Modern Messaging for Distributed Sytems

2015 J. Phys.: Conf. Ser. 608 012038

(http://iopscience.iop.org/1742-6596/608/1/012038)

Modern Messaging for Distributed Sytems

L Magnoni[1]

[1] CERN, European Laboratory for Particle Physics (CERN), Geneva, Switzerland

E-mail: luca.magnoni@cern.ch

Abstract.

Modern software applications rarely live in isolation and nowadays it is common practice to rely on services or consume information provided by remote entities. In such a distributed architecture, integration is key. Messaging, for more than a decade, is the reference solution to tackle challenges of a distributed nature, such as network unreliability, strong-coupling of producers and consumers and the heterogeneity of applications. Thanks to a strong community and a common effort towards standards and consolidation, message brokers are today the transport layer building blocks in many projects and services, both within the physics community and outside. Moreover, in recent years, a new generation of messaging services has appeared, with a focus on low-latency and high-performance use cases, pushing the boundaries of messaging applications. This paper will present messaging solutions for distributed applications going through an overview of the main concepts, technologies and services.

1. Introduction

This paper presents an overview of messaging concepts, functionalities and modern technologies. It starts with an introduction of messaging for distributed communication and system integration. A review of the main messaging features is then provided, followed by an overview of the major technologies for messaging, from broker to broker-less systems. In conclusion, a list of successful stories concerning the use of messaging to solve the problem of communication for distributed applications is presented.

2. Messaging for loosely coupled communication

Modern distributed systems can be comprised of hundreds, if not thousands, of applications operating in multiple tiers and providing different services and functionality to each other. In such a distributed architecture, there are many challenges such as network unreliability, strong-coupling of producers and consumers and the heterogeneity of applications which need to be addressed to build a solid and reliable system.

2.1. Connection-oriented communication

Connection-oriented communication is a simple solution to exchange information among remote entities. As presented in Figure 1, consider opening a socket over a connection-oriented protocol such as TCP/IP and transmit a raw stream of data through it. That would be a fast and inexpensive way to exchange information, but at the same time that tightly-coupled communication would be based on a number of assumptions which needs to be satisfied in order the communication to take place:

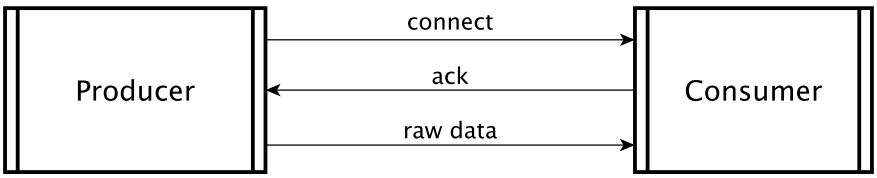


Figure 1: Tightly coupled communication.

- **Temporal dependency**: all components have to be available at the same time.
- **Location**: each component must know each other address.
- **Data structure and representation**: in its simplest implementation, all components have to agree on the data format and on the binary representation.

2.2. Messaging for loosely coupled communication
Coupling can be measured as the number of assumptions parties make about each other when they communicate. Messaging is an example of loosely coupled communication solution where *message* is the information building block, which aims at minimizing those assumptions. Instead of sending information directly to a specific address, it can be sent to an addressable *channel*, to resolve the location dependency. In order to remove the temporal dependency, that channel can be enhanced to queue up the information until the remote components are ready to receive it. This way, the producer can now send requests into the channel and continue processing without having to worry about delivery. Messaging makes no assumption on data representation, so that standard data format, e.g. self-describing and platform independent as JSON or XML, can be used to remove the need to share data handling logic among all components.

2.3. Messaging scenario
Typical messaging use cases are: *Information Publishing*: an entity publishes volatile information with no a-priori knowledge about who is interested (e.g. sensor); *Information Storing*: an entity collects information from multiple sources (e.g. log collector); *Remote Procedure Call*: an entity sends request to one or more remote entity and expect reply.

2.4. Messaging middleware
Messaging is a loosely coupled communication solution which minimizes producer and consumer dependencies. Removing these dependencies makes the overall architecture more flexible and tolerant to changes, but it comes with additional complexity. Therefore, dedicated messaging middleware has been developed over the years to provide messaging functionality without having to deal with the inner complexity. The next section describes the main concepts and principles of messaging systems.

3. Messaging systems
A messaging system, as shown in Figure 2, acts as an indirection layer among entities that want to communicate. Usually referred as *message broker*, it is responsible for transferring data, as messages, from one application to another, so that producer and consumer can focus on what to share rather than on how to share it. Like many other technologies, messaging is based on some basic concepts and properties which are shared among all the different flavours and implementations.

