Publish Subscribe Software **Requirements Specification**

Group 2

Version:	2.0
Print Date:	2019/02/24
Release Date:	
Release State:	Final
Approval State:	Approved
Approved by:	Yansheng Xie, Jiachen Luo, Shunxi Pang, Mingcong Zhou, Shulan Yang
Prepared by:	Yansheng Xie, Jiachen Luo, Shunxi Pang, Mingcong Zhou, Shulan Yang
Reviewed by:	Yansheng Xie, Jiachen Luo, Shunxi Pang, Mingcong Zhou, Shulan Yang
Path Name:	
File Name:	
Document No:	

Document Change Control

Version	Date	Authors	Summary of Changes
1.0	Feb 10	Yansheng Xie	Initial Write
2.0	Feb 24	Mingcong Zhou	Final write

Document Sign-Off

Name (Position)	Signature	Date
Shulan Yang	Shulan Yang	Feb. 3
Shunxin Pang	Shunxin Pang	Feb. 7
Jiachen Luo	Jiachen Luo	Feb. 15
Yansheng Xie, Mingcong Zhou	Yansheng Xie, Mingcong Zhou	Feb. 24

Content

1 Introduction

1.1 Purpose

This document details the requirements of a subscribe-publish system to allow a subscription-based one-way communication application. The document gives five use cases of the system, and elaborates their corresponding sequence diagram and communication diagram to give both a dynamic view of how the use case achieves its purpose. The business scenario, domain model, and all actors are also further explained, as well as all the relevant non-functional requirements.

1.2 Overview

This project aims to develop a publish-subscribe (pub/sub) system that enables the one-way communication of news or other messages. With the system, the subscribers can subscribe to a channel, which represents an abstraction of a topic in which the subscriber is interested in. At the same time, the administrator can determine whether to block or unblock a subscriber from a specific channel to stop that person from receiving any messages on that channel. the publishers can generate events (the message) and use his or her publisher strategy to decide which channel or channels to post to, or totally rely on the strategy to both generate the message itself and specify a channel or channels. Then, the Channel Message Dispatcher will post the messages on all specified channel (channels), and each channel will alert all unblocked subscribers to send the message. Each subscriber will respond to different messages depending on the subscriber's corresponding state.

A number of design patterns will be used to facilitate all the functionalities above, including the Singleton, the State, the Strategy, the Proxy, and the Observer design patterns. A list of non-functional requirements are also considered within the context of the pub/sub system to ensure its stability, security, and efficiency.

1.3 References

JAIN-SIP:	https://jain-sip.dev.java.net
SIP:	https://sip-communicator.dev.java.net/
Eclipse:	http://www.eclipse.org/downloads/index.php
WSDP:	http://java.sun.com/webservices/jwsdp/index.jsp
JAXR:	http://java.sun.com/webservices/jaxr/index.jsp
JAXRPC:	http://java.sun.com/webservices/jaxrpc/index.jsp
WSIF:	http://ws.apache.org/wsif/

ebXML: http://www.ebxml.org/
UDDI: http://www.uddi.org/
WSDL: http://www.w3schools.com/wsdl/default.asp
Java Server: http://java.sun.com/j2ee/1.4/download.html#sdk
GANTT: http://wiki.phprojekt.com/index.php/Gantt-diagram

2 Business Scenario Model

2.1 Actors

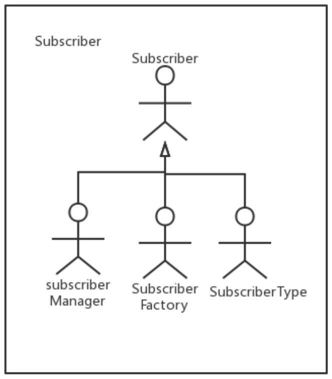
2.1.1 Overview

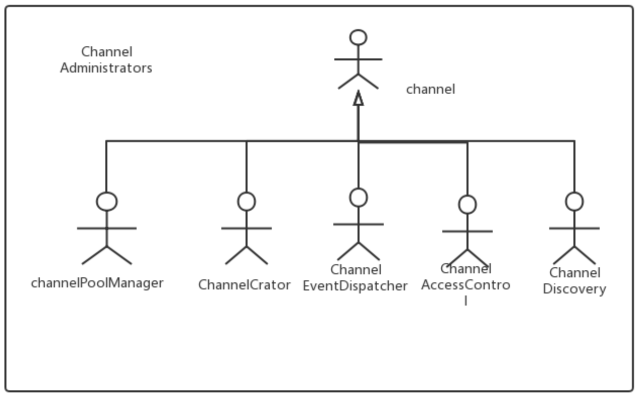
The actor in our system includes publisher of the event, the subscriber of the channel, and the administrators of the channels. The publishers generate events and post them on a channel, where many subscribers have access to. Subscribers subscribe to specific channels, in order to get access to the events posted in those channels A channel knows its subscriber and notifies them when a new event is posted on the channel. The administrators ensures the functionality of the whole system by managing channels, subscribers, events, and their relationships.

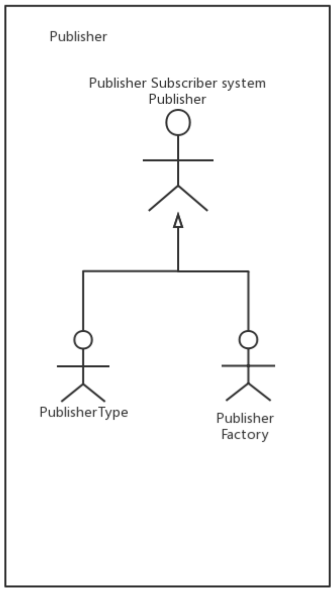
2.1.2 Actor Diagram

The figure below represents the actors in our system. Based on their interactions with the system, the actors are characterized into four general groups: a) Publisher b) Subscriber c) Channel d) Event.

Fig. 2.1 Actor Diagram


Subscriber
Subscriber
subscriber Manager
Subscriber Factory
SubscriberType


Channel Administrators
channel
channelPoolManager
ChannelCrator
Channel EventDispatcher
Channel AccessControl
Channel Discovery


Publisher
Publisher Subscriber system Publisher
PublisherType
Publisher Factory

2.1.3 Actor Definitions

Channel Administrator

Description	Channel is an entity that denotes an abstraction of a communication medium. It maintains a list of subscribers, and a queue of events. Once an event is added to the queue, the channel notifies its subscribers. There may be more than one channels in the system.
Aliases	ChannelDiscovery, ChannelAccessControl Channe,;EventDispatcher,ChannelCreator, ChannelPoolManager
Inherits	None
Actor Type	Active Also has passive roles
Contact Person	
Contact Details	

Subscriber

Description	Subscriber object handles events published on a channel, and has strong association with Channel object and Pub Sub Entity object. Subscriber object subscribe or unsubscribe channel, alert, and set the state.
Aliases	SubscriberType,SubscriberManager,SubscriberFactory
Inherits	None
Actor Type	Passive Actor
Contact Person	
Contact Details	

Publisher

Description	Publisher Object publishes events on channel or channels. The publisher object will send the notice of publishing to a strategy class.
Aliases	PublisherType,PublisherFactory
Inherits	None.
Actor Type	Passive Actor
Contact Person	
Contact Details	

2.2 Use Case 1 - Use Case Descriptions

This section documents is Publisher publishes an event.

2.2.1 Publishing an event

In this business scenario, Agent A publishes an event through Agent B. Agent A can either tell Agent B what to publish by invoke Event Factory or do nothing (rely on Agent B). Agent B will then post the event to Agent C. Agent C will determine which channel or channels to post.

Description:
This scenario pertains to the publication of an event by the Publisher (Agent A). This process is triggered by the Publisher (Agent A), and the Publisher can either:

a) specifying an event through Event Factory, and send it to Strategy (Agent B), or
b) send it without specifying (relying on Strategy)

If Strategy (Agent B) receives an event from a Publisher (Agent A), it will forward it to Channel Event Dispatcher (Agent C) directly. Otherwise, Strategy (Agent B) will generate an Event through Event Factory and transmit it to the Channel Event Dispatcher (Agent C).
When Channel Event Dispatcher (Agent C) receives an event, it will look through the list of channels and determine which to post. If the Channel does not exist, it will be created, then post.

Actors:
The actors associated with this business scenario are as follows:

1. Publisher (Agent A)
2. Strategy (Agent B)
3. Event Factory
4. Channel Event Dispatcher (Agent C)

The details of the above mentioned actors can be found in Section 2.1.3 (Actor Definitions).

Preconditions:
Before this scenario, the precondition of the use case is that the channel exist

1. The publisher and the event must exist
2. The event is about to be published does not exist then it is created.
3. The publishing occurs via a strategy each publisher is associated with.
4. The strategy refers to channels by name.

Postconditions:
The postcondition is that the subscriber is added in the list of subscribers for the given channel.

1. There should be an event in the queue of the specified channel, and the channel exists.

Scenario Text:

1. Publisher Publishes a Strategy
   1.1. Specifying which event to publish by calling Event Factory, then publish to strategy.
   1.2. Publish to strategy without specifying.
2. Strategy Post an Event to Channel Event Dispatcher
   2.1. If channel exist, publish event to channel.
   2.2. Else send the request of create channel to Channel Creator, and then post it to channel.

Alternative Courses:
None.

Extends:
None.

User Interfaces:
None.

Constraints:
None.

