PRISMTECH

Messaging Technologies for the Industrial Internet and the Internet of Things

A Comparison Between DDS, AMQP, MQTT, JMS, REST and CoAP

Version 1.2, November 2013

Andrew Foster, Product Manager

Table of Contents

Messaging Technologies Comparison

1. Definitions, Acronyms and Abbreviations

AMQP	The Advanced Message Queuing Protocol is an open standard application layer protocol for message-oriented middleware.
CDR	Common Data Representation (CDR) is used to represent structured or primitive data types passed as arguments or results during remote invocations on Common Object Request Broker Architecture (CORBA) distributed objects it is also used by the DDSI wire protocol for the representation of DDS data types.
CoAP	Constrained Application Protocol (CoAP) is a software protocol to be used in very simple electronics devices that allows them to communicate over the Internet.
DDS	The Data Distribution Service for Real-Time Systems is an Object Management Group (OMG) M2M middleware standard that aims to enable scalable, real-time, dependable, high performance and interoperable data exchanges between publishers and subscribers.
Industrial Internet	The Industrial Internet is the convergence of the global industrial system with the power of advanced computing, analytics, low-cost sensing and new levels of connectivity permitted by the Internet.
IoT	The Internet Of Things is a world where physical objects are seamlessly integrated into the information network, and where the physical objects can become active participants in business processes.
JMS	Java Message Service API is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients.
Java EE	Java Platform, Enterprise Edition is Oracle's enterprise Java computing platform.
MOM	Message-oriented middleware is software or hardware infrastructure supporting sending and receiving messages between distributed systems.
MQTT	Message Queuing Telemetry Transport is an open message protocol for M2M communications that enables the transfer of telemetry-style data in the form of messages from pervasive devices, along high latency or constrained networks, to a server or small message broker.
M2M	Machine to Machine refers to technologies that allow both wireless and wired systems to communicate with other devices.
OpenSplice	PrismTech's Open Source implementation of the The Data Distribution Service for Real-Time Systems standard.
QoS	Quality of Service refers to several related aspects of computer networks that allow the transport of traffic with special requirements.
REST	Representational state transfer is a style of software architecture for distributed systems such as the World Wide Web.

2. Executive Summary

The most important messaging technologies proposed as the foundation of the next generation of Internet of Things (IoT) and more specifically the Industrial Internet applications are reviewed in this document. An understanding of both the architecture and the message/data sharing requirements of each target system is an important pre-requisite for choosing the most appropriate messaging solution.

AMQP and JMS have been designed to address applications requiring fast and reliable business transactions. JMS is focused on Java-centric systems although there are a number of vendors who have developed proprietary C and C++ JMS API mappings that can be used with a JMS broker. As an API standard, JMS cannot guarantee interoperability between producers and consumers using different JMS implementations.

MQTT provides a simple and lightweight device data collection solution, although only partial interoperability between MQTT publishers and subscribers can be guaranteed. Messages can be exchanged between different MQTT implementations but unless the format of the message body is agreed between peers, the message cannot be un-marshaled.

REST provides a simple client-server (request/reply) style of communications that is useful for systems that need to communicate over the Internet, but it cannot provide asynchronous loosely coupled publish-and-subscribe message exchanges. The stateless model supported by HTTP can simplify server design, however the disadvantage of statelessness is that it may be necessary to include additional information in every request and this extra information will need to be interpreted by the server. This can be very inefficient is terms of request processing time and resources consumed (e.g. number of TCP/IP connections).

CoAP was designed to support the connectivity of simple low power electronic devices (e.g. wireless sensors) with Internet based systems. It can be used for data collection in systems that do not require very high performance, real-time data sharing or real-time device control. In many cases data is collected for subsequent “offline” processing. A CoAP device is connected to a cloud-based system via a HTTP proxy using a standard CoAP-HTTP mapping. Using a proxy/bridge adds an additional communication overhead and increases message latency.

Choosing AMQP, MQTT, JMS or REST for systems where a device needs to fan-out messages to perhaps thousands of other networked devices can result in poor performance and much complexity (e.g. requiring a multi broker configuration). CoAP however, supports IP multicast, enabling a single request to be issued to multiple CoAP devices concurrently.

Unlike DDS, which provides support for dynamic discovery, configuring a system that uses AMQP, MQTT or JMS is through the broker. Accessing the broker is usually via a well known network address or a lookup service such as JNDI. If the broker is moved to a different server then clients must be re-configured to use the address of broker’s new location.

Only DDS can provide the real-time, many-to-many, managed connectivity required by high-performance device-to-device applications. DDS is also emerging as a key interoperable messaging protocol for connecting real-time device networks to Cloud based Data Centers. Vendor specific implementations of DDS such as PrismTech’s OpenSplice DDS can also offer exceptional intra-nodal data sharing performance.

Ensuring that a system is fault-tolerant and secure is a key consideration in an IoT world consisting of potentially many thousands of devices all exchanging information. Most of the messaging technologies discussed in this document view security as an orthogonal issue to their core messaging functionality.

The leading vendor implementations typically provide proprietary solutions based on tried and tested third party security technologies such as SSL or TLS. AMQP specifies the use of SASL to provide a pluggable message authentication interface and the forthcoming OMG DDS Security Specification will standardize a comprehensive security framework for DDS-based systems.

Finally, it is worth mentioning that in some cases it may make sense (perhaps for legacy reasons) to create a system that uses more than one messaging technology within the same architecture. In this case, custom mediation schemes or protocol bridges to exchange messages between clients using different protocols are required. This may not provide the optimal solution but is a common scenario particularly as a system evolves and there is need to integrate new and legacy applications.

3. Introduction

Conceptually the Internet of Things refers to the general idea of things, especially everyday objects that are readable, recognizable, locatable, addressable, and/or controllable via the Internet; whether via RFID, wireless LAN, wide-area network, or other means. Everyday objects include not only the electronic devices we encounter everyday and not only the products of higher technological development such as vehicles and equipment, but also things that we do not ordinarily think of as electronic at all - such as food, clothing and shelter, materials, parts, and subassemblies.

This paper focuses on messaging technologies that are emerging as the most important for the Industrial Internet, a sub-set of the broader IoT that targets systems composed of thousands of devices connected over real-time machine-to-machine (M2M) networks.

The definition of the Industrial Internet includes two key components:

1. The connection of industrial machine sensors and actuators to local processing and to the Internet.

2. The onward connection to other important industrial networks that can independently generate value.

The main difference between the consumer/social IoT and the Industrial Internet is in how much value is created. For consumer/social Internets, the majority of value is created from advertisements. The value created from the Industrial Internet is much greater from the same amount of data, and has three components:

- The value of increased efficiency of the industrial plant equipment and long-term maintenance and management of the equipment. In research this is found to be always above 10% and as high as 25%.