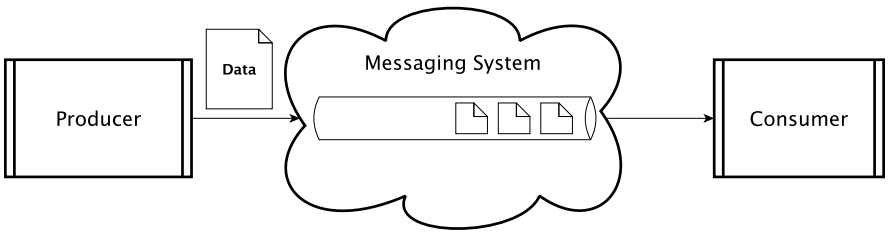


Figure 2: Messaging for loosely coupled communication.

3.1. Message
A message is the information building block. It is composed by a *body*, which is immutable and contains the structured data (e.g. JSON, XML, serialization protocols) object of the communication and by a set of *header*, normally key value pairs which can be processed by the broker and used for routing.

3.2. Communication models: topic and queue
Messaging systems support different communication models, each one defining how the information is exchanged among producer and consumer. The most common communication models are *queues* and *topics*. Queue are used to implement a point-to-point communication, where, if no consumer exists when the information is produced, the message is kept in the channel for later delivery, whilst if there are multiple consumers the message is delivered only once. Topic is for the classic publish/subscribe scenario, where if no consumer exists the message is discarded and in case of multiple consumers the message system delivers it to each of them. A part from queue and topic which are widely supported, more complex delivery semantics exist at protocol level (e.g. exchange/nodes from AMQP) and many others are middleware-specific.

3.3. Protocols
The lack of a unique standard way to interact with message brokers has been a known issue for the messaging technology for many years. The AMQP protocol has been designed by the main messaging actors, companies and software producers, to overcome this limitation. Nevertheless, the inner complexity of a unified protocol which defines both wire communication and delivery semantics requires a major development effort for the messaging system to become fully compliant. This section presents an overview of the most common standard protocols which are supported today by the main messaging systems. The protocol choice is a crucial design decision for message-oriented architecture, in respect to the strong coupling it has within the application.

3.3.1. AMQP (Advanced Message Queuing Protocol) [1] is the result of a standardization effort by the major contributors in the messaging scene (e.g. Cisco, Microsoft, Red Hat, banks). It is designed for interoperability between different messaging systems. It provides the definition for a binary wire protocol and a complete delivery semantic, allowing, theoretically, for an AMQP messaging client to be able to interact seamlessly with different brokers implementations which are AMQP compliant. Nowadays, the adotion of the latest stable version (1.1) of the protocol is not yet extensive, but given that it is already supported by the major message brokers, a much wider implementation is expected in the upcoming years.

3.3.2. STOMP (Streaming Text Orientated Messaging Protocol) [2] is a text-based protocol meant to be simple and widely-interoperable. It is mainly a wire protocol, it comes with very basic messaging semantic built-in (e.g. no support for communication models, the destination is just a string message header) , requiring appropriate configuration at the message system level (e.g. the destination have to be appropriately mapped to a queue or a topic). Thanks to its simplicity, there is an extensive set of clients available in many languages and it is supported by the majority of brokers.

3.3.3. MQTT (Message Queue Telemetry Transport) [3] is lightweight protocol designed originally from IBM. It is meant for low bandwidth, high-latency networks. It defines a compact binary format with very limited overhead on the communication, few tens of Bytes, which makes it suitable for *Internet of Things* style applications (e.g. mobile phones, sensors) in a simple *produce-and-forget* scenario.

3.4. Capabilities

As introduced in Section 2, a messaging system can be thought as a intermediate channel which is enhanced with additional features, such as queuing, to improve the communication experience of remote entities. Over the years, although no formal agreement exists, the different messaging systems converged over a common set of capabilities which become de-facto standard for messaging middleware. The list of features includes *Persistence*, which is the ability to save message on persistent storage, such as file-system or database; *Fail-over*, which allow clients to automatically reconnect in case of broker failure; *Guaranteed delivery*, which defines the policy for the message delivery (e.g. *at-least* once or *exactly-once*); *Ordering*, to deliver messages in the order they are produced; *Transaction*, the ability to consider multiple requests as part of a distributed transaction, with roll-back option and *Clustering*, which is the possibility to create network of message brokers for high-availability and load-balance. Nevertheless, each messaging system may provide different interpretation for the same features. Many other *unique* broker-specific features exist, but their usage imply hard coupling the application with a specific broker flavour.

