A Scalable and Reliable Message Transport Service for the ATLAS Trigger and Data Acquisition System

Andrei Kazarov, CERN / Petersburg NPI, NRC “Kurchatov Institute”

19th IEEE Real-Time Conference 2014, Nara
Content

• Introduction to TDAQ system
• Error and Message reporting in TDAQ
• Requirements for Message Transfer Service
• High-level design overview
• Implementation aspects
  • CORBA
  • Content filtering with Boost::Spirit
• Performance and scalability tests
• Conclusions
TDAQ in pictures

- Mass storage
  - 1 GB/s
- ATLAS Control Room
- Messages
- 20-60x10^3 applications
- 6 GB/s
- 160 GB/s
- ATLAS
Error reporting in TDAQ

- TDAQ libraries and applications report structured messages ("Issues") using ERS library to predefined streams (DEBUG, INFO, WARNING, ERROR, FATAL), for example

```cpp
throw CanNotOpenFile("filename") ;
ERS_DEBUG(2, "A verbose debug message") ;
ers::warning( BadTestPolicy(relationship, "No objects found", cause) ) ;
```

- ERS allows runtime configuration of the routing of the streams, like

```cpp
TDAQ_ERS_FATAL='stderr, abort'
TDAQ_ERS_ERROR='filter(ABC), stdout, mts'
TDAQ_ERS_DEBUG='null'
```

- Other applications (ERS receivers aka subscribers) can subscribe to certain types of messages, using content filtering

- TDAQ infrastructure includes a service for transferring and content-based filtering of the messages between TDAQ applications, called MTS

- Constraint: non-blocking reporting
Requirements to MTS

- The number of TDAQ applications running in the final system is expected to grow up, along with the reporting rates.

- **Scalability**: MTS architecture shall be scalable with respect to the number of senders $O(10^4)$ and receivers $O(10^2)$ and shall be able to handle high message rates (few kHz of reported messages).

- **Reliability**: MTS shall provide redundancy and shall tolerate a crash of a worker node, there shall not be a single point of failure (though it may rely on external services).

- Handling of *slow* subscribers: in case of presence of slow (or even blocked) subscribers, MTS shall not impose delays neither in notification code, nor in notification of normal subscribers.

- **Simplicity**: architecture and implementation shall be simple and allow easy maintenance in the long time life of the ATLAS experiment.
MTS architecture overview

- MTS is a service for transferring and selecting the messages, based on notify-subscribe communication pattern with content-based message filtering.

- The MTS infrastructure consists of a few identical workers, whose tasks are to share and balance the load and to provide fault-tolerant service in case of a failure of a particular worker process or its host.

- The worker is selected by senders in a random manner to share the load (normally workers run on identical nodes).
MTS high-level design

For sharing of MTS runtime information (the list of active workers and the list of active subscriptions) within MTS cloud other common TDAQ services are used, namely the TDAQ naming service and information sharing service.

MTS includes sender and receiver libraries (C++ and Java) and a worker application.
**CORBA protocol**

- **CORBA** protocol is used for all TDAQ core services as a cross-platform communication middle-ware.

- It hides the details of TCP/IP network communication between MTS actors (senders, workers and receivers)

- Actual IDL declarations are as simple as

  ```
  enum Severity { debug, log, information, warning, error, fatal } 
  struct Issue { string message, Severity sev, Issue cause, ... } 
  interface receiver { 
    oneway void notify(in Issue an_issue); } 
  interface worker: ipc::servant { 
    twoway void report(in Issue an_issue); } 
  ```

- 'report' method uses 'two-way' CORBA pattern, in order to get an explicit confirmation from a worker or to get a fast failure notification (timeout from CORBA), allowing sender to select another worker

- 'notify' method is declared 'one-way' (i.e. asynchronous), in order to release the notification threads as soon as possible (leaving CORBA/TCP to manage buffering, retransmissions etc)
MTS Worker architecture

- Receiving and sending of messages are decoupled in the worker.
- Messages are buffered in queues (one per subscriber), allowing individual handling of subscribers.
- Buffers are unlimited and can keep few millions of messages of a run.
- There are 2 thread pools, one for input and one for output tasks.
Handling of 'slow' receivers