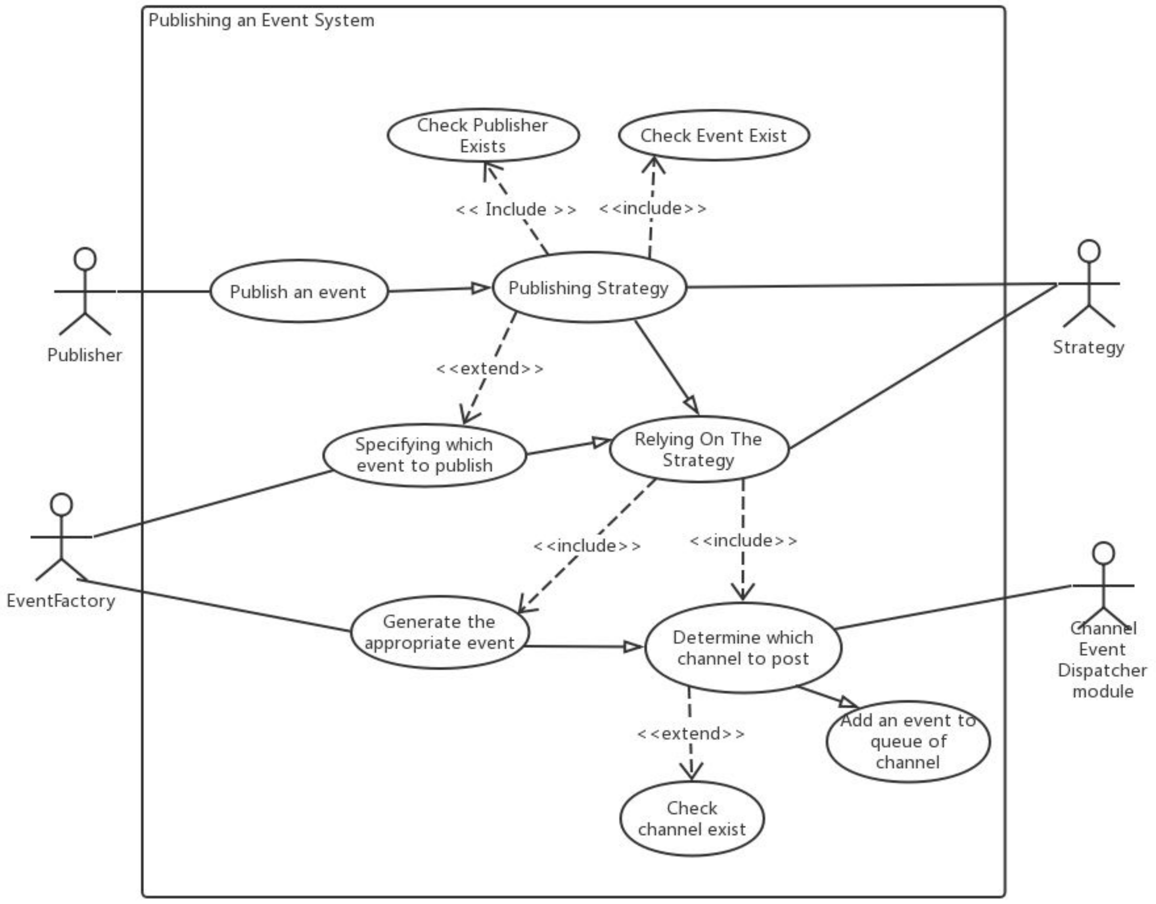
Questions:
None.

Notes:
None
Authors: MingCong, Zhou

Source Documents:

2.3 Use Case 1 - Use Case Diagrams

This section presents the business scenarios of the subject area in a graphical form.



2.4 Use Case 2 - Use Case Descriptions

This section documents is Channel notifying subscribers.

2.4.1 Channel notifying subscribers of event been published

In this system scenario, the Channel uses the system to notify subscribers by checking blocking list from Channel Access Control.

Description:
This scenario pertains to the situation where an event has just been posted to a channel and the subscribers of the channel need to be notified. The process starts off by the Channel has been posted an newly event. This scenario requires that Channel has a number of subscribers and Channel Access Control can aware of whether the subscriber is blocked in Channel. A subscriber is notified of the newly posted event only if it is not blocked on this channel.
Actors:
The actors associated with this scenario are as follows:

1. Channel
2. Channel Access Control
3. Concrete Subscriber A
4. Event ID Maker
5. Channel Event Dispatcher
6. Channel Pool Manager

The details of the above mentioned actors can be found in Section 2.1.3 (Actor Definitions).
Preconditions:
Before this scenario, the precondition of this use case is that the event must be posted on a channel, and the channel has subscribers.
Postconditions:
The postcondition is that all subscribers on this channel which are not blocked, are notified.
Scenario Text:

1. Find Channel With Newly Posted Event
   - 1.1. Supply Newly Event ID
   - 1.2. Find Channel According to Event ID
2. Check For the Subscribers in this channel
   - 2.1. Supply list of subscribers
3. Check For Each Subscriber
   - 3.1. Supply Blocking List Details
   - 3.2. Find Subscriber’s Name in Blocking List
   - 3.3. Return Response from Control Access Control
4. Notifying Subscribers
   - 4.1. Get the Response from Control Access COntrol
   - 4.2. Channel Notify Subscribers

Alternative Courses:
None.

Extends:
None.

User Interfaces:
None.

Constraints:
None.

Questions:
None.

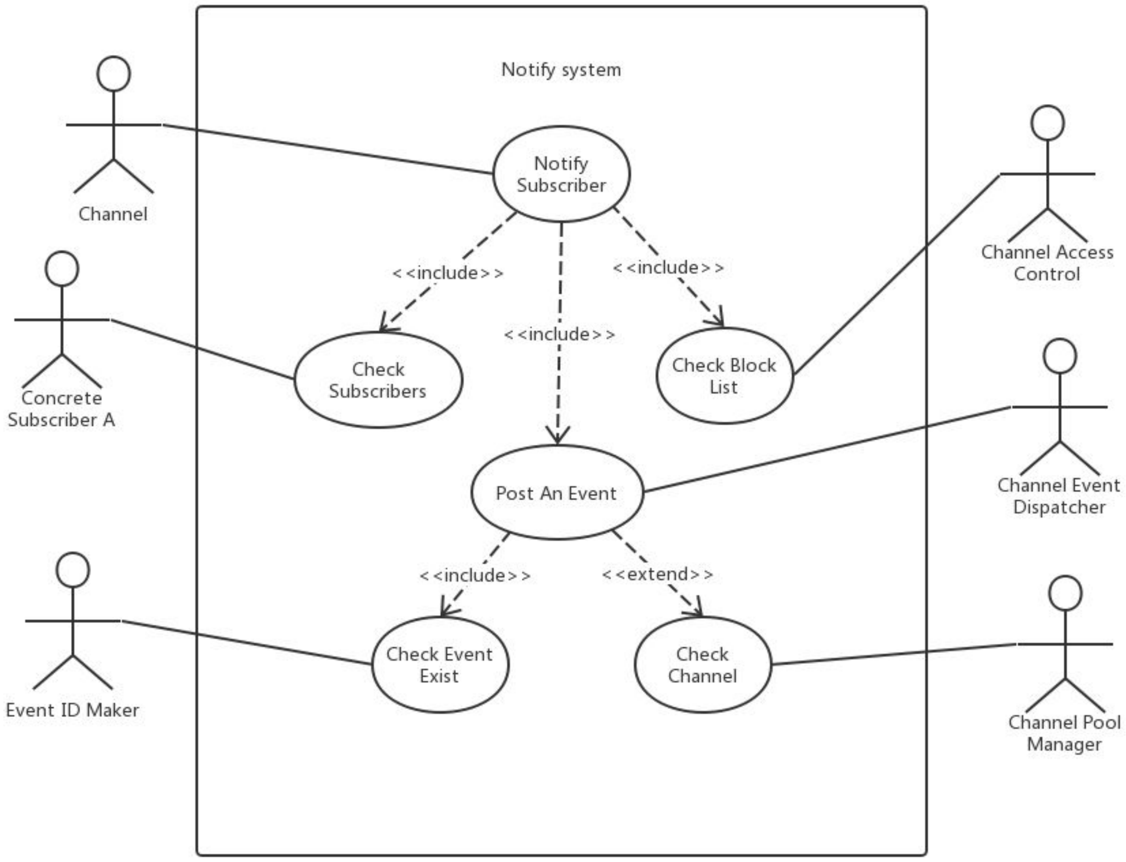
Notes:
Every effort has been made not to include details that obstruct a high level view of the scenario sequence. Details are provided in the diagrams below for this use case.
No alternative have been listed because of the pre-conditions stating that servers are properly functioning and that the event posted on the channel and subscribers must be exist.

Authors:

Source Documents:

2.5 Use Case 2 - Use Case Diagrams

This section presents the business scenarios of the subject area in a graphical form.



2.6 Use Case 3 - Use Case Descriptions

This section documents is Subscriber handling an event.

2.6.1 Subscriber handling an event (respond to an event)

In this business scenario, the channel notifies all its unblocked subscribers, and a subscriber responds to the messages depending on his or her corresponding state.

Description:
This scenario explains how a subscriber handles an event. The process begins with a channel notifying all its unblocked subscribers by using the alert method of subscriber objects. A subscriber, after being called upon, receives the message and the channel name. Then the subscriber passes both information to his or her corresponding state by using the the method handleEvent. At the end, it is the State that determines what actions to take to handle/respond to the event from a channel.

Actors:
The actors associated with this business scenario are as follows:

1. **Subscriber (of a specific channel in which he or she is not blocked)**

The details of the above mentioned actors can be found in Actor Definitions.

Preconditions:
Before this scenario can be performed:

1. **The subscriber exists.**
2. **The subscriber has a state.**
3. **The subscriber has subscribed a channel in which the event is posted and is in the channel's subscribers list (unblocked).**

Postconditions:

1. **The subscriber handles the event based on the logic specified by its state.**

Scenario Text:

1. **The subscriber receives the event from a subscribed channel through being alerted by that channel.**
2. **The subscriber passes the message and the channel name to the State.**
3. **The State handles the event.**

Alternative Courses:
None.

Extends:
None.

User Interfaces:
None.

Constraints:
None.

Questions:
None.