- The value contribution to adjoining Industrial Internet networks, such as balancing short-term positive cash flow against additional long-term equipment costs such as maintenance.

- The value contribution of disruptive new business models is a wild-card that could, in a few cases, be dramatically high, or in most cases moderate or zero.

Industrial Internet application domains already using these technologies include: Aerospace and Defense (Air Traffic Control, Combat Management Systems, Modeling & Simulation, Vetronics), Healthcare (Smart Devices), Transportation (Railway Control Systems, Vehicle Management), Smart Energy (Large Scale SCADA Systems) and Smart Cities.

There are many different messaging technologies. However a much smaller subset are emerging as the

most important that will power the future Industrial Internet. These include:

- Object Management Group's (OMG) Data Distribution Service for Real-Time Systems (DDS)
- OASIS' Advanced Message Queuing Protocol (AMQP)
- MQ Telemetry Transport (MQTT) a proprietary protocol originally developed by IBM but now a proposed OASIS standard
- Java Message Service (JMS) an international messaging standard developed through the Java Community Process (JCP)
- Representational State Transfer (REST) a common style of using HTTP for Web-based applications and not a standard
- Constrained Application Protocol (CoAP) is a software protocol to be used in very simple electronics devices such as Wireless Sensor Networks (WSN) that allows them to communicate over the Internet.

This paper compares and contrasts the key architectural features of each messaging paradigm and provides a context for choosing the right technology to support the requirements for a particular type of system.

4. Background

4.1 DDS

The Object Management Group's DDS standard is a data-centric publish-and-subscribe technology that emerged from the Aerospace and Defense community to address the data distribution requirements of mission-critical systems. It enables scalable, real-time, reliable, high performance and interoperable data exchanges between publishers and subscribers. DDS is designed to address the needs of mission and business-critical applications like financial trading, air traffic control, smart grid management, and other big data applications. It is being increasingly used in a wide range of Industrial Internet applications.

The DDS specification defines:

- A Data Centric Publish Subscribe (DCPS) layer providing a set of APIs that present a coherent set of standardized "profiles" targeting real-time information-availability for domains ranging from small-scale embedded control systems right up to large-scale enterprise information management systems.
- A DDS Interoperability Wire Protocol (DDSI)
- An Extensible and Dynamic Topic Types for DDS standard

DDS is both language and OS independent. The DCPS APIs have been implemented in a range of different programming languages including Ada, C, C++, C#, Java, JavaScript, CofeeScript, Scala, Lua, and Ruby. Using standardized APIs helps ensure that DDS applications can be ported easily between different vendor's implementations.

DDS also specifies a wire protocol, the DDS Interoperability Wire Protocol [2], referred to as DDSI. A wire-level protocol refers to the mechanism for transmitting data from point-to-point. A wire protocol is needed if more than one application has to interoperate. In contrast to protocols at the transport level (like TCP or UDP), the term wire-protocol is used to describe a common way to represent information at the application level. All DDS implementations complying with DDSI will interoperate. The protocol also supports automatic "Discovery" that allows DDS participants to declare the information that they can provide or what data they would like to receive, in terms of topic, type and QoS. The protocol will automatically connect appropriate publishers to subscribers. This significantly simplifies the process of configuring systems with many nodes and many devices exchanging data.

A recent addition to the DDS standards is the inclusion of a new specification, Extensible and Dynamic Topic Types [8], that defines how Topic data types can be extended dynamically while ensuring application portability and interoperability.

4.2 AMQP

AMQP is a message-centric protocol that emerged from the Financial sector with the aim of freeing users from proprietary and non-interoperable messaging systems. AMQP mandates the behavior of the messaging provider and client to the extent that implementations from different vendors are truly interoperable. Previous attempts to standardize middleware have happened at the API level (e.g. JMS) and thus did not ensure interoperability. Unlike JMS, which merely defines an API, AMQP is a wire-protocol. Consequently any product that can create and interpret messages that conform to this data format can interoperate with any other compliant implementation irrespective of the programming language.

AMQP is a binary, application layer protocol, designed to efficiently support a wide variety of messaging applications and communication patterns. It provides flow controlled, message-oriented communication with message-delivery guarantees such as at-most-once (where each message is delivered once or never), at-least-once (where each message is certain to be delivered, but may do so multiple times) and exactly-once (where the message will always certainly arrive and do so only once), and authentication and/or encryption based on SASL and/or TLS It assumes an underlying reliable transport layer protocol such as Transmission Control Protocol (TCP).

4.3 MQTT

MQTT is a message-centric wire protocol designed for M2M communications that enables the transfer of telemetry-style data in the form of messages from devices, along high latency or constrained networks, to a server or small message broker. Devices may range from sensors and actuators, to mobile phones, embedded systems on vehicles, or laptops and full scale computers. It supports publish-and-subscribe style communications and is extremely simple.

4.4 JMS

JMS is one of the most widely used publish-and-subscribe messaging technologies. It is a message-centric API for sending messages between two or more clients. JMS is a part of the Java Platform, Enterprise Edition (Java EE), and is defined by a specification [5] developed under the Java Community Process as JSR 914. It is a messaging standard that allows application components based on Java EE to create, send, receive, and read messages. It allows the communication between different components of a distributed application to be loosely coupled, reliable, and asynchronous. JMS supports both point-to-point and publish-and-subscribe style routing.

The main limitation of JMS is that it is a Java API standard only and does not define a wire protocol. Therefore JMS implementations from different vendors will not interoperate.

4.5 REST

REST has emerged as the predominant Web API design model. RESTful style architectures conventionally consist of clients and servers. Clients initiate requests to servers; servers process requests and return appropriate responses. Requests and responses are built around the transfer of representations of resources. A resource can be essentially any coherent and meaningful concept that may be addressed. A representation of a resource is typically a document that captures the current or intended state of a resource.

REST was initially described in the context of HTTP, but it is not limited to that protocol. RESTful architectures may be based on other Application Layer protocols if they already provide a rich and

uniform vocabulary for applications based on the transfer of meaningful representational state.

4.6 CoAP

CoAP is a document transfer protocol that was designed for use with very simple electronic devices, allowing them to communicate over the Internet. The Internet Engineering Task Force (IETF) Constrained Restful Environments (CoRE) Working Group is currently working on standardizing CoAP.

