Comparative Study of Middleware solutions For Control and Monitoring systems

Suman Patro
Department of Computer Engineering
K. J. Somaiya College of Engineering
Mumbai, India
suman.patro@somaiya.edu

Prof. Manish Potey
Department of Computer Engineering
K. J. Somaiya College of Engineering
Mumbai, India
manishpotey@somaiya.edu

Amit Golhani
Electrical & Automation
Larsen & Toubro Ltd.
Navi Mumbai, India
amit.golhani@lntebg.com

Abstract— **Communication systems in today's world are no longer traditional in nature. Platforms required for Industry 4.0 that fall under Control and Monitoring systems, are data-gathering units which gather data from remote devices and monitor them. Such platforms have various components which communicate with each other for data handling and monitoring. In the traditional peer-to-peer model, every module communicates with every other in a point-to-point manner that leads to connection overheads; the modules also have to understand the semantics of every other module in the system. This proves inefficient for a system where data influx and outflux is in huge amount. Therefore, a Middleware approach based on the publish-subscribe (pub-sub) model is used where modules communicate with the middleware instead of the connected peers, resulting in the performance gain of the system. Middleware is a software that connects software components or enterprise applications. This paper focuses on the study and comparison of middleware solutions(RabbitMQ, ZeroMQ, Mosquitto, Apache Qpid, YAMI4) based on parameters such as message throughput, latency, routing, messaging patterns supported, security, etc.YAMI4-message oriented middleware(Message Broker) proves suitable based on its optimal message throughput and latency for deployment in Control and Monitoring systems.**

***Keywords*— Middleware, publish-subscribe, RabbitMQ, ZeroMQ, Mosquitto, Apache Qpid, YAMI4, Message Broker**

I. INTRODUCTION

A. *Motivation*

Traditionally, a software package such as Supervisory Control and Data Acquisition Unit (SCADA) [8] employs polling-based data fetch that leads to the tracking of redundant data. This proves inefficient for systems where data influx and outflux is huge. Therefore, Event-driven architecture [14] is used in modern systems to track changed data. Event mechanism is modeled after a pub-sub design pattern [4, 5, 23]. 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 based on Object [5], Service [5], Data and Message [5] Paradigms is used to facilitate data dissemination between entities in modern software platforms. 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.

B. *Paper Organization*

The paper is organized as follows: Section II enlists the Master criteria set for middleware evaluation. Section III focuses on the theoretical understanding of the middleware solutions based on the enlisted criteria in Section II. Section IV gives a practical analysis of the middleware solutions based on the studies of various authors. Section V gives a detailed comparison report of YAMI4 & Apache Qpid based on their performance parameters and chooses YAMI4 as the optimal solution. Section VI focuses on YAMI4's internal mechanism of message ordering and queuing based on experimental tests. Section VII provides a summary of the paper and highlights further work to enhance YAMI4 message broker from Web context and Application-level security perspective.

II. CRITERIA FOR MIDDLEWARE EVALUATION

Middleware has feature set that complies with specific use-cases. Fig. 1 shows categorization of such features into levels of requirements of a system, viz. Fundamental, Mandatory or Desirable.

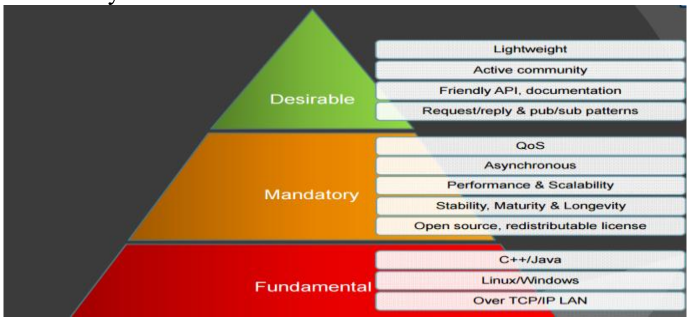
Desirable
Lightweight
Active community
Friendly API, documentation
Request/reply & pub/sub patterns
Mandatory
QoS
Asynchronous
Performance & Scalability
Stability, Maturity & Longevity
Open source, redistributable license
Fundamental
C++/Java
Linux/Windows
Over TCP/IP LAN

Fig. 1. 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 [6]. For example, WebSocket 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. the 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.
- 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, 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.
- CAP(consistency, Availability, Partition tolerance) support
- 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 or, 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 [21, 22].
- 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, but a 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 support 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
- ➢ Licensing - 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.