4. Messaging technology

Message-oriented middleware has been developed for more than a decade to what today is a rich and solid ecosystem of services and libraries. Message brokers, as intermediate standalone services which offer messaging capabilities to distributed applications, are the most common type of messaging systems. Message brokers have been extensively used over the years for implementing communication and integration in distributed system [4], with the exception of data-intensive and high-performance use cases, where the existence of an intermediate entity is not a suitable option. In recent years, a new generation of messaging systems has appeared, with a focus on low-latency and high-performance use cases, pushing the boundaries of messaging applications. The next section is going to present an overview of the main messaging technology.

4.1. Message brokers

Message brokers are the most common implementation of messaging system. A message broker is standalone intermediate entity which offers messaging functionality via standard or custom protocols. Many message brokers exist, different in capabilities, protocols, implementation languages, platform support. The focus of this review is on open-source solutions, but many exist as part of enterprise commercial software too.

Message brokers are the most feature-rich type of messaging system, in term of capabilities an protocol support as described in Section 2. Brokers can be *polyglot*, allowing producer and

consumer to use different protocol (e.g. sender over AMQP, receiver over STOMP) and they can support *message transformation* (e.g. transforming message payload from XML to JSON) to further decouple applications.

4.1.1. ActiveMQ is one of the most widely adopted open-source message broker. It is an Apache project, it is written in Java and it is commercially supported by Red Hat. ActiveMQ has an extensive protocol support (e.g. AMQP, STOMP, MQTT, Openwire, HTTP and many others), it provides many cross language clients and it is fully JMS compliant. ActiveMQ offers many advanced capabilities, such as rich deliver semantic (e.g. virtual queues, composite destination, wildcards), JDBC message store (e.g. to persist messages in any JDBC compliant database) and advanced clustering configuration (e.g. master-slave, network of brokers). ActiveMQ is a feature-complete messaging solution, which can be used to implement many communication and integration patterns [4].

4.1.2. RabbitMQ is a lightweight open-source message broker written in Erlang which profits from the message passing capabilities from the language underneath. RabbitMQ architecture is deeply modular, it mainly supports AMQP and STOMP but additional protocols can be loaded as plug-in (e.g. MQTT, HTTP). It supports the main messaging capabilities, such as persistence, clustering, high-availability and federation. RabbitMQ remains a lightweight messaging solution which can be found embedded into several projects (e.g. Logstash) for its simplicity and reliability.

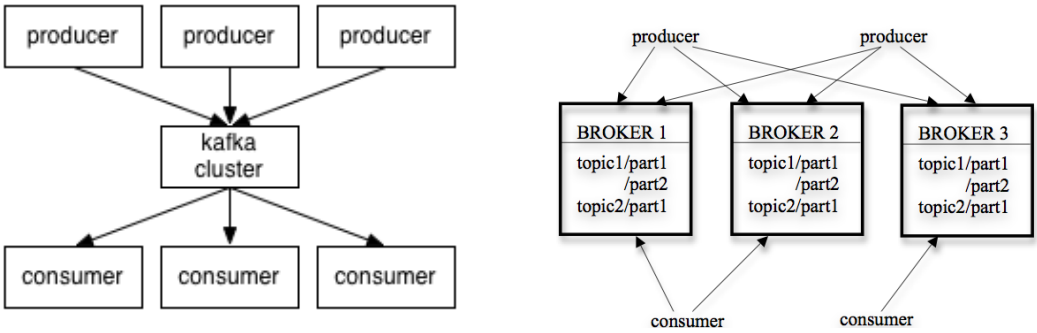
4.1.3. Performance and Scalability For messaging systems, the quantitative measure of *messages per second (msg/s)* has very little meaning without detailed contextualization. The protocol used (e.g. binary or textual), plays a major role, but many other latency factors exist: *persistent* messages can be orders of magnitude slower, the *amplification factor* (e.g. the number of topic consumers) can impact the system with multiple in-memory copy of the message, and the same is true for the *payload size*. Suboptimal clients can lead to high number of *open/close connections* and misbehaving consumers can lead to the *low-subscriber* problem, one of the most common issue for messaging infrastructure.

The comparison presented in [5], where several message brokers are evaluated via the STOMP protocol in several communication models, shows how, within realistic scenario, the performance may vary across 100000 msg/s and 1000 msg/s.

