Institute for Software Integrated Systems

Vanderbilt University

Nashville, Tennessee, 37212

Understanding the Performance Requirements and Challenges for the Industrial Internet of Things (IIOT)

TECHNICAL REPORT

June 2018

Michael A. Walker

Abhishek Dubey

Douglas C. Schmidt

Introduction

The Industrial Internet of Things (IIoT) represents a significant shift away from traditional stove-piped, vertical silos of systems that focus inwardly within an individual application domain to a world where significant horizontal interaction among domains is the norm. This approach represents the culmination of a convergence between distributed system science and machine learning science. Due to scale and architecture, assembling reliable distributed applications from reusable software components is often encouraged in this paradigm. Moreover, the platform layer is designed to ensure separation between information flows of unrelated applications. Moreover, due to the inherently distributed nature of IIoT systems, remote deployment is essential, including support for reconfiguration to recover from anomalous conditions.

Consider the following two use-cases:

1. **Smart Factories.** Well run factories, be they creating high tolerance parts for aeroplanes, using additive manufacturing to produce specialty one-off products, or even creating speciality craft brews, all strive to produce high quality goods as inexpensively, safely, and reliably as possible. To achieve this goal, factories around the globe invest in new production capacity to deliver top-quality products in real-time and to ensure production controls and quality to match demand. The introduction of advanced IIoT technology into manufacturing plants can create smart factories that ensure business opportunities and customer trust are not negatively impacted in case of production spikes or machine failures. This goal is achieved by focusing on the following.

   - Stable supply chain resulting from securing materials based on demand forecasts, optimized inventory management based on better tracking, and improved production utilization.
   - Real-time maintenance response, due to automated early detection of equipment failures and improved order management/inventory of maintenance parts.
   - Reduction in losses due to defective products by quality monitoring and low-latency to degraded quality.

   For example, in a speciality craft beer factory, a supervising brewmaster is typically responsible for monitoring ongoing complex biochemical reactions and managing the quality of the batch. These tasks require the beer factory supervisor to manage inventory, schedule routine inspections, and perform maintenance. Moreover, periodic quality checks are performed to ensure that they can consistently secure raw materials, control inventory, etc.

   Alternatively, a smart factory can apply IIoT technologies to automate the real-time detection of both equipment abnormalities and quality degradation, thereby facilitating timely and accurate responses. Most importantly, these technologies allow the factory to scale to meet increased demand, thereby allowing the factory owner to scale production and expand operations.

2. **Windmill Farms.** Consider another use case related to wind-farms. which are increasing in popularity due to an increased awareness of the potential environmental, social, and political impacts that the use of traditional fossil fuel based power generation creates. This awareness has increased the need/desire for larger wind farms (e.g., with more

windmills and windmills of greater dimensions) and power generation capacity. Unlike traditional power generation stations, windmills requires a wide geographical deployment of a large number of autonomous—yet coordinated—modules, i.e., windmills.

The variations in wind patterns not only creates control challenges(such as determining the speed and power to generate from each area), it also creates various system management challenges (such as predictive health management and prognosticating the remaining time to maintenance). Addressing these challenges motivates constant monitoring of turbines and windmills, which yields a tight interplay between the sensors and actuators that control the windmill.

The remoteness of the typically installations also motivate edge analytics because it is often hard to move all the data back to the cloud in real-time. Windmill farms are therefore a good use case for real-time data collection, online learning, and real-time decision dissemination, all of which requires—and benefits from—edge computing.

The following two tables describe typical data generation and collection requirements for windmills.

Message Use Case Scenarios		
Data Category	**Period**	**Real-Life Example**
Stream Data Logging	Continuous	Continuous log of entire windmill's state for offline-analysis
State Change	Sporadic - Uncommon	Windmill stops spinning, due to too low wind speeds
Critical Event	Sporadic - Rare	Windmill Immediately Stops due to mechanical fault

Message Use Case Comparative Constraints[1]			
Data Category	Expected Messages/ second (per device)	QoS Priority	Round trip Latency Requirements
Stream Data Logging	100	low	125μs - 300μs
State Change	0~10	High	0.1s - 2s
Critical Event	0~1	Highest	0.1s - 1s

Although these use cases motivate both the use of optimal data collection mechanisms and edge computing, we initially review the middleware requirement for data collection.

[1] These estimates are based on discussions with Siemens engineers on this scenario.

Overview of Middleware

Effective middleware abstractions enable developers to build applications that are not large, complex, monolithic software entities. Instead, they are componentized and often built from reusable, configurable and integratable components through composition. The composition defines possible interactions among the components and these interactions are the only interactions allowed. Such restrictions allow better management of application execution, from resource management, fault management, and security management perspectives.

