Location independent inter-process communication as software buses

Erik Samuelsson
Abstract

Telecommunication networks will transform and gradually migrate into virtualized cloud environments as a result of the potential for higher profitability through reduced costs and increased revenues. The purpose of this thesis is to investigate architectural mechanisms for location-independent communication between software components in a virtualized base station. Systems that provide such mechanisms are typically referred to as middleware and deployed as Platform-as-a-Service (PaaS). The overall goal is to achieve desired characteristics in cloud deployment regarding on-demand self-service, rapid elasticity of capacity while upholding services and high availability. Four communication protocols are examined and evaluated based on a set of functional and non-functional requirements that are especially relevant for a virtualized base station. In comparison with the Advanced Message Queuing Protocol (AMQP), Message Queuing Telemetry Transport (MQTT) and the eXtensible Messaging and Presence Protocol (XMPP), the Data Distribution Service (DDS) standard is found to have excellent performance characteristics. Its complexity might have implications for the development and deployment though, that will increase the time it takes to reap the benefits from its advantages.
Acknowledgements

I wish to sincerely thank my supervisors at Tieto, Clas Högvall, Rickard Sjöström and Johan Forsman for introducing an interesting and challenging thesis idea and for entrusting me with the task. I would also like to thank them for spending so much of their time and energy to support and guide me.

I would also like to extend my sincere thanks to my supervisors Ewnetu Bayuh Lakew and Cristian Klein. Their encouragement, enthusiasm and guidance has been invaluable.
Contents

1 An introduction to base station virtualization and messaging systems 1

2 Requirements analysis 2

3 Middleware for distributed systems 5
   3.1 Protocols for Message Oriented Middleware (MOM) 6
      3.1.1 AMQP 6
      3.1.2 DDS 7
      3.1.3 MQTT 8
      3.1.4 XMPP 9

4 Results 10
   4.1 Functional requirements 10
   4.2 Non-functional requirements 12
      4.2.1 Messaging model 12
      4.2.2 Security and extensibility 12
      4.2.3 Scalability and reliability 13
      4.2.4 Interoperability, efficiency and usability 13

5 Conclusions 14
   5.1 Future work 15
1 An introduction to base station virtualization and messaging systems

Telecommunication networks will transform and gradually migrate into virtualized cloud environments as a result of the potential for increased profitability it provides by reducing the cost of equipment and increasing revenue [1]. This transformation will have a dramatic impact on the whole industry and it will create new challenges that must be addressed and solved. Telecom infrastructures have high requirements on performance since this has a direct impact on revenue streams and customer churn.

In this report a survey of different existing architectures for message queuing systems that provides inter-process communication between processes in a virtualized environment will be performed. After the initial survey, four existing communication standards (AMQP, DDS, MQTT and XMPP) are selected and examined more carefully in a comparative study with regards to functional and non-functional requirements in the telecom industry.

The inter-communicating processes considered in this report are virtualized network functions. Network Function Virtualization (NFV) is a concept that originates from a white paper by the European Telecommunications Standards Institute (ETSI) [2] in 2012 that involves the implementation of network functions in software that can be instantiated and executed at various locations in a network. This allows execution of network functionality at data centers with high processing and storage capabilities, and in essence the decoupling of the functions. The virtualization of network functions aims to reduce the need to populate a network with hardware that results in increased costs of energy, maintenance and hardware turnover, as well as the complexity of integrating hardware by various proprietors.

Some examples of network functions that have been virtualized are network address translation (NAT), firewalls, intrusion detection and domain name service (DNS). These virtualized network functions must maintain the capability to communicate with other software components in the network, independently of where the other components might be instantiated logically or physically. Parts of the network infrastructure that facilitates sending and receiving data between software components is commonly referred to as middleware, which is a software layer on top of the network operating system that provides a layer of abstraction for software developers.
2 Requirements analysis

In order to decompose a base station into decoupled components and achieve desired characteristics of cloud computing, like rapid elasticity (i.e. scalability of processing power, storage and other resources), there must be architectural mechanisms in place for inter-process communication between these components. On a high level abstraction components can be said to offer their services on a software message bus (see Figure 1). Other components request services on the same bus. This allows for interoperability and integration of software components independent of platform. Services can be scaled up by replicating instances of particular components since there is no need for users of the services to know which particular instance provides the service at any given time.

