A Literature Review on Middleware solutions for Industry 4.0

Suman Patro
Department of Computer Engineering
K. J. Somaiya College of Engineering
Mumbai, India

Rushikesh Nikam
Department of Computer Engineering
K. J. Somaiya College of Engineering
Mumbai, India

Manish Potey
Department of Computer Engineering
K. J. Somaiya College of Engineering
Mumbai, India

ABSTRACT
Communication systems work in synchronous or asynchronous mode. Asynchronous working of a system is based on event paradigm, wherein only the changed state of the system is recorded. This benefits the system performance drastically since redundant tracking of states or data is not performed. Event-based architectures are modeled through a Middleware component which, in a general sense connects Business, Enterprise or Software in a distributed environment. Middleware are essentially based on the publish-subscribe (pub-sub) pattern. Modern software Platforms that fall under Industry 4.0 employs a middleware for communication between entities in the system. This additional layer reduces the connection overhead of the system, which is not the case with the conventional peer-to-peer model. Hence, Messaging systems based on the Middleware approach, with event-driven principle and pub-sub pattern provide added benefits, of dynamic reception of data to all those entities in the system that are interested in a specific data type and maintaining communication links between entities and the Middleware, and not with every other entity within the system. This paper aims to review and evaluate Middleware solutions such as RabbitMQ, ZeroMQ, Mosquitto, Apache Qpid and YAMI4 based on factors such as middleware paradigms, available messaging patterns, middleware performance (message throughput and latency), message priority and queuing, message routing, etc. Based on optimal throughput and latency measures, YAMI4-message oriented middleware (Message Broker) proves feasible for Industry 4.0 platforms. This paper also focuses on the Open issues and solutions with respect to specific middleware types.

Keywords
Middleware, Industry 4.0, publish-subscribe, RabbitMQ, ZeroMQ, Mosquitto, Apache Qpid, YAMI4, Message Broker

1. INTRODUCTION

1.1 Motivation
Traditionally, a software package such as Supervisory Control and Data Acquisition Unit (SCADA) [4] employs polling-based data fetch that lead to the tracking of redundant data, this proved inefficient for systems where data influx and outflux is huge. Therefore, Event-driven architecture [10] is used in modern systems to track changed data. Event mechanism is modeled after a pub-sub design pattern [19]. Pub-sub, unlike request response, is based on a principle which serves information to only those components of the system that have prior subscribed for a particular data type. This leads to efficient data handling in the modern software platforms.

In the past, system modules used to be tightly coupled in a point-to-point manner. This induced connection overheads, since every module used to be connected to every other in the system. Modules also had to know each other semantically for communication purpose; this posed an additional burden on the system. Hence, a middleware approach (message broker) in modern software platforms is used to facilitate data dissemination between entities. This additional layer communicates with all the other modules in the system, and vice-versa, instead of all the other modules communicating with each other.

1.2 Paper Organization
Section 2 deals with concepts related to Event mechanism. Section 3 briefs about the existing middleware Paradigms. Section 4 enlists the Master criteria set for middleware evaluation. Section 5 focuses on the theoretical understanding of the middleware solutions based on the criteria set. Section 6 gives a practical analysis of the middleware solutions based on the existing studies. Section 7 gives a detailed comparison report of YAMI4 & Apache Qpid based on their performance parameters and chooses YAMI4 as the optimal solution. Section 8 focuses on YAMI4's internal mechanism based on experimental tests conducted. Section 9 enlists Open issues in middleware and best available solutions based on specific middleware types. Section 10 provides a summary of the research conducted and highlights further work to enhance YAMI4 message broker in Web and Application security context.

2. BACKGROUND

2.1 The Middleware Approach
Middleware is software that connects software components or enterprise applications. It allows application modules to be distributed over heterogeneous platforms and reduces the complexity of developing applications that span multiple operating systems and network protocols. The middleware creates a distributed communications layer that insulates the application developer from the details of the various operating systems and network interface.

2.2 Event-driven architecture
Event-driven architecture [10] is an architectural style that builds on the fundamental aspects of event notifications to facilitate immediate information dissemination and reactive business process execution. It has producers, consumers which are components in an event mechanism. Any change in the state recorded is termed as an event. This event is fired by emitters and handled by consumers. As event sources publish these notifications, event receivers can choose to listen to or filter out specific events, and make proactive decisions in real-

time about how to react to the notifications. The Event handlers are used to process the events and event listeners act as interfaces on the consumers. These architectures are fundamental blocks in asynchronous environments. Event driven architectures have loose coupling within space, time and synchronization, providing a scalable infrastructure for information exchange and distributed workflows. The architecture is extremely loosely coupled because the event itself doesn't know about the consequences of its cause.

2.3 The Pub-Sub model [19]

The design pattern followed in Middleware is known as pub-sub since there are producers (publishers) who publish data and subscribers (receivers) who subscribe to particular data of their interest. Fig. 1 shows the process:

1. Publishers and subscribers find the broker address from the broker discovery service.
2. Each of them registers to the broker over topics, using which publishers publish messages and subscribers receive those messages.
3. The publisher publishes messages and the topic of publication to the broker.
4. Message broker matches the topic of publication with the saved subscriptions and pushes data to the relevant subscribers only.

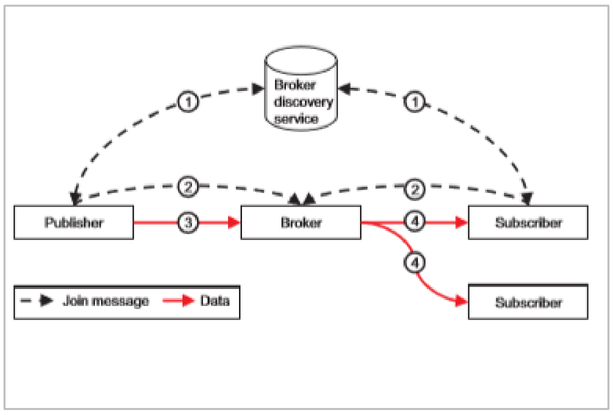

Fig 1: Basic pub-sub communications

The benefit of this approach is, not all the consumers are overloaded with all the data, and if there are consumers in the system that are interested in similar data type then their request can be processed all at once.

Hence the functionalities viz. message serialization, message routing, message transfer in multiple supported patterns, etc. are all handled at the broker level, and the other communicating parties need to simply transfer data.

3. MIDDLEWARE PARADIGMS

Middleware paradigms involve Object, Service, Data and Message. Middleware systems based on such paradigms are Object-oriented, Service-oriented, Data-oriented and Message-Oriented.