4.2. Apache Kafka

Apache Kafka is an open-source project originally from LinkedIn, now part of the Apache foundation. It has been developed for *real-time* activity stream analytics, to solve the need for an effective way to move big amount of data (e.g. user metrics, computer farm monitoring) from the producers to many potential consumers. The scale and the data size (billions of messages and hundreds of gigabytes per day) and the time constraint makes the use case not suitable for standard brokers, as by the comparison in [6].

The innovative idea of Kafka is to be a stateless broker, so to not retain any information about consumers. A consumer has to retain its own state (e.g. the information about the last data read) and poll Kafka for new data when needed. This allows Kafka to persist a single message copy independently from the number of consumers (e.g. messages are not removed on consumption, but by retention period or other policy), with a resulting high-throughput for read and write operations. Kafka persistence is implemented as a distributed commit log, as shown in Figure 3, designed as distributed system easy to scale out (based on Zookeeper), which allows for automatic balancing of consumer/producer/broker.



(a) Kafka cluster (b) Kafka topic partitioning

Figure 3: Kafka architecture.

In contrast with standard message brokers, Kafka provides limited messaging capabilities (e.g. mainly topic semantic, file-system as unique persistent storage, strict guaranteed ordering). Although many client libraries are available, it only supports its custom binary format over TCP. Kafka is an optimal solution for *data-movement*, frequently adopted as *pipe* to different processing systems (e.g. Hadoop, Storm).

4.3. ZeroMQ

Despite the name, ZeroMQ (also known as 0MQ or ZMQ) [7] is not a standard message broker but a lightweight messaging library which provides messaging capabilities. Distributed applications can use ZeroMQ for high-throughput and low-latency communication, profiting from its ability to implement direct connection among producer and consumer, with no intermediate entities involved. Although this may seems in contradiction with one of the main assumption of messaging, ZeroMQ implements loosely coupled communication via an innovative approach, acting as a new layer on the networking stack. It expands the concept of socket with a similar API but enhaced with built-in messaging patterns: *Request/Reply*, *Publish/Subscribe*, *Pipeline* and *Exclusive pair*, as presented in Figure 4. In contrast with classic socket, each ZeroMQ socket comes with an internal queue to allow for asynchronous communication. The result is that, for example in a request/reply scenario used for point to point communication, if the data is produced when the consumers is not running, the ZeroMQ library will take care of deferred delivery with no additional load on the producer side.

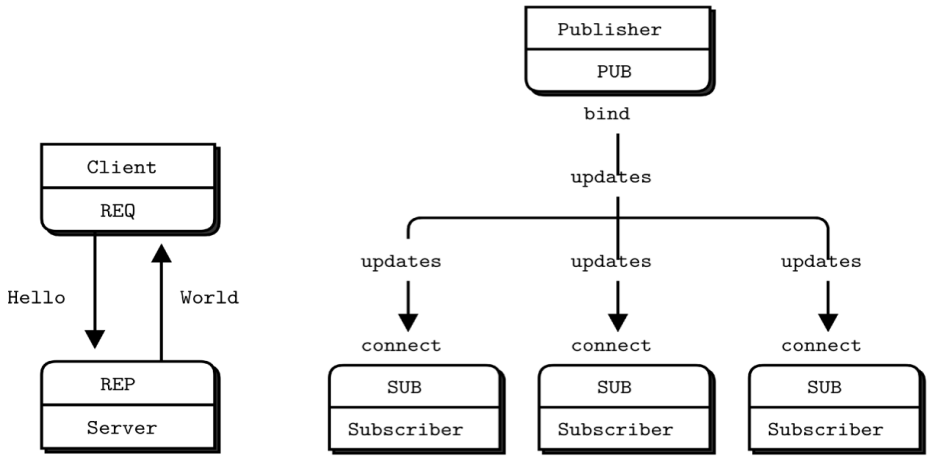
The idea behind ZeroMQ is powerful, it allows for high-performance and low-latency communcation, but comes with additional complexity at the application level. ZeroMQ mainly supports its own binary protocol and provides limited messaging capabilities (e.g. failover, multicast support for 1-N topology). Although several features can be easily implemented using ZMQ API (e.g. acknowledgement), implementing advanced messaging capabilities (e.g. guaranteed delivery, persistence) may require considerable effort, making it suitable for data-intensive scenario where simple messaging semantic is needed.

5. Use cases

This section presents several use cases where messaging-based communication has been successfully adopted to solve the problem of exchange information in distributed system.

