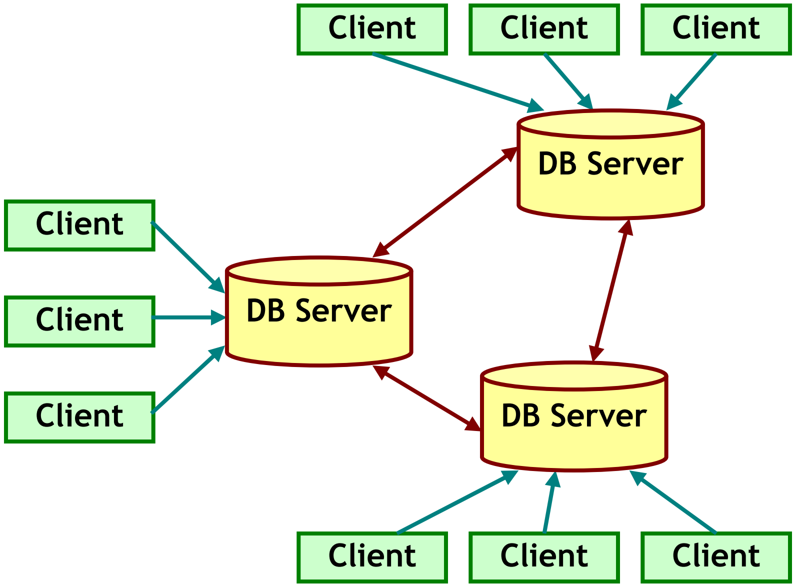
    (~30 minutes over 1Gbps connection)
  - all clients request configuration data from database at the same instant
  - single server cannot handle such load
- Positive points:
  - all clients get identical data from database (one set for LVL2 and wider set for EF)
  - database server needs to ship only single copy of the data O(10MB) (fraction of a second over 1Gbps)

Approaches

- Have to reduce both the number of connections from clients to server and the volume of the data
- Simple solution exists
  - increase number of servers to reduce number of connections
  - bring servers “closer” to clients to reduce network traffic
- Servers can be either
  - “real” servers (DB clustering), or
  - specialized “proxy” servers

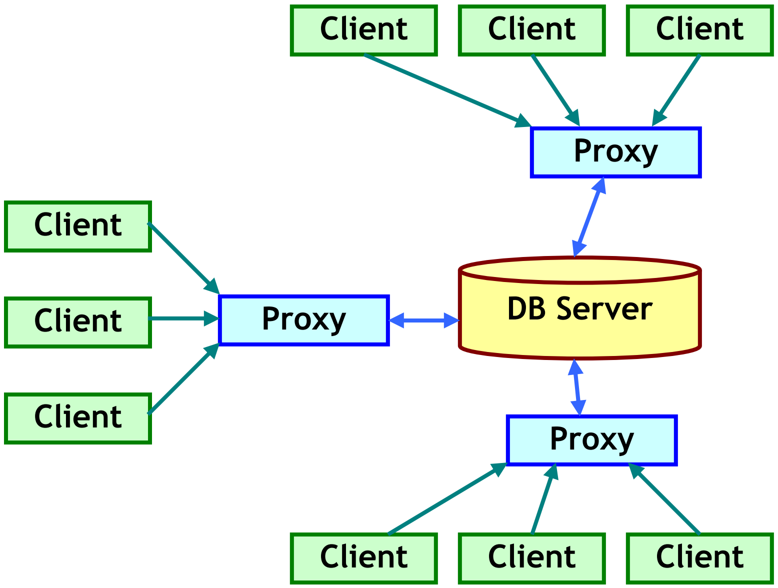
Clustering Approach

- Cluster consists of more than one tightly-connected servers
- Client chooses one (usually less loaded) server
- Cluster servers need very special and expensive hardware
- High management cost
- Solves different problems from what we need