In Object-oriented approach [6], an object-oriented middleware focuses on the receiver's identity to make the application's name in an unambiguous manner, provides location independency even for migrating applications and provides interface and inheritance abilities. The physicality of the code remains hidden from the programmer; hence the distribution of the whole system becomes a deployment issue and is not necessary to deal with in the coding stage. The code is implemented in a similar manner to both local and remote (distributed) invocations. Even the management issue is not much critical since the cost of delivery is quite same to both local and remote invocations. This is due to similar implementation of remote and local invocations. But, due to the isolation of the programmer from the physical aspects; time to send and receive messages, total number of messages, etc. are not known to the user code, and thus management becomes difficult.

Examples of Object-oriented middleware systems are CORBA (omniORB, JacORB, TAO, etc.) and ICE.

Limitations of Object-oriented middleware are:

- High memory footprint.
- C++ and Java implementations differ.
- Complex error-prone API.
- No direct support for pub-sub.
- Blocking issues
- Shrinking community.
- Lack of new releases and bug fixes.

The Service-oriented approach [6] is quite similar to the object-oriented approach. In this middleware, there is less focus on the target of invocation and more on the operation to be performed; this puts most of the effort on defining the operation and the data structure that is being sent or received, and the user of such a system is not concerned much with the receiver. It is simpler than Object-oriented because issues like identity and lifetime of remote objects do not have to be resolved. Thrift is an example of service-oriented middleware.

A Data-centric middleware [6] focuses on the purpose and meaning of data that is subject to transmission and not on who is sending the data, and most of the effort is spent on ensuring effective routing of information to all interested parties, the sender and receiver are not responsible for any data handling, only data transfer is to be taken care of by the end users. The decoupled components in this approach lead to resilient and fault tolerant systems. The physicality of data is not hidden from the programmer, hence management of data is easy, but decoupling makes it difficult to set up communication in the request-response manner (typical for client-server interactions) which is common to a number of scenarios. Examples of data-centric middleware systems are all DDS implementations (OpenSpliceDDS, OpenDDS, RTI DDS, etc).

In Message oriented middleware [6], the physicality of messages is not hidden, hence management of messages is possible within the user code, even the messages can be processed in sequence and in parallel. In contrast to DATA-CENTRIC, this approach supports both peer-to-peer interactions in the request-response style, as well as with decoupled publish-subscribe, for data transfer. Hence, the message-oriented approach highlights the physicality of communication without any loss of generality. It also focuses on reliability, scalability, and fault tolerance with optimal throughput and caters to Enterprise solutions as well. The disadvantage of this middleware type is it has an overhead of another layer - broker which requires an additional hop to reach the consumer. Examples are Apache Qpid, YAMI4, etc.

4. CRITERIA FOR MIDDLEWARE EVALUATION

Middleware has feature set that complies with specific use-cases. Fig. 2 shows categorization of such features into levels of requirements of a system, viz. Fundamental, Mandatory or Desirable.

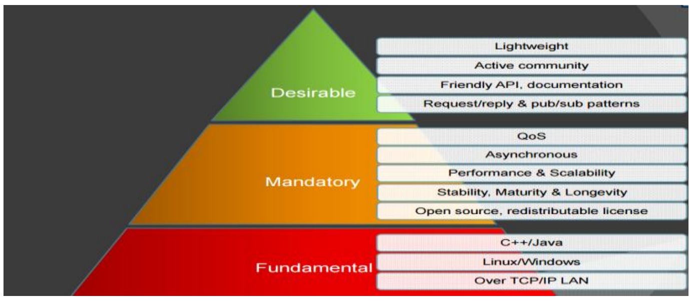
Desirable
Lightweight
Active community
Friendly API, documentation
Request/reply & pub/sub patterns
Mandatory
QoS
Asynchronous
Performance & Scalability
Stability, Maturity & Longevity
Open source, redistributable license
Fundamental
C++/Java
Linux/Windows
Over TCP/IP LAN

Fig 2: Categorizing features according to the system requirements

Evaluation criteria based on feature set is as follows-

- Language used for middleware development
- Supported languages for client API
- Age of the solution
- Middleware domain subset-Data centric, message-centric, object-centric, service-oriented
- Memory Footprint- the size of the application when in-process or, in steady state. Lines of code are a form of measure for analyzing both the complexity level of the code and memory footprint.
- Application protocols supported- Middleware support their own messaging frameworks based on application -level protocols. For example, Websockets for real-time web applications, Advanced Message Queuing Protocol (AMQP) for enterprise-level communication, Messaging Queuing Telemetry Transport (MQTT) for embedded systems, etc.
- Platform support- OS platforms supported by the middleware.
- Messaging/Communication pattern support in the middleware are categorized as, Pub-Sub (push model), Request-Response or pull model (e.g. HTTP model), Point-to-Point (or Peer-to-Peer), ACTive (Availability for Concurrent Transactions), Pipeline (for aggregation and load-balancing), Survey (a single request for the state of multiple applications).
- Message handling capacity (Throughput) - This is nothing but the rate of transfer of messages per unit time in the system.
- Latency- It is measured as the time required for a single message to traverse from one end-point to another.
- Persistence- It is the process of saving data real-time so that if the middleware crashes, there is a backup. Persistence mechanisms available are in-memory persistence, resident memory persistence, and disk drive persistence, for e.g., RAM, distributed caches or database.
- Load Balancing- This means, if an application system seems busy, the message could be forwarded to a parallel process in an alternate location. Federated cluster mechanism is used to achieve load balancing.
- Scalability is the capability of a system to serve multiple clients. This can be measured with respect to client connections i.e. maximum number of connections supported by the system keeping a performance benchmark. Scalability is also used to quantify queue volume of message queue servers.
- Routing: It is nothing but selecting a path to transfer data from source to destination. Based on topology and frequency of topology change, routing mechanisms are included into the middleware. This is a complex criterion when publishers or subscribers are mobile, for example in IOT applications.
- Queues are a part of messaging semantics. The purpose of queuing is, when a receiver entity is down, or when there is network congestion, the messages sent by the sender application is buffered at the queue, avoiding loss. Queue categories are, the store and forward within the broker (point to point), pub-sub (in broker) and store and forward at the receiver, pub-sub with topic filtering (exchange), pub-sub based on fan out, pub-sub with content filtering and pub-sub based on headers. Queue creation types are, durable or ephemeral, and persistent or non-persistent. Middleware support for task queues, for message queues, etc.
- Support for delayed jobs is a feature in the Message queue and Task queue managers.
- Data interchange format available at the middleware level, for example, XML, JSON, customized solution such as message pack, etc.
- QoS-of messaging- Message Acknowledgement schemes, support for error notification for message acknowledgement using 'ACK' or 'NACK', Delivery Policies of whether a message should be delivered at least once, no more than once or at most once, Purging Policies based on TTL, Message Size/Format supported, Message Ordering, support for message batching (Message Batch size), etc.
- Security at the middleware layer is categorized based on authentication, confidentiality, integrity and availability goals.
- Middleware Discovery is a feature that allows the middleware to get discovered by the applications even in geographically distant locations.
- High Availability (HA)/Failover- This is achieved by employing multiple middleware nodes in the system, so that even if a single node fails, there are other backup nodes with the consistent state to take over.
- Reliability- It is the measure with which the system conforms to some specification.
- Event-Driven Services- Some services are used on an as-and-when required, hence some are up all the time and others are event based, set as and when needed.

