Cyber RT Documents

Apollo Baidu

Sep 26, 2019

QUICKSTART

1 Getting started 3

2 Cyber RT Terms 7

3 F.A.Q. 11

4 Cyber RT API tutorial 15

5 Python API tutorial 39

6 Apollo Cyber RT Developer Tools 47

7 Develop inside Docker Environment 57

8 Migration guide from Apollo ROS 59

9 C++ API 65

10 Python API 105

Index 115

Apollo Cyber RT is an open source, high performance runtime framework designed specifically for autonomous driving scenarios. Based on a centralized computing model, it is greatly optimized for high concurrency, low latency, and high throughput in autonomous driving.

During the last few years of the development of autonomous driving technologies, we have learned a lot from our previous experience with Apollo. The industry is evolving and so is Apollo. Going forward, Apollo has already moved from development to productization, with volume deployments in the real world, we see the demands for the highest level of robustness and performance. That's why we spent years building and perfecting Apollo Cyber RT, which addresses that requirements of autonomous driving solutions.

Key benefits of using Apollo Cyber RT:

- Accelerate development
  - Well defined task interface with data fusion
  - Array of development tools
  - Large set of sensor drivers
- Simplify deployment
  - Efficient and adaptive message communication
  - Configurable user level scheduler with resource awareness
  - Portable with fewer dependencies
- Empower your own autonomous vehicles
  - The default open source runtime framework
  - Building blocks specifically designed for autonomous driving
  - Plug and play your own AD system

CHAPTER ONE

GETTING STARTED

Apollo Cyber RT framework is built based on the concept of component. As a basic building block of Apollo Cyber RT framework, each component contains a specific algorithm module which process a set of data inputs and generate a set of outputs.

In order to successfully create and launch a new component, there are four essential steps that need to happen:

- Set up the component file structure
- Implement the component class
- Set up the configuration files
- Launch the component

The example below demonstrates how to create a simple component, then build, run and watch the final output on screen. If you would like to explore more about Apollo Cyber RT, you can find a couple of examples showing how to use different functionalities of the framework under directory `/apollo/cyber/examples/`.

> **Note**: the example has to be run within apollo docker environment and it's compiled with Bazel.

1.1 Set up the component file structure

Please create the following files, assumed under the directory of `/apollo/cyber/examples/common_component_example/`:

- Header file: common_component_example.h
- Source file: common_component_example.cc
- Build file: BUILD
- DAG dependency file: common.dag
- Launch file: common.launch

1.2 Implement the component class

1.2.1 Implement component header file

To implement `common_component_example.h`:

- Inherit the Component class
- Define your own `Init` and `Proc` functions. Proc function needs to specify its input data types

- Register your component classes to be global by using `CYBER_REGISTER_COMPONENT`

```
#include <memory>
#include "cyber/class_loader/class_loader.h"
#include "cyber/component/component.h"
#include "cyber/examples/proto/examples.pb.h"

using apollo::cyber::examples::proto::Driver;
using apollo::cyber::Component;
using apollo::cyber::ComponentBase;

class CommonComponentSample : public Component<Driver, Driver> {
 public:
  bool Init() override;
  bool Proc(const std::shared_ptr<Driver>& msg0,
            const std::shared_ptr<Driver>& msg1) override;
};

CYBER_REGISTER_COMPONENT(CommonComponentSample)
```

1.2.2 Implement the source file for the example component

For `common_component_example.cc`, both `Init` and `Proc` functions need to be implemented.

```
#include "cyber/examples/common_component_example/common_component_example.h"
#include "cyber/class_loader/class_loader.h"
#include "cyber/component/component.h"

bool CommonComponentSample::Init() {
  AINFO << "Commontest component init";
  return true;
}

bool CommonComponentSample::Proc(const std::shared_ptr<Driver>& msg0,
                                 const std::shared_ptr<Driver>& msg1) {
  AINFO << "Start common component Proc [" << msg0->msg_id() << "] ["
        << msg1->msg_id() << "]";
  return true;
}
```

1.2.3 Create the build file for the example component

Create bazel BUILD file.

```
load("//tools:cpplint.bzl", "cpplint")

package(default_visibility = ["//visibility:public"])

cc_binary(
    name = "libcommon_component_example.so",
    deps = [":common_component_example_lib"],
    linkopts = ["-shared"],
    linkstatic = False,
)
```

(continues on next page)

(continued from previous page)

```
cc_library(
    name = "common_component_example_lib",
    srcs = [
        "common_component_example.cc",
    ],
    hdrs = [
        "common_component_example.h",
    ],
    deps = [
        "//cyber",
        "//cyber/examples/proto:examples_cc_proto",
    ],
)

cpplint()
```

1.3 Set up the configuration files

1.3.1 Configure the DAG dependency file

To configure the DAG dependency file (common.dag), specify the following items as below:

- Channel names: for data input and output
- Library path: library built from component class
- Class name: the class name of the component

```
# Define all coms in DAG streaming.
component_config {
    component_library : "/apollo/bazel-bin/cyber/examples/common_component_example/
↪libcommon_component_example.so"
    components {
        class_name : "CommonComponentSample"
        config {
            name : "common"
            readers {
                channel: "/apollo/prediction"
            }
            readers {
                channel: "/apollo/test"
            }
        }
    }
}
```

1.3.2 Configure the launch file

To configure the launch (common.launch) file, specify the following items:

- The name of the component
- The dag file you just created in the previous step.
- The name of the process which the component runs within

```xml
<cyber>
    <component>
        <name>common</name>
        <dag_conf>/apollo/cyber/examples/common_component_example/common.dag</dag_
↪conf>
        <process_name>common</process_name>
    </component>
</cyber>
```

1.4 Launch the component

Build the component by running the command below:

```
bash /apollo/apollo.sh build
```

Note: make sure the example component builds fine

Then configure the environment:

```bash
cd /apollo/cyber
source setup.bash
```

There are two ways to launch the component:

- Launch with the launch file (recommended)

```
cyber_launch start /apollo/cyber/examples/common_component_example/common.launch
```

- Launch with the DAG file

```
mainboard -d /apollo/cyber/examples/common_component_example/common.dag
```

CHAPTER

TWO

CYBER RT TERMS

This page describes the definitions of the most commonly used terminologies in Cyber RT.

2.1 Component

In an autonomous driving system, modules(like perception, localization, control systems…) exist in the form of components under Cyber RT. Each component communicates with the others through Cyber channels. The component concept not only decouples modules but also provides the flexibility for modules to be divided into components based individual module design.

2.2 Channel

Channels are used to manage data communication in Cyber RT. Users can publish/subscribe to the same channel to achieve p2p communication.

2.3 Task

Task is the abstract description of an asynchronous computation task in Cyber RT.

2.4 Node

Node is the fundamental building block of Cyber RT; every module contains and communicates through the node. A module can have different types of communication by defining read/write and/or service/client in a node.

2.5 Reader/Writer

Message read/write class from/to channel. Reader/Writer are normally created within a node as the major message transfer interface in Cyber RT.

2.6 Service/Client

Besides Reader/writer, Cyber RT also provides service/client pattern for module communication. It supports two-way communication between nodes. A client node will receive a response when a request is made to a service.

2.7 Parameter

Parameter service provides a global parameter access interface in Cyber RT. It's built based on the service/client pattern.

2.8 Service discovery

As a decentralized design framework, Cyber RT does not have a master/central node for service registration. All nodes are treated equally and can find other service nodes through `service discovery`. `UDP` is used in Service discovery.

2.9 CRoutine

Referred to as Coroutine concept, Cyber RT implemented CRoutine to optimize thread usage and system resource allocation.

2.10 Scheduler

To better support autonomous driving scenarios, Cyber RT provides different kinds of resource scheduling algorithms for developers to choose from.

2.11 Message

Message is the data unit used in Cyber RT for data transfer between modules.

2.12 Dag file

Dag file is the config file of module topology. You can define components used and upstream/downstream channels in the dag file.

2.13 Launch files

The Launch file provides a easy way to start modules. By defining one or multiple dag files in the launch file, you can start multiple modules at the same time.

2.14 Record file

The Record file is used to record messages sent/received to/from channels in Cyber RT. Reply record files can help reproduce the behavior of previous operations of Cyber RT.

CHAPTER

THREE

F.A.Q.

3.1 What is Apollo Cyber RT?

Apollo's Cyber RT is an open source runtime framework designed specifically for autonomous driving scenarios. Based on a centralized computing model, it is highly optimized for performance, latency, and data throughput

---

3.2 Why did we decide to work on a new runtime framework?

- During years of development of autonomous driving technologies, we have learned a lot from our previous experience with Apollo. In autonomous driving scenarious, we need an effective centralized computing model, with demands for high performance, including high concurrency, low latency and high throughput
- The industry is evolving, so does Apollo. Going forward, Apollo has already moved from development to productization, with volume deployments in the real world, we see the demands for the highest robustness and high performance. That's why we spent years of building Apollo Cyber RT, which addresses that requirements of autonomous driving solutions.

3.3 What are the advantages of the new runtime framework?

- Accelerate development
  - Well defined task interface with data fusion
  - Array of development tools
  - Large set of sensor drivers
- Simplify deployment
  - Efficient and adaptive message communication
  - Configurable user level scheduler with resource awareness
  - Portable with fewer dependencies
- Empower your own autonomous vehicles
  - The default open source runtime framework
  - Building blocks specifically designed for autonomous driving

   - Plug and play your own AD system

3.4 Can we still use the data that we have collected?

- If the data you have collected is compatible with the previous versions of Apollo, you could use our recommended conversion tools to make the data compliant with our new runtime framework
- If you created a customized data format, then the previously generated data will not be supported by the new runtime framework

---

3.5 Will you continue to support ROS?

We will continue to support previous Apollo releases (3.0 and before) based on ROS. We do appreciate you continue growing with us and highly encourage you to move to Apollo 3.5. While we know that some of our developers would prefer to work on ROS, we do hope you will understand why Apollo as a team cannot continue to support ROS in our future releases as we strive to work towards developing a more holistic platform that meets automotive standards.

3.6 Will Apollo Cyber RT affect regular code development?

If you have not modified anything at runtime framework layer and have only worked on Apollo's module code base, you will not be affected by the introduction of our new runtime framework as most of time you would only need to re-interface the access of the input and output data. Additional documents are under cyber with more details.

3.7 Recommended setup for Apollo Cyber RT

- Currently the runtime framework only supports running on Trusty (Ubuntu 14.04)
- The runtime framework also uses apollo's docker environment
- It is recommended to run source setup.bash when opening a new terminal
- Fork and clone the Apollo repo with the new framework code which can be found at apollo/cyber

3.8 How to enable SHM to decrease the latency?

To decrease number of threads, the readable notification mechanism of shared memory was changed in CyberRT. The default mechanism is UDP multicast, and system call(sendto) will cause some latency.

So, to decrease the latency, you can change the mechanism, The steps are listed as following:

1. update the CyberRT to the latest version;
2. uncomment the transport_conf in https://github.com/ApolloAuto/apollo/blob/master/cyber/conf/cyber.pb.conf;

3. change **notifier_type** of **shm_conf** from “multicast” to “condition”;
4. build CyberRT with opt like `bazel build -c opt --copt=-fpic //cyber/...`;
5. run talker and listener;

Note:You can select the corresponding transmission method according to the relationship between nodes.For example, the default configuration is **INTRA** in the process, **SHM** between the host process, and **RTPS** across the host. Of course you can change all three to RTPS. Or change `same_proc` and `diff_proc` to **SHM**;

3.9 How to use the no serialization message?

The message types supported by Cyber RT include both serializable structured data like protobuf and raw sequence of bytes. You can refer the sample code:

- apollo::cyber::message::RawMessage
- talker: https://github.com/gruminions/apollo/blob/record/cyber/examples/talker.cc
- listener: https://github.com/gruminions/apollo/blob/record/cyber/examples/listener.cc

3.10 How to configure multiple hosts communication?

Make sure the two hosts(or more) are under the same network segment of the local area network, Like `192.168.10.6` and `192.168.10.7`.

You just need to modify `CYBER_IP` of `/apollo/cyber/setup.bash`

```
export CYBER_IP=127.0.0.1
```

Suppose you have two hosts A and Bthe ip of A is `192.168.10.6`, and the ip of B is `192.168.10.7`. Then set `CYBER_IP` to `192.168.10.6` on host A, and set `CYBER_IP` to `192.168.10.7` on host B. Now host A can communicate with host B.

More FAQs to follow…

CHAPTER

FOUR

CYBER RT API TUTORIAL

This document provides an extensive technical deep dive into how to create, manipulate and use Cyber RT's API.

4.1 Table of Contents

- *Talker-Listener*
- *Service Creation and Use*
- *Param parameter service*
- *Log API*
- *Building a module based on Component*
- *Launch*
- *Timer*
- *Time API*
- *Record file: Read and Write*
- *C++ API Directory*
  - *Node*
  - *Writer*
  - *Client*
  - *Parameter*
  - *Timer*
  - *Time*
  - *Duration*
  - *Rate*
  - *RecordReader*
  - *RecordWriter*

4.2 Talker-Listener

The first part of demonstrating CyberRT API is to understand the Talker/Listener example. Following are three essential concepts: node (basic unit), reader(facility to read message) and writer(facility to write message) of the example.

4.2.1 Create a node

In the CyberRT framework, the node is the most fundamental unit, similar to the role of a `handle`. When creating a specific functional object (writer, reader, etc.), you need to create it based on an existing node instance. The node creation interface is as follows:

```
std::unique_ptr<Node> apollo::cyber::CreateNode(const std::string& node_name, const␣
↪std::string& name_space = "");
```

- Parameters:
  - node_name: name of the node, globally unique identifier
  - name_space: name of the space where the node is located

    name_space is empty by default. It is the name of the space concatenated with node_name. The format is `/namespace/node_name`
- Return value - An exclusive smart pointer to Node
- Error Conditions - when `cyber::Init()` has not called, the system is in an uninitialized state, unable to create a node, return nullptr

4.2.2 Create a writer

The writer is the basic facility used in CyberRT to send messages. Every writer corresponds to a channel with a specific data type. The writer is created by the `CreateWriter` interface in the node class. The interfaces are listed as below:

```
template <typename MessageT>
   auto CreateWriter(const std::string& channel_name)
       -> std::shared_ptr<Writer<MessageT>>;
template <typename MessageT>
   auto CreateWriter(const proto::RoleAttributes& role_attr)
       -> std::shared_ptr<Writer<MessageT>>;
```

- Parameters:
  - channel_name: the name of the channel to write to
  - MessageT: The type of message to be written out
- Return value - Shared pointer to the Writer object

4.2.3 Create a reader

The reader is the basic facility used in cyber to receive messages. Reader has to be bound to a callback function when it is created. When a new message arrives in the channel, the callback will be called. The reader is created by the `CreateReader` interface of the node class. The interfaces are listed as below:

```
template <typename MessageT>
auto CreateReader(const std::string& channel_name, const std::function<void(const␣
↪std::shared_ptr<MessageT>&)>& reader_func)
    -> std::shared_ptr<Reader<MessageT>>;

template <typename MessageT>
auto CreateReader(const ReaderConfig& config,
                  const CallbackFunc<MessageT>& reader_func = nullptr)
    -> std::shared_ptr<cyber::Reader<MessageT>>;

template <typename MessageT>
auto CreateReader(const proto::RoleAttributes& role_attr,
                  const CallbackFunc<MessageT>& reader_func = nullptr)
-> std::shared_ptr<cyber::Reader<MessageT>>;
```

- Parameters:
  - MessageT: The type of message to read
  - channel_name: the name of the channel to receive from
  - reader_func: callback function to process the messages
- Return value - Shared pointer to the Reader object

4.2.4 Code Example

Talker (cyber/examples/talker.cc)

```
#include "cyber/cyber.h"
#include "cyber/proto/chatter.pb.h"
#include "cyber/time/rate.h"
#include "cyber/time/time.h"
using apollo::cyber::Rate;
using apollo::cyber::Time;
using apollo::cyber::proto::Chatter;
int main(int argc, char *argv[]) {
  // init cyber framework
  apollo::cyber::Init(argv[0]);
  // create talker node
  std::shared_ptr<apollo::cyber::Node> talker_node(
      apollo::cyber::CreateNode("talker"));
  // create talker
  auto talker = talker_node->CreateWriter<Chatter>("channel/chatter");
  Rate rate(1.0);
  while (apollo::cyber::OK()) {
    static uint64_t seq = 0;
    auto msg = std::make_shared<apollo::cyber::proto::Chatter>();
    msg->set_timestamp(Time::Now().ToNanosecond());
    msg->set_lidar_timestamp(Time::Now().ToNanosecond());
    msg->set_seq(seq++);
    msg->set_content("Hello, apollo!");
    talker->Write(msg);
    AINFO << "talker sent a message!";
    rate.Sleep();
  }
```

(continues on next page)

(continued from previous page)

```
  return 0;
}
```

Listener (cyber/examples/listener.cc)

```
#include "cyber/cyber.h"
#include "cyber/proto/chatter.pb.h"
void MessageCallback(
    const std::shared_ptr<apollo::cyber::proto::Chatter>& msg) {
  AINFO << "Received message seq-> " << msg->seq();
  AINFO << "msgcontent->" << msg->content();
}
int main(int argc, char *argv[]) {
  // init cyber framework
  apollo::cyber::Init(argv[0]);
  // create listener node
  auto listener_node = apollo::cyber::CreateNode("listener");
  // create listener
  auto listener =
      listener_node->CreateReader<apollo::cyber::proto::Chatter>(
          "channel/chatter", MessageCallback);
  apollo::cyber::WaitForShutdown();
  return 0;
}
```

Bazel BUILD file(cyber/samples/BUILD)

```
cc_binary(
    name = "talker",
    srcs = [ "talker.cc", ],
    deps = [
        "//cyber",
        "//cyber/examples/proto:examples_cc_proto",
    ],
)

cc_binary(
    name = "listener",
    srcs = [ "listener.cc", ],
    deps = [
        "//cyber",
        "//cyber/examples/proto:examples_cc_proto",
    ],
)
```

Build and Run

- Build: bazel build cyber/examples/…
- Run talker/listener in different terminals:
    - ./bazel-bin/cyber/examples/talker

    – ./bazel-bin/cyber/examples/listener

- Examine the results: you should see message printing out on listener.

4.3 Service Creation and Use

4.3.1 Introduction

In an autonomous driving system, there are many scenarios that require more from module communication than just sending or receiving messages. Service is another way of communication between nodes. Unlike channel, service implements `two-way` communication, e.g. a node obtains a response by sending a request. This section introduces the `service` module in CyberRT API with examples.

4.3.2 Demo - Example

Problem: create a client-server model that pass Driver.proto back and forth. When a request is sent in by the client, the server parses/processes the request and returns the response.

The implementation of the demo mainly includes the following steps.

Define request and response messages

All messages in cyber are in the `protobuf` format. Any protobuf message with serialize/deserialize interfaces can be used as the service request and response message. `Driver` in examples.proto is used as service request and response in this example:

```
// filename: examples.proto
syntax = "proto2";
package apollo.cyber.examples.proto;
message Driver {
    optional string content = 1;
    optional uint64 msg_id = 2;
    optional uint64 timestamp = 3;
};
```

Create a service and a client

```
// filename: cyber/examples/service.cc
#include "cyber/cyber.h"
#include "cyber/examples/proto/examples.pb.h"

using apollo::cyber::examples::proto::Driver;

int main(int argc, char* argv[]) {
  apollo::cyber::Init(argv[0]);
  std::shared_ptr<apollo::cyber::Node> node(
      apollo::cyber::CreateNode("start_node"));
  auto server = node->CreateService<Driver, Driver>(
      "test_server", [](const std::shared_ptr<Driver>& request,
                        std::shared_ptr<Driver>& response) {
        AINFO << "server: I am driver server";
```

(continues on next page)

(continued from previous page)

```
        static uint64_t id = 0;
        ++id;
        response->set_msg_id(id);
        response->set_timestamp(0);
      });
  auto client = node->CreateClient<Driver, Driver>("test_server");
  auto driver_msg = std::make_shared<Driver>();
  driver_msg->set_msg_id(0);
  driver_msg->set_timestamp(0);
  while (apollo::cyber::OK()) {
    auto res = client->SendRequest(driver_msg);
    if (res != nullptr) {
      AINFO << "client: response: " << res->ShortDebugString();
    } else {
      AINFO << "client: service may not ready.";
    }
    sleep(1);
  }

  apollo::cyber::WaitForShutdown();
  return 0;
}
```

Bazel build file

```
cc_binary(
    name = "service",
    srcs = [ "service.cc", ],
    deps = [
        "//cyber",
        "//cyber/examples/proto:examples_cc_proto",
    ],
)
```

Build and run

- Build service/client: bazel build cyber/examples/…
- Run: ./bazel-bin/cyber/examples/service
- Examining result: you should see content below in apollo/data/log/service.INFO

```
I1124 16:36:44.568845 14965 service.cc:30] [service] server: i am driver server
I1124 16:36:44.569031 14949 service.cc:43] [service] client: response: msg_id: 1␣
↪timestamp: 0
I1124 16:36:45.569514 14966 service.cc:30] [service] server: i am driver server
I1124 16:36:45.569932 14949 service.cc:43] [service] client: response: msg_id: 2␣
↪timestamp: 0
I1124 16:36:46.570627 14967 service.cc:30] [service] server: i am driver server
I1124 16:36:46.571024 14949 service.cc:43] [service] client: response: msg_id: 3␣
↪timestamp: 0
I1124 16:36:47.571566 14968 service.cc:30] [service] server: i am driver server
I1124 16:36:47.571962 14949 service.cc:43] [service] client: response: msg_id: 4␣
↪timestamp: 0
```

(continues on next page)

(continued from previous page)

```
I1124 16:36:48.572634 14969 service.cc:30] [service] server: i am driver server
I1124 16:36:48.573030 14949 service.cc:43] [service] client: response: msg_id: 5␣
↪timestamp: 0
```

4.3.3 Precautions

- When registering a service, note that duplicate service names are not allowed
- The node name applied when registering the server and client should not be duplicated either

4.4 Parameter Service

The Parameter Service is used for shared data between nodes, and provides basic operations such as `set`, `get`, and `list`. The Parameter Service is based on the `Service` implementation and contains service and client.

4.4.1 Parameter Object

Supported Data types

All parameters passed through cyber are `apollo::cyber::Parameter` objects, the table below lists the 5 supported parameter types.

Parameter type | C++ data type | protobuf data type :————- | :————- | :————– apollo::cyber::proto::ParamType::INT | int64_t | int64 apollo::cyber::proto::ParamType::DOUBLE | double | double apollo::cyber::proto::ParamType::BOOL | bool |bool apollo::cyber::proto::ParamType::STRING | std::string | string apollo::cyber::proto::ParamType::PROTOBUF | std::string | string apollo::cyber::proto::ParamType::NOT_SET | - | -

Besides the 5 types above, Parameter also supports interface with protobuf object as incoming parameter. Post performing serialization processes the object and converts it to the STRING type for transfer.

Creating the Parameter Object

Supported constructors:

```
Parameter();  // Name is empty, type is NOT_SET
explicit Parameter(const Parameter& parameter);
explicit Parameter(const std::string& name);  // typeNOT_SET
Parameter(const std::string& name, const bool bool_value);
Parameter(const std::string& name, const int int_value);
Parameter(const std::string& name, const int64_t int_value);
Parameter(const std::string& name, const float double_value);
Parameter(const std::string& name, const double double_value);
Parameter(const std::string& name, const std::string& string_value);
Parameter(const std::string& name, const char* string_value);
Parameter(const std::string& name, const std::string& msg_str,
          const std::string& full_name, const std::string& proto_desc);
Parameter(const std::string& name, const google::protobuf::Message& msg);
```

Sample code of using Parameter object:

```
Parameter a("int", 10);
Parameter b("bool", true);
Parameter c("double", 0.1);
Parameter d("string", "cyber");
Parameter e("string", std::string("cyber"));
// proto message Chatter
Chatter chatter;
Parameter f("chatter", chatter);
std::string msg_str("");
chatter.SerializeToString(&msg_str);
std::string msg_desc("");
ProtobufFactory::GetDescriptorString(chatter, &msg_desc);
Parameter g("chatter", msg_str, Chatter::descriptor()->full_name(), msg_desc);
```

Interface and Data Reading

Interface list:

```
inline ParamType type() const;
inline std::string TypeName() const;
inline std::string Descriptor() const;
inline const std::string Name() const;
inline bool AsBool() const;
inline int64_t AsInt64() const;
inline double AsDouble() const;
inline const std::string AsString() const;
std::string DebugString() const;
template <typename Type>
typename std::enable_if<std::is_base_of<google::protobuf::Message, Type>::value, Type>
↪::type
value() const;
template <typename Type>
typename std::enable_if<std::is_integral<Type>::value && !std::is_same<Type, bool>
↪::value, Type>::type
value() const;
template <typename Type>
typename std::enable_if<std::is_floating_point<Type>::value, Type>::type
value() const;
template <typename Type>
typename std::enable_if<std::is_convertible<Type, std::string>::value, const␣
↪std::string&>::type
value() const;
template <typename Type>
typename std::enable_if<std::is_same<Type, bool>::value, bool>::type
value() const;
```

An example of how to use those interfaces:

```
Parameter a("int", 10);
a.Name();  // return int
a.Type();  // return apollo::cyber::proto::ParamType::INT
a.TypeName();  // return string: INT
a.DebugString();  // return string: {name: "int", type: "INT", value: 10}
int x = a.AsInt64();  // x = 10
x = a.value<int64_t>();  // x = 10
x = a.AsString();  // Undefined behavior, error log prompt
```

(continues on next page)

(continued from previous page)

```
f.TypeName();  // return string: chatter
auto chatter = f.value<Chatter>();
```

4.4.2 Parameter Service

If a node wants to provide a Parameter Service to other nodes, then you need to create a `ParameterService`.

```
/**
 * @brief Construct a new ParameterService object
 *
 * @param node shared_ptr of the node handler
 */
explicit ParameterService(const std::shared_ptr<Node>& node);
```

Since all parameters are stored in the parameter service object, the parameters can be manipulated directly in the ParameterService without sending a service request.

Setting parameters:

```
/**
 * @brief Set the Parameter object
 *
 * @param parameter parameter to be set
 */
void SetParameter(const Parameter& parameter);
```

Getting parameters:

```
/**
 * @brief Get the Parameter object
 *
 * @param param_name
 * @param parameter the pointer to store
 * @return true
 * @return false call service fail or timeout
 */
bool GetParameter(const std::string& param_name, Parameter* parameter);
```

Getting the list of parameters:

```
/**
 * @brief Get all the Parameter objects
 *
 * @param parameters pointer of vector to store all the parameters
 * @return true
 * @return false call service fail or timeout
 */
bool ListParameters(std::vector<Parameter>* parameters);
```

4.4.3 Parameter Client

If a node wants to use parameter services of other nodes, you need to create a `ParameterClient`.

```
/**
 * @brief Construct a new ParameterClient object
 *
 * @param node shared_ptr of the node handler
 * @param service_node_name node name which provide a param services
 */
ParameterClient(const std::shared_ptr<Node>& node, const std::string& service_node_
↪name);
```

You could also perform `SetParameter`, `GetParameter` and `ListParameters` mentioned under *Parameter Service*.

4.4.4 Demo - example

```
#include "cyber/cyber.h"
#include "cyber/parameter/parameter_client.h"
#include "cyber/parameter/parameter_server.h"

using apollo::cyber::Parameter;
using apollo::cyber::ParameterServer;
using apollo::cyber::ParameterClient;

int main(int argc, char** argv) {
  apollo::cyber::Init(*argv);
  std::shared_ptr<apollo::cyber::Node> node =
      apollo::cyber::CreateNode("parameter");
  auto param_server = std::make_shared<ParameterServer>(node);
  auto param_client = std::make_shared<ParameterClient>(node, "parameter");
  param_server->SetParameter(Parameter("int", 1));
  Parameter parameter;
  param_server->GetParameter("int", &parameter);
  AINFO << "int: " << parameter.AsInt64();
  param_client->SetParameter(Parameter("string", "test"));
  param_client->GetParameter("string", &parameter);
  AINFO << "string: " << parameter.AsString();
  param_client->GetParameter("int", &parameter);
  AINFO << "int: " << parameter.AsInt64();
  return 0;
}
```

Build and run

- Build: bazel build cyber/examples/…
- Run: ./bazel-bin/cyber/examples/paramserver

4.5 Log API

4.5.1 Log library

Cyber log library is built on top of glog. The following header files need to be included:

```
#include "cyber/common/log.h"
#include "cyber/init.h"
```

4.5.2 Log configuration

Default global config path: cyber/setup.bash

The configs below could be modified by devloper:

```
export GLOG_log_dir=/apollo/data/log
export GLOG_alsologtostderr=0
export GLOG_colorlogtostderr=1
export GLOG_minloglevel=0
```

4.5.3 Log initialization

Call the Init method at the code entry to initialize the log:

```
apollo::cyber::cyber::Init(argv[0]) is initialized.
If no macro definition is made in the previous component, the corresponding log is␣
↪printed to the binary log.
```

4.5.4 Log output macro

Log library is encapsulated in Log printing macros. The related log macros are used as follows:

```
ADEBUG << "hello cyber.";
AINFO  << "hello cyber.";
AWARN  << "hello cyber.";
AERROR << "hello cyber.";
AFATAL << "hello cyber.";
```

4.5.5 Log format

The format is `<MODULE_NAME>.log.<LOG_LEVEL>.<datetime>.<process_id>`

4.5.6 About log files

Currently, the only different output behavior from default glog is that different log levels of a module will be written into the same log file.

4.6 Building a module based on Component

4.6.1 Key concepts

1. Component

The component is the base class that Cyber RT provides to build application modules. Each specific application module can inherit the Component class and define its own `Init` and `Proc` functions so that it can be loaded into the Cyber framework.

2. Binary vs Component

There are two options to use Cyber RT framework for applications:

- Binary based: the application is compiled separately into a binary, which communicates with other cyber modules by creating its own `Reader` and `Writer`.
- Component based: the application is compiled into a Shared Library. By inheriting the Component class and writing the corresponding dag description file, the Cyber RT framework will load and run the application dynamically.

The essential Component interface

- The component's `Init()` function is like the main function that does some initialization of the algorithm.
- Component's `Proc()` function works like Reader's callback function that is called by the framework when a message arrives.

Advantages of using Component

- Component can be loaded into different processes through the launch file, and the deployment is flexible.
- Component can change the received channel name by modifying the dag file without recompiling.
- Component supports receiving multiple types of data.
- Component supports providing multiple fusion strategies.

3. Dag file format

An example dag file:

```
# Define all coms in DAG streaming.
module_config {
    module_library : "lib/libperception_component.so"
    components {
        class_name : "PerceptionComponent"
        config {
            name : "perception"
            readers {
                channel: "perception/channel_name"
            }
        }
    }
    timer_components {
        class_name : "DriverComponent"
        config {
            name : "driver"
```

(continues on next page)

(continued from previous page)

```
            interval : 100
        }
    }
}
```

- **module_library**: If you want to load the .so library the root directory is the working directory of cyber (the same directory of `setup.bash`)
- **components & timer_component**: Select the base component class type that needs to be loaded.
- **class_name**: the name of the component class to load
- **name**: the loaded class_name as the identifier of the loading example
- **readers**: Data received by the current component, supporting 1-3 channels of data.

4.6.2 Demo - examples

Common_component_example(cyber/examples/common_component_example/*)

Header definition(common_component_example.h)

```
#include <memory>

#include "cyber/class_loader/class_loader.h"
#include "cyber/component/component.h"
#include "cyber/examples/proto/examples.pb.h"

using apollo::cyber::examples::proto::Driver;
using apollo::cyber::Component;
using apollo::cyber::ComponentBase;

class Commontestcomponent : public Component<Driver, Driver> {
 public:
  bool Init() override;
  bool Proc(const std::shared_ptr<Driver>& msg0,
            const std::shared_ptr<Driver>& msg1) override;
};
CYBER_REGISTER_COMPONENT(Commontestcomponent)
```

Cpp file implementation(common_component_example.cc)

```
#include "cyber/examples/common_component_smaple/common_component_example.h"

#include "cyber/class_loader/class_loader.h"
#include "cyber/component/component.h"

bool Commontestcomponent::Init() {
  AINFO << "Commontest component init";
  return true;
}

bool Commontestcomponent::Proc(const std::shared_ptr<Driver>& msg0,
                               const std::shared_ptr<Driver>& msg1) {
  AINFO << "Start commontest component Proc [" << msg0->msg_id() << "] ["
        << msg1->msg_id() << "]";
```

(continues on next page)

(continued from previous page)

```
  return true;
}
```

Timer_component_example(cyber/examples/timer_component_example/*)

Header definition(timer_component_example.h)

```
#include <memory>

#include "cyber/class_loader/class_loader.h"
#include "cyber/component/component.h"
#include "cyber/component/timer_component.h"
#include "cyber/examples/proto/examples.pb.h"

using apollo::cyber::examples::proto::Driver;
using apollo::cyber::Component;
using apollo::cyber::ComponentBase;
using apollo::cyber::TimerComponent;
using apollo::cyber::Writer;

class TimertestComponent : public TimerComponent {
 public:
  bool Init() override;
  bool Proc() override;

 private:
  std::shared_ptr<Writer<Driver>> driver_writer_ = nullptr;
};
CYBER_REGISTER_COMPONENT(TimertestComponent)
```

Cpp file implementation(timer_component_example.cc)

```
#include "cyber/examples/timer_component_example/timer_component_example.h"

#include "cyber/class_loader/class_loader.h"
#include "cyber/component/component.h"
#include "cyber/examples/proto/examples.pb.h"

bool TimertestComponent::Init() {
  driver_writer_ = node_->CreateWriter<Driver>("/carstatus/channel");
  return true;
}

bool TimertestComponent::Proc() {
  static int i = 0;
  auto out_msg = std::make_shared<Driver>();
  out_msg->set_msg_id(i++);
  driver_writer_->Write(out_msg);
  AINFO << "timertestcomponent: Write drivermsg->"
        << out_msg->ShortDebugString();
  return true;
}
```

Build and run

Use timertestcomponent as example:

- Build: bazel build cyber/examples/timer_component_smaple/…
- Run: mainboard -d cyber/examples/timer_component_smaple/timer.dag

4.6.3 Precautions

- Component needs to be registered to load the class through SharedLibrary. The registration interface looks like:

```
CYBER_REGISTER_COMPONENT(DriverComponent)
```

If you use a namespace when registering, you also need to add a namespace when you define it in the dag file.

- The configuration files of the Component and TimerComponent are different, please be careful not to mix the two up.

4.7 Launch

cyber_launch is the launcher of the Cyber RT framework. It starts multiple mainboards according to the launch file, and loads different components into different mainboards according to the dag file. cyber_launch supports two scenarios for dynamically loading components or starting Binary programs in a child process.

4.7.1 Launch File Format

```
<cyber>
    <module>
        <name>driver</name>
        <dag_conf>driver.dag</dag_conf>
        <process_name></process_name>
        <exception_handler>exit</exception_handler>
    </module>
    <module>
        <name>perception</name>
        <dag_conf>perception.dag</dag_conf>
        <process_name></process_name>
        <exception_handler>respawn</exception_handler>
    </module>
    <module>
        <name>planning</name>
        <dag_conf>planning.dag</dag_conf>
        <process_name></process_name>
    </module>
</cyber>
```

Module: Each loaded component or binary is a module

- **name** is the loaded module name
- **dag_conf** is the name of the corresponding dag file of the component
- **process_name** is the name of the mainboard process once started, and the same component of process_name will be loaded and run in the same process.

- **exception_handler** is the handler method when the exception occurs in the process. The value can be exit or respawn listed below.
  - exit, which means that the entire process needs to stop running when the current process exits abnormally.
  - respawn, the current process needs to be restarted after abnormal exit. Start this process. If there is no such thing as it is empty, it means no treatment. Can be controlled by the user according to the specific conditions of the process

4.8 Timer

Timer can be used to create a timed task to run on a periodic basis, or to run only once

4.8.1 Timer Interface

```
/**
 * @brief Construct a new Timer object
 *
 * @param period The period of the timer, unit is ms
 * @param callback The tasks that the timer needs to perform
 * @param oneshot True: perform the callback only after the first timing cycle
 *                False: perform the callback every timed period
 */
Timer(uint32_t period, std::function<void()> callback, bool oneshot);
```

Or you could encapsulate the parameters into a timer option as follows:

```
struct TimerOption {
  uint32_t period;                   // The period of the timer, unit is ms
  std::function<void()> callback;  // The tasks that the timer needs to perform
  bool oneshot;  // True: perform the callback only after the first timing cycle
                 // False: perform the callback every timed period
};
/**
 * @brief Construct a new Timer object
 *
 * @param opt Timer option
 */
explicit Timer(TimerOption opt);
```

4.8.2 Start Timer

After creating a Timer instance, you must call `Timer::Start()` to start the timer.

4.8.3 Stop Timer

When you need to manually stop a timer that has already started, you can call the `Timer::Stop()` interface.

4.8.4 Demo - example

```
#include <iostream>
#include "cyber/cyber.h"
int main(int argc, char** argv) {
    cyber::Init(argv[0]);
    // Print current time every 100ms
    cyber::Timer timer(100, [](){
        std::cout << cyber::Time::Now() << std::endl;
    }, false);
    timer.Start()
    sleep(1);
    timer.Stop();
}
```

4.9 Time API

Time is a class used to manage time; it can be used for current time acquisition, time-consuming calculation, time conversion, and so on.

The time interfaces are as follows:

```
// constructor, passing in a different value to construct Time
Time(uint64_t nanoseconds); //uint64_t, in nanoseconds
Time(int nanoseconds); // int type, unit: nanoseconds
Time(double seconds); // double, in seconds
Time(uint32_t seconds, uint32_t nanoseconds);
// seconds seconds + nanoseconds nanoseconds
Static Time Now(); // Get the current time
Double ToSecond() const; // convert to seconds
Uint64_t ToNanosecond() const; // Convert to nanoseconds
Std::string ToString() const; // Convert to a string in the format "2018-07-10␣
↪20:21:51.123456789"
Bool IsZero() const; // Determine if the time is 0
```

A code example can be seen below:

```
#include <iostream>
#include "cyber/cyber.h"
#include "cyber/duration.h"
int main(int argc, char** argv) {
    cyber::Init(argv[0]);
    Time t1(1531225311123456789UL);
    std::cout << t1.ToString() << std::endl; // 2018-07-10 20:21:51.123456789
    // Duration time interval
    Time t1(100);
    Duration d(200);
    Time t2(300);
    assert(d == (t1-t2)); // true
}
```

4.10 Record file: Read and Write

4.10.1 Reading the Reader file

RecordReader is the component used to read messages in the cyber framework. Each RecordReader can open an existing record file through the `Open` method, and the thread will asynchronously read the information in the record file. The user only needs to execute ReadMessage to extract the latest message in RecordReader, and then get the message information through GetCurrentMessageChannelName, GetCurrentRawMessage, GetCurrentMessageTime.

RecordWriter is the component used to record messages in the cyber framework. Each RecordWriter can create a new record file through the Open method. The user only needs to execute WriteMessage and WriteChannel to write message and channel information, and the writing process is asynchronous.

4.10.2 Demo - example(cyber/examples/record.cc)

Write 100 RawMessage to`TEST_FILE` through `test_write` method, then read them out through `test_read` method.

```
#include <string>

#include "cyber/cyber.h"
#include "cyber/message/raw_message.h"
#include "cyber/proto/record.pb.h"
#include "cyber/record/record_message.h"
#include "cyber/record/record_reader.h"
#include "cyber/record/record_writer.h"

using ::apollo::cyber::record::RecordReader;
using ::apollo::cyber::record::RecordWriter;
using ::apollo::cyber::record::RecordMessage;
using apollo::cyber::message::RawMessage;

const char CHANNEL_NAME_1[] = "/test/channel1";
const char CHANNEL_NAME_2[] = "/test/channel2";
const char MESSAGE_TYPE_1[] = "apollo.cyber.proto.Test";
const char MESSAGE_TYPE_2[] = "apollo.cyber.proto.Channel";
const char PROTO_DESC[] = "1234567890";
const char STR_10B[] = "1234567890";
const char TEST_FILE[] = "test.record";

void test_write(const std::string &writefile) {
  RecordWriter writer;
  writer.SetSizeOfFileSegmentation(0);
  writer.SetIntervalOfFileSegmentation(0);
  writer.Open(writefile);
  writer.WriteChannel(CHANNEL_NAME_1, MESSAGE_TYPE_1, PROTO_DESC);
  for (uint32_t i = 0; i < 100; ++i) {
    auto msg = std::make_shared<RawMessage>("abc" + std::to_string(i));
    writer.WriteMessage(CHANNEL_NAME_1, msg, 888 + i);
  }
  writer.Close();
}

void test_read(const std::string &readfile) {
  RecordReader reader(readfile);
```

(continues on next page)

(continued from previous page)

```
  RecordMessage message;
  uint64_t msg_count = reader.GetMessageNumber(CHANNEL_NAME_1);
  AINFO << "MSGTYPE: " << reader.GetMessageType(CHANNEL_NAME_1);
  AINFO << "MSGDESC: " << reader.GetProtoDesc(CHANNEL_NAME_1);

  // read all message
  uint64_t i = 0;
  uint64_t valid = 0;
  for (i = 0; i < msg_count; ++i) {
    if (reader.ReadMessage(&message)) {
      AINFO << "msg[" << i << "]-> "
            << "channel name: " << message.channel_name
            << "; content: " << message.content
            << "; msg time: " << message.time;
      valid++;
    } else {
      AERROR << "read msg[" << i << "] failed";
    }
  }
  AINFO << "static msg=================";
  AINFO << "MSG validmsg:totalcount: " << valid << ":" << msg_count;
}

int main(int argc, char *argv[]) {
  apollo::cyber::Init(argv[0]);
  test_write(TEST_FILE);
  sleep(1);
  test_read(TEST_FILE);
  return 0;
}
```

Build and run

- Build: bazel build cyber/examples/…
- Run: ./bazel-bin/cyber/examples/record
- Examining result:

```
I1124 16:56:27.248200 15118 record.cc:64] [record] msg[0]-> channel name: /test/
↪channel1; content: abc0; msg time: 888
I1124 16:56:27.248227 15118 record.cc:64] [record] msg[1]-> channel name: /test/
↪channel1; content: abc1; msg time: 889
I1124 16:56:27.248239 15118 record.cc:64] [record] msg[2]-> channel name: /test/
↪channel1; content: abc2; msg time: 890
I1124 16:56:27.248252 15118 record.cc:64] [record] msg[3]-> channel name: /test/
↪channel1; content: abc3; msg time: 891
I1124 16:56:27.248297 15118 record.cc:64] [record] msg[4]-> channel name: /test/
↪channel1; content: abc4; msg time: 892
I1124 16:56:27.248378 15118 record.cc:64] [record] msg[5]-> channel name: /test/
↪channel1; content: abc5; msg time: 893
...
I1124 16:56:27.250422 15118 record.cc:73] [record] static msg=================
I1124 16:56:27.250434 15118 record.cc:74] [record] MSG validmsg:totalcount: 100:100
```

4.11 API Directory

4.11.1 Node API

For additional information and examples, refer to *Node*

4.11.2 API List

```
//create writer with user-define attr and message type
auto CreateWriter(const proto::RoleAttributes& role_attr)
    -> std::shared_ptr<transport::Writer<MessageT>>;
//create reader with user-define attr, callback and message type
auto CreateReader(const proto::RoleAttributes& role_attr,
    const croutine::CRoutineFunc<MessageT>& reader_func)
    -> std::shared_ptr<transport::Reader<MessageT>>;
//create writer with specific channel name and message type
auto CreateWriter(const std::string& channel_name)
    -> std::shared_ptr<transport::Writer<MessageT>>;
//create reader with specific channel name, callback and message type
auto CreateReader(const std::string& channel_name,
    const croutine::CRoutineFunc<MessageT>& reader_func)
    -> std::shared_ptr<transport::Reader<MessageT>>;
//create reader with user-define config, callback and message type
auto CreateReader(const ReaderConfig& config,
                  const CallbackFunc<MessageT>& reader_func)
    -> std::shared_ptr<cybertron::Reader<MessageT>>;
//create service with name and specific callback
auto CreateService(const std::string& service_name,
    const typename service::Service<Request, Response>::ServiceCallback& service_
↪calllback)
    -> std::shared_ptr<service::Service<Request, Response>>;
//create client with name to send request to server
auto CreateClient(const std::string& service_name)
    -> std::shared_ptr<service::Client<Request, Response>>;
```

4.12 Writer API

For additional information and examples, refer to *Writer*

4.12.1 API List

```
bool Write(const std::shared_ptr<MessageT>& message);
```

4.13 Client API

For additional information and examples, refer to *Client*

4.13.1 API List

```
SharedResponse SendRequest(SharedRequest request,
                           const std::chrono::seconds& timeout_s =␣
↪std::chrono::seconds(5));
SharedResponse SendRequest(const Request& request,
                           const std::chrono::seconds& timeout_s =␣
↪std::chrono::seconds(5));
```

4.14 Parameter API

The interface that the user uses to perform parameter related operations:

- Set the parameter related API.
- Read the parameter related API.
- Create a ParameterService to provide parameter service related APIs for other nodes.
- Create a ParameterClient that uses the parameters provided by other nodes to service related APIs.

For additional information and examples, refer to *Parameter*

4.14.1 API List - Setting parameters

```
Parameter();  // Name is empty, type is NOT_SET
explicit Parameter(const Parameter& parameter);
explicit Parameter(const std::string& name);  // Type is NOT_SET
Parameter(const std::string& name, const bool bool_value);
Parameter(const std::string& name, const int int_value);
Parameter(const std::string& name, const int64_t int_value);
Parameter(const std::string& name, const float double_value);
Parameter(const std::string& name, const double double_value);
Parameter(const std::string& name, const std::string& string_value);
Parameter(const std::string& name, const char* string_value);
Parameter(const std::string& name, const std::string& msg_str,
          const std::string& full_name, const std::string& proto_desc);
Parameter(const std::string& name, const google::protobuf::Message& msg);
```

4.14.2 API List - Reading parameters

```
inline ParamType type() const;
inline std::string TypeName() const;
inline std::string Descriptor() const;
inline const std::string Name() const;
inline bool AsBool() const;
inline int64_t AsInt64() const;
inline double AsDouble() const;
inline const std::string AsString() const;
std::string DebugString() const;
template <typename Type>
typename std::enable_if<std::is_base_of<google::protobuf::Message, Type>::value, Type>
↪::type
```

(continues on next page)

(continued from previous page)

```
value() const;
template <typename Type>
typename std::enable_if<std::is_integral<Type>::value && !std::is_same<Type, bool>
↪::value, Type>::type
value() const;
template <typename Type>
typename std::enable_if<std::is_floating_point<Type>::value, Type>::type
value() const;
template <typename Type>
typename std::enable_if<std::is_convertible<Type, std::string>::value, const␣
↪std::string&>::type
value() const;
template <typename Type>
typename std::enable_if<std::is_same<Type, bool>::value, bool>::type
value() const;
```

4.14.3 API List - Creating parameter service

```
explicit ParameterService(const std::shared_ptr<Node>& node);
void SetParameter(const Parameter& parameter);
bool GetParameter(const std::string& param_name, Parameter* parameter);
bool ListParameters(std::vector<Parameter>* parameters);
```

4.14.4 API List - Creating parameter client

```
ParameterClient(const std::shared_ptr<Node>& node, const std::string& service_node_
↪name);
bool SetParameter(const Parameter& parameter);
bool GetParameter(const std::string& param_name, Parameter* parameter);
bool ListParameters(std::vector<Parameter>* parameters);
```

4.15 Timer API

You can set the parameters of the Timer and call the start and stop interfaces to start the timer and stop the timer. For additional information and examples, refer to *Timer*

4.15.1 API List

```
Timer(uint32_t period, std::function<void()> callback, bool oneshot);
Timer(TimerOption opt);
void SetTimerOption(TimerOption opt);
void Start();
void Stop();
```

4.16 Time API

For additional information and examples, refer to *Time*

4.16.1 API List

```
static const Time MAX;
static const Time MIN;
Time() {}
explicit Time(uint64_t nanoseconds);
explicit Time(int nanoseconds);
explicit Time(double seconds);
Time(uint32_t seconds, uint32_t nanoseconds);
Time(const Time& other);
static Time Now();
static Time MonoTime();
static void SleepUntil(const Time& time);
double ToSecond() const;
uint64_t ToNanosecond() const;
std::string ToString() const;
bool IsZero() const;
```

4.17 Duration API

Interval-related interface, used to indicate the time interval, can be initialized according to the specified nanosecond or second.

4.17.1 API List

```
Duration() {}
Duration(int64_t nanoseconds);
Duration(int nanoseconds);
Duration(double seconds);
Duration(uint32_t seconds, uint32_t nanoseconds);
Duration(const Rate& rate);
Duration(const Duration& other);
double ToSecond() const;
int64_t ToNanosecond() const;
bool IsZero() const;
void Sleep() const;
```

4.18 Rate API

The frequency interface is generally used to initialize the time of the sleep frequency after the object is initialized according to the specified frequency.

4.18.1 API List

```
Rate(double frequency);
Rate(uint64_t nanoseconds);
Rate(const Duration&);
void Sleep();
void Reset();
```

(continues on next page)

(continued from previous page)

```
Duration CycleTime() const;
Duration ExpectedCycleTime() const { return expected_cycle_time_; }
```

4.19 RecordReader API

The interface for reading the record file is used to read the message and channel information in the record file.

4.19.1 API List

```
RecordReader();
bool Open(const std::string& filename, uint64_t begin_time = 0,
          uint64_t end_time = UINT64_MAX);
void Close();
bool ReadMessage();
bool EndOfFile();
const std::string& CurrentMessageChannelName();
std::shared_ptr<RawMessage> CurrentRawMessage();
uint64_t CurrentMessageTime();
```

4.20 RecordWriter API

The interface for writing the record file, used to record the message and channel information into the record file.

4.20.1 API List

```
RecordWriter();
bool Open(const std::string& file);
void Close();
bool WriteChannel(const std::string& name, const std::string& type,
                  const std::string& proto_desc);
template <typename MessageT>
bool WriteMessage(const std::string& channel_name, const MessageT& message,
                  const uint64_t time_nanosec,
                  const std::string& proto_desc = "");
bool SetSizeOfFileSegmentation(uint64_t size_kilobytes);
bool SetIntervalOfFileSegmentation(uint64_t time_sec);
```

CHAPTER

FIVE

PYTHON API TUTORIAL

5.1 1. Background

The core functions of Cyber RT are developed in C++. We also provide more python interfaces to help developers build their own utilities for specific projects.

5.2 2. Cyber RT Python Interfaces

The python interfaces of Cyber RT are wrapper the corresponding C++ interfaces. The implementation doesn't rely on other third-party tools, e.g. swig, which makes it easier to maintain.

5.3 3. Overview of Python Interfaces in Cyber RT

So far, the python interfaces covers:

- access the information of channels
- server/client communication
- query informatoin in record files
- read and write from/to record files
- Time/Duration/Rate related operations
- Timer

5.3.1 3.1 Read/Write of Channels

Steps shown as below:

1. First create a Node
2. Create corresponding reader or writer;
3. If write to a channel, use write interface in writer.
4. If read from a channel, use the spin interface in the node, and process the messages in your callback function

The interfaces are shown below:

```
class Node:
    """
    Class for cyber Node wrapper.
    """

    def create_writer(self, name, data_type, qos_depth=1):
        """
        create a topic writer for send message to topic.
        @param self
        @param name str: topic name
        @param data_type proto: message class for serialization
        """

   def create_reader(self, name, data_type, callback, args=None):
        """
        create a topic reader for receive message from topic.
        @param self
        @param name str: topic name
        @param data_type proto: message class for serialization
        @callback fn: function to call (fn(data)) when data is
                   received. If args is set, the function must
                   accept the args as a second argument,
                   i.e. fn(data, args)
        @args any: additional arguments to pass to the callback
        """
        def create_client(self, name, request_data_type, response_data_type):
        """
        """
        def create_service(self, name, req_data_type, res_data_type, callback,␣
↪args=None):

    def spin(self):
        """
        spin in wait and process message.
        @param self
        """

class Writer(object):
    """
    Class for cyber writer wrapper.
    """
    def write(self, data):
        """
        writer msg string
        """
```

5.3.2 3.2 Record Interfaces

Read from record

1. Create a RecordReader;
2. Read messages from Record;

Write to record

1. Create a RecordWriter

2. Write messages to record

The interfaces are shown below:

```
class RecordReader(object):
    """
    Class for cyber RecordReader wrapper.
    """
    def read_messages(self, start_time=0, end_time=18446744073709551615):
        """
        read message from bag file.
        @param self
        @param start_time:
        @param end_time:
        @return: generator of (message, data_type, timestamp)
        """
        def get_messagenumber(self, channel_name):
        """
        return message count.
        """
        def get_messagetype(self, channel_name):
        """
        return message type.
        """
        def get_protodesc(self, channel_name):
        """
        return message protodesc.
        """
        def get_headerstring(self):
        """
        return message header string.
        """
        def reset(self):
        """
        return reset.
        ""
        return _CYBER_RECORD.PyRecordReader_Reset(self.record_reader)

    def get_channellist(self):
        """
        return channel list.
        """
        return _CYBER_RECORD.PyRecordReader_GetChannelList(self.record_reader)


class RecordWriter(object):
    """
    Class for cyber RecordWriter wrapper.
    """
        def open(self, path):
        """
        open record file for write.
        """
        def write_channel(self, channel_name, type_name, proto_desc):
        """
        writer channel by channelname,typename,protodesc
        """
        def write_message(self, channel_name, data, time, raw = True):
```

(continues on next page)

(continued from previous page)

```
        """
        writer msg:channelname,data,time,is data raw
        """

        def set_size_fileseg(self, size_kilobytes):
        """
        return filesegment size.
        """

    def set_intervaltime_fileseg(self, time_sec):
        """
        return file interval time.
        """

    def get_messagenumber(self, channel_name):
        """
        return message count.
        """

    def get_messagetype(self, channel_name):
        """
        return message type.
        """

    def get_protodesc(self, channel_name):
        """
        return message protodesc.
        """
```

5.3.3 3.3 Time Interfaces

```
class Time(object):
        @staticmethod
    def now():
        time_now = Time(_CYBER_TIME.PyTime_now())
        return time_now

    @staticmethod
    def mono_time():
        mono_time = Time(_CYBER_TIME.PyTime_mono_time())
        return mono_time

    def to_sec(self):
        return _CYBER_TIME.PyTime_to_sec(self.time)

    def to_nsec(self):
        return _CYBER_TIME.PyTime_to_nsec(self.time)

    def sleep_until(self, nanoseconds):
        return _CYBER_TIME.PyTime_sleep_until(self.time, nanoseconds)
```

5.3.4 3.4 Timer Interfaces

```
class Timer(object):

    def set_option(self, period, callback, oneshot=0):
        """
        set the option of timer.
        @param period The period of the timer, unit is ms.
        @param callback The tasks that the timer needs to perform.
        @param oneshot 1:perform the callback only after the first timing cycle
        0:perform the callback every timed period
        """

    def start(self):

    def stop(self):
```

5.4 4. Examples

5.4.1 4.1 Read from Channel (in cyber/python/examples/listener.py)

```
import sys
sys.path.append("../")
from cyber_py import cyber
from modules.common.util.testdata.simple_pb2 import SimpleMessage

def callback(data):

    """
    reader message callback.
    """
    print "="*80
    print "py:reader callback msg->:"
    print data
    print "="*80

def test_listener_class():
    """
    reader message.
    """
    print "=" * 120
    test_node = cyber.Node("listener")
    test_node.create_reader("channel/chatter",
            SimpleMessage, callback)
    test_node.spin()

if __name__ == '__main__':

    cyber.init()
    test_listener_class()
    cyber.shutdown()
```

5.4.2 4.2 Write to Channel(in cyber/python/examples/talker.py)

```
from modules.common.util.testdata.simple_pb2 import SimpleMessage
from cyber_py import cyber
"""Module for example of talker."""
import time
import sys
sys.path.append("../")

def test_talker_class():
    """
    test talker.
    """
    msg = SimpleMessage()
    msg.text = "talker:send Alex!"
    msg.integer = 0
    test_node = cyber.Node("node_name1")
    g_count = 1
    writer = test_node.create_writer("channel/chatter",
                                     SimpleMessage, 6)
    while not cyber.is_shutdown():
        time.sleep(1)
        g_count = g_count + 1
        msg.integer = g_count
        print "="*80
        print "write msg -> %s" % msg
        writer.write(msg)

if __name__ == '__main__':
    cyber.init()
    test_talker_class()
    cyber.shutdown()
```

5.4.3 4.3 Read and Write Messages from/to Record File(in cyber/python/examples/record.py)

```
"""Module for example of record."""

import time
import sys

sys.path.append("../")
from cyber_py import cyber
from cyber_py import record
from google.protobuf.descriptor_pb2 import FileDescriptorProto
from modules.common.util.testdata.simple_pb2 import SimpleMessage

TEST_RECORD_FILE = "test02.record"
CHAN_1 = "channel/chatter"
CHAN_2 = "/test2"
MSG_TYPE = "apollo.common.util.test.SimpleMessage"
STR_10B = "1234567890"
TEST_FILE = "test.record"

def test_record_writer(writer_path):
```

(continues on next page)

(continued from previous page)

```
    """
    record writer.
    """
    fwriter = record.RecordWriter()
    if not fwriter.open(writer_path):
        print "writer open failed!"
        return
    print "+++ begin to writer..."
    fwriter.write_channel(CHAN_1, MSG_TYPE, STR_10B)
    fwriter.write_message(CHAN_1, STR_10B, 1000)

    msg = SimpleMessage()
    msg.text = "AAAAAA"

    file_desc = msg.DESCRIPTOR.file
    proto = FileDescriptorProto()
    file_desc.CopyToProto(proto)
    proto.name = file_desc.name
    desc_str = proto.SerializeToString()

    fwriter.write_channel('chatter_a', msg.DESCRIPTOR.full_name, desc_str)
    fwriter.write_message('chatter_a', msg, 998, False)
    fwriter.write_message("chatter_a", msg.SerializeToString(), 999)

    fwriter.close()

def test_record_reader(reader_path):
    """
    record reader.
    """
    freader = record.RecordReader(reader_path)
    time.sleep(1)
    print "+"*80
    print "+++begin to read..."
    count = 1
    for channelname, msg, datatype, timestamp in freader.read_messages():
        print "="*80
        print "read [%d] msg" % count
        print "chnanel_name -> %s" % channelname
        print "msg -> %s" % msg
        print "msgtime -> %d" % timestamp
        print "msgnum -> %d" % freader.get_messagenumber(channelname)
        print "msgtype -> %s" % datatype
        count = count + 1

if __name__ == '__main__':
    cyber.init()
    test_record_writer(TEST_RECORD_FILE)
    test_record_reader(TEST_RECORD_FILE)
    cyber.shutdown()
```

CHAPTER

SIX

APOLLO CYBER RT DEVELOPER TOOLS

Apollo Cyber RT framework comes with a collection of useful tools for daily development, including one visualization tool cyber_visualizer and two command line tools cyber_monitor and cyber_recorder.

Note: apollo docker environment is required to use the tools, please follow apollo wiki to enter docker

All the tools from Apollo Cyber RT rely on Apollo Cyber RT library, so you must source the setup.bash file for environment setup before using any Apollo Cyber RT tools, shown as below:

```
username@computername:~$: source /your-path-to-apollo-install-dir/cyber/setup.bash
```

6.1 Cyber_visualizer

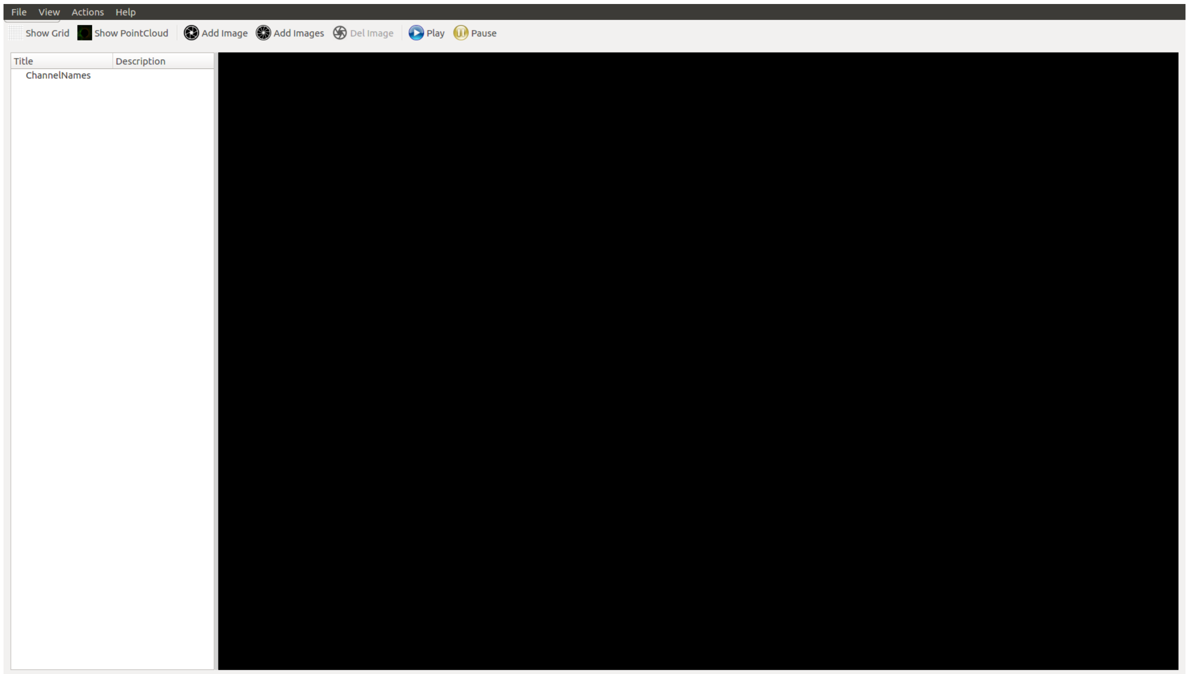
6.1.1 Install and run

`cyber_visualizer` is a visualization tool for displaying the channel data in Apollo Cyber RT.

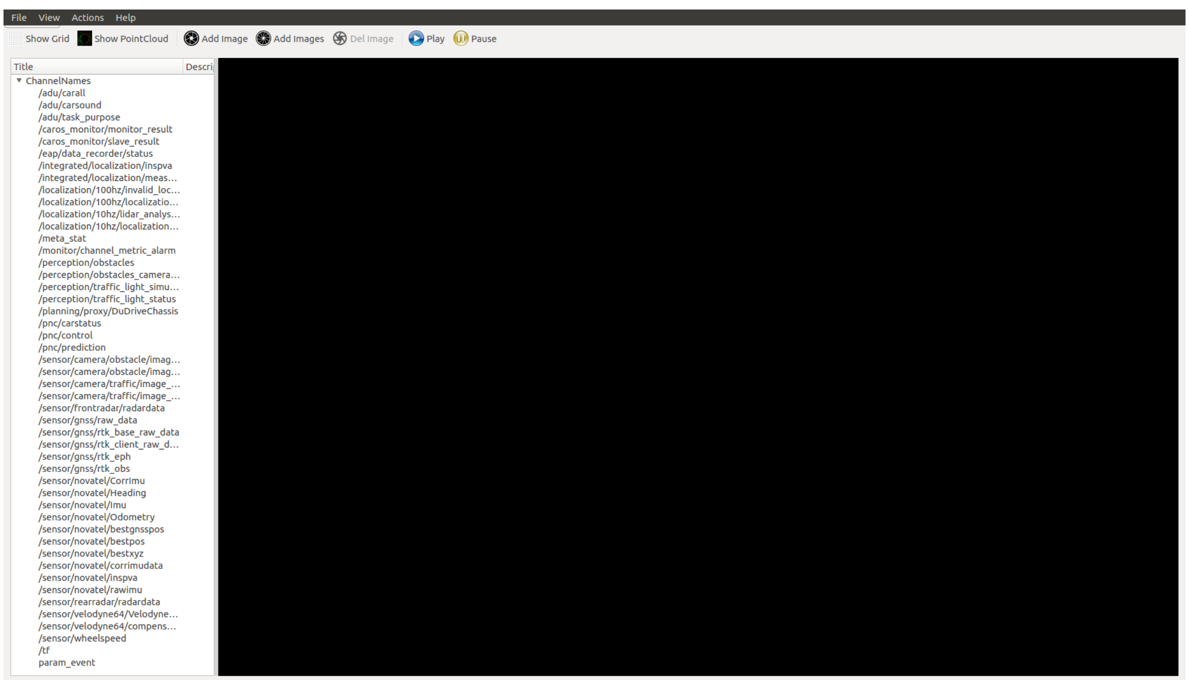
```
username@computername:~$: source /your-path-to-apollo-install-dir/cyber/setup.bash
username@computername:~$: cyber_visualizer
```

6.1.2 Interacting with cyber_visualizer

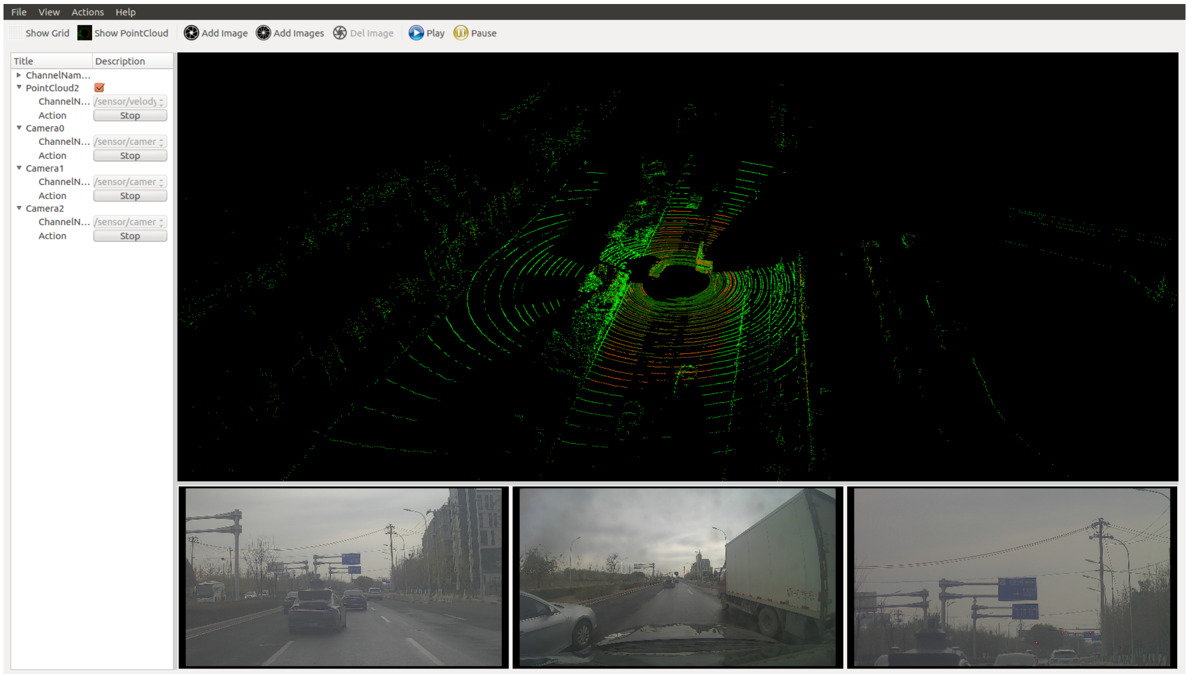
- After launching cyber_visualizer, you will see the following interface:



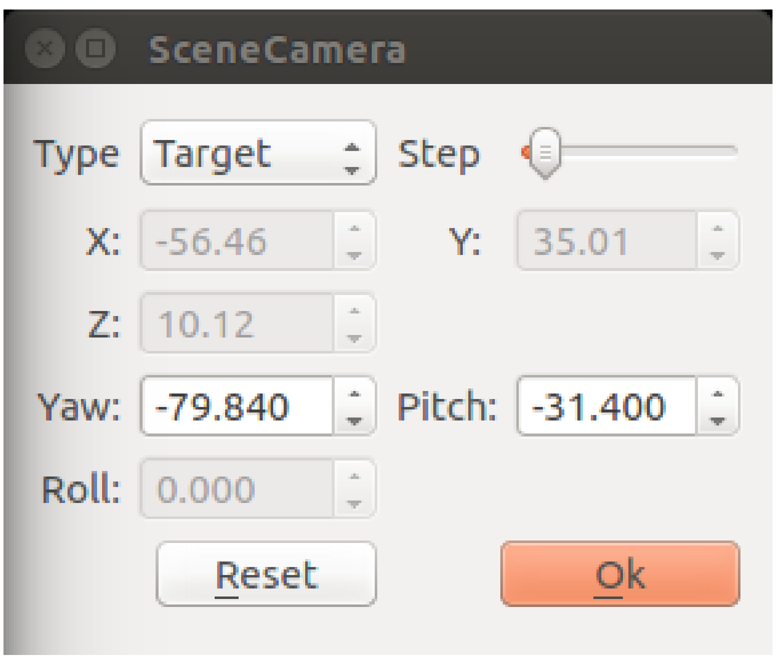
- When data flow through channels in Apollo Cyber RT, the list of all channels are displayed under `ChannelNames` as seen in the figure below. For example, you can use the record tool(cyber_recorder) of Apollo Cyber RT to replay data from another terminal, then `cyber_visualizer` will receive information of all active channels(from replay data) and display it.



- By clicking on options in toolbar, you can enable reference grid, display point clouds, add images, or display multiple camera's data at the same time. If you have `Show Grid` option enabled, you can set the color of the grid by double-clicking the `Color` item of the `Grid` list below `ChannelNames`. The default color is gray. You can also edit the value of `CellCount` to adjust the number of cells in the grid. As for a point cloud or an image, you can select the source channel through its `ChannelName` sub-item, and `Action` sub-item to play or stop the data from the corresponding channel. As shown in figure below, three cameras' channel data on the button sections and one point cloud channel data on the top section are displayed simultaneously.



- To adjust the virtual camera in the 3D point cloud scene, you can right click on the point cloud display section. A dialog box will pop up as shown in figure below.



The point cloud scene supports two types of cameras: Free and Target.(select Type from pop up dialog above)

- **Free type Camera**: For this type of camera in the point cloud scene, you can change the pose of the camera by holding down either left or right mouse button and move it. To change the pitch of camera, you can scroll the mouse wheel.
- **Target type Camera**: For this type of camera in the point cloud scene, to change the camera's viewing angle, you can hold down the left mouse button and then move it. To change the distance of the camera to

the observed point (the default observation point is the coordinate system origin (0, 0,0)), you can scroll the mouse wheel.

You can also modify the camera information directly in the dialog box to change the camera's observation status in the point cloud scene. And the "Step" item is the step value from the dialog box.

Place the mouse on the image of the camera channel, you can double-click the left button to highlight the corresponding data channel on the left menu bar. Right click on the image to bring up menu for deleting the camera channel.

Play and Pause buttons: when clicking the `Play` button, all channels will be showed. While when clicking the `Pause` button, all channels will stop showing on the tool.

6.2 Cyber_monitor

6.2.1 Install and run

The command line tool `cyber_monitor` provides a clear view of the list of real time channel information Apollo Cyber RT in the terminal.

```
username@computername:~$: source /your-path-to-apollo-install-dir/cyber/setup.bash
username@computername:~$: cyber_monitor
```

6.2.2 Useful commands

Display help information

Use the -h option to get help for cyber_monitor

```
username@computername:~$: cyber_monitor -h
```

Specify the channel