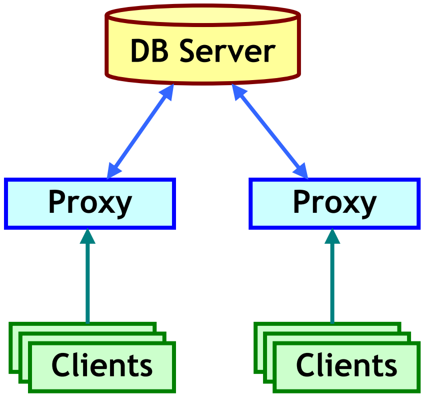
Proxy Approach

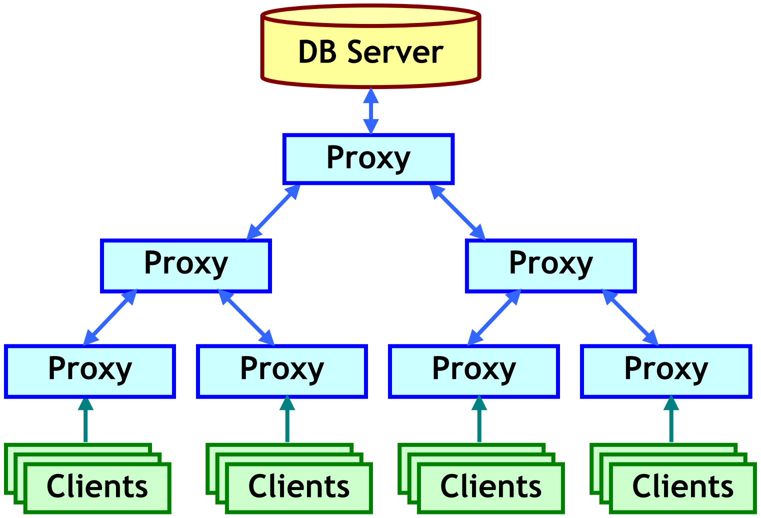
- One central database server
- Several proxy servers connect to database server
- Proxies cache the results returned by server, reduce repeated queries
- Clients connect to a "closest" proxy server



Proxy – Design

- Two keywords — *caching* and *multiplexing*
  - caching eliminates duplicate queries going to the server
  - multiplexing reduces number of connections from clients to server
- Should be possible to build hierarchies of the proxies
  - essential for scalability beyond several hundred clients

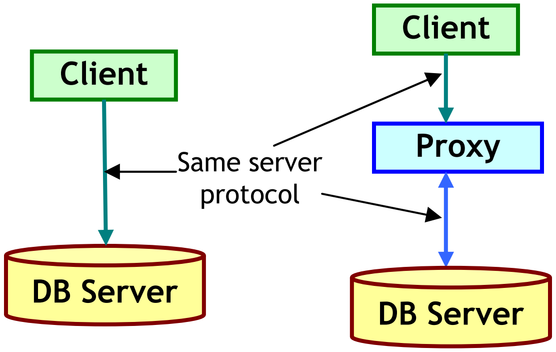


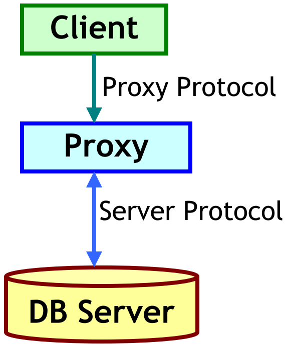


Proxy – Proxy Transparency

Two possible types of proxies:

- *Transparent proxy*
  - does not need any modifications on the client side (except possibly configuration such as host name or port number)
  - big benefit, client code is not touched, do not need debugging on client side
- *Non-transparent proxy*
  - client has to talk different “proxy language”, new code has to be added on client side (debugged, tested, etc.)
  - can be more optimal w.r.t. caching or multiplexing