Industry standards and pragmatic experience shows that a well-defined and (relatively) small set of interaction patterns can provide a solid foundation for building applications. This set of patterns include the following:

- **Point-to-point interactions** that occur when an object wants to invoke specific services of another object. These interaction can be synchronous (request/response) or asynchronous (registration/callback), and
- **Publisher-Subscriber interactions** that occur when a publisher generates data samples, which are then asynchronously or synchronously consumed by interested subscribers.

The primary difference between point-to-point interaction and Publisher-Subscriber interaction is the decoupling between producers and consumers of data, which is advantageous if new producers and consumers are added into the system or if the producers and consumers can fail over time. As a result, the Publisher-Subscriber interaction pattern has emerged as the *de facto* standard for IIoT. Below we discuss several middleware platforms for IIoT, focusing initially on middleware that implements the OMG Data Distribution Service (DDS) standard.

Overview of The Data Distribution Service (DDS) Middleware

The OMG DDS specification defines a distributed pub/sub communications architecture. At the core of DDS is a data-centric architecture for connecting anonymous data publishers with data subscribers. The DDS architecture promotes loose coupling between system components. The data publishers and subscribers are decoupled with respect to (1) time (i.e., they need not be present at the same time), (2) space (i.e., they may be located anywhere), (3) flow (i.e., data publishers must offer equivalent or better quality-of-service (QoS) than required by data subscribers), and behavior (i.e., business logic-independent), (4) platforms, and (5) programming languages (e.g., DDS applications can be written in many programming languages, including C, C++, Java, Scala, etc.).

A DDS data publisher produces typed data-flows identified by names called topics. The coupling between a publisher and subscriber is expressed only in terms of topic name, its data type schema, and the offered and requested QoS attributes of publishers and subscribers, respectively. Below we outline the key architectural elements of the DDS specification:

- **Domain** is a logical communication environment used to isolate and optimize network communications within the group of distributed applications that share common interests (i.e., topics and QoS). DDS applications can send and receive data among themselves only if they have the same domain ID.

- **Participant** is an entity that represents either a publisher or subscriber role of a DDS application in a domain, and behaves as a container for other DDS entities (i.e., DataWriters and DataReaders).
- **DataWriter and DataReader**. DataWriters (data publishers) and DataReaders (data subscribers) are endpoint entities used to write and read typed data messages from a global data space, respectively.
- **Topic** is a logical channel between DataWriters and DataReaders that specifies the data type of publication and subscription. The topic names, types, and QoS of DataWriters and DataReaders must match to establish communications between them.
- **Content Filtered Topic (CFT)**. Content filters refine a topic subscription and help to eliminate samples that do not match the defined application-specified predicates. The predicate is a string encoded SQL-like expression based on the fields of the data type. The filter expression and the parameters may change at run-time. Data filtering can be performed by the DataWriter or DataReader.
- **Quality-of-service (QoS)**. DDS supports around two dozen QoS policies that can be combined in different ways. Most QoS policies have requested/offered semantics, which are used to configure the data flow between each pair of DataReader and DataWriter, and dictate the resource usage of the involved Entities.

DDS implementations ensures that the endpoints are compatible with respect to topic name, their data type, and QoS configurations. Creating a DataReader with a known topic and data type implicitly creates a subscription, which may or may not match with a DataWriter depending upon the QoS.

The DDS specification provides a rich vocabulary for specifying QoS properties, though not all policies described in the DDS specification are needed in an IIoT setting. Here we summarize a few QoS that are critical for common IIoT systems:

- **Reliability QoS** – The Reliability QoS provides a mechanism to indicate that the middleware should ensure that messages are delivered from Publishers to associated Subscribers reliably. This QoS policy is set and enforced entirely in the user space middleware. The Publisher will cache samples up to the specified Lifespan QoS for that Publisher. Subscribers to Reliable Publishers must also have the Reliable QoS enabled so that they will transmit acknowledgements of sample receipt as they arrive.
- **Lifespan QoS** – The Lifespan QoS Policy causes all samples to be time stamped. Publishers use the value of this policy to remove stale samples from their send queue. This is especially useful in conjunction with the Reliability QoS to provide a timeout for messages that are being retried. Subscribers use the value of this policy to expire samples from their queue that have not been collected or are being buffered as a result of History QoS. This QoS Policy is configured and enforced by the user space middleware.
- **Time-based Filtering** – The TimeBasedFilter QoS Policy provides a mechanism to control the rate at which Subscribers receive samples. This is useful in the event that any Publishers produce data at a faster rate than required by the Subscriber. This QoS Policy is configured on a Subscriber and enforced in the User Space middleware
- **Content Filtered Topic** – The ContentFilteredTopic QoS Policy provides a mechanism to restrict the types of updates that are provided to a consumer based on the content of the topic. The value for this policy is an expression that uses a subset of SQL92 to specify a range of values that the subscriber is interested in. Samples that fall outside

that range are automatically dropped by the middleware without the application being notified.