III. 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 applications. Hence, middleware requirements for Industry 4.0 platform comply with Data and Message paradigms.

This paper therefore surveys and compares solutions from Message-Oriented Middleware as shown below.

Table I shows the comparison of middleware solutions in a nutshell.

C. Mosquitto

Mosquitto [2] is designed to run on networks that use TCP/IP as the underlying transport. The MQTT [5] protocol has no request /response support, but only pub-sub style communication. It is designed for machine-to-machine (M2M) communication at device level along high latency or constrained networks, to a server or small message broker. Therefore, being lightweight is beneficial.

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, 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, a username and password are passed in an MQTT packet. This helps 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 is handled independently from the MQTT protocol using SSL or TLS.

D. Apache Qpid

Apache Qpid [11, 15] is a message-oriented middleware that stores, routes, and forwards messages using AMQP. It supports AMQP 1.0 and AMQP 0-10 at the application level and works on Linux and Windows platform. Pub-sub and request/reply (slower transfer) are supported in Qpid.

Number of Queues are unlimited and its size can be set, and 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. Exchange-Bind-Queue is the principle used in Qpid.

Clustering in Qpid is supported where all queue data and metadata is replicated across all nodes that make up a cluster, hence supports queue replication and transaction, but each client needs to know in advance which nodes make up the 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 through 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, 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.

E. RabbitMQ

RabbitMQ [12, 15] 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) and 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 that can be configured 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 an only persistent mode is not enough.

The Queue specifics of RabbitMQ allows multiple consumers to be configured for a single queue, and they all get mutually exclusive messages; 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 has 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 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.

F. ZeroMQ

ZeroMQ [9] is largely concerned with business-type systems, has no open standard protocols at the application layer but is based on 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), etc. are supported. It runs on most platforms, even on LynxOS and iPhone apps. It 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 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. 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.

G. YAMI4

YAMI4 (1.10.0) [3, 10] functions both in Brokered as well as Brokerless scenarios. It has Windows and POSIX systems support.

YAMI4 supports both request-response and pub-sub pattern. Since the connections are messaging pattern independent, runtime decision to switch patterns is available; this avails 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; therefore 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 used in YAMI4 is as follows: a main thread, a single thread for handling I/O events and dispatcher threads for processing I/O events. Dispatcher threads are limited in number. The single I/O thread accepts a request 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.

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, update message are specified. Messaging Semantics supported in YAMI4 are, Delivery policies-at-most once, Message size/format- raw data and typed data, Ordering, Message acknowledgments, Purging, etc. Design pattern support in YAMI4 shows no singleton or shared memory pattern but memory partitioning for each block.

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, automatic failover and HA is achieved. For e.g., If 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), a randomized server(either tcp://somehost:12345 or tcp://otherhost:12345) is selected and message is routed to the client through the randomized selected server. This achieves Load balancing. Automatic failover is achieved when one of the target servers is not available and the message is routed through other available target servers. Hence, after successful transmission to one of the targets, the other targets in the set are not even checked for availability. But, communication is synchronous in the case of failover, since next available target is selected only after non-availability of the first target is obtained. HA is achieved through forwarding subscriptions in which brokers within clustered installations act as subscriber clients to the central brokering service (part of the cluster). Hence, messages sent by publisher clients to the central broker service can also be received by subscriber clients listening on the forwarded brokers within the cluster.

Error codes and exceptions are provided for debug logs and system state before a 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.

IV. RELATED STUDIES

The above comparison table deals with theoretic parameters for evaluation purpose. There have been several works which show practical analysis of middleware solutions.

