CS3301 Practical Publish/Subscribe System Report

170011474

April 2020

1 Overview

This report is about implementing an Event-driven publish/subscribe system based on Java RMI library. Channels are endpoints for publishers and subscribers can operate on and proper issue handlers are integrated.

2 Design and Implementation

2.1 Communication Mechanism: Event, Message, and Call-back

Event and Message essentially are the same but here `Event` is for posting and updating fruits and fruit prices from publishers to server (persistent), and `Message` is to send out acknowledgement and subscribed messages). `Event` is **persistent** event which can be stored in to `UUIDStore` and `Message` is **non-persistent** event. *In this way, separating them with different names brings more clear responsibility.*

Each `Event` has its own unique `UUID` for further identification. It is stored in `UUIDStore` used for duplication check and is removed after consumption and all subscribers have responded. `Message channel` for acknowledgement from subscribers needs to record corresponding `Event UUID` as `MessageChannelId` so it knows which event is subscribers responding to.

In order to send real-time message bidirectionally in RMI, clients themselves need to create a registry and rebind a port number. The server needs to store all client's host address to do **call-back** (particularly for subscribers).

2.2 Register Process

To discover and access via server, publishers and subscribers can call `register()` in server to enroll themselves for further discovery. Corresponding acknowledgement messages are sent back.

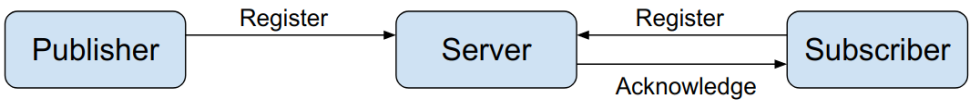

Figure 1: Register Process

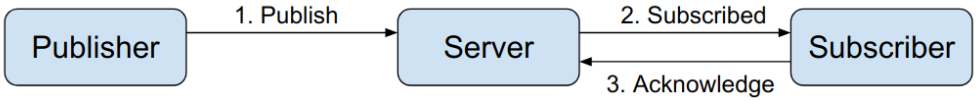

Figure 2: Publish Process

2.3 Publish/Subscribe Process

Publishers invoke `addEvent()` to push events into corresponding topic channels. The channel's individual executing thread consumes events and eventually calls `publish()`. Subscribers would be notified with information about subscribed fruits. To acknowledge they have received messages, they call `addMessage()` to push acknowledging messages into a unique message channel related to that event.

`Message channel` starts its thread to consume and update "unreplied" subscriber list. If a certain period expires and there is still some unreplied subscriber, then server would resend and wait for responses, up to 3 times resending. Otherwise, the message channel is removed.

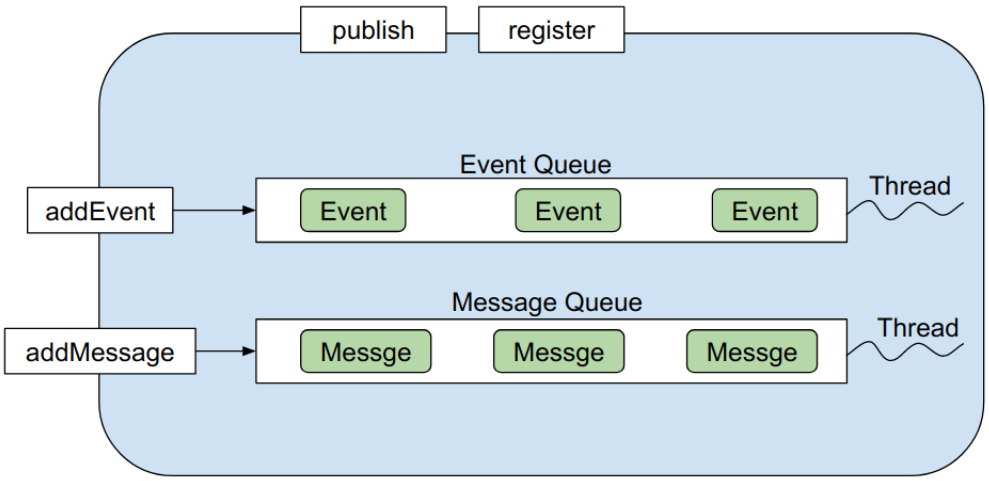

Figure 3: Publish Details

2.4 Thread-safe Queue Management

Each event channel and each message channel has some message queues in a list. When a new channel is created, its queue list is instantiated and invoked a consumer thread to process elements in queues. A thread is running forever to consumes events or messages in each channel. Each queue has a limit size and a new queue is created when the limit is reached and added to the queue list.