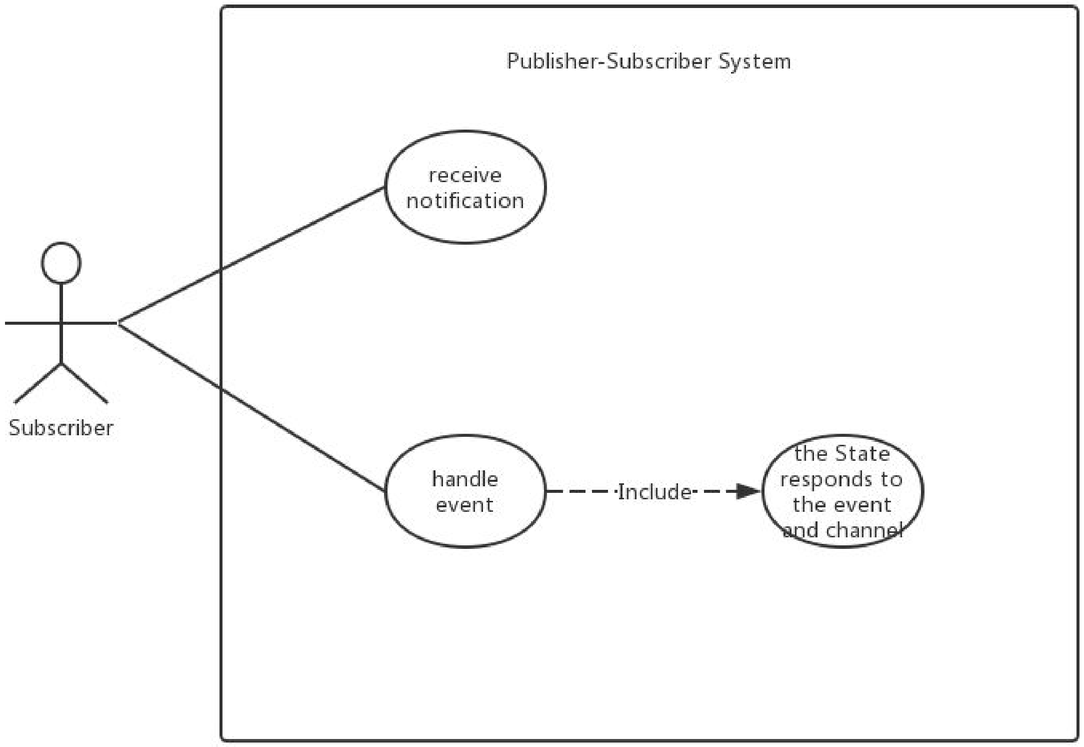
Notes:
None.

Authors:
Shunxin Pang

Source Documents:

2.7 Use Case 3 - Use Case Diagrams

This section presents the business scenarios of the subject area in a graphical form.


Publisher-Subscriber System
receive notification
Subscriber
handle event
Include
the State responds to the event and channel

2.8 Use Case 4 - Use Case Descriptions

This section document is Controlling Access to a Channel.

2.8.1 Controlling Access to a Channel

In this system scenario, administration server use Channel Access Control system to decide whether block or unblock a subscriber in specific channel only if the subscriber exist in specific channel and check through by Channel Pool Manager system.

Description:
Channel Access Control Module provides services to block or unblock a subscriber, when decide to block a subscriber, then add the subscriber to the black list; otherwise, to unblock a subscriber, remove from the black list. Mean while, Channel Pool Manager needs provides all available channel list and subscribers list before performing the blocking command.
Actors:
The actors associated with this scenario are as follows:

1. **Channel Pool Manager**
2. **Channel Access Control**

The details of the above-mentioned actors can be found in Section 2.1.3 (Actor Definitions).
Preconditions:
Before this scenario, the precondition of this use case is that the channel exists.
Postconditions:
The postconditions are that all subscribers must added in list of blocked subscribers of the specified channel (blocking case), or removed from the list blocked subscribers of the specified channel (unblocking case).
Scenario Text:

1. **Find available channel list and subscriber list**
2. **find the channel that has the subscriber to block or unblock**
   2.1. Find the subscribers in the channel
3. **If the subscriber does not exist in the channel, then return null**
4. **If the subscriber does exist, then return true**
   4.1. Block the subscriber (add to black list) (block case)
   4.2. Unblock the subscriber (remove from the list) (unblock case)

Alternative Courses:
None.

Extends:
None.

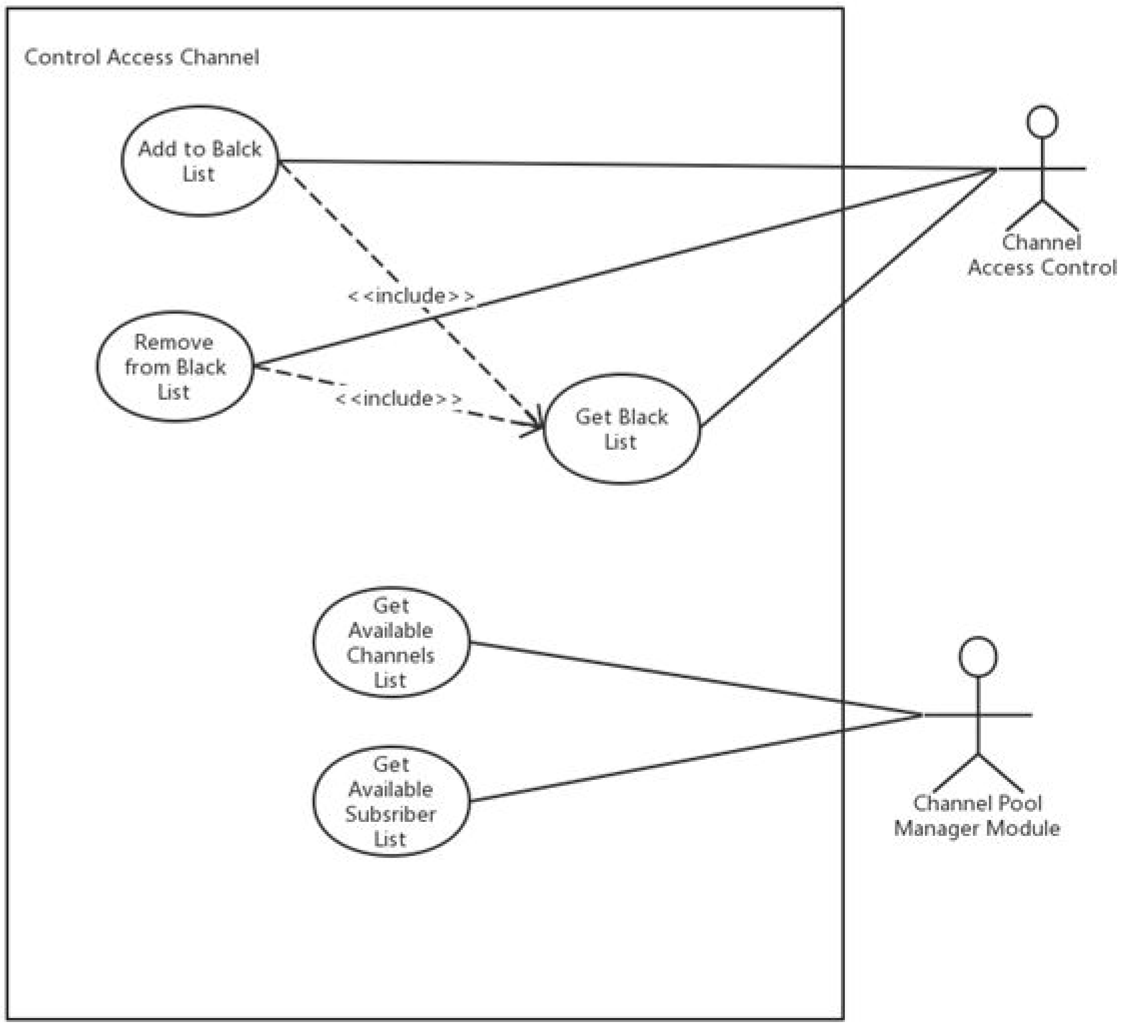
User Interfaces:
None.

Constraints:
Must return null or true when looking for the subscriber in the channel.
When the subscriber exist, must added to black list or remove from the black list.

Questions:
None.

2.9 Use Case 4 - Use Case Diagrams

This section presents the business scenarios of the subject area in a graphical form.



2.10 Use Case 5 - Use Case Descriptions

This section documents is Subscriber subscribe the channel.

2.10.1 find the channel and subscribing the channel

In this Subscribing scenario, one Agent A (subscriber) uses the system to subscribe the channel by another Agent B(Channel pool manager), Agent C(subscription Manager) control add subscriber to the list, the Agent D(Channel) that add the Agent A' to the list.

Description:

This scenario Agent A(subscriber) can subscribe the Agent D(Channel), Agent A should use the Agent B to find the channel and subscribe that channel through the Agent C, and add the Agent A to the list by the Agent D(channel) .

Actors:

The actors associated with this business scenario are as follows:

1. Subscriber (Agent A)
2. Channel Pool Manager (Agent B)
3. Subscription Manager (Agent C)
4. State Factory (Agent B)
5. Subscribing Channel System

The details of the above mentioned actors can be found in Section 2.1.3 (Actor Definitions).

Preconditions:

Before this scenario, the precondition of the use case is that the channel exist

Postconditions:

The postcondition is that the subscriber is added in the list of subscribers for the given channel.

Scenario Text:

1. **Get the instance from the subscriber manager**
   1.1. Send the getInstance request to the subscriber manager
   1.2. Find the instance and return to the subscriber
2. **Get the channel**
   2.1. Use the instance which we get at the first step to send the subscribe request to the subscriber manager
   2.2. Find the channel by the channel name(done by the channel pool manager)
   2.3. Send response back and return the channel to subscriber manager ,
3. **Subscribe the channel**
   3.1. Subscriber subscribe the channel which we found at the previous step
   3.2. Channel add the subscriber to the subscriber list
   3.3. Return the subscriber to the subscriber manager
4. **Reply the subscriber**
   4.1. Return true to the subscriber

Alternative Courses:

None.

Extends:
None.

User Interfaces:
None.

