

DB R&D Activity Report

STAR Online API/infrastructure: current status

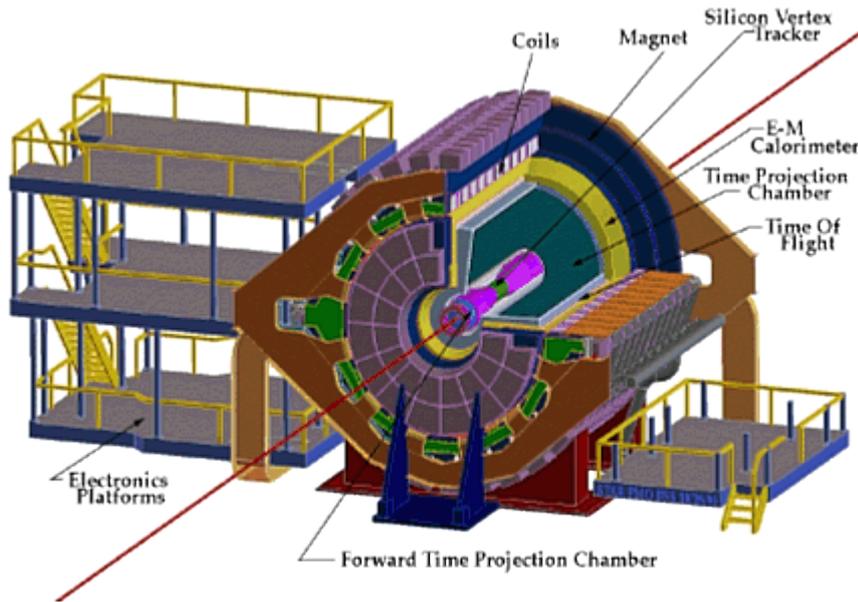
Dmitry Arkhipkin

May 4th 2010

Outline

- Overview
- Online API: intro
 - What features we wanted to have
 - What we have now: cons and pro
 - Proposed solution
- Progress report
 - Primary Components
 - Messaging: AMQP
 - Serialization: Google Protocol Buffers
 - “Proof-of-Concept” implementation
 - daemons: EPICS input, db storage (MySQL)
 - application: monitoring GUI
- Project timeline
- Summary

What kind of data we are dealing with?

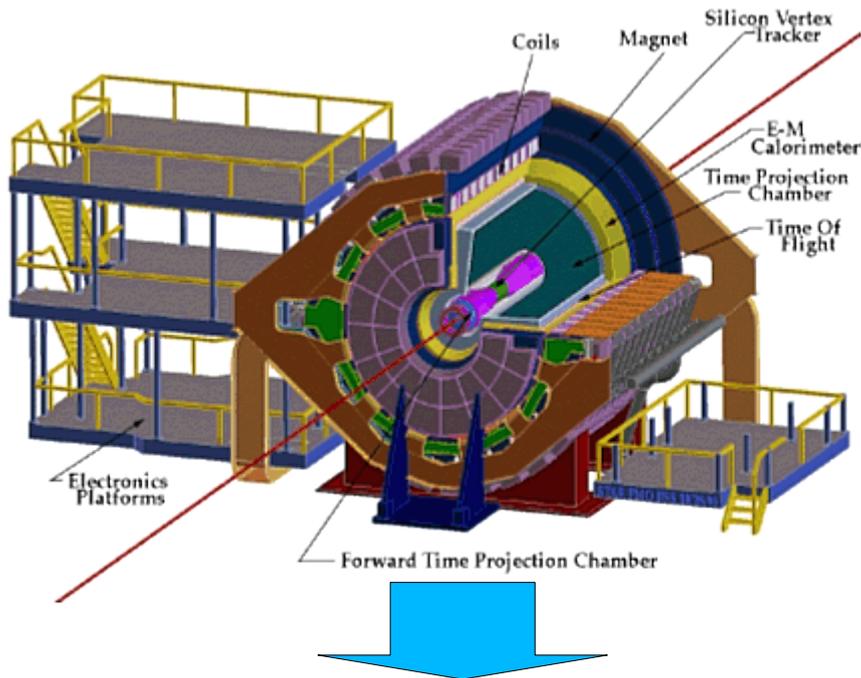


- **Machine state :**
- Beam information
 - species, energy, intensity etc.
- Scalers
- **Experiment state :**
- Scalers
 - beam signal vs. background
 - trigger counts
- **RTS/DAQ/Trigger**
 - runs, file locations, events etc.
- **Subsystem states :**
 - on/off
 - power supplies :
 - voltages, currents
- **Environment :**
 - temperature, humidity

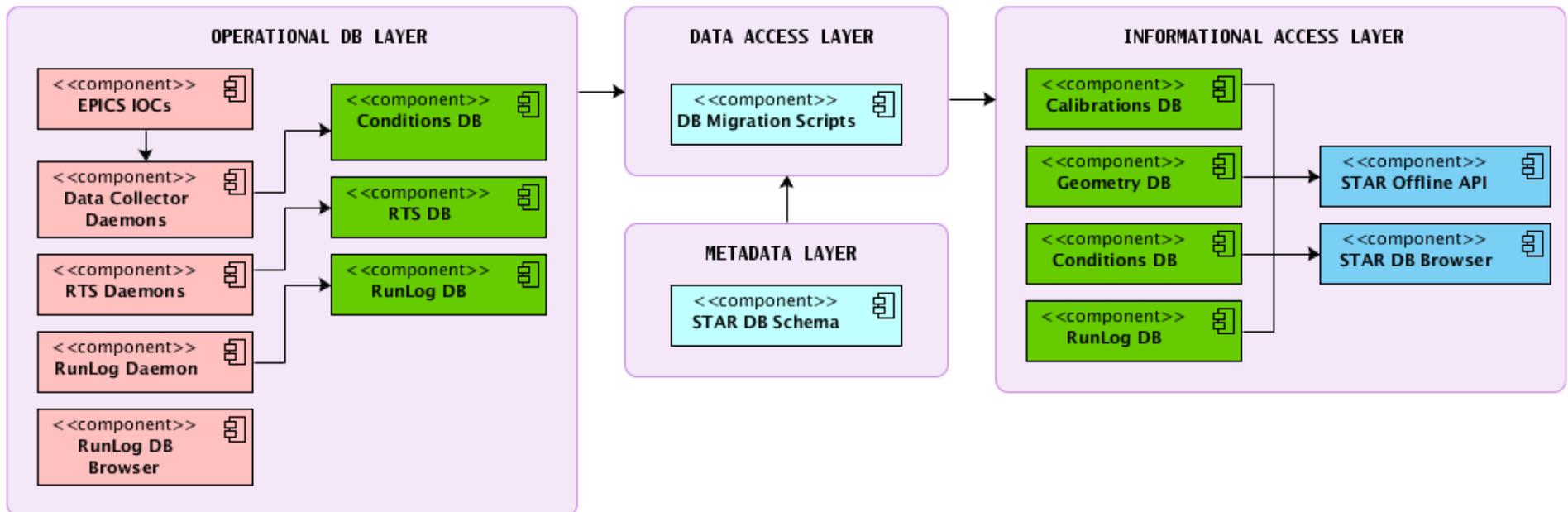
Basic requirements :