CoAP is targeted for small low power sensors, switches, valves and resource constrained internet devices such as Wireless Sensor Networks (WSNs) and is designed to easily translate to HTPP for simplified RESTful web integration. CoAP is lightweight, simple and runs over UDP (not TCP) with support for multicast addressing. It is often used in conjunction with WSNs implementing the IETF's emerging IPv6 over Low Power Wireless Personal Area Networks (6LoWPAN) standard. This new standard enables the use of IPv6 in Low-power and Lossy Networks (LLLNs) such as those based on IEEE 802.15.4.

CoAP supports a client/server programming model based on a RESTful architecture in which resources are server controlled abstractions made available by an application process and identified by Universal Resource Identifiers (URIs). Clients can manipulate resource using HTPP: GET, PUT, POST and DELETE methods. It also provides in built support for resource discovery as part of the protocol.

A mapping between CoAP and HTTP is also defined, enabling proxies to be built to provide access to COAP resources in a uniform way via HTTP.

5. What Problems Are We Trying to Solve?

The messaging technologies discussed in this document can be used to connect devices and people (e.g., sensors, mobile devices, single board computers, micro controllers, desktop computers, local servers, servers in a Data Center) in a distributed network (LAN or WAN) via a range of wired and wireless communication technologies including: - Ethernet, Wi-Fi, RFID, NFC, Zigbee, Bluetooth, GSM, GPRS, GPS, 3G, 4G).

The problem has a number of variations that can be categorized as follows:

- Inter Device communication - message exchanges between device nodes on a Local Area Network (LAN)
- Device to Cloud communication - message exchanges between a device node and an Internet based Data Center or between devices via the Internet
- Inter Data Center communication - message exchanges between Internet based Data Centers
- Intra Device communication where messages are exchanges between processes within the same device node, although this is not generally considered an IoT use case

Each messaging technology discussed in this document is suited to addressing one, more or all of the connectivity problems identified above and illustrated in Figure 1.

AMQP, MQTT, JMS and REST were all designed to run on networks that use TCP/IP as the underlying transport. AMQP, MQTT and JMS support brokered publish-and-subscribe message exchanges between device nodes (Inter Device). JMS is focused on Java-centric systems although there are a number of vendors who have developed proprietary C and C++ JMS API mappings and that can be used with a JMS provider. REST encourages a client-server (request/reply) pattern of inter nodal communication using HTTP.

COAP is also based on a RESTful architecture and a client/server interaction pattern. It uses UDP as the underlying transport and can also support IP multicast addressing to enable group communications

between devices. CoAP was designed to minimize message overhead and reduce fragmentation when compared to a HTTP message. When used with UDP the entire message must fit within a single datagram or a single IEEE 802.15.4 frame when used with 6LoWPAN.

AMQP, MQTT and JMS are broker based and can encounter similar issues with respect to reduced performance (lower throughput) and real-time predictability as system scale increases (when the number of publishers, subscribers and nodes grow).

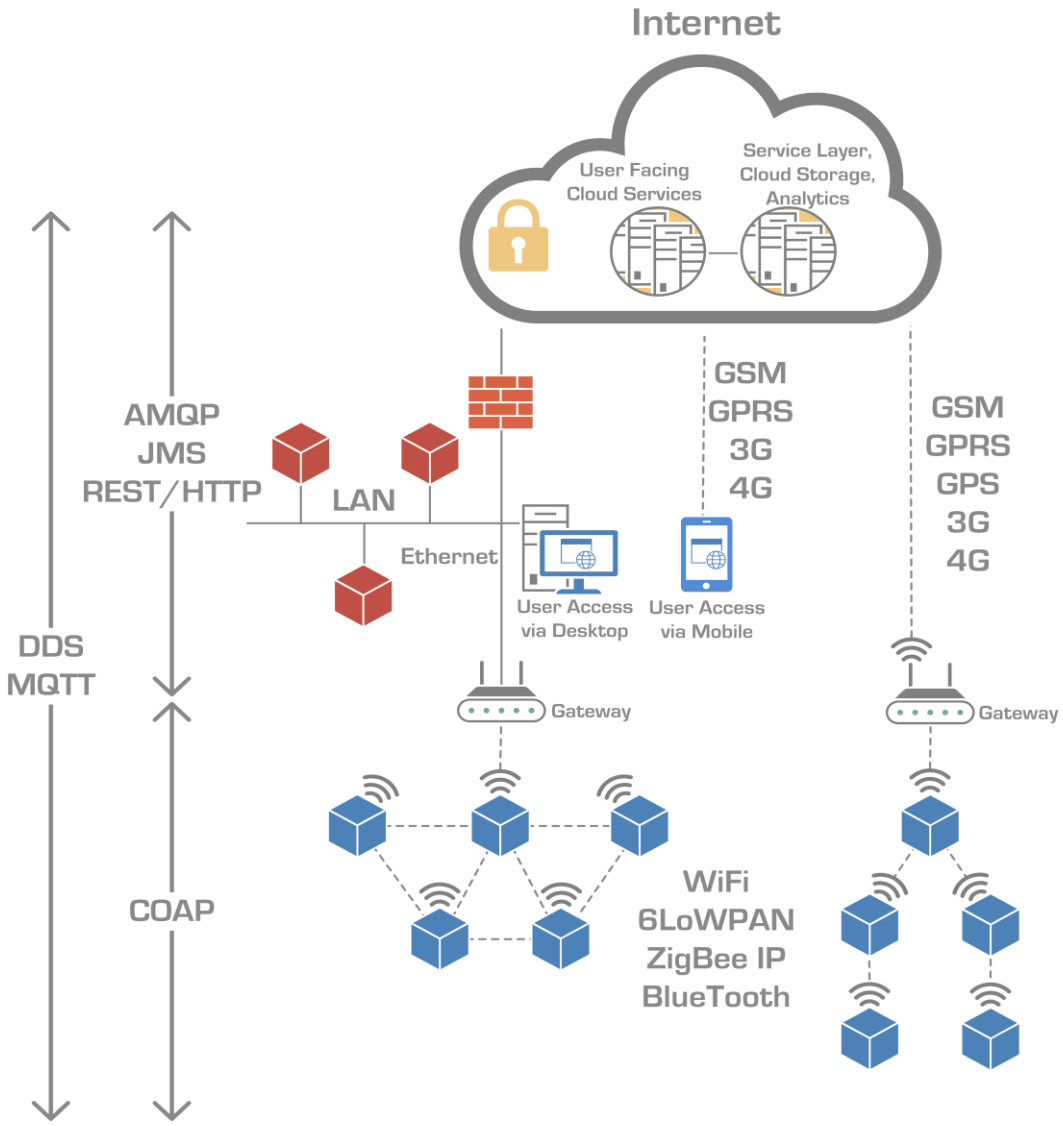

Figure 1 – IoT Connectivity Problem Space

DDS was designed to support large scale, real-time data sharing between devices on a network. It is used in many mission critical systems with large device-to-device data exchanges requiring efficient, predictable, low latency and reliable data sharing. It can be used with either reliable or unreliable networks. Communication reliability is provided by the DDSI wire protocol itself and not dependent on the physical transport. By default DDS uses UDP as its underlying transport but other transports can also be supported (e.g., IP multicast, TCP/IP, shared memory, etc.). DDS is language and OS independent and

can run on very small embedded devices (e.g. a simple wireless sensor) up to large-scale enterprise systems.

