Code-Reusable Platform User Guide

Document Revision History

Version	**Page, Section**	**Change Content**	**Changed By**	**Change Date**
V1.0	/	Create	Yifei Peng	

Table of Contents

1 summarize

As the application demand for network determinism and reliability grows, researchers are committed to developing higher-level protocols or entirely new network protocol stacks. Real-world testing, while most convincing, faces challenges of scale and budget, making simulation experiments an important pre-step. However, compatibility issues between simulation code and real devices limit its direct application. Meanwhile, the code developed on real devices needs to be verified by building a large number of test cases. To address this issue, this paper proposes a lightweight code reusability framework to realize the migration of protocol stack code across multiple emulation software and operating systems.

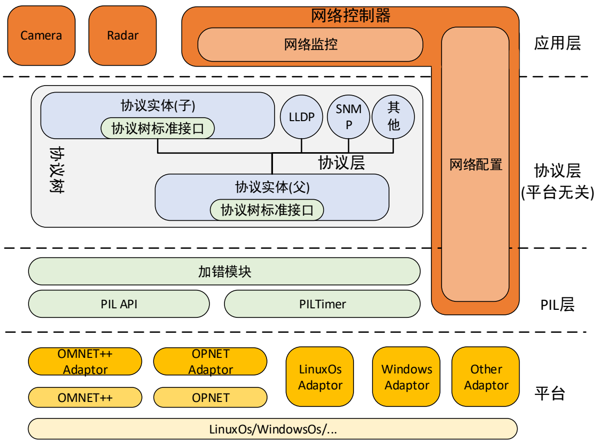


Figure 1 Code Reusability Framework

As shown in Figure 1, this is the module block diagram of cross-platform middleware. The whole middleware consists of the following parts: platform layer, platform-independent layer, protocol layer and application layer. In order to make this framework scalable, different adapters are designed for different platforms. The adapters encapsulate clock-related interfaces and packet transceiver-related interfaces and provide standard interfaces to the PIL (Platform Independence Layer) layer. This allows the PIL layer to design the PILTimer and PIL API based on these standard interfaces, where the PIL API completes the data sending and receiving, the PILTimer maintains a list of discrete events and provides a unified timer-related functions for the upper layer, and the error-adding module provides the ability to add errors to the data.On top of the PIL layer is the protocol layer, which utilizes the relevant APIs in PIL to complete the relevant operations required by the network protocols, such as The protocol layer is above the PIL layer, which utilizes the relevant APIs of PIL to complete the relevant operations required by the network protocols, such as setting the timeout timer of the protocol through the interface provided by the PIL layer. In order to facilitate protocol management and data transfer between protocols, we maintain a protocol tree in the protocol layer and define a series of interfaces for protocol tree expansion and protocol management. The application layer realizes the configuration management of basic applications and protocols.

2 Description of the basic data structure

2.1 packetStruct

The packetStruct is similar to the skb_buff structure in the linuxOs stack. This data is used to unify the maintenance of frames and frame-related information. packetStruct structure mainly contains the frame buffer, and frame-related information. The frame buffer contains the byte information of the frame. The frame-related information is the key fields extracted from the byte information of the frame as well as the sending and receiving moments of the frame, etc. The packetStruct structure is also used for information transfer between the upper and lower layers of the protocol. The following table shows the key variables in the packetStruct.

Table 1 some key variables in the packetStruct structure

variable name	meaning	type
deviceId	Used to label which device the frame belongs to	int
portNum	Used to label which port the frame originated from	int
type	Types used for standard protocols	int
frameSize	Used to label the length of the frame	unsigned int
frameBuf	Byte stream for storing frames	unsigned char
receiveTime	Used to store the received moment of the frame	double
outputPortList	Used to store which ports the frame should be sent to	vector<int>

The user is free to extend this structure. For example, the variable vector<int> outputPortList; is extended by the user on demand to indicate the ports from which data frames are sent.

2.2 netifItem

netifItem is used to store interface information. This structure is mainly used to shield the difference between different platforms. Since different platforms describe the network interface in different ways, for example, linuxOs uses netif or socketFd to describe the network interface, Omnet++ uses string to describe Gate, and Opnet uses StreamHandler to describe the connection relationship of device pieces. In order to enable the user program to be platform-independent, we created the netifItem structure to implement the mapping of the platform interface to the logical interface portNum. Thereafter, the user only needs to build the mapping relationship in the adaptor.cpp of different platforms by himself, and when writing the protocol-related code, he only needs to use the logical port.

The following table gives some key variables in netifItem.

Table 2 some key variables in the netifItem structure

variable name	meaning	type

netifType	interface type, which is required under linuxOs.	int
socketFd	Socket descriptors are required for linuxOs.	unsigned int
netifName	interface name	char
portNum	Numbering of logical interfaces	unsigned int

2.3 registFuncTab

Callback function structure, we use the c++ function wrapper technology, all the functions that need to be executed through the function callback way placed together, constitute the registFuncTab structure. The callback functions are divided into three categories, namely, data flow related callback functions, scheduler related callback functions and protocol management related callback functions. The details are shown in the following table:

Table 3 Some key variables in the registFuncTab structure

classification	variable name	meaning
Data flow related callback functions	receive	Receive function for handling data frames
	send	Transmit function for handling the amount of data
Scheduler-related callback functions	QosTimerUpdate	Time event pair update scheduler function
	QosPacketUpdate	Data frame event to update scheduler function
	QosPacketCheck	Packet compliance check function
	QosQueueCheck	Queue compliance check function
	SchedToolsNotify	Scheduler event notification functions
Protocol management related callback functions	config	Functions to configure this protocol object
	notify	Functions to notify other objects for use with this protocol object

Its specific use is explained in other sections.

2.4 timerType

Timer basic structure. Similar to the reason for creating the netifItem structure, since different platforms perform event timing in different ways and functions, the timerType structure has been defined in order to harmonize this difference. This structure is used to provide a description of the timed event and is platform independent. It is described in the following table:

Table 4 some key variables in the timerType structure

variable name	meaning	type
thisID	ID of the timed event	int
deviceId	ID of the device to which the timed event belongs	int
setTime	Logical time of the timed event	double
actSetTime	Platform time at which the timing event occurred	double
protTimerId	Protocol timer ID to which the timing event belongs	int
protObjPtr	Pointer to the protocol that set the event	void *
cbFunc	The callback function to be executed when the event occurs.	function<void(timerType*)>
privDataPtr	Some private variables passed by the event	void *

Its specific use is explained in other sections.

3 Platform Adapter

Platform adapters are platform-specific and platform-independent layers. It contains the following set of functions that need to be customized by the user according to the platform.

Table 5 User-defined functions

classification	name	clarification
Data Frame Related Functions	netifList	A list of the platform's data interfaces;
	createNetif()	This function is registered by the user to complete the mapping of platform data interfaces to platform data logic interfaces;
	plantformRx()	The function is registered by the user to complete the data reception and processing of different platforms. Unify the data to be processed as packetStruct, and then submit it to the platform-independent layer;
	plantformTx()	According to the mapping rules in netifItem with the data frame of packetStruct, send the data frame to the corresponding platform network interface.
Time Related Functions	_APIplatformSetTimer()	Platform-independent timed time setting functions;
	_APIplatformCancelTimer()	Platform-related timed event cancel functions;
	platformGetCurTimerSec()	Platform-related functions for obtaining platform time;
User-defined	writeDriver()	User-defined operation functions
	readDriver()	User-defined operation functions

The main function of the platform adapter is to complete the customization of the above functions according to different platforms.

4 Platform-independent layer

The platform adapter completes the platform-independence of both data frame sending and receiving and event timer. On the basis of the platform adapter, we have constructed a unified handler function for data frame sending and receiving and event timer processing. They correspond to the classes PIL and PIL Timer respectively.

4.1 PIL class

After processing by the platform adapter, data frames are uniformly described by packetStruct and network data ports are uniformly described by portNum (logical port). On this basis the PIL class implements platform-independent send and receive functions.

Table 5 PIL function

name	clarification
receive	Based on the packetStruct and the protocol tree, deliver the packetStruct to the corresponding protocol entity;
send	Based on packetStruct and netifItemList, send the data frame to the corresponding platform adapter's plantformTx function;

4.2 PILTimer class

The PILTimer class implements a set of platform-independent timed event-related functions using a red-black tree data structure. These functions include:

Table 6 PILTimer function

name	clarification
createTimer	Used to create platform-independent timer objects; different protocol entities will use this function to register a series of private variables;
setTimer	for completing the setup of timed events;
cancelTimer	for canceling timed events;
platformCbfunc	for processing timed events;

5 Protocol Templates and Protocol Trees

5.1 Agreement templates

We provide protocolGen protocol template generation tool, through the CMD window to enter the name of the protocol can be completed to generate the protocol code template class. The protocol template class is divided into several important parts that the user needs to verify and modify after generating the template class. When using the protocolGen tool, it will generate 'xxx_template.cpp', 'xxxCommon_template.cpp' and ' xxx_template.h' files.

