ISSN XXXX XXXX © 2017 IJESC

Research Article Volume 7 Issue No.7

A Publisher/Subscriber Model Based Event Notification System for a Storage Operating System

Asha R[1], Dr Nagaraja[2]
M. Tech Student[1], Professor & PG Coordinator[2]
Department of ISE
Bangalore Institute of Technology, Bangalore, India

Abstract:
A table look up through the kernel has to be performed in order to get the information about the storage activities being performed in the storage operating system currently. This causes a considerable amount of latency as the table look up involves context switching. A Publisher/Subscriber model based event notification system reduces this latency as it works on the principles of pub-sub model.

Keywords: Message Queue Telemetry Transport (MQTT), VerneMQ, Storage Operating System (S-OS)

I. INTRODUCTION

The storage operating system maintains all aspects related to storage. It is responsible for allocation and deal location of storage space, garbage collection and maintenance. It consists of filers. The storage space is divided into various chunks called filers where all the operations occur.. Hence filers are considered to be the source of storage events. This makes them the publishers. A publish-subscribe architecture is a messaging pattern in which the sending party, known as the publisher, do not process the message being sent to the receiving party, known as the subscribers. They categorize the messages into topics. The concept of topics is used to ensure that the receiving parties get only the messages that they have shown their interest in i.e., to which they have subscribed to. So instead of the client-server based model, the pub-sub model provides less latency as the messages are passed to the users without much of processing. A sample of publish-subscribe architecture looks as given below.

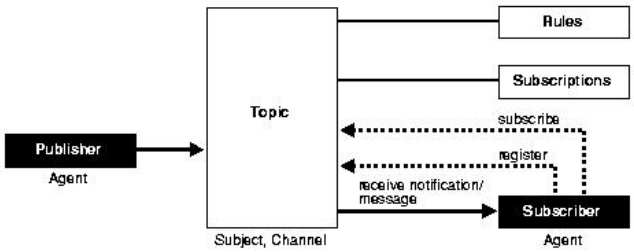


Figure.1. Publisher/Subscriber Functionality

The user registers to a queue with an interest to get the messages that are available on that queue. The data to be exchanged between the user and the publisher is based on a filter or set rules that filter out the messages. During the course of execution, senders send the messages to different queues. From there the underlying delivery mechanism will deliver the message that accord with the various registrations and then it is delivered to appropriate subscribers. Each topic of registration has a unique queue. The developer can decide the size of the queue. The rule on a queue is mentioned as a conditional expression that uses a bunch of pre-specified operators on the format attributes of the message or on the header. Every queue has a message format for the content that explains the structure of the memo signified in that queue. Some of the formats are defined in UTF, BCD etc. Along with the pub-sub model described above we make use of Message Queue Telemetry Transport (MQTT) protocol for message exchange. It is a publish-subscribe, very simple and lightweight protocol used in messaging. The ideologies behind the design are to reduce network bandwidth and resource consumption by the device along with making an attempt ensure reliability and assurance of delivery to an extent. These design principles are suitable for the emerging technologies. MQTT nodes make use of a mapping model to interconnect where a message sent by the sender is given to many receivers. This type of mapping is known to be one - to-many Messages are exchanged via a fundamental node known as the broker. A simple overview of MQTT operation can be as shown in the below figure.

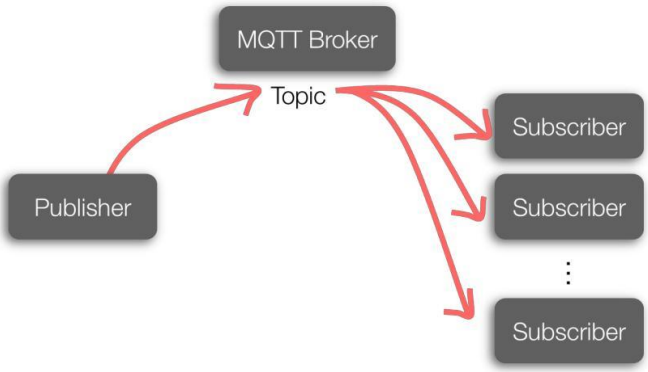


Figure.2. Protocol Overview

A single packet in MQTT comprises of a header that is fixed with a size rage between two to five bytes. This makes the protocol very light when compared to HTTP, SNMP etc. The fixed header is followed by a variable header and payload which is optional. The header starts with a 4-bit packet type identifier and some header flags. The following bytes starting from the second are used to convey the optional header and payload. The protocol uses a scheme of variable length to accommodate the rest of the information. This is in the range of one byte to four bytes. The most significant bit in every bite is considered to be the continuation bit. If this bit is 0 it is the last