![Diagram of software components communicating through a software message bus](image)

**Figure 1:** Software components communicating through a software message bus

In this chapter, a number of requirements that might be imposed on a software message bus in a telecom environment are presented. The requirements are divided into functional and non-functional requirements, which are considered separately. Functional requirements defines the necessary behavior of a system such as the capability to deliver messages from point A to point B. The functional requirements are complemented with a set of non-functional requirements that specify qualitative criteria to compare and evaluate systems that fulfill the functional requirements.

The functional requirements that are placed upon the middleware are enumerated in Table 1. These functional requirements have initially been authored by Johan Forsman and Rickard Sjöström at Tieto, to express their specific needs of a messaging bus. An essential requirement is the asynchronous communication model, illustrated in Figure 2. In the asynchronous communication model a producer pro-
duces a message, sends it to the message bus, and is able to continue execution immediately thereafter. In other words the function call to send a message must be non-blocking, until a reply is delivered from the message bus by either Pushing or Pulling mechanisms. This is in contrast to a synchronous messaging model which is blocking, where the producer is suspended while it waits for a reply.

![Asynchronous interaction model](image)

**Figure 2:** Asynchronous interaction model

In addition to the functional requirements, the following non-functional requirements have been considered in order to differentiate between middlewares:

- **Security** The ability to defend against threats that could jeopardize its functionality or its users integrity.
- **Extensibility** The ability to add customized features to the system.
- **Scalability** The ability to handle increased amount of work.
- **Reliability** The ability to resist/recover from failures.
- **Interoperability** The ability to integrate other products/systems without restrictions.
- **Efficiency** The amount of required resources in terms of CPU load, memory consumption and disk storage.
- **Usability** The ease of use for users of the system, i.e. developers.
<table>
<thead>
<tr>
<th>Table 1 Functional requirements</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application components (FR1)</strong></td>
<td>The Application Components are designed to provide one or more services. Application Components communicate using asynchronous messages.</td>
</tr>
<tr>
<td><strong>Message sending (FR2)</strong></td>
<td>A message is sent to a destination Application Component identified by a destination Address Point.</td>
</tr>
<tr>
<td><strong>Address Point (FR3)</strong></td>
<td>Address Point names shall be unrelated to the Application Component Name or task name.</td>
</tr>
<tr>
<td><strong>Redeployment (FR4)</strong></td>
<td>No change shall be needed in the code when an Address Point (service) is redeployed.</td>
</tr>
<tr>
<td><strong>Singleton Address Point (FR5)</strong></td>
<td>Singleton Address Point shall be supported, i.e. one Address Point with a unique name in the system.</td>
</tr>
<tr>
<td><strong>Local Address Point (FR6)</strong></td>
<td>Local Address Point shall be supported, i.e. the scope of the registered Address Point is restricted to a specific domain, e.g. local CPU.</td>
</tr>
<tr>
<td><strong>Multiple Instantiated Address Points (FR7)</strong></td>
<td>Multiple Instantiated Address Points shall be supported, i.e. the Address Point exists in several instances, distributed over all CPUs. Each instance is uniquely identified by a qualifier. The qualifier is a string.</td>
</tr>
<tr>
<td><strong>Address Point Lookup (FR8)</strong></td>
<td>The Application Components shall be able to lookup an Address Point. The lookup blocks while waiting for the Address Point to appear in the system.</td>
</tr>
<tr>
<td><strong>Address Point Discovery (FR9)</strong></td>
<td>The Application Components shall be able to discover for an Address Point. The discovery of an Address Point is asynchronous, i.e. the Application Component is notified each time a matching AP is registered in the system.</td>
</tr>
<tr>
<td><strong>Address Point Supervision (FR10)</strong></td>
<td>It shall be possible for an Application Component to supervise the logical connection to an Address Point. The Application Component is notified when the logical link is lost, e.g. due to physical link loss, CPU crash, task is killed.</td>
</tr>
</tbody>
</table>
3 Middleware for distributed systems

Mascolo, Capra and Emmerich [3] has proposed that middleware for fixed distributed systems can be classified into three categories; object-oriented middleware (also known as Object Request Broker or ORB), message-oriented middleware (MOM) and transaction-oriented middleware (TOM).

ORB-based middleware and its related architecture standard, Common Object Request Broker Architecture (CORBA), evolved from Remote Procedure Calls. It is based on a client-server model, and allow applications to make method calls directly on objects instantiated in a remote application. The style of communication in an ORB-based system is synchronized (i.e. blocking) [4, 3]. This puts a limit on the scalability of such systems, and experience in the industry has shown that ORB-based solutions are not feasible for current needs. ORB implementations are therefore not considered further in this report.

