# The WLCG Messaging Service and its Future[1]

**Lionel Cons, Massimo Paladin**

CERN - IT Department, 1211 Geneva 23, Switzerland

E-mail: Lionel.Cons@cern.ch, Massimo.Paladin@cern.ch

**Abstract.**

Enterprise messaging is seen as an attractive mechanism to simplify and extend several portions of the Grid middleware, from low level monitoring to experiments dashboards. The production messaging service currently used by WLCG includes four tightly coupled brokers operated by EGI (running Apache ActiveMQ and designed to host the Grid operational tools such as SAM) as well as two dedicated services for ATLAS-DDM and experiments dashboards (currently also running Apache ActiveMQ). In the future, this service is expected to grow in numbers of applications supported, brokers and technologies.

The WLCG Messaging Roadmap identified three areas with room for improvement (security, scalability and availability/reliability) as well as ten practical recommendations to address them. This paper describes a messaging service architecture that is in line with these recommendations as well as a software architecture based on reusable components that ease interactions with the messaging service. These two architectures will support the growth of the WLCG messaging service.

## 1. Enterprise Messaging

Enterprise messaging[1] is a technology which allows messages exchange between peers and where the exchange is mediated by messaging brokers. A messaging broker is a server whose role is to route messages between multiple peers offering different exchange patterns which enable the peers to customize messages routing.

Messaging enables loosely coupled architectures where producers and consumers do not need to know about each other; it enables asynchronous communications and offers high flexibility to client applications.

Messaging can also be explained with a simple analogy: messaging is for software applications what electronic mail is for human beings.

## 2. Messaging in WLCG: history

Messaging has been introduced in the Grid middleware in 2008, to simplify and improve several loosely coupled components that needed to exchange some of their data.

It started as a prototype to evaluate its usefulness in some parts of the Grid middleware, like monitoring, with a relative low requirements in terms of security, scalability and reliability.

To support this evaluation effort, EGI operated a small cluster of four tightly coupled brokers running Apache ActiveMQ[12]: two at CERN, one at AUTH and one at SRCE[2].

[1] This work was partially funded by the EMI project under European Commission Grant Agreement INFSO-RI-261611

In this context, messaging has proved to be a very suitable technology ([3], [4], [5], [6], [7], [8], [9], [10] and [11]). With time, more and more applications wanted to also take advantage of messaging.

## 3. WLCG Messaging Roadmap

The interest for messaging as an integration middleware increased both inside CERN (from the computer center to the accelerators control) and outside (from low level monitoring to experiments dashboards).

A lot of research and investigations have been conducted by the CERN Messaging Team[13] and the EMI Messaging Product Team[14]. The requirements have been analyzed, several messaging technologies have been evaluated and the service architecture has been defined in order to satisfy the foreseen increase in demand.

This effort lead to the WLCG Messaging Roadmap[15] that aims to address the combined requirements of potential applications using messaging in the WLCG context. This roadmap lists main areas for improvement as well as concrete recommendations.

### *3.1. Areas for improvement*

Although the WLCG messaging service is successfully used by Grid operational tools, three areas has been identified with room for improvement:

**Security:** security was not a priority in the beginning, the situation changed and it has now to be taken into account.

**Scalability:** more and more applications used or wanted to use messaging, yielding to a major increase in capacity requirements. The monolithic model which was in use could not easily scale. Different applications may require different broker tunings and in order to get a very good performance, one may have to select the best technology combination for each use case.

**Availability/Reliability:** although the four EGI brokers were part of a kind of cluster, the whole service needed to be stopped during some interventions (e.g. software upgrades or major configuration changes). In addition, some applications had single point of failures (e.g. Apache Camel[16] to aggregate messages): the unavailability of one machine stopped the application.

### *3.2. Recommendations*

Here are the proposed changes which address the three areas identified in order to provide a better messaging service for WLCG:

(i) authenticate all messaging clients
(ii) authorize only what is required (*deny all* by default)
(iii) audit the broker activity
(iv) use application level security
(v) use dedicated messaging services
(vi) scale each service according to its requirements
(vii) loosely connect the services that need to exchange messages
(viii) use standalone brokers instead of tightly coupled ones
(ix) get rid of single point of failures
(x) build more reliable messaging clients

## 4. Messaging service architecture

### *4.1. Using dedicated messaging services*

The WLCG Messaging Roadmap recommends to use dedicated messaging services. Each service will use the best technology combination to fulfill the requirements of the applications it supports. These dedicated services could use different hardware and/or different broker software and/or different communication protocols.

