Pacific Northwest
NATIONAL LABORATORY

PNNL-30858

Message Bus Usage Recommendations

December 2020

Shwetha Niddodi
Benjamin LaRoque
Craig H Allwardt
Chandrika Sivaramakrishnan
Jereme N Haack

U.S. DEPARTMENT OF ENERGY

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

DISCLAIMER

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information,
P.O. Box 62, Oak Ridge, TN 37831-0062;
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service
5301 Shawnee Rd., Alexandria, VA 22312
ph: (800) 553-NTIS (6847)
email: orders@ntis.gov <https://www.ntis.gov/about>
Online ordering: http://www.ntis.gov

Acronyms and Abbreviations

CA	Certificate Authority
ECC	Elliptical Cryptographic Curve
GE	General Electric
HVAC	heating, ventilation, and air-conditioning
ML	Machine Learning
MQTT	Message Queuing Telemetry Transport
PNNL	Pacific Northwest National Laboratory
RMQ	RabbitMQ
RPC	Remote Procedure Call
VIP	VOLTTRON Interconnect Protocol
ZAP	ZeroMQ Authentication Protocol
ZMQ	ZeroMQ
JSON	JavaScript Object Notation

Contents

Figures

Tables

1.0 Introduction

The VOLTTRON platform developed by Pacific Northwest National Laboratory (PNNL) and funded by the U.S. Department of Energy's Building Technologies Office incorporates two different message bus technologies, each with its own strengths and use cases. The message bus is a key platform component responsible for moving data from one endpoint to another. It is also essential for meeting the security and interoperability goals of the platform with implications on ease of deployment, scalability, and integration. This document describes ZeroMQ (ZMQ) and RabbitMQ (RMQ) as used in VOLTTRON and discusses the most appropriate message bus choice for a specific VOLTTRON deployment use case.

VOLTTRON was initially developed with ZMQ as the message bus. ZMQ's messaging library is lightweight and extensible allowing for development of scalable distributed applications. However, many custom-built features had to be added on top of the core ZMQ library to meet platform requirements. As VOLTTRON became more mature and the number of use cases increased, the VOLTTRON team continuously attempted to maintain and extend features to these custom-built features to keep up with community needs. Based on community feedback and to reduce maintenance cost, it was decided to refactor VOLTTRON's message bus layer, to use a more industry-accepted messaging library that provides many of the needed features. RMQ is one such messaging library that has seen major investments by commercial companies. Rabbit Technologies, now part of Pivotal Technologies (VMWARE spin-off) saw a $105 million investment by General Electric in 2013. It is used by Instagram, Indeed.com, Google Cloud Platform, Tesla, etc. Huge industry backing and the ability to benefit from community-driven message bus improvements, led PNNL to include RMQ as the next message bus for VOLTTRON. The goals of this message bus extension are as follows:

- Maintain essential features of current message bus and minimize transition cost.
- Leverage an existing and growing community dedicated to the further development of RMQ.
- Move services provided currently by VOLTTRON agents to services natively provided by RMQ.
- Decrease VOLTTRON development time spent on supporting the message bus, which is now a commodity technology.
- Address community concerns about ZMQ.

To allow community members to select the best option for their needs, this document provides recommendations about which message bus is better suited for various use cases and deployment scenarios. A summary of recommendations can be found in section 2.0. Sections 3.0 and 4.0 provide a short description of ZMQ-based VOLTTRON and RMQ-based VOLTTRON, and Section 5.0 describes the differences between the two. Section 0 describes performance benchmarking performed on the two message bus technologies on Raspberry Pi and the results collected from various test scenarios. Section 7.0 describes various deployment use cases and when to use each message bus. Section 8.0 describes how RMQ-based VOLTTRON can be used to integrate with various third-party tools to make heterogeneous systems work cohesively.

2.0 Deployment Recommendations Summary

This section provides a short summary of the analysis and consideration of common use cases with recommendations for when to use each message bus. More in-depth analysis and discussion follows in the rest of the document.

ZMQ-based VOLTTRON is easy to install because it involves very few installation steps. Non-software engineers can quickly bootstrap the environment with minimal steps and start running the platform. It is lightweight, has well-defined security features, and a low memory footprint. These features make it easy to deploy in low-cost devices that have memory constraints, such as raspberry Pis. ZMQ-based VOLTTRON performs well on small boards as in Section 0 for the performance benchmarking comparison conducted on a Raspberry Pi 4 model B. ZMQ-based VOLTTRON is perfect for single platform deployment or multi-platform deployments with fewer VOLTTRON instances (<20) connecting to a central instance or to each other. ZMQ-based VOLTTRON can hold ~100 agents per VOLTTRON instance without any degradation in the message bus performance. ZMQ-based VOLTTRON can handle a low to medium volume of traffic, as shown in the benchmark results in Sections 6.2 and 0. In ZMQ-based VOLTTRON, we have custom-built a ForwardHistorian agent to forward messages from one platform to another. ForwardHistorian also provides backup cache support, which is useful when connection to the remote platform is lost. ZMQ-based VOLTTRON has a custom-built multi-platform feature where the connections between multiple platforms are maintained by the internal router module, and the agents themselves do not have to manage the connection. They can publish/subscribe to messages and make Remote Procedure Calls (RPCs) to agents in other platforms seamlessly. However, this does not scale as well as an RMQ-based VOLTTRON instance because it needs $O(n^2)$ connections between n instances. The platforms cannot be daisy chained together and have messages be sent over multiple hops to a destination platform or have multiple groups of VOLTTRON instances connect to each other. For ZMQ-based VOLTTRON to provide this kind of flexible deployment options, these features need to be custom-built, which will involve lot of time and effort from the VOLTTRON team.

Installation of RMQ-based VOLTTRON is more involved because it has more steps related to configuring the RMQ broker and setting up SSL certificates for the VOLTTRON platform and its agents. This added complexity is perhaps a higher barrier to enter for some non-software engineers and would make them hesitant to adopt RMQ-based VOLTTRON for their deployment use case. The VOLTTRON team continuously works on streamlining the installation and troubleshooting steps based on user feedback. RMQ is useful for large-scale deployments involving numerous VOLTTRON instances either connected to each other or to a central instance. It has several easy-to-use plugins that are integrated into RMQ-based VOLTTRON. It also provides more flexibility in deployment. The shovel plugin can be used to forward messages from one platform to another. But the drawback is that, it has limited caching capability so when the connection to the remote broker is lost for an extensive period, data will be lost. RMQ provides a standard in-built federation plugin to connect multiple platforms together. This plugin can be used to connect multiple VOLTTRON instances and have them work together as a group with loose coupling. The agents themselves do not have to manage the connection; they can publish/subscribe to messages and make RPCs to agents in other platforms seamlessly. The platforms can be daisy chained together or have multiple groups of VOLTTRON instances connect to each other using the federation plugin. Unlike the ZMQ-based VOLTTRON instance, the federation plugin does not need $O(n^2)$ connection between n instances, so it scales better. In terms of message bus performance, RMQ-based VOLTTRON performs better as the number of messages being published on the message bus increases to

very high extent and message payload size becomes very high. So, it is well suited for deployment use cases that need to withstand very large amounts of data traffic. RMQ provides a web interface to manage and monitor RMQ resources. This interface enables creation, deletion, and authorization management of users, queues, and more. It also allows the platform user to monitor performance metrics such as queue length, message rates, connection information, etc. This is a useful feature for understanding the status of a deployed instance and performing some quick troubleshooting. ZMQ-based VOLTTRON has custom-built features to control the status of the agents and add/delete/control access of some of VOLTTRON's resources. However, it does not have built-in capability to monitor the status and gather message bus performance metrics.

Integration with third-party tools, such as Message Queuing Telemetry Transport (MQTT) and Elasticsearch, is easier with RMQ-based VOLTTRON. Because RMQ is a well-known messaging library and widely accepted in industry, several easy-to-use plugins have been developed either within RMQ or in the external tools to establish connection with each other. A git repository (https://github.com/VOLTTRON/external-clients-for-rabbitmq) maintained by the VOLTTRON team shows examples of how to connect to some of them. In ZMQ-based VOLTTRON, custom agents must be created to connect to these disparate tools and send messages to each other. These agents also must be regularly updated to keep up with changes in newer versions of the external tool. Having the choice of various community-developed plugins instead of custom agents, makes it easier for VOLTTRON to cater to the emerging needs of the community.

3.0 ZeroMQ-based VOLTTRON

This section provides brief description of features of ZMQ-based VOLTTRON. ZMQ-based VOLTTRON uses ZMQ (https://zeromq.org/get-started/) as the underlying message library. The VOLTTRON platform process acts as the server and is responsible for accepting incoming client connections and routing the messages between agents. The VOLTTRON agents are client applications and connect to the VOLTTRON platform to be part of the VOLTTRON ecosystem. The agents send messages to each other using the VOLTTRON Interconnect Protocol (VIP).

3.1 VOLTTRON Interconnect Protocol

VIP is a routing protocol invented by PNNL that allows agents to send messages to each other in a known, common message format, while maintaining interoperability and security goals. VIP uses the ZMQ's router pattern. Specifically, the router runs within the VOLTTRON platform and binds to a ROUTER socket and acts as the server, and peers/agents connect using a DEALER or ROUTER socket. The router is responsible for routing messages between peers/agents. Each agent must be associated with unique identity string so that the router knows where to route the message to. Each message follows the format below.

RECEIVER	SENDER	PROTOCOL	USER_ID	MSG_ID	SUBSYSTEM	ARG_1	ARG_2		ARG_N

- Sender: identity of the sending (source) peer.
- Receiver: identity of the recipient (destination) peer.
- Protocol: set to "VIP1".

- User_ID: VIP authentication metadata set in the authenticator.
- Msg_ID: message identifier set by the sending peer. Replies SHALL echo the request id without modifying it.
- Subsystem: this specifies the peer subsystem for which the data are intended.
- Arguments: provides the arguments for the given subsystem. The number of frames required is defined by each subsystem.

When an agent, Agent1, wants to make an RPC “set_point” to Agent2, it packages the message in the format below and sends it over the message bus. The router looks at the first frame and forwards the message to the intended recipient. The recipient, Agent2, then performs the appropriate actions based on the subsystem frame, which is “RPC” in this example, and sends the response back by exchanging the first and second frames. The router again looks at the first frame and routes it to caller of RPC method which is Agent1.

Agent2	Agent1	VIP1	abcd	001	RPC	set_point	Agent2	PointA	0.9

Agent1	Agent2	VIP1	abcd	001	RPC	set_point	True

More information about the VIP protocol can be found at
https://volttron.readthedocs.io/en/develop/platform-features/message-bus/vip/vip-overview.html

3.2 Authentication

VIP uses ZMQ’s ZeroMQ Authentication Protocol (ZAP) and Elliptical Cryptographic Curve (ECC) key mechanism to provide authentication. Only agents authenticated by the platform can connect and use the encrypted channel for communication. VIP authentication is implemented in the auth module and extends the ZAP to VIP by including the ZAP User-ID in the VIP payload, thereby allowing the platform to authorize access based on ZAP credentials. VOLTTRON automatically generates an encryption key and enables CurveMQ by default on all Transmission Control Protocol connections.

ZAP defines a method for verifying credentials exchanged when a connection is initially established. The authentication mechanism provides three main pieces of information useful for authentication:

- domain: a name assigned to a locally bound address (to which peers connect)
- address: the remote address of the peer
- credentials: includes the authentication method and any associated credentials.

During authentication, VOLTTRON checks these pieces against a list of accepted peers defined in a file, referred to as the “auth file” in this document. This JSON-formatted file is located at $VOLTTRON_HOME/auth.json and contains an “allow” list defining the list of allowed credentials. Authentication goes through when credentials match. Domain and address details are supplemental information and are not mandatory for authentication. Each agent must create an ECC-based private-public key pair, and the agent’s public credentials must be added to the auth file before attempting to connect to the platform.

3.3 Authorization

VIP authorization gives the platform owner the ability to limit the capabilities of authenticated agents. For example, the platform owner can set “capabilities” to allow only certain agents such as the PlatformDriver agent to publish to the “devices” topic. If any other agent attempts to publish to “devices” topic, the platform will raise an “Unauthorized” error. Another example would be adding capability to an agent’s RPC method. For example, setting “CAN_SET_TEMP” capability to agent’s set_point() RPC method restricts its access to agents that have the “CAN_SET_TEMP” capability defined in their auth file entry. More details about VIP authorization can be found at https://volttron.readthedocs.io/en/develop/platform-features/message-bus/vip/vip-authorization.html,

4.0 RabbitMQ-based VOLTTRON

RMQ-based VOLTTRON uses the pika (https://pika.readthedocs.io/en/stable/) library for the RMQ message bus implementation. The RMQ broker runs outside the VOLTTRON platform and all the agents, including the platform, connect to the broker. RMQ exchange is responsible for routing messages between agents. RMQ-based VOLTTRON uses SSL-based authentication mechanism with x509 certificates. Each VOLTTRON instance needs to be set up to connect to an RMQ broker, create exchange, create users for agents, generate certificates for connecting to the broker, etc. After each agent connects to the broker, it creates a VIP queue and binds the queue to the exchange with a unique binding key <instance-name>.<identity> and uses the queue to send and receive messages from the exchange. The binding key is used for routing and because each binding key is unique, the exchange will know where to forward the message to.

4.1 VIP in RabbitMQ VOLTTRON

To maintain backward compatibility with ZMQ-based VOLTTRON, one of the main goals of the refactoring efforts was to decouple the VOLTTRON-specific code from the message bus implementation without compromising the existing features of the platform. The next step was to encapsulate all messages sent from the application code into a message bus agnostic VIP message object. The message parameters continue to follow the VIP protocol frames such as sender, receiver, protocol, subsystem, etc., but they are mapped to pika properties before publishing the message. The message is published on the RMQ message bus using pika library APIs, as follows:

```
# Fit VIP frames in the PIKA properties dict
# VIP format - [SENDER, RECIPIENT, PROTO, USER_ID, MSG_ID, SUBSYS,
ARGS...]
message_property = {
'user_id': userid, # USER_ID
'app_id': <Routing key of SENDER>
'headers': dict(
        recipient= <Routing Key of destination>, # RECEIVER
        proto='VIP', # PROTO
        user=user, # USER_ID
),
'message_id': msg_id, # MSG_ID
'type': <subsystem>, # SUBSYS
```

```
'content_type': 'application/json'
}
```

The exchange looks at the destination routing key (binding key) and routes it to an appropriate destination agent. On the receiver end, the pika message properties are examined and appropriate action is taken. For example, if the subsystem property is “RPC”, it is processed by the corresponding subsystem component of the agent code and sent higher up to the application code.

4.2 Authentication

RMQ-based VOLTTRON uses SSL-based authentication (Figure 1), rather than the default username and password authentication. VOLTTRON adds SSL-based configuration entries to the rabbitmq.conf file for RMQ broker to use during the setup process.

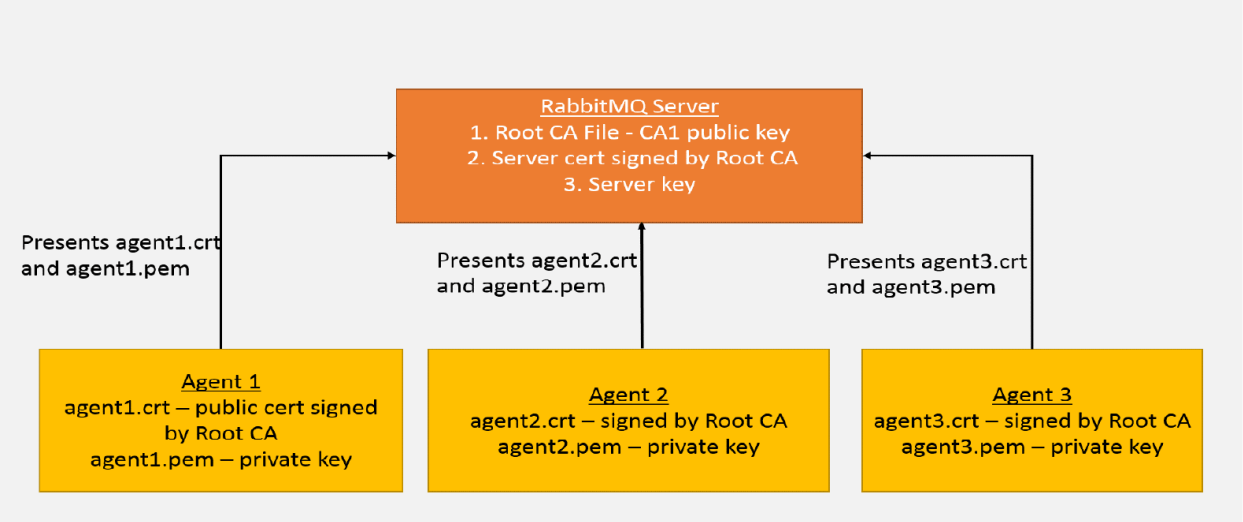

Figure 1. SSL-based authentication.

Every RMQ-based VOLTTRON instance has a single self-signed root Certificate Authority (CA) and server certificate signed by the root CA. This is created during VOLTTRON setup and the RMQ server is configured and started with these two certificates. Every time an agent is started, the platform automatically creates a pair of public-private keys for that agent and a certificate that is signed by the same root CA. When an agent communicates with the RMQ message bus it presents its public certificate and private key to the server and the server validates whether it is signed by a root CA it trusts; i.e., the root certificate it was started with. Because there is only a single root CA for one VOLTTRON instance, all the agents in this instance can communicate with the message bus over SSL.

4.3 Authorization

This feature needs to be implemented in RMQ-based VOLTTRON. RMQ has several access controls and permission settings that can be applied on the RMQ queues, exchanges, and users to control access to them. This feature needs to be leveraged within RMQ-based VOLTTRON and applied to resources used by VOLTTRON.

5.0 Differences between ZMQ-based VOLTTRON and RMQ-based VOLTTRON

ZMQ-based VOLTTRON	RMQ-based VOLTTRON
Platform acts as the broker and is responsible for routing messages	Separate broker runs outside the platform and all agents connect to the broker. Exchanges are responsible for routing messages.
Authentication is based on the ZAP protocol using the ECC keys.	SSL-based authentication uses TLS x509 certificates.
Remote agent authentication is achieved by adding the public key of the remote agent to auth.json	Remote agent authentication uses Certificate-Signing Request operation.
Custom agents such as ForwardHistorian agent for forwarding messages from one platform to another.	Shovel plugin can forward messages from one platform to another.
VOLTTRON specific implementation for multi-platform connection	Federation plugin can be used for multi-platform connection.
Custom agents to connect to third-party tools. Example: MQTT historian agent.	Tool integration using RMQ plugins (example: https://www.rabbitmq.com/mqtt.html) or third-party plugins (example: https://www.elastic.co/guide/en/logstash/current/plugins-integrations-rabbitmq.html)
Need to build custom agent to monitor the status of the message bus	Management plugin that provides web user interface to monitor status of message bus • Message rates • Resource usage of queues • Data rates of client connections
Scalable multi-platform connections	Highly scalable. It does not require $O(n^2)$ connections between n brokers.

Table 1 Differences between ZMQ-based and RMQ-based VOLTTRON

6.0 Performance Benchmarking

This section describes a comparison of ZMQ-based VOLTTRON and RMQ-based VOLTTRON performance in a small system. For the purposes of comparison, several measurements were conducted using VOLTTRON installed on a Raspberry Pi 4 model B. The Pi was running Raspbian 10 and all tests were local to the system (that is, no physical devices, external databases, or other services were used). The tests explored several different parameters as described in sections below. The test configuration used a single platform driver with varying numbers of instances of a fake device with 18 points. The driver was configured with zero offset between devices and a default scrape interval of 1 minute. A single custom listener agent is available in the scripts/scalability-testing/multilistener directory of the main VOLTTRON repository. This agent was used to measure the time interval between two times, the timestamp in the message header for the first message in a scrape conducted by the platform driver, and the time at which the final message from the scrape was received by the listener. The time interval values reported here are this interval averaged over five sequential

scrapes. For configurations with multiple listener agents, the time interval is also averaged over the measurement made for each listener

6.1 Scaling the Number of Devices

The first set of benchmarks was collected by running the default configuration described above with the device scrape interval increased to 2 minutes and an increasing number of identical devices installed. This is summarized in **Error! Reference source not found.**, where the time interval is as described above, and the normalized column is that value divided by the number of devices.

	ZMQ		RMQ	
Number of Devices	Time Interval (seconds)	Normalized (seconds)	Time Interval (seconds)	Normalized (seconds)
2000	6.8425	0.0034	7.1926	0.0036
4000	15.7469	0.0039	15.3346	0.0038
6000	27.3608	0.0046	23.1164	0.0039
8000	38.9499	0.0049	30.1291	0.0038
10000	52.9024	0.0053	39.2204	0.0039
15000	92.7736	0.0062	59.5759	0.0040

Table 2 Scaling number of devices

From these results we see that for the lower numbers of devices, the time to complete a scrape scales with the number of devices on either message bus. On ZMQ, at a sufficiently large number of devices, the time to publish begins to increase more quickly with increasing numbers of devices. The measurements were not continued to larger numbers of devices because a publication time larger than the device scrape interval would not produce a well defined measurement with the code as implemented.

The behavior using RMQ did not scale worse than linearly with the number of devices for the cases considered. This implies that as the number of devices and hence the corresponding publications increase to a very high value, RMQ fairs better than ZMQ. RMQ has built-in load balancing capabilities, which helps to balance the heavy traffic on the message bus and hence the performance of RMQ is better with a larger number of publications. RMQ also has several back pressure capabilities to regulate the high amount of traffic from producers as explained in https://www.rabbitmq.com/blog/2015/10/06/new-credit-flow-settings-on-rabbitmq-3-5-5/.

6.2 Python Version Comparison

In addition to the measurements of the previous section, a comparison was made between performance when running using python 2 and python 3. Because there are subtleties when installing erlang dependencies on arm-based system, these tests were conducted using a Debian 10 (Buster) running on relatively small virtual machines on a MacBook Pro. Each virtual machine was allocated 1 CPU and 1024 MB of RAM. The full python versions used were 2.7.16 and 3.7.3, and for the python 2 tests the releases/6.x branch of VOLTTRON was used because VOLTTRON versions 7 and above are incompatible with python 2. The measurements were otherwise done in the same way as in Section 6.1.

The results of these comparisons are summarized in Table 3. There it is clear that for any combination of number of devices and message bus considered, the performance of VOLTTRON version 7 running on python 3 is faster than using VOLTTRON version 6 with python 2. In addition to the performance improvements, at this time python 2 has reached end of life and is no longer being supported by the python community. Python 3 and with VOLTTRON 7 or newer is therefore recommended for all new systems.

		ZMQ		RMQ	
	Number of Devices	Time Interval (seconds)	Normalized (seconds)	Time Interval (seconds)	Normalized (seconds)
Python 2	4000	22.6471	0.0057	11.9526	0.0030
	8000	49.2673	0.0062	41.5333	0.0052
Python 3	4000	11.4655	0.0029	4.4405	0.0019
	8000	32.0258	0.0040	14.9644	0.0019

Table 3 Python Version Comparison

6.3 Scaling the Number of Agents

The third set of benchmarks fixed the number of devices at 25 and scaled the number of listening agents. Here the results are more similar between the two buses, with both showing a publication time that exceeds the scrape interval at around 40 listeners installed. The results are summarized in **Error! Reference source not found.**.

Number of Listeners	ZMQ (time interval)	RMQ (time interval)
1	0.1385	0.0967
5	0.2649	0.2912
10	0.3543	0.4311
20	0.6048	0.6192
40	0.9850	0.9401
60	1.3602	1.3350
100	2.1723	2.1973

Table 4 Scaling number of agents

6.4 Large Number of Points per Device

To compare the performance of both message buses that had a large payload size, a single measurement was made on each bus for the case of a single listener with 400 devices installed and 4998 points per device. In this case, the average publish interval over five runs on ZMQ was 579.8724 seconds (1.4497 seconds/device), while on RMQ it was found to be 38.9306 seconds (0.0973 seconds/device). The RMQ bus completed these publications in less time than scrape interval, indicating that the configuration could be sustained, whereas with ZMQ the publication time results in an accumulation of backup and eventual failure or data loss. This implies that for a large message payload size, RMQ fairs much better than ZMQ.

6.5 Qualitative Historian Backlog Test

The final set of observations went back to looking at performance as a function of the number of devices. In this case, the custom listener agent was replaced by the SQLite historian and the driver was configured to perform scrapes continuously (as opposed to stopping after a five-scrape measurement). The platform was monitored to see if the historian was able to keep up with the rate of messages being published, or if it started to build up a backlog. On both message buses, no backlogging was observed until the publication rate had exceeded the default limit of the historians. The default behavior publishes points received in batches and limits the amount of time spent inserting data into the database to also allow for receiving new data, and this limit was reached when the number of devices was increased from 1600 to 3200. The performance of the platforms was not observably different between the two message bus cases in this test.

It is important to note that this test was done using default configurations only. It is certainly plausible that a particular use case could finetune or use alternate configurations that would allow significantly larger numbers of points to be consumed. Some examples would be modifying the batch size used by the historian, having multiple historian agents each subscribed to a subset of points, or using a historian for a different SQL flavor using different storage. These configurations should be made specific to the particular use case.

7.0 Deployment Use Cases

A typical deployment scenario of VOLTTRON is installing a VOLTTRON instance in one or many buildings within a campus to collect building device data from individual buildings and send them to a central instance for monitoring, control, or data visualization purposes (Figure 2). The central instance can be running in central server or in the cloud. The PlatformDriver (formerly MasterDriver) agent is configured to collect data from different types of devices in the buildings that typically use BACnet or Modbus protocols such as heating, ventilation, and air-conditioning (HVAC), heat pumps, water heaters, air-handling units, etc. The PlatformDriver agent is configured to perform regular scrapes of device data points such as zone temperature, power, etc., at a pre-configured polling frequency. The scrapes are then published on the message bus on the “devices” topic to be picked up by interested agents. A ForwardHistorian agent is responsible for forwarding data from a local instance to a remote instance. If a VOLTTRON instance in each individual building needs to send messages (for example, device messages) to the central instance, then a ForwardHistorian agent needs to be installed on that instance with connection set up to remote central instance. The forwarder will then forward messages from the local to the remote instance for monitoring, control, or data visualization purposes. The remote instance needs to authenticate any incoming connection either through the command line or web interface. The central instance has a data historian (for example, SQLite or mongo or timescale historian) for storing data collected from the devices, results from experiments, etc.

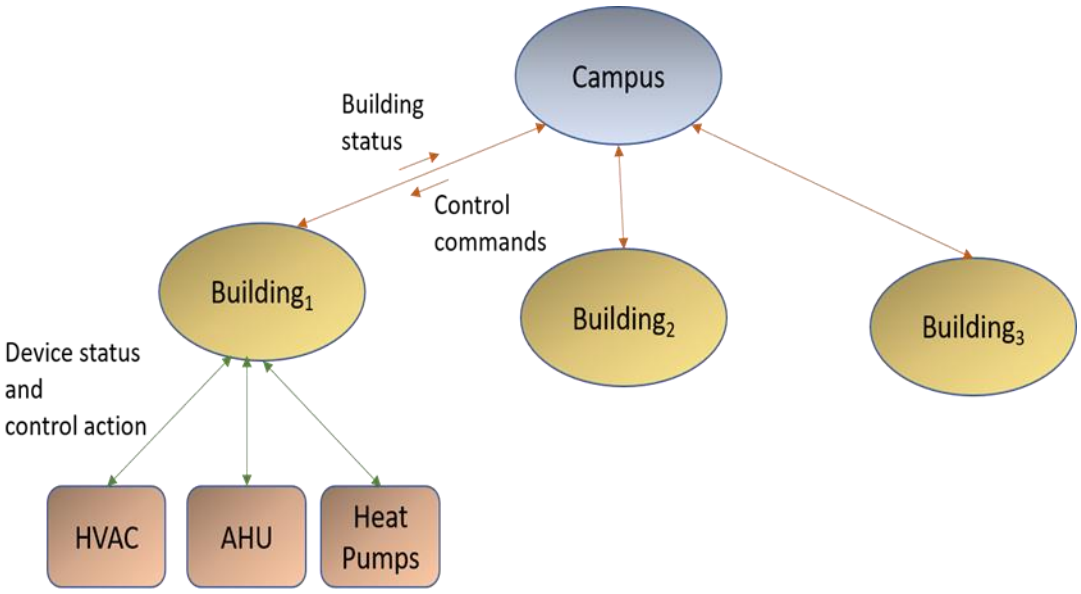

Figure 2. Typical VOLTTRON deployment.

7.1 Simple Installations

ZMQ-based VOLTTRON installation steps are easy and self-contained. Non-software engineers can quickly bootstrap the environment with minimal steps and start running the platform. It provides easy-to-use and robust security feature using the ECC key mechanism that is known to provide high security with short, fast keys. If an agent on one instance, V1, wants to connect to another instance, the public key of the agent can be easily copied over to a remote instance to provide authentication to the connecting agent. In contrast, RMQ-based VOLTTRON installation has several more steps with respect to configuring the RMQ broker and setting up SSL certificates for the VOLTTRON instance and its agents. During the setup process, each VOLTTRON instance needs to be configured to create its own CA and generate certificates signed by same CA and configure the RMQ broker and agents to use the certificates properly. This process has been automated but involves more steps than ZMQ-based VOLTTRON's single instance setup. Establishing connections between multiple connections is more complex when using SSL certificates. For example, if an agent on one instance, V1, wants to connect to another instance, V2, it must initiate a web-based certificate-signing request operation with the remote instance. The administrator/platform owner on the remote instance can accept/reject the incoming connection request. If accepted, the remote instance creates a signed certificate and returns it to the agent, which is then used to establish the remote connection. Similar to agent authentication, we are in the process of streamlining federation and shovel connections with SSL certificate authentication.

ZMQ-based VOLTTRON is perfect for single platform deployment or multi-platform deployment use cases that have fewer VOLTTRON instances connecting to a central instance or to each other. ZMQ-based VOLTTRON can hold ~100 (based on Section 6.2) agents per VOLTTRON instance without any degradation in the message bus performance. ZMQ-based VOLTTRON can handle a low to medium volume of traffic, as shown in the benchmark results in Sections 6.2 and 0. As the number of devices and payload size became very high, performance of ZMQ-based VOLTTRON starts to degrade. This is because the router module within the VOLTTRON process that is responsible for routing all the messages becomes a performance bottleneck. Significant changes must be made to ZMQ-based VOLTTRON to add back pressure capabilities

and high availability measures for very large-scale deployment. Typically, message-based systems become overloaded when a publisher is sending messages at a rate faster than the consumer can accept and process. We can then apply back pressure mechanisms to regulate the traffic. Some of the measures are forcing the publisher to stop sending until messages are consumed, discarding messages if the message bus limit is reached, or creating pull-based systems where consumers pull the messages from publisher's queue when it is ready. RMQ-based VOLTTRON has these features and hence is recommended for large-scale deployment.

7.2 Installation with a Very Large Message Payload

RMQ-based VOLTTRON is very well suited for deployment scenarios in which the agents are sending very high volumes of data over the message bus; for example, an agent configured to collect data from a few hundreds of devices, each having 500–1000 data points, and publish the data collected over the message bus at regular intervals. As shown in Section 0, in such a scenario RMQ-based VOLTTRON can handle the traffic much better than ZMQ-based VOLTTRON.

7.3 Deployment Options

For a multi-platform connection, many of the features had to be custom-built in ZMQ-based VOLTTRON. For example, for forwarding messages from one platform to another, a custom ForwardHistorian agent with a caching feature had to be created in ZMQ-based VOLTTRON. The RMQ library provides shovel plugin that performs a similar operation. The shovel plugin allows users to reliably and continually move messages from a source in one broker to a destination in another broker. A shovel behaves like a well-written client application that connects to its source and destination broker, consumes messages from the source queue, and re-publishes messages to the destination if the messages match the routing key (or the topic). RMQ-based VOLTTRON has an integrated shovel plugin feature and by taking a few simple steps the shovel can be reconfigured to forward messages of desired topics from a local VOLTTRON instance to a remote VOLTTRON instance. One drawback of the shovel plugin compared to the ForwardHistorian agent is that it has limited caching capability and when connection to a remote instance is lost, the data may be lost after a pre-configured maximum cache size is reached.

Another way to connect multiple platforms is to make and manage the connections at the platform level. This alleviates the need for individual agents to connect to the remote instance directly for sending/receiving messages to/from the other platform. In ZMQ-based VOLTTRON, this is accomplished through customizations in the router module, which is responsible for routing messages. With this type of connection, agents can send and receive messages to and from other platforms without explicitly managing the connection. All the connected platforms can work together as group. This type of connection provides loose coupling between platforms; i.e., not everything needs to be shared with other platforms, and there can still be local-only components and messages. This is a very useful feature. One of the drawbacks of this custom-built feature is that it is not highly scalable because it needs $O(n^2)$ connections between n VOLTTRON instances. Another drawback is that multiple VOLTTRON instances cannot be daisy chained together.

Loose coupling between multiple RMQ instances can be achieved using RMQ's federation plugin. The federation plugin allows users to federate exchanges and queues. A federated exchange or queue can receive messages from one or more upstream (remote) exchanges and

queues on other brokers. A federated exchange can route messages published upstream to a local queue. A federated queue lets a local consumer receive messages from an upstream queue. This plugin does not require $O(n^2)$ connections for n brokers and hence scales better than its ZMQ-based counterpart. Another advantage of federation is its ability to daisy chain multiple brokers, which provides more flexibility in deployment. RMQ-based VOLTTRON integrates with the federation plugin and enables loose coupling between VOLTTRON instances with just a few simple configuration steps. This allows agents to publish/subscribe to messages and make RPC calls to each other without having to manage the connections themselves. It is also highly scalable, so users can connect numerous buildings spread over a large geographical area. Clusters of VOLTTRON instances also can be tightly coupled to each other using RMQ's cluster plugin. This is not currently integrated with RMQ-based VOLTTRON but can be integrated if there is a community need.

Consider an example energy management system shown in Figure 3.This system forecasts and optimizes zonal energy demand by Machine Learning (ML) of occupant behavior and environmental data from each zone to improve occupancy comfort and reduce energy cost. VOLTTRON is installed at the campus, building, and zonal levels. The VOLTTRON instances at the zonal level are configured to collect data from and control devices in their respective zones. Typical devices are HVAC, lighting, and smart plugs. The device information is sent upward to the building-level VOLTTRON and similarly building-level status is sent to the campus node to be aggregated at the campus level. The building status and occupancy behavior are also continuously sent to a ML application in the cloud. The ML application uses the occupancy behavior information and environment data to learn the behavioral pattern and make real-time predictions for reducing energy demand and increasing occupancy comfort. Based on those predictions, control commands are sent to building and zonal VOLTTRON instances for adjusting the setpoints of the respective devices. The connection between building-level VOLTTRON instances and ML applications in the cloud needs to be secure to prevent cybersecurity attacks on the campus infrastructure if the ML application is compromised. The ML application is built using either RMQ or MQTT or Kafka message libraries.

If a ZMQ-based VOLTTRON installation is used for this system, the VOLTTRON instances at campus, building, and zone levels can be connected using custom-built multi-platform connection. A custom agent that allows integration with RMQ/MQTT/Kafka with proper security features must be created to forward messages from building instance to ML application and back. Additional authorization features need to be added to control access of resources in building level VOLTTRON instances and restrict traffic being sent from ML application to VOLTTRON.

If an RMQ-based VOLTTRON installation is used, then federation plugins can be used to connect the campus, building and zonal instances. We can create two-way federation links between zone and building-level VOLTTRON instances to send device status and control actions. Similar two-way federation links can be created between building and campus-level instances to send building-level status and control actions. If the ML application is built using MQTT or Kafka, RMQ provides easy integration with these libraries in a secure manner using SSL certificates. If the ML application is built using RMQ, then a one-way federation link needs to be created to forward messages from the building to the ML app. A two-way connection is not recommended because it would expose campus infrastructure to components in the cloud. Instead, a shovel connection can be created to send control commands from the ML app to the building-level VOLTTRON instances. In this way, users can limit the number of topics that can be sent in this direction. Since it is a matter of configuring existing and tested features in RMQ-based VOLTTRON, using RMQ-based VOLTTRON is recommended for this use case.

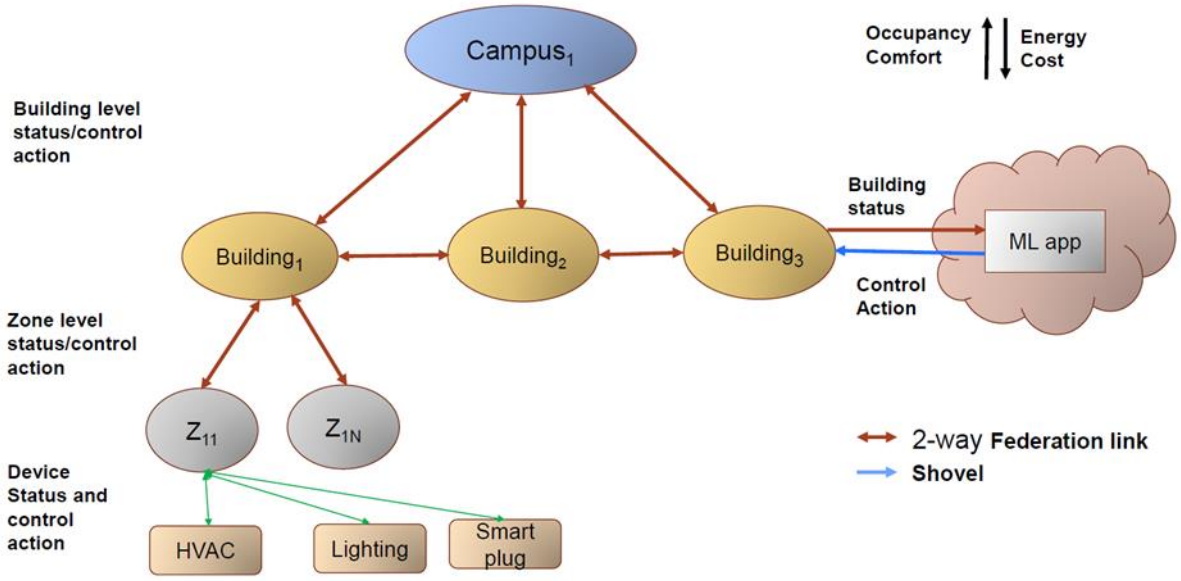

Figure 3. Flexible deployment.

8.0 Integration with Third-Party Tools

8.1 Message Queuing Telemetry Transport

MQTT is an Organization for the Advancement of Structured Information Standards standard message protocol for Internet of Things applications. It is widely used in the automotive, manufacturing, telecommunications, oil, and gas industries. Recently, there has been wide-spread use of MQTT in the buildings and power systems domain for data collection and visualization applications. Many VOLTTRON users need VOLTTRON and MQTT based applications be integrated together such that these heterogenous systems can work together. One such user request is to integrate the LORAWAN gateway with VOLTTRON to collect data from air quality sensors to monitor the air quality in buildings. In addition, the LORAWAN gateway can connect to many other edge devices such as vending machines, water meters, etc., each speaking different message protocols but primarily MQTT. For VOLTTRON to collect data and send control actions to these edge devices, it must have an integration mechanism with MQTT. RMQ-based VOLTTRON uses RMQ's MQTT plugin (https://github.com/VOLTTRON/external-clients-for-rabbitmq/tree/master/mqtt-volttron-client) to establish two-way communication with MQTT devices. In contrast, if ZMQ-based VOLTTRON is used, custom-built MQTT historian agent can be used to send data from VOLTTRON to MQTT client. But the agent must be extended for data to flow from MQTT client to VOLTTRON.

8.2 RabbitMQ

RMQ-based VOLTTRON can be easily integrated with any other RMQ-only client application. An example use case would be a cyber defense RMQ-based application that detects any malicious data in building sensor measurements. Here, VOLTTRON collects data from buildings network and sends them to an RMQ-based analysis module containing forward decision and

classifier components. If the analysis module declares the data malicious, they are not passed to the destination endpoint. Because the analysis module uses the RMQ messaging library, RMQ-based VOLTTRON can be set up to collect data from the building network. The RMQ-based analysis module (INGRESS) can connect to the same broker as an external client and read data from VOLTTRON's message bus (Figure 4). To provide the same integration in ZMQ-based VOLTTRON, a custom agent needs to be created to forward data to the RMQ-based INGRESS application.

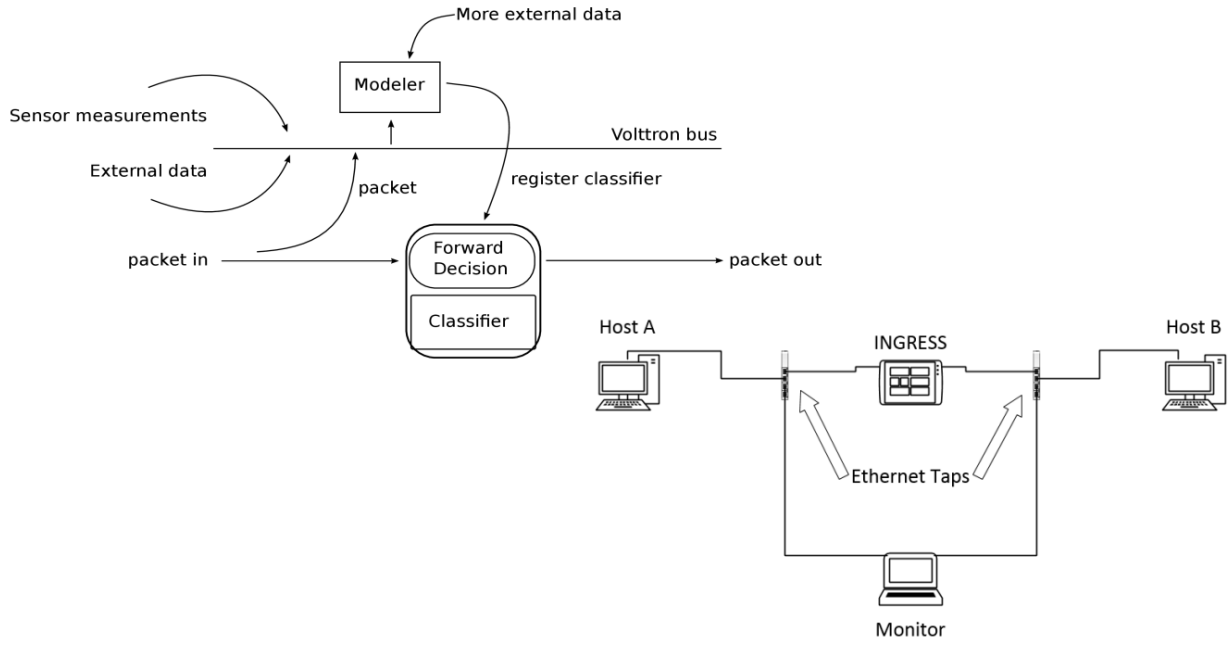

Figure 4. INGRESS application.

8.3 Elasticsearch

Elasticsearch is a tool used to reliably and securely search, analyze, and visualize data in real time. It is typically used for performing visual analytics on the ingested data. Elasticsearch can be used to ingest buildings data and perform data analysis to detect anomalies in data, such as the zone temperature being higher than expected, etc. RMQ-based VOLTTRON can be integrated with Elasticsearch using Apache Nifi (https://nifi.apache.org/docs.html), which is a data flow management tool. Apache Nifi can be configured to pull data from any RMQ data source (in our case it will be RMQ-based VOLTTRON) and forward the data to Elasticsearch. It essentially creates a data pipeline between the two endpoints. After the data are ingested by ElasticSearch, they can be stored in Kibana and users can use various features in Elasticsearch to create data visualization and anomaly detection applications. Integration of Elasticsearch with ZMQ-based VOLTTRON would involve creation and maintenance of a custom agent to forward data to Apache Nifi and Elasticsearch.

9.0 Conclusion

In general, simpler VOLTTRON deployments are good fits for the ZMQ-based message bus: single/low number instance install, little peer-to-peer interaction, and interacting with devices with existing drivers. RMQ comes into the forefront for the more complex deployments: large

numbers of instances communicating peer-to-peer and interacting with MQTT and other third parties.

This document will continue to be updated to remain current with platform development and as the community gains experience using the message buses in different deployments. We welcome users of the platform to engage with the authors to relate their experience and help make this guidance document as relevant to the community as possible.

Pacific Northwest National Laboratory

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99354
1-888-375-PNNL (7665)

www.pnnl.gov