- Independence in connection to applications means coupling of the middleware with the supporting applications is low.
- Support for dynamic subscribers/listeners- This allows subscribers to join and leave the system dynamically, difference in message receipt time of new members with respect to old ones are observed in such a scenario.
- Listeners re-querying for best sources even after finding an acceptable publisher/feed.
- Open standards support- Middleware that supports Open messaging formats, open protocols, and open data formats.
- Management Console support- Availability of management console to monitor message broker status or message queue status, to avail visualized statistics such as the number of messages per second and the consumption of resources, such as memory, sockets, and the crucial file descriptors.
- Management/maintenance of the middleware
- Ease of use/Deployment model
- Forum support- Community support for research and development, bug fixes, etc.
- Licensing and Royalty structure - Middleware licensing policies such as open source GPL, LGPL, BSD, etc), proprietary or commercial.
- Self-hosted/remotely hosted- Middleware for specific business models are self-hosted or remotely hosted.
- Operational and financial cost- Financial or operational burden of the solution.
- No Vendor lock-in - Systems that are based on open standards are interoperable and hence support no vendor lock-in.
- Application support- Real-time systems that employ the middleware and its flexibility in use.

5. RELATED LITERATURE

Modern systems to meet the requirements of Industry 4.0 need a middleware that connects all the software modules which participate in data fetch from driver applications and display those over the user interface. The Object and Service-oriented middleware prove feasible for connecting Business and ERP solutions. Hence, middleware requirements for Industry 4.0 platform comply with Data and Message paradigms.

This paper therefore reviews and evaluates solutions from Message-Oriented Middleware as shown below.

5.1 Mosquitto

Mosquitto [2] is a message-centric broker, based on MQTT wire protocol, developed in C language and designed essentially for TCP/IP networks.

Licensing scheme supported is open source-BSD. The protocol has no request /response support, but only pub-sub style communication and designed for machine-to-machine (M2M) at device level along high latency or constrained networks, to a server or small message broker. Therefore, being lightweight is beneficial.

MQTT-based publishers and subscribers are not interoperable completely; it works for device data collection though. Message unmarshalling between different MQTT pub-sub is possible only if the format of the message body is agreed between peers.

It provides 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 unmarshaled. QoS is an attribute of an individual MQTT message being published; 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: exactly once, at the most once and at least once. There is no provision for queues at the protocol level; hence sender and receiver must be up simultaneously. But Mosquitto at the broker level supports queuing. It also supports both persistent and non-persistent messaging. MQTT has no flow control or selective acknowledgment to prevent app-locks. There is even no transactions support for the application server.

With respect to Security as of MQTT v3.1, the username-password is used over the key-based system. This helps in efficient key management, and also serves the purpose of authenticating clients. SSL and TLS based encryption is available at the protocol level. Active directory (Kerberos) support is not available.

Mosquitto has Websocket support for real-time messaging for web-based applications.

5.2 Apache Qpid

Apache Qpid [7, 11] is a message-oriented middleware written in C++ that stores, routes, and forwards messages using AMQP based on open source Apache 2.0 license. It supports AMQP 1.0 and AMQP 0-10 at the application level and works on both Linux and Windows platform. Pub-sub and request/reply (slower transfer) are supported in Qpid. Exchange-Bind-Queue is the principle used in Qpid.

Numbers of Queues are unlimited and its size can be set. After queue overflow, overflow policies such as reject, flow to disk, ring, ring strict are used. Queuing policies supported are, FIFO and Last Value Queue (LVQ). Exchange types supported are, Built in exchanges such as default (nameless) exchange- never replicated, the AMQP standard exchanges (amq.direct, amq.topic, amq.fanout and amq.match) and the management exchanges (qpid.management, qmf.default.direct and qmf.default.topic).

Pluggable persistence is supported in Qpid. It stores its queues in memory or in the database. For the persistence of messages, relational Apache Derby database and the Oracle Berkeley DB are supported. Routing is not available but is done through AMQP; supports Header-based routing. Qpid replicates data and metadata across a cluster combined of nodes, hence supports queue replication and transaction, and the client should have the information about the node-cluster relationship i.e. which node forms which cluster.

Message grouping in Qpid: the broker uses this group identification to enforce policies to control how messages from a given group are to be distributed to consumers; this is done using Qpid config tool. For example, if both group A and group B messages are in the same queue with B group messages being lined after A group, this doesn't imply B messages will always be fetched after A group messages, the current consumer up for receiving messages will receive A group, by the time other consumers can access B group.

HA in Qpid is achieved by multiple brokers: Initially, Clients connect to a primary broker, no backup brokers accept client connections. If the primary fails, only then a new primary broker is created from the backups and other backups are connected to the new primary. The new primary selection is done by the cluster resource manager-rgmanager. rgmanagers support virtual IP (VIP). A VIP is an IP address that is assigned to multiple domain names or servers that share an IP address based on a single Network Interface Card. Even if the servers relocate, routing and DNS service are not necessary to be implemented.

Security in Qpid is achieved through authentication using SASL framework, GSSAPI (provides Kerberos authentication) CRAM-MD5, DIGEST-MD5 and plain SASL with SSL supplement. Anonymous can also be used. Authorization is done using ACL permissions and rules. Encryption is carried out using SSL. Encryption and certificate management for Qpid are provided by Mozilla's Network Security Services Library (NSS).The certificate database is created and managed by NSS.

Limitation: AMQP is still evolving and not yet stable; hence the implementations have versions of AMQP that are non-compliant with each other. Therefore, AMQP-based solutions would need continuous improvements to meet changes in AMQP [14]. Clients and Brokers need to be based on the same AMQP version; only then data transfer is possible.

5.3 RabbitMQ

RabbitMQ [8, 11] was developed in the year 2007 and is based on Mozilla Public license. It is a message-oriented middleware based on Erlang language which is especially suited for distributed applications, as concurrency and availability are well-supported. It runs on almost all major platforms (at least almost all places where Erlang/OTP runs). It supports application layer protocols such as, AMQP which is appropriate for heterogeneous environments, MQTT, REST, Streaming Text Oriented Messaging Protocol (STOMP), STOMP over WebSocket and Extensible Messaging and Presence Protocol (XMPP) over a gateway. Hence, RabbitMQ can be used to build ESB, due to STOMP over WebSocket support.

