A High Performance, Scalable Event-Bus System for Distributed Data Synchronization

[Supervisor: Wei Zhang]

Kiran Patrudu Gopalasetty
Texas Tech University
kiran-patrudu.gopalasetty@ttu.edu

Arun Kumar Jegarkal
Texas Tech University
arun-kumar.jegarkal@ttu.edu

ABSTRACT

Indirect communication, apart from direct communication mechanism like RPC and RMI, plays an important role in many distributed computing scenarios. As two of the most widely used indirect communication techniques, pub-sub system and message queues provide both space uncoupling and time uncoupling [1]. They can be used in many scenarios [2-4], such as message delivering during streaming processing, data synchronization between replicas for fault-tolerance, asynchronous data preprocessing for real-time processing, inter-service communication, etc.

One of the biggest challenges in research is an effective dissemination of huge quantity of information from publisher to subscriber. The general model is based on delivering the information to consumer based on few simple static rules. As the system grows, we finds that these implementation are not effective neither feasible. As such, publish subscribe communication system is recognize as a supportive model for handling the propagation of information from publisher to subscriber.

In this project, we have designed an event bus system which combines the pub-sub system with message queue technology. We have implemented our pub-sub system using web-sockets that is what stands out from other pub-sub systems. We have synchronized the message delivery between the publisher-eventbus-subscriber using Avro. Most importantly, we have implemented the scalability by transferring the messages onto a different event bus whenever the messages in the first event bus reaches the threshold limit, thereby balancing the load of the event bus.

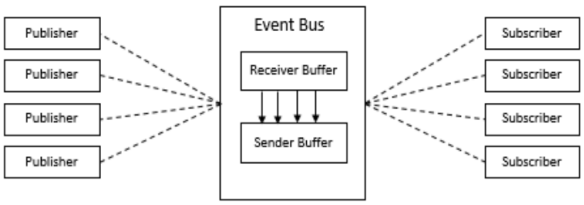
1. INTRODUCTION

Pub-Sub is a communication topology where a single entity called Publisher produces messages that it sends out to other entities called Subscribers. Subscribers may receive everything the publisher sends, or they may subscribe to message subsets called Topics.

Publishers publish events, and subscribers subscribe to and receive the events they are interested in. The main characterization of pub/sub is in the way notifications flow from senders to receivers. Receivers are not directly targeted from publisher but indirectly addressed according to the content of notifications. Subscriber expresses its interest by issuing subscriptions for specific notifications, independently from the publishers that produces them, and then it is asynchronously notified for all notifications, submitted by any publisher, that match their subscription.

EventBus allows publish-subscribe-style communication between components without requiring the components to explicitly register with one another (and thus be aware of each other). It is designed exclusively to replace traditional Java in-process event distribution using explicit registration. It is not a general-purpose publish-subscribe system, nor is it intended for interprocess communication.

Notification Service(i.e., Event bus) is a propagation mechanism that acts as an intermediary(middle-layer) between publishers and subscribers to avoid each publisher to have to know all the subscription for each possible subscriber. Both publishers and subscribers communicate only with a single entity, the Notification Service, that (i) stores all the subscriptions associated with the respective subscribers, (ii) receives all the notifications from publishers, (iii) dispatches all the published notification to the correct subscribers. With this, publishers and subscribers exchange information without directly knowing each other. This anonymity is one of the main features of the pub/sub system.



The publisher/subscriber (P/S) model has been recognized as an appropriate high-level communication scheme to connect autonomous components in large distributed control systems . Particularly, P/S supports autonomy of components by an asynchronous notification mechanism omitting

any implicit control transfer coupled with the exchange of information. This is in contrast to other high-level interaction models as remote procedure calls or remote invocations which create global dependencies and side effects which require complex mechanisms to handle temporal and functional fault situations. Secondly, P/S relies on a content-based communication scheme. This means that the content of a message is used to route a message rather than an address. A subscriber, which is interested in particular information, e.g. the temperature data of some sensor, subscribes to the particular information rather than to a specific sensor. This has the advantage that the subscriber does not have to know any specific sensor name or address and thus, a content-based addressing mechanism substantially encourages dynamic adaptability and extensibility requirements. Moreover, it may be the basis of transparent fault-tolerance mechanisms in which multiple sources provide the same information

Subscription strategies can differ based on system architecture-

If the system has a broker, clients subscribe to the broker rather than to the server, and the broker takes care of routing the messages to the clients based on the subscribed topics.