The paper [1] shows an experimental analysis of AMQP and MQTT protocols over mobile networks, to find 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, whereas 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 of all due to 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 a number of clients as compared to ZeroMQ.

In [16], ZeroMQ, RabbitMQ (AMQP, STOMP), Apache Qpid (AMQP) are compared for various scenarios of enqueues and de queues. The conclusions obtained are, ZeroMQ is the best one for simple architecture requirements, whereas RabbitMQ outperforms all only with the fact that it is based on AMQP and not STOMP. Apache Qpid behaves optimally in NO-persistence mode.

In [17], 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%.

[19, 20] 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 and messages might even get lost during this process. Hence, ZeroMQ does not provide guaranteed delivery.

A. 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 Control and Monitoring systems.

TABLE I. 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 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	YES	YES	NO	YES
Lightweight	YES	NO	YES	YES	YES
Application protocol supported	ZMTP	AMQP,MQTT,REST, STOMP, STOMP over websockets XMPP over gateway	AMQP	YAMI4- a WIRE level protocol	MQTT, Websocket
HA support	NO	YES	YES	YES	Not Directly , tries to do this through bridging between two brokers.
Routing support	YES	YES	YES(through AMQP)	YES	NO
Priority Queue	NO	YES	YES	YES	NO
Licensing	LGPLv3with a static linking exception	Open Source for Development and Commercial for Support	Open Source (Apache 2.0)	GPL (open source applications) & Commercial License (closed source)	Open Source (EPL/EDL)

V. 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, Server scalability, Memory Footprint and CPU usage (in process).

The test setup consists of a Publisher, a Subscriber and a Broker running on the same machine with machine configuration, Windows HP Intel core i5 -4590S CPU @3.0 GHz 3.0 GHz, 64 bit OS, 8GB 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 its performance measures.

A. Throughput

1) Entire system

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

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

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

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 the 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. 2 shows throughput values for n = 1million and m = 1024 bytes.

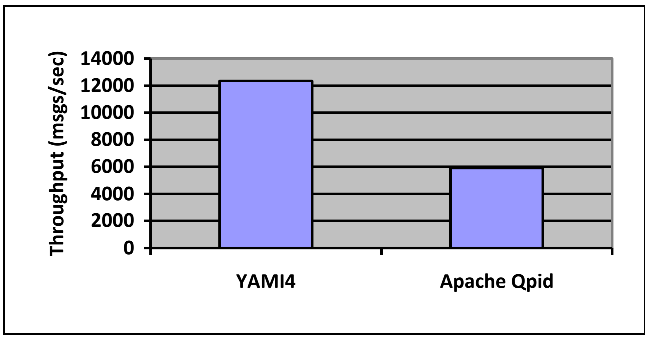

Fig. 2. Throughput (of the entire system) YAMI4 versus Qpid

Fig. 3 show throughput measures of YAMI4 and Apache Qpid for varying message count and message size. Fig. 3 shows a linear decrease in message rate in YAMI4 and Qpid with increase in message count. Fig. 4 shows that message rate in YAMI4 decreases with increase in message size, whereas it is constant in the case of Apache Qpid. This shows that YAMI4 does not support message batching [13].