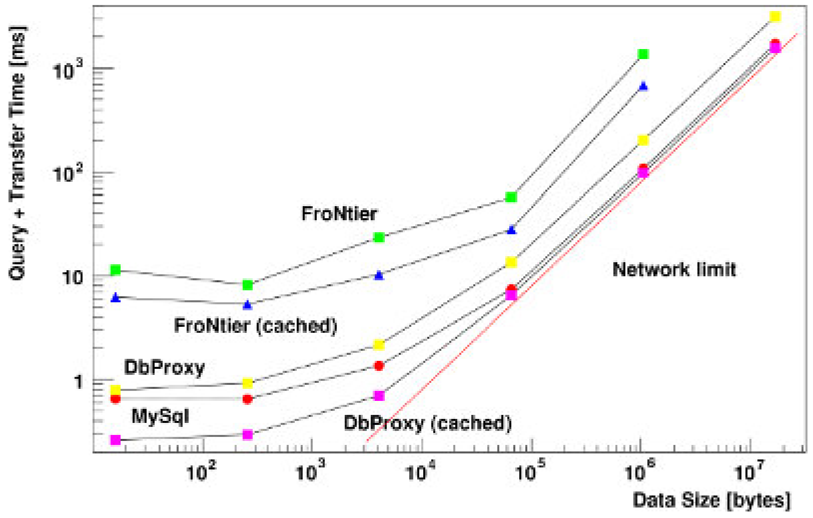


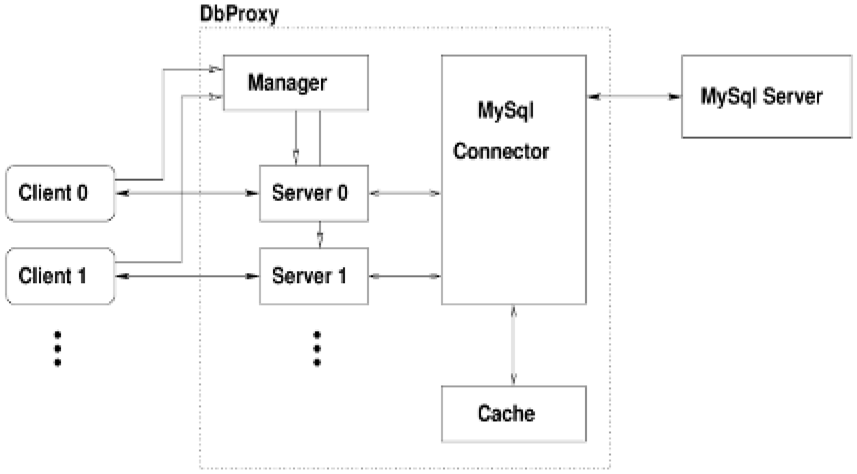
DbProxy – MySQL Implementation

- MySQL was used for HLT commissioning
- MySQL protocol is open, easy to build transparent proxy which looks exactly like MySQL server
- Some limitation though, MySQL protocol was not designed to support multiplexing
  - SQL requests have to be self-contained, no relying on external context (such as “`USE DATABASE`”)
  - special care needed, even small modifications to the CORAL client library need to be validated