In broker-less systems, clients may either send their topics to the server, and the server then sends each message only to the clients that have subscribed for that topic.

Pub-sub system and message queues provide both space decoupling and time decoupling:

Space decoupling - Interacting parties do not need to know each other. Message addressing is based on their content.

Time decoupling - Interacting parties do not need to be active, the pub/sub interaction model participating in the interaction at the same time. Information delivery is mediated through a third party.

Or the client s filter the messages at their end. The server then simply sends all messages to all clients. (This approach is fine in smaller, local scenarios but does not scale well.)

2. MOTIVATION

Traditional publisher/subscriber communication model employs Remote Procedure Call, message queue, shared memory etc. They are synchronous and tightly - coupled request invocations. But the most important drawback of this Traditional publisher/subscriber communication model is that it is very restrictive for distributed applications, especially for WAN and mobile environments. Also, whenever node/links fail, system is affected and therefore, fault tolerance must be built in order to support this. It also requires a more flexible and decoupled communication style that offers anonymous and asynchronous mechanisms.

3. PROBLEM STATEMENT

To design an event bus system which combines the pub-sub system with message queue technology.

4. BACKGROUND

The web was built around the idea that a clients job is to request data from a server, and a servers job is to fulfill those requests. This paradigm went unchallenged for a number of years but with the introduction of AJAX around 2005 many people started to explore the possibilities of making connections between a client and server bidirectional.

Web applications had grown up a lot and were now consuming more data than ever before. The biggest thing holding them back was the traditional HTTP model of client initiated transactions. To overcome this a number of different strategies were devised to allow servers to push data to the client. One of the most popular of these strategies was long-polling. This involves keeping an HTTP connection open until the server has some data to push down to the client.

The problem with all of these solutions is that they carry the overhead of HTTP. Every time you make an HTTP request a bunch of headers and cookie data are transferred to the server. This can add up to a reasonably large amount of data that needs to be transferred, which in turn increases latency. If you are building something like a browser-based game, reducing latency is crucial to keeping things running smoothly. The worst part of this is that a lot of these headers and cookies are not actually needed to fulfill the clients request.

What we really need is a way of creating a persistent, low latency connection that can support transactions initiated by either the client or server. This is exactly what WebSockets provide.

We have used websockets in java for implementing our project. WebSocket is a protocol which allows for communication between the client and the server/endpoint using a single TCP connection. The advantage WebSocket has over HTTP is that the protocol is full-duplex (allows for simultaneous two-way communication) and its header is much smaller than that of a HTTP header, allowing for more efficient communication even over small packets of data.

The life cycle of a WebSocket is easy to understand as well: 1.Client sends the Server a handshake request in the form of a HTTP upgrade header with data about the WebSocket it is attempting to connect to. 2.The Server responds to the request with another HTTP header, this is the last time a HTTP header gets used in the WebSocket connection. If the handshake was successful, the server sends a HTTP header telling the client it is switching to the WebSocket protocol. 3.Now a constant connection is opened and the client and server can send any number of messages to each other until the connection is closed. These messages only have about 2 bytes of overhead.

In order to serialize the message delivery between the publisher-eventbus-subscriber, we have used Avro serialization technique.

Avro is a remote procedure call and data serialization framework developed within Apache s Hadoop project. It uses JSON for defining data types and protocols, and serializes data in a compact binary format. Its primary use is in

Apache Hadoop, where it can provide both a serialization format for persistent data, and a wire format for communication between Hadoop nodes, and from client programs to the Hadoop services.

It is similar to Thrift and Protocol Buffers, but does not require running a code-generation program when a schema changes (unless desired for statically-typed languages). Avro schemas are defined using JSON. Schemas are composed of primitive types (null, boolean, int, long, float, double, bytes, and string) and complex types (record, enum, array, map, union, and fixed).