In RabbitMQ, Persistence is built-in and is controlled at the message level. The Erlang database-Mnesia is configurable to be either in RAM or disk, allows RabbitMQ to offer in-memory/disk based queues very easily, but there are DNS errors that cause the DB-Mnesia to crash at times. It supports transactions with unlimited queues. With respect to messaging, RabbitMQ TRANSACTED mode (not just durable/persistent) is necessary for guaranteed delivery of messages, and only persistent mode is not enough.

The Queue specifics of RabbitMQ allow multiple consumers to be configured for a single queue, and they all get mutually exclusive messages. It also supports multiple types of queues such as direct, fan-out, etc. so semantics such as broadcast to multiple clients listening on multiple queues is achieved. Messages that are unordered, not FIFO delivered or lost are auto-requeued (based on timeout). A single RabbitMQ instance doesn't scale to a lot of queues with each queue having fair load since all queues are stored in memory (queue metadata) and also in a clustered setup. Each queue's metadata (but not the queue's messages) is replicated on each node. Hence, there is the same amount of overhead due to queues on every node in a cluster. It is a message queue that can be used as a work queue as well but requires additional semantics such as burying jobs that need to be implemented by submitting a failed job to a "buried" queue.

Messaging semantics such as No ONCE-ONLY semantics hence, messages may be sent twice by RabbitMQ to the consumer(s).The consumer(s) has/have to do the rate limiting by not consuming messages too fast and not the broker itself adds responsibility to the end entities. It is basically a push model.

Redundancy and HA are built-in features in RabbitMQ and are available through the Erlang OTP platform. Multi-tenancy in RabbitMQ is supported via hosts. Security is attained at multiple levels. The management plug-in provides an appealing web console that allows easy administration with visualized statistics such as the number of messages per second and the consumption of resources, such as memory, sockets, file descriptors, etc.

5.4 ZeroMQ

ZeroMQ [9] a message-oriented middleware library is largely concerned with business-type systems. It has no open standard protocols at the application layer but is based on its own customized protocol. ZeroMQ uses different protocols depending on the peer's location (TCP, PGM multicast, IPC, inproc shared memory).

The core of the library is written in C; bindings for C++, Java (through JNI) and many more languages are supported. It runs on most platforms, even on LynxOS. ZeroMQ is available under the LGPLv3 with a static linking exception (even for iPhone apps). There is no commercial licensing alternative. ZeroMQ is basically a Brokerless solution. The direct connection between the system parts results in reduced maintenance costs as there is no need for brokers or daemons. The sender of a message is responsible for routing to the right destination and the receiver of a message is responsible for queuing, this shows the division of responsibility by the sender and receiver since it is a Brokerless solution.

ZeroMQ supports communication patterns such as request-reply, pub-sub, workload distribution and transports like in-process, inter-process, TCP and multicast achieving concurrency. It supports synchronous or asynchronous communication.

Serialization at the middleware end is not supported. ZeroMQ has no type specification and does not know anything about the data a user sends. For this reason, it is to be used with an external serializer. Message batching is supported with unlimited queues support. Message transfer is higher than any other solution in transient mode. ZeroMQ does not handle persistence, hence requires higher layers to manage it.

ZeroMQ has a small memory footprint since it is free of unnecessary dependencies. Routing in this middleware is available but is complex to implement. It is scalable. ZeroMQ has no mechanism to support failover and HA. It is possible to implement Enterprise Messaging system over ZeroMQ.

5.5 YAMI4

YAMI4 1.10.0 [3, 6] was developed in the year 2010. The messaging library is developed using C++, objective-C, C (Industry package) and the client APIs in Ada, C++, Java, .NET, Python.YAMI4 is guarded against GPLv3 or Professional package: Boost licensed. It functions both in

Brokered and Brokerless scenarios and has cross platform support.

YAMI4 supports both request-response and pub-sub pattern. Since the connections are messaging pattern independent, runtime decision to switch communication patterns is available; this achieves asynchronous type with full duplex communication in YAMI4.

Boost library is resource-heavy and isn't supported by Lynx OS and Unison platforms, for which YAMI4 has been implemented; hence boost is used in limited context. Thus, YAMI4 is lightweight and portable due to lesser external library dependency. Socket implementation in YAMI4 is based on TCP sockets. For windows, Winsock and for other OS, C++ based POSIX library is used. Thread model in YAMI4 is as follows: a main thread, a single thread for handling I/O events and another dispatcher thread for processing events are used. Dispatcher threads are limited in number. The I/O thread accept requests from an application and is released for new incoming requests; the processing is given to dispatcher threads, hence non-blocking is achieved, this helps to scale applications. Reliability is achieved in YAMI4; using a single thread, by preventing deadlocks and allowing thread safety. Programs are allotted their own private allocator; hence interference is less leading to reliability.

Persistence is not supported in YAMI4. Routing in YAMI4 is either Point-to-point or pub-sub which is implemented using multiple tag matching and hierarchy concept. Priority queue in YAMI4 is implemented using dynamic lists. Queue Overflow policies such as: reject message, drop message and update message are specified. Messaging Semantics supported in YAMI4 are, Delivery policies-at-most once, Message size/format- raw data and typed data, Ordering, Prioritization, Message acknowledgments, Purging, etc. Design pattern support in YAMI4 shows no singleton pattern, no shared memory concept, and memory is partitioned for each block.

For Serialization, raw binary data or custom serialization schemes are used. In addition to the standard data model and the parameters object as its implementation, YAMI4 allows using raw binary data for efficient transfer of opaque data and custom serialization schemes that allow integrating other models like XML, JSON, ASN.1, etc.

Using the concept of Clustering and federation of brokers, load balancing and failover are achieved. For e.g., Suppose a message is to be routed via a server, a set of target servers are specified i.e., failover: (tcp://somehost:12345| tcp://otherhost:12345), first tcp://somehost:12345- target1 is checked, if not available, then tcp://otherhost:12345- target 2 is opted for. If the transmission is successful to the 1st target, then 2nd target is not even checked. Hence it is synchronous communication for failover. For load balancing, when both the target servers are available, any random server is selected and message is routed through it. HA is achieved through forwarding principle, in which brokers forming a cluster transfer messages to other brokers within the cluster. This allows subscribers listening on a different server to receive messages being published by publishers on a different server.

Error codes and exceptions are provided for debug logs, and system state before the crash is preserved for recovery purpose.