- Capable of reading over 2000 channels simultaneously, new streams added as STAR grows.
- Will reach ~5k channels in two years from now;
- Read frequency varies from “read once per 5 minutes” to “read as fast as it is technically possible”;
- Fault-tolerant setup: failure to read channels from group A should not affect reading channels from group B;
- Automatic data provider service (re)discovery;
- Reliable messaging: data transfer errors/timeouts should be handled automatically;

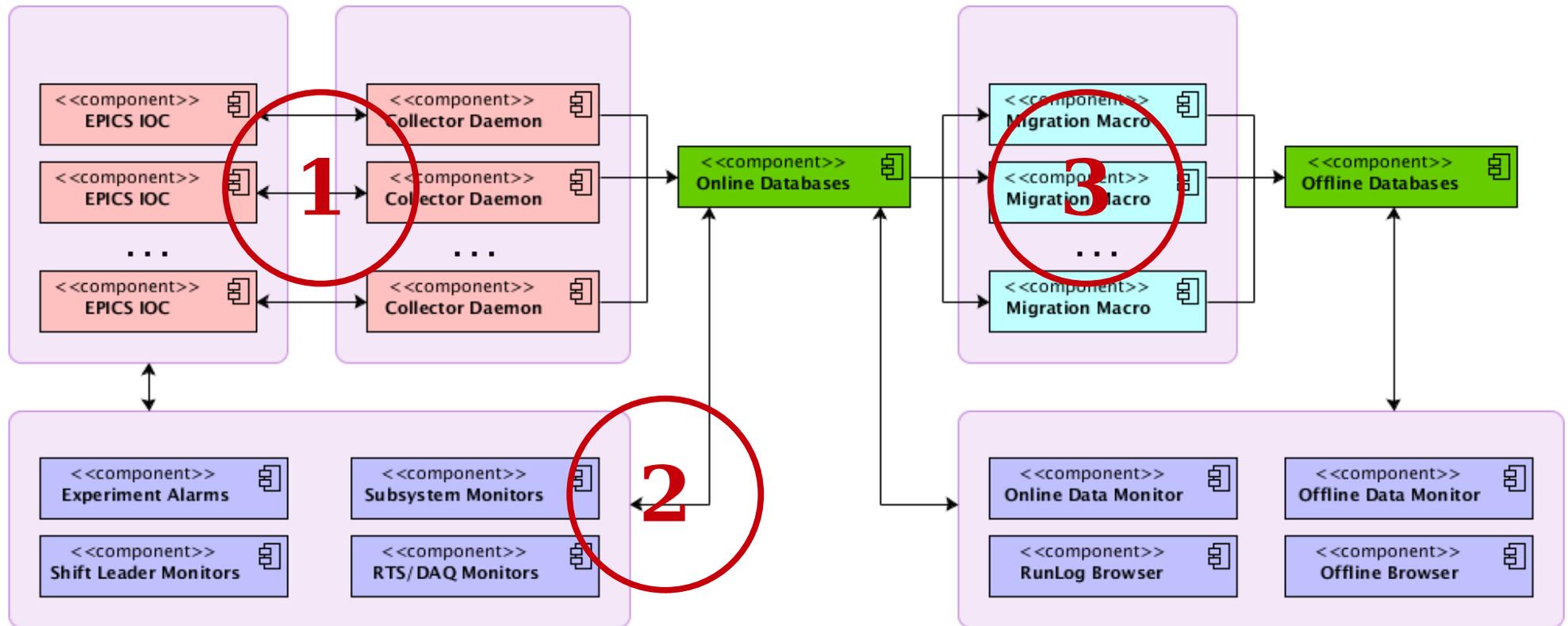
Implemented Solution



- Designed in '90s, implemented in early '00s;
- Utilizes typical data warehousing schema: operational layer, data access + metadata layers, informational access layer;
- Implements “pull” model for getting data from EPICS IOCs;
- Online data processing and monitoring is done by accessing database tables directly (MySQL);
- Data migration (Online DB to Offline DB) is performed by ROOT macros attached to cron;
- 20 independent daemon processes query EPICS IOCs via EZCA, do basic preprocessing, push results directly to MySQL server. Daemons do not have any alarm/notification capabilities;
- Final DB (Offline) is using replicated setup – horizontal scaling;



“Old” schema: hot spots



1. Communications failures between EPICS and STAR Services
2. Subsystem Monitors using OnlineDB instead of EPICS
3. Migration macros: no complex event processing (requested)

Expected features of NEW Online API

- Primary expectations from Online API :
 - Standard API for all clients
 - **unified, stable interface** to access database;
 - minimal external library dependencies;
 - Asynchronous data storage access for writes:
 - **non-blocking** mechanism for sending data to db;
 - “fire and forget” principle: **reliable** data storage access, without sacrificing performance;
 - Automatic load balancing & failover :
 - both read **and** write load balancing;
 - **simple** configuration (or 0-conf) for all clients;

Drawbacks: we planned to provide interface through some custom library, which leaves a question of portability (c,c++,java,shell?). Also, this potentially, will require long-term support for many applications built upon this interface;