In Event channel, to guarantee there is a upper bound so memory space would not be overflowed, the maximum number of queues can be set in the code. In Message channel, however, we can determine the maximum size is the number of subscribers we are waiting for, so it can be set automatically in the constructor.

2.5 Costumed Exceptions

To simplify the process, costumed exceptions are implemented used particularly in publishing process. When a publisher is trying to `addEvent()` to publish an event, it needs to handle exceptions thrown by the remote server. *This simplifies communication because the server does not need to send back a message including error message, type and etc. A simple and straightforward exception would represent everything and is easy to be handled.*

3 Testing and Issue Handlers Design

Notice: All following terminal outputs are included in the code block with green comments for more readable outputs than direct screenshots.

3.1 R1. Subscribing and Publishing

3.1.1 normal subscription and publish

Subscriber subscribed "apple" and the output:

```
sender: server content: Subscriber Registered
Menu
(0) Subscribe
(1) UnSubscribe
Please enter a number to act
0
Enter the fruit name you want to subscribe:
apple
Successfully subscribed

Menu
(0) Subscribe
(1) UnSubscribe
Please enter a number to act
```

```
sender: server content: fruit name: apple fruit price: 1.0 // real time message
```

Publisher published fruit "apple":

```
Menu:
(0) Publish / Update a fruit
Please enter a number
0
Enter Fruit Name:
apple
Enter Fruit Price:
1
```

Notice that the **last line** in the output of Subscriber. That is the real-time message from server, because subscribed "apple" info was published by the publisher and the message would be displayed immediately. From here, our program can do publishing and subscribing successfully.

3.1.2 unsubscription

To test unsubscribing function, simply "subscribe" and then "unsubscribe" apple before publishing. The subscriber then should not receive any update about apple from publishers:

```
Menu
(0) Subscribe
(1) UnSubscribe
Please enter a number to act
sender: server content: fruit name: apple fruit price: 1.0 // received apple
    information
1 // unsubscribe from apple
Enter the fruit name you want to unsubscribe:
apple

Menu
(0) Subscribe
(1) UnSubscribe
Please enter a number to act
// no more apple information event if apple's price changed to 2
```

```
Menu:
(0) Publish / Update a fruit
Please enter a number
0
Enter Fruit Name:
apple
Enter Fruit Price:
2 // newly published apple information
```

```
Menu:
(0) Publish / Update a fruit
Please enter a number
```

3.1.3 multiple publishers

To test multiple publishers, running 2 publishers in parallel and the subscriber subscribed "apple". 2 publishers published an event about apple respectively:

```
// publisher 1
Menu:
(0) Publish / Update a fruit
Please enter a number
0
Enter Fruit Name:
apple
Enter Fruit Price:
1
Menu:
(0) Publish / Update a fruit
Please enter a number

// publisher 2
Menu:
(0) Publish / Update a fruit
Please enter a number
0
Enter Fruit Name:
apple
Enter Fruit Price:
2 // updates apple's price from 1 to 2
Menu:
(0) Publish / Update a fruit
Please enter a number

// subscriber
sender: server
content: Subscriber Registered
Menu
(0) Subscribe
(1) UnSubscribe
Please enter a number to act
0
Enter the fruit name you want to subscribe:
apple
Successfully subscribed
```

```
Menu
(0) Subscribe
(1) UnSubscribe
Please enter a number to act
sender: server
content: fruit name: apple fruit price: 1.0
sender: server
content: fruit name: apple fruit price: 2.0
```

3.1.4 multiple subscribers

```
// subscriber 1
sender: server
content: Subscriber Registered
Menu
(0) Subscribe
(1) UnSubscribe
Please enter a number to act
0
Enter the fruit name you want to subscribe:
apple
Successfully subscribed

Menu
(0) Subscribe
(1) UnSubscribe
Please enter a number to act
sender: server
content: fruit name: apple fruit price: 1.0 // received

// subscriber 2
sender: server
content: Subscriber Registered
Menu
(0) Subscribe
(1) UnSubscribe
Please enter a number to act
0
Enter the fruit name you want to subscribe:
apple
Successfully subscribed

Menu
(0) Subscribe
(1) UnSubscribe
Please enter a number to act
```

```
sender: server
content: fruit name: apple fruit price: 1.0 // received
```

3.2 R2. Lookup, Discovery, and Access of Event Channels

Please see **R1**. Since subscribers and publishers can always find the correct topic channel in the channel map to push and pull events.