byte of the data else if its 1 then there are more bytes to follow. Always the continuation bit of the last byte is zero. In a single byte, values up to 127 can be encoded. There are three different levels of Quality of Service (QoS) supported by MQTT that is MQTT supports three levels of QoS, detailed by every published message and when subscribers are connecting to the broker. **QoS 0:** Also known as the best effort mechanism, this level of service does not wait for an acknowledgement. The sender sends the data and forgets about it. **QoS 1:** Every message sent by the publisher should be acknowledged by the receiver. In case the acknowledgement is not received, the sender resends the message to the receiver after a fixed time interval of time. This service guarantees the delivery at least once. **QoS 2:** The sender sends the message only once and to ensure this the sender and the receiver exchange 4 different types of messages. There are no duplicates sent during this level of QoS

When discussing the QoS there are two different portions of sending a message:

(1) Sending the published message to the broker (2) sending the message to appropriate subscribers. These are mentioned separately as there will be subtle differences. The publisher to broker QoS level is dependent on the QoS level the publisher decides for the message. When the broker transfers a message to a subscriber it makes use of the QoS mentioned by the subscriber at the stage of registration. Based on the decided levels the QoS can be upgraded or downgraded. A summary of the QoS features can be as shown below

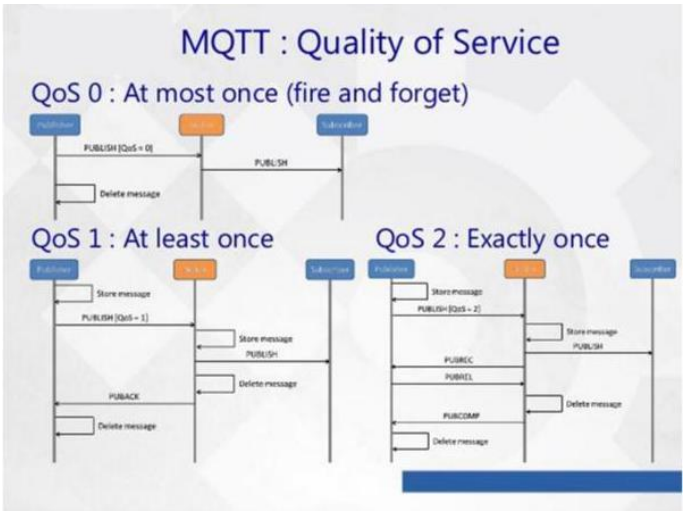


Figure.3. Qos Levels in MQTT

The sequence of the protocol of MQTT is very basic. The connection is initialized by a client that seeks permission from the broker to connect to it. Once the connection is acknowledged, the customer sends a subscription request with the topic of interest. The broker acknowledges this message and processes it. Then it sends the requested information to the customer.

Later MQTT commands are exchanged over the connection. Apart from the publisher and the subscriber, there is a third element in the pub-sub model known as the message broker, sometimes also referred to as the agent. In this flavor of the event notification system the broker used is the VerneMQ message broker. VerneMQ is primarily a MQTT publish /subscribe message broker which implements the MQTT protocol. But VerneMQ is also built to take messaging and IoT applications to the next level by providing a unique set of features related to scalability, reliability and high-performance as well as operational simplicity. VerneMQ uses a master-less clustering technology. There are no special nodes like masters or slaves to consider when the inevitable infrastructure changes or maintenance windows require adding or removing nodes.

This makes operating the cluster safe and simple. The proposed system is a real-time event notification which logs and sends notifications to the subscribed users. Every storage related activity that happens on the filer is considered as an event. Therefore, all the events are picked up by the publisher and sent to the message broker. The message broker looks for the subscribers for that topic and sends the message to those subscribers. Along with sending the notification to the subscribers, the events are also logged as and when the activity occurs. The proposed system is also useful in the case of any unwarranted activity occurs on the filer due to code corruption or unexpected behavior of the new code integrated to the system. The debugging of the code becomes simpler as the developer need not wait till the testing phase. When a deletion occurs on the filer, a notification is generated and the user is notified. This helps him check for the bug in the code in the early stages of development

II. RELATED WORK

In Doc.oracle, 2002[1], it is stated that the Publish/Subscribe form and Communication Queueing both form the basis of any MQTT deployment and use case. But it's vital to explain that MQTT is an TCP/IP based application level protocol. Many agents and client libraries only provision the TCP/IP layer; although in theory MQTT could use diverse transport layers too. The major differences between VerneMQ and other message brokers are discussed in the article of Erlang 2017and it states that the main difference relies on the underlying distributed storage engine. Emqtt and RabbitMQ makes use of Mnesia[2] (Erlang, 2017) for its underlying distributed storage engine (subscriptions, configurations, and retained messages). Whereas VerneMQ uses an eventual consistent datastore backed by LevelDB.