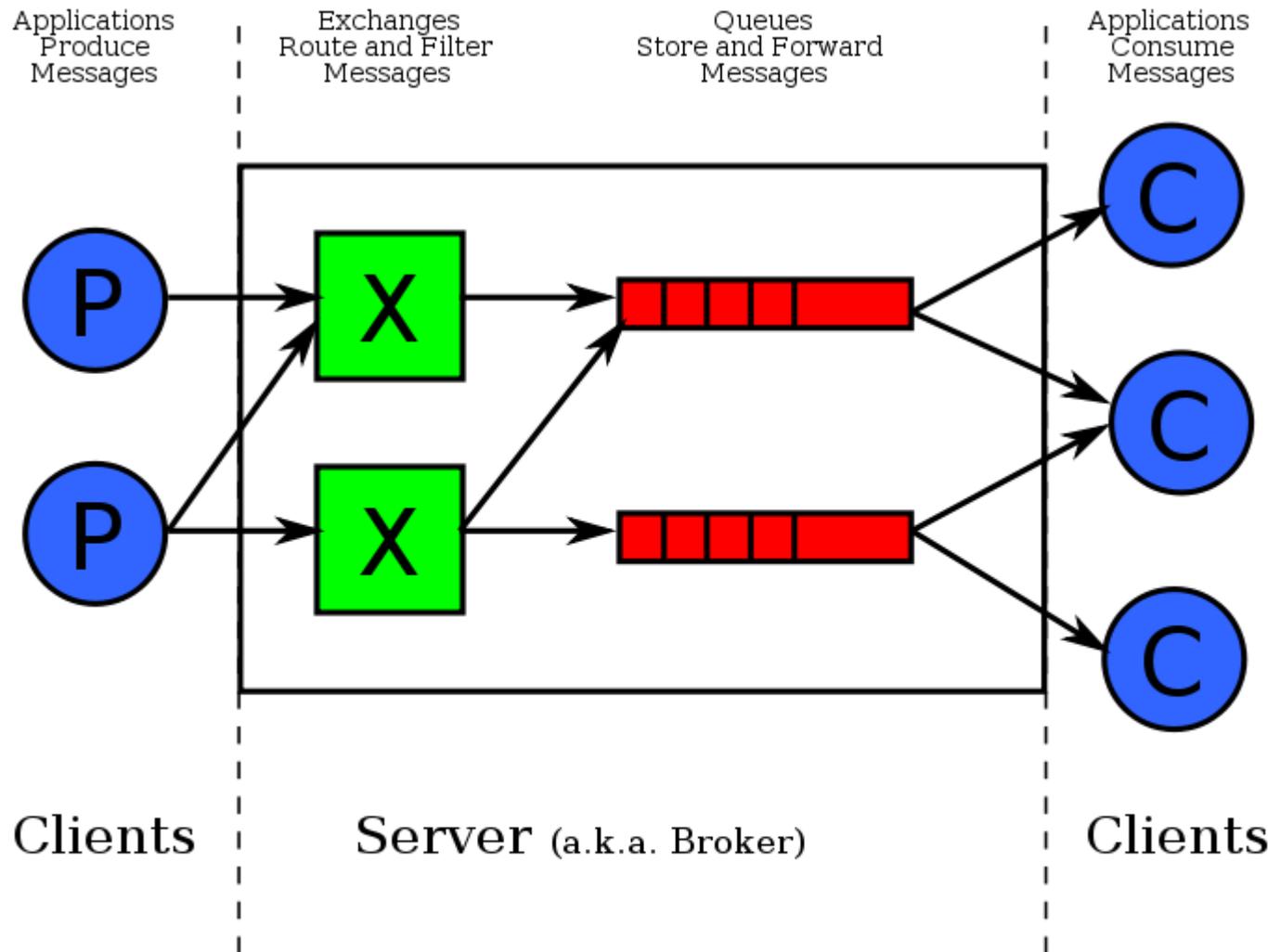
Online API: what we have now

- Existing status: direct access to MySQL DB
 - **good:** all clients use that very same original MySQL API, available for many languages;
 - **bad:** MySQL API lacks features we want to have, plus :
 - **storage** system is used as **messaging** system:
 - RunLog daemon constantly polls database for updates from RTS;
 - RTS daemons (many) constantly write to db and poll for updates made by other RTS daemons;
 - Monitoring tools (many) constantly poll database for updates of specific subsystems;
 - Online QA system is polling DB for updates;
 - There's no way to throttle low-priority polls when system is overloaded;
 - There's no easy/cheap way to coordinate efforts/queries across clients;

Proposed solution

- **Polling is responsible for ~90% of operations : we are using storage engine (MySQL) as messaging system! Let's change the internal schema of online systems : we are going to plug our DB access API into Messaging API, not the other way around..**
- **Most of our needs are fulfilled by almost any Messaging Queue System (MQ), available on a market today:**
 - Async IO, Publish-Subscribe, Request-Response, 1-to-many, many-to-many, 1-to-1 are all part of modern messaging systems by design;
 - Reliable messaging;
 - Defines protocol, not interface, thus allowing many compatible implementations from different vendors/community;
 - Extremely high performance;
- **There are open-source, industry-grade MQ implementations:**
 - Types: broker-based, broker-less;
 - Client API already implemented for many languages: c/c++, java, python etc..

How MQs work, internal design



Entities in the Advanced Message Queuing Protocol (AMQP) model used for message transfer

AMQP + protobuf

AMQP: Advanced Message Queuing Protocol

<http://www.amqp.org>

Originally, standard was developed by JP Morgan-Chase, reference implementation was made by Red-Hat. Since its inception the AMQP Working group has grown to include 20 companies with the specification being improved through refinement, simplification and valuable insights from new members. Today AMQP is connecting hundreds of critical systems in Finance, Telecommunications, Defence, Manufacturing, Internet and Cloud Computing and many additional market segments.

RedHat provides qpid (c++ or java) AMQP broker + many client libraries for free – this is what I propose to use in our setup. NOTE: AMQP does not define message format/structure, it is considered to be a blob. Therefore, we might need an extra lib :

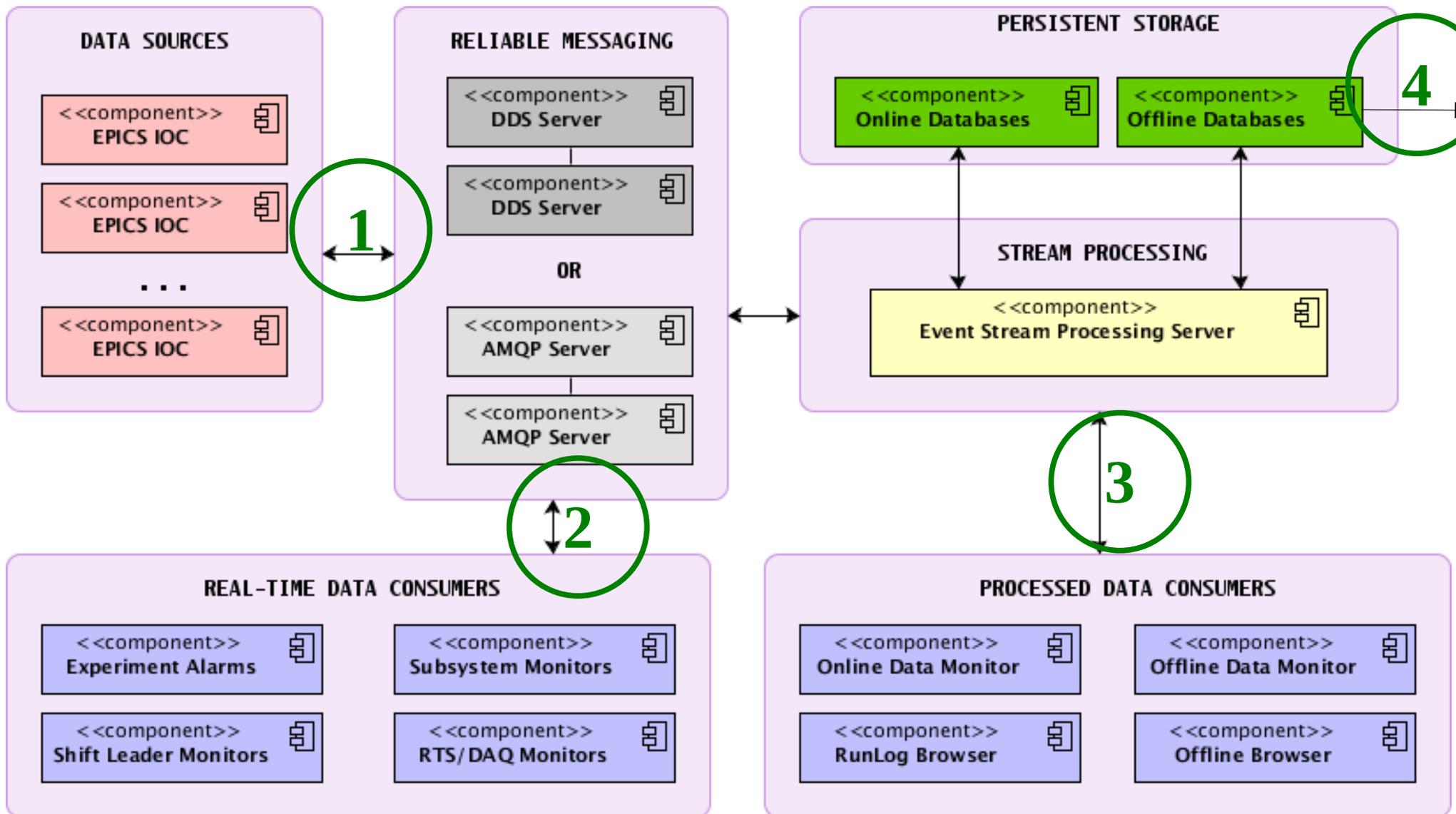
Google Protocol Buffers (protobuf)