Avro schema example:

```
{
 "namespace": "com.Avro",
 "type": "record",
 "name": "msgfmt",
 "fields": [
       {"name": "Topic", "type": { "type": "string","avro.java.stringImpl":"String"} },
       {"name": "Type", "type": ["int"]},
       {"name": "Message", "type": ["string"]},
       {"name": "Time", "type": ["string"]}
 ]
}
```

In addition to supporting JSON for type and protocol definitions, Avro includes experimental[support for an alternative interface description language (IDL) syntax known as Avro IDL. Previously known as GenAvro, this format is designed to ease adoption by users familiar with more traditional IDLs and programming languages, with a syntax similar to C/C++, Protocol Buffers and others.

4.0.1 *Existing Systems*

All the existing systems do not support fully transactional communication between the clients and the brokers. Also the Queues can be durable. This means, when messages are placed in the queue, the message will be stored on the disk before being made available to clients. If the server dies and restarts, the queue will retain all the messages written to that queue. We will now briefly discuss about some of the existing systems and how our system stands out from them.

4.0.2 *Mangos Approach*

Mangos uses special, protocol-aware sockets. In a PubSub scenario, the pub socket just sends out its messages to all receivers (or to nirvana if no one is listening). The sub socket will be able to filter the incoming messages by topic and only delivers the messages that match one of the subscribed topics. This is certainly a rather simple and robust approach, as the server does not need to manage the clients and their subscriptions but on the downside, as noted above, filtering on client side does not scale well with the number of clients.

4.0.3 *RabitMQ Approach*

The AMQP 0-9-1 Model has the following view of the world: Messages are published to exchanges, which are often compared to post offices or mailboxes. Exchanges then distribute message copies to queues using rules called bindings. Then AMQP brokers either deliver messages to consumers subscribed to queues, or consumers fetch/pull messages from queues on demand.

When publishing a message, publishers may specify various message attributes (message meta-data). Some of this metadata may be used by the broker, however, the rest of it is completely opaque to the broker and is only used by applications that receive the message. Networks are unreliable and applications may fail to process messages therefore the AMQP model has a notion of message acknowledgements: when a message is delivered to a consumer the consumer notifies the broker, either automatically or as soon as the application developer chooses to do so. When message acknowledgements are in use, a broker will only completely remove a message from a queue when it receives a notification for that message (or group of messages).

In certain situations, for example, when a message cannot be routed, messages may be returned to publishers, dropped, or, if the broker implements an extension, placed into a so-called ”dead letter queue”. Publishers choose how to handle situations like this by publishing messages using certain parameters. Queues, exchanges and bindings are collectively referred to as AMQP entities.

Exchanges and Exchange types:

Exchanges are AMQP entities where messages are sent. Exchanges take a message and route it into zero or more queues. The routing algorithm used depends on the exchange type and rules called bindings. AMQP 0-9-1 brokers provide four exchange types:

Name	Default pre-declared names
Direct exchange	(Empty string) and amq.direct
Fanout exchange	amq.fanout
Topic exchange	amq.topic
Headers exchange	amq.match (and amq.headers in RabbitMQ)

Exchanges can be durable or transient. Durable exchanges survive broker restart whereas transient exchanges do not (they have to be redeclared when broker comes back online). Not all scenarios and use cases require exchanges to be durable.

4.0.4 *IBM MQ publish/subscribe*

Publish/subscribe messaging allows to decouple the provider of information, from the consumers of that information. The sending application and receiving application do not need to know anything about each other for the information to be sent and received. Before a point-to-point IBM WebSphere MQ application can send a message to another application, it needs to know something about that application. For example, it needs to know the name of the queue to which to send the information, and might also specify a queue manager name.IBM WebSphere MQ publish/subscribe removes the need for your application to know anything about the target application. All the sending application must do, is put a IBM WebSphere MQ message, containing the infor-

mation that it wants, and assign it a topic, that denotes the subject of the information, and let IBM WebSphere MQ handle the distribution of that information. Similarly, the target application does not have to know anything about the source of the information it receives.There is one publisher, one queue manager, and one subscriber. A subscription is sent from the subscriber to the queue manager, a publication is sent from the publisher to the queue manager, and the publication is then forwarded by the queue manager to the subscriber.