3.3 R3. Dynamic Message Queue Management

Please see **R7**.

3.4 R4. Connection Failures

When publishers and subscribers cannot connect to the server, they would sleep for a period waiting for recovery and then reconnect. **In this case, we simply do not run `server` so the publisher cannot connect to server**. This simulates the case where the server is permanently lost, and the upper bound of retry is 3. The benefit of setting an upper bound is to prevent spinning lock. The corresponding screen would display as below:

```
reconnecting...
reconnecting...
reconnecting...
Cannot reconnect to the server
```

If server recovers, in this case, we run `server` immediately after running `PublisherClient` then:

```
reconnecting... // only 1 "reconnecting..." means the target recovered after 1
    trial
Menu:
(0) Publish / Update a fruit
Please enter a number
```

3.5 R6. Crashing Subscribers

When some subscriber of the current channel "apple" crashes and cannot receive the newest update, the reconnecting mechanism (introduced above) and re-sending mechanism are used (at most 3 times of re-sending). **In this case, we can comment out from line 30 to 46 in SubscriberClientImpl.java to simulate the case where subscribers do not send back acknowledge messages**.

```
// healthy server
server ready
consumer starts
```

```
message thread starts
Debug: all subscribers responded
```

```
// when there is some crashing customer
server ready
consumer starts
message thread starts
// because at most 3 re-sending, each time the subscriber does not respond
Debug: subscriber not available: [ranley]
Debug: subscriber not available: [ranley]
Debug: subscriber not available: [ranley]
```

3.6 R7. Increasing Number of Queries

To test queue management, simply initialises the limit of the queue to 0 and the maximum queues we can have is 1, which means any event pushed would immediately raise `QueueIsFullException`**.** On the server side, the event consumer thread would notice the full queue when `produce()` is called. On the client side, re-publishes it after sleeping.

```
Menu:
(0) Publish / Update a fruit
Please enter a number
0
Enter Fruit Name:
apple
Enter Fruit Price:
1
queue is full // 3 times "queue is full" because re-sent 3 times after waiting
queue is full
queue is full
```

From above, after `QueueIsFullException` was thrown, the queue limit increased to 1 and the next time a new event would be pushed successfully.

3.7 R9. Dropped Messages/Events

When data loss occurs, the checker would notice and throw a `DataLossException` handled by publishers. **To test it, publisher would firstly send** `null` **to get an exception shown and then send a real event to simulate the case where data loss occurs.** If successful, then the new menu would show:

```
Menu:
(0) Publish / Update a fruit
Please enter a number
0
```

```
Enter Fruit Name:
apple
Enter Fruit Price:
1
DataLossException

Menu:
(0) Publish / Update a fruit
Please enter a number
```

From above, the function works fine.

3.8 R11. Duplicated Events

Each `Event` has a unique UUID to be saved into `UUIDStore` on event received and be removed once all subscribers have responded. If a duplicate event occurs, then `UUIDStore` notices it on receiving and throw `DuplicateException`. The duplicate event is dropped. **To test it, we `addEvent()` twice to simulate duplicated events.**

```
Menu:
(0) Publish / Update a fruit
Please enter a number
0
Enter Fruit Name:
apple
Enter Fruit Price:
1
Duplicate event, dropped
```

4 Questions and Evaluation

Those testings submitted are typical and representative.

All requirements have been satisfied and the program is relatively stable and basic functionalities work properly. Multiple challenging issues with solutions provided.

5 Conclusion

During the implementation, I gained deeper insights into publish/subscribe system in concepts and structure and thread manipulation. A database storing persistent events can be made if more time given (event store) to be more event-driven.

6 Instruction

You can use IntelliJ to run this program. The `Server` runs firstly. Usage can be viewed from above testings.

```
java Server
java Publisher Client
java SubscriberClient [port number] [name]
```