DbProxy – MySQL Implementation

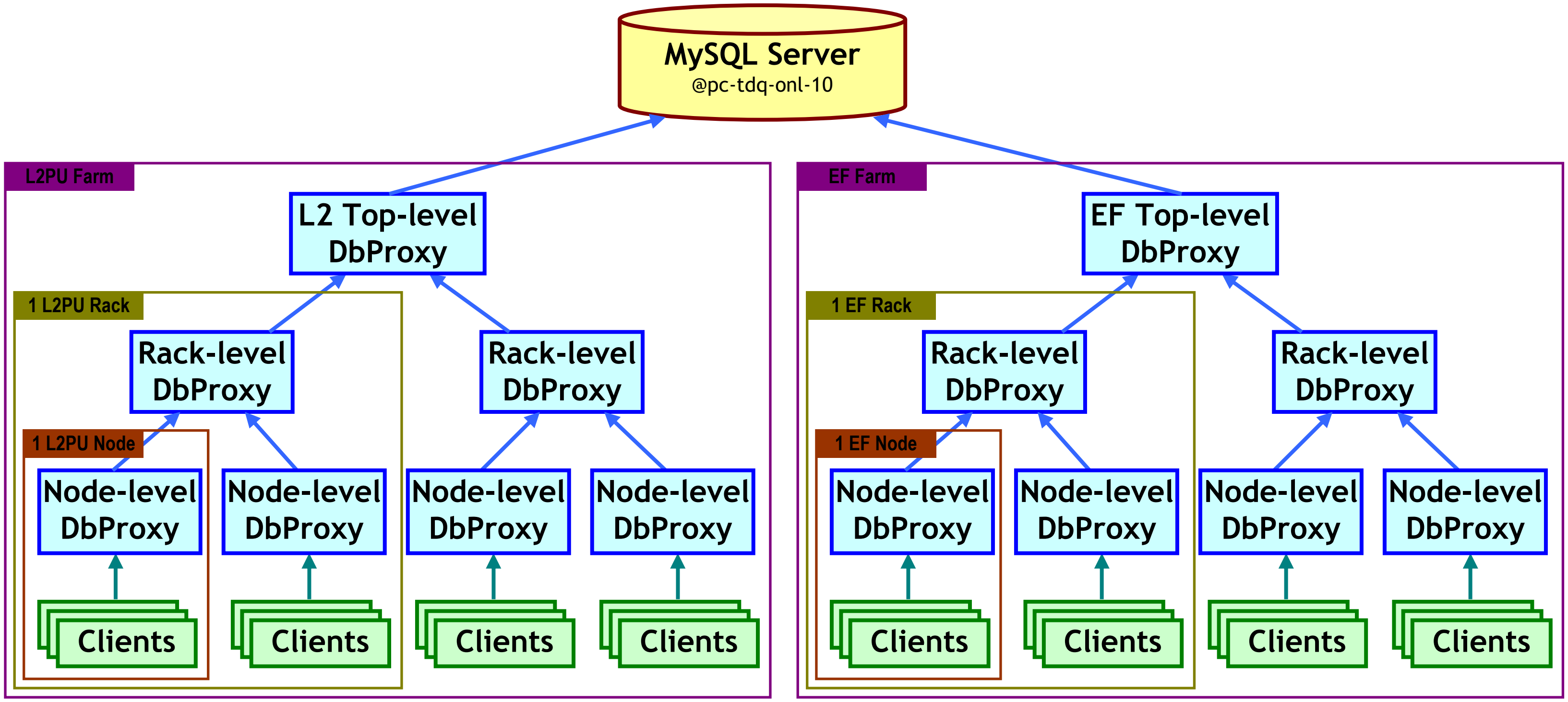
- First proxy created at SLAC
- Extensive studies with MySQL and Frontier during 2006 by Amedeo Perazzo
- Initial design and implementation by Amedeo
- Successfully tested at Point1 in the course of several Technical Runs during 2007
- Essential part of the TDAQ system (until CoralServer was deployed)
- Even at scale of 5 racks it was not possible to configure partition without proxy





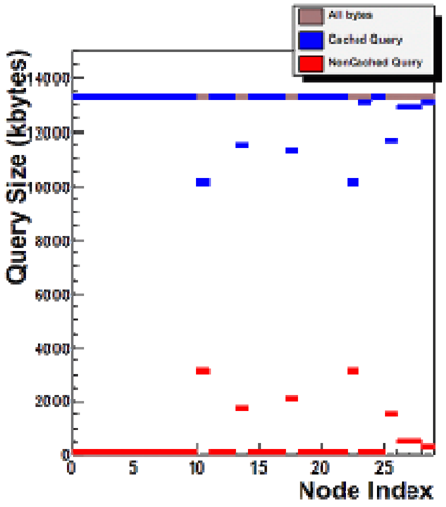
DbProxy Tree at Point1

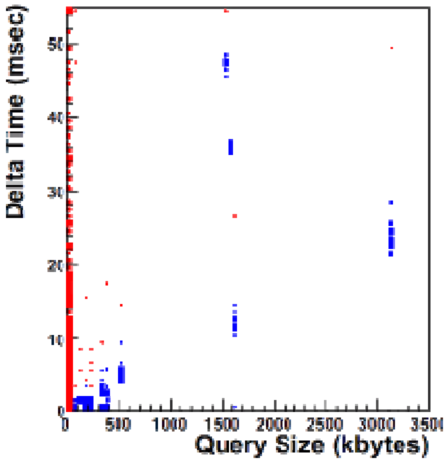
- Three-layer proxy setup during the Technical Runs at Point1
  - Node-level proxy serves up to 8 L2PU/EF processes on the same node
  - Rack-level proxy serves all node-level proxies in the same rack
  - Top-level proxy serves all rack-level proxies in the L2PU or EF segment
- MySQL server has only two clients