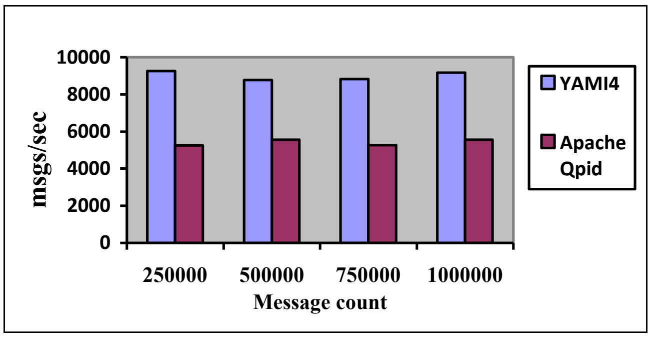

Fig. 3. For varying message counts

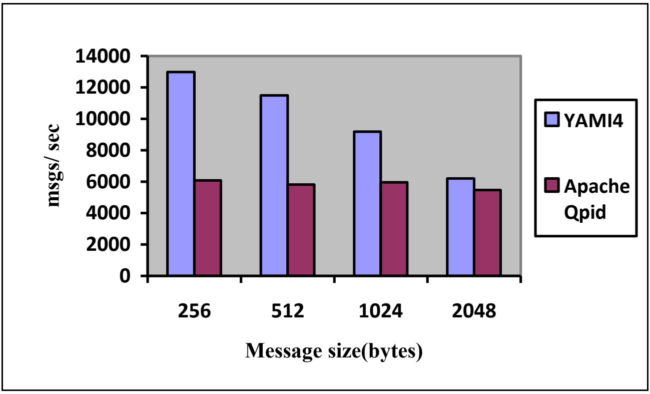

Fig. 4. For varying message size

2) At the Publisher and Subscriber ends

Throughput at the Publisher and subscriber ends denote number of messages sent to received per unit time.

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

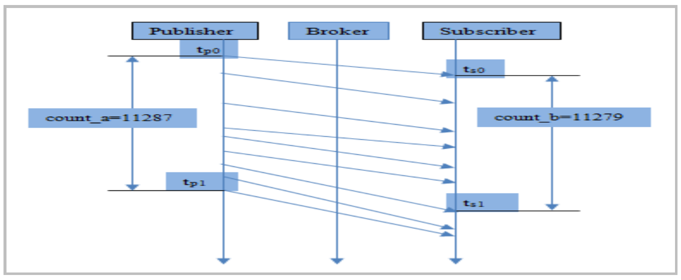

Fig.5a. YAMI4 throughput

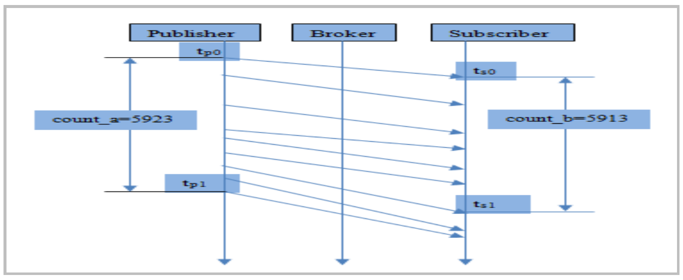

Fig5b. Apache Qpid throughput

subscriber both stop and amount of messages sent to receipt is calculated. Ideally,

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

In Fig. 5a and Fig. 5b; $t_{p0}, t_{p1}, t_{s0}, t_{s1}$ are time instants when the 1st message is sent, time elapsed is 1 second at the Publisher, the 1st message is received at the Subscriber, time elapsed is 1 second at the Subscriber and $count\_a, count\_b$ are a number of messages sent and received at respective ends after a time frame of 1 second. Fig.6. shows similar result as shown in Fig.4.

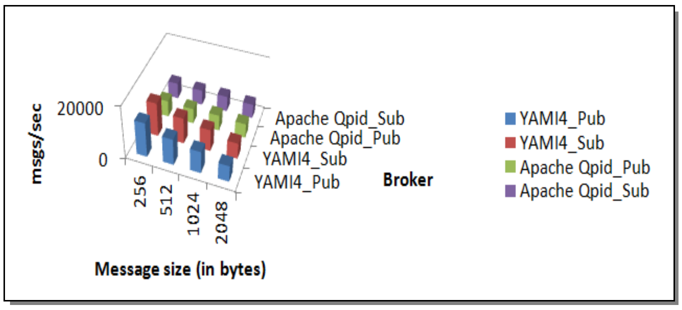

Fig. 6. For varying message size

B. 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

$$\begin{array}{c} lat_1 = t_{s1} - t_{p1} \\ | \quad | \quad | \\ lat_n = t_{sn} - t_{pn} \end{array} \qquad (3)$$

where t_{p1}, t_{s1}, lat_1 are time instants when the 1st message is sent at the Publisher, received at the Subscriber 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. 7 shows Min, Max, the Average latency of the Brokers. Min, Max values represent the minimum, the maximum latency of all the messages in the system. Average latency is found out, since latency values of all the messages are not same, using (4).