DDS provides a decentralized broker-less (see section 6) architecture with direct peer-to-peer communications between publishers and subscribers. DDS comprehensive Quality-of-Service (QoS) support allows users to fine-tune and prioritize data exchanges to ensure maximum throughput and reduce CPU utilization and maximize network bandwidth.

All of the protocols can support distributed message exchanges between processes on a single node (Intra Device). In the case of AMQP, MQTT, JMS and REST they are required to use a reliable transport such as TCP/IP, or UDP in the case of DDS and CoAP. However, CoAP and REST were not designed for high performance message exchanges within the same node and are more appropriate when used to communicate between nodes or with Internet based applications.

By design DDS's connectionless architecture scales better than the other protocols when the number of applications on the node producing and consuming the data increases. PrismTech's OpenSplice DDS provides a 'federated' deployment option where multiple applications on a computer share information via shared memory and where network-traffic to/from that federation is arbitrated by a unique network-scheduler based upon urgency and importance of each exchanged piece of information. This shared-memory based deployment architecture features ultra-low latency inter-core communication along with extreme nodal scalability. This results in better scalability, more efficient data-sharing and better peer-to-peer determinism on the same and between nodes.

AMQP, MQTT and JMS can provide device to Data Center connectivity over the Internet using TCP/IP connections to brokers deployed in the Data Center. RESTful applications can implement request/reply message exchanges from a client to a server in a Data Centre using HTTP. Again JMS restricts this type of configuration to applications written for the Java platform, which can rule out its use in very resource constrained environments. AMQP, MQPPT and CoAP do not define a language specific API for using the protocol and vendors are free to provide implementations that can support a number of different languages (e.g. Java, C++, C, C#, others). For example, if a resource constrained embedded network device needs to publish a message (perhaps containing alarm information) to a management application running in a Cloud-based Data Center, it usually makes more sense if a publisher application running on the device is written in C and not Java in order to minimize its memory footprint.

CoAP nodes are designed to provide device to Data Center connectivity via HTTP proxies using a standard mapping.

As the name suggests MQTT (Message Queuing Telemetry Transport) targets device data collection. Its purpose is to collect data from many devices and transport that data to a Data Center. When the role of the Data Center is simply to collect and process data without the need for real-time data sharing or other QoS requirements then both MQTT and CoAP are useful protocols as they are both very lightweight and can run on the smallest resource constrained device (e.g. low power wireless sensor network), while at the same time providing connectivity to Internet based applications.

Due to computing and platform (e.g. Java for JMS) resources required to host them, AMQP, JMS and REST applications are better suited to support message exchanges between applications running on servers on a LAN or in a Data Center or between Data Centers over the Internet and not resource constrained embedded environments.

By default, compliant DDS implementations must support UDP as the underlying transport used by the DDSI wire protocol. However, a number of vendors also support DDSI implementations that can also use TCP/IP (e.g. PrismTech's OpenSplice DDS), enabling Data Center connectivity over the Internet. DDS has the advantage that it can support low latency, real-time data sharing regardless of location. This

includes device-to-device data exchanges (including over the Internet) and large fan-outs where one device publishes data that is consumed by many subscriber devices or even other Data Centers. Lightweight DDS implementations can support the most resource constrained environments with API support available in a range of different programming languages.

6. Message Broker or Data Bus?

The majority of implementations of AMQP, MQTT and JMS are broker-based. Publishers post messages to a trusted message routing and delivery service, or broker, and subscribers register subscriptions with the broker which also performs any message filtering. The broker normally performs a store and forward function to route messages from publishers to subscribers. In addition, the broker may prioritize messages in a queue before routing. Subscribers may register for specific messages at build time, initialization time or runtime.

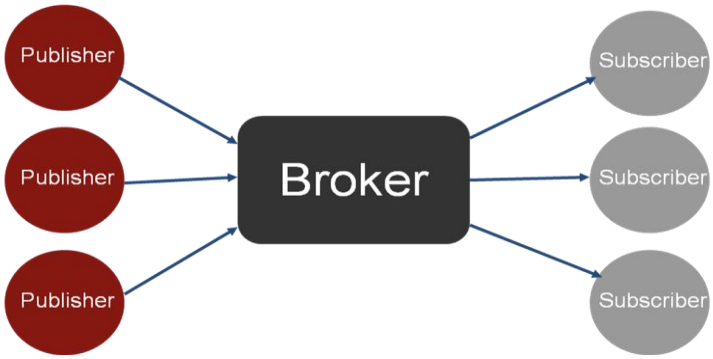

Figure 2 - Message Broker Architecture

Brokers may have a single queue, multiple queues with messages distributed amongst them, copies of each message duplicated in each queue or some other kind of delivery pattern. Flexible routing patterns are a benefit of using a broker. Originating from the Financial sector in which message exchanges are frequently transactional, both AMQP and JMS provide transactional modes of operation that allow them to take part in a multi phase commit sequence. MQTT and AMQP brokers support communications between publishers and subscribers over TCP/IP to provide reliable communications. Most JMS brokers are also TCP-based although this is not mandated by the standard.

Brokers can be deployed in various configurations in a networked environment to suit specific system needs. Common broker configurations include:

- Centralized broker – where the broker resides on a single centralized server and all traffic flows via the server. This model is easy to implement and administer and requires the fewest number of network connections. However the broker can become a single point of failure, or a bottle neck, it doesn't scale and in a real-time system is not predictable.
- Centralized multi broker – in this configuration each queue or topic is hosted on a different server. This model is more complicated to implement, but allows the number of queue and topics to scale better. More client connections are potentially needed and each individual broker becomes a single point of failure. When many publishers or subscribers are talking to the same queue scalability can become an issue. With this configuration, load balancing can be implemented by federating topics or queues over a number of brokers.
- De-centralized broker – most decentralized architectures currently use IP multicast at the network level. A messaging system based on multicasting has no centralized message server. Some of the server functionality, such as persistence, security, and transactions is embedded as a local part of the client, while message routing is delegated to the network

layer by using the IP multicast protocol. A de-centralized brokered model is typically a hybrid architecture in that a publisher or subscriber first connects to a daemon process using TCP/IP, which in turn communicates with other broker processes using IP multicast groups. This model can scale to large numbers of queues, with low numbers of connections while reducing the single points of failure. However, it offers the worst latency and real-time predictability compared to other models.