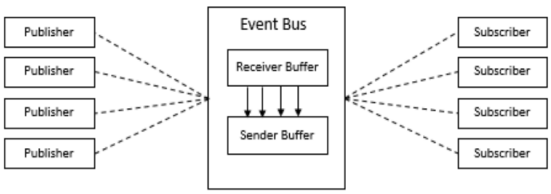
Considering the above approaches along with each of its advantages and disadvantages in implementing the pub-sub model, we have used java websockets to implement our pub-sub system which is unique from the existing systems as none of them till date have implemented the pub-sub system using websockets. We have designed our system in such a way that, multiple publishers will be able to connect to the event bus each one of them having its own unique session id once the connection is established between the publisher and the event bus. Similarly, multiple subscribers will be able to connect to the event bus each one of them having its own unique session id. Whenever a publisher publishes a message onto an event bus, we will be capturing the timestamp of when the message is published at the publisher end and when the message is received at the event bus end. This will therefore allow us to determine the latency of messages being delivered at both publisher and event bus end.

In order to serialize the messages being delivered, initially we have used java serialization and then we have enhanced the performance by implementing it with Avro which is one the serialization techniques used in Big Data.

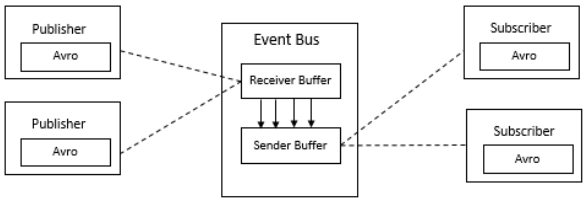
Considering all the above existing approaches, we have also done the scalability for load balancing the messages at the event bus end i.e., whenever the messages in the first event bus reaches to a particular threshold value, message delivery will be automatically transferred onto a new event bus thereby balancing the load at the event bus end and assuring a smooth flow of messages between the publisher-eventbus and eventbus-subscriber. This is the standout feature in our pub-sub system as none of the existing systems have considered the scalability for load balancing the messages at the eventbus end.

5. DESIGN

In this project, we basically focus to overcome the limitations of the existing techniques and our primary goal is to provide the support for fully transactional communication between the clients and the eventbus thereby making the communication more effective.



Above figure represents the complete architecture of our project where both the publisher and subscriber sends the handshake request in the form of a HTTP upgrade header with data about the EventBus WebSocket address itâĂŹs attempting to connect. Event bus responds to the request with another HTTP header, indicating handshake was successful, once the connection is established between the client(i.e., publisher/subscriber) and the event bus, publisher will be then able to publish the message to the event bus including all the required parameters with respect to the topic. The messages published by the publisher will be stored in the receive buffer at the event bus end. Once the event bus receives the message from the publisher, it checks for the information about all the subscribers and their registered topics. Now based on the topic id obtained in the message from the publisher, the eventbus then filters the subscribers to whom the message should be delivered and thereby delivers the message to the respective subscriber using the sender buffer present at the event bus end. Once the subscriber receives the message it sends the acknowledgment to the eventbus that the message has been received. Also, the publisher will be able to send different events in parallel based on the topic id.



We have serialized the message delivery between publisher-eventbus- subscriber by using Avro which is one of the serialization techniques used in hadoop, which uses the schema of message type which includes Typecode, Topic, Message, Timestamp. At the publisher end Serialize the message into bytestream and encode it into UTF-8 format and write it into sender buffer. At the eventbus end message packet is take and topic field is deserialized into the text format and to check topic and serialized and encode and transmitted into receiver buffer to broadcast to all the subscriber that subscribed to that topic. At the subscriber end message packet is taken from the receiver buffer and deserialized and decoded into string by the avro.