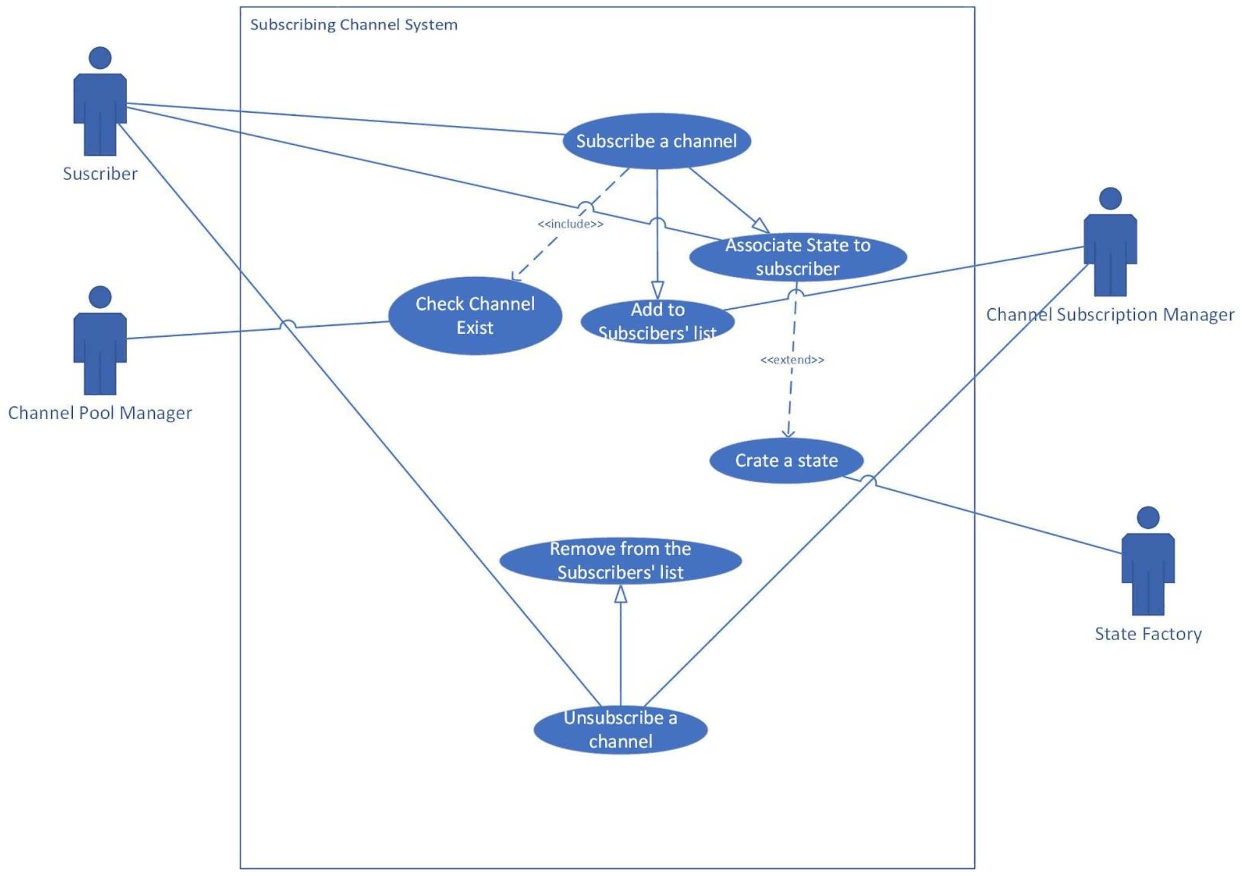
Constraints:
None.

Questions:
None.

Notes:
Every effort has been made not to include details that obstruct a high level view of the scenario sequence. Details are provided in the diagrams below for this use case.
No alternative have been listed because of the pre-conditions stating that servers are properly functioning and that the receiver party has web services registered.
Graceful Exit Use Case Note :
The graceful exit use case is behavioral functionality that our system will implement to indicate an error to the user and stop transactions midway with a priority on placing the least amount of interruption and disturbance to the user. This use case is not expanded upon in this version of the SRS to conform with the deliverable requirements. The Graceful Exit use case of course, handles many exit scenarios.
Authors:YanSheng Xie

Source Documents:

2.11 Use Case 5 - Use Case Diagrams

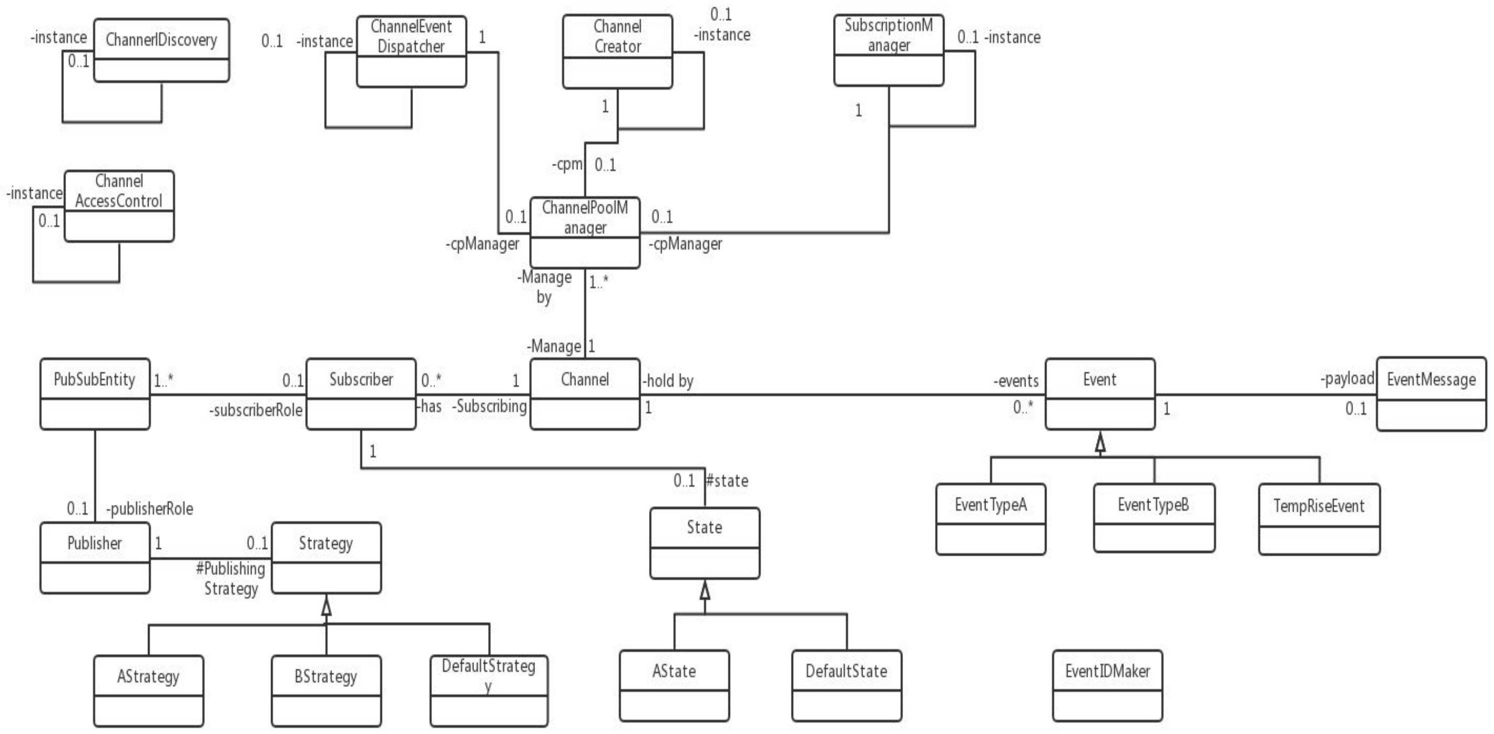


3 Domain Model

3.1 Domain Model Class Diagram

The domain model class diagram for the SIP-based Communication System appears below:

Fig. 3.1 Domain Model Class Diagram



3.2 Domain Model Class Definitions

3.2.1 Channel Access Control

Description	Class that acts as an access control module that allows for the blocking and unblocking of specific subscribers for specific channels.
Attributes	**blackList: Map<String, List<AbstractSubscriber>>**
Responsibilities	Decide whether add subscriber into blacklist (blocking case), or remove from the blacklist (unblocking case). Check the user is block or not.
Business Rules	

3.2.2 Channel Discovery

Description	Allows for the discovery of available channels for subscription from that want to subscribe to them.
Attributes	
Responsibilities	List all the channel and find the specific channel.
Business Rules	

3.2.3 Channel Event Dispatcher

Description	Class providing an interface for objects to cover their publishing needs

Attributes	**cpManager: private ChannelPoolManager**
Responsibilities	This class intend to post event on the channel.
Business Rules	

3.2.4 Channel Creator

Description	This class is responsible for creating and deleting channels it's also the only class that can do so
Attributes	**Cpm: private ChannelPoolManager**
Responsibilities	This class can do add a new channel or delete a exist channel.
Business Rules	

3.2.5 Subscription Manager

Description	Exposes the subscribe, and unsubscribe methods to the clients
Attributes	**cpManage: rprivate ChannelPoolManager**
Responsibilities	This class put the subscriber into the specific channel subscription list. Also can execute unsubscribe by removing the subscriber from the specific channel subscription list.
Business Rules	

3.2.6 Channel Pool Manager

Description	Holds the collection of AbstractChannel type entities and provides the methods for manipulating thes collections
Attributes	**Instance: private static ChannelPoolManager** **channelsMap: private Map<String, AbstractChannel>** **channelList: private List<AbstractChannel>**
Responsibilities	This class can do create a new channel, delete an exist channel, list of all the available channel, find a specific channel from existing channel list. Also can access the channel map.
Business Rules	

3.2.7 Channel

Description	This object denotes an abstraction of a communication medium. It contains a list of subscribers, and a queue of events. Channel object associates with Subscribers object and Event object and is managed by Channel Pool Manager object.
Attributes	**Abstract Channel**
Responsibilities	The Channel object publish the event, add or remove subscribers from the subscribe list, and notify the subscribers when new event is posted.
Business Rules	

3.2.8 PubSub Entity

Description	This class define a role to publisher and subscriber object.
Attributes	
Responsibilities	This class define a role to publisher and subscriber object.
Business Rules	

3.2.9 Publisher

Description	Publisher Object publishes events on channel or channels. The publisher object will send the notice of publishing to a strategy class.
Attributes	**Abstract Publisher**
Responsibilities	Publisher object can either specifying an event and then publish to a strategy object or publish without passing anything to a strategy object.
Business Rules	

3.2.10 Strategy

Description	A strategy object will receive a request of publishing event from a publisher object, and keep passing the event to the channel event dispatcher. Strategy object can be in three types, A strategy, B strategy and default strategy
Attributes	**IStrategy**
Responsibilities	The request from a publisher can either contains additional event information or have nothing. If nothing is passing by, then keep passing the request of publishing an event to channel event dispatcher. Else: create an event and then passing it.
Business Rules	

3.2.11 A Strategy

Description	One of the three strategy option.