MQTT, AMQP and JMS do not provide automatic discovery, unlike DDS, This means that configuring a distributed system that uses one of these technologies is through the broker. Publishers and subscribers exchange messages through well know named queues (and topics in the case of JMS) and broker/server addresses. In the case of JMS an initial reference to a JMS broker (provider) is usually retrieved from a lookup service such as the Java Naming and Directory Interface (JNDI).

DDS supports a decentralized broker-less architecture to enable seamless data sharing between producers and consumers. DDS is based on the idea of a virtual “global data space” where producers write to the data space and consumers read from the data space. A data model consisting of named topics, their user defined data types and associated QoS is used to by the DDS infrastructure to control how data is shared. DDS connects producers to consumers over the data bus.

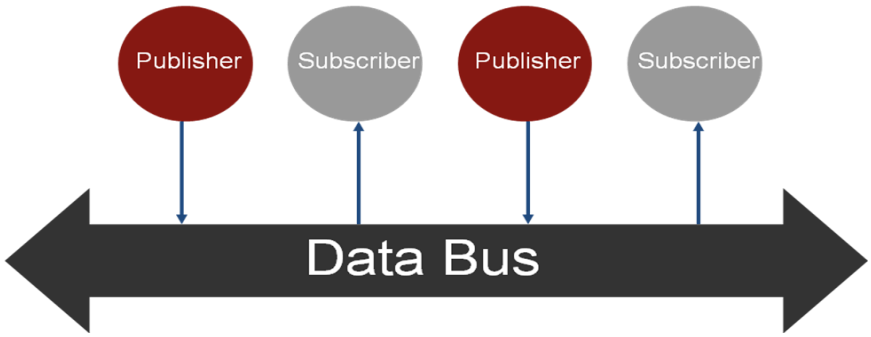

Figure 3 – DDS Data Bus Architecture

DDS provides a dynamic discovery mechanism to automatically match DataReaders and DataWriters. DDS can run over many transports including TCP/IP, UDP (unicast or multicast), shared memory or any other specialist transport. It does no rely on the underlying transport for reliability as this is provided by the DDSI wire protocol.

It is worth noting that there are DDS designs that are also brokered but these are the exceptions and are not usually optimal for most use cases.

7. Data-Centricity or Message-Centricity?

AMQP, MQPP, JMS, REST and CoAP are all message-centric technologies. DDS on the other hand is a data-centric technology. They both can do similar things with respect to providing connectivity in a distributed system, however the way they do it is quite different. In a message-centric system the focus is on delivery the message itself regardless of the data payload it contains and the infrastructure's role is to ensure that messages get to their intended recipients.

In a data-centric system the focus is on user defined data (the data model). The unit of exchange in this type of system is a data value. The middleware understands the data and ensures that all interested subscribers have a consistent view of the data. The infrastructure has done its job not when a message is delivered, but when all nodes have the correct understanding of that value.

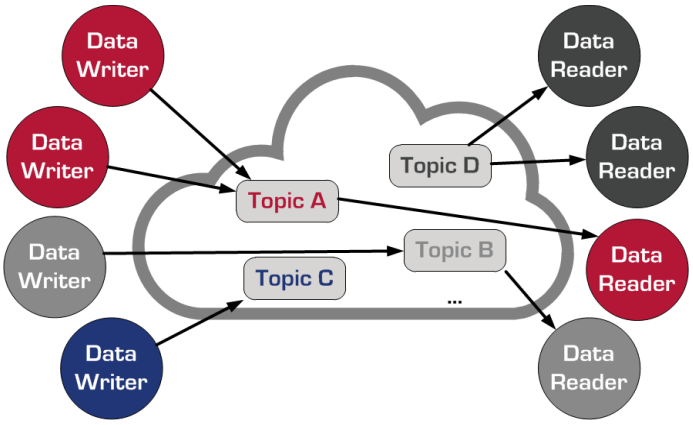

Figure 4 - DDS Global Data Space

One of the key advantages of data-centric technology like DDS is that the data sharing provides a much higher level of abstraction for users of the technology. Data is something that users understand since it represents something in their domain. Message-centric systems on the other hand provide a lower level abstraction as it requires users to implement data sharing through exchange of messages. A data-centric system is therefore easier to maintain and extend, enabling users to focus on developing their business logic and not on writing message handling logic. DDS was designed to support complex systems, inherently providing excellent scalability, fan-out characteristics and state management capabilities. In a data-centric based system applications interact with the data model and not directly with each other. This helps to reduce coupling and enables the system to evolve much more easily and dynamically.

8. Interoperability

DDS enables interoperable data sharing and specifies the DDSI [2] wire protocol to exchange messages between publishers and subscribers. The protocol defines a standard data representation format based on an extension to CDR (Common Data Representation) rules. DDS implementations that support the DDSI are fully interoperable. Messages can be exchanged and understood by different DDS implementations providing that they support DDSI.

MQTT is a wire protocol focused on the interoperable exchange of messages and was designed to be open, simple, lightweight and easy to implement. These characteristics make it ideal for use in constrained environments. For example where the network is expensive, has low bandwidth or is unreliable or when run on an embedded device with limited processor or memory resources. MQTT is a messaging transport that is agnostic to the content of the payload. It does not specify the layout or how data is represented in a message. Although publishers and subscribers can exchange messages applications must agree on serialization schemes otherwise the messages cannot be understood. In large scale distributed systems this can be difficult and costly to implement. MQTT was designed to be used with TCP/IP.

Where JMS provides a standard messaging API for the Java Platform, AMQP provides a standard message protocol across all platforms. Like MQTT, AMQP does not provide a specification for an industry standard API although there is now some work to define standard mappings between the AMQP protocol and common programming APIs (e.g. JMS). It does however provide a specification for a standard wire protocol to describe how the messages should be structured and sent across the network.

AMQP messages have a payload (the data that they carry), which AMQP brokers treat as an opaque byte array. The broker will not inspect or modify the payload. It is possible for messages to contain only attributes and no payload. It is common to use serialization formats like JSON, Thrift, Protocol Buffers and MessagePack to serialize structured data in order to publish it as the message payload. AMQP peers typically use the "content-type" and "content-encoding" fields to communicate this information, but this is by convention only. This means that although publishers and subscribers can exchange messages unless the data serialization scheme is understood by both parties the data payload cannot be interpreted. One option here is to use the AMQP type system to send structured, self describing data.