Below is the message format at the publisher end:

TypeCode	Topic	Message	Timestamp

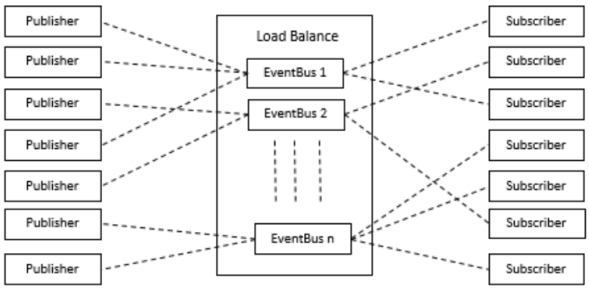
Below is the message format at the eventbus end:

TypeCode	Topic	Message	Publisher Timestamp	Received Timestamp

TypeCode used:

001 to achieve the behaviour of the Publisher

002 to achieve the behaviour of the Subscriber



Above figure gives a clear picture on how we have handled the scalability with respect to different event buses. We have done the scalability for load balancing the messages at the event bus end i.e., whenever the messages in the one event bus reaches to a particular threshold value then eventbus broadcast the some of the the subscriber and published related to a particular topic to change eventbus with a message type code of 003 with a message of change eventbus, then those subscriber and publisher reconnect to neighbor eventbus. Neighbor eventbus will then handle message delivery thereby balancing the load at the event bus end and assuring a smooth flow of messages between the publisher-eventbus and eventbus-subscriber. If neighbor eventbus reaches its threshold limit then it broadcast its switch to its neighbor eventbus this continues in load among all the eventbus in the network. As the number of messages increase at the eventbus end leading to the buffer overflow, the current eventbus load switches to a new eventbus in order to balance the load at its end.

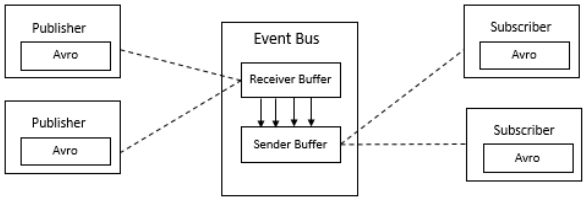
6. IMPLEMENTATION

Websockets replaces the native socket and multithreading concepts of java, which is the efficient way of establishing the pub-sub connection. Specifically, Annotation endpoint classes such as clientendpoint and serverendpoint are used for http connections. Websocket clientendpoint is used by the client to connect to eventbus and serverendpoint is used by the eventbus for connecting to the server and serving the subscribers.

Client (i.e., Subscriber/Publisher) connects to eventbus at the port number and registers to the respective topic which is passed as the argument. Once the connection is established a websocket session will be created and assigned with unique session id, Session info and the topic the subscriber is registered will be stored into the hashmap for the future reference at the eventbus end. Also there is a possibility for the subscriber to register to multiple topics. Therefore, if the subscriber register for multiple topics multiple entries will be create in the hashmap.

Once the publisher establishes the connection to the eventbus, a new thread will be created for publishing the messages to the eventbus. Publisher will send the message in the above message format while publishing the messages to the eventbus.

Message will be serialized by using Avro as shown in the below figure through which it is converted to the bytestream and writes into the sender buffer present at the eventbus end.



All the Messages will be sent to eventbus using session. getBasicRemote(). sendObject(dataFileWriter). Once the message is received at the eventbus end, OnMessage event will be triggered at the eventbus once the publisher publishes the message to the eventbus. Once the OnMessage event is triggered, the topic of the message will be checked and the Hashmap will be scanned in order to get the subscriber s registered for that topic and thereby message will be written into the receiver buffer at the eventbus end and broadcasts the messages to all the subscribers registered to the topic by using client. getBasicRemote(). sendObject(dataFileWriter). Whenever a message is received by the subscriber, Websocket OnMessage event will be triggered at the subscriber indicating message is received. Here the message received at the subscriber end will be deserialized from receive buffer.

6.1 For Load Balancing

If the EventBus is OverLoaded by the message i.e. whenever the message limit crosses a certain threshold set at the eventbus end, it broadcast the message to clients to switch the eventbus.

Message with TypeCode 003 and the new port number of neighbor eventbus to which it has to connect will be sent indicating the eventbus change event.

Once clients receive the Message of Typecode 003 the connection with the existing eventbus will be closed and a new connection to neighbor eventbus will be established thereby switching to a new eventbus and all the messages which has to be published will be then published to this new eventbus. Thus scalability among the event buses is achieved in order to load balance the messages.

6.1.1 Key Features

1.Publisher publishes the message to the event bus including all the required parameters.

