Thirty Meter Telescope (TMT)

Event Service Prototype – Solution Approach

Persistent Systems Ltd
Bhageerath, 402 Senapati Bapat Road
Pune 411 016, India

1. Introduction

Thirty-Meter-Telescope (hereafter referred as TMT) is a collaborative project to build a 30-meter telescope and related software applications. This project is collaboration between Department of Science and Technology of India, the Association of Canadian Universities for Research in Astronomy (ACURA), the California Institute of Technology and other entities.

Along with construction of the actual Telescope itself, other related software services are being built as part of this activity. One of the critical software service being planned is the Event Service.

The primary design consideration for the event service is to use a third party messaging platform. As a part of the event service project, some candidate messaging platforms will be chosen for evaluation that will have to undergo performance benchmark tests to validate their performance characteristics against the performance requirements of TMT.

Once the final message product is selected, an Event Service prototype would be developed as part of third phase of the project

2. Purpose of this document

The purpose of the documents it to capture the solution approach for Event Service Prototype to address the requirements and document key decisions taken

3. References

1. OSW TN010-EventServiceAPINotes_REL01.pdf
2. TMT Event service RFP

4. Assumptions

Following are the assumptions made while arriving at the solution for the prototype

1. The purpose of the prototype is to demonstrate the use of HornetQ through an EventService API. The prototype implementation of the EventService API would be a wrapper that provides basic capabilities for publish, subscribe and unsubscribe. It would not be a production ready API which requires proper analysis of the interface usage in the given content and design. However, prototype would help in taking decisions and development of code for production state

2. It is assumed that in context of TMT CSW, following are the very likely scenarios

    a. Multiple subscribers for a given topic

    b. Multiple instances of the same subscriber for a given topic

    The above scenarios would help to achieve scalability in case of high volume of events on a topic

3. There could be other option to achieve scalability - A single instance of a subscriber subscribing to same topic using multiple callbacks. However, it not true scalability. A production state would involve cluster of subscriber components subscribing to the same topic using same callback. Hence, this option is not being considered for the implementation.

5. Key Decisions

5.1. Topic and Queues

During performance evaluation, temporary subscriber queues were created (which are live only during the HornetQ session) in the code. Temporary queues by nature are not durable. Also, messages were configured to be non-durable

For the prototype,

1. All messages and topics would be set as non-durable (with provision to change the configuration)

2. Subscriber queues would be created from within the program using some logic at runtime

3. A queue would be created that will exist as long as there are consumers. When the last consumer is closed the queue will be deleted. It would avoid any message accumulation

Pros

- Subscriber queues would be created dynamically when consumer subscription happens to a topic

- No additional configuration required

Cons

- Not an ideal approach. Administrative tasks (i.e. queue creation/deletion) should be separated from programming logic

5.2. Multiple Call Backs Implementation

Each subscriber and multiple instances of a subscriber could subscribe to the given topic. When there are multiple instances of the subscriber subscribed to the same topic, only one instance would receive the message

With above assumptions, interface methods would implemented as follows

i. Subscribe(): A subscriber would subscribe to a FQN Topic using a callback. If same subscriber instance again subscribes to the same topic, it would result in an exception

ii. Unsubscribe(): Subscriber instance could unsubscribe the callback from the topic. On unsubscription, subscriber instance would not receive any event. In case, subscriber unsubscribes from a topic for which no subscription exists, EventService API would throw an Exception

iii. UnSubscribeAll(): All subscriptions on the given topic would be unsubscribed for a given subscriber(i.e. All subscriber instances would unsubscribe from the topic). If there are no subscriptions to the topic, EventService API would throw an Exception

5.3. ES-1020

Instead of demo web application, Standalone demo java programs would be used to showcase API usage. Programs would be interactive to support publish, subscriber and unsubscribe operations

Pros

- Since the objective is to showcase the usage, standalone programs are quick to develop, debug and add functionality
- Easier to test use case with multiple instances of the same subscriber subscribing to the same topic

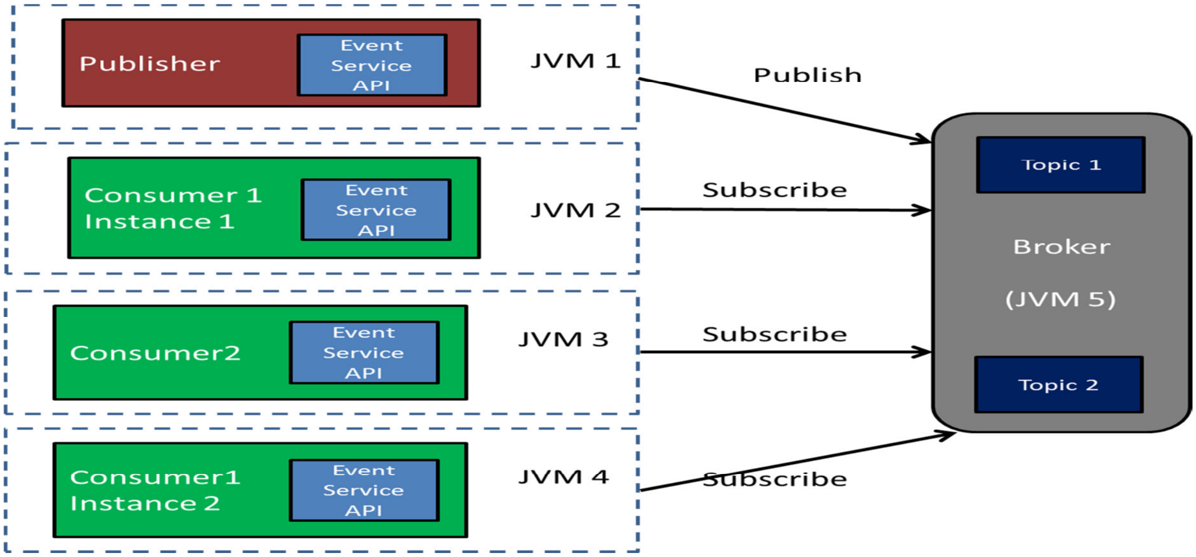
Cons

- Console based, less appealing

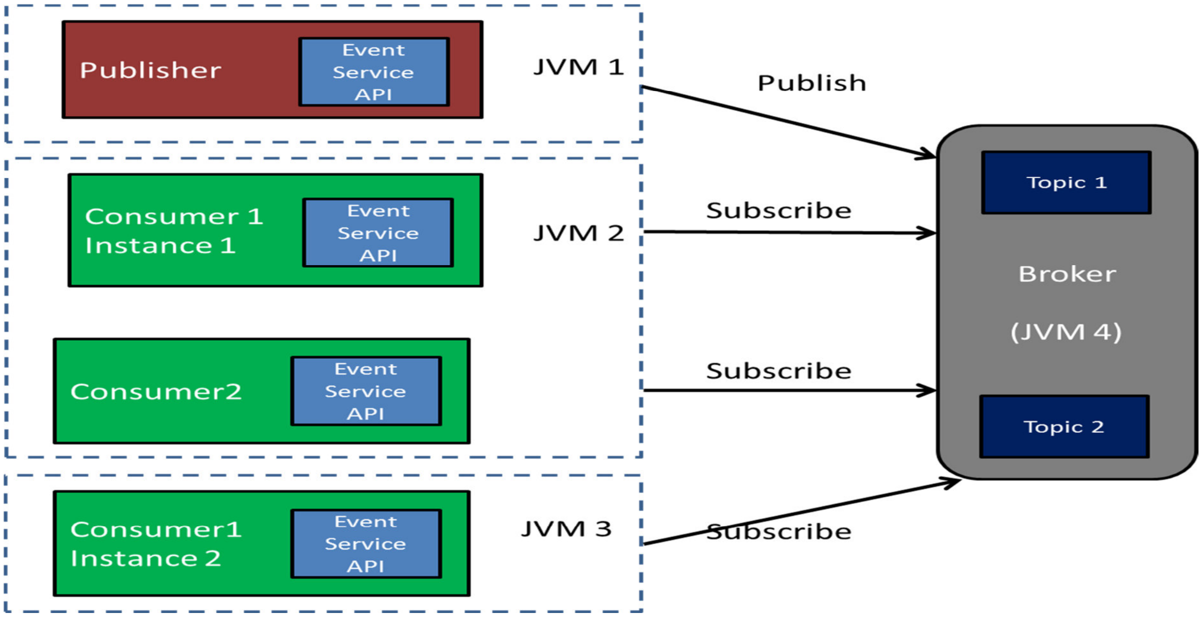
5.4. EventService API Usage

Below are some sample usage options

5.4.1. Option -1



5.4.2. Option – II



Publisher and consumer could also be in the same JVM

5.5. Subscription Cache & Recovery

Prototype implementation of EventService API would implement an in-memory cache of subscriptions (topic, queue and callback mapping). On event of failure of subscriber component, cache of topic and asynchronous listeners mapping maintained by the EventService API code would be destroyed.

On recovery, subscriber instance would have to re-subscribe to the topic. HornetQ might also require some cleanup as well. Production code would require an appropriate solution for the TMT context to handle these scenarios

5.6. Event Messaging Structure

The API would internally use HashMap for event message communication. Publishers would pass Event objects to the API which would do the transformation to HashMap. Registered Callbacks would receive the Event objects after transformation from HashMap

5.7. ES-1008

ES-1008 is dropped from the Prototype requirements (after discussion with Kim)

5.8. Priority 3 Requirements

Prototype solution would focus only on priority 1 & 2 requirements. Depending upon the availability of time, nice to have requirements would be considered.

5.9. NFRs

There would be various Non-Functional Requirements applicable for a production state EventService API. These NFRs are not in the prototype scope

5.9.1. Security

1. Secure communication between Publishers/Subscribers and Broker

2. Secured access to EventService and Broker

For the prototype, these NFRs are not considered and are out of scope. Any subscriber or publisher can subscribe or publish message to the broker using the EventService API without any authentication & authorization and secure communication

5.9.2. Performance

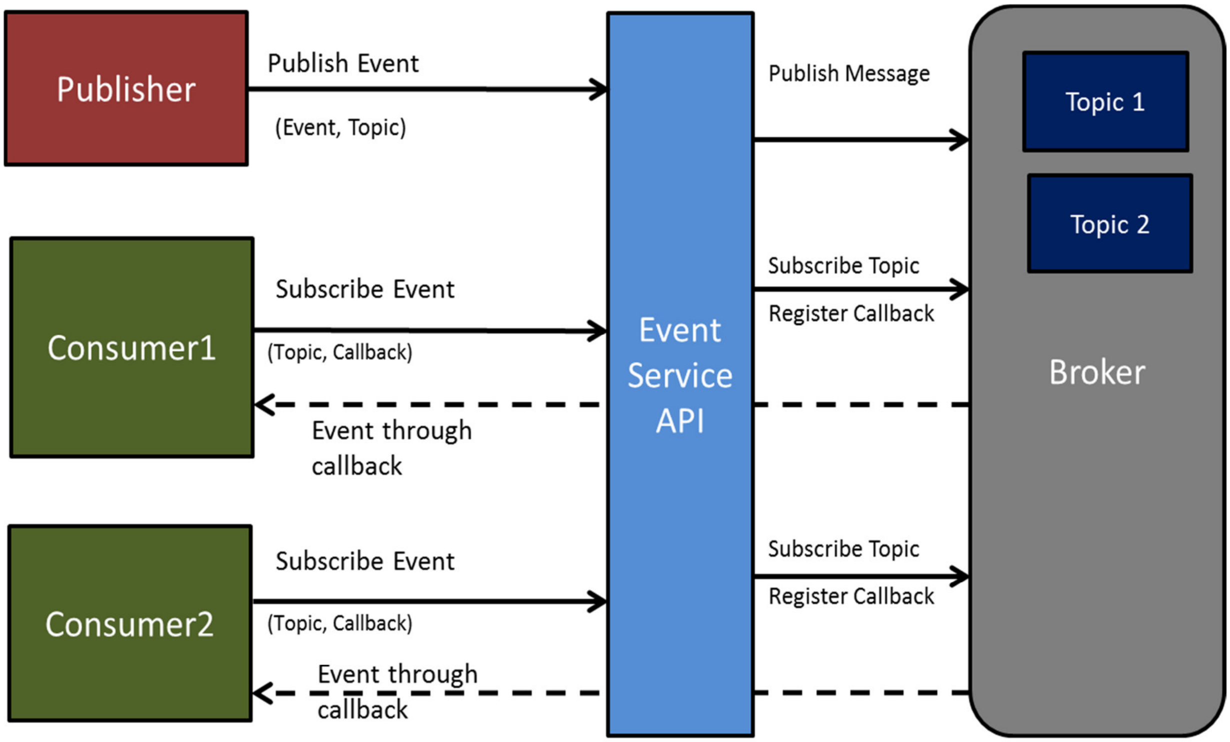
EventService API is a library to publish and subscribe events. Any performance testing of API/Broker is considered out of scope for the prototype. However, prototype would use all the broker configurations from Phase 2

5.9.3. Scalability

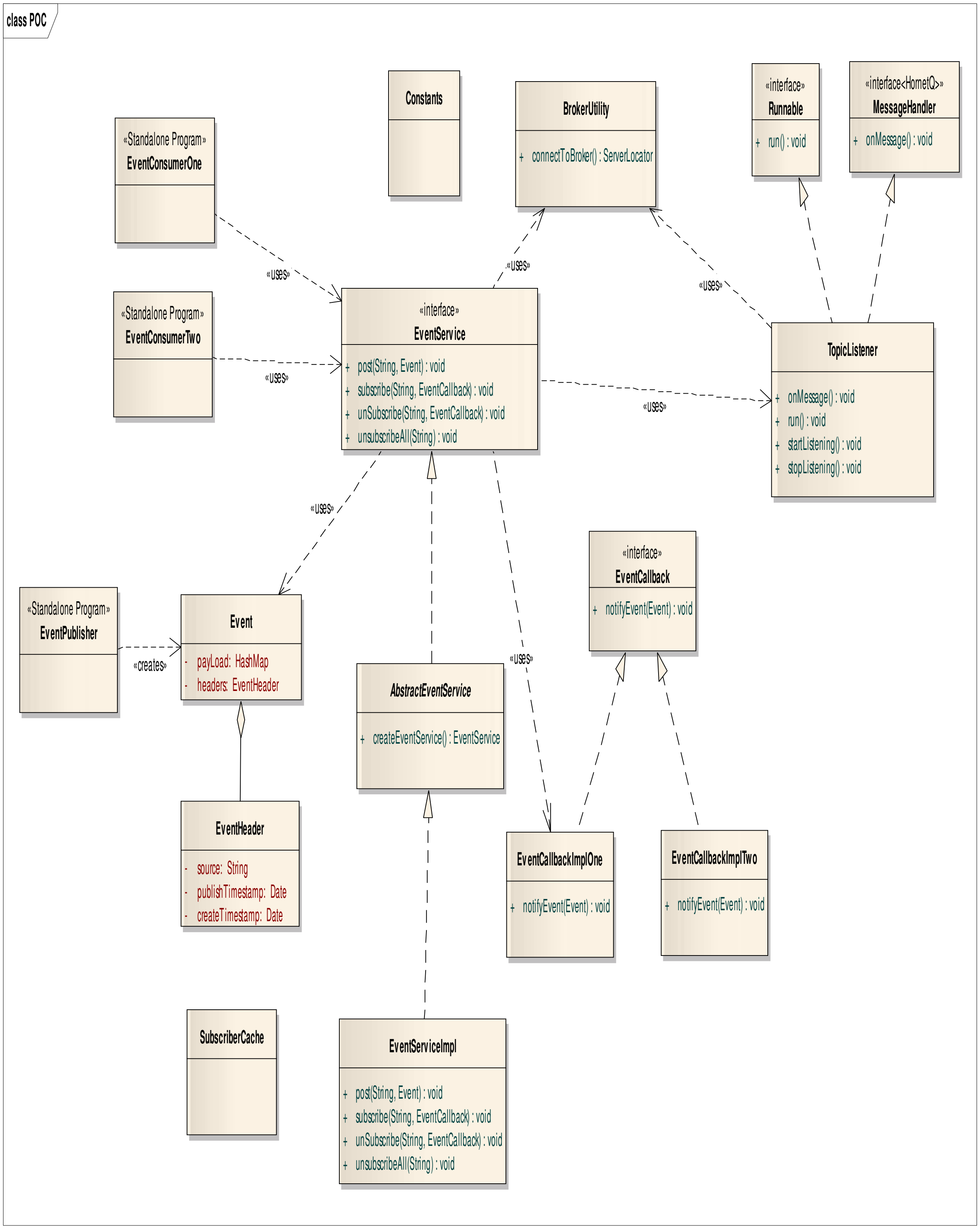
To achieve the required scalability at broker end, it might require cluster environment. For the prototype, a non-clustered single instance of broker would be used

6. Event Service Prototype

6.1. Overview



6.2. *Class Design*

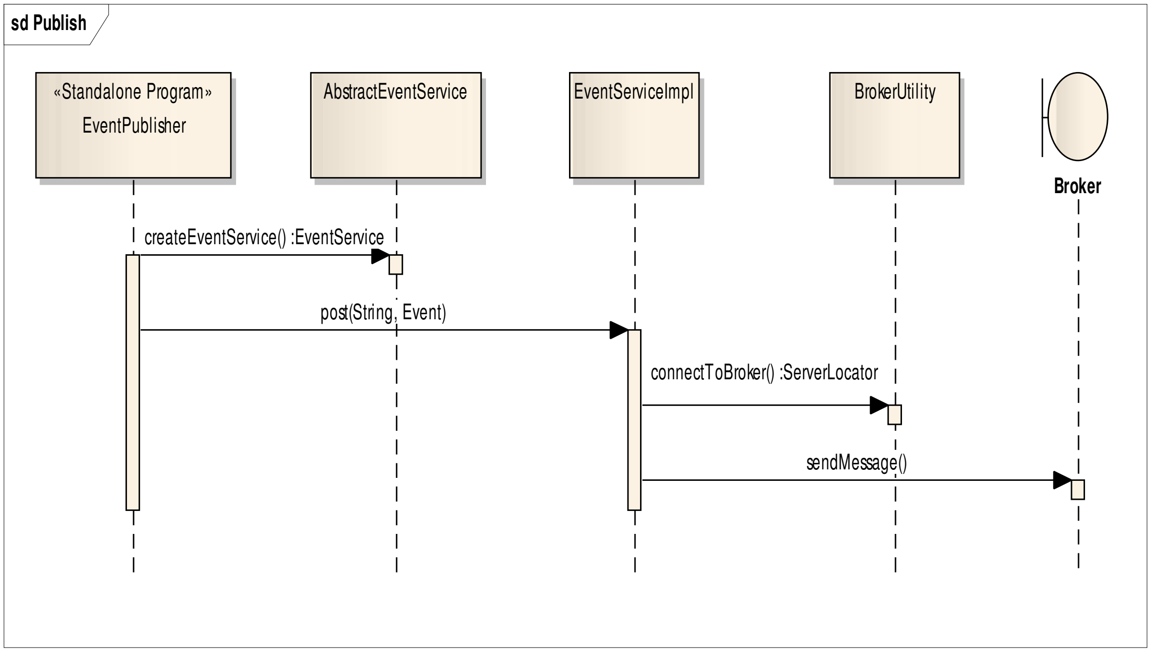


Class	Method	Description
Event	None	Represents an Event. Event object is passed to Event Service API. It is composed of payload (HashMap) and EventHeader objects
EventHeader		Represents the message headers passed along with the payload
Constants	None	Holds constants used by the Event Service API implementation
EventService		Event Service API used by all publishers and consumers to publish messages and subscribe to topics
	post()	It accepts the FQN topic and the Event to the published to the topic. It connects to the broker using Core API and publishes event. The Event object is transformed to HashMap before publishing to topic
	Subscribe()	Any subscriber instance desiring to subscribe to a topic uses the API passing the FQN Topic name and the Callback to be used for receiving the messages.
	UnSubscribe()	Subscribers can unsubscribe to a topic by passing the FQN topic name and callback used for receiving the message.
	UnSubscribeAll()	It unregisters all the callbacks for the given topic and subscriber
AbstractEventService		Provides an instance of the Event Service
	createEventService()	Factory method to create event service instances
EventServiceImpl		Implementation class for EventService API
EventCallback		Callback interface used by consumers to receive messages
	notifyEvent()	Callback method invoked by async message handler on receipt of message

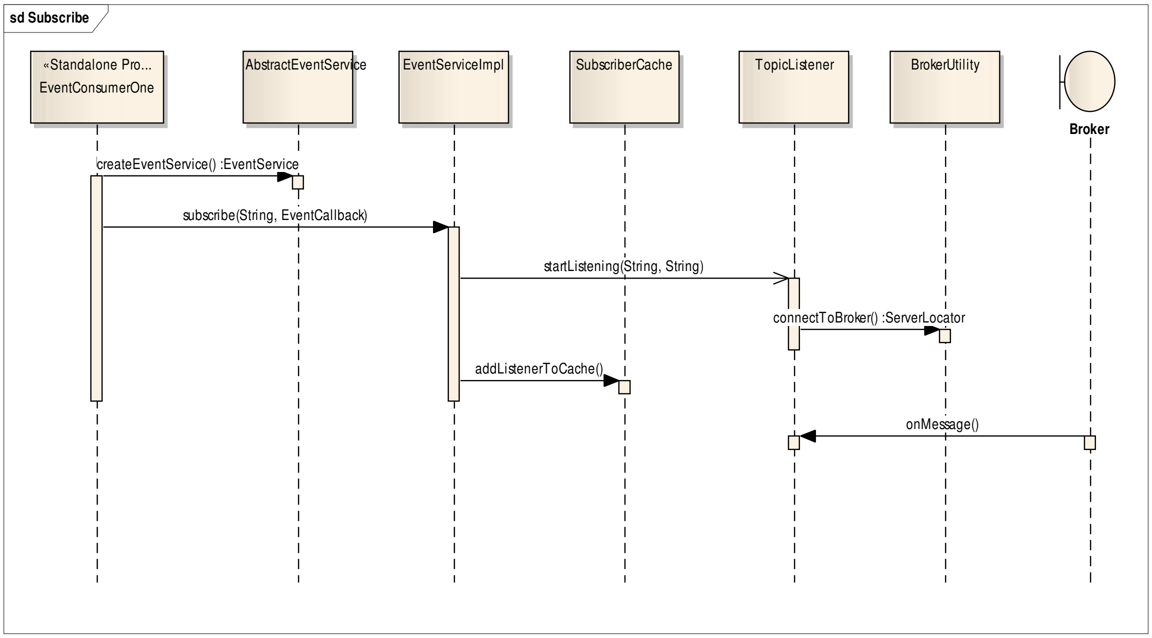
EventCallbackImplOne EventCallbackImplTwo		Sample implementations for EventCallback interface
EventConsumerOne EventConsumerTwo EventPublisher		Standalone Consumer programs.
BrokerUtility		Utility class to connect to the broker instance
TopicListener		Responsible for spawning new thread and registering/un-registering async message handlers , create & delete queues and receive message asynchronously on behalf of subscriber instance for the given FQN Topic
	startListening()	Creates Client Consumer, registers the message listener and spawns a new thread waiting for messages
	stopListening()	Triggers the active thread to stop and close the consumer session
SubscriberCache		Singleton class that holds cache of topic, queue and callback mappings
Configuration.properties		All Broker configurations are provided here

6.3. Sequence Diagram

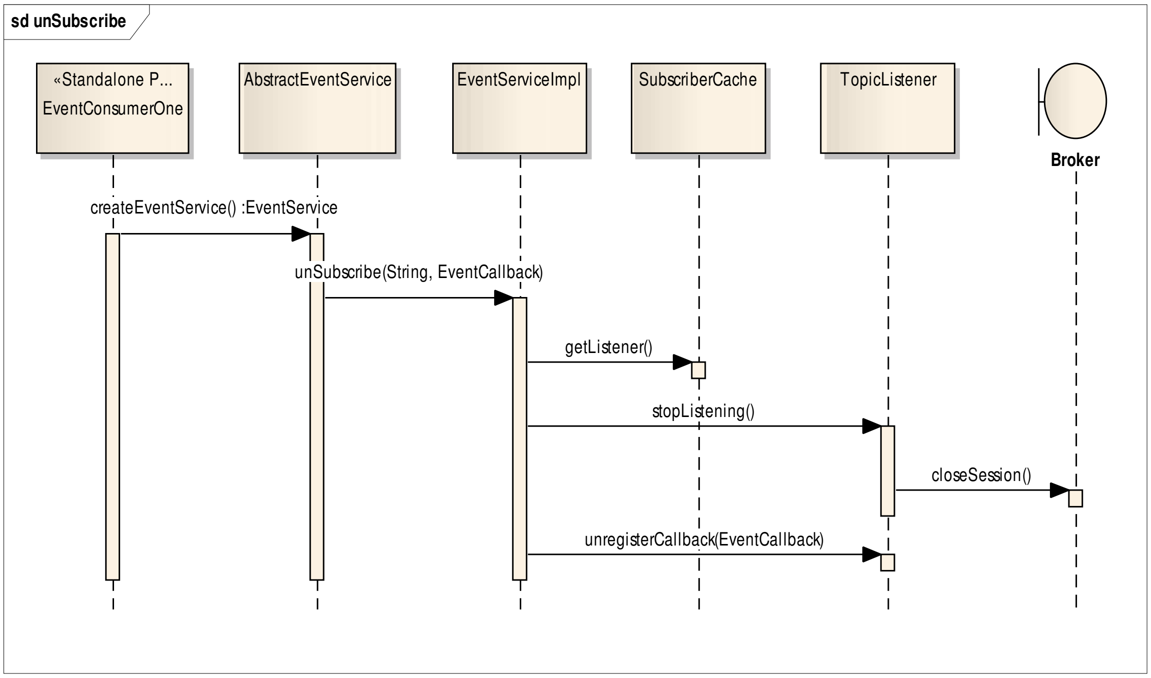
6.3.1. Post



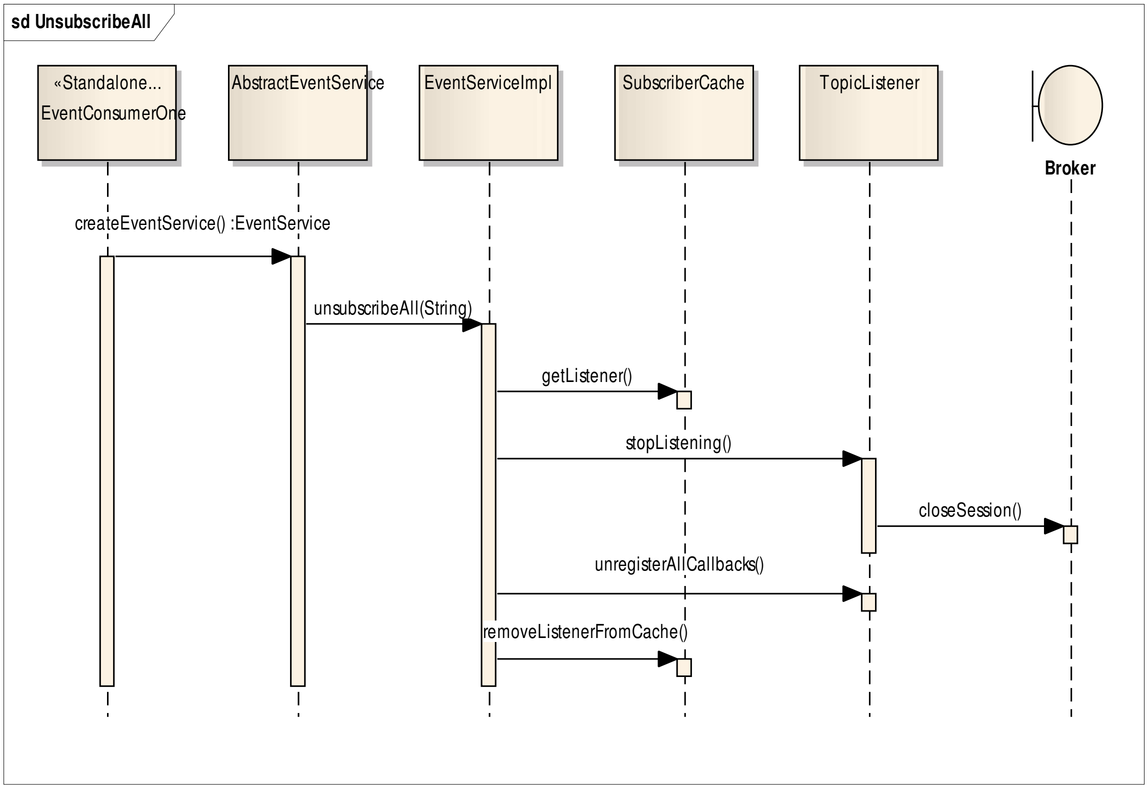
6.3.2. Subscribe



6.3.3. UnSubscribe



6.3.4. UnsubscribeAll



6.4. Development & Build

1. Ant would be used to package the code and build.
2. Eclipse IDE would be used for development of the prototype
3. JUnit would be used for unit testing the prototype code
4. Code Package Structure: org.tmt.csw.*

6.5. Deployment & Usage

1. The existing shell script would be modified to include required environment variables. Script could be executed from any location on the machine

2. EventService API would be available as a jar along with the required HornetQ client libraries

3. Two Consumers and One Publisher standalone programs would be packaged along with the EventService API jar

4. The entire prototype including standalone programs would be packaged as a zip file. On unzip, extract contents into a location and run scripts to execute the programs

6.6. Traceability Matrix

6.6.1. Priority 1 and 2 requirements

#	Requirement	Priority	Section	Remark
1	ES-1000	1	4.3 Class Design	HornetQ Core API would be used for developed
2	ES-1002	1	4.3 Class Design	Configuration.properties
3	ES-1004	1	4.3 Class Design	Configuration.properties
4	ES-1006	1	4.3 Class Design	Configuration.properties
5	ES-1008	2	4.2 Key Decisions	Removed from Prototype
5	ES-1012	1	4.5 Deployment & Usage	
7	ES-1013	2		Initial Assessment indicates packaging is not required
8	ES-1014	1	4.3 Class Design	
9	ES-1018	1	4.3 Class Design	Class based callbacks
10	ES-1020	1	4.3 Class Design 4.5 Deployment & Usage	Instead of demo web applications, standalone programs would be

				bundled with the EventService API jar
11	ES-1022	1	4.3 Class Design	org.tmt.mobie.filter is used as FQN Topic for the demo
12	ES-1024	1	4.3 Class Design	HashMap would be used as Key Value pair data structure
13	ES-1027	1	4.2 Key Decisions 4.3 Class Design	
14	ES-1030	1	4.4 Development & Build	JUnit would be used for unit testing the API

6.6.2. Priority 3 requirements

#	Requirement	Priority		Remark
1	ES-1010	3		Would be considered if time permits
2	ES-1016	3		Would be considered if time permits
3	ES-1026	3		Would be considered if time permits
4	ES-1028	3	4.3 Class Design	Asynchronous message listeners along with callbacks implemented in the prototype

6.7. Development Tasks

#	Task	Requirement
1	Post()	ES-1000, ES-1014, ES-1018, ES-1022, ES-1024, ES-1027, ES-1028
2	Subscribe()	ES-1000, ES-1014, ES-1018, ES-1022, ES-1024, ES-1027, ES-1028
3	UnSubcribe()	ES-1000, ES-1014, ES-1018, ES-1022, ES-1024, ES-1027, ES-1028
4	UnsubscribeAll()	ES-1000, ES-1014, ES-1018, ES-1022, ES-1024, ES-1027, ES-1028
5	Consumer One	ES-1020
6	Consumer Two	ES-1020
7	Publisher	ES-1020
8	HornetQ Configuration & Broker Script	ES-1002, ES-1012, ES-1013, ES-1004, ES-1006, ES-1012
9	Build Script	ES-1020

10	JUnits	ES-1030

6.8. Test Scenarios

Documents few functional prototype test scenarios

6.8.1. Post Message with no consumers

Message should not be available to consumers on subscription

6.8.2. Post Message with one consumer and one callback

Consumer should receive the message through callback

6.8.3. Post Message with multiple consumers (each with one callback)

All consumers should receive the message through their callbacks

6.8.4. Post Message with one consumer and multiple callbacks

All the consumer callbacks should receive the message

6.8.5. Post Message with multiple consumers and multiple callbacks

All the consumer callbacks should receive the message

6.8.6. Unsubscribe a Callback for a consumer and Post Message

The unregistered callback should not receive the message

6.8.7. Unsubscribe all consumer callbacks and Post Message

Consumer should not receive message