JMS provides a standard messaging API for the Java platform. With JMS you can replace a JMS-compliant message broker with another implementation with few or no changes to your source code (usually configuration changes are needed). It also allows for interoperability between other Java Platform languages such as Scala and Groovy and provides a level of abstraction that frees you from having to worry about specific vendor's wire protocols and different JMS brokers. JMS does not provide a standard for interoperability outside of the Java platform or between other languages.

RESTful clients and servers based on HTTP are interoperable, since all that is needed to support message exchanges is an HTTP stack (either on the client or the server). Almost every platform and device has that today so interoperability is not a problem.

Like HTTP, CoAP also supports content negotiation. Clients can express a preferred representation of a resource and servers can inform the clients what they will receive (Content-Type). This allows clients and servers to evolve independently, adding new representations independently without affecting each other.

9. Qualities-Of-Service

DDS provides an extremely rich set of Quality-of-Service (QoS) Policies to control the flow of data through the system. There are over 20+ QoS defined by the standard [1] that can be used to control reliability, volatility, liveliness, resource utilization, filtering and delivery, ownership, redundancy, timing deadlines and latency of the data. Table 1 below lists the comprehensive set of QoS Policies and their purpose that are provided by DDS.

QoS Policy	Applicability	RxO*	Modifiable	Description	
DURABILITY	T, DR, DW	Y	N	Controls how data will be stored. Values of TRANSIENT and PERSISTENT result in data outliving the DataWriter.	**Data Availability**
DURABILITY SERVICE	T, DW	N	N	This QoS configures how much data is stored after it is published.	
LIFESPAN	T, DW	-	Y	Setting an expiration time will ensure that a receiving application will not receive values that are too old.	
HISTORY	T, DR, DW			Controls the storing of values before they are delivered.	
WRITER_DATA_LIFECYCLE	DW	-	Y	This policy controls the behavior of the DataWriter with regards to the lifecycle of the data-instances it manages.	
READER_DATA_LIFECYCLE	DR	-	Y	This policy controls the behavior of the DataReader with regards to the lifecycle of the data-instances it manages.	
PRESENTATION	P, S	Y	N	Controls the extent to which changes to data-instances can be made dependent on each other	**Data Delivery**
RELIABILITY	T, DR, DW	Y	N	This policy indicates the level of reliability requested by a DataReader or offered by a DataWriter. BEST_EFFORT being lower than RELIABLE.	
PARTITION	P, S	N	Y	This policy allows the introduction of a logical partition concept inside the 'physical' partition induced by a domain.	
DESTINATION ORDER	T, DR, DW	Y	N	This policy controls how each subscriber resolves the final value of a data instance that is written by multiple DataWriter objects (which may be associated with different Publisher objects) running on different nodes	
OWNERSHIP	T, DR, DW	Y	N	This policy controls whether the Service allows multiple DataWriter objects to update the same instance (identified by Topic + key) of a data-object.	
OWNERSHIP STRENGTH	DW	-	Y	The value of the OWNERSHIP_STRENGTH is used to determine the ownership of a data-instance (identified by the key).	
DEADLINE	T, DR, DW	Y	Y	This policy is useful for cases where a Topic is expected to have each instance updated periodically. On the publishing side this setting establishes a contract that the application must meet. On the subscribing side the setting establishes a minimum requirement for the remote publishers that are expected to supply the data values.	**Data Timeliness**
LATENCY BUDGET	T, DR, DW	Y	Y	This policy provides a means for the application to indicate to the middleware the "urgency" of the data-communication	
TRANSPORT PRIORITY	T, DW	-	Y	The purpose of this QoS is to allow the application to take advantage of transports capable of sending messages with different priorities	
TIME BASED FILTER	DR	-	Y	This policy allows a DataReader to indicate that it does not necessarily want to see all values of each instance published under the Topic. Rather, it wants to see at most one change every minimum_separation period.	**Resources**
RESOURCE LIMITS	T, DR, DW	N	N	This policy controls the resources that the Service can use in order to meet the requirements imposed by the application and other QoS settings.	
USER_DATA	DP, DR, DW	N	Y	The purpose of this QoS is to allow the application to attach additional information to the created Entity objects such that when a remote application discovers their existence it can access that information and use it for its own purposes.	**Configuration**
TOPIC_DATA	T	N	Y	The purpose of this QoS is to allow the application to attach additional information to the created Topic such that when a remote application discovers their existence it can examine the information and use it in an application-defined way	
GROUP_DATA	P, S	N	Y	The purpose of this QoS is to allow the application to attach additional information to the created Publisher or Subscriber.	
ENTITY FACTORY	DPF, DP, P, S	N	Y	This policy controls the behavior of the Entity as a factory for other entities.	
LIVELINESS	T, DR, DW	Y	N	This policy controls the mechanism and parameters used by the Service to ensure that particular entities on the network are still "alive."	**System Availability**

*RxO indicates that QoS Policy objects need to be set in a compatible manner between the publisher and subscriber ends

T=Topic, DW=Data Writer, DR=Data Reader, P=Publisher, S=Subscriber, DP=Domain Participant, DPF = Domain Participant Factory

Table 1 – DDS QoS Policies

Data objects in a DDS system are identified by Topics. When a DataReader's Topic is compatible with a DataWriter's Topic, then the "publication" and "subscription" become associated and data is published between them. Topics are compatible when they have the same name, the same data type and the QoS polices are not in conflict. The Topic, DataReader, DataWriter, Publisher, and Subscriber all have QoS polices. The QoS policies of Publisher, DataWriter, and Topic control the data on the sending side. QoS policies of Subscriber, DataReader, and Topic control the data on the receiving side.

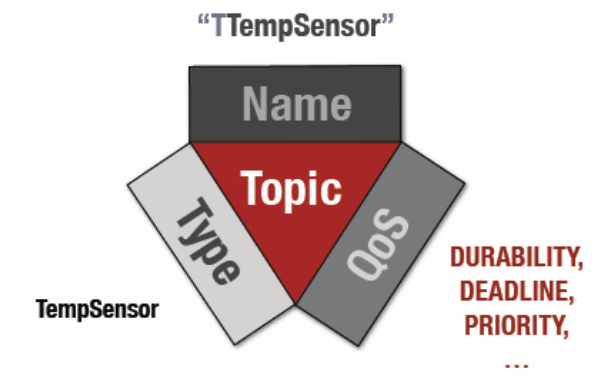

Figure 5 – DDS Topic Name, Type and QoS Relationships

DDS QoS Policies provide an extremely powerful and flexible mechanism that can be used to provide per-stream optimization of data flows.