<http://code.google.com/p/protobuf/>

Protocol Buffers are a way of encoding structured data in an efficient yet extensible format. Google uses Protocol Buffers for almost all of its internal RPC protocols and file formats.

“...Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. You can even update your data structure without breaking deployed programs that are compiled against the "old" format...”

New schema for Online :



1) EPICS PV+PROTOBUF, 2) CUSTOM FORMAT+PROTOBUF, 3) PURE PROTOBUF, 4) OFFLINE API FORMAT

Primary components:

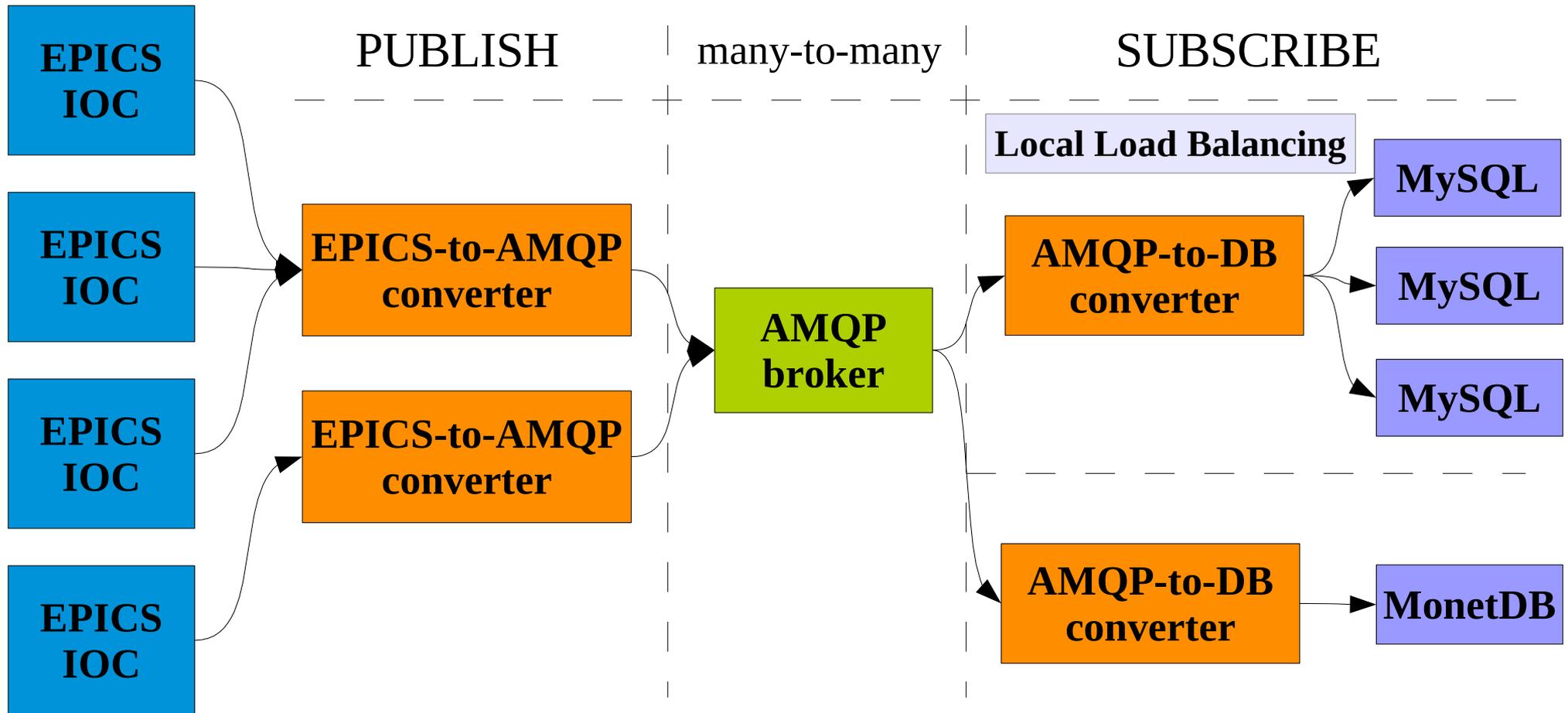
- **PRIMARY COMPONENTS are defined:**
 - MQ server: **qpid** (AMQP 0.10);
 - Message data format & streaming: **protobuf** (Google serialization);
 - Pluggable storage backends (archiver / online db) :
 - **MySQL** : HDD + SSD + RAM hybrid approach, full SQL;
 - **MonetDB** : fastest(RAM-based), limited size SQL frontend / NoSQL core;
 - **Apache/Cassandra** : write-heavy, distributed, scalable NoSQL db;
 - **MongoDB** : distributed, scalable, **round-robin** NoSQL db;
- **API is easy:**
 - Clients will use standard **AMQP 0.10** + **Protobuf** to send/receive data – we don't need to support those at all;
 - EPICS-feed daemons are now greatly simplified: only ship data to AMQP – other clients will decide on how to store/consume that data;
 - Regular DB access: through standalone service, connected to AMQP server: Request/Response and Publish/Subscribe modes simultaneously – all types of LB, caching and throttling are under control now;
- **PERFORMANCE is critical:**
 - **QPID** : 600k messages per second on 1Gb network and commodity hardware;
 - **Google Protobuf** : serialize/de-serialize : ~100-300k objects per second;

