Simple Public Transport System

Wen Li (11660347)
Washington State University, Pullman, USA
li.wen@wsu.edu

I. Approach

In this section, the introduction of the *Simple Public Transport System* will be presented from basic architecture design to detail of implementation logic, the whole system is programmed with language c++ based on the middle-ware *RTI-DDS*. The project code could be fetched *here*.

A. Overview

With the strong support of *RTI-DDS*, a three-layered architecture could be easily constructed as showed in Figure 1. Based on the code generated by *rtiddsgen*, the message publisher and subscriber are redesigned with object-oriented approach to build the message layer, on which the three applications (Passenger, Operator, Vehicle) were implemented independently.

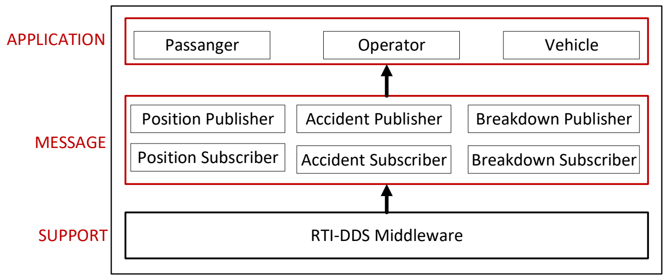


Fig. 1. Overview of Simple Public Transport System.

B. Message Layer

The message layer is built on the support of *RTI-DDS* middle-ware, containing two parts: Publisher and Subscriber. The basic class of Publisher or Subscriber abstracts the initiation of procedure for a entity that is needed for message publishing or subscribing. Sub-classes relevant to specific message types would implement the details strongly correlated with the message itself.

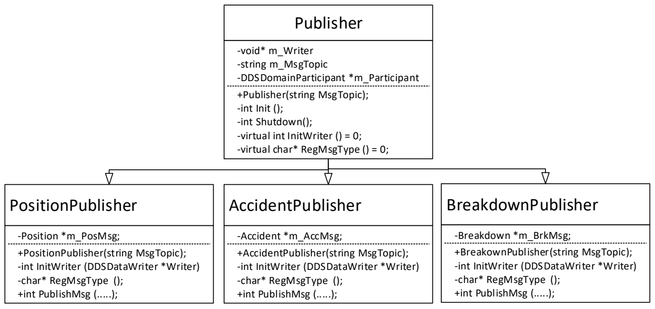

Publisher
-void* m_Writer
-string m_MsgTopic
-DDSDomainParticipant *m_Participant
+Publisher(string MsgTopic);
-int Init ();
-int Shutdown();
-virtual int InitWriter () = 0;
-virtual char* RegMsgType () = 0;
PositionPublisher
-Position *m_PosMsg;
+PositionPublisher(string MsgTopic);
-int InitWriter (DDSDataWriter *Writer)
-char* RegMsgType ();
+int PublishMsg (.....);
AccidentPublisher
-Accident *m_AccMsg;
+AccidentPublisher(string MsgTopic);
-int InitWriter (DDSDataWriter *Writer)
-char* RegMsgType ();
+int PublishMsg (.....);
BreakdownPublisher
-Breakdown *m_BrkMsg;
+BreakownPublisher(string MsgTopic);
-int InitWriter (DDSDataWriter *Writer)
-char* RegMsgType ();
+int PublishMsg (.....);

Fig. 2. Classes Design of Publisher.

The Figure 2 shows the class design of Publishers, and Figure 3 the design of Subscribers. As a sub-class of publisher, it needs to initialize a Message Writer entity and register the message type for the procedure of message publisher setup. Then it provides a interface for message publishing, which could be utilized directly by applications. More easily for a sub-class of subscriber, it only needs to implement a function for message registering, however, while instantiating this sub-class an application needs to provide a *DDSDataReaderListener* entity which would be utilized for message subscribing.