The MQTT protocol provides very basic QoS support for delivering messages between clients and servers. QoS is an attribute of an individual MQTT message being published. An application sets the QoS for a specific message by setting the MQTTClient_message.qos field to the required value. A subscribing client can set the maximum quality of service a server uses to send messages that match the client subscriptions. The MQTTClient_subscribe() and MQTTClient_subscribeMany() functions set this maximum. The QoS of a message forwarded to a subscriber might be different to the QoS given to the message by the original publisher. The lower of the two values is used to forward a message.

The three QoS settings provided by MQTT are:

- At-most-once - the message is delivered at most once, or it may not be delivered at all. Its delivery across the network is not acknowledged. The message is not stored. The message could be lost if the client is disconnected, or if the server fails. It is the fastest mode of transfer. It is sometimes called "fire and forget". The MQTT protocol does not require servers to forward publications to a client. If the client is disconnected at the time the server receives the publication, the publication might be discarded, depending on the server implementation.

- At-least-once - the message is always delivered at least once. It might be delivered multiple times if there is a failure before an acknowledgment is received by the sender. The message must be stored locally at the sender, until the sender receives confirmation that the message has been received by the intended recipient. The message is stored in case the message must be sent again.

- Exactly-once - the message is always delivered exactly once. The message must be stored locally at the sender, until the sender receives confirmation that the message that the message has been received by the intended recipient. The message is stored in case the message must be sent again. It is the safest, but slowest mode of transfer. A more sophisticated handshaking and acknowledgement sequence is used to ensure no duplication of messages occurs.

JMS messages have a number of QoS properties that can be set. These QoS properties include the following:

- JMSDeliveryMode - there are two types of delivery modes in JMS: persistent and non-persistent. A persistent message should be delivered once-and-only-once, which means that a message is not lost if the JMS provider fails, it will be delivered after the server recovers. A non-persistent message is delivered at-most-once, which means that it can be lost and never delivered if the JMS provider fails. In both persistent and non-persistent delivery modes the message server should not send a message to the same consumer more than once, but it is possible. In general non-persistent messages perform better than persistent messages. They are delivered more quickly and require less system resources on the message server. However, non-persistent messages should only be used when a loss of messages due to a JMS provider failure is not an issue.

- JMSExpiration - a message object can have an expiration date. The expiration date is useful for messages that are only relevant for a fixed amount of time. The expiration time for messages is set in milliseconds by the producer using the setTimeToLive() method on either the QueueSender or TopicPublisher. The JMSExpiration is the date and time that the message will expire. JMS clients should be written to discard any unprocessed messages that have expired, because the data and event communicated by the message is no longer valid. Message providers (servers) are also expected to discard any undelivered messages that expire while in their queues and topics. Even persistent messages are supposed to be discarded if they expire before being delivered.

- JMSPriorityPurpose - messages may be assigned a priority by the message producer when they are delivered. The message servers may use a message's priority to order delivery of messages to consumers; messages with a higher priority are delivered ahead of lower priority messages

AMQP messages have similar QoS properties to MQTT. This includes supporting message queuing and delivery semantics covering at-most-once, at-least-once and once-and-only-once (reliable messaging).

REST qualities of service provided by the underlying transport. HTTP is an application layer protocol designed within the framework of the Internet Protocol Suite. Its definition presumes an underlying and reliable transport layer protocol and TCP/IP is most commonly used. However HTTP can use unreliable protocols such as the User Datagram Protocol (UDP), for example in Simple Service Discovery Protocol (SSDP).

CoAP provides only rudimentary message delivery QoS. CoAP requests and reply messages may be marked as "confirmable" or "nonconfirmable". Confirmable messages must be acknowledged by the receiver with an ack packet. Nonconfirmable messages are "fire and forget".

10. Performance

In a very simple point-to-point configuration between nodes AMQP, MQTT, JMS, REST/HTTP, CoAP and DDS may have comparable performance characteristics, although broker-based routing adds an additional overhead when compared to a broker-less infrastructure such as DDS, HTTP or CoAP request. Requests to a CoAP resource from HTTP client have the additional overhead of having to be forwarded via a CoAP/HTTP proxy. CoAP should only be considered where low latency and real-time performance are not a requirement. CoAP's support for IP multicast does mean that a single CoAP client can issue the same request to multiple CoAP servers concurrently (e.g. a single request could be issued to multiple wireless devices to turn off all of lights in a street at once). CoAP also has the ability to observe a

resource. When a observe flag is set on a GET request the server can continue to reply after the initial document transfer, allowing state changes to be streamed to a client as they occur.

When DDS is deployed on a LAN, communication between publishers and subscribers will be over UDP multicast and combined with and rich and flexible set of QoS polices enables exceptional “fan-out” scalability. DDS implementations can scale to tens of 1000s messages/sec per peer on networks consisting of thousands of devices. In a real-time system where latency is measured in micro seconds and predictable data delivery is key requirement only DDS out of the messaging technologies discussed in this document can provide the timing control necessary for these types of system.

Typically a single message broker will deliver 100s to1000+ messages/sec. By deploying more instances of the broker and by federating queues across brokers, increased message throughput and scalability can be achieved. However, this comes at the expenses of potentially more network connections and the introduction of additional points for failure.

In certain types of non real-time systems such as Business-to-Business (B2B), Business-To-Consumer (B2C) or Financial applications, message throughput may not be the overriding concern when compared to the need for reliable transactional message delivery. In these cases AMQP and JMS implementations have distinct advantages.

11. Security

In order to build a trusted and fault-tolerant system in a connected IoT world security issues must be considered. These include how to protect communications, how to manage authentication and access control to resources in systems that may consist of thousands of devices and how to ensure integrity and confidentiality of the data in the system.

JMS does not provide an API for controlling the privacy and integrity of messages. It also does not specify how digital signatures or keys are distributed to clients. Security is considered an issue for specific to each JMS provider. The main JMS vendors provide various levels of proprietary security functionality within their implementations. Typically this involves providing facilities to support client authentication and access control to JMS queues and topics. Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL) which are cryptographic protocols are often used to provide communication security. They use asymmetric cryptography for authentication of key exchange, symmetric encryption for confidentiality and message authentication codes for message integrity. In a number of the leading JMS implementations the Java Authentication and Authorization Service (JAAS) is used to provide pluggable authentication and authorization support for the provider.

As of MQTT v3.1, a user name and password can be passed in an MQTT packet. This can help simplify the authentication of individual clients in a system by reducing the number of keys that need to be distributed and managed in comparison to an exclusively key based system. Encryption of data exchanged across the network can be handled independently from the MQTT protocol using SSL or TLS.