From Security context, SSL support is available, Digital signatures can also be used for data security. Access controls and encryption mechanisms can be implemented at the application layer. A feature to monitor the health of the message broker is available through the 'event monitors'.

YAMI4 has its own **wire level** protocol (YAMI4) and does not support any available standards; hence interoperability between systems is not possible. For interoperability, application level efforts are to be taken.

Table 1 shows evaluation of middleware solutions in a nutshell.

6. RELATED STUDIES

The above comparison table deals with theoretical parameters for evaluation purpose. There have been several works which show practical analysis of solutions based on the criteria set.

The paper [1] shows an experimental analysis of AMQP and MQTT protocols over mobile networks finds out the applicability of the protocols for such unstable networks. It infers that both the protocols definitely account for jitter, but no message losses are found. In message burst conditions, AMQP follows LIFO ordering i.e. messages are fetched in a reverse order at the receiver, which is not the case with MQTT. Considering message payload, MQTT has a larger payload capability than AMQP. This is because of only a 2-byte header in MQTT, with a much larger header of 8-bytes in AMQP. Hence, it recommends using MQTT for energy-efficient requirements and AMQP for security aspects. In [3], the tests have been done for Controls Middleware project to operate CERN accelerators. The authors test ZeroMQ, Apache Qpid (AMQP) and YAMI4 based on request-reply and pub-sub patterns for throughput and scalability factors respectively. They conclude ZeroMQ is faster due to the automatic message batching. YAMI4 and Qpid behave average with a message transfer rate of 3500 and 3200 messages/sec respectively. They also prove that YAMI4 does not scale well with an increase in number of clients as compared to ZeroMQ.

In [12], ZeroMQ, RabbitMQ (AMQP, STOMP), Apache Qpid (AMQP) are evaluated for various scenarios of enqueues and de queues. The conclusions obtained are, ZeroMQ is the best one for simple architecture requirements, RabbitMQ outperforms all but with the fact, that it is based on AMQP and not STOMP. Apache Qpid behaves optimally in no-persistence mode.

In [13], Stress testing of Mosquitto broker based on MQTT protocol is done on Linux/Unix-like systems. The results obtained show that the broker has the capability of handling 20000 client connections, with a message transfer rate of 7000 messages/sec. The CPU usage statistics show single core usage and a memory usage of 0.3%.

[15, 16] show that even though ZeroMQ is the fastest in sending messages, the reception rate is slow, this creates a large disparity in sending and receiving of messages. There is a possibility of even message loss during the process. Hence, ZeroMQ does not provide guaranteed delivery.

6.1 Discussion

Based on the studies, ZeroMQ is a Brokerless solution in which responsibility is shared among the sender and receiver applications both.

RabbitMQ performs best in Brokered category, but its advanced features make the library heavy. It does not support C/C++ as the development language, which is an important requirement for modern software development.

Mosquitto is a very lightweight messaging library, but the protocol does not come with all the functionalities such as message priority and routing built-in, and the development efforts increase for incorporating such functions.

Therefore, Apache Qpid and YAMI4 are tested and analyzed further for deployment in Industry 4.0 platforms.

Table 1. Comparison Table

Middleware/ Features	**ZeroMQ**	**RabbitMQ**	**Apache Qpid**	**YAMI4**	**Mosquitto**
Language used for development	C++	Erlang	JAVA, C++	C++, Objective-C	C
Age of the middleware	2007	2007	2005	2010	2009
Application using it	Hootsuite Mongrel, Zato, Zero Cache	UIDAI, Google Compute Engine, Mozilla, AT&T	Used in a PowerVC environment within IBM Power Virtualization Center	Intel Galileo	Facebook Messenger, Mobile Platforms
Middleware Paradigm	Message-oriented	Message-oriented	Message-oriented	Message-oriented	Message-oriented
Broker/ Brokerless	Brokerless	Brokered	Brokered From AMQP 1.0. Brokerless form can also be implemented.	Can be used both as Brokered and the Brokerless solutions.	Brokered
Messaging patterns supported	Request-Response, Pub-sub, Workload distribution	Request-Response, Pub-sub	Request-Response, Pub-sub	Request-Response, Pub-sub	Only Pub-sub
Support for persistence	NO (needs to be handled at the application layer)	YES	YES	NO	YES
Lightweight	YES	NO	YES	YES	YES
Application protocol supported	ZMTP	AMQP, MQTT, REST, STOMP, STOMP over WebSockets	AMQP	YAMI4- a WIRE level protocol	MQTT
HA support	NO	YES	YES	YES	Not Directly, tries to do this through bridging between two brokers.
Routing support	YES (complex to implement)	YES	YES (through AMQP)	YES	NO
Priority Queue	NO	YES	YES	YES	NO

Licensing	LGPLv3	DUAL (Open Source for Development) and (Commercial for Support).	Open Source	GPL (open source applications) & Commercial License (closed source)	Open Source (BSD)
Default Config Settings	NO	Through environment variables	Through command line and XML file	Through config file	N.A
Royalty structure	NO	TBD	NO	NO	NO

7. AN EXPERIMENTAL STUDY: YAMI4 VERSUS APACHE QPID

Based on the above study, YAMI4 and Apache Qpid are analyzed on performance parameters such as Throughput, Latency, Memory Footprint and CPU usage when in-process. The test setup consists of a Publisher, a Subscriber and a Broker running on the same machine with machine configuration, Windows 10 DELL Inspiron15, Intel core i5 - 4200U CPU @1.60 GHz 2.30 GHz, 64 bit OS, 6GB RAM.

The test condition is Publisher sends n messages of m bytes to the broker, based on a topic. A Subscriber subscribes to its topic of interest at the broker. The Broker then matches the topic received from the Publisher to that of the topic registered by the Subscriber, if the match is successful, only then the Broker forwards messages of m bytes to the Subscriber. This paper prepares the test bed for finding out YAMI4's performance and uses Apache Qpid's benchmarking tools such as qpid_perftest.exe and qpid_latency.exe to find out its throughput and latency measures.

7.1 Throughput

7.1.1 When a number of messages are constant:

It is the amount of messages received per unit time by the subscriber. The general formula to calculate throughput is as follows:

$$throughput = \frac{n}{t_{total\_time}} \qquad (1)$$

$$t_{total\_time} = t_{s\_stop\_time} - t_{p\_start\_time} \qquad (2)$$

where n is the total number of messages of m bytes sent by the Publisher, t_{total_time} is total time required for n messages to traverse from publisher to subscriber, $t_{s\_stop\_time}$ is time of the n th message recorded at the subscriber, $t_{p\_start\_time}$ is the time of the 1st message at the publisher. Fig. 3 shows throughput values for n = 1million and m = 1024 bytes.