Transaction-oriented middleware (TOM-based middleware) supports asynchronous messaging and transactions very well by utilizing the two-phase commit protocol, and is primarily used where components are database applications. Its disadvantage is that it causes unnecessary overhead for messages that does not require to be handled as transactions [5, 3]. Since virtualized network functions are not typically huge database-applications TOMs are not investigated further in this report either.

Message oriented middleware (MOM-based middleware) is designed for event-based interaction and the asynchronous interaction model is central in a MOM system. In a MOM, clients send messages to a central message queue where the messages are stored until they have been processed [6, 7]. However, a consequence of the asynchronous communication is that a client that sends a request has no way of knowing if or when it will receive a response. There is no guarantee in the model that the message will be delivered at all.

For instance, physical network errors, congestion, or crashes of the software components that are involved in processing a request are possible errors that might occur. One method MOMs use to handle this is by storing messages in the queue combined with the use of message acknowledgements [6]. The queue can be configured with regards to how messages are stored (in memory/on disk) and how many retry attempts it will perform upon delivery failure, among other configurations.

Curry [6] categorizes MOMs into two basic communication paradigms: Point-to-Point and Publish/Subscribe. The common denominator is the message queue which provides the basis for asynchronous nonblocking communication.

- In the Point-to-Point message model, while there may be more than one consumer per queue, every message is consumed and processed by only a single receiver/consumer.
- In the **Publish/Subscribe message model** clients publish messages to a queue which is labeled with a *Topic*. Consumer clients subscribe to topics it wishes to consume. Contrary to the Point-to-Point model, the same message can be received by multiple consumers.

### 3.1 Protocols for Message Oriented Middleware (MOM)

There exists a variety of communication protocols for MOMs and message queuing services to achieve platform neutrality. Alternatives include the Message Queuing Telemetry Transport (MQTT) protocol, Data Distribution Service (DDS), and eXtensible Messaging and Presence Protocol (XMPP) and the Advanced Message Queuing Protocol (AMQP) [8].

According to Albano et. al [9] Publish/Subscribe MOMs can be subdivided into *Topic-oriented*, *Content-oriented* or *Data-oriented*. They go on to suggest that the protocol XMPP is tightly linked to Point-to-Point modeled systems, while AMQP is linked to the publish/subscribe model and DDS is best suited for data-oriented interaction. Note however that one of the motivations behind a communication protocol is interoperability and there are MOM implementations that are somewhat protocol agnostic by the same rationale. An implementation may accept messages in one of several formats in order to not impose assumptions or restrictions on software developers and applications that use the middleware.

As an example it is possible to have a scenario where a mobile device sends requests to a distributed system through MQTT. The request is received and routed by the middleware to a distributed software component that processes the request. While processing the request, the component requests other services in the system through messages in AMQP format, before a response is sent back to the original sender of the request as an MQTT message.

#### 3.1.1 AMQP

AMQP is a protocol that originates from the work by John O’Hara at JP Morgan Chase & Co as a solution to the lack of interoperability between proprietary applications in banking and financial software [7]. It was designed to become a substitution for an array of messaging systems for various use cases that were incompatible or difficult to integrate with each other. Due to the requirements of the financial industry it had to be able to handle a huge volume of business transactions in a reliable way. In the AMQP model clients send and receive messages through a *message broker*. The broker utilizes two fundamental AMQP entities: *exchanges* and *queues*. Senders publish messages to exchanges. The broker then distributes them to different queues through a set of rules called *bindings*. Message consumers interact with the broker through these queues. A consumer either subscribes to specific queues and get messages from those queues pushed to them, or pull messages from the queues on demand [10]. This conceptual model is visualized in Figure 3.

The concept of exchanges allows the broker to apply different types of communication patterns. Through this mechanism it is able to provide flexibility for different
use cases. The AMQP model provides four exchange types: direct, fanout, topic and headers exchange. The exchange type determines whether the message will be delivered point-to-point or as publish-subscribe.

- A Direct Exchange emulates a point to point message delivery. A message is labeled with a routing key in the message header, and the broker routes the message to a queue with a matching key. This exchange type can be used to implement basic round robin load balancing between consumers.

- In a Fanout Exchange the routing key of a message is ignored, and an incoming message will be copied to every queue that is bound to that exchange. This emulates a one-to-many broadcasting communication pattern where all consumers will receive a copy of the incoming message.

- A Topic Exchange implements the Publish/Subscribe pattern. Messages are routed to queues by pattern matching the routing key, similar to a direct exchange. The difference is that the routing keys may contain multiple words and the bindings may contain wildcards.