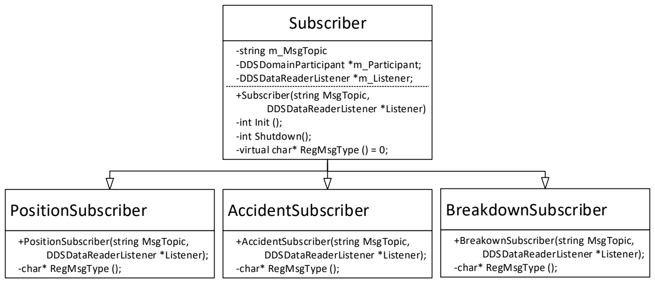

Subscriber
-string m_MsgTopic
-DDSDomainParticipant *m_Participant;
-DDSDataReaderListener *m_Listener;
+Subscriber(string MsgTopic, DDSDataReaderListener *Listener)
-int Init ();
-int Shutdown();
-virtual char* RegMsgType () = 0;
PositionSubscriber
+PositionSubscriber(string MsgTopic, DDSDataReaderListener *Listener);
-char* RegMsgType ();
AccidentSubscriber
+AccidentSubscriber(string MsgTopic, DDSDataReaderListener *Listener);
-char* RegMsgType ();
BreakdownSubscriber
+BreakownSubscriber(string MsgTopic, DDSDataReaderListener *Listener);
-char* RegMsgType ();

Fig. 3. Classes Design of Subscriber.

Based on the design, the support for a new message could easily be implemented with little impact to existing messages, it helps the systems obtain better expansibility. The whole message layer would be compiled into a dynamic library, which help with system upgrades and patches.

C. Application Layer

As described in the requirements, there are two kinds of applications: Subscriber (Passenger, Operator) and Publisher (Vehicle), which are constructed on the message layer. Each application will be released as an executable.

■ **Subscriber**

In the application layer, three message listeners Inheritance from *DDSDataReaderListener* need to be implemented, and an entity of basic application class called *AppSubcriber* would be a member of the three listeners, so that the applications could process the message received by the listeners. The class design of Subscriber is showed as Figure 4. Both Passenger and Operator inherit from *AppSubcriber* which has handled the message subscribing, and they only needs to focus on their own business logic through override the three message processing *API: PosMsgProc, AccMsgProc, BrkMsgProc*. To launch a Passenger or Operator, three subscribers would be created in three independent threads, so that a *AppSubcriber* could process all messages simultaneously.

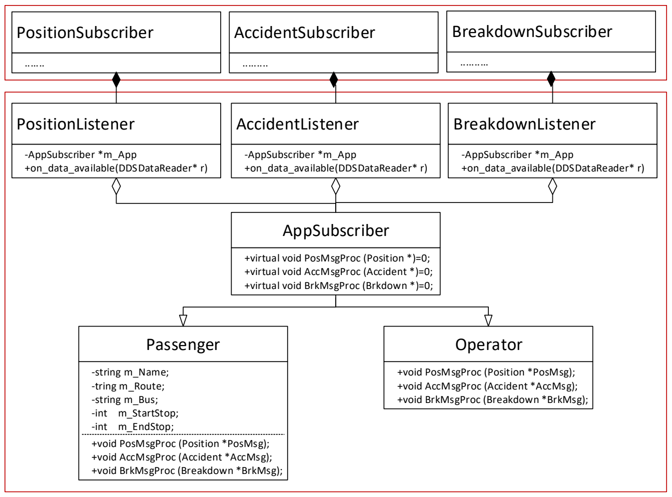

PositionSubscriber
AccidentSubscriber
BreakdownSubscriber
PositionListener
-AppSubscriber *m_App
+on_data_available(DDSDataReader* r)
AccidentListener
BreakdownListener
AppSubscriber
+virtual void PosMsgProc (Position *)=0;
+virtual void AccMsgProc (Accident *)=0;
+virtual void BrkMsgProc (Brkdown *)=0;
Passenger
-string m_Name;
-tring m_Route;
-string m_Bus;
-int m_StartStop;
-int m_EndStop;
+void PosMsgProc (Position *PosMsg);
+void AccMsgProc (Accident *AccMsg);
+void BrkMsgProc (Breakdown *BrkMsg);
Operator

Fig. 4. Classes Design of Subscriber.