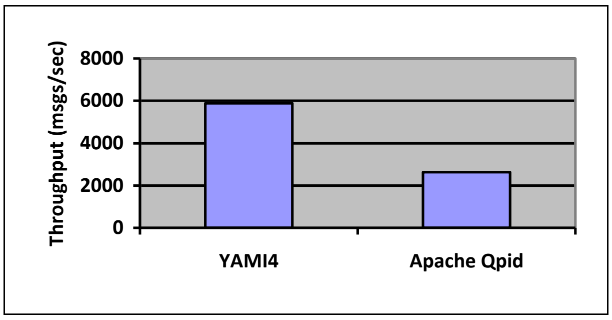

Fig 3: Throughput (no: of messages are constant) YAMI4 versus Qpid

7.1.2 When time constant

Throughput is calculated when time = 1 second is set at both the Publisher and the Subscriber, and messages of m (1024) bytes are pushed. At the end of 1 second, Publisher and Subscriber both stop and amount of messages sent to received are calculated. Ideally,

$$\text{number of messages received} \le \text{number of messages sent}$$

In Fig. 4a and 4b, $t_{p0}, t_{p1}, t_{s0}, t_{s1}$ are time instants, when the 1st message is sent, when the time elapsed is 1 second at the Publisher, when the 1st message is received at the Subscriber, when time elapsed is 1 second at the Subscriber. $count\_a, count\_b$ are number of messages sent and received at respective ends after a time frame of 1 second.

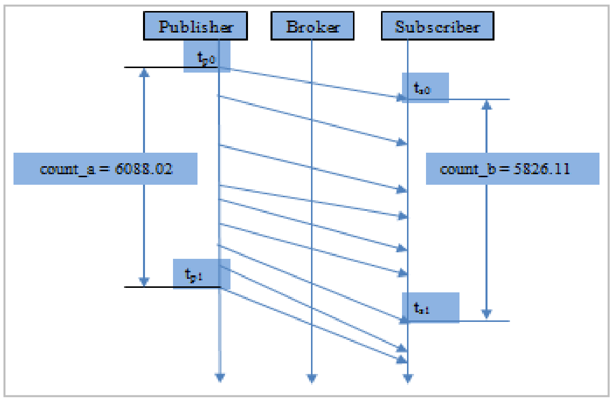

Fig 4a: YAMI4 throughput

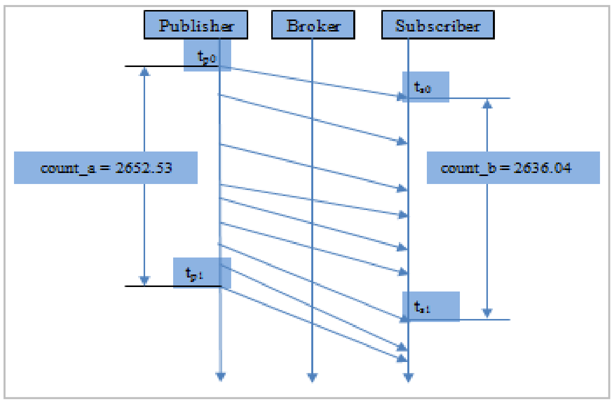

Fig.4b: Apache Qpid throughput

7.2 Latency

It is the time required for a single message to traverse between endpoints i.e. from Publisher to Subscriber.

Latency is calculated using formula

$$\begin{aligned} lat_1 &= t_{s1} - t_{p1} \\ &\vdots \\ lat_n &= t_{sn} - t_{pn} \end{aligned} \qquad (3)$$

where t_{p1}, t_{s1}, lat_1 are time instants, when the 1st message is sent at the Publisher end, received at the Subscriber end and the time required for the message to travel from Publisher to Subscriber respectively. Similarly, (3) shows the latency of each message for all n messages. Fig. 5 shows Min, Max, Average latency of the Brokers. Min, Max values represent the minimum and the maximum latency of all the messages in the system. Average latency using (4) is found out since latency values of all the messages are not the same.

$$lat_{avg} = \frac{lat_1 + lat_2 + lat_3 + \cdots + lat_n}{n} \qquad (4)$$

where t_{lavg} is the average latency of each message in the system. Here, n = 1million and m = 1024 bytes.

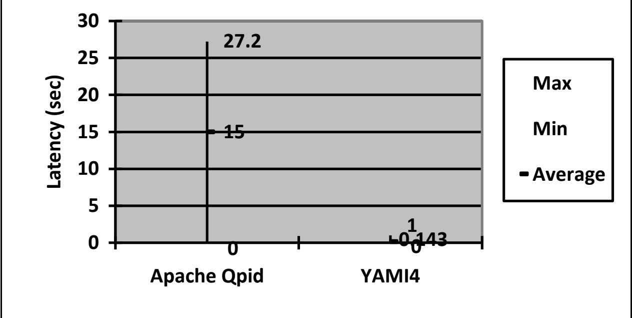

Fig 5: Latency measure

7.3 Memory footprint and CPU usage

Table 2 shows the usage measures when messages are in transit and when the Broker is in an idle (steady) state.

Table 2. Memory and CPU usage statistics

Features/Broker	**ApacheQpid0.34 (non-persistent mode)**	**YAMI4 1.10.0**
Memory Footprint(when transferring 10lakh messages of 1024 bytes)	**4.2- 44 MB of memory, with 36-44 % CPU usage**	**0.6 MB of memory, with 22-24 % CPU usage.**
Memory Footprint(in steady state)	**6.8 MB of memory, with 0.4 % CPU usage**	**0.6 MB of memory, with 0% CPU usage.**

7.4 Conclusions from the experiment

Throughput measure of YAMI4 is better as shown in Fig 3. Fig. 4a and Fig. 4b show the disparity in messages sent to received, which is high in Apache Qpid than in YAMI4.

As shown in Fig. 5, Latency which is known as delay otherwise is more in Apache Qpid than YAMI4, is undesirable.

YAMI4 is lightweight, due to lesser external library dependency factor.

Hence, YAMI4 as a message broker is an optimal solution for Industry 4.0 platforms.

8. YAMI4

Messaging fundamentals such as message priority and queuing, and internal socket mechanism are tested to get better insights into YAMI4.

8.1 Message priority

YAMI4 defines message priority in its messaging API, which is absent in TCP/IP stack. This helps to deliver messages of maximum importance prior to other messages. Implementing message priority nulls the possibility of batching messages to achieve better throughput measures, which other Message brokers implement. Such an effort was consciously made by YAMI4 [5]. Hence, YAMI4 is not the fastest broker but a reliable one that delivers message of high importance first.

To check whether and in which conditions, priority behaviour is reflected, the below tests are conducted.

Priority definition as defined in the library: 0 - least prior, 1- prior, 2- most prior.