- In a Headers Exchange messages are routed based on queries on attributes that are expressed in the message header instead of the routing key. A queue can be bound to an exchange with a list of header names and properties, and can subsequently be matched either to any or all fields.

### 3.1.2 DDS

The Data Distribution Service (DDS) is a standard for publish-subscribe communication that is designed to be data-oriented and uses a Data Centric Publish Subscribe (DCPS) model [9, 11]. In essence, clients share memory-access to a distributed global data space. Subscribers only have read-access while publishers have both read and write privileges. As such it utilizes a brokerless architecture based on reliable UDP multicasting, which makes it very well suited for real-time systems which require low latency and propagation of messages to a large amount of recipients [12, 13]. As Stan Schneider puts it [14]:

![AMQP semantic model](image)
"DDS, with its support of reliable UDP multicast and flexible QoS, shines at ‘fanout’ scalability”

The data in the global data space is stored in structures which are labeled with a Topic and a Type. When publishers write data to a topic, the middleware is responsible for data verification and notifying all subscribers to that topic of the update [9]. A significant characteristic of DDS is also its policies for Quality of Service (QoS) [9, 13]. In DDS, each topic is associated with a set of QoS policies, which provides extreme flexibility. The QoS policy can be configured for a large array of attributes such as reliability, durability, priority and timeliness to name a few. Figure 4 depicts a conceptual overview of the DDS model.

3.1.3 MQTT

The MQTT protocol is designed to have as small a footprint as possible with minimal overhead, for use within constrained devices with regards to storage or computation power and in networks characterized by low bandwidth and high latency. It is well suited for sensors, mobile devices and wearable technology and was designed with that in mind [15, 16, 17]. However, it suffers from a lack of security features [18, 16]. MQTT’s origins can be traced back to 1999 and became an ISO-standard in 2013 (ISO/IEC 20922) [19, 20]. A conceptual overview of the MQTT entities and architecture is depicted in Figure 5.
3.1.4 XMPP

The Extensible Messaging and Presence Protocol (XMPP), which is based on XML, was primarily designed for instant messaging on the web [21, 12, 9]. Architecturally it passes messages between clients through a client-server model. While it is considered an improvement for web-applications, especially for instant messaging, it is not considered a contender for communication between components in a distributed system. In a benchmark of machine-to-machine protocols Talaminos-Barroso et. al [22] conclude that XMPP is not suitable for applications that require low latency and high predictability due to its high CPU usage and bandwidth consumption. Likewise, Albano et al [9] finds XMPP lacking in terms of scalability and latency, although they point out the possibility to extend the system with an Extension Protocol (XEPs). One of the weaknesses of XMPP is that it is text-based which causes the protocol to suffer from a higher overhead than binary protocols.

Figure 6 shows a visualization of the XMPP communication model. Clients connect to decentralized servers that can communicate for routing messages between domains. It is based on a point-to-point message-passing model, but can implement the publish/subscribe pattern with extension XEP-0060.

![XMPP Communication Model](image_url)

**Figure 6:** The XMPP communication model.
4 Results

This chapter contains comparisons of the MOM communication protocols (AMQP, DDS, MQTT and XMPP) with regards to the functional and non-functional requirements that were presented in chapter 2.

4.1 Functional requirements

The communication protocols’ support of the functional requirements are presented in Table 2.

<table>
<thead>
<tr>
<th></th>
<th>AMQP</th>
<th>DDS</th>
<th>MQTT</th>
<th>XMPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application components (FR1)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Message sending (FR2)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Address Point (AP) (FR3)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Redeployment (FR4)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Singleton AP (FR5)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Local AP (FR6)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MultipleInstantiated AP (FR7)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>AP Lookup (FR8)</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>AP Discovery (FR9)</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>XEP-0030</td>
</tr>
<tr>
<td>AP Supervision (FR10)</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>XEP-0184</td>
</tr>
</tbody>
</table>

The first seven functionalities are present in all protocols, they are essential for the MOM model. All protocols allow components (FR1) to send asynchronous messages (FR2) to other components.
The *topic* in any of the models can be regarded as the Address Point, and the topic’s name is unlinked to an individual consumer’s name (FR3). New components/consumers can register to topics in the messaging system on the fly (FR4), and each topic is uniquely named (FR5). After an application component registers in the middleware, the middleware is subsequently responsible for keeping track of clients through some type of client identifiers (FR6). Multiple consumers are able to subscribe to and consume messages under the same topic (FR7), which in some cases allow the middleware to perform basic load balancing techniques.