So, what we get :

- Original Online API requirements are 100% satisfied:
 - Standard API to access system (AMQP+Protobuf);
 - Asynchronous, reliable protocol (AMQP);
 - Load Balancing and Caching :
 - AMQP server knows how to handle its own load (clustering, internal load balancing);
 - Our daemon has full control over incoming data (queues, cache whatever) – everything is possible with no “cheap bypass” for power-users. System is expected to be much more stable;
 - Client API implementations available for many of languages;
- Extra: “messaging” issue is solved:
 - All sorts of communications between clients are possible now (many programming languages supported);
 - Network load would be significantly decreased: subscribe to channel updates, no more expensive polling every few seconds;
 - Locking issues with database are gone;

Use-Case 1

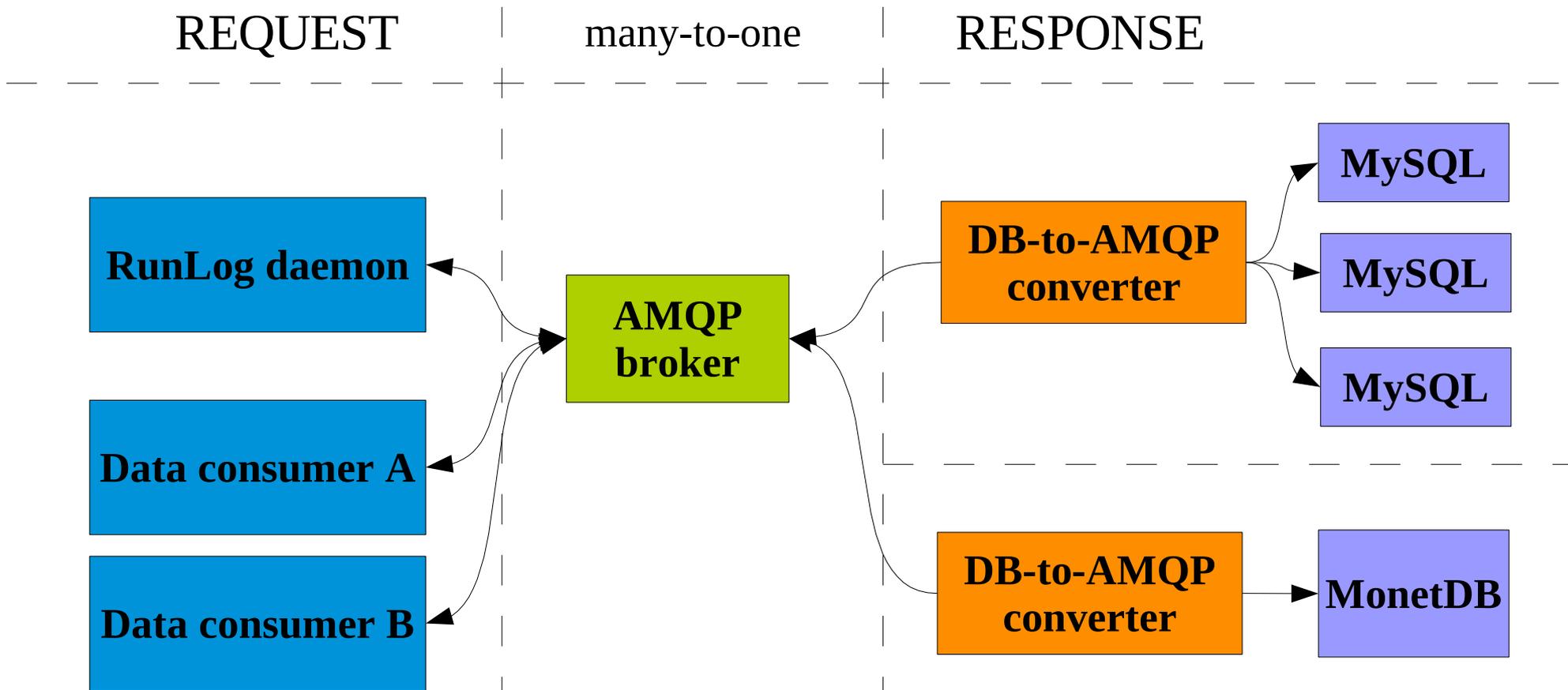
Detector conditions data collection : from EPICS to data storage



Bonuses: a) simple converters, b) pluggable databases with no interruption of service

Use-Case 2

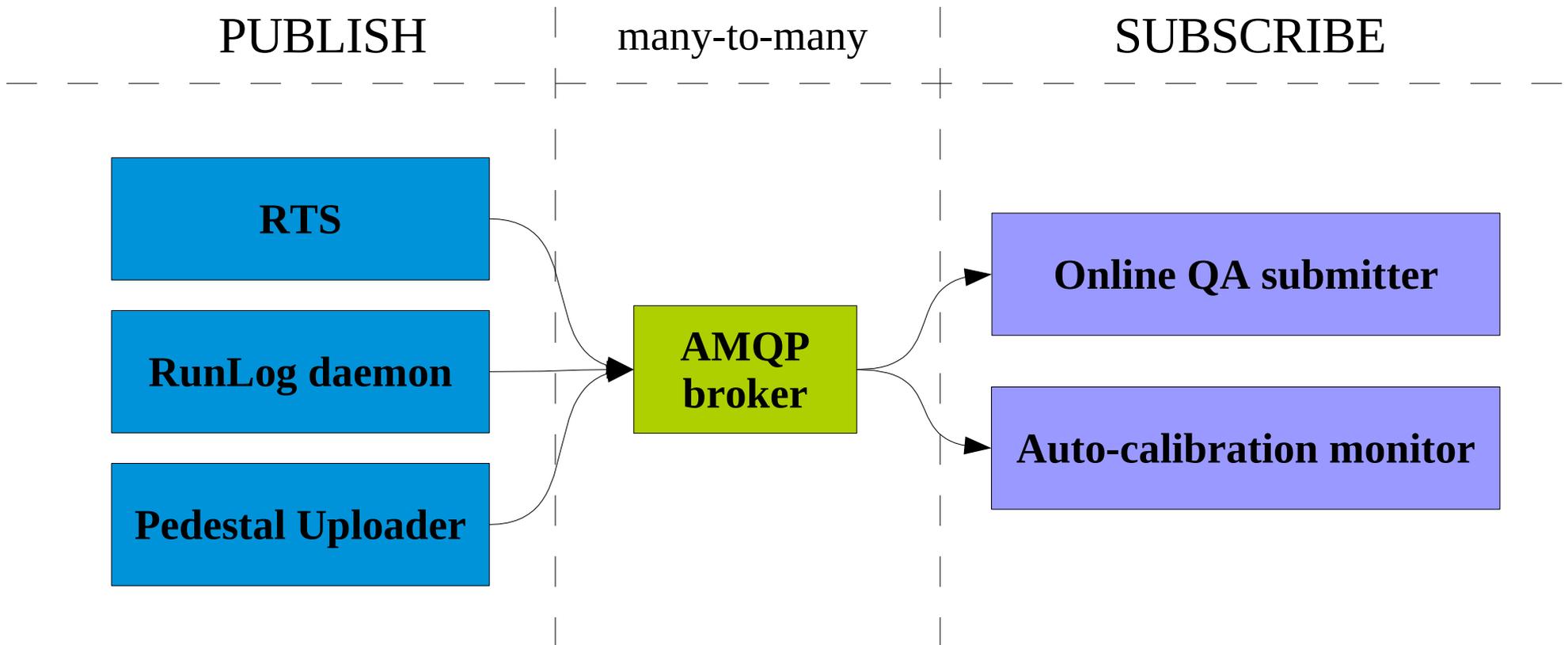
Read data from data storage (e.g. RunLog daemon)