①Macro definitions section

Table 7 Macro definitions

Macro Definition Type	clarification
XXX_TYPE	Requires user specification and is used to define the type of this protocol, e.g. this field is 0x0800 for IPv4 protocols;
PACKET_TYPE_DEMO	Requires user specification, if present, to define all data frame types used by the current protocol, respectively;
DEMO_TIMEOUT_EVENT	If the current protocol requires a number of different types of timers, user specification is required; timers of the same type have the same event handling function;
DEMO_CIDBASE1	If the user needs that multiple instances of a particular type of timer exist, it is recommended that the macro be defined;
TIMERIDMAX	If the user needs, a certain type of timer exists more than one instance, then it is recommended to define this macro; TIMERIDMAX indicates the maximum number of instances of this type of timer;
XXX_MAINSTATE_0	User-specified is required for overall control of the current protocol; overall control means that the protocol can be reset, initialized, etc. as a whole;
XXX_S1	need to be specified by the user, the state is the state in the state machine of the normal operation of the protocol;

②Data structure component

The 'xxx_template.h' file generated by the protocolGen tool includes the following types of macro definitions:

Table 8 Data structure description

data structure type	clarification
XXXMsg	Requires user specification and is used to define the individual fields of the data frame;
XXX_timerDescriptor	A user-specified requirement that each timer instance corresponds to a XXX_timerDescriptor object that holds some private variables;
class XXX	XXX protocol structure, which contains declarations of XXX protocol frame functions, and the user can add customized variables and functions to this class;
XXXportStruct	XXX protocol port structure, customized according to actual needs, some protocols will exist some port private functions;
XXX_TYPE	Need to be specified by the user, used to define the type of this protocol, for example, IPv4 protocol the field is 0x0800;

③Description of Basic Main Functions and Variables

The 'xxxCommon_template.cpp' file generated by the protocolGen tool includes the following variables and functions as public basic variables and functions, which usually do not need to be modified, or only need to be modified briefly:

Table 9 Key Variables and Function Descriptions

name	clarification
XXX(protocolTree* protTreeObjPtr);	Constructor, mainly to complete the initialization operation;
_sonProtObjPtrList	Used to hold the child protocol object of this protocol;
_fatherProt	for pointing to the parent protocol object of this protocol;
_broProtAPIList	for storing the sibling protocol object of this protocol; (not used yet)
_funcTab	An interface function for holding objects of this protocol;
_protObjPtrList	for storing objects of this protocol;
_notifyAPIList	for storing a list of functions to be notified by this protocol object;
_registProtConfigNotify	Register_notifyAPIList;
registProt	No longer used;
Send	The send interface for the protocol;
Receive	The receive interface for the protocol;
config	The configuration interface for the protocol;
notify	The protocol's notification interface;
timerList	The protocol entity's timer list;
RegistTimer	The timer registration function;
CreateTimerEvent	No need to care;
SetTimerEvent	Functions for setting timed events;
StopTimer	No need to care;
DeleteTimer	Timer delete functions;
GetTimerEvent	Timer event callback functions;
mainStateExecute, mainStateSet	State control state machine functions;

④Customized Functions and Variables

The 'xxx_template.cpp' file generated by the protocolGen tool includes the following variables and functions that need to be customized by the user:

Table 10 Key Variables and Function Descriptions

name	clarification
run	Protocol entry function;
reset	Protocol reset function;
init	Protocol initialization functions;
TimerEventHandlerAPI	APIs for the protocol to handle timing events;
ReceiveHandlerAPI	APi for protocol handling of received data frames;
NEWARCHStateSet NEWARCHStateExecute	Protocol state machine related functions;

5.2 Protocol tree

The main purpose of the protocol tree is to create a protocol object, it will automatically create the protocol's parent protocol object;

Step1: create a protocol tree, the structure of the protocol tree is, in fact, a tree consisting of parent protocols and sub-protocols, and inside each protocol is stored an object called protocolAbstract; the protocolAbstract object maintains a chained table consisting of all the instances belonging to the current protocol;

Step2: when you want to insert a protocol, first create a protocol object, and then call insertProtInst function, the function according to the needs of interpretation of the protocol object is ① to create a new parent protocol object; ② insert an existing parent protocol object;

Step3: Whether inserting a newly created parent protocol object or inserting an existing parent protocol object, the parent protocol object pointer should be passed to the child protocol object;

Step4: Whether inserting a newly created parent protocol object or inserting an existing parent protocol object, the method that should be used in the process of the parent protocol object looking for the child protocol object is; the child protocol object is correctly inserted into the _sonProtObjPtrList in the parent protocol object;

Table 11 Key Variables and Function Descriptions

name	clarification
_ipdlPtr	Protocol-shared PIL pointer;
_IPDLtcPtr	Protocol-shared PIL Timer pointer;
protocolAbstract	A data structure describing a protocol at a particular layer;
protTree	Protocol tree;
insertPortTree()	Creating a protocol tree branch;
insertProtInst()	Inserting a protocol entity for a node in the protocol tree;
registProtObj()	Completing the binding of parent-child protocol entities;

Subsequently, when the protocol needs to call a sub-protocol, it only needs to retrieve the protocol's _sonProtObjPtrList.

6 Scheduler

6.1 networkSchedulerTool

For ease of use, we have created a network scheduling tool that allows the user to register customized scheduling algorithms according to the parameters we have defined. The network scheduling tool will automatically complete the time update related operations.

Table 12 Key Variables and Function Descriptions

name	clarification
sched	Contains three input parameters: struct packetStruct* pkptr:the data frame to be adjudicated vector<vector<packetStruct*>>& pkptrQueue: the queue associated with data frame pkptr. vector<bool >& queueFlag: the current status of the queue

	associated with the data frame pkptr. Inside the function, the scheduler registered by the user is called and the time of the next event is returned. 
schedPacketUpdate()	Contains three input parameters: struct packetStruct* pkptr:the data frame to be adjudicated vector<vector<packetStruct*>>& pkptrQueue: the queue associated with data frame pkptr. vector<bool >& queueFlag: the current status of the queue associated with data frame pkptr. This function is used in conjunction with the sched() function to update the user's scheduler based on the scheduling results.
netSchTools_GetTimerEvent()	networkSchedulerTool's timed callback function, when a certain event occurs, it will call the callback function registered by the user; 

6.2 xxxArbiter

Users can use the genArbiter tool to get a customized scheduler template, which in turn completes the customized scheduler.

Table 13 Key Variables and Function Descriptions

name	clarification
qosQueueCheck	vector<bool >& queueFlag: current status of the queue associated with the data frame pkptr. Inside the function, the user-registered scheduler is called and the time of the next event is returned. User-defined, mainly used to be afraid of whether there is a packet in that queue that can be scheduled. This function is used in conjunction with

	the sched() function to update the user's scheduler based on the scheduling results.
qosPacketUpdate ()	Contains three input parameters: struct packetStruct* pkptr:the data frame to be adjudicated vector<vector<packetStruct*>>& pkptrQueue: the queue associated with data frame pkptr. vector<bool >& queueFlag: the current status of the queue associated with data frame pkptr. User-defined, mainly used to realize, whether the data packet pkptr can be inserted into the queue pkptrQueue. This function is used in conjunction with the sched() function to update the user's scheduler based on the scheduling results.

7 Fault injection

In order to facilitate the testing of protocols and implementation code, it is usually necessary to build a variety of failure scenarios, so this paper designs a fault creation tool, each device has a corresponding fault descriptor XML, the user through the fault descriptor XML way to illustrate the failure of each device, a fault descriptor includes: ① the port where the failure occurs; ② the type of the protocol where the failure occurs; ③ the time when the failure starts; ④ the time when the failure ends; ⑤ the type of the failure; ⑥ the value that needs to be modified. A fault descriptor includes: ① the port where the fault occurred; ② the protocol type where the fault occurred; ③ the time when the fault started; ④ the time when the fault ended; ⑤ the type of the fault; ⑥ the value that needs to be modified.

The fault injection protocol layer of each device reads the fault descriptor XML of its own device at the beginning of the simulation and initializes the fault event list, which stores the descriptors of each fault in the order of the start time of the fault, and Fig. 6 shows the schematic diagram of the fault event list.

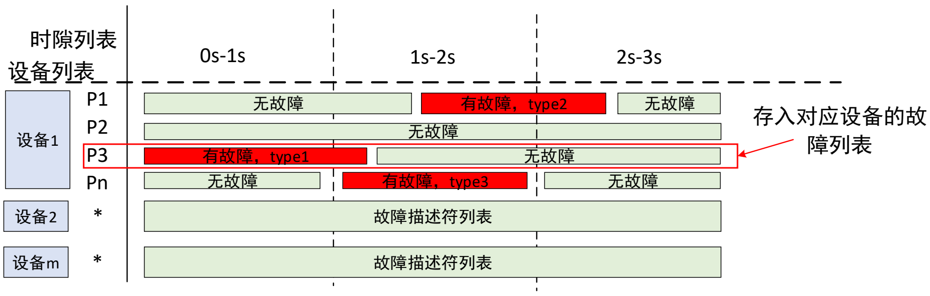


Figure 6 Fault Event List

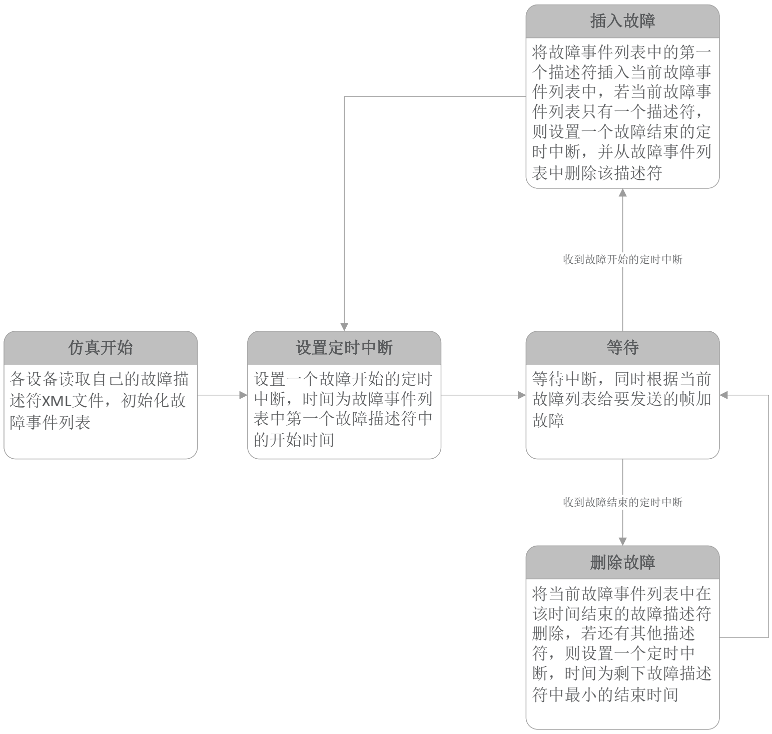


Figure 7 Fault Insertion Deletion Process

The fault injection process is shown in Figure 7, and the specific steps are described as follows: Step 1: After completing the initialization, the fault event exists in the form of a descriptor in a fault descriptor queue maintained by the fault injection layer of each port of each device.

Step 2: By means of setting a timer, at a fault start moment, the corresponding fault descriptor is taken out of the fault descriptor queue and put into the current fault descriptor queue.

Step 3: At the same time, by setting a timer, at the end of the fault, the corresponding descriptor is removed from the current fault descriptor queue.

Step 4: When the fault injection layer receives a data frame from the upper layer protocol, the fault injection layer traverses the current fault descriptor queue. If there is a fault descriptor for the current data frame protocol, the data frame is modified or delayed according to the descriptor.

8 Examples of instructions for use

8.1 Basic workflow

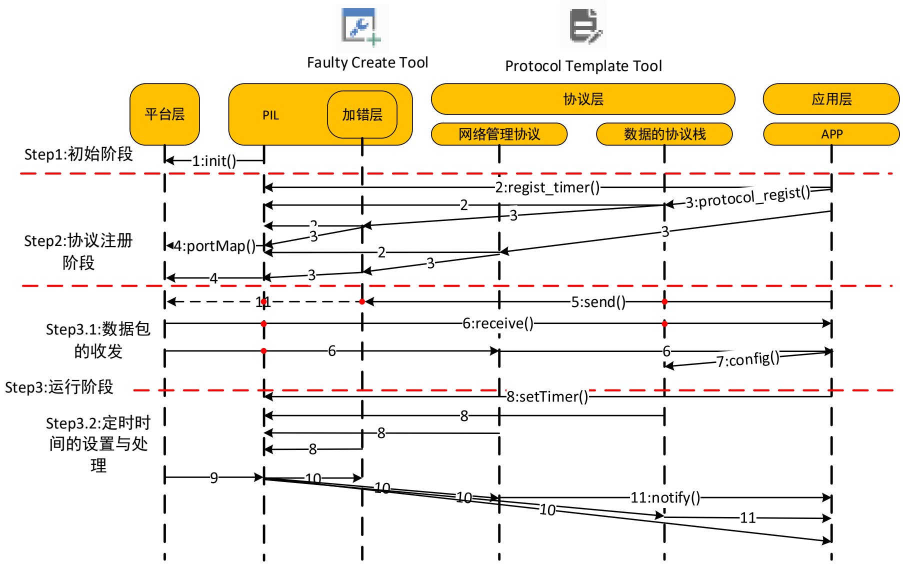


Figure 8 Business Flow Chart

In this subsection we explain the workflow of the code reusability platform. The workflow of the code reusability framework designed in this paper is shown in Fig. As shown in Fig. 5, the flow can be roughly divided into three phases.

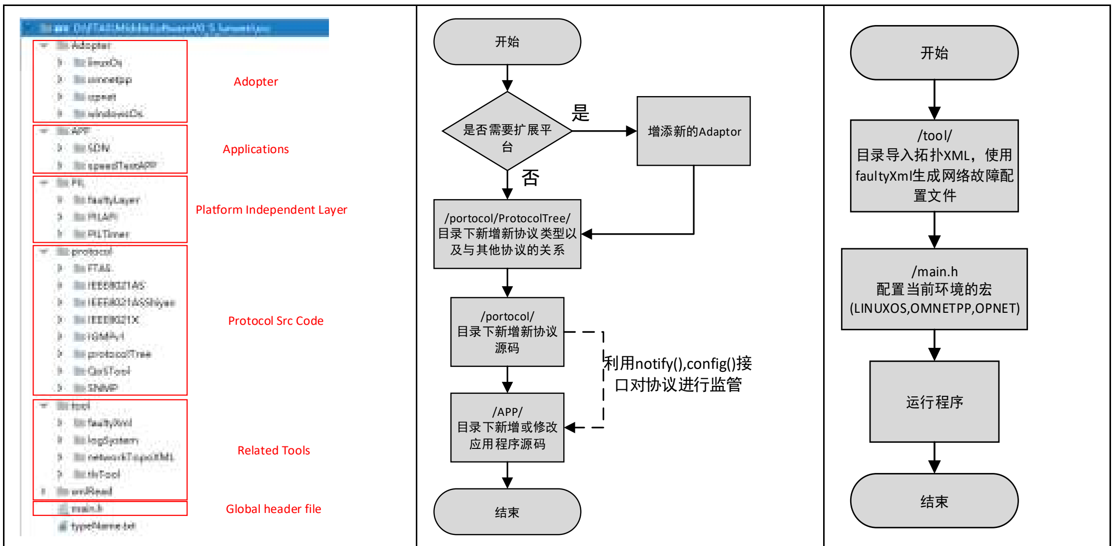
Initialization phase: In this phase, the PIL layer will select different adapters according to the underlying conditions (macro definitions), and complete the creation of ports under different platforms, the mapping of actual ports to logical ports of the PIL, and the initialization of platform timers;

Protocol initialization phase: this phase completes the creation of protocol objects, the binding of upper and lower layer protocols and the binding function of protocol timers. For example, if the APP layer creates a network management class protocol, the protocol will be automatically created and registered into its lower layer protocol. If there is a timer operation in the network management protocol, the network management protocol will also be created and registered as a timer object.

Run phase: The run phase is mainly divided into data flow and timer events. For the data flow, in the sending direction, the send function of the lower layer protocols will be called one by one according to the registration of the previous protocol tree. When passing through the fault layer, the data flow will be selectively forwarded to the PIL layer ① normally; ② delayed and forwarded to the PIL layer; ③ discarded according to the mapping relationship of the PIL layer and finally sent to the physical port. The receiving direction is similar. In addition, if the configuration management data is received, the config function of the corresponding protocol object will be called by the APP layer to complete the management of the protocol. For timed time, the PIL layer maintains a set of discrete time event list, and the registration of timed events is completed through the setTimer() method of the timer registered in each protocol object, i.e., a timed event is added to the discrete time event list of the PIL layer. When a certain moment is reached, the callback function of the corresponding timer object is executed, which in turn processes the timed event. The network management protocol can periodically notify some status information like the APP through the notify() method.

8.2 Basic usage

In this subsection, we put ourselves in the user's perspective and explain how the user can use the platform to realize the authentication of cross-platform network protocols. As shown in Figure 6(a) below, the project catalog of the reusable framework, including adapters, platform-independent layers, protocol source code, applications, and related tools, has several parts. Figure 6(b) adds the flow of adding a new protocol by the user, first determining whether the user needs to add a new adapter. Then the user needs to specify the type of the added protocol and the relationship with other protocols (e.g., the type of the lower layer protocol of the added protocol) in the protocolTree. Add the source code of the protocol in the /protocol/ folder. And write a test program for this protocol under the /APP/ folder. Figure 6(c) shows how to run the program, the user needs to first import the topology description XML file and the fault description XML file, followed by macro definitions in main.h to indicate the adaptor currently being used.



(a) Reusable platform engineering structure	(b) Adding new protocol processes	(c) Running processes

Figure 6 Description of the user's utilization flow

8.3 TicToc Protocol Example

8.3.1 Introduction to the TicToc Protocol

8.3.2 Customized Platform Adopter

This is customized to suit the situation.

8.3.3 Customized Protocol Trees

Open the “templateGen/genProtocolTree” folder, and edit the protocols.xml file in the folder according to the actual protocol layers. Among them, IPDL and faultyLayer are the basic layers of PIL, we don't need to modify them. ticToc should be a sub-layer of faultyLayer, so we add a new tictoc protocol layer, add:

```xml
<protocol>
    <protName>tictocLayer</protName>
    <protType>tictocLayer_TYPE</protType>
    <fatherProtName>faultyLayer</fatherProtName>
    <fatherProtType>faultyLayer_TYPE</fatherProtType>
</protocol>
```

Note that when using the Protocol Generation Template tool to generate the tictoc protocol layer, the protocol layer name and protocol type name should be consistent with those in the protocol tree.

Save and run “ genProtocolTree.exe” to get the “protocolTree.ex.cpp” code. Copy this code to the folder “ src\protocol\protocolTree ” and replace the original file.

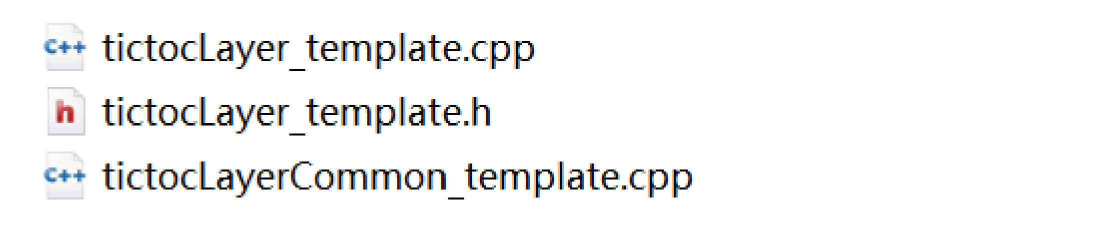
8.3.4 Customizing the tictoc protocol

①Customize the tictoc protocol template

Open “templateGen.exe” in the “templateGen\protocolGen” folder, enter the name of the protocol class, and note that the name should be consistent with that in the protocol tree.

```
D:\FTAS\MiddleSoftwareV1_3\templateG
请输入协议类名: tictocLayer
```

The tictoc agreement template is automatically generated after enter confirmation.



The "tictocLayer_template.cpp" is the user-defined code framework. Where "tictocLayer_template.h" is the tictoc protocol header file. The "tictocLayerCommon_template.cpp" is the auto-generated code framework, including timer, packet sending and receiving some basic functions.

Then create a new tictocLayer folder in the "src\protocol" directory and copy the production files to this folder. And declare the path of the protocol in "main.h".

② Simple modifications to faultyLayer

Because the PIL framework receives packets through the faultyLayer, we need to find the faultyLayerReceive() function in faultyLayer.cpp, and add the following to its switch statement.

③ Modify tictocLayer_template.h

We make a simple modification to the header file.

```
//协议类型
#define tictocLayer_TYPE 0x7777
//协议数据帧类型
#define PACKET_TYPE_tictoc 0x77
//定时器事件
#define Send_TIMEOUT_EVENT 0
//定时器步长
#define TIMERIDMAX 90
//定时器基地址
#define send_CIDBASE1 0
#define DEMO_CIDBASE2 1000

#define tictocLayer_INIT 0
#define tictocLayer_WAITPACKET 1
#define tictocLayer_SENDPACKET 2

#define tictocLayer_MAINSTATE_0 0
#define tictocLayer_MAINSTATE_1 1
#define tictocLayer_MAINSTATE_2 2
#define tictocLayer_MAINSTATE_3 3
```

The protocol type, protocol frame format, timer event and state name are modified according to the actual requirements. We only do the example here.

④ Modify tictocLayerCommon_template.cpp

This function is basically generated automatically by the script, but we still need to modify it slightly. Pass the correct parent protocol type to the send function:

```
/*****************************
**函数名: Send
**函数功能: 发送函数
*******************************/
ssize_t tictocLayer::Send(void *dataPtr,int port){
    //1.本层进行一系列处理, 一般是封装报文
    //1.1有可能会查询应该发往哪个端口

    //2.向下一层传递
    return static_cast<class faultyLayer*>(_fatherProt)->_funcTab->send(dataPtr,port);
}
```

It is also necessary to modify the relationship of the parent-child agreement according to the actual situation:

```
/*****************************
**函数名: Send
**函数功能: 发送函数
*******************************/
ssize_t tictocLayer::Send(void *dataPtr,int port){
    //1.本层进行一系列处理, 一般是封装报文
    //1.1有可能会查询应该发往哪个端口

    //2.向下一层传递
    return static_cast<class faultyLayer*>(_fatherProt)->_funcTab->send(dataPtr,port);
}
```

It is also necessary to modify the relationship of the parent-child agreement according to the actual situation:

```
    //3.bind brother list
    protTreeObjPtr->insertProtInst(tictocLayer_TYPE,faultyLayer_TYPE,static_cast<void*>(this),NULL);
    //4._protObjPtrList
```

⑤ Modify tictocLayer_template.cpp

Modify the init() function according to the requirement, because tictoc only needs one timer, so we only create one timer.

According to the protocol, write the state machine of the protocol:

```
/****************************
**函数名: tictocLayerStateSet
**函数功能: 主状态机用于设置下一状态的函数
******************************/
int tictocLayer::tictocLayerStateSet(int curState){

    int nextState = curState;

    switch(curState){
        case(tictocLayer_INIT):
            nextState = tictocLayer_WAITPACKET;
        break;
        case(tictocLayer_SENDPACKET):
            nextState = tictocLayer_WAITPACKET;
        break;
        case(tictocLayer_WAITPACKET):
            if(rcvPacketFlag)
                nextState = tictocLayer_SENDPACKET;
            else
                nextState = tictocLayer_WAITPACKET;
        break;
        default:break;
    }

    return nextState;
}

/****************************
**函数名: tictocLayerStateExecute
**函数功能: 主状态机执行函数
******************************/
void tictocLayer::tictocLayerStateExecute(int curState){
    printf("Enter tictocLayerStateExecute ,cur tictocLayerState = %d \r\n",curState);
    tictocLayerNextState = tictocLayerStateSet(curState);
    switch(curState){
        case(tictocLayer_INIT):
            rcvPacketFlag = false;
            userSendPacket();
        break;
        case(tictocLayer_SENDPACKET):
            rcvPacketFlag = false;
            SetTimerEvent(send_CIDBASE1+0,userClockCurPit+1.0);
        break;
        case(tictocLayer_WAITPACKET):
        break;
        default:break;
    }


    tictocLayerState = tictocLayerNextState;
    if(tictocLayerState == curState)
        return;
    tictocLayerStateExecute(tictocLayerState);
}
```

The user writes the packet sending function according to the demand, we first define the frame format, because tictoc is relatively simple, so the frame format is also relatively simple.

```
struct tictocLayerHeader
{
    UInteger8 dMac[6];
    UInteger8 sMac[6];
    UInteger16 etherType;
};

struct tictocLayerMsg
{
    tictocLayerHeader header;
    UInteger8 subtype;
    UInteger8 payload[100];
};
```

Then the user needs to define his own send function:

```
void tictocLayer::userSendPacket(){
        //1.创建基本结构体
        struct packetStruct* pkptr = (struct packetStruct*)malloc(sizeof(struct packetStruct));
        memset(pkptr,0x0,sizeof(struct packetStruct));

        struct tictocLayerMsg* tictocPkptr = (struct tictocLayerMsg*)malloc(sizeof(struct tictocLayerMsg));
        memset(tictocPkptr,0x0,sizeof(struct tictocLayerMsg));
        //2.赋值
        tictocPkptr->header.dMac[0] = 0x01;
        tictocPkptr->header.dMac[1] = 0x80;
        tictocPkptr->header.dMac[2] = 0xC2;
        tictocPkptr->header.dMac[3] = 0x00;
        tictocPkptr->header.dMac[4] = 0x00;
        tictocPkptr->header.dMac[5] = 0x0E;

        tictocPkptr->header.sMac[0] = deviceId;
        tictocPkptr->header.sMac[1] = domainId;
        tictocPkptr->header.sMac[2] = 0x33;
        tictocPkptr->header.sMac[3] = 0x44;
        tictocPkptr->header.sMac[4] = 0x55;
        tictocPkptr->header.sMac[5] = 0xFF;

        tictocPkptr->header.etherType = static_cast<UInteger16>(htons(tictocLayer_TYPE));

        tictocPkptr->subtype = PACKET_TYPE_tictoc;
        for(int i = 0;i<64;i++)
            tictocPkptr->payload[i] = static_cast<UInteger8>(i);
        //3.packetStruct赋值
        pkptr->frameSize = sizeof(struct tictocLayerMsg);
        pkptr->type = tictocLayer_TYPE;
        pkptr->subtype = PACKET_TYPE_tictoc;
        memcpy(pkptr->frameBuf,tictocPkptr,pkptr->frameSize);
        pkptr->portNum = 1;
        //4.调用发送函数
        Send(static_cast<void*>(pkptr),1);
        //5.释放
        free(tictocPkptr);
        free(pkptr);
}
```

Then the user needs to customize what the protocol should do when the message is received:

```
/****************************
**函数名：ReceiveHandlerAPI
**函数功能：接收接口函数
*****************************/
void tictocLayer::ReceiveHandlerAPI(struct packetStruct* pkStruct){
    printf("ReceiveHandlerAPI \r\n");
    int domainId=0;
    int deviceId = pkStruct->deviceId;
    int frameSize = pkStruct->frameSize;

    switch(pkStruct->subtype){
        case PACKET_TYPE_tictoc:
            //我们只做简单打印
            DEBUGEN(EN)
            {
                printf("-------frameSize = %d----\r\n",frameSize);
                for(int i=0;i<frameSize;i++){
                    if(i%16 == 0)
                        printf("\r\n");
                    printf(" 0x%x ",pkStruct->frameBuf[i]);
                }
                printf("\r\n");
            }
            //修改相关变量值
            rcvPacketFlag = true;
            //继续状态执行
            mainStateExecute(mainState);
        break;
        default:break;
    }
}
```

At the same time, if the message needs to be passed further, the user can also traverse the _sonProtObjPtrList traversal by himself for cross-layer passing of the data message. An example is as follows:

```
    void* sonProtObjPtr = NULL;

    if( (!FindMapEmpty(_sonProtObjPtrList,faultyLayer_TYPE)) && _sonProtObjPtrList[faultyLayer_TYPE].size()!=0){
        sonProtObjPtr = _sonProtObjPtrList[faultyLayer_TYPE][0];
    }
    else{
        printf("son prot Not found = %x \r\n",faultyLayer_TYPE);
        return;
    }
    //2.
    static_cast<faultyLayer*>(sonProtObjPtr)->_funcTab->receive(dataPtr,port);
```

The timer callback function is customized because the timed send is implemented in tictoc and the timer is set, so we also need to customize the timed event handler function. The so-called timed event handler function is an action that the protocol needs to perform when a certain preset timed event occurs.

```
/****************************
**函数名：tictocLayer_TimerEventHandlerAPI
**函数功能：定时事件的回调函数
*****************************/
void tictocLayer::TimerEventHandlerAPI(struct tictocLayer_timerDescriptor * tictocLayer_timerDescriptor){
    int thisDomain ,thisPort;
    thisPort = tictocLayer_timerDescriptor->thisPort;
    thisDomain = tictocLayer_timerDescriptor->thisDomain;
    //下面是用户逻辑
    switch(tictocLayer_timerDescriptor->subprotocol){
    case(Send_TIMEOUT_EVENT):
        //1.执行定所有发生这个事件所需要的工作
        cout<<"Send_TIMEOUT_EVENT"<<endl;
        userSendPacket();
        //2.调用状态机
        mainStateExecute(mainState);
        break;

    default:break;
    }
}
```

8.3.5 Operation

8.3.5.1 Based on OPNET Framework

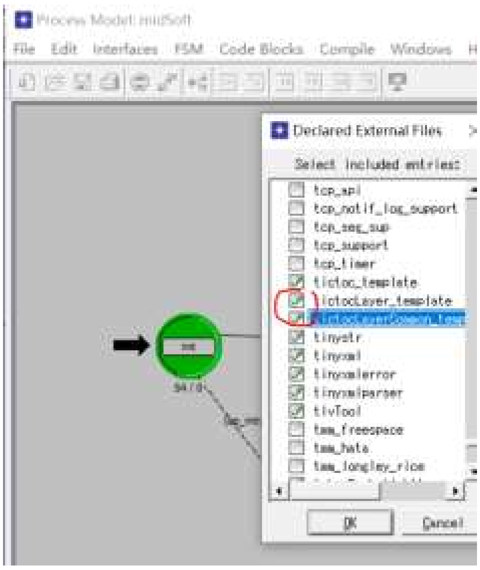
①Set up the platform macros

You need to declare the platform macros in main.h:

```
using namespace std;
#define OPNET
```

② Create OPNET nodes and confirm the connection relationship

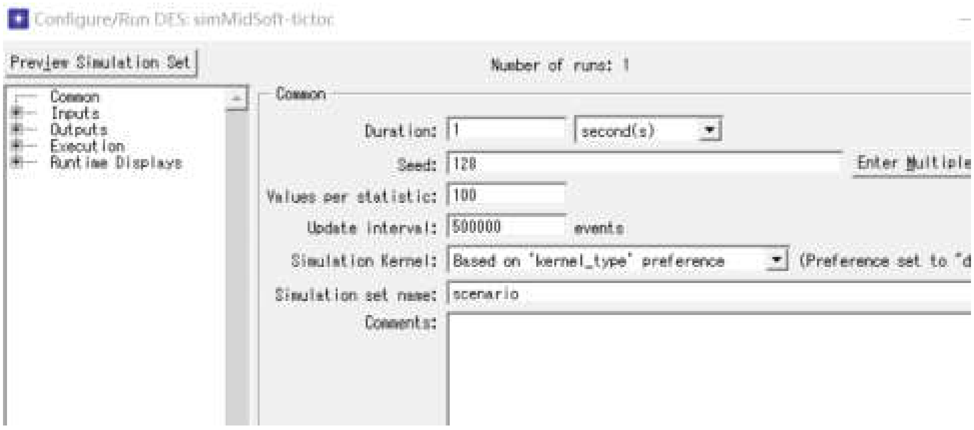
③ Add tictocLayer module to OPNET



④ Create protocol entities

⑤Run

Set the run time and you are ready to start.




```
packet Send ok
Enter tictocLayerStateExecute ,cur tictocLayerState = 1
```

Packet delivery complete

```
(ODB 14.5.A: Event)

  * Time   :  0.00000092 sec, [0s . 000ms 000us 920ns]
  * Event  :  execution ID (11), schedule ID (#11), type (stream intrpt)
  * Source :  execution ID (7), top.es_1.pr_1 [Objid=84] (pt-pt receiver)
  * Data   :  instrm (1), packet ID (0)
  > Module :  top.es_1.p_0 [Objid=114] (processor)


ODB> next
receive Packet
Receive
ReceiveHandlerAPI
subType = 77
-------frameSize = 115----

 0x1  0x80  0xc2  0x0  0x0  0xe  0x1  0xa  0x33  0x44  0x55  0xff  0x77  0x77  0x77  0x0
 0x1  0x2  0x3  0x4  0x5  0x6  0x7  0x8  0x9  0xa  0xb  0xc  0xd  0xe  0xf  0x10
 0x11  0x12  0x13  0x14  0x15  0x16  0x17  0x18  0x19  0x1a  0x1b  0x1c  0x1d  0x1e  0x1f  0x20
 0x21  0x22  0x23  0x24  0x25  0x26  0x27  0x28  0x29  0x2a  0x2b  0x2c  0x2d  0x2e  0x2f  0x30
 0x31  0x32  0x33  0x34  0x35  0x36  0x37  0x38  0x39  0x3a  0x3b  0x3c  0x3d  0x3e  0x3f  0x0
 0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0
 0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0
 0x0  0x0  0x0
Enter mainStateExecute ,cur mainState = 1
Enter tictocLayerStateExecute ,cur tictocLayerState = 1
Enter tictocLayerStateExecute ,cur tictocLayerState = 2
tictocLayer_SetTimerEvent usrClock = 9.2e-007
size = 0,timerEventList.begin() = 4ddd140
!!!setTime = 1
size = 1,timerEventList.begin() = 4c584e8
Enter tictocLayerStateExecute ,cur tictocLayerState = 1
```

Packet reception and printing

```
 tictocLayer_GetTimerEvent ! type = 0 ,deviceId = 2,domainId = 0,portId = -1,timerId = 0,setTime = 1.000001.
Send_TIMEOUT_EVENT
faultyLayer::faultyLayerSend
frame size = 73
pkptr==nullptr? 0,port = 1
faultyDescriptorCur size = 0,protType = 0
IPDL send()port = 1
this = 4d7d1a8
IPDL send()sockFd = 1
IPDL this = 4d7d1a8 ,socketfd = 1
-------txSize = 115----

 0x1  0x80  0xc2  0x0  0x0  0xe  0x2  0x0  0x33  0x44  0x55  0xff  0x77  0x77  0x77  0x0
 0x1  0x2  0x3  0x4  0x5  0x6  0x7  0x8  0x9  0xa  0xb  0xc  0xd  0xe  0xf  0x10
 0x11  0x12  0x13  0x14  0x15  0x16  0x17  0x18  0x19  0x1a  0x1b  0x1c  0x1d  0x1e  0x1f  0x20
 0x21  0x22  0x23  0x24  0x25  0x26  0x27  0x28  0x29  0x2a  0x2b  0x2c  0x2d  0x2e  0x2f  0x30
 0x31  0x32  0x33  0x34  0x35  0x36  0x37  0x38  0x39  0x3a  0x3b  0x3c  0x3d  0x3e  0x3f  0x0
 0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0
 0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0  0x0
 0x0  0x0  0x0
port = 1,sendDelayCum = 0.0000000000
packet Send ok
```

Reach timed moment, resend

8.4 Simple logging system: observing the frequency of sending Tictoc protocol packets

We have customized a unified logging class for the system, which is used to save the data we care about to a file for easy analysis. We have defined a simple logging tool, the source code of which is located in the “src/tool/logSystem” folder. To use it, we first create our own logging tool in the protocol class:

```
protocolTree* _protTreeObjPtr;
class IPDLTimerClass* IPDLtcPtr;
class IPDL* ipdlAPI;
class logSystem* logToolPtr;
class logSystem* mylogToolPtr;

int deviceId;
int domainId;
```

and initialization is done in the constructor

```
//协议实例初始化
void tictocLayer::init(){
    //1.通用变量初始化
    mylogToolPtr = new logSystem(deviceId,"tictocSendStatLog");mylogToolPtr->setSaveFileEnable();
    //1.1根据需求创建定时器
    RegistTimer(tictocLayer_TYPE,Send_TIMEOUT_EVENT,send_CIDBASE1+0,0.0,-1);

    //1.2变量初始化
    int ttPortNum = enNumberPorts;
```

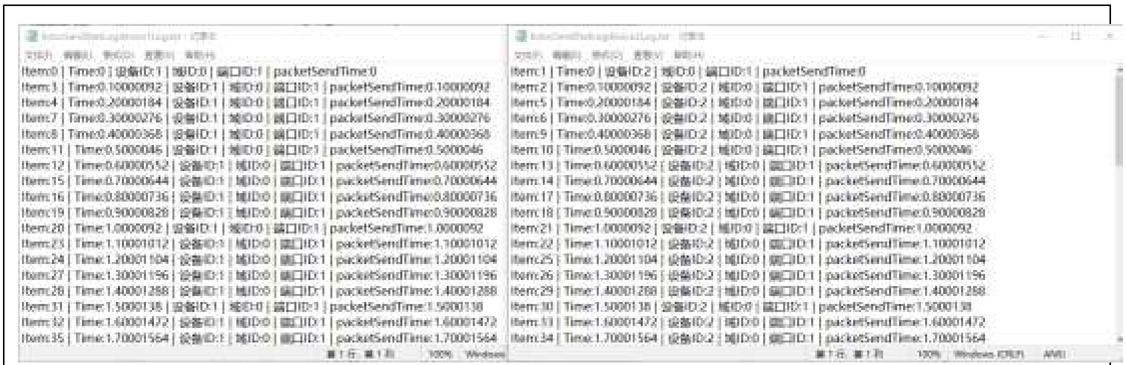
When logSystem is created, the first function is the device ID of the current device, the second parameter is the name of the file to be generated, and the final statistic information will be saved in the file “tictocSendStatLog+deviceId.txt”. Note that after creating the logSystem object, you need to enable the file saving function.

Then we can add the statistic code:

```
95  void tictocLayer::userSendPacket(){
96      //1.创建基本结构体
97      struct packetStruct* pkptr = (struct packetStruct*)malloc(sizeof(struct packetStruct));
98      memset(pkptr,0x0,sizeof(struct packetStruct));
99
100     struct tictocLayerMsg* tictocPkptr = (struct tictocLayerMsg*)malloc(sizeof(struct tictocLayerMsg));
101     memset(tictocPkptr,0x0,sizeof(struct tictocLayerMsg));
102     //2.赋值
103     tictocPkptr->header.dMac[0] = 0x01;
104     tictocPkptr->header.dMac[1] = 0x80;
105     tictocPkptr->header.dMac[2] = 0xC2;
106     tictocPkptr->header.dMac[3] = 0x00;
107     tictocPkptr->header.dMac[4] = 0x00;
108     tictocPkptr->header.dMac[5] = 0x0E;
109
110     tictocPkptr->header.sMac[0] = deviceId;
111     tictocPkptr->header.sMac[1] = domainId;
112     tictocPkptr->header.sMac[2] = 0x33;
113     tictocPkptr->header.sMac[3] = 0x44;
114     tictocPkptr->header.sMac[4] = 0x55;
115     tictocPkptr->header.sMac[5] = 0xFF;
116
117     tictocPkptr->header.etherType = static_cast<UInteger16>(htons(tictocLayer_TYPE));
118
119     tictocPkptr->subtype = PACKET_TYPE_tictoc;
120     for(int i = 0;i<64;i++)
121         tictocPkptr->payload[i] = static_cast<UInteger8>(i);
122
123     //3.packetStruct赋值
124     pkptr->frameSize = sizeof(struct tictocLayerMsg);
125     pkptr->type = tictocLayer_TYPE;
126     pkptr->subtype = PACKET_TYPE_tictoc;
127     memcpy(pkptr->frameBuf,tictocPkptr,pkptr->frameSize);
128     pkptr->portNum = 1;
129     //4.调用发送函数
130     Send(static_cast<void*>(pkptr),1);
131     mylogToolPtr->logData(IPDLtcPtr->platformGetCurTimerSec(),deviceId,0,1,"packetSendTime",userClockCurPit);
132     printf([illegible]);
133     //5.释放
134     free(tictocPkptr);
135     free(pkptr);
136
```

We added 131 new lines of code, the first parameter represents the system time, the second parameter is the device iD, the third parameter is not used for the time being, the fourth parameter represents the port ID, and the last two parameters are the description of the message to be recorded and the value of the message. To make the effect obvious, we change the packet sending period to 100ms. run the program:

tictocSendStatLogdevice1Log.txt	2025-02-14 16:15
tictocSendStatLogdevice2Log.txt	2025-02-14 16:15



Item:0 | Time:0 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:0
Item:3 | Time:0.10000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:0.10000092
Item:4 | Time:0.20000184 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:0.20000184
Item:7 | Time:0.30000276 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:0.30000276
Item:8 | Time:0.40000368 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:0.40000368
Item:11 | Time:0.5000046 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:0.5000046
Item:12 | Time:0.60000552 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:0.60000552
Item:15 | Time:0.70000644 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:0.70000644
Item:16 | Time:0.80000736 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:0.80000736
Item:19 | Time:0.90000828 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:0.90000828
Item:20 | Time:1.0000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:1.0000092
Item:23 | Time:1.10001012 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:1.10001012
Item:24 | Time:1.20001104 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:1.20001104
Item:27 | Time:1.30001196 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:1.30001196
Item:28 | Time:1.40001288 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:1.40001288
Item:31 | Time:1.5000138 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:1.5000138
Item:32 | Time:1.60001472 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:1.60001472
Item:35 | Time:1.70001564 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:1.70001564

Item:1 | Time:0 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:0
Item:2 | Time:0.10000092 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:0.10000092
Item:5 | Time:0.20000184 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:0.20000184
Item:6 | Time:0.30000276 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:0.30000276
Item:9 | Time:0.40000368 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:0.40000368
Item:10 | Time:0.5000046 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:0.5000046
Item:13 | Time:0.60000552 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:0.60000552
Item:14 | Time:0.70000644 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:0.70000644
Item:17 | Time:0.80000736 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:0.80000736
Item:18 | Time:0.90000828 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:0.90000828
Item:21 | Time:1.0000092 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:1.0000092
Item:22 | Time:1.10001012 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:1.10001012
Item:25 | Time:1.20001104 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:1.20001104
Item:26 | Time:1.30001196 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:1.30001196
Item:29 | Time:1.40001288 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:1.40001288
Item:30 | Time:1.5000138 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:1.5000138
Item:33 | Time:1.60001472 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:1.60001472
Item:34 | Time:1.70001564 | 设备ID:2 | 域ID:0 | 端口ID:1 | packetSendTime:1.70001564

We can see that the sending interval is 1ms.

8.5 Scheduler/Shaper Customization: Adding Token Buckets to Tictoc

Very often, it is not enough to implement simple protocols, but also various algorithms to meet the QoS requirements of network communication. For this reason, we design unified templates to help users perform fast verification of algorithms.

Experiment description: Based on the 8.3 TicToc protocol, we customize a 1packet/s token bucket, which can guarantee that, regardless of the sending interval of tictoc, the packets are sent according to 1packet/s after passing through the token bucket.

First of all, we generate a scheduling/shaping tool template, open the “templateGen\arbiterGen” and run the “templateGen.exe” program:

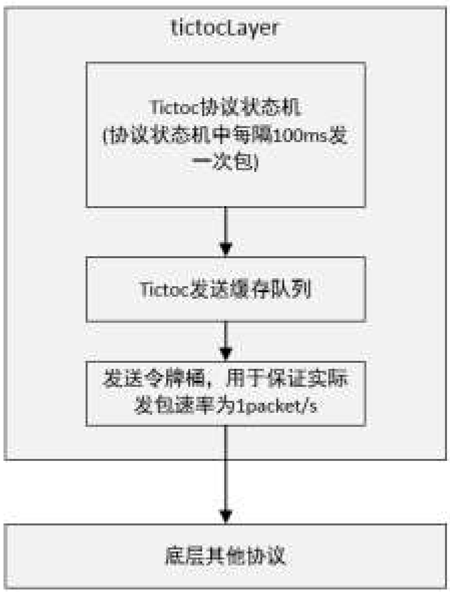
```
D:\FTAS\MiddleSoftwareV1_3\templateGen\arbiterGen\templateGen.exe
请输入调度器类名: tokenBucket
文件已成功保存为 tokenBucket_template.h
文件已成功保存为 tokenBucket_template.cpp

--------------------------------
Process exited after 9.393 seconds with return value 0
请按任意键继续. . .
```

Open the generated template code:

```
#ifndef __tokenBucketARBITER_H
#define __tokenBucketARBITER_H

#include "../../../main.h"

class tokenBucketArbiter{
    public:
        tokenBucketArbiter(class IFDLTimerClass* _IFDLtcPtr);
        ~tokenBucketArbiter(void);
        struct registFuncTab *funcPtr;

        double qosPacketCheck(struct packetStruct* pkptr,vector<vector<packetStruct*>>& pkptrQueue,vector<bool >& queueFlag);
        double qosQueueCheck(vector<vector<packetStruct*>>& pkptrQueue,vector<bool >& queueFlag);
        void qosTimerUpdate();
        void qosPacketUpdate(struct packetStruct* pkptr,vector<vector<packetStruct*>>& pkptrQueue,vector<bool >& queueFlag);

        struct registFuncTab* _funcTab;
        //基本信息
        class IFDLTimerClass* IFDLtcPtr;
        double userClockCurFit;
        double userClockLastFit;
        void updateLocalTimer();
};
```

We only need to customize these functions according to the needs. To analyze the current requirements, we want the output of the tictoc protocol to go through a token bucket, which is used to ensure that no matter how fast the tictoc protocol sends the data, the actual data arriving at the network is still at a rate of 1 packet/s. Therefore, we should implement the logic shown in the following figure:

8.5.1 Creating Queues



Therefore, we first have to implement the send buffer queue as well.

```
    //3.3端口变量
    vector<class tictocLayerportStruct*> portList;
    vector<vector<packetStruct*>>& sendPacketQueueBuffers;

};
```

```
//协议实例初始化
void tictocLayer::init(){
    //1.通用变量初始化
    mylogToolPtr = new logSystem(deviceId,"tictocSendStatLog");mylogToolPtr->set
    QUEUEBUFFERSINIT(sendPacketQueueBuffers,1);

    //1.1根据需求创建定时器
    RegistTimer(tictocLayer_TYPE,Send_TIMEOUT_EVENT,send_CIDBASE1+0,0.0,-1);
```

For simplicity, we added the send buffer queue group directly to the tictocLayer class. By queue group, we mean that we can have more than one queue at the same time. For this experiment, the fact that we only need one queue is enough.

8.5.2 Creating the QoS Tool

We have implemented the networkSchedulerTool class, which actively manages user-defined schedulers/shapers, so we have to create the networkSchedulerTool class first.

```
    //3.3端口变量
    vector<class tictocLayerportStruct*> portList;
    vector<vector<packetStruct*>>& sendPacketQueueBuffers;
    networkSchedulerTools* sendQosTool;

};
```

```
//协议实例初始化
void tictocLayer::init(){
    //1.通用变量初始化
    mylogToolPtr = new logSystem(deviceId,"tictocSendStatLog");mylogToolPtr->setSaveFileEnabl
    QUEUEBUFFERSINIT(sendPacketQueueBuffers,1);
    sendQosTool =  new networkSchedulerTools(IPDLtcPtr);
    //1.1根据需求创建定时器
    RegistTimer(tictocLayer_TYPE,Send_TIMEOUT_EVENT,send_CIDBASE1+0,0.0,-1);

    //1.2变量初始化
```

The user-defined scheduler/shaper is then bound to it.

```
//协议实例初始化
void tictocLayer::init(){
    //1.通用变量初始化
    mylogToolPtr = new logSystem(deviceId,"tictocSendStatLog");mylogToolPtr->setSaveFileEnable();


    {
        //1.创建队列
        QUEUEBUFFERSINIT(sendPacketQueueBuffers,1);
        //2.创建网络调度工具
        sendQosTool =  new networkSchedulerTools(IPDLtcPtr);
        //3.创建用户自定义调度/整形器
        tictocTBArbiter* tmpTBObjPtr = new tictocTBArbiter(IPDLtcPtr);
        //4.为调度/整形器绑定定时事件回调函数
        struct registFuncTab* _funcTab = new registFuncTab();
        _funcTab->SchedToolsNotify = bind(&tictocLayer::sendQosToolTimeoutCB,this,placeholders::_1);
        //5.完成绑定
        sendQosTool->QosToolRegister(tmpTBObjPtr->_funcTab);
        sendQosTool->timeCbRegister(_funcTab);
    }


    //1.1根据需求创建定时器
    RegistTimer(tictocLayer_TYPE,Send_TIMEOUT_EVENT,send_CIDBASE1+0,0.0,-1);
```

Then we modify the userSendPacket() function to determine the traffic shaping for whether it can go into the queue:

```
    {
        //需要先过调度/整形器，判断一下当前能否进入队列，如果队列满了，这个包
        //是要被丢弃的
        double setNextTime = 0.0;
        //1.初始化队列标志，队列标志的作用为，用于表示当前哪些队列可以使用
        //队列标志与队列个数一一对应，调度器只会使用那些标志为true的队列
        vector<bool > queueFlag;
        for(int i=0;i<sendPacketQueueBuffers.size();i++){
            queueFlag.push_back(false);
        }
        //2.本实例中只使用了单一队列
        queueFlag[0] = true;
        setNextTime = sendQosTool->sched(pkptr,sendPacketQueueBuffers,queueFlag);
        //3.查看当前有没有帧允许调度,哪个队列对应的queueFlag为true,就代表当前数据包应该放在哪个队列
        int position = networkSchedulerTools::findQueueFlagPosition(queueFlag);
        if(position < 0){
            //3.1删除该数据包
            printf("destroy\r\n");
            free(pkptr);
        }
        else{
            //3.2数据包入队
            sendPacketQueueBuffers[position].push_back(pkptr);
            //3.3更新，必须有这一行
            sendQosTool->schedPacketUpdate(pkptr,sendPacketQueueBuffers,queueFlag);
        }

        //4.让networkSchedulerTools继续运行
        sendQosTool->schedSetTimer(setNextTime);
    }

    free(tictocPkptr);
}
```

Next we create the scheduler's timing callback function to determine if there are data frames in the queue that can be scheduled

```
/*****************************
**函数名:sendQosToolTimeoutCB
**函数功能: networkSchedulerTools定时事件回调函数*/
void tictocLayer::sendQosToolTimeoutCB(networkSchedulerTools* schedObjPtr){
    double setNextTime = 0.0;
    //1.构造queueFlag
    vector<bool > queueFlag;
    for(int i=0;i<sendPacketQueueBuffers.size();i++){
        queueFlag.push_back(true);
    }

    setNextTime = schedObjPtr->sched(NULL,sendPacketQueueBuffers,queueFlag);
    printf("setNextTime = %f \r\n",setNextTime);
    //2.查看当前有没有帧允许调度
    int position = networkSchedulerTools::findQueueFlagPosition(queueFlag);
    if(position >= 0){

        for(int i = 0; i < 3; i++)
        {
            cout << "i = " << i << ", sendPacketQueueBuffers.size() = " << sendPacketQueueBuffers[i].size() << endl;
        }

        cout << "portSeq = " << portSeq << endl;
        cout << "position = " << position << endl;
        packetStruct* pkptr = *(sendPacketQueueBuffers[position].begin());
        sendPacketQueueBuffers[position].erase(sendPacketQueueBuffers[position].begin());
        {
            //调用发送函数
            Send(static_cast<void*>(pkptr),1);
            mylogToolPtr->logData(IPDLtcPtr->platformGetCurTimerSec(),deviceId,0,1,"packetSendTime",userClockCurPit);
            printf("packet Send ok\r\n");
        }
        schedObjPtr->schedPacketUpdate(pkptr,sendPacketQueueBuffers,queueFlag);
        //5.释放
        free(pkptr);
    }
    //3.触发发送调度器的事件
    schedObjPtr->schedSetTimer(setNextTime);

}
```

8.5.3 User customization of the scheduler/shaper

③ User customization of the scheduler

First of all, customize the qosPacketCheck() function, we assume that the queue capacity of two packets, here to determine whether the current queue is full, if full, it means that can not be stuffed into the packet, note that once the judgment can not be stuffed again, the return value needs to be a negative number:

```cpp
double tictocTBArbiter::qosPacketCheck(struct packetStruct* pkptr,
                                       vector<vector<packetStruct*>>& pkptrQueue,
                                       vector<bool>& queueFlag) {
    // 遍历 pkptrQueue 的每一行
    for (size_t i = 0; i < pkptrQueue.size(); ++i) {
        if(queueFlag[i] == true){
            if(pkptrQueue[i].size() == 0){
                return userClockCurPit;
            }
            else if(pkptrQueue[i].size()<3)
                return -1.0;

            queueFlag[i] = false;
        }
    }

    return -1.0;
}
```

Next customize the qosQueueCheck() function.

```cpp
double tictocTBArbiter::qosQueueCheck(vector<vector<packetStruct*>>& pkptrQueue,
                                      vector<bool>& queueFlag) {

    packetStruct* tmpPkptr = nullptr;
    // 遍历 pkptrQueue 的每一行
    for (size_t i = 0; i < pkptrQueue.size(); ++i) {

        if(queueFlag[i] == true){
            if(pkptrQueue[i].size()!=0 && doubleOP::dASEdB(nextSendPitInterval,0.0)){
                tmpPkptr = pkptrQueue[i][0];
                nextSendPitInterval = 1.0;
                return userClockCurPit + 1.0;//定制下一次事件生效时间为1s后，符合1packet/s的要求
            }
            else
                queueFlag[i] = false;
        }

    }
    return -1.0;
}
```

For the qosPacketUpdate() function, no customization is currently required. Finally, note that the interval control variables are updated: the qosTimerUpdate function is called automatically, and it is only necessary to note that the relevant variables are updated each time it is called.

```cpp
void tictocTBArbiter::qosTimerUpdate(){
    cout<<"tictocTBArbiter qosTimerUpdate finish"<<endl;

    nextSendPitInterval -= (IPDLtcPtr->getCurTimerSec()-userClockLastPit);
    if(doubleOP::dASEdB(nextSendPitInterval,0.0))
        nextSendPitInterval = 0.0;

    printf("nextSendPitInterval = %f,userClockLastPit = %f,curTime = %f \r\n",nextSendPitInterval,userClockLastPit,IPDLtcPtr->getCurTimerSec());
    updateLocalTimer();

}
```

The following experiment is performed:

tictocSendStatLogdevice1Log.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

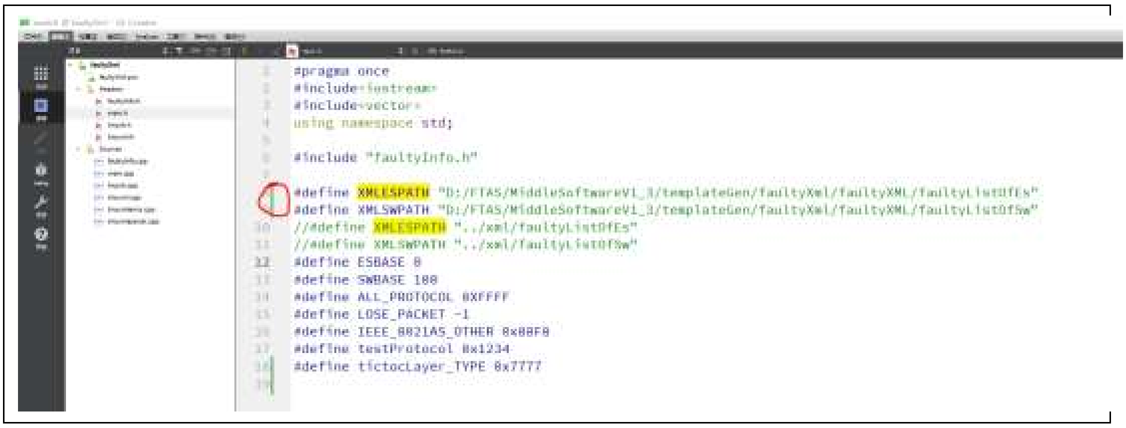
```
Item:0 | Time:0 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:0
Item:2 | Time:1 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:1
Item:4 | Time:2 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:2
Item:6 | Time:3 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:3
Item:8 | Time:4 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:4
Item:10 | Time:5 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:5
```

8.6 Fault Customization TicToc Protocol Fault Injection Example

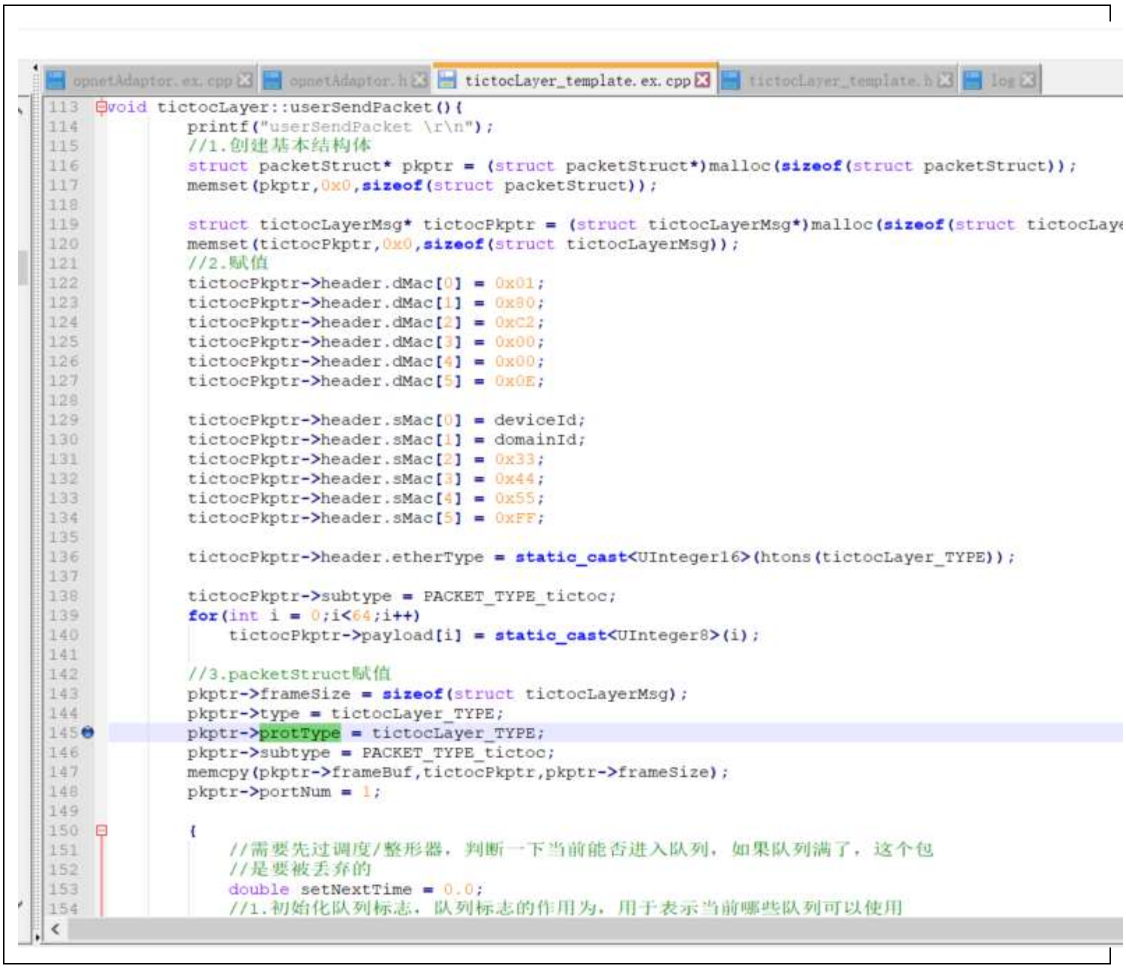
In the following, we demonstrate the fault injection function, we provide the fault injection tool in the “templateGen\faultyXml” folder:

As shown in the figure above, line 12 indicates that the ES0 device will delay the packet of tictocLayer_TYPE by 0.01s in 1-3s. line 15 indicates that the device ES1 will discard the packet of tictocLayer_TYPE in 6-9s. line 16 indicates that the device ES1 will discard the packet of tictocLayer_TYPE in 12-15s.

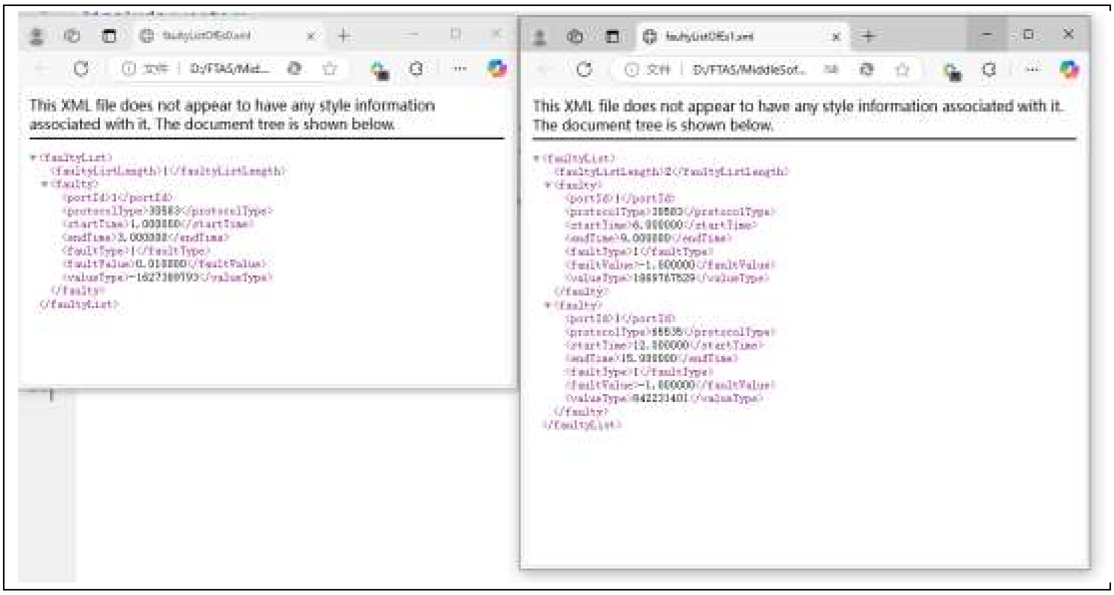
Note that you need to change the path in main.h:



Note the assignment of values to packetStruct when grouping packets:



The XML file is generated after running:



Then you need to copy the generated xml file to the “src\xmlRead\faultConfigXML” directory, the specific path is specified in the main.h:

```
class protocolTree;
class IPDL;
class IEEE8021ASOther;
class IEEE8021AS;
class faultyLayer;


#define TOPOPATH "../../../workspace/topoXML.xml"
#define FLOWPATH "../../../workspace/flowXML.xml"
#define PLANPATH "../../../workspace/routePlan.xml"
#define CBSPATH ("../../../workspace/CBSdeviceID" + to_string((long long)deviceId) + ".xml")
#define QBVPATH ("../../../workspace/QbvdeviceID" + to_string((long long)deviceId) + ".xml")


#define FAULTYXMLPath "../xmlRead/faultConfigXML/"




#include "PIL/PILTimer/IPDLTimer.h"
#include "PIL/PILAPI/IPDL.h"
```

Add a Receive Record Log:

```
//协议实例初始化
void tictocLayer::init(){
    //1.通用变量初始化
    mylogToolPtr = new logSystem(deviceId,"tictocSendStatLog");mylogToolPtr->setSaveFileEnable();
    recceivelogToolPtr = new logSystem(deviceId,"tictocRcvStatLog");mylogToolPtr->setSaveFileEnable();

    {
```

Then just run the program.

```
create a new tt = 04DBA608,tt->thisID = 7
faultyLayer_RegistTimer
create a new tt = 04DBA658,tt->thisID = 8
port 2 timer create complete...
port 2init complete...
faultyLayer::initFaultyDescriptorList
../xmlRead/faultConfigXML/faultyListOfEs1.xml
faulty xml success
faultyLayer::setFaultyTimer
upgateLocalClock upgateLocalClock = 0
upgateLocalClock upgateLocalClock = 0
```

```
Item:3 | Time:9.2e-007 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetRCVTime:9.2e-007
Item:7 | Time:1.00000092 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetRCVTime:1.00000092
Item:11 | Time:2.00000092 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetRCVTime:2.00000092
Item:17 | Time:3.00000092 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetRCVTime:3.00000092
Item:21 | Time:4.00000092 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetRCVTime:4.00000092
Item:25 | Time:5.00000092 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetRCVTime:5.00000092
Item:30 | Time:6.00000092 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetRCVTime:6.00000092
Item:44 | Time:10.00000092 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetRCVTime:10.00000092
Item:48 | Time:11.00000092 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetRCVTime:11.00000092
Item:53 | Time:12.00000092 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetRCVTime:12.00000092
Item:67 | Time:16.00000092 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetRCVTime:16.00000092
Item:71 | Time:17.00000092 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetRCVTime:17.00000092
Item:75 | Time:18.00000092 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetRCVTime:18.00000092
Item:79 | Time:19.00000092 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetRCVTime:19.00000092
```

```
Item:1 | Time:0 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:0
Item:6 | Time:1 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:1
Item:9 | Time:2 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:2
Item:15 | Time:3 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:3
Item:19 | Time:4 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:4
Item:23 | Time:5 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:5
Item:27 | Time:6 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:6
Item:32 | Time:7 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:7
Item:35 | Time:8 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:8
Item:38 | Time:9 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:9
Item:42 | Time:10 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:10
Item:46 | Time:11 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:11
Item:50 | Time:12 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:12
Item:55 | Time:13 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:13
Item:58 | Time:14 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:14
Item:61 | Time:15 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:15
Item:65 | Time:16 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:16
Item:69 | Time:17 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:17
Item:73 | Time:18 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:18
Item:77 | Time:19 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:19
Item:81 | Time:20 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetSendTime:20
```

```
Item:2 | Time:9.2e-007 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:9.2e-007
Item:8 | Time:1.01000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:1.01000092
Item:12 | Time:2.10000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:2.10000092
Item:16 | Time:3.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:3.00000092
Item:20 | Time:4.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:4.00000092
Item:24 | Time:5.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:5.00000092
Item:29 | Time:6.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:6.00000092
Item:33 | Time:7.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:7.00000092
Item:36 | Time:8.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:8.00000092
Item:40 | Time:9.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:9.00000092
Item:43 | Time:10.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:10.00000092
Item:47 | Time:11.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:11.00000092
Item:52 | Time:12.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:12.00000092
Item:56 | Time:13.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:13.00000092
Item:59 | Time:14.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:14.00000092
Item:63 | Time:15.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:15.00000092
Item:66 | Time:16.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:16.00000092
Item:70 | Time:17.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:17.00000092
Item:74 | Time:18.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:18.00000092
Item:78 | Time:19.00000092 | 设备ID:1 | 域ID:0 | 端口ID:1 | packetRCVTime:19.00000092
```

```
Item:0 | Time:0 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:0
Item:5 | Time:1 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:1
Item:10 | Time:2 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:2
Item:14 | Time:3 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:3
Item:18 | Time:4 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:4
Item:22 | Time:5 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:5
Item:26 | Time:6 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:6
Item:31 | Time:7 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:7
Item:34 | Time:8 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:8
Item:37 | Time:9 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:9
Item:41 | Time:10 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:10
Item:45 | Time:11 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:11
Item:49 | Time:12 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:12
Item:54 | Time:13 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:13
Item:57 | Time:14 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:14
Item:60 | Time:15 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:15
Item:64 | Time:16 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:16
Item:68 | Time:17 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:17
Item:72 | Time:18 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:18
Item:76 | Time:19 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:19
Item:80 | Time:20 | 设备ID:0 | 域ID:0 | 端口ID:1 | packetSendTime:20
```