Mnesia is a distributed database that is bundled together with the Erlang programming language, so it might be the logical choice when you implement a message broker. As a matter of fact VerneMQ used this method in the beginning too. But soon later used plumtree implementation based on Basho's great Riak distributed KV store and adapted it to use LevelDB according to an article in GitHub 2016[4]. J Leitao, J Pereira, L Rodrigues (2007)[3], describes plumtree as an epidemic broadcast protocol that uses an algorithm to efficiently broadcast messages to thousands of nodes in an ever-changing network where cluster nodes are dynamically added or removed, network partition occurs and network latencies are real.

III. METHODOLOGY

Keeping track of what operations are performed on the filers is important for an efficient storage management. Reacting to the events that occur as a result of code corruption and mismatch at the earliest is necessary to ensure non-disruptive working of the system. To do this a simple publisher/subscriber model based event notification system is proposed. The methodology implemented is better understood with the use case diagrams

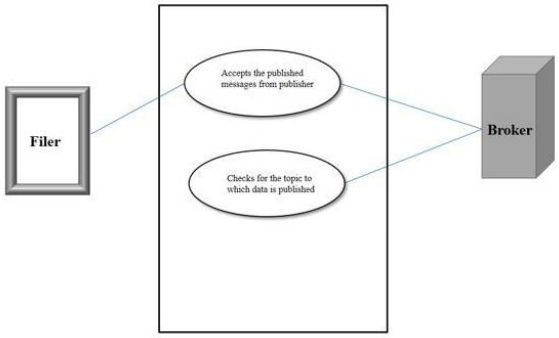


Figure.4. Use case Diagram for Interaction between Filer and Broker

As seen above the system involves three main nodes, 1. Filer, which plays the role of publisher, 2. Broker, in this case the VerneMQ message broker, 3. Subscriber, the users of the storage operating system. The filer is a part of the storage operating system where the storage related operations such as creation, deletion and update of volumes, aggregates etc., occur. Hence this is made the publisher. Subscribers are the users of the storage operating system who create and maintain storage spaces. Hence they require information about the activities that are happening on the filer. The broker is the mediator between the users and the filer. It maintains the list of topics, subscribers who have registered for a particular topic, access control lists and it also has a mechanism for authenticating the parties. As evident from the use case diagram, the filer publishes the events such as a new creation of a volume to the broker with the volume name and space instantly after a successful creation. The broker processes the header information of the message and compares it with the list of topics stored to check which all subscribers must get the message.

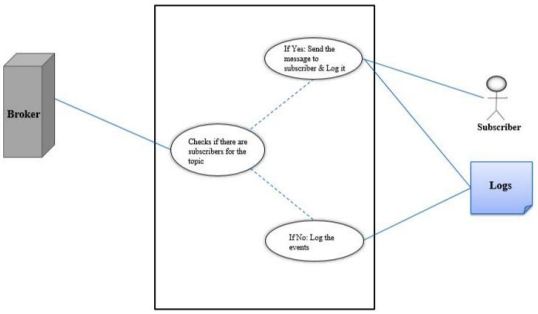
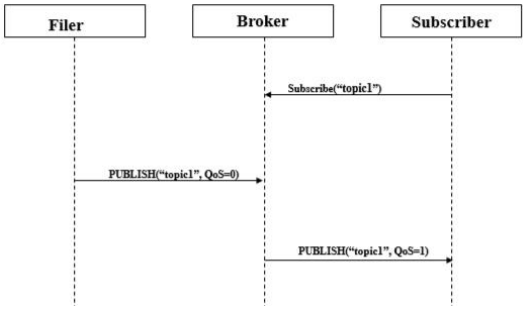


Figure.5. Use case diagram for Interaction between Broker and Subscriber

The next step is to send the message or notification to all the subscribers that matched the topic list. Along with sending it to the users, the broker logs the notification message to a file that is accessible to developers, testers and some customers. The overall functioning of the system can be summarized as a sequence diagram shown below.