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

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

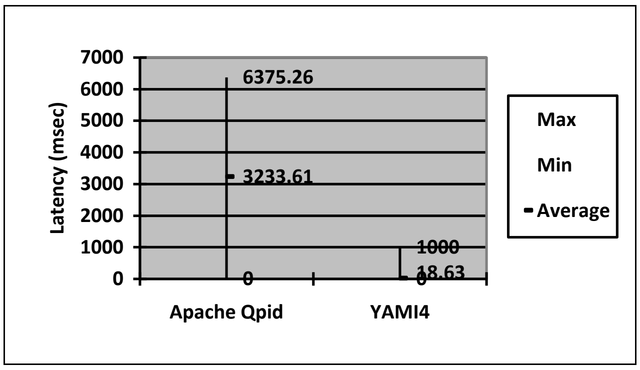

Fig. 7. Latency Measure

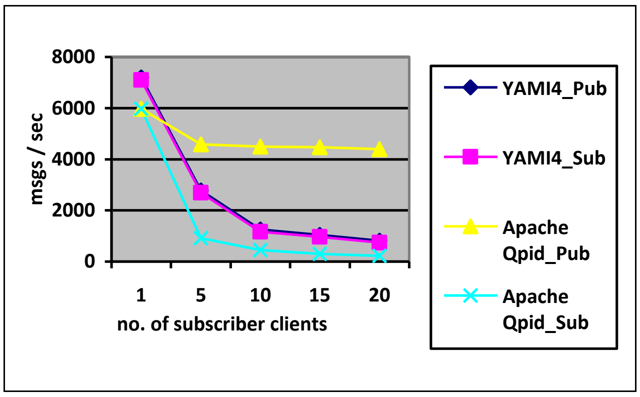

Fig. 8. Server scalability

C. Server scalability

Server scalability is the number of client connections that are supported by the server (Broker) over a performance benchmark. Scalability of the broker is tested for client connections ranging from 1, 5, 10, 15 and 20 subscribers and a single publisher running on the same machine. Test scenario comprises of calculating number of messages (of 1024 bytes) published to the number of messages received at the subscribers per unit time.

Fig. 8 shows that with increase in number of clients, rate of YAMI4 publisher decreases exponentially, on the contrary to Apache Qpid which remains almost constant. But the number of messages received at the subscriber clients of Apache Qpid are much less than those of YAMI4. This proves that ratio of messages sent to received is more, leading to decrease in messages received at the subscriber in Apache Qpid as compared to YAMI4. Therefore, YAMI4 is much more efficient in message handling in case of scalability.

D. Memory footprint and CPU usage

The table II shows the usage measures when messages are in transit and when the Broker is in idle (steady) state.

TABLE II. 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)	4MB of memory , with 51% CPU usage	808 KB of memory, with 27-29% CPU usage.
Memory Footprint(in steady state)	3.8MB of memory, with 0% CPU usage	808KB of memory, with 0% CPU usage.

E. Conclusions from the experiment

Throughput measure of YAMI4 is better as shown in Fig. 2 and Fig.3. Fig. 5a and Fig. 5b show a disparity in messages sent to receipt, which is high in Apache Qpid than in YAMI4. A similar scenario is achieved for increasing client connections.

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

YAmi4 is also lightweight due to lesser external library dependency factor.

Hence, YAMI4 as a message broker is an optimal solution for Control and Monitoring systems.

VI. YAMI4

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

A. Message ordering

YAMI4 orders messages in a prioritized fashion, on the contrary to other performance-efficient middleware that batch messages to achieve better throughput measures. Such an effort was consciously made by YAMI4 [13]. Therefore, even though YAMI4 is not the fastest message broker, it is also not late [13] in delivering messages of maximum importance.

Since, message priority is absent in TCP/IP stack, YAMI4 has defined and implemented priority at the application level. 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.