■ Publisher

There is only one Publisher (Vehicle) that will publish all the three kinds of message. a class *Property* is designed for the purpose of loading property file which maintains information of routes and vehicles, based on which all entities of routes and vehicles would be initialized, then a class *PubThread* is defined to represent a dynamic status that a vehicle runs on a route, in each *PubThread*, three knids of *Publishers: PositionPublisher, BreakownPublisher, AccidentPublisher* would be created to publish messages. Class design is showed as Figure 5.

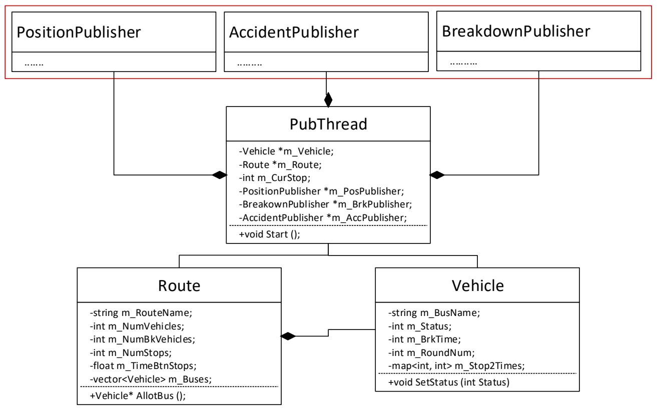


Fig. 5. Classes Design of Publisher.

[1] Procedure of *PubThread* Launching

The initiation part does mainly tree tasks: Load properties, Launch all *PubThreads* and Launch a thread for bus fixing.

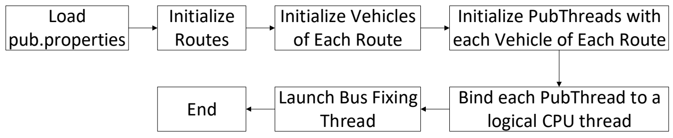

Load pub.properties
Initialize Routes
Initialize Vehicles of Each Route
Initialize PubThreads with each Vehicle of Each Route
Bind each PubThread to a logical CPU thread
Launch Bus Fixing Thread
End

Fig. 6. Procedure of *PubThread* Launching.

[2] Procedure of *PubThread* Running

Each *PubThread* collects traffic status before arriving at a stop, and checks the bus condition (breakdown or accident) while reaching a stop. When a bus breaks down, a breakdown message will be published first, then wait for 15 seconds for allotting a new bus to replace it. When a new bus is allotted, a position message will be published immediately. The broken bus will be fixed by fixing thread after 20 seconds and changed as a backup bus. When a bus happens an accident, it will publish a accident message and then waits for 10 seconds, after that publish a position message. If the bus has run three rounds on the route, then exit, otherwise publish a position message.

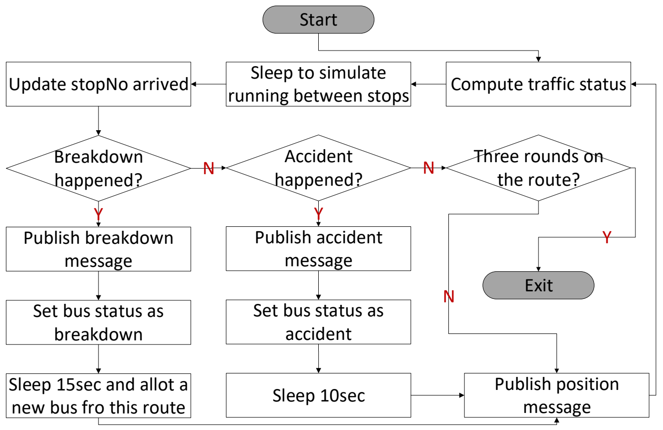

Start
Compute traffic status
Sleep to simulate running between stops
Update stopNo arrived
Breakdown happened?
N
Accident happened?
N
Three rounds on the route?
Y
Publish breakdown message
Y
Publish accident message
Y
Exit
N
Set bus status as breakdown
Set bus status as accident
Sleep 15sec and allot a new bus fro this route
Sleep 10sec
Publish position message

Fig. 7. Procedure of *PubThread* Running.

[3] Fixing of Breakdown Buses

The responsibility of bus fixing thread is to scan each vehicle of each route, and check whether there is a bus broken down, if the bus has broken down for 20 seconds, then it could be fixed and the status would be set as backup. This thread will exit when all *PubThreads* exit.