Attributes	**IStrategy**
Responsibilities	A type of strategy that need to be passing, and the type is A.
Business Rules	

3.2.12 B Strategy

Description	One of the three strategy option.
Attributes	**IStrategy**
Responsibilities	A type of strategy that need to be passing, and the type is A.
Business Rules	

3.2.13 Default Strategy

Description	One of the three strategy option.
Attributes	**IStrategy**
Responsibilities	A type of strategy that need to be passing, and the type is A.
Business Rules	

3.2.14 Subscriber

Description	Subscriber object handles events published on a channel, and has strong association with Channel object and Pub Sub Entity object. Subscriber object subscribe or unsubscribe channel, alert, and set the state.
Attributes	**Abstract Subscriber**

Responsibilities	Add or remove channel name from the channel list, and set the state for subscribers. When the event is published, alert subscriber that channel has the new event.
Business Rules	

3.2.15 State

Description	State object is allowed to handle the event and is associated with object Subscriber.
Attributes	**IState**
Responsibilities	By comparing the state of subscriber, State object will handle the event.
Business Rules	

3.2.16 A State

Description	AState object is a specified type of state. It inherit IState object and the Event will be handled by checking for the state of subscriber
Attributes	
Responsibilities	Store all information about type AState and A State also handle the event according to the state of subscriber
Business Rules	

3.2.17 Default State

Description	Default State object is general type of state which including all types of state and AState as well. It inherit IState object and contains the general information about default state. Default State also can handle the event regarding to the state of subscriber
Attributes	

Responsibilities	Store all information about default state and handle the event by considering the state of subscriber
Business Rules	

3.2.18 Event

Description	Event object denotes a piece of information posted on a channel. It contains type of event, publisher of event, and payload system. Event object
Attributes	**Abstract Event**
Responsibilities	Store the information about type and publisher of event, and Event also contains the information about payload.
Business Rules	

3.2.19 Event Type A

Description	Event Type A object is a specified type of event. It inherit Event object and contains information about event publisher ID and payload system.
Attributes	
Responsibilities	Store the information about Event Type A, and Event Type A will provide the type, publisher and payload message if needed.
Business Rules	

3.2.20 Event Type B

Description	Event Type B object is a specified type of event. It inherit Event object and contains information about event publisher ID and payload system.
Attributes	
Responsibilities	Store the information about Event Type B, and Event Type B will provide the type, publisher and payload message if needed.

Business Rules	

3.2.21 Temp Rise Event

Description	Temp Rise Event object is a specified type of event. It inherit Event object and contains information about event type, event publisher ID and payload system.
Attributes	
Responsibilities	Store the information about Temp Rise Event, and Temp Rise Event will provide the type, publisher and payload message if needed.
Business Rules	

3.2.22 Event ID Maker

Description	Event ID Maker object produces the ID message for every newly posted event. Every newly posted event has a different ID.
Attributes	
Responsibilities	Supply ID message for newly posted event, and get the information about ID message from Event ID Maker object.
Business Rules	

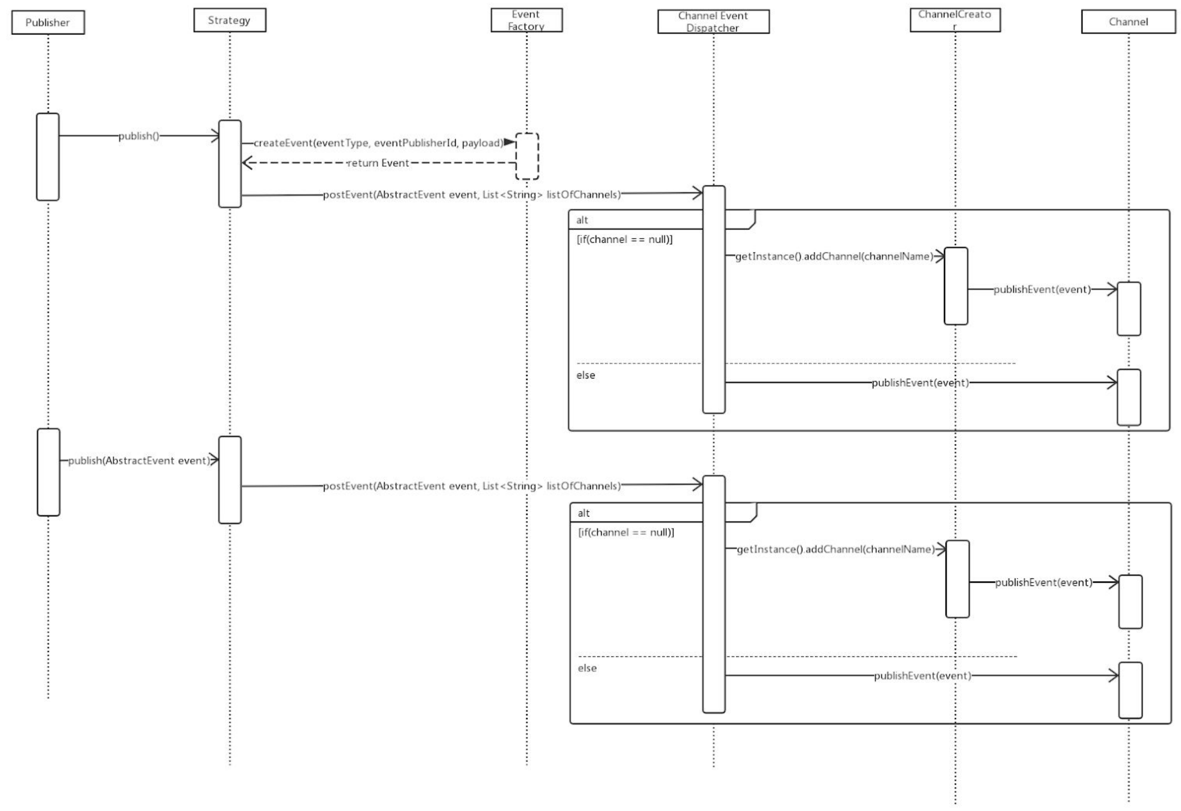
3.2.23 Event Message

Description	Event Message object receives header and body of event and supply a complete event message to Event object.
Attributes	
Responsibilities	Produce a new event message and supply the information about header and body of event if needed.
Business Rules	