Fig. 9 shows the test scenarios used for analysis.

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

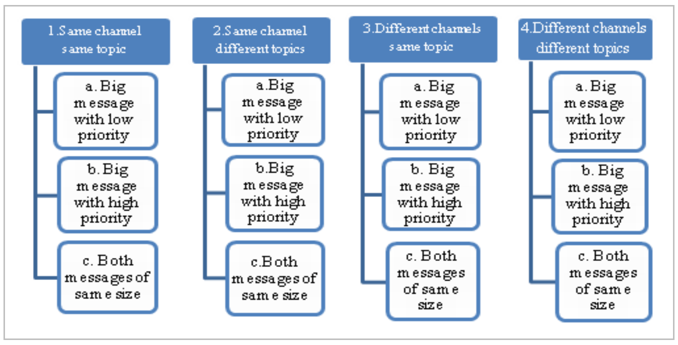
1.Same channel same topic
a. Big message with low priority
b. Big message with high priority
c. Both messages of same size
2.Same channel different topics
a.Big message with low priority
b.Big message with high priority
c.Both messages of same size
3.Different channels same topic
a. Big message with 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. 9. Message priority test scenarios

Fig. 10 shows message size used for message priority analysis.

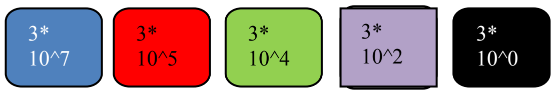

Fig. 10. Message size Categories

Table III shows message published sequence (at the publisher) and received sequence (at the receiver). The symbols used in Table III are defined as follows:

- Shows received message when message published is once.

...- Shows received message when messages are published continuously and with a sleep of 5 seconds at the subscriber.

It 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.

B. 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, 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.

TABLE III. MESSAGE PRIORITY RESULTS TABLE

Published sequence		Received sequence	
			...
1.a	0 2 1	2 1 0	2 1 0
b. I	0 2 1	0 2 1	0 2 1
b. II	0 1	0 1	0 1
b. III	0 2	0 2	0 2
c. I	0 2	0 2	0 2
c. II	0 2	0 2	0 2
2.a	0 2 1	2 1 0	2 1 0
b. I	0 2 1	0 2 1	0 2 1
b. II	0 1	0 1	0 1
b. III	0 2	0 2	0 2
c. I	0 2	2 0	2 0
c. II	0 2	0 2	0 2
3.a	0 2 1	0 2 1	random order
b. I	0 2 1	0 2 1	random order
b. II	0 1	0 1	0 1
b. III	0 2	0 2	0 2
	0 2	Not applicable	random order
c. I	0 2	0 2	0 2
c. II	0 2	0 2	0 2
4.a	0 2 1	0 2 1	random order
b. I	0 2 1	0 2 1	random order
b. II	0 1	0 1	0 1
b. III	0 2	0 2	0 2
	0 2	Not applicable	random order
	1 0	0 1	Not applicable
c. I	0 2	0 2	0 2
c. II	0 2	0 2	0 2

VII. 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, compares and studies middleware solutions based on Message paradigms. The solutions such as RabbitMQ, Apache Qpid, ZeroMQ, Mosquitto and YAMI4 are surveyed and analyzed based on messaging semantics viz. message serialization, message queuing & routing, messaging patterns supported, HA & load balancing, message priority, throughput, latency, memory footprint, etc. Based on the experimental analysis, message throughput and server scalability are better in YAMI4 than in Apache Qpid. Latency in each message is less in YAMI4 as compared to Apache Qpid. YAMI4 also proves to be lightweight than Qpid. Therefore, YAMI4 is found to be suitable for deployment in Control and Monitoring systems.

YAMI4 is rigorously tested for its internal fundamentals of message ordering and message queuing. The results infer that message is priority ordered only in cases of network throttling, and is first-in-first-out in the normal scenario.

Further, the paper plans in extending work to enhance YAMI4 from Web context [24] and Application security perspective so as to make it an all-purpose middleware.