Figure 6 shows the scenarios used for analysis.

A channel is a connection between the client and the server i.e. between the publisher to the broker & the broker to the subscriber.

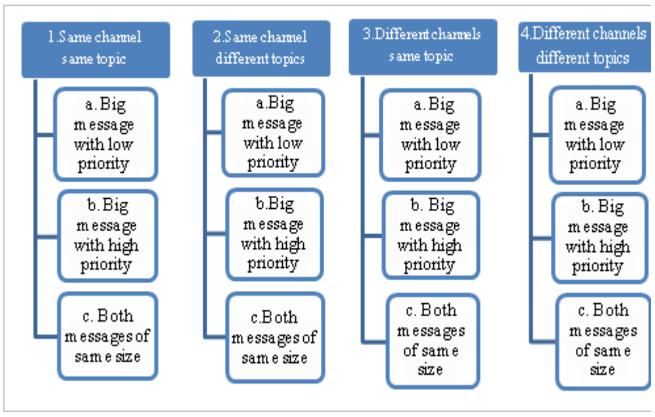
1.Same channel same topic
a. Big message with low priority
b. Big message with high priority
c. Both messages of same size
2.Same channel different topics
a.Big message with low priority
b.Big message with high priority
c.Both messages of same size
3.Different channels same topic
a. Big message with low priority
b. Big message with high priority
c. Both messages of same size
4.Different channels different topics
a. Big message with low priority
b. Big message with high priority
c. Both messages of same size

Fig 6: Message priority test scenarios

Table 3 shows that prioritized messages are queued out based on the level of importance, but this is observed only in cases where message size is relatively large than other messages in the system. Hence priority works only in conditions, where network traffic is at peak, or the receiver at the other end is slow in receiving, which leads to buffering of messages in the queue.

Table 3. Message priority results table

Message published sequence		Message received sequence	
message size (in characters)	priority assigned	When messages published once	When messages published continuously &with a sleep of 5 seconds at the subscriber
1. Same channel and same topic			
a. 3*(10^5) 3*(10^4) 3	0 2 1	2 1 0	2 1 0 (repetitive in 2 1 0)
b. I} with 3 messages 3 3*(10^5) 3*(10^4)	0 2 1	0 2 1	0 2 1 (repetitive in 0 2 1)
II}with 2 messages 3 3*(10^4)	0 1	0 1	0 1 (repetitive in 0 1)
III}with 2 messages 3 3*(10^5)	0 2	0 2	0 2 (repetitive in 0 2)
c. I} big message 3*(10^4) 3*(10^5)	0 2	0 2	2 0 (repetitive in 2 0)
II}small message 3*100 3*100	0 2	0 2	0 2 (repetitive in 0 2)
2. Same channel and different topics			
a. 3*(10^5) 3*(10^4) 3	0 2 1	2 1 0	2 1 0 (repetitive in 2 1 0)

b. I} with 3 messages 3 3*(10^5) 3*(10^4)	0 2 1	0 2 1	0 2 1 (repetitive in 0 2 1)
II} with 2 messages 3 3*(10^4)	0 1	0 1	0 1 (repetitive in 0 1)
III}with 2messages 3 3*(10^5)	0 2	0 2	0 2 (repetitive in 0 2)
c. I} big message 3*(10^4) 3*(10^5)	0 2	0 2	2 0 (repetitive in 2 0)
II} small message 3*100 3*100	0 2	0 2	0 2 (repetitive in 0 2)
3. Different channels and same topic			
a. 3*(10^5) 3*(10^4) 3	0 2 1	0 2 1	(random order- no sequence could be determined)
b. I} with 3 messages 3 3*(10^5) 3*(10^4	0 2 1	0 2 1	(random order- no sequence could be determined)
II}with 2 messages 3 3*(10^4)	0 1	0 1	0 1 (alternative in 0 1)
III}with 2messages 3 3*(10^5)	0 2	0 2	0 2 (alternative in 0 2)
Miscellaneous (Sent continuously) 3 3*(10^7)	0 2	Not applicable	0 (4 times) 2 (2 times)
c. I} big message 3*(10^4) 3*(10^5)	0 2	0 2	0 2 (alternative in 0 2)
II}small message 3*100 3*100	0 2	0 2	0 2 (alternative in 0 2)
4. Different channels and different topics			

a. 3*(10^5) 3*(10^4) 3	0 2 1	0 2 1	(random order- no sequence could be determined)
b. I} with 3 messages 3 3*(10^5) 3*(10^4)	0 2 1	0 2 1	(random order- no sequence could be determined)
II }with 2messages 3 3*(10^4)	0 1	0 1	0 1 (alternative in 0 1)
III}with 2messages 3 3*(10^5)	0 2	0 2	0 2 (alternative in 0 2)
Miscellaneous (Sent continuously) 3 3*(10^7)	0 2	Not applicable	0 (4 times) 2 (2 times)
c. I} big message 3*(10^4) 3*(10^5)	0 2	0 2	0 2 (alternative in 0 2)
II}small message 3*100 3*100	0 2	0 2	0 2 (alternative in 0 2)
Miscellaneous (sent once) 3*(10^7) 3	1 0	0 1	Not applicable

8.2 Message Queue

The message queue is used in conditions, where the network is saturated, or subscriber is slow so that the messages are buffered, preventing losses. In YAMI4, each channel has a single queue and NOT multiple queues based on topics.

Reasoning: Considering test scenarios 1 & 2 in Table 3, queue full occurs at same published message count for both the cases; else it would have occurred for higher message counts. Hence there is NO topic based queue in the broker structure.

8.3 Maximum number of socket connections

The number of socket connections in YAMI4 is bounded by the select call of Winsock library. Hence, for a specific fd_setsize (a select() parameter), a specific set of connections are achieved. For example, 72 connections for fd_setsize = 64, 1033 for fd_setsize = 1024, and so on. Therefore, a number of socket connections limit the maximum number of client connections possible in YAMI4 messaging framework.

9. OPEN ISSUES AND SOLUTIONS

This paper also proposes the best-known solutions for Open issues based on specific middleware use-cases. They are:

9.1 For Web-based solution

Conventionally, Web-based solutions used HTTP as the underlying application protocol, but that does not prove performance-efficient for real-time messaging. Hence, WebSocket protocol [20] (TCP-based) is specifically developed for real-time applications that provide long-lived persistent connections.

This paper proposes WebSocket protocol implementation for Web-based communication at the message broker level.

9.2 Application-level security

Security is a trending issue that needs to be taken care of at various levels. At the middleware level, secure messaging solutions must be based on certain goals and essentials as:

Authentication: the process of determining whether someone or something is, in fact, who or what it is declared to be [17].