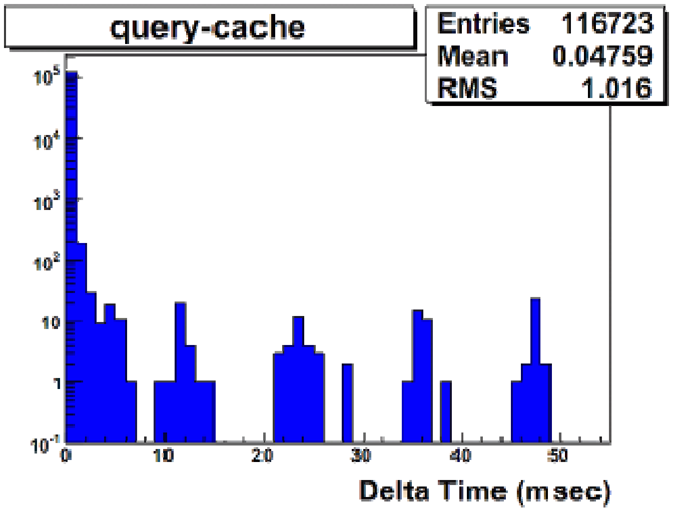


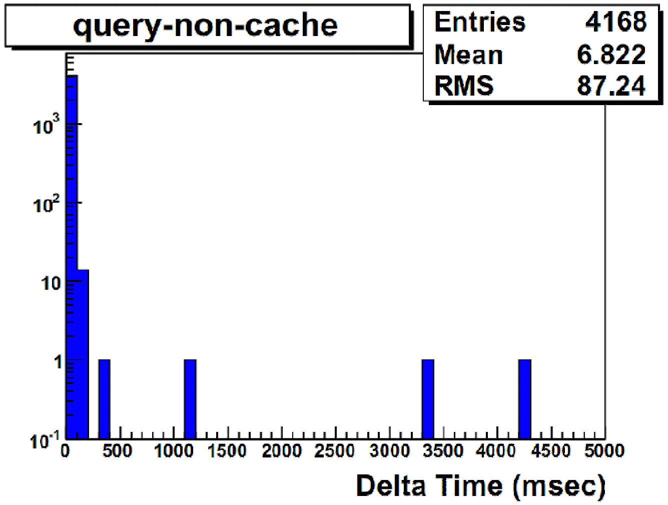
DbProxy as Monitoring Tool

- Performance plots regularly produced from the proxy logs
- Wealth of useful information about database access patterns, response time, and cache behavior