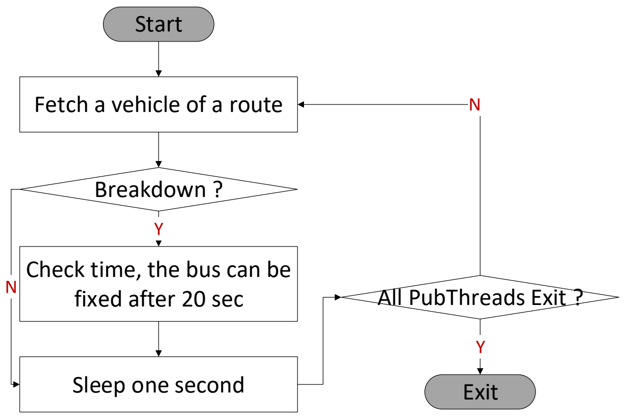

Start
Fetch a vehicle of a route
N
Breakdown ?
Y
Check time, the bus can be fixed after 20 sec
N
All PubThreads Exit ?
Y
Sleep one second
Exit

Fig. 8. Procedure of Breakdown Buses Fixing.

II. Lesson And Learn

In this projects, two problems has blocked the process for a while: How compile the *Message Layer* into a dynamic library and A resource limit for *RTI-DDS* participants.

■ Compile *Message Layer* into dynamic library

After all code of message publisher and subscriber are re-constructed with object-oriented approach, the final step is to compile these code into a dynamic library, which could be shared to all up-layer applications. However, the makefile generated by *rtiddsgen* only support to compile the code into executable. So what i have done is to read the original makefile carefully and collect all compilation parameters, then rewrite the makefile for this module.

■ **Resource limit for *RTI-DDS* participants**
As the original design, i supposed to let all publishers with same topics in all *PubThreads* share one same participant, but i not sure whether this approach would cause read and write conflict and i do not want to implement a synchronization mechanism between *PubThreads*, which would increase the complexity of my design. So i choose a simple design, each *PubThread* owns a private participant, which leads to an resource limit error, after i read some references, it seems that one progress could only support 8 participants at most by default. There are many approach recommended in google, however few are useful and some are too complicated. Fortunately, i find one easy solution in RTI help documents, utilizing API DDSTheParticipantFactory::set_qos to extend support number of participants.

Overall, i have learned how to build a message publish-subscribe system with *RTI-DDS* middle-ware through this simple project.

III. Final Status of Submission

The project has been implemented completely with the requirement documents. Three applications would be generated: Vehicle, Operator and Passenger.
[1] Vehicle: simulate all buses (except backup buses) running on a route, reporting normal message (Position) and abnormal messages (Breakdown,Accident) with specific probability. A bus fixing thread is created to monitor the broken bus and try to fix it.
[2] Operator: Subscriber all three message and print out with a required format.
[3] Passenger: Before getting on a bus, a passenger will subscribe all message on the route and print messages received, then he/she only deal with the message on the bus, when the bus breakdown, he/she will re-subscribe all messages on the route until getting on a bus again.