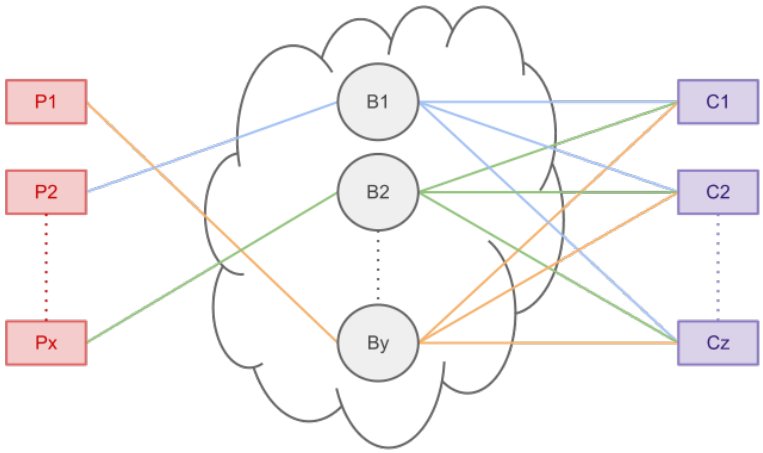


**Figure 1.** Dedicated messaging service of independent brokers.

Dedicated messaging services also have a positive impact on security:

- a security attack against one service would normally not impact the other services
- separated services can be more tightly controlled and secured; for instance, a service with clients using known and fixed IP addresses could have a firewall to limit who can connect to it

### *4.2. Scaling each service according to its requirements*

A messaging service could be provided by a machine that already provides other services such as web services or database services. After all, we can see a messaging broker as just one more daemon to enable on a server.

At the other end of the scale, a single messaging service could have tens of dedicated servers, running nothing else than the broker daemon. Scaling up may require more powerful hardware and/or increased network capacity. It all depends on the application requirements.

When multiple machines are used for the same service, we recommend to have them completely independent. This may put some constraints on the message clients (that may have to contact several brokers to send or receive the messages they want) but this provides true linear scalability. In the typical Grid use case (many distributed producers, very few consumers), this is very easy to achieve as illustrated in figure 1:

- producers connect to any broker (randomly)
- consumers connect to all brokers

This simple architecture has already been validated (e.g. by ATLAS-DDM) and brings additional benefits:

- improved availability (the service works as long as one broker works)
- simplified management (one can stop/upgrade/restart one broker at a time without disrupting the service)

*4.3. Loosely connecting the services that need to exchange messages*

With a monolithic service, any client can connect to any broker and potentially receive or send any message (provided that security allows it). Once we use different services, how can messages been shared between applications?

We recommend to use *shovels* for this. These are dedicated applications that do nothing but copying messages from one broker to another. They scale very well: if you need more throughout, just throw more shovels at it. By shoveling only the messages that need to be shoveled (e.g. with filtering), they minimize the load on the network and on the brokers. They are also compatible with the proposed enhanced security since, from the broker point of view, they appear just as normal clients that need to be authenticated and authorized.

These independent services provide application isolation so they increase the overall reliability by limiting the consequences of a bogus application.

Better reliability must also come from the messaging clients themselves, it is not enough to be implemented on the server side. Designing and writing a solid message client application is not straightforward and error handling is often neglected.

## 5. Messaging applications design

Applications using messaging should have a solid architecture in order to offer redundancy and reliability. It often happens that reliability is expected from the broker side but not reflected on the client side. Making full confidence on the broker side is a common mistake on integrating an application with messaging.

Using messaging in an unreliable way requires very few lines of code, using it in a reliable way takes much more time and thoughts. All the unexpected behaviors should be taken into consideration and handled properly, most of the code needed is about exception handling:

- what to do if you cannot connect to the broker? reconnect? how many times? how long do you wait?
- what if the delivery of one message fails? retry? when to give up? what to do with the message?
- what if you do not get a reply/acknowledgment quickly enough? wait more? abort?