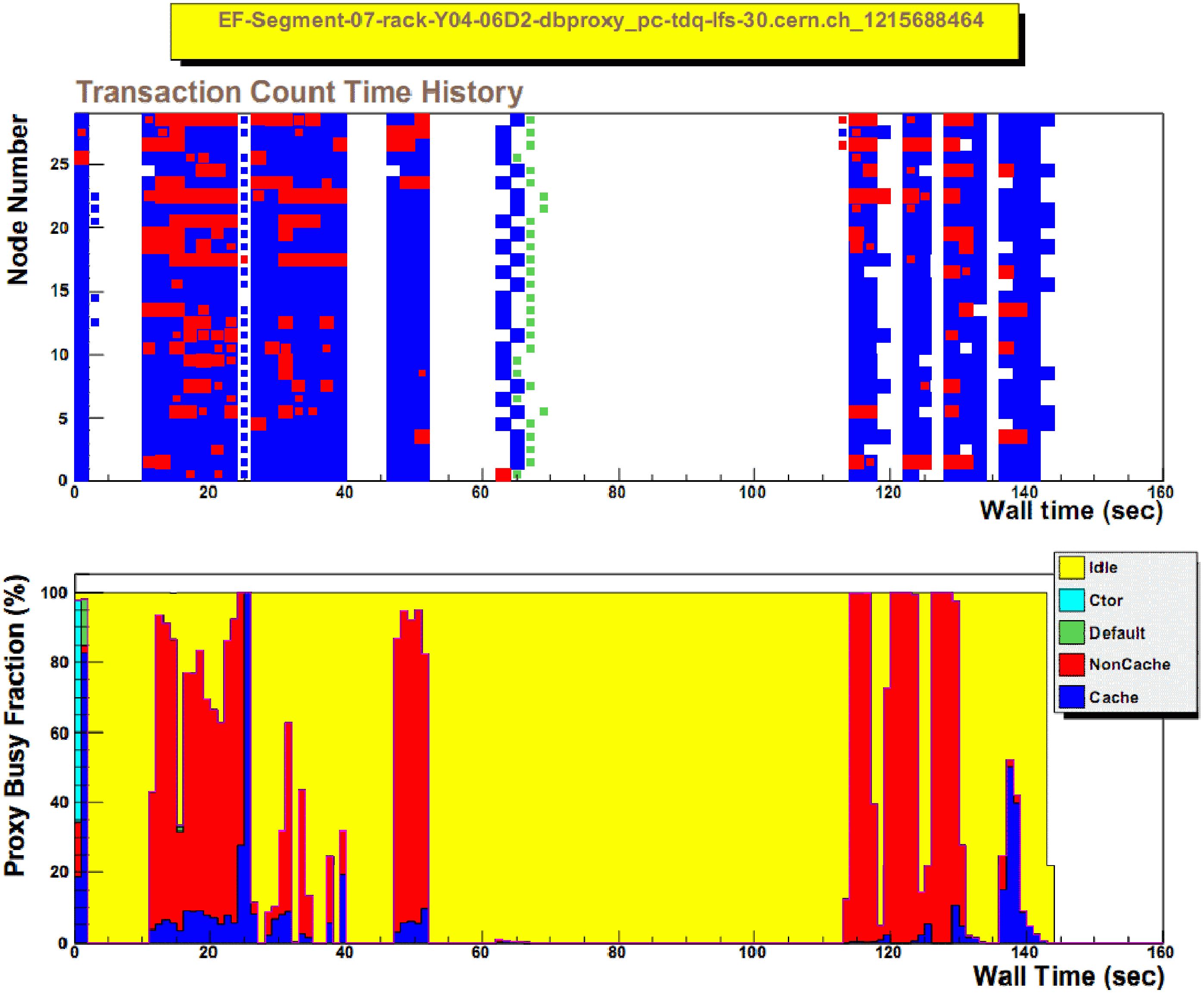








DbProxy as Monitoring Tool



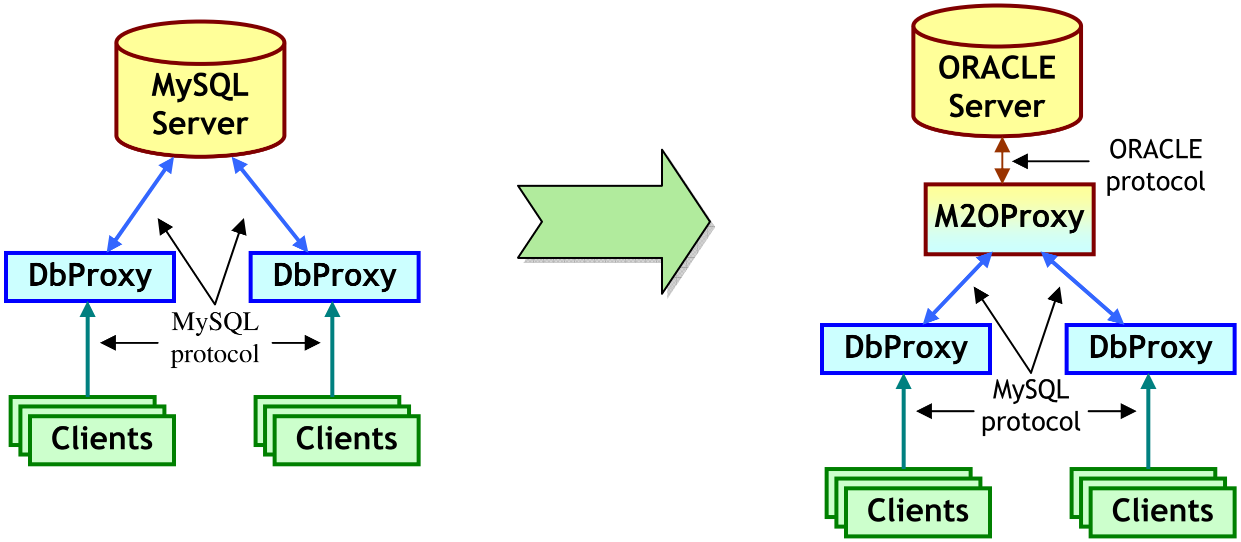
DbProxy and ORACLE

- For production running need proxy for ORACLE server, but transparent proxy for ORACLE protocol is not feasible, protocol is closed and proprietary
- Two non-transparent options exist:
  - Keep MySQL protocol for proxy tree, translate to ORACLE at topmost level
    - relatively easy to implement for subset of SQL used by CORAL
    - difficult to support in the long term, have to watch all CORAL changes for potential incompatibilities in SQL requests
    - have all drawbacks of MySQL protocol
  - Better option – CoralServer
    - new CORAL-specific protocol optimized for caching/multiplexing
    - translated to ORACLE or MySQL via existing CORAL plug-ins

MySQL-to-ORACLE bridging

- ATLAS needed *short-term* solution until CoralServer is fully functional
- Development parallel with CoralServer
- New proxy server which speaks MySQL wire-level protocol on one end and ORACLE on another
- Depends on several CORAL features
  - MySQL ANSI mode, most queries follow SQL standard and can be sent to ORACLE without rewriting
  - Type conversion between ORACLE and MySQL is done after the rules used by CORAL plug-ins for ORACLE and MySQL
- Few MySQL-specific queries need rewriting

M2OProxy



- M2OProxy is a drop-in replacement for MySQL server
- Mostly transparent, except for schema and user names
  - simple change to CORAL configuration files
- Essential part of ATLAS TDAQ since 2008 (until CoralServer deployed)

CORAL Server

- Non-transparent proxy for ORACLE access
- LHC offline world has its own potential uses for non-transparent proxy, main considerations are security and large number of clients
- Project “CORAL Server”:
  - CORAL team: Dirk Duellmann, Alexander Kalkhof, Zsolt Molnar, Andrea Valassi (CERN/IT), Martin Wache (U. of Mainz/Atlas)
  - SLAC team: Rainer Bartoldus, Andy Salnikov

CORAL Server – Main Components

- CORAL plug-in [CORAL team]
  - client-side plug-in library which talks new CORAL protocol
- CORAL server [CORAL team]
  - standalone server application which understands new CORAL protocol and translates it into calls to CORAL API
  - uses existing ORACLE or MySQL plug-ins to communicate to real database server
- DbProxy (CoralServerProxy) [SLAC]
  - complete re-write of the current DbProxy which understands new CORAL protocol
  - does not need to understand all details of CORAL protocol, only small part sufficient for caching and multiplexing