Completely decouples “data storage” from “data consumer” component

Use-Case 3

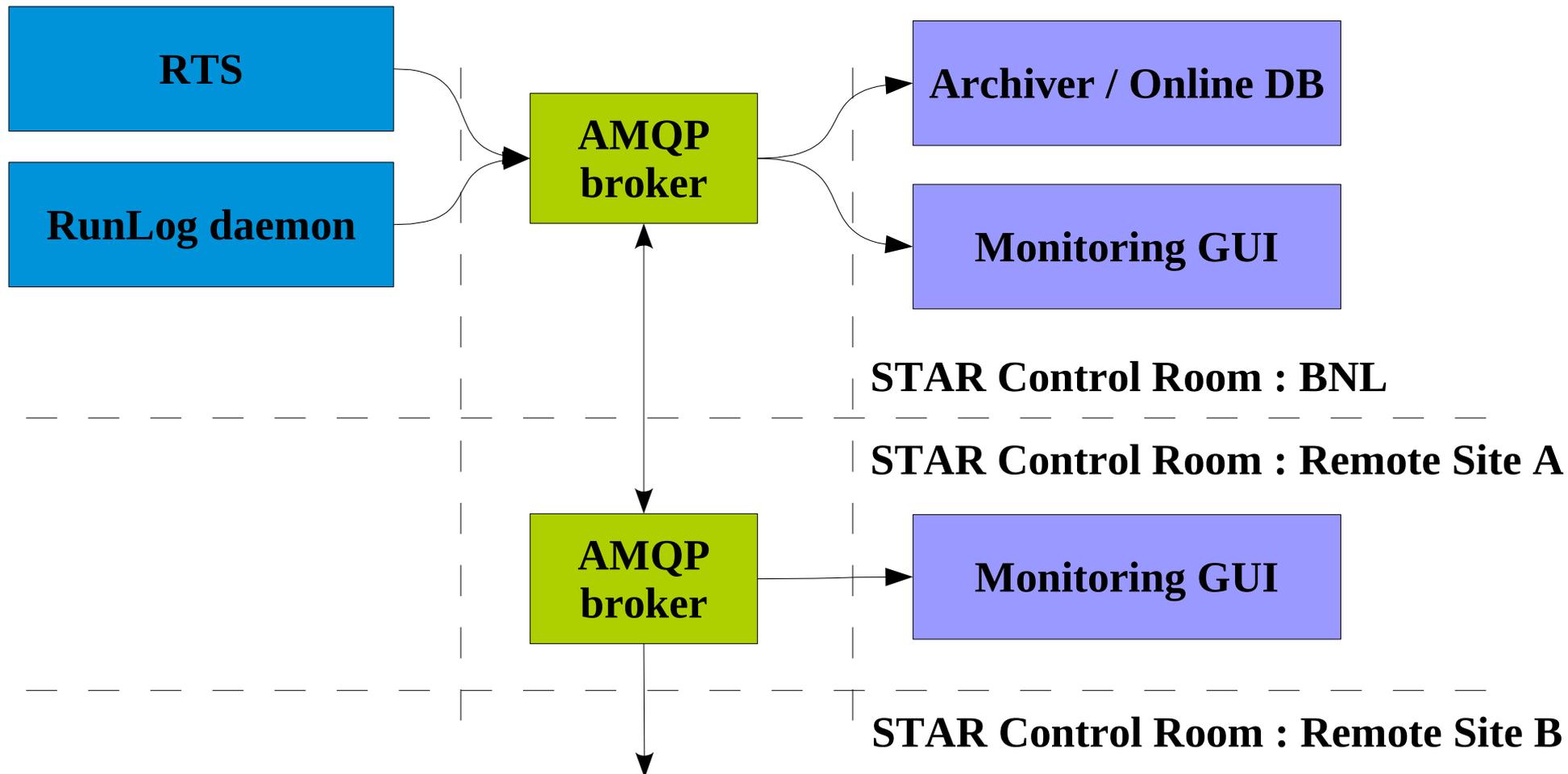
Instant reaction on events (run ended, pedestals uploaded, channel tripped)