The last three requirements (FR8-FR10) is where the standards show their different approaches. AMQP does not support blocking address point lookup (FR8) for application components in its protocol specification. Instead of a lookup method, AMQP messages can be marked with the flags *mandatory* and *immediate* when they are published, which dictates how the broker handles unroutable messages (either return the message to the publisher or silently drop the message). It is then up to the producer to decide on future action. Delivery failures are normally handled by setting a timer and retrying until the message is routed successfully, or defining some custom exchange type that receives and retains unrouted messages at the broker. Address point discovery (FR9) or supervision (FR10) for application components is not part of the AMQP specification.

DDS, conversely, supports both address point lookup (FR8), discovery (FR9) and supervision (FR10). Blocking lookup is accomplished through *WaitSets* [23]. Every DDS entity has a set of *statuses* associated with it, and by first indicating statuses of interest followed by attaching it to a *WaitSet*, the process can make a method call to the *WaitSet’s* *wait()* function which blocks until there is a change in the status of interest. Address point discovery is supported through the use of a mechanism called *Simple Discovery Protocol* (SDP). In short, every client maintains a database of information about topics, publishers and consumers which is updated according to the SDP. Address point supervision is achieved with the *Liveliness* QoS Policy.

MQTT does not provide address point lookup (FR8), discovery (FR9) or supervision (FR10). Like in AMQP, it must be accomplished through some workaround. Best practice to achieve discovery and supervision is to setup a dedicated topic and have clients publish a notification to that topic when it connects. Every client that subscribes to this topic will then be notified when clients connects and disconnects. Notice however that this mechanism is not part of the protocol specification.

The XMPP protocol does not support synchronous address point lookup (FR8), discovery (FR9) or supervision (FR10) in the core protocol either. Like AMQP and MQTT these functionalities must be implemented on top of the protocol itself, and for XMPP there exists a lot of available extensions to pick and choose from. The extension XEP-0030 defines a protocol extension for address point discovery. To achieve address point supervision the XMPP servers can utilize TCP-heartbeats, and on the client-side XEP-0184, XEP-0198 and XEP-0199 defines protocol extensions for message acknowledgment and XMPP Ping.

---

1Some AMQP brokers supply protocol extensions and/or plugins that provide replacements for some of these functionalities, for example RabbitMQ’s *Dead Letter Exchange* (https://www.rabbitmq.com/dlx.html), *Consumer Cancellation* (https://www.rabbitmq.com/consumer-cancel.html) and *Management plugin* (https://www.rabbitmq.com/management.html)
### 4.2 Non-functional requirements

A comparison of communication protocols with regards to the non-functional requirements are presented in Table 3.

<table>
<thead>
<tr>
<th>Message model</th>
<th>AMQP</th>
<th>DDS</th>
<th>MQTT</th>
<th>XMPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messaging model</td>
<td>Pub/Sub</td>
<td>Pub/Sub</td>
<td>Pub/Sub</td>
<td>Pub/Sub</td>
</tr>
<tr>
<td>Security</td>
<td>SASL/TLS</td>
<td>TLS/DTLS/DDS Security</td>
<td>Must be implemented independently</td>
<td>SASL/TLS</td>
</tr>
<tr>
<td>Extensibility</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Scalability</td>
<td>Moderate</td>
<td>High</td>
<td>Moderate</td>
<td>Low</td>
</tr>
<tr>
<td>Reliability</td>
<td>Moderate</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Interoperability</td>
<td>Yes</td>
<td>Yes</td>
<td>Partial</td>
<td>Yes</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Moderate</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Usability</td>
<td>Moderate</td>
<td>Complex</td>
<td>Simple</td>
<td>Simple</td>
</tr>
</tbody>
</table>

#### 4.2.1 Messaging model

All the protocols in this report are based on the publish/subscribe messaging model. This is essential to allow more than one consumer to receive the same message. However, all protocols support direct point-to-point communication as well, in the form of private topics set up for direct communication between two clients. Note however that there is no protocol enforcement that prevents other parties to subscribe to a private topic, which could have implications if the message payload is not encrypted.

#### 4.2.2 Security and extensibility

AMQP and XMPP both feature support for SASL\(^2\) and TLS\(^3\) in their respective specifications. For DDS, support for TLS is not part of the standard, but it defines a Security Model and a Service Plugin Interface (SPI) architecture with five plugin SPIs: authentication, access-control, cryptographic, logging, and data tagging. Most vendors provide built-in security features. MQTT allows for basic username/password verification, but encryption must be implemented independently.