Caching in CORAL Proxy

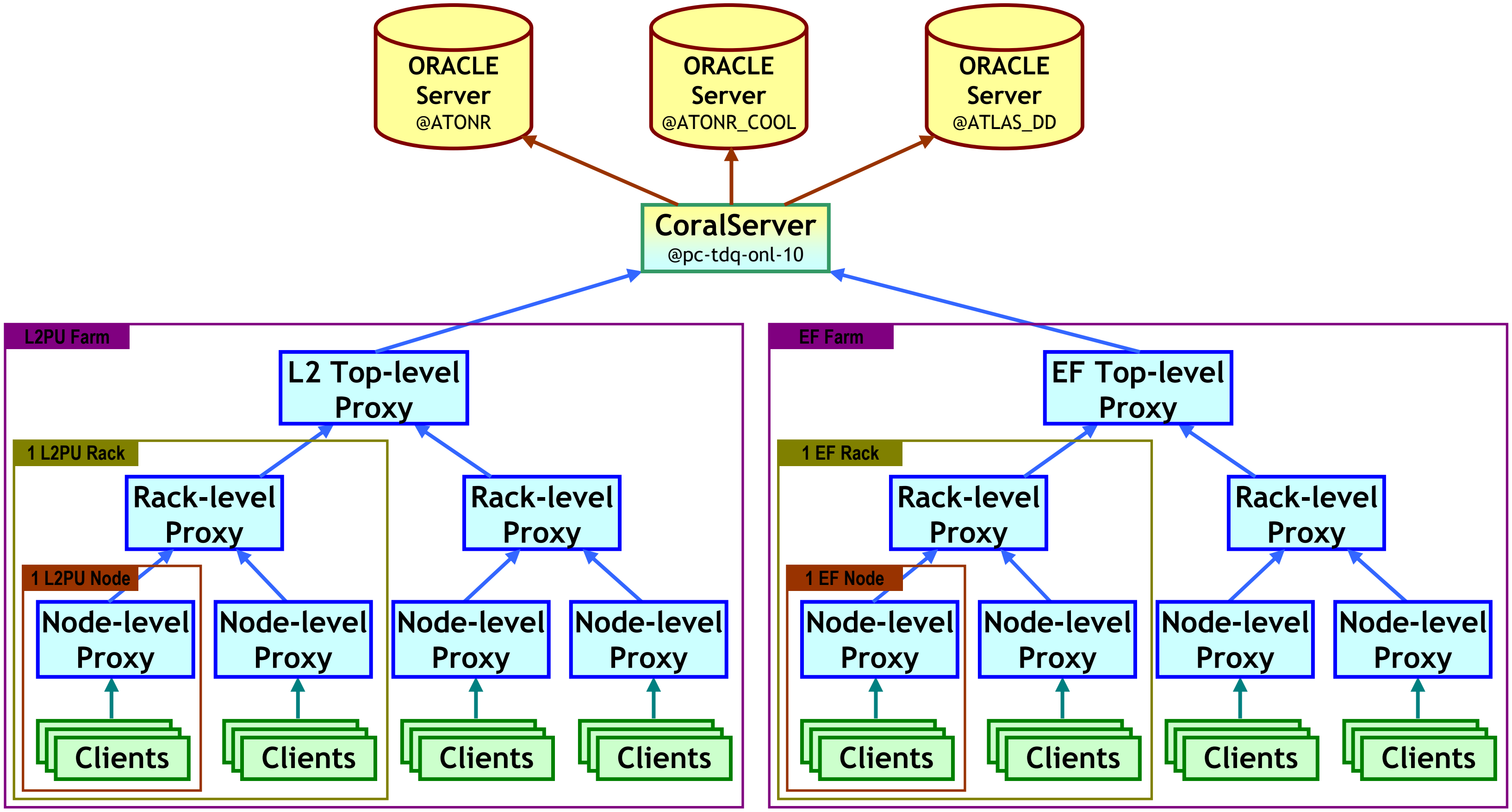
- Series of meetings to define transport-layer protocol for CORAL server
- New protocol is very flexible w.r.t. caching and multiplexing
  - All three components can make decision about caching of particular request
  - Multiplexing is in a core part of the protocol
- Right mixture of features, optimal for solving ATLAS HLT configuration problems

CORAL Server Deployment

- Tests with CoralServer were performed on SLAC TDAQ farm during whole development progress
  - stability, crashes, memory leaks, performance
  - tests were done in real HLT environment
- Learned also how to run ATLAS HLT client application standalone
- Pre-deployment tests were done at pre-series farm
- Thanks to all these tests the deployment was very smooth
  - One issue with transaction mode quickly resolved
  - Maybe the least problematic deployment at Point1 in ATLAS history

Proxy Tree with CoralServer

- Top-level proxies connect to the CoralServer
- CoralServer gets data from three Oracle Servers



Current Status

- Basic set of features is enough to run ATLAS HLT
  - Performance is adequate at current scale, on par with M2OProxy
  - Stability is OK, no problems so far
  - Simplified authentication handling
  - We are still learning
- We need tools to monitor and understand how our system behaves (both CoralServer and ATLAS HLT)
  - Need monitoring built-in into CoralServer and Proxy (already under development)
  - Need tools to watch/analyze monitoring data, integrated into TDAQ
- There is interest from other fronts
  - Using proxy for writing into database

Last Slide

- Proxy servers provide efficient and scalable solution for the HLT configuration problem
- MySQL DbProxy, MySQL-to-ORACLE bridging proxy
  - were the integral part of online system
  - already a history
- For ORACLE access CoralServer is a natural solution
  - deployed few weeks ago
  - running smoothly
  - ATLAS HLT would benefit from further development (monitoring)
  - SLAC team plans to stay involved into development
- Many thanks to CORAL team for such significant effort