2. Once the broker/event bus receives the message from the publisher, it checks into the information about all the subscribers and their registered topics.

3. Based on the topic id obtained in the message from the publisher, the event bus filters the subscribers to whom the message should be delivered. Thus, it delivers the message to all those subscribers whose topic id matches with the one in the message.

4. Once the subscriber receives the message it sends the acknowledgment to the event bus that the message has been received. Message delivery is synchronized between publisher-eventbus-subscriber by using Avro.

5. We have achieved scalability by balancing the load between the event buses whenever messages in one event bus reaches its threshold.

7. EVALUATION

We have conducted our evaluation on one of the Data-Intensive Scalable Computing Instrument (DISCI) server which has the below configuration

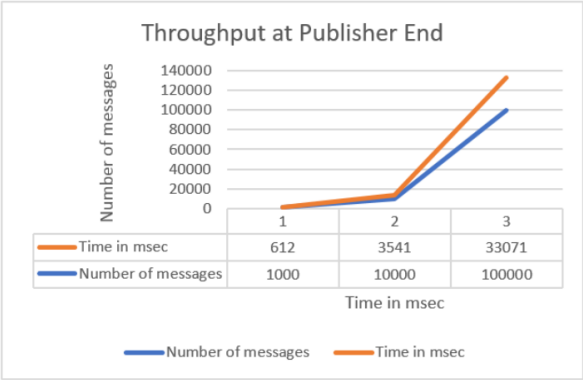
Server vendor model	Data-Intensive Scalable Computing Instrument (DISCI)
OS	Linux SMP x86_64
Memory (GB)	64 GB
CPU Version	Xeon E5-2650v2 2.6 GHz
CPUs count	8-core processors

We have measured the throughput at the publisher end by varying the limit of messages starting from 1000 and increasing it to 100000.

Publisher End:

Number of messages	Time in msec
1000	612
10000	3541
100000	33071

Above table show the time taken in milliseconds to publish the respective number of messages at the publisher end

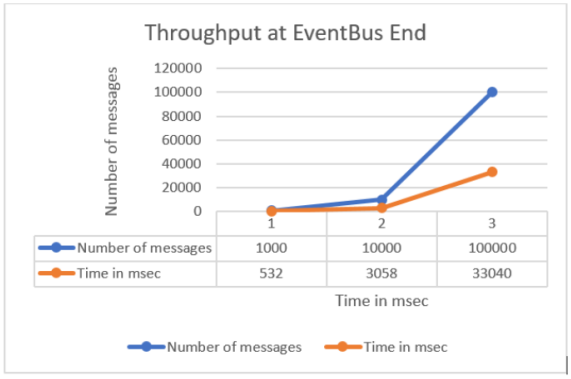


Above Graph represents the throughput at the publisher end

Event Bus End:

Number of messages	Time in msec
1000	532
10000	3058
100000	33040

Above table show the time taken in milliseconds to publish the respective number of messages at the event bus end



Above Graph represents the throughput at the eventbus end

Message delivery Latency:

Performance using Java Serialization -

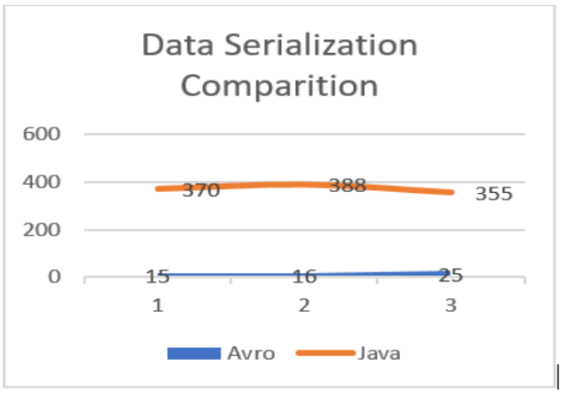
Published Time	Received Time	Time in msec
21:20:22.550	21:20:22.922	372
21:20:22.562	21:20:22.950	388
21:20:22.605	21:20:22.960	355
Average time		371.66

Above table shows the latency between the published time and the received time using Native Java Serialization