5.1. CERN Beam Control middleware

The Beam Control department at the CERN laboratory is using messaging for highly reliable control/monitoring/alarm applications for the Large Hadron Collider (LHC). Since 2005, a



(a) Request/Reply (b) Publish/Subscribe

Figure 4: Examples of ZeroMQ sockets.

cluster of ActiveMQ brokers, in a store and forward configuration, is used to collect the critical data generated by the safety systems (e.g. 30 producers, 2MB/s, 4.5K msg/s) and to forward it to many consumers (e.g. monitoring tool, dashboards). Being safety data mission critical, the store and forward configuration allow to completely decouple data production from consumption, preventing misbehaving clients to affect data collection and archiving [8]. Moreover, the LHC Control framework has been recently migrated from CORBA to ZeroMQ as communication layer [9].

5.2. DAQ Online Monitoring

Messaging has been also extensively used in several monitoring tools for Data Acquisition (DAQ) systems, which are responsible to filter and collect data from detectors (e.g. high energy physic experiments) to storage facilities.

5.2.1. The ATLAS TDAQ shifter assistant project [10] relies on messaging to distribute operational alarms from private TDAQ network to GPN to a number of heterogeneous consumers. An ActiveMQ cluster is used in a master/slave configuration in order to minimize the impact on the required firewall configuration to a single outbound connection.

5.2.2. The STAR Online framework relies on an AMQP-based system for flexible, loosely-coupled distribution of detector metadata, using messaging as unified transport layer for processing, storage and monitoring. Moreover, investigation has been done to re-write the control framework over MQTT, profiting from the protocol flexibility and interoperability [11].

5.3. WLCG Messaging Service

Messaging has been also successfully used on large-scale geographically distributed infrastructure. The WLCG (Worldwide LHC Computing Grid) messaging service is the backbone transport layer used for monitoring WLCG sites and services around the world, with more than 50000 clients and an average message rate of 100 KHz. The monitoring infrastructure is based on STOMP with JSON payload. Thanks to the interoperability of the STOMP protocol across several broker flavours, heterogeneous message-broker clusters (ActiveMQ, Apollo or

RabbitMQ) are used in a scenario where client applications produce to any and consume to all [12].

6. Summary

Messaging is fundamentally a pragmatic reaction to the problem of distributed systems [4]. As presented in Section 2, it allows loosely coupled communication acting as intermediate layer between producer and consumer. It brings many benefits in distributed applications flexibility and scalability, with implications in application and infrastructure complexity. Messaging systems are still evolving technology, as shown in Section 3, with the AMQP standardization effort pointing in the good direction, but still with partial adoption. Message brokers are solid and reliable technology used as transport layer building blocks in many projects and services, both within the physics community and outside. In the recent years, a new generation of systems is promoting messaging for low-latency / high-throughput / data-intensive communication, as presented in Section 5, narrowing use cases and relaxing assumptions, but pushing the boundaries of messaging applications towards new domains.

References

[1] AMQP (Advanced Message Queuing Protocol) http://www.amqp.org
[2] STOMP (Simple Text Oriented Messaging Protocol) http://stomp.github.io
[3] MQTT (MQ Telemetry Transport) http://mqtt.org
[4] G Hohpe and B Woolf 2003 *Enterprise Integration Patterns* Addison-Wesley Professional
[5] Chirino H *STOMP benchmark* http://hiramchirino.com/stomp-benchmark
[6] Kreps J, Narkhede N and Rao J *Kafka: A Distributed Messaging System for Log Processing.* NetDB Workshop (Athens, GREECE)
[7] Hintjens P *ZeroMQ: The Guide* http://zeromq.org
[8] Ehm F *Running a Reliable Messaging Infrastructure for CERN's Control System.* Proceedings of ICALEPCS2011 (Grenoble, FRANCE)
[9] Dworak A, Ehm F, Sliwinski W and Sobczak M 2011 *Middleware Trends and Market Leaders 2011.* Proceedings of ICALEPCS2011 (Grenoble, FRANCE)
[10] Kazarov A, Miotto G L and Magnoni L 2012 *The AAL project: automated monitoring and intelligent analysis for the ATLAS data taking infrastructure.* Journal of Physics: Conference Series, Volume 368
[11] Arkhipkin D, Lauret J and Betts W 2011 *A message-queuing framework for STARs online monitoring and metadata collection.* Journal of Physics: Conference Series, Volume 331
[12] Cons L and Paladin M 2011 *The WLCG Messaging Service and its Future.* Journal of Physics: Conference Series, Volume 396