AMQP is an extendable protocol which, for instance, allow users to define custom exchange types, with many open source extensions available. DDS itself is not extensible per se, but its data-centric approach supplies an extensible type system as defined by the user. MQTT is not an extensible protocol, its goal is simplicity. XMPP, as its name suggests, is extensible to a very high degree, and there exists a multitude of implemented extensions.

---

\(^2\)Simple Authentication and Security Layer

\(^3\)Transport Layer Security
4.2.3 Scalability and reliability

Scalability and reliability are the strong suit of DDS. It excels at fanout message distribution over UDP and achieves reliability through an advanced QoS policy-scheme. AMQP was specifically intended for use cases which require high scalability and reliability. AMQP achieves scalability through the form of clustered brokers (i.e. deployment of additional instances of one and the same broker). However, it does not match the scalability of DDS and in edge cases even the reliability has been questioned in certain network topologies (see the experiences of [24] for example). XMPP was not designed with high throughput as a priority.

AMQP, MQTT and XMPP all depend on TCP as a reliable underlying Transport Layer. On the application layer they have in common that they provide very basic QoS support for message delivery. Through message acknowledgements (ACKs) message delivery can be set to at-most-once, at-least-once or exactly-once. AMQP also supports durable exchanges, queues and messages that survive broker restarts. DDS has a rich set of more than 20 QoS policies that can be configured to control the data flow. It can be implemented on top of either TCP or UDP, providing extreme flexibility.

4.2.4 Interoperability, efficiency and usability

All mentioned protocols are interoperable and can be implemented across platforms. One caveat is that MQTT does not define the data layout in the message payload. This means that publishers and subscribers must establish the format of the message body beforehand. MQTT excels at operating in resource-constrained environments. DDS with its data-centric model is comparatively memory-intensive, while XMPP has a high CPU usage.

MQTT and XMPP are very simple protocols to implement. XMPP, however, is largely dependent on extensions which in turn increases the complexity and diminishes its rationale in the first place. On the other spectrum is DDS. It provides extreme flexibility and performance at the cost of complexity.
5 Conclusions

This report presents a survey of different architectures for middleware to achieve location-independent inter-process communication for a distributed system. Out of the technologies and architectures discussed in this document, message-oriented middleware is found to be the architecture of choice over object-request-broker-based and transaction-based middleware.

A comparative study of four existing communication standards that implements a message-oriented middleware shows that the DDS standard appear as the only solution that meets all the functional requirements, as well as the non-functional requirements of high performance, security and reliability of the telecom industry to the highest extent. Its flexibility has implications on the usability however, in terms of that the configuration is complex which increases its ”time to market”, meaning that the cost of development and deployment will be higher until the system is ready for a production environment. XMPP and MQTT, contrarily, excels at simplicity, a property that is not the most essential in a telecom setting.

AMQP emerges as a balanced alternative which is very capable with relatively quick and easy integration compared to DDS. In cases where an AMQP-solution is able to match the throughput requirements it would be preferable over DDS thanks to a quicker implementation at a lesser cost. One might notice that AMQP does not fulfill the explicit requirements of address point lookup, discovery or supervision though. However, the message-oriented middleware systems are designed to decouple producers and consumers of data, which motivates a revisit of the idea that the application components themselves are left with these responsibilities. It is possible to utilize alternative mechanisms, like wildcard topic subscriptions instead of address point discovery while leaving the supervision of logical connections to the middleware itself, to achieve the desired properties of the system.
5.1 Future work

The next immediate step of the work started in this paper is to benchmark different implementations of primarily DDS or AMQP. Relevant benchmarks could initially be message throughput with varying amount of publishers/subscribers and message sizes, latency, jitter (variance in latency) and CPU and memory utilization. There is a multitude of both open source and proprietary software that would be interesting to benchmark. For DDS, a good start would be to test the open source alternatives OpenSplice and OpenDDS, together with a proprietary software like RTI Connext DDS. For AMQP, RabbitMQ is one of the most popular open source implementations which could be compared with Apache Apollo and the proprietary StormMQ.

Another interesting project to look into is Apache Kafka, an open source message broker originally developed by LinkedIn. It implements its own messaging protocol, so one might question how future proof it is, but at a glance it seems to be able to match and exceed the performance characteristics of DDS implementations while maintaining a simple and easy-to-use interface.
References