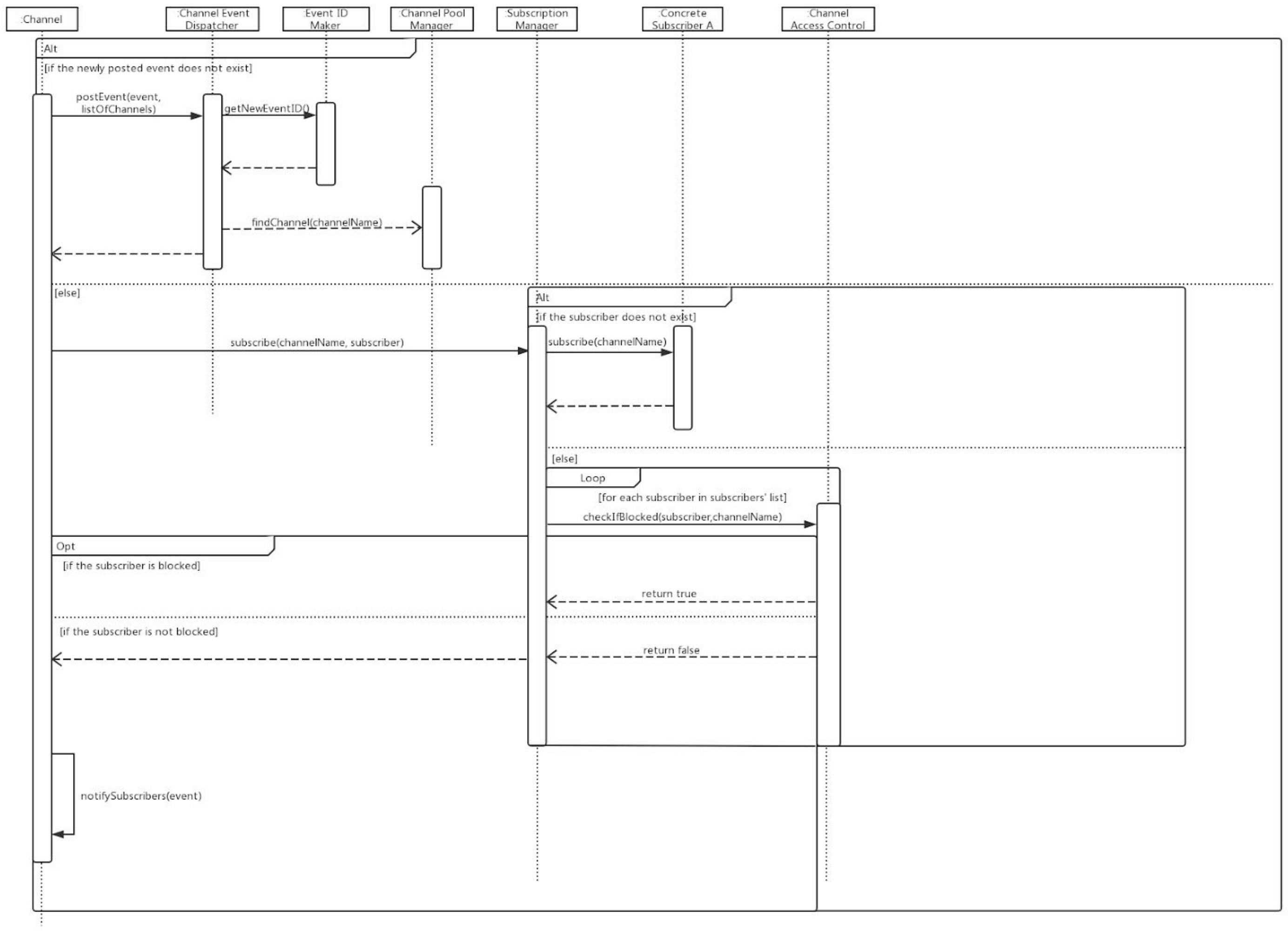
4 Interaction Diagrams

4.1 Sequence Diagram

4.1.1 Use Case 1 - Sequencing Diagrams

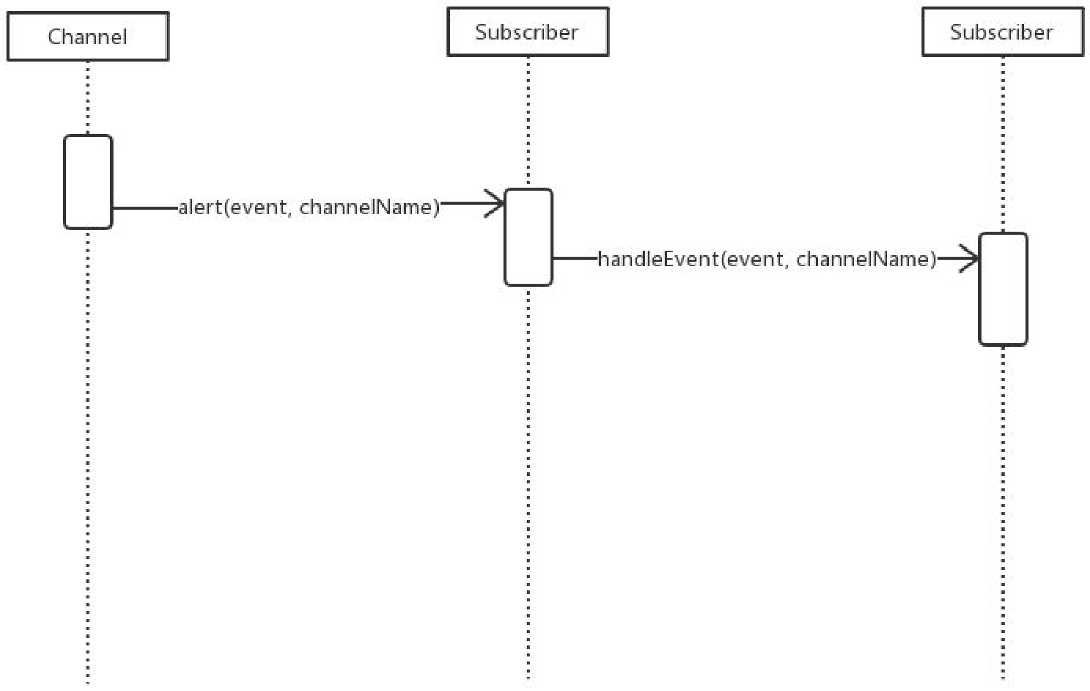

Publisher
Strategy
Event Factory
Channel Event Dispatcher
ChannelCreator
Channel
publish()
createEvent(eventType, eventPublisherId, payload)
return Event
postEvent(AbstractEvent event, List<String> listOfChannels)
alt
[if(channel == null)]
getInstance().addChannel(channelName)
publishEvent(event)
else
publishEvent(event)
publish(AbstractEvent event)
postEvent(AbstractEvent event, List<String> listOfChannels)
alt
[if(channel == null)]
getInstance().addChannel(channelName)
publishEvent(event)
else
publishEvent(event)

4.1.2 Use Case 2 - Sequencing Diagrams

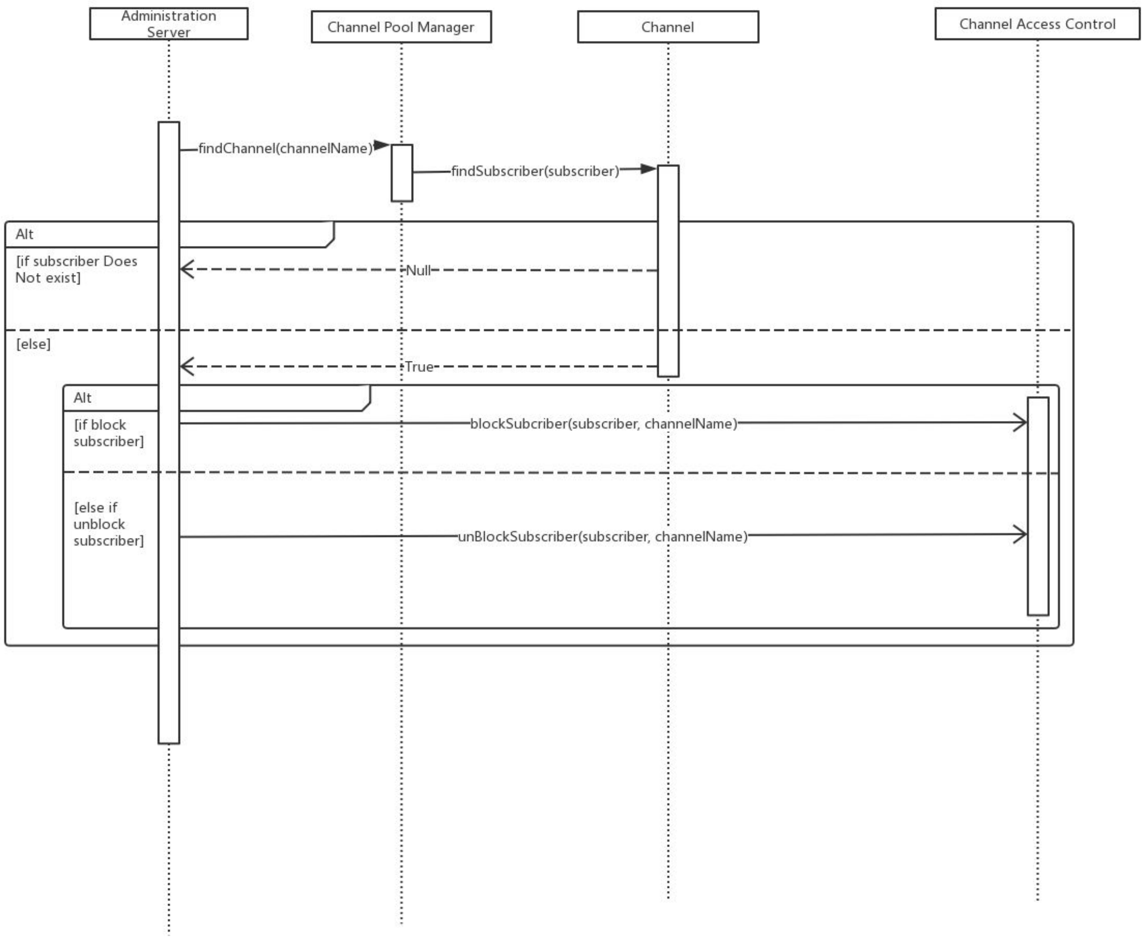


:Channel
:Channel Event Dispatcher
:Event ID Maker
:Channel Pool Manager
:Subscription Manager
:Concrete Subscriber A
:Channel Access Control
Alt
[if the newly posted event does not exist]
postEvent(event, listOfChannels)
getNewEventID()
findChannel(channelName)
[else]
Alt
[if the subscriber does not exist]
subscribe(channelName, subscriber)
subscribe(channelName)
[else]
Loop
[for each subscriber in subscribers' list]
checkIfBlocked(subscriber,channelName)
Opt
[if the subscriber is blocked]
return true
[if the subscriber is not blocked]
return false
notifySubscribers(event)