Confidentiality: the process of making the information private that cannot be understood by anyone, other than for whom it was intended [18].

Integrity: with no alteration of information, whether in transit or in stored state or the alteration that gets detected. [18].

Security essentials of a middleware:

a) Publishers and Subscribers must be authenticated primarily.

b) Data Confidentiality must be achieved between publisher and subscriber.

c) The Publisher must not know the Subscriber's details and the Subscribers must also not know the details of other subscribers present in the system.

d) The routing framework should not be exposed to the event contents or the event subscription. Such a security fundamental is useful for content-centric networks.

e) User's access must be restricted to specific resources, so as to control overloading of the physical resources.

Hence, this paper proposes Authentication and Data confidentiality between the message broker and its associated clients.

9.3 Persistence
Persistence in middleware has a scope at Data, Subscription and Queue level.

In memory (RAM) persistence mechanism is the best approach, since accessing data at run time requires less time.

9.4 Queuing
Queuing is to be taken care of since messages that do not get delivered to the subscribers are to be queued until the subscriber is up.

Queue creation mechanisms using Custom Database or No SQL (key-value, document, and graph data structures) are some of the suitable solutions. The benefit of No SQL is it supports co-operative multitasking, and thread model based on Co-operative multitasking lowers overhead of message queues for data exchange.

9.5 Routing
Routing methodology depends on the type of middleware. For example, Content-based routing is a desired mechanism Content-centric networks.

9.6 Support for messaging/communication patterns
For a middleware to be an all-purpose one, Messaging patterns to be supported are, ACTive (Availability for Concurrent Transactions) – unique to XMPP, Pipeline (for aggregation and load-balancing) and Survey (a single request for the state of multiple applications).

9.7 Provision for ESB support
Enterprise service bus is the one that connects the modern software platform to the higher business layers. This deals with implementing service-oriented middleware functionalities within Message Oriented middleware.

9.8 Support for Event Driven Scenarios
Event Driven Scenarios imply commonly used services to be active all the time and non-frequent services to be executed on a periodic basis to process requests. Such scenarios must be implemented at the middleware layer to achieve low load solutions.

9.9 Listeners requiring re-querying for best sources even after finding an acceptable publisher
This signifies a possible use-case for systems that decide the best possible solution to a problem intelligently.

9.10 Support for delayed jobs
Middleware that require some intentional delay must provision for delayed jobs.

10. CONCLUSIONS AND FUTURE WORK
Modern software platforms adopt the middleware approach based on an event-driven architecture and pub-sub model for data handling and monitoring. This paper, therefore, reviews and analyzes middleware solutions based on Object, Service, Data and Message paradigms. The solutions such as RabbitMQ, Apache Qpid, ZeroMQ, Mosquitto and YAMI4 are reviewed and analyzed based on messaging semantics viz. message routing, messaging patterns supported, message priority, throughput, latency, memory footprint, etc. Based on the experimental analysis, YAMI4 is found to be suitable for Industry 4.0 platforms.

YAMI4 is experimentally tested for its internal fundamentals of message priority, message queuing and socket connections. The results show that message priority is achieved only in cases of network throttling and are FIFO-ordered otherwise. The tests also show that YAMI4 messaging library has a maximum number of socket connections limitation.

Further, the paper also focuses on the open issues of Middleware and the best-known solutions, based on specific middleware use-cases.

An extended work for enhancing YAMI4 from Web [20] and Application security context are also planned.

11. ACKNOWLEDGMENTS
The authors remain grateful to the anonymous reviewers.

12. REFERENCES
[1] Jorge E. Luzuriaga, et al. "A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks," in Conf. Proc. 12th Annual IEEE Consumer Communications and Networking Conference (CCNC), 2015, pp. 931-936.

[2] A Foster. (2014, July). "A Comparison Between DDS, AMQP, MQTT, JMS, REST and CoAP." Messaging Technologies for the Industrial Internet and the Internet of Things. [On-line]. 1.7, pp. 1-25. Available: www.prismtech.com.

[3] Andrzej Dworak, et al. "Middleware trends and market leaders 2011," in Conf. Proc. Vol. 111010. No. CERN-ATS-2011-196, 2011.2015, pp. 931-936.

[4] Tarun Agarwal. "Know all about SCADA Systems Architecture and Types with Applications." Internet: http://www.edgefxkits.com/blog/scada-system-architecture-types-applications/, Sep. 19, 2014.

[5] "YAMI4 vs ZeroMQ." Internet: http://www.inspirel.com/articles/YAMI4_vs_ZeroMQ.html.

[6] "YAMI4." Internet: http://www.inspirel.com/yami4/.

[7] "Apache Qpid." Internet: https://qpid.apache.org/.

[8] "RabbitMQ." Internet: https://www.rabbitmq.com/.

[9] "ØMQ Community." Internet: http://zeromq.org/community.

[10] Margaret Rouse. "event-driven architecture (EDA)." Internet: http://searchsoa.techtarget.com/definition/event-driven-architecture-EDA, 2011.

[11] "RabbitMQ vs Apache ActiveMQ vs Apache qpid ." Internet: http://bhavin.directi.com/rabbitmq-vs-apache-activemq-vs-apache-qpid/, May 6, 2010.

[12] "A quick message queue benchmark: ActiveMQ, RabbitMQ, HornetQ, QPID, Apollo." Internet: http://www.voidcn.com/blog/hanruikai/article/p-

[13] Rex Xia."Stress testing Mosquitto MQTT Broker." Internet: http://rexpie.github.io/2015/08/23/stress-testing-mosquitto.html, Aug. 23, 2015.

[14] "Selecting a Message Queue – AMQP or ZeroMQ." Internet: http://bhavin.directi.com/selecting-a-message-queue-amqp-or-zeromq/, Apr. 4, 2010.

[15] Tyler Treat. "Dissecting Message Queues." Internet: http://bravenewgeek.com/dissecting-message-queues/, Jul. 7, 2014.

[16] "Message Queue Shootout." Internet: http://mikehadlow.blogspot.in/2011/04/message-queue-shootout.html, Apr. 10, 2011.

[17] Margaret Rouse. "What is authentication." Internet: http://searchsecurity.techtarget.com/definition/authentication, 2007.

[18] "Cryptography Defination." Internet: http://searchsoftwarequality.techtarget.com/definition/cryptography, 2014.

[19] "Publish/Subscribe." Internet: https://msdn.microsoft.com/en-us/library/ff649664.aspx, Jun. 2004.

[20] Peter Lubbers and Frank Greco. "HTML5 Web Sockets: Internet: A Quantum Leap in Scalability for the Web." Internet: https://www.websocket.org/quantum.html, Mar. 2010.