With AMQP, security layers are expected to be defined externally to the AMQP specification, for example the use of TLS for data encryption. The exception to this is the use of the Simple Authentication Security Layer (SASL) which is specified by the standard and can be used to enable application protocols such as AMQP to negotiate an agreed authentication mechanism.

DDS does not currently provide a standardized security capability. Leading DDS vendors provide various proprietary security solutions. PrismTech’s OpenSplice DDS for example provides authentication and encryption of all data exchanged between nodes using SSL. It also provides a pluggable access control module that can be applied to DDS topics and partitions in order to support different access control strategies such as Mandatory Access Control (MAC) or Role Based Access Control. Other DDS

implementations offer similar capabilities to varying degrees. The OMG is currently in the process of adopting a new DDS Security standard that should be available by the end of 2013. The new standard defines the Security Model and Service Plugin Interface (SPI) architecture for compliant DDS implementations. The DDS Security Model is enforced by the invocation of these SPIs by the DDS implementation. The specification also defines a set of built-in implementations of these SPIs.

- The specified built-in SPI implementations enable both out-of-the box security and interoperability between compliant DDS applications.
- The use of SPIs allows DDS users to customize the behavior and technologies that the DDS implementations use for Information Assurance, specifically allowing customization of Authentication, Access Control, Encryption, Message Authentication, Digital Signing, Logging and Data Tagging.

In a RESTful system securing message exchanges over HTTP is typically over SSL. Referred to as HTTPS, HTTP was the first protocol to use SSL. HTTP can also be used with newer TLS implementations. Both SSL and TLS provide HTTP client and servers with asymmetric cryptography for authentication of key exchange and symmetric encryption for confidentiality.

CoAP is built on top of UDP and as such it cannot rely on SSL/TLS (available with TCP/IP) to provide security capabilities. In the case of UDP, Datagram Transport Layer Security (DTLS) provides the same assurances as TCP but for message exchanges over UDP.

12. Conclusion

A number of key messaging technologies are emerging that will support the next generation of IoT applications. The messaging technologies discussed in this document include DDS, MQTT, AMPQ, JMS REST and CoAP, each of which can be used to connect devices in a distributed network. However, their suitability to support the different operational scenarios considered, including Inter and Intra Device communication, Device to Cloud communication and Inter Data Center communication varies, especially when key system requirements such as performance, quality-of-service, interoperability, fault tolerance and security are taken into account.

For device-to-device applications that require high performance, real-time, many-to-many managed connectivity then DDS has distinct advantages over the other messaging technologies. DDS is also emerging as a key enabler for connecting real-time device networks to cloud-based Data Centers.

The choice of the most appropriate messaging solution should be based on an understanding of both the architecture and the message/data sharing requirements for each target system. A summary of the key criteria considered in this document is shown below in Table 2.

	DDS	MQTT	AMQP	JMS	REST/HTPP	CoAP
Abstraction	Pub/Sub	Pub/Sub	Pub/Sub	Pub/Sub	Request/Reply	Request/Reply
Architecture Style	Global Data Space	Brokered	P2P or Brokered	Brokered	P2P	P2P
QoS	22	3	3	3	Provided by transport e.g. TCP	Confirmable or nonconfirmable messages
Interoperability	Yes	Partial	Yes	No	Yes	Yes
Performance	10s of 1000s of messages per second. Massive fan-out performance	Typically 100s to 1000+ messages per second per broker	Typically 100s to 1000+ messages per second per broker	Typically 100s to 1000+ messages per second per broker	Typically 100s of requests per second	Typically 100s of requests per second
Real-time	Yes	No	No	No	No	No
Transports	UDP by default but other transports such as TCP can also be used	TCP	TCP	Not specified but typically TCP	TCP	UDP
Subscription Control	Partitions, Topics with message filtering	Topics with hierarchical matching	Exchanges, Queues and bindings in v0.9.1 standard, undefined in latest v1.0 standard	Topics and Queues with message filtering	N/A	Provides support for Multicast addressing
Data Serialization	CDR	Undefined	AMQP type system or user defined	Undefined	No	Configurable
Standards	OMG's RTPS and DDSI standards	Proposed OASIS MQTT standard M	OASIS AMQP	JCP JMS standard	Is an architectural style rather than a standard	Proposed IETF CoAP standard
Encoding	Binary	Binary	Binary	Binary	Plain Text	Binary
Licensing Model	Open Source & Commercially Licensed	Open Source & Commercially Licensed	Open Source & Commercially Licensed	Open Source & Commercially Licensed	HTTP available for free on most platforms	Open Source & Commercially Licensed
Dynamic Discovery	Yes	No	No	No	No	Yes
Mobile devices (Android, iOS)	Yes	Yes	Yes	Dependent on JAVA capabilities of the OS	Yes	Via HTTP proxy
6LoWPAN devices	Yes	Yes	Implementation specific	Implementation specific	Yes	Yes
Multi-phase Transactions	No	No	Yes	Yes	No	No

	DDS	MQTT	AMQP	JMS	REST/HTPP	CoAP
Security	Vendor specific but typically based on SSL or TLS with proprietary access control	Simple Username/Password Authentication, SSL for data encryption	SASL authentication, TLS for data encryption	Vendor specific but typically based on SSL or TLS. Commonly used with JAAS API	Typically based on SSL or TLS	DTLS

Table 2 – Summary of Key Comparison Criteria

13. References

1. Data Distribution Service for Real-Time Systems Version 1.2, OMG Available specification formal/07-01-01
2. The Real-Time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification Version 2.1, OMG Document Number: formal/2009-01-05
3. MQ Telemetry Transport (MQTT) Specification v3.1, IBM, Eurotech
4. Advanced Message Queuing Protocol (AMQP) Version 1.0 Specification, OASIS
5. Java Message Service, Nigel Deakin, Oracle, Version 2.0, March 2013
6. Constrained Application Protocol (CoAP) draft-ietf-core-CoAP-18, June 28, 2013
7. MQTT For Sensor Networks (MQTT-S) Protocol Specification, Version 1.2, June 6, 2011
8. Extensible and Dynamic Topic Types for DDS, Version 1.0 OMG Document Number: formal/2012-11-10

14. PrismTech Contacts

USA Headquarters
PrismTech Corporation
400 TradeCenter
Suite 5900
Woburn, MA
01801
USA

Tel: +1 781 270 1177

European Headquarters
PrismTech Limited
PrismTech House
5th Avenue Business Park
Team Valley
Gateshead, NE11 0NG
UK

Tel: +44 (0)191 497 9900

PrismTech France
28 rue Jean Rostand
91400 Orsay
France

Tel: +33 (1) 69 015354

Web: *http://www.prismtech.com*
E-mail: *info@prismtech.com*

15. Notices