- The “Input” thread pool is managed by CORBA. It’s main task is to match a message against each subscriber and to copy it to buffers
  - These threads are not involved in notification
- A “slow” receiver is a client which can not cope with the given messages rate, which causes overflow of TCP buffers and blocking of the sending part
- The “delivery” thread pool is managed by a custom scheduler, implementing an algorithm of 'fair' receivers handling:
  - Each receiver is assigned a dynamic priority, taking into account:
    - number of threads already allocated to this receiver
    - average delays (microseconds spent in the 'notify' remote call)
- Priorities are counted by the scheduler when it assigns notification tasks to free threads in the pool
  → normal subscribers always get sufficient amount of the resources, still leaving some minimal amount for slow subscribers
EBNF and Boost::Spirit for message filtering

- MTS allows to subscribe to messages with a selection criteria using message attributes: message ID, sender application ID, message severity and message qualifiers, e.g.

(sev=ERROR or sev=FATAL) or (app=Tile* and not qual=debug)

- Extended Backus–Naur Form (EBNF) is used to describe the message selection syntax (aka 'grammar')

\[
\begin{align*}
\text{key} & = \text{`app' | `msg' | `qual'} \\
\text{sev} & = \text{`sev'} \\
\text{selev} & = \text{`fatal' | `error' | `warning' | `info' | `information'} \\
\text{token} & = +[a-zA-Z:\_\-]* \\
\text{item} & = (\text{key} (= | !=) \text{token}) | (\text{sev} (= | !=) \text{selev}) \\
\text{factor} & = \text{item} | (\text{expression}) | (\text{`not'} \text{ factor}) \\
\text{expression} & = \text{`*'} | \text{factor} *((\text{`and'} \text{ expression}) | (\text{`or'} \text{ expression}))
\end{align*}
\]

- Use of EBNF allows almost literally translate the above syntax to C++ code for Boost::Spirit parser generator framework
  - Boost implements expression validation, parsing and compiling (i.e. matching)
  - easy modifications of filtering syntax are possible if needed
Performance tests

- Tests were performed on a cluster of ~90 nodes with number of applications simulating the conditions of the final TDAQ system.
- Number of senders and receivers were varying, producing different message rates through MTS (2 and 4 workers) up to the saturation.

<table>
<thead>
<tr>
<th>#of senders</th>
<th>#of receivers</th>
<th>#of workers</th>
<th>total input rate, kHz</th>
<th>total output rate, kHz</th>
<th>Network link utilization, GBps</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>80</td>
<td>2</td>
<td>3</td>
<td>285</td>
<td>50</td>
</tr>
<tr>
<td>3000</td>
<td>96</td>
<td>2</td>
<td>2.9</td>
<td>237</td>
<td>70</td>
</tr>
<tr>
<td>15000</td>
<td>32</td>
<td>2</td>
<td>14.6</td>
<td>460</td>
<td>119 (saturation)</td>
</tr>
<tr>
<td>15000</td>
<td>32</td>
<td>4</td>
<td>15</td>
<td>680</td>
<td>90</td>
</tr>
<tr>
<td>30000</td>
<td>16</td>
<td>2</td>
<td>30</td>
<td>460</td>
<td>114 (saturation)</td>
</tr>
<tr>
<td>30000</td>
<td>32</td>
<td>4</td>
<td>30</td>
<td>880</td>
<td>77</td>
</tr>
</tbody>
</table>

Net result: a single MTS worker (running on 24-core 2.4GHz Intel E5645 node) can handle up to **15k** senders, **15kHz** of input and **230kHz** of output messages rate, in this conditions saturating 1Gbps network link.

Configuration of 2 workers can serve 30000 senders, and more workers can be added when needed.
Scalability & latency tests

- MTS keeps average message delivery latency unnoticeable for operator (below 10ms) for all possible rates
Real run tests

- MTS was validated during test runs in 2014, when TDAQ was running up to 60000 applications, producing up to 18kHz of messages rates.
Conclusions

- A reliable, high-performance and scalable message transfer service is implemented for the message reporting for applications in the ATLAS TDAQ system.
- Functionality and performance of the service was validated during TDAQ technical and first ATLAS milestone runs.
- Wide usage of other TDAQ services and third-party software (like CORBA and Boost) allowed the MTS implementation to be compact and easy to maintain.
  - The total size of the package is just 2300 lines of C++ and 520 lines of Java code.