- **Latency Budget** – The LatencyBudget QoS Policy provides an indication of the maximum amount of time allowed to deliver samples. This QoS Policy is used to provide scheduling hints to the user space middleware and the networking layer (that supports this QoS). Lower values will result in higher scheduling priority when there are several samples from different Publishers to be sent at once. This QoS Policy is configured on a Publisher and enforced by the User Space middleware.
- **Rate Limit**- The RateLimit QoS Policy provides a mechanism through which a maximum rate at wish a Publisher may publish samples to the network may be established. This is used to protect against a malicious actor flooding the network with excess samples and impacting the QoS of other Actors.

Overview of Other Middleware Platforms for IIoT

Other middleware platforms beyond DDS support the Publisher-Subscriber pattern, as described below.

Message Queuing Telemetry Transport (MQTT)

MQTT is an ISO standard connectivity protocol developed to support machine-to-machine (M2M) communications in IIoT. MQTT supports a Publisher-Subscriber communication model that uses the term “client” to refer to entities that either publish topics or subscribe to topics, while the term “server” refers to mediators/brokers that relay messages between the clients. The original target of MQTT was to support IIoT resource-constrained devices, so it is designed as a lightweight protocol.

Example use cases of MQTT include sensors communicating to a broker via a satellite link, dial-up connections with health care providers, and a range of home automation and small device scenarios. Even mobile applications can use MQTT due to its support for small size, low power usage, smaller data packet payloads, and efficient distribution of information to one or many receivers. MQTT operates over TCP or any other transport protocol that supports ordered, lossless message communication. MQTT supports three levels of quality-of-service (QoS) for message deliver: (a) at-most-once, (b) at-least-once, and (c) exactly-once. MQTT was originally developed in 1999 and has recently become an ISO standard starting from version 3.1.1.

Kafka

A number of message brokers, such as Apache Kafka, the Advanced Message Queue Protocol (AMQP), and Active MQ, are finding applications in areas of IIoT. In particular, Apache Kafka is an open-source distributed streaming platform used to build real-time data pipelines between different systems or applications. Kafka provides high throughput, low latency and fault tolerant pipelines for streaming data with a tradeoff between performance and reliability.

Kafka can be deployed as a cluster of servers than thandles the messaging system with the help of four core APIs: producers, consumers, streams, and connectors. Other key portions of the Kafka architecture include the topic, broker, and records. Kafka divides data into topics, which are further divided into partitions for brokers to handle them. Apache Zookeeper is used to provide synchronization between multiple brokers running Kafka.

The Robot Operating System (ROS)

ROS is a framework that provides a collection of tools, libraries, and conventions to write robust, general-purpose robotics software. It is designed to work with various robotic platforms, including industrial applications (see https://rosindustrial.org/). ROS nodes are processes that perform computation. These nodes can be combined together form a network (graph) of nodes that communicate with each other using Publisher-Subscriber or Request-Response interaction patterns.

Publisher-Subscriber interaction in ROS is facilitated via topics. Multiple publishers and subscribers can be associated with a topic. Request-Response interaction, in contrast, is performed via a service. A node that provides a service offers its service using a string name, and a client calls a provided service by sending the request message and awaiting the reply.

Topics and services are both monitored by the ROS Master. This master performs the task of matching nodes that need to communicate with each other, regardless of the interaction pattern. The ROS master is also a single point of failure. Recently, the next generation of ROS middleware has integrated its messaging layer with OMG DDS.

NATS

NATS is an open-source messaging system developed by Synadia Communications, Inc. and is now a Cloud Native Computing Foundation member (CNCF) project. The system is made up of several components: the NATS Server, NATS Streaming, Client Libraries, and a connector framework. The primary use case for NATS is the webserver model. As such it is designed around the idea of having an always online server, and services, that clients can connect to via a text based protocol. The NATS Streaming server connects to, and communicates with, subscriber clients through the NATS Server, which provides simple Pub-Sub mechanism, providing support for Authentication and Clustering. The server also supports single and multi-user/client authentication. Clustering allows multiple Servers to communicate, thereby it lets messages published to one server be routed and received by a subscriber on another server.