ACKNOWLEDGMENT

This project is undertaken at Larsen & Toubro Ltd, Navi Mumbai. The authors remain grateful to the anonymous reviewers for their valuable comments that helped in improving the paper.

REFERENCES

[1] Jorge E. Luzuriaga, et al. "A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks," Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE, vol. no, no., pp. 931,936, 9-12 Jan. 2015 doi: 1O.1109/CCNC.2015.7158101

[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] Dworak, A., Sobczak, M., Ehm, F., Sliwinski, W., Charrue, P., "Middleware trends and market leaders 2011," International Conference on Accelerator and Large Experimental Physics Control System (ICALEPCS2011) , vol. 111010, no.CERN-ATS-2011-196, pp. FRBHMULT05, 10-14 Oct. 2011

[4] Hoefling, Michael, et al. "Integration of IEEE C37. 118 and publish/subscribe communication," Communications (ICC), 2015 IEEE International Conference on, vol., no., pp. 764,769, 8-12 June 2015 doi:10.1109/ICC.2015.7248414

[5] Park, J., Lee, M.J., " SCondi: A Smart Context Distribution Framework Based on a Messaging Service for the Internet of Things, "Journal of Applied Mathematics, Aug. 2014.

[6] Fersi, G.," Middleware for Internet of Things: a study," Distributed Computing in Sensor Systems (DCOSS), 2015 International Conference on, vol., no., pp. 230,235, 10-12 June 2015doi: 10.1109/DCOSS.2015.43

[7] Pandya, H.B. and Champaneria, T.A.," Internet of Things : Survey and Case Studies," Electrical, Electronics, Signals, Communication and Optimization (EESCO), 2015 International Conference on, vol., no., pp. 1,6, 24-25 Jan. 2015 doi: 10.1109/EESCO.2015.7253713

[8] Know all about SCADA Systems Architecture and Types with Applications, Available on [URL: http://www.edgefxkits.com/blog/scada-system-architecture-types-applications/]

[9] YAMI4 vs ZeroMQ", Available on [URL: http://www.inspirel.com/articles/YAMI4_vs_ZeroMQ.html]

[10] YAMI4, Available on [URL: http://www.inspirel.com/yami4/]

[11] Apache Qpid, Available on [URL: https://qpid.apache.org/]

[12] RabbitMQ, Available on [URL: https://www.rabbitmq.com/]

[13] ØMQ Community, Available on [URL: http://zeromq.org/community]

[14] event-driven architecture (EDA), Available on [URL: http://searchsoa.techtarget.com/definition/event-driven-architecture-EDA]

[15] RabbitMQ vs Apache ActiveMQ vs Apache qpid, Available on [URL: http://bhavin.directi.com/rabbitmq-vs-apache-activemq-vs-apache-qpid/]

[16] A quick message queue benchmark: ActiveMQ, RabbitMQ, HornetQ, QPID, Apollo, Available on [URL: http://www.voidcn.com/blog/hanruikai/article/p-4338440.html]

[17] Stress testing Mosquitto MQTT Broker, Available on [URL: http://rexpie.github.io/2015/08/23/stress-testing-mosquitto.html]

[18] Selecting a Message Queue – AMQP or ZeroMQ, Available on [URL: http://bhavin.directi.com/selecting-a-message-queue-amqp-or-zeromq/]

[19] Dissecting Message Queues, Available on [URL: http://bravenewgeek.com/dissecting-message-queues/]

[20] Message Queue Shootout, Available on [URL: http://mikehadlow.blogspot.in/2011/04/message-queue-shootout.html]

[21] What is authentication, Available on [URL: http://searchsecurity.techtarget.com/definition/authentication]

[22] Cryptography Defination, Available on [URL: http://searchsoftwarequality.techtarget.com/definition/cryptography]

[23] Publish/Subscribe, Available on [URL:https://msdn.microsoft.com/en-us/library/ff649664.aspx]

[24] HTML5 Web Sockets: Internet: A Quantum Leap in Scalability for the Web Available on [URL:https://www.websocket.org/quantum.html]