The problem get even more complicated if a given application need to support multiple protocol and/or multiple programming languages yielding to code duplication and bugs multiplication.

*5.1. Reusing solid messaging components*

We can use Lego$^{TM}$bricks concept: small, robust and flexible components which can be combined into complete solutions addressing the most common use cases. The simple *bricks* which suit messaging needs are:

**Message Queue (MQ):** file-system based message queue with a simple yet robust API which supports concurrent access

**Messaging Transfer Agent (MTA):** program that can transfer messages between a broker and a message queue (all combinations)

These two components can be easily combined. In figure 2 it is shown how components can be combined for the consumer use case and to scale it up in case of bottlenecks:

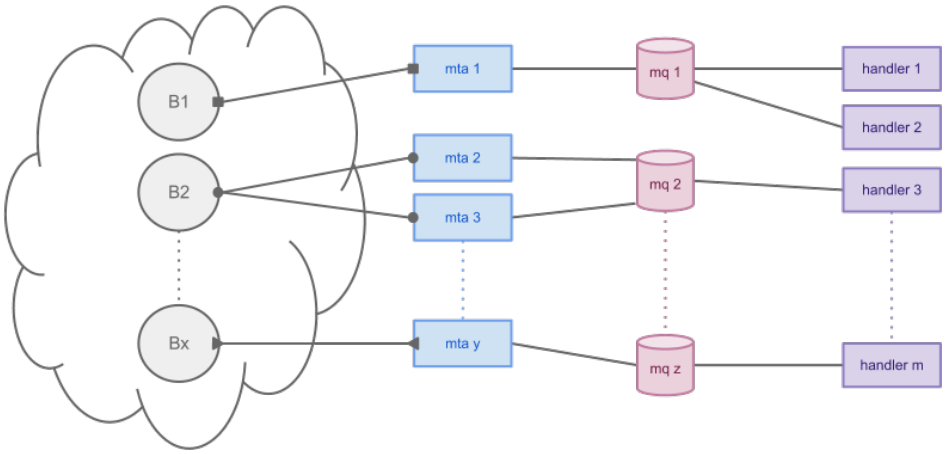


**Figure 2.** Messaging bricks and the consumer use case.

- *broker:* the number of brokers can be increased to support the volume of messages
- *MTA:* one or more *MTA* can be configured to consume messages from a broker; having more than one *MTA* per broker increase the throughput by load balancing the consumption of messages
- *MQ:* more than one *MQ* can be used to balance the load if the file-system is a bottleneck
- *handler:* if the application handler is a bottleneck, more instances can run in parallel

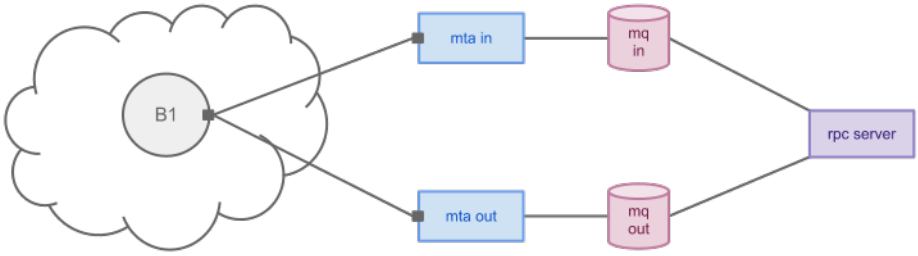


**Figure 3.** Messaging bricks and the RPC use case.

The producer use case is symmetric to the consumer side. Another important use case which is often used is the RPC use case, in figure 3 a set of *MTA* and *MQ* is combined to solve the use case:

- *input:* one *MTA* is configured to receive messages from the broker and store them in an input *MQ*
- *rpc server:* it processes requests from the input *MQ* and produces responses in an output *MQ*

- *output:* one *MTA* is configured to forward responses from the output *MQ* to the broker

Looking at the two previous use cases it should be clear that such flexible components give significant benefits:

- an application can use messaging without knowing anything about messaging, protocols and other details
- only the interaction with the *MQ* is required (very simple API)
- a single *MTA* implementation can be reused across multiple applications avoiding completely code duplication
- a swap of *MTA* is enough to support different protocols/broker technologies

### *5.2. Assembling many messaging components*

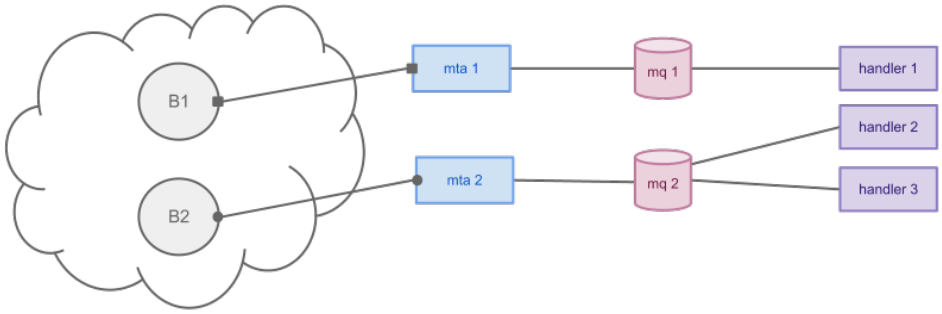


**Figure 4.** Example of a consumer setup.

An application using messaging could be developed for environment which grows and shrinks as needed, if a simple use case (e.g. figure 4) is taken into account there are already five processes to run and monitor.

The best way to manage these components is to use *supervisors*[17]. Like in Erlang OTP it is possible to declare hierarchies of supervisors and workers to build services. Hierarchies are useful and important because supervisors can be configured to handle children failures in a custom way.

The example shown in figure 4 could be implemented with the supervision tree in figure 5:

- *mta1* and *handler1* are grouped under the same supervisor. *handler1* depends on *mta1* because if there is no messages it is pointless for him to run and we may want to monitor their aggregated status
- *mta2* and *handler2/3* are grouped under the same supervisor for a similar reason to the previous point
- since a hierarchy is expected we can then aggregate the two previous supervisors in a top one where we can customize the failure policies

### *5.3. Software availability*

All the components mentioned here are freely available. The EMI Product Team has selected and/or developed some of them[18]:

For the message queue:

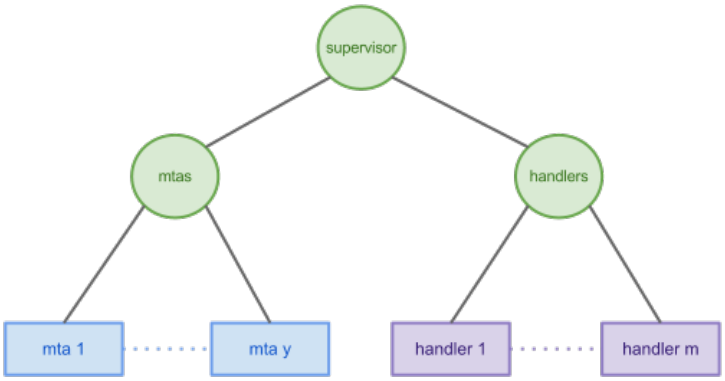


**Figure 5.** Supervision tree for example in figure 4.

- Perl libraries: perl-Messaging-Message + perl-Directory-Queue
- Python libraries: python-messaging + python-dirq

For the messaging transfer agent, there are two implementations[19]:

- stompclt: a versatile client for the STOMP[20] protocol
- amqpclt: a versatile client for the AMQP[21] protocol

For the supervisor[19]:

- simplevisor: a simple daemons supervisor

## 6. Current state

Currently, three different messaging services are used by WLCG:

**EGI:** this is the service which is operated by EGI and used by the Grid operational tools.

**ATLAS-DDM:** this service is used by the ATLAS experiment for several applications like the DDM Tracer and other operational tools; it consists of 2 uncoupled production brokers plus 1 testing broker. The two production brokers are running Apache ActiveMQ and the test broker is running both ActiveMQ and ActiveMQ Apollo[22].