Performance Using Avro -

Published Time	Received Time	Time in msec
21:35:22.639	21:35:22.654	15
21:35:22.654	21:35:22.67	16
21:35:22.700	21:35:22.725	25
Average time		18.6

Above table shows the latency between the published time and the received time using Avro



Above graph represents the time comparison between Avro and Java serialization techniques

Time taken to switch EventBus:

Switch Message Published time	Switch Message Received time	Time in msec
22:25:35.492	22:25:35.47	22
22:25:40.564	22:25:40.56	40
22:25:40.630	22:25:40.659	29
Average time		30.33

Above table represents the Average time taken to disconnect from current even bus and establish connection to a new event bus

Latency after switching EventBus:

Published Time	Receiver Time	Time in msec
22:05:22.639	22:05:22.653	14
22:05:22.654	22:05:22.67	16
22:05:22.700	22:05:22.723	23
Average time		17.66

Above table represents the latency after switching to a new event bus using Avro serialization technique

Form the above table is we can observe that switching the event bus would not affect the performance of the system.

8. CONCLUSION

Achieved our goal for gaining high performance in Scalable Event-Bus System for Distributed Data Synchronization by using Web Socket as a communication path and Avro for serializing the data into byte stream for message publication greatly increased the performance. Load balancer handle the loads at the eventbus so that load is equally distributed among all the eventbus so that high performance is maintained.

Future enhancement can be made on Load Balancer like if there is only one publisher for the topic then instance of publisher can be created at the neighbor eventbus instead of switching all the subscriber and publisher Âărelated to that topic to other event bus. Another scenario can be only the some of the subscriber can be switched to other event bus and then eventbus can transfer messages related to that topic to other eventbus, so that load at the current event bus can be reduced.

9. REFERENCES

[1] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. 2011. *Distributed Systems: Concepts and Design (5th ed.)*.. Addison-Wesley Publishing Company, USA.

[2] H. Subramoni, G. Marsh, S. Narravula, P. Lai, and D. K. Panda, *âĂIJDesign and evaluation of benchmarks for ïňĄnancial applications using advanced message queuing protocol (amqp) over inïňĄniband,âĂİ in High Performance Computational Finance, 2008*. WHPCF 2008. Workshop on. IEEE, 2008, pp. 1âĂŞ8.

[3] Sadooghi, S. Palur, A. Anthony, I. Kapur, K. Belagodu, P. Purandare, K. Ramamurty, K. Wang, and I. Raicu, *âĂIJAchieving efiňĄcient distributed scheduling with message queues in the cloud for many-task computing and high-performance computing,âĂİ in Cluster, Cloud and Grid Com- puting (CCGrid), 2014*. 14th IEEE/ACM International Symposium on. IEEE, 2014, pp. 404âĂŞ413.

[4] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki, *âĂIJFlexpath: Type-based publish/subscribe system for large-scale science analytics,âĂİ in Cluster, Cloud and Grid Computing (CCGrid), 2014*. 14th IEEE/ACM International Symposium on. IEEE, 2014, pp. 246âĂŞ255.

[5] *Mangos Technique, https://appliedgo.net/pubsub/.*

[6] *Queues, http://queues.io.*

[7] *Pivotal, RabbitMQ - AMQP concept, https://www.rabbitmq.com/tutorials/amqp-concepts.html.*

[8] *Redis.io, Redis Pub-Sub, https://redis.io/topics/pubsub.*

[9] Jai Dayal, Drew Bratcher, Greg Eisenhauer, Karsten Schwan, Matthew Wolf, Xuechen Zhang Georgia Institute of Technology, Atlanta, GA Hasan Abbasi, Scott Klasky, Norbert Podhorszki Oak Ridge National Labs, Oak Ridge, TN *"Flexpath: Type-Based Publish/Subscribe System for Large-scale Science Analytics"*.

[10] Iman Sadooghi, Sandeep Palur, Ajay Anthony, Isha Kapur, Karthik Belagodu, Pankaj Purandare, Kiran Ramamurty, Ke Wang, Ioan Raicu Department of Computer Science, Illinois Institute of Technology, Chicago IL, USA *"Achieving Efficient Distributed Scheduling with Message Queues in the Cloud for Many-Task Computing and High -Performance Computing"*.