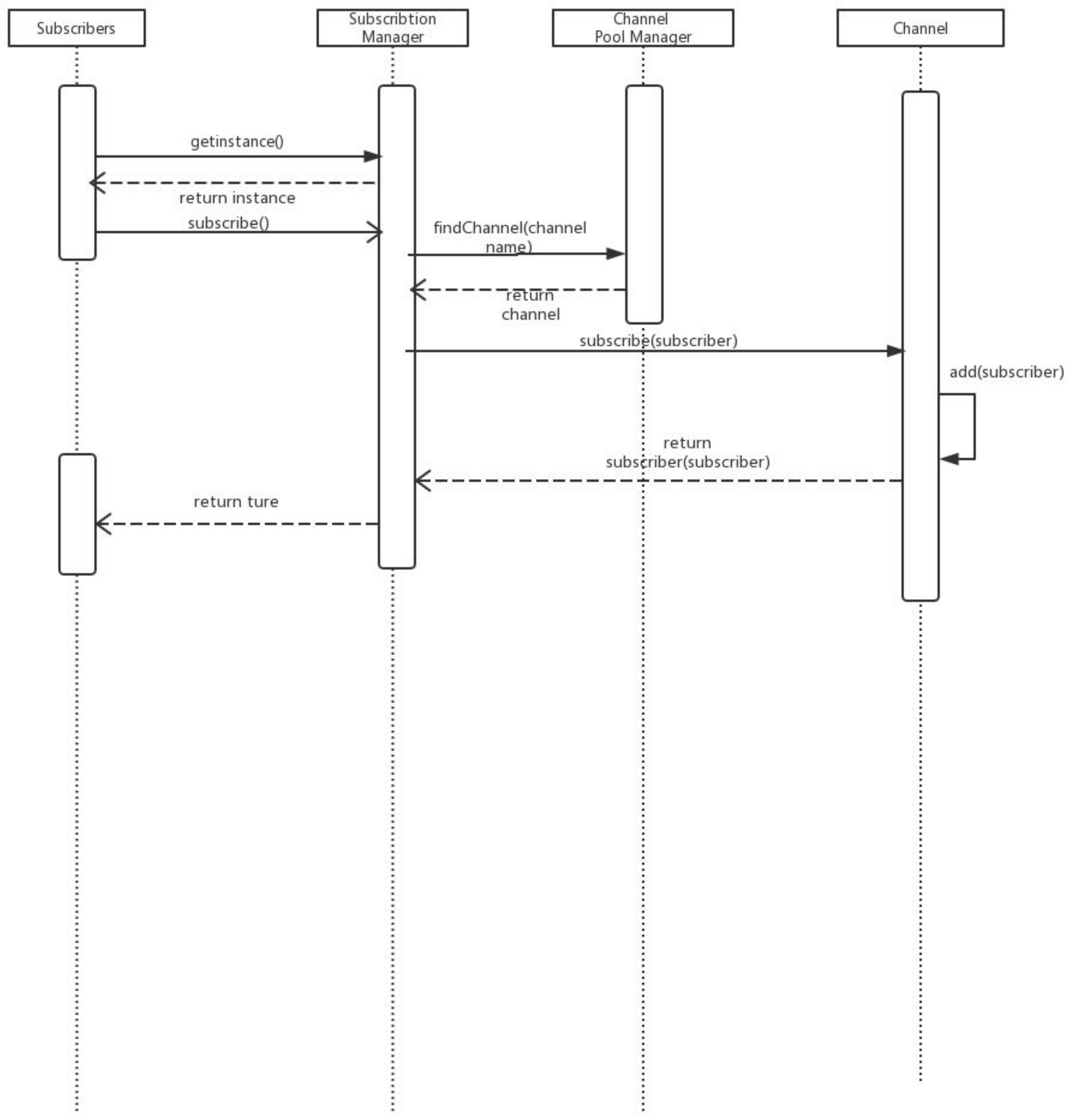
4.1.3 Use Case 3 - Sequencing Diagrams



4.1.4 Use Case 4 - Sequencing Diagrams

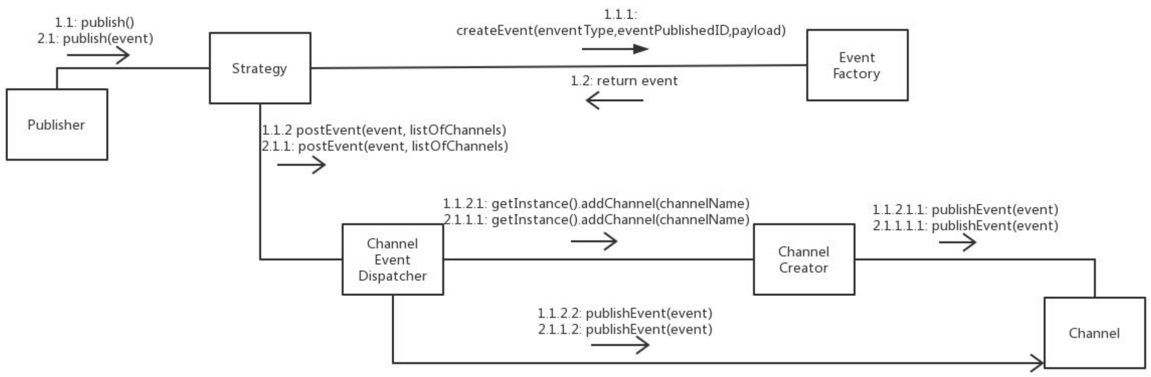

Administration Server
Channel Pool Manager
Channel
Channel Access Control
findChannel(channelName)
findSubscriber(subscriber)
Alt
[if subscriber Does Not exist]
Null
[else]
True
Alt
[if block subscriber]
blockSubcriber(subscriber, channelName)
[else if unblock subscriber]
unBlockSubscriber(subscriber, channelName)

4.1.5 Use Case 5 - Sequencing Diagrams

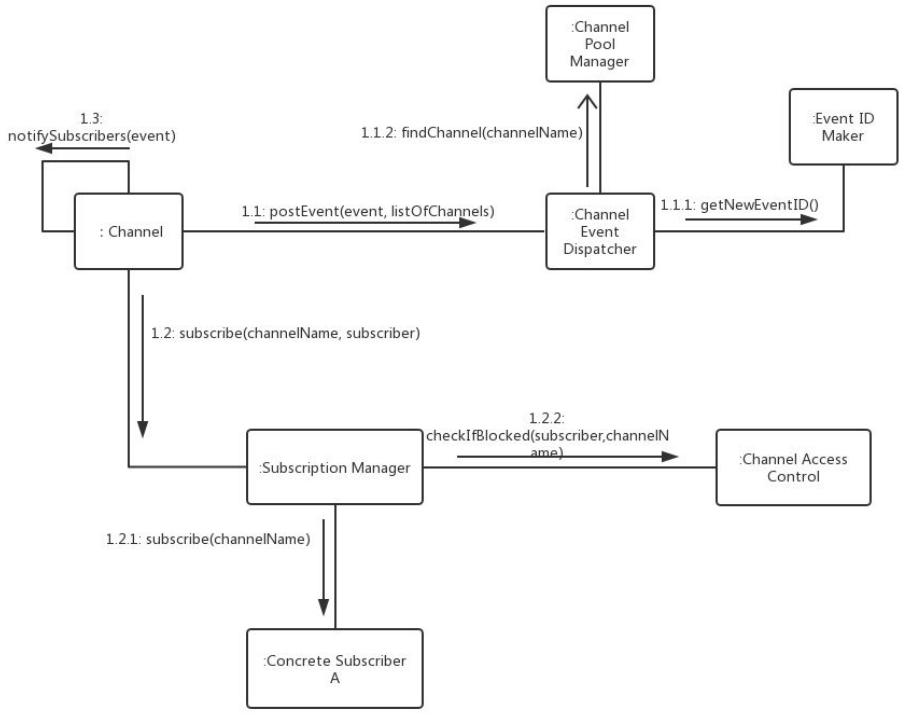


4.2 Collaboration Diagrams

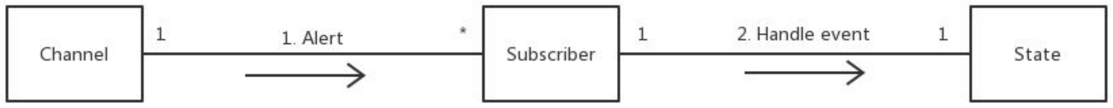
4.2.1 Use Case 1 - Collaboration Diagrams


1.1: publish()
2.1: publish(event)
Publisher
Strategy
1.1.1: createEvent(enventType,eventPublishedID,payload)
1.2: return event
Event Factory
1.1.2 postEvent(event, listOfChannels)
2.1.1: postEvent(event, listOfChannels)
Channel Event Dispatcher
1.1.2.1: getInstance().addChannel(channelName)
2.1.1.1: getInstance().addChannel(channelName)
Channel Creator
1.1.2.1.1: publishEvent(event)
2.1.1.1.1: publishEvent(event)
1.1.2.2: publishEvent(event)
2.1.1.2: publishEvent(event)
Channel

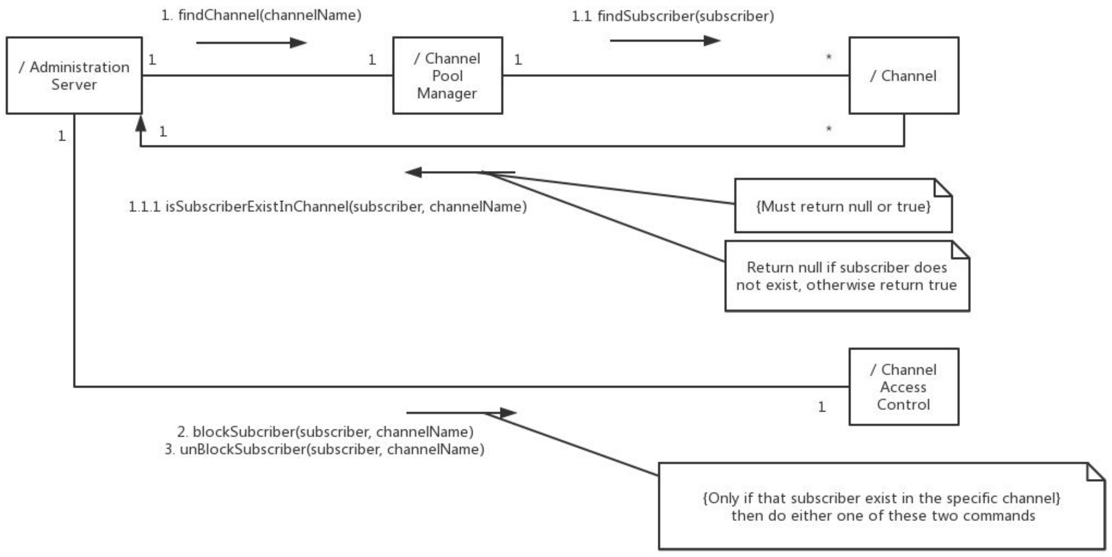
4.2.2 Use Case 2 - Collaboration Diagrams


:Channel Pool Manager
1.1.2: findChannel(channelName)
:Event ID Maker
1.3: notifySubscribers(event)
: Channel
1.1: postEvent(event, listOfChannels)
:Channel Event Dispatcher
1.1.1: getNewEventID()
1.2: subscribe(channelName, subscriber)
:Subscription Manager
1.2.2: checkIfBlocked(subscriber,channelName)
:Channel Access Control
1.2.1: subscribe(channelName)
:Concrete Subscriber A