IV. Compilation and Utilization

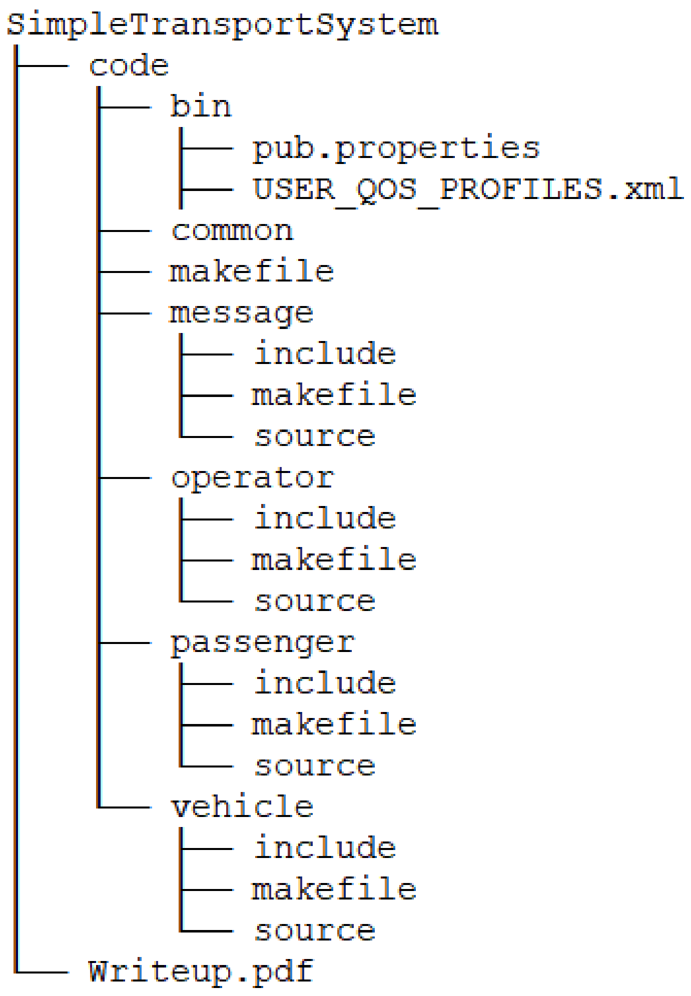
This section is to introduce the directory structure of the project, then show how to compile the whole project and the way to execute the programs.

A. Directory Introduction

As showed in Figure 9, *Message Layer* is implemented in directory message, and directories operator/passenger/vehicle are for the three applications.

B. Compilation

Entry directory SimpleTransportSystem/code, then utilize command **”make clean && make”** to compile the whole project, and three executable would be generated in the directory SimpleTransportSystem/code/bin, the dynamic library of



```
SimpleTransportSystem
├── code
│   ├── bin
│   │   ├── pub.properties
│   │   ├── USER_QOS_PROFILES.xml
│   ├── common
│   ├── makefile
│   ├── message
│   │   ├── include
│   │   ├── makefile
│   │   └── source
│   ├── operator
│   │   ├── include
│   │   ├── makefile
│   │   └── source
│   ├── passenger
│   │   ├── include
│   │   ├── makefile
│   │   └── source
│   └── vehicle
│       ├── include
│       ├── makefile
│       └── source
└── Writeup.pdf
```

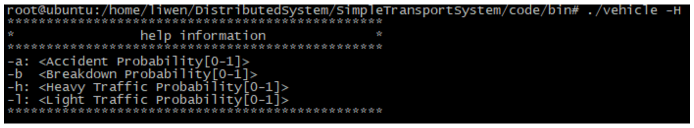
Fig. 9. Directory of The Whole Project.

Message Layer would be generated in the directory SimpleTransportSystem/code/library.

C. Utilization

After the project is compiled, then executalbes bould be run in SimpleTransportSystem/code/bin.

[1] Launch Vehicle:
Open a terminal, entry SimpleTransportSystem/code/bin, Utilize **”./vehicle -H”** for help (Figure 10), if launching vehicle with no parameters, it will start by default. Otherwise we could use the parameters to change the probability, for example: command ”./vehicle -a 0.3 -h 0.25” will change the probability of accident to 0.3 and probability of heavy traffic to 0.25.



```
root@ubuntu:/home/liwen/DistributedSystem/SimpleTransportSystem/code/bin# ./vehicle -H
************************************************
*                 help information             *
************************************************
-a: <Accident Probability[0-1]>
-b  <Breakdown Probability[0-1]>
-h: <Heavy Traffic Probability[0-1]>
-l: <Light Traffic Probability[0-1]>
************************************************
```

Fig. 10. Directory of The Whole Project.

[2] Launch Passenger:
Open a terminal, entry SimpleTransportSystem/code/bin, help information will be printed by entering command **”./passenger”** as ”passenger [passenger-name] [route] [start-stop] [end-stop], range: routeExpress1[1,4], route-Express2[1-6]”.

For example: command "./passenger Passenger1 Express1 3 1" represents Passenger1 is waiting for bus on route Express1 at stop 3, and the end stop is 1.

[3] Launch Operator:
Open a terminal, entry SimpleTransportSystem/code/bin and enter command **"./operator"** to launch Operator.

V. Experiment Results