Just subscribe for events and let AMQP do its job – no need for frequent polling

Use-Case 4

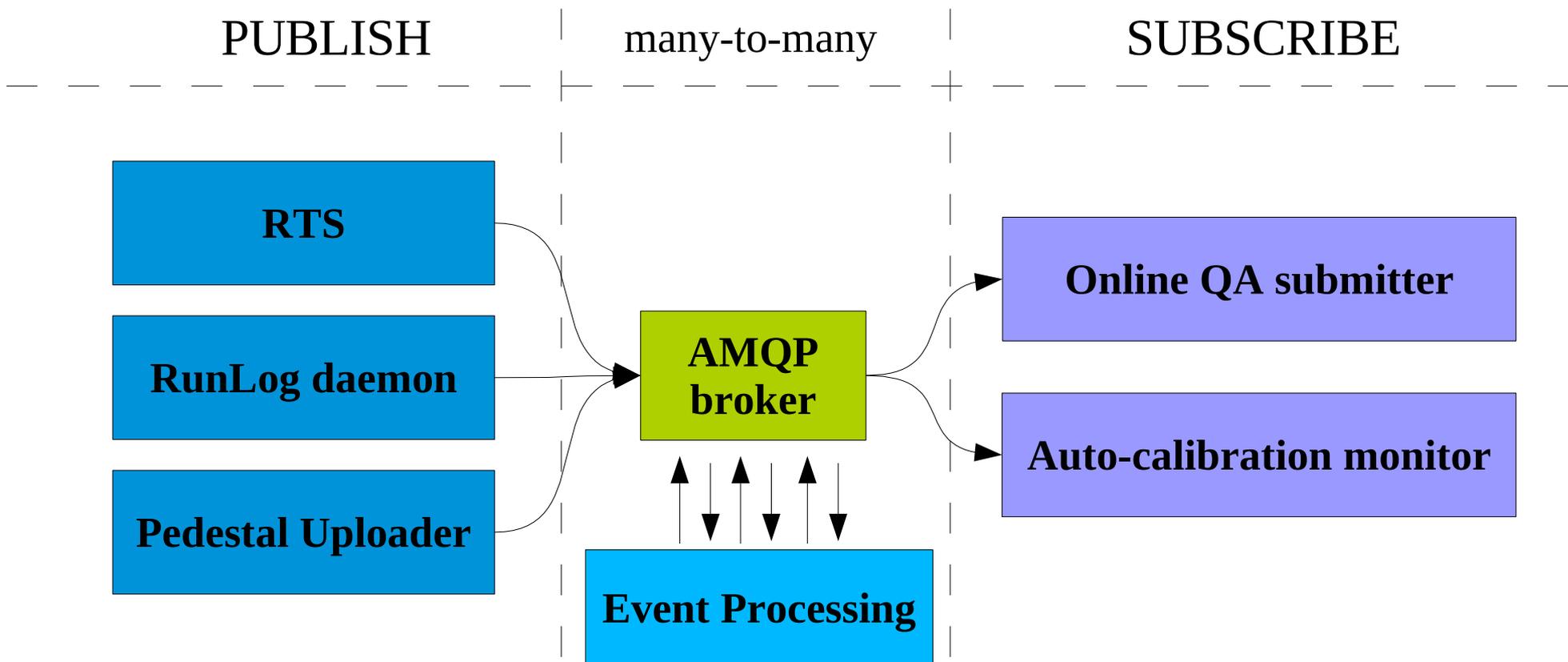
Automatic redirection of streams to remote server : remote CR



It is easy to create completely identical
STAR control room monitors e.g. at LBNL

Use-Case 5

Data transformation and filtering : Event Processing



Event Processing daemons will accept, convert and republish data streams, with no modifications required at client (subscriber) side, still leaving an option to hook into unmodified streams!

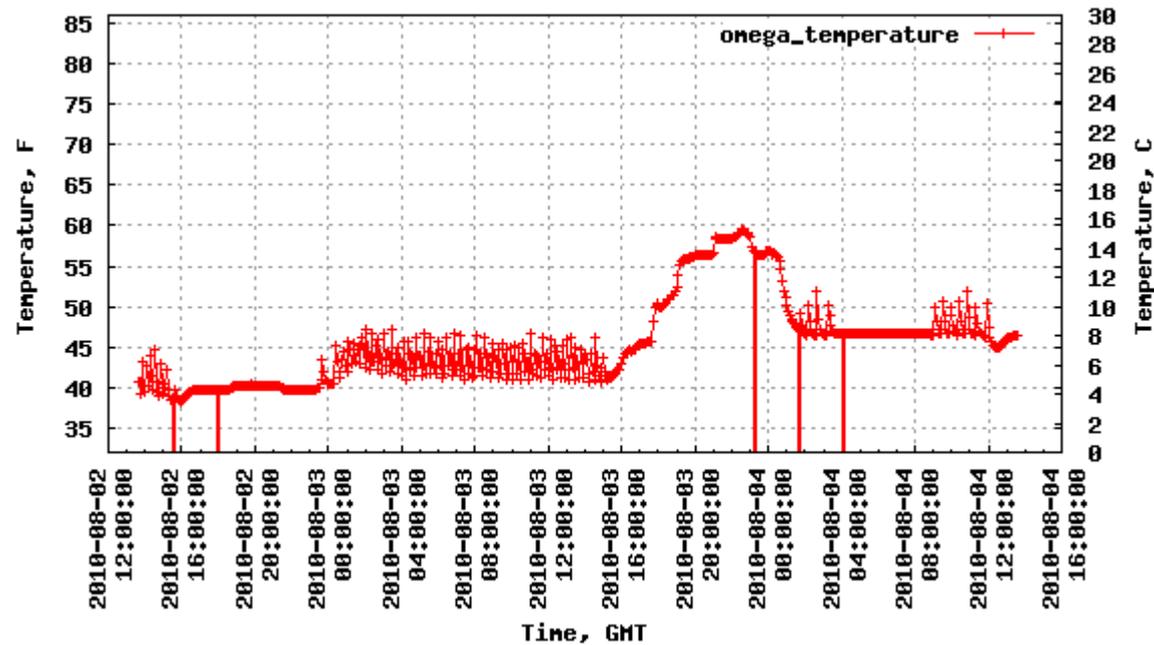
Use-Case 5

Data Flow Monitoring: AJAX-based Web GUI

PUBLISH

many-to-many

SUBSCRIBE



NEXT: Proof-of-concept prototype

- lacks proper error handling
- messy code
- only basic functionality implemented

Proof-of-concept/prototype status

Prototype should cover the following areas :

- EPICS to Online DB data flow example (data storage);
- Online DB to client example (data retrieval);
- Online DB to Monitoring example (data monitoring);
- Client to Client messaging example (inter-component communications);
- Testing should be possible in parallel to current data taking;
- Documentation must cover basic use-cases;

Implemented :

- EPICS base + qpid installed at onl10/11.starp.bnl.gov (protected online nodes);
- EPICS to AMQP example daemon ready (protobuf serializer);
- AMQP to DB example daemon ready (protobuf deserializer);
- DB to AMQP example daemon ready (protobuf serializer/deserializer);
- Monitoring Client (AMQP) ready (ROOT+QT4);
- Client to Client example ready;
- Event processing example ready – need more use-cases;
- Documentation at Drupal CMS – almost ready (1-2 days);

Implementation: data flow standards

EPICS to AMQP : message body/binary + control headers;

AMQP message body: Google Protobuf “message” :

- **simple:** string channel_name, [type] value – fastest (de)serialization;
- **complex:** repeated arrays of key/value pairs – most coverage;

example: *std_msg.proto* :

```
message std_msg_pair_double {  
  required string key = 1;  
  required double value = 2;  
}
```

Control headers for AMQP message :

- required – message serialization format type : plain text, protobuf, PV, custom;
- required – event timestamp;
- optional – extra timestamps : values requested, values received, request sent;
- optional – user categories : e.g.: db name, table name hints for persistent storage;

example: *Content-Type: application/vnd.star.mq.protobuf.msg_complex*

Easy-to-follow standards are defined for all required use-cases, but there is still some room for custom client-client messaging (required for STAR RTS);