NATS Streaming provides the features beyond just simple pub/sub that NATS Server provides. This includes support for subscribing to a Channel based subscriptions by clients. A client creates a subscription on a given channel. This subscription can be of one of three types: Regular, Durable, and Queue Group. Regular Subscriptions have their state removed from the server when they unsubscribe or are closed. Durable Subscriptions, after creation by a client, maintain the state of the last delivered message on the server, allowing for reconnecting after the connection is closed and resuming at the previous message location. Queue Group Subscriptions allows for multiple clients to consume from the same Channel, without receiving the same information. The exact delivery order when multiple clients are connected to a Queue Group is not defined. A Queue Group Channel Subscription is by default Regular, but can also be Durable.

Middleware Requirements

To compare different middleware solutions fairly, standard benchmarking metrics must be used. Key middleware metrics include the following:

- **Latency**, which is the delay between the beginning of transmission of some data and the full receipt of that data at the intended target. However, there is two types of latency characteristics that we must consider for different distributed systems middleware. The first is the above delay between transmission of the data and full receipt of said data. The second is the delay between beginning of the data transmission and the receipt at the sender of any acknowledgement return packets, if expected.

- **Packet Loss**, which is the number of packets of information that are lost during transmission. This impacts either the number of retransmissions required or the amount of data loss at the recipient.
- **Throughput**, which is the maximum amount of data transmission along a given path.
- **Multiple application criticalities**, which is the expectation that the middleware can support different criticalities of application flows and can prioritize one over the other if the current system cannot support all the flows at once.

These metrics can be used to determine the base networking efficiency and capacity of a middleware solution, given a specific network topology and data transmission requirements All of these metrics depend upon both standardizing the data packet being sen, and the network upon which the test is being run. Likewise, the type and amount of data being transmitted on the network being tested will also impact the characteristics of the data the middleware is transmitting.

Several of these requirements also often conflict with each other. For example, reducing packet loss may require retransmission, which effects available bandwidth and can lower average latency. Most modern middleware implementations therefore provide configurable quality of service mechanisms that can be used to fine tune a deployment in a network.

Quality-of-Service (QoS) Configurations

Depending on the middleware solution used, there are potential QoS configuration settings that can be leveraged to better configure the middleware to fit the needs of a specific use-case. These configuration settings fall into the following two broad categories:

- **Middleware QoS**, which are settings internal to the middleware itself that affect how data is processed and/or routed. Examples of middleware QoS include the following:
  - *A middleware specific routing policy identifier that gives priority to one data type over another once the data has reached the middleware for processing, e.g., NATS Queue Group load balancing.*
  - *Guaranteed delivery* of data configuration vs Best-Effort delivery
- **Hardware/Network QoS**, which are settings that extend beyond the middleware layer and affect how the host operating system or networking equipment. An example of hardware/network QoS includes differentiated services (Diffserv), which is a networking architecture that provides fast and scalable mechanism for classifying and managing network traffic. Allowing low-latency prioritization of critical data, such as voice or streaming media, and best-effort prioritization of non-critical services such as web traffic or file transfers.

Most middleware solutions for distributed systems provide at least some simple Middleware QoS functionality. Not all middleware solutions, however, provide access to Hardware/Network QoS functionalities. The main differentiator is if the middleware exposes to the user lower-level primitives to configure these settings or not.

For instance, DDS exposes to the user primitives for Diffserv, whereas NATS does not. The design goals of different middleware solutions explains this discrepancy. NATS for instance, has the design goal of being as simple for the user to use as possible, using a plain text based protocol. This is because the intended use-case of NATS is for Datacenter hosted web servers to communicate between each other in a scalable and reliable manner as easily as possible for the developer. Whereas DDS, is designed for much more specific configuration by the user to match their exact use case.

Preliminary Middleware Benchmarking Test Results

When analyzing the impact that different QoS or other parameters have upon the benchmarking statistics of a system, tests must be consistent except for one variable. The following figures compare the exact same settings tested on two different networks, Figure 1 being a cloud service scenario and Figure 2 being a wireless IIoT scenario.