**Dashboard:** this service is used by the dashboard team for their monitoring application; it consists of 2 uncoupled production brokers plus 1 testing broker. Like the ATLAS service the two production brokers are running Apache ActiveMQ and the test broker is running both Apache ActiveMQ and Apache ActiveMQ Apollo.

All the applications use STOMP (mainly) or OpenWire[23] (rarely) to access the brokers that run either ActiveMQ 5.x or ActiveMQ Apollo (that should become ActiveMQ 6).

## 7. Future

In the future, we expect many more applications using messaging and therefore we expect both the number of brokers and the number of dedicated services to grow significantly.

Messaging is a dynamic technology and will in parallel also change. ActiveMQ is getting to the end of its life-cycle while other broker software like ActiveMQ Apollo, RabbitMQ[24] and Qpid[25] get more and more solid for the future. In terms of protocols, AMQP usage will very likely grow now that it has matured (the 1-0 specification has been released on October 2011).

So we also expect a growth in the number of broker types used as well as in the number of protocols used.

Beyond this, we foresee that the same kinds of services and components will be used outside of the Grid. For instance, at CERN, the *Agile Infrastructure* project is about to use a dedicated service running ActiveMQ Apollo and RabbitMQ.

## 8. Conclusions

The architectures presented in this paper (services: independent brokers; software: components to be assembled) can support this multidimensional growth in the use of messaging for WLCG.

Future usage of messaging in WLCG should follow guidelines and recommendations for a more sustainable messaging service. It is fundamentally important to be broker and protocol agnostic when making usage of messaging; it is required in order to have the flexibility to switch easily between different technologies and avoid to remain stuck on a legacy technology.

## References


[1] Enterprise Messaging `http://en.wikipedia.org/wiki/Enterprise_messaging_system`.
[2] EGI Messaging Service `https://wiki.egi.eu/wiki/Message_brokers`.
[3] Service Availability Monitoring framework based on commodity software, #404 CHEP 2012.
[4] The ATLAS DDM Tracer monitoring framework, #337 CHEP 2012.
[5] New solutions for large scale functional tests in the WLCG infrastructure with SAM/Nagios: the experiments experience, #263 CHEP 2012.
[6] SYNCAT - Storage Catalogue Consistency, #419 CHEP 2012.
[7] Next generation WLCG File Transfer Service (FTS), #436 CHEP 2012.
[8] Service monitoring in the LHC experiments, #245 CHEP 2012.
[9] OSG Ticket Synchronization: Keeping Your Home Field Advantage In A Distributed Environment, #180 CHEP 2012.
[10] WMSMonitor advancements in the EMI era, #487 CHEP 2012.
[11] Implementing data placement strategies for the CMS experiment based on a popularity mode, #176 CHEP 2012.
[12] Apache ActiveMQ `http://activemq.apache.org/`.
[13] CERN Messaging Team `https://tomtools.cern.ch/confluence/display/MIG/Home`.
[14] EMI Messaging Product Team `https://twiki.cern.ch/twiki/bin/view/EMI/EMIMessaging`.
[15] WLCG Messaging Roadmap `https://tomtools.cern.ch/confluence/download/attachments/983148/wlcg-messaging-roadmap.pdf`.
[16] Apache Camel `http://camel.apache.org/`.
[17] Erlang supervisors `http://www.erlang.org/doc/design_principles/sup_princ.html`.
[18] EMI Messaging Libraries `https://twiki.cern.ch/twiki/bin/view/EMI/MessagingLibraries`.
[19] EMI Messaging Software `https://twiki.cern.ch/twiki/bin/view/EMI/MessagingSoftware`.
[20] STOMP - The Simple Text Oriented Messaging Protocol `http://stomp.github.com/`.
[21] AMQP - Advanced Message Queuing Protocol `http://www.amqp.org/`.
[22] Apache ActiveMQ Apollo `http://activemq.apache.org/apollo/index.html`.
[23] OpenWire protocol `http://activemq.apache.org/openwire.html`.
[24] RabbitMQ `http://www.rabbitmq.com/`.
[25] Apache Qpid `http://qpid.apache.org/`.