Implementation: OS + packages

+++++++ **system packages** ++++++

OS : Scientific Linux 5.3 - default for STAR online pool;
OS kernel : default SMP kernel, will switch to RT kernel later;
AMQP broker : qpid package taken from RedHat MRG server;
EPICS version : EPICS base 3.14.11 + EZCA;

+++++++ **client-specific packages** ++++++

EPICS2AMQP : C++ daemon, based on qpidc+EZCA example;
AMQP2DB : C++ daemon, qpidc + Protobuf + MySQL API;
DB2AMQP : C++ daemon, qpidc + Protobuf + MySQL API;
Client2Client : examples in C++ (java examples on demand);
Monitoring Client : C++/QT4 + qpidc;
Event Processing : C++ daemon, qpidc library;

+++++++

NEXT: Project Timeline

Project timeline

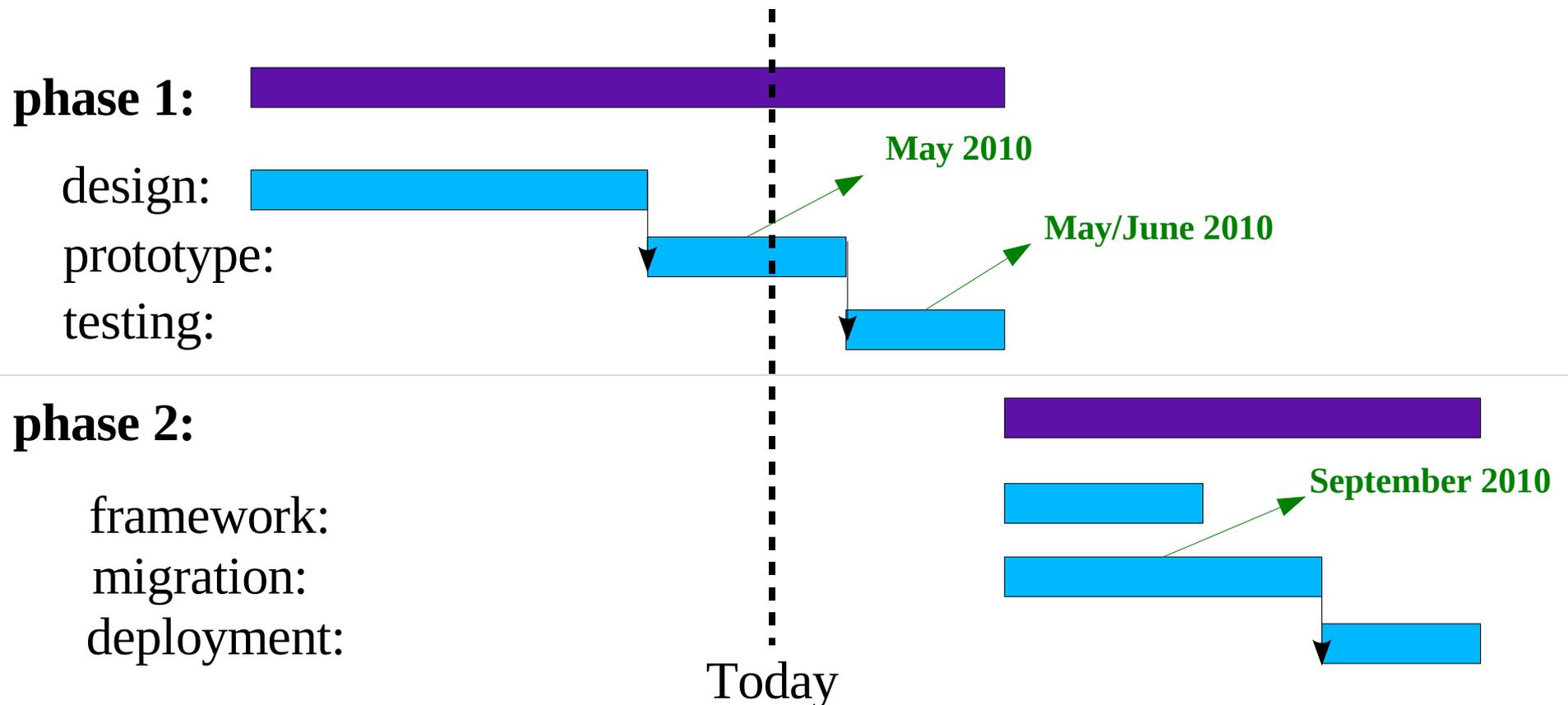
Milestones :

1. phase 1

1. develop new infrastructure schema, including inter-component dependencies;
2. test prototype while STAR is taking data;

2. phase 2

1. create C++ framework for Online DB, tune system performance, stabilize code;
2. migrate/extend existing applications to support proposed schema;
3. use in production Run 11 (starts ~dec 2010) in parallel with existing setup;
4. use in production Run 12 (starts ~dec 2011), drop support for old setup;



THANKS!

qpid API example: publisher (sender)

```
#include <qpid/client/Connection.h>
#include <qpid/client/Session.h>
#include <qpid/client/AsyncSession.h>
#include <qpid/client/Message.h>
...
const char* host = "127.0.0.1";
int port = 5672;
std::string routingKey = "gov.bnl.star.rts.control";
Connection connection;
connection.open(host, port);
Session session = connection.newSession();
Message message;
message.getDeliveryProperties().setRoutingKey(routingKey);
message.setData("some text string or binary object");
async(session).messageTransfer(arg::content=message, arg::destination="amq.topic");
...
```

qpid API example: subscriber

```
#include <qpid/client/Connection.h>
#include <qpid/client/Session.h>
#include <qpid/client/Message.h>
#include <qpid/client/MessageListener.h>
#include <qpid/client/SubscriptionManager.h>
...
const char* host = "127.0.0.1";
int port = 5672;
std::string routingKey = "gov.bnl.star.rts.control";
Connection connection;
connection.open(host, port);
Session session = connection.newSession();
std::string queueName = session.getId().getName();
session.queueDeclare(arg::queue=queueName, arg::exclusive=true, arg::autoDelete=true);
session.exchangeBind(arg::exchange="amq.topic", arg::queue=queueName,
arg::bindingKey=routingKey);
SubscriptionManager subscriptions(session);
LocalQueue lqueue;
subscriptions.subscribe(lqueue, queueNameFull);

Message msg;
while(1) {
    if (lqueue.get(msg, 1000000)) { <process message> }
}
...
```

Protobuf message example

...message.proto...

```
message std_msg_pair {  
  required string key = 1;  
  required string value = 2;  
}
```

```
message std_msg_array {  
  repeated std_msg_pair pair = 1;  
}
```

...

"compiles" into C++ or Java native files with :

```
$> protoc message.proto --cpp_out=.
```