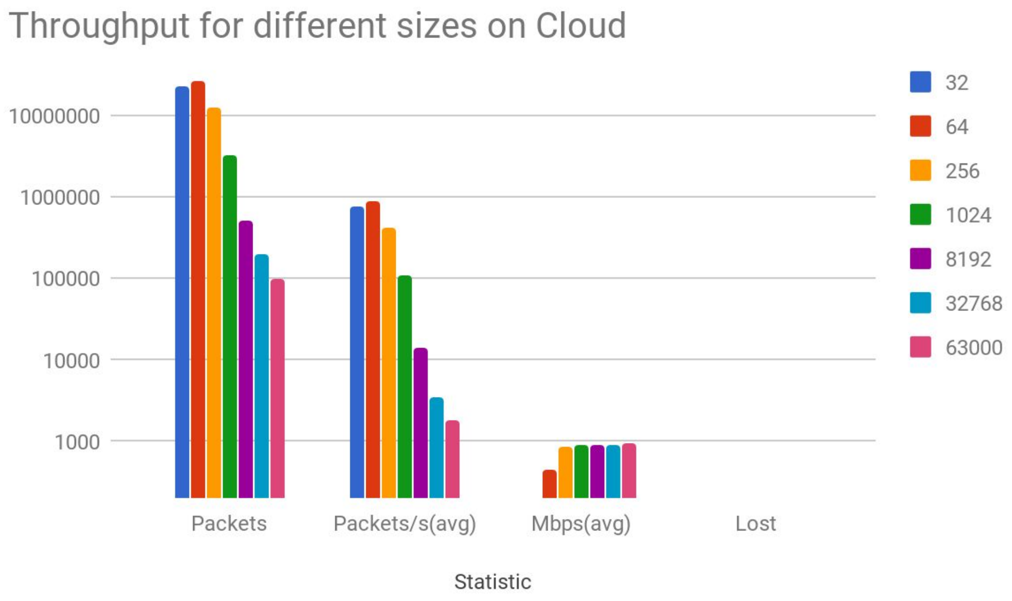

Figure 1: RTI DDS Performance Results Between 2x 100mbps servers

Both tests shown in Figures 1 and 2 compare data packets of different sizes and graph their impacts on number of data packets (Separate from TCP/IP Packets) sent, number of packets per second, Mbps throughput, and number of packets lost when tested on the cloud and when tested on a network of PIs.

These two tests were run with guaranteed delivery QoS settings, but didn't vary much when using best-effort QoS, with the reliable TCP/IP networks that the tests were ran on. On UDP broadcast networks (or more unreliable wireless networks), however, the variation on Delivery QoS settings would have increased number of data packets being re-sent with guaranteed delivery QoS.

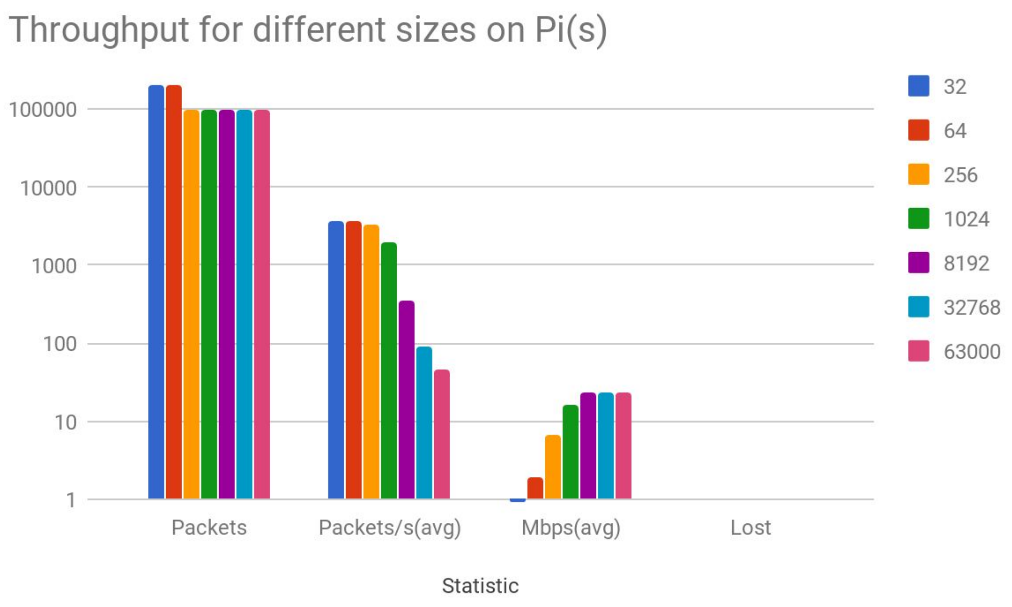

Figure 2: RTI DDS Performance Results between 2x WiFi connected Raspberry Pi 3Bs