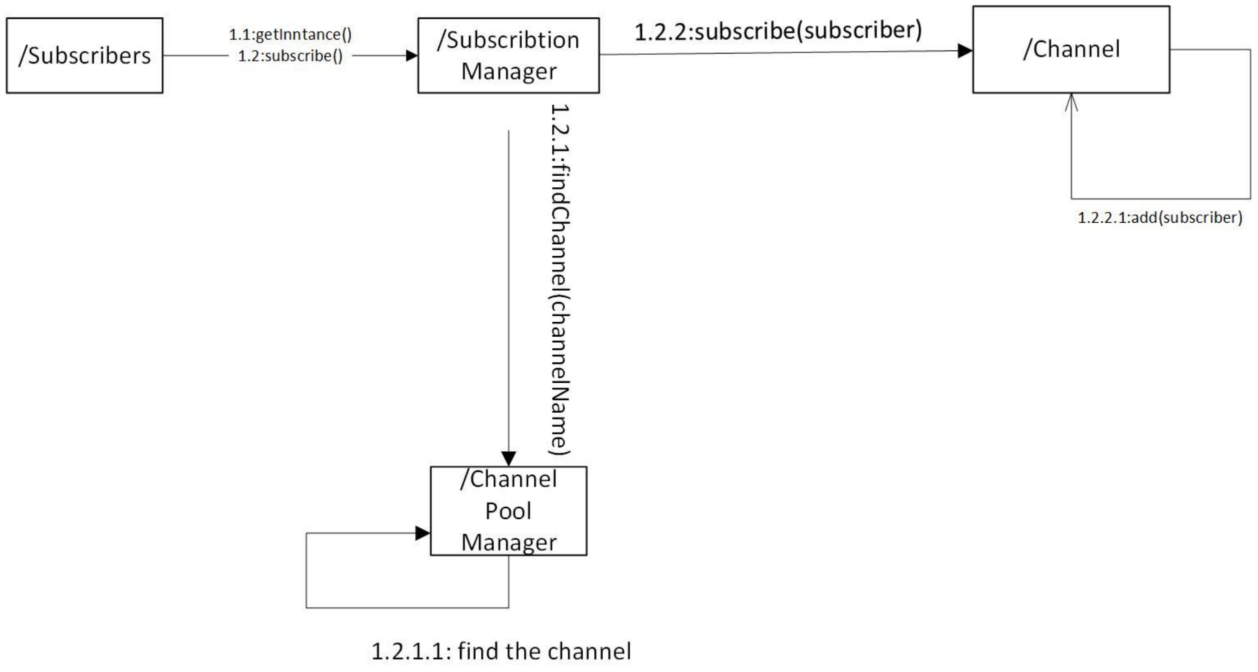
4.2.3 Use Case 3 - Collaboration Diagrams



4.2.4 Use Case 4 - Collaboration Diagrams


1. findChannel(channelName)
1.1 findSubscriber(subscriber)
/ Administration Server
1
1
/ Channel Pool Manager
1
*
/ Channel
1
1
*
1.1.1 isSubscriberExistInChannel(subscriber, channelName)
{Must return null or true}
Return null if subscriber does not exist, otherwise return true
/ Channel Access Control
1
2. blockSubcriber(subscriber, channelName)
3. unBlockSubscriber(subscriber, channelName)
{Only if that subscriber exist in the specific channel} then do either one of these two commands

4.2.5 Use Case 5 - Collaboration Diagrams



5 Non-Functional Requirements Specification

5.1 Overview

The non-functional requirements of the system comprises of utilities, environments and other specifications that are necessary for the smooth operation of the system as a whole. This includes interfaces, development environment, capacity specifications, network and operational parameters.

5.2 Enabling Technologies

5.2.1 Target Development Environment

The system should be developed in a Windows environment and using Java. Eclipse 3.1 will be the Integrated Development Environment (IDE) used for coding purposes.

5.2.2 System Interfaces

The current version of the publisher/subscriber system will not implement a graphic user interface, and text-based input and output interface will be used.

For the input, text based configuration files will be used to provide two group of information. One is the publishers (the Publisher) and their corresponding publishing strategies (the Strategy) in the form of tuples, <publisher-ID, strategy-ID>. The other is the subscribers and their corresponding event handling state (the State), also in forms of <subscriber-ID, state-ID>. All the files should be read at the beginning when the system starts running.

As to the output, all the important activities of the system will be printed by the console to the screen. Examples will be: “Channel x created”, “Subscriber x is on state y”, and “subscriber x is blocked on channel y”.

5.3 Capacity Planning

5.3.1 Runtime Capacity

The system must be able to handle at least 100 channels, and at least 200 publishers and 200 subscribers at different configuration. This should include their creation, configuration, and operation: the system should be able to create each entity and link publishers and subscribers with their corresponding strategies and states; during the running time, subscription, unsubscription, block, unblock, publishing events, notifying subscribers, and responding to events should be efficient without any noticeable delays.

5.3.2 Storage Capacity

The system should have enough capacity to store all the system codes, configuration files, and also the activity log of the publisher-subscriber system. The contents should be properly backed up either by an online cloud, another place within the same operation system, or any portable devices.

5.4 Workstations

The minimum system requirements and configurations for the computers used for the development, deployment and execution of the system are:
A hard disk space of 3GB to install Windows, a Java Virtual Machine, and also enough space for the publisher-subscriber system with configuration files and activity logs. A processor speed of 300 MHz and memory of 64 MB is sufficient. A display monitor should also be used to enable the text-based output.

5.5 Operational Parameters

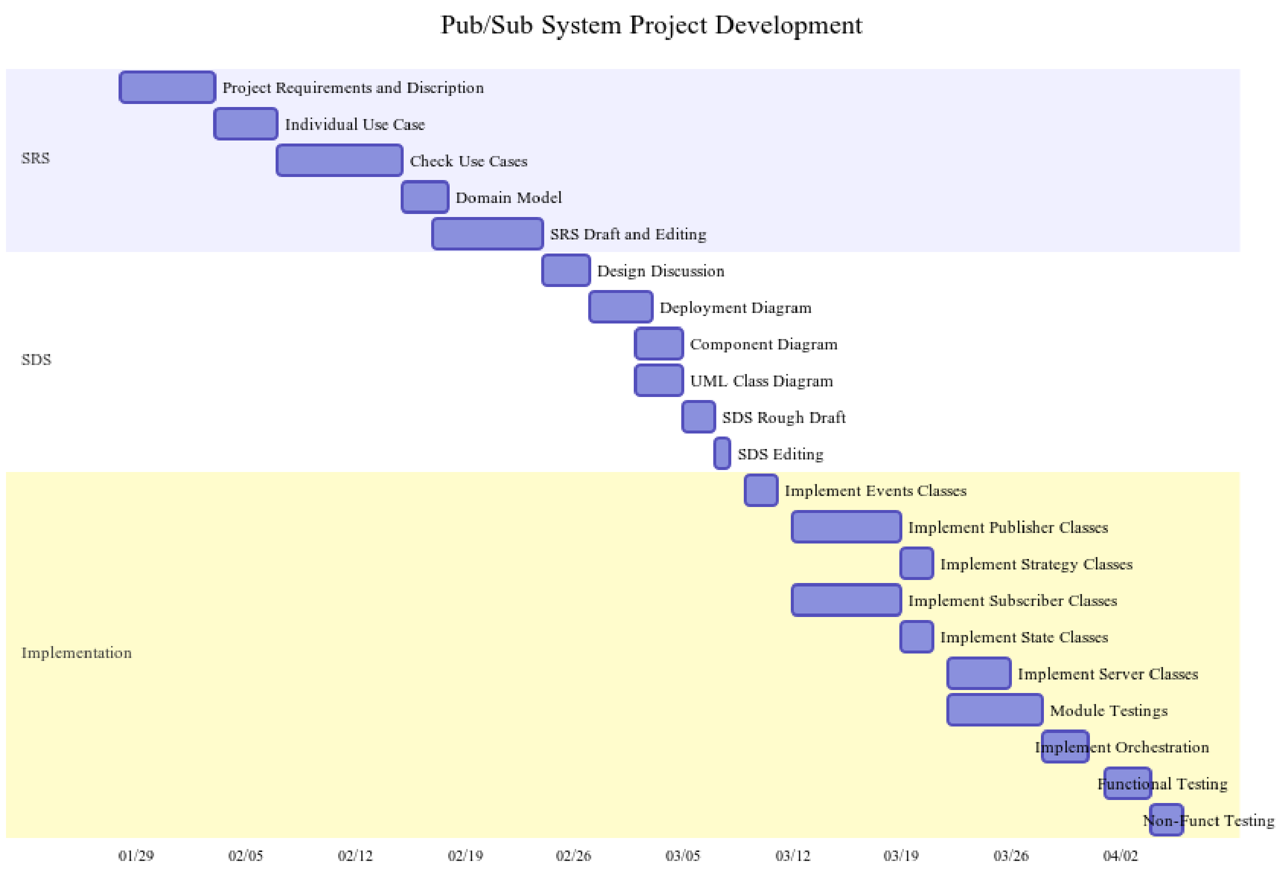
5.5.1 Usability

The publisher-subscriber system should be intuitive to use by the average computer users who has used other pub/sub systems. As there is no graphic user interface, an read-me file should be included in the system to guide users to set up and operate.

5.5.2 Reliability

The system should be backed up properly as required in the storage requirements. The recovery mechanism should be automatic or easy to implement by the users. Two levels should be minimized to ensure the system’s reliability: the number of failures that happen within a session and the time needed to recover the system.

6. Work Plan


Pub/Sub System Project Development
SRS
Project Requirements and Discription
Individual Use Case
Check Use Cases
Domain Model
SRS Draft and Editing
SDS
Design Discussion
Deployment Diagram
Component Diagram
UML Class Diagram
SDS Rough Draft
SDS Editing
Implementation
Implement Events Classes
Implement Publisher Classes
Implement Strategy Classes
Implement Subscriber Classes
Implement State Classes
Implement Server Classes
Module Testings
Implement Orchestration
Functional Testing
Non-Funct Testing
01/29
02/05
02/12
02/19
02/26
03/05
03/12
03/19
03/26
04/02

7 Meeting Minutes

Report on February 3rd:

- First meeting with every member and introduce individual.
- Discuss the project and separate Use Case to each member.
- Starting the initial writing.

Report on February 7th:

- Understand each member's project process
- Discuss and solve the problem that members have found
- Overview of the whole project for each member

Report on February 15th:

- Finishing and checking on three diagrams (Use Case Diagram, Sequence Diagram, Collaboration Diagram) and use case descriptions
- Communicates and working on Domain Model Class Diagram together

Report on February 18th:

- Begin processing on Software Requirement Specification Document
- Focus on Domain Model Class Definition

Report on February 22th:

- Finish the final writing and continue to final checking process

Report on February 24th:

- Complete the final checking and the document is completed

8 Domain Dictionary

All terminology used in this document has been described either in this document or in a referred document. Please consult reference documentation for additional details (see Section 1.3).