Built-in client libraries that are available for the broker are used to integrate the broker with the storage operating system. By inheriting from the client classes, the filer is made to interact with the broker. Once a connection is established between the filer and the broker, the next step is to connect the users to the broker. The users or subscribers register with the broker. The users also inherit certain class functions from the library to register. The processing happens at the broker and the users get the appropriate messages. The notification can be either printed on the standard output or can be sent as a system generated email. Consider a case where a user has not subscribed for a topic, but at some point he needs to see what has appened. To serve this purpose all the events, regardless of whether there are any subscribers or not, are logged into a log file which is accessible to all the legitimate users with login credentials.

IV. FLOWCHART

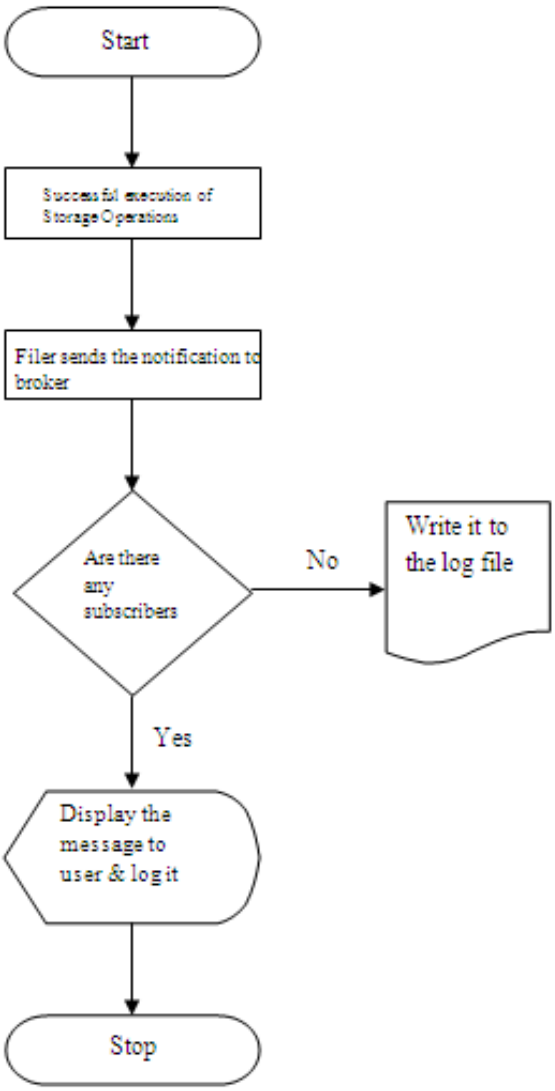


Figure.6. Functionality of the System

The workflow is briefly explained as follows:

As it is said earlier, the filer is the publisher, hence the actions starts at the filer once the setup phase is done.

- The broker is started and is continuously waiting for the input from the filer.
- The filer, as and when an event occurs i.e., an operation that changes the storage space state successfully occurs, sends the notification to the broker.
- The broker, on receiving the message, checks the topic-subscriber list.
- If there are any subscribers, sends the message to the subscriber and also writes to the log file.

- If there are no subscribers, it will write to the log file and discard the message.

V. EXPERIMENTAL RESULTS

Some of the benefits of using this method are:

- No external hardware required to implement this proposed event notification system.
- The different QoS levels offered by the protocol provide a choice to the developers and users to use the one that suits their environmental requirements.

The results are obersved and it is noted that there is a 10% gain on the overall performance of the event management system. The log file is more up to date as it is updated as soon as an event occurs. This make the logs of almost real time precision.

VI. CONCLUSION

The proposed solution gives an optimistic solution to notify users of the events that are being performed on the storage filers. The logging of the events happened two times faster. Along with logging the subscribed users are notified of the events. The debugging becomes easier when constantly accessing the storage space. The unwarranted operations that occur due to code corruption are brought to the notice of the developers well ahead of the testing phase. The system has been tested on the virtual machines so far and the future work includes porting it to the actual operating system that handles the storage management.

VII. REFERENCES

[1]. Doc.oracle (2002) 'Using the Publish-Subscribe Model For Applications'

[2]. Erlang (2017), 'Mnesia'. Availabe online at http:// erlang.org/doc/apps/mnesia/Mnesia_overview.html

[3]. Leitao, J., Pereira, J., & Rodrigues, L "Epidemic broadcast trees. In Reliable Distributed Systems", 26th IEEE International Symposium on (pp. 301-310). IEEE.

[4]. GItHub (2016), 'VerneMQ VS eMQTT'. Available online at: https://github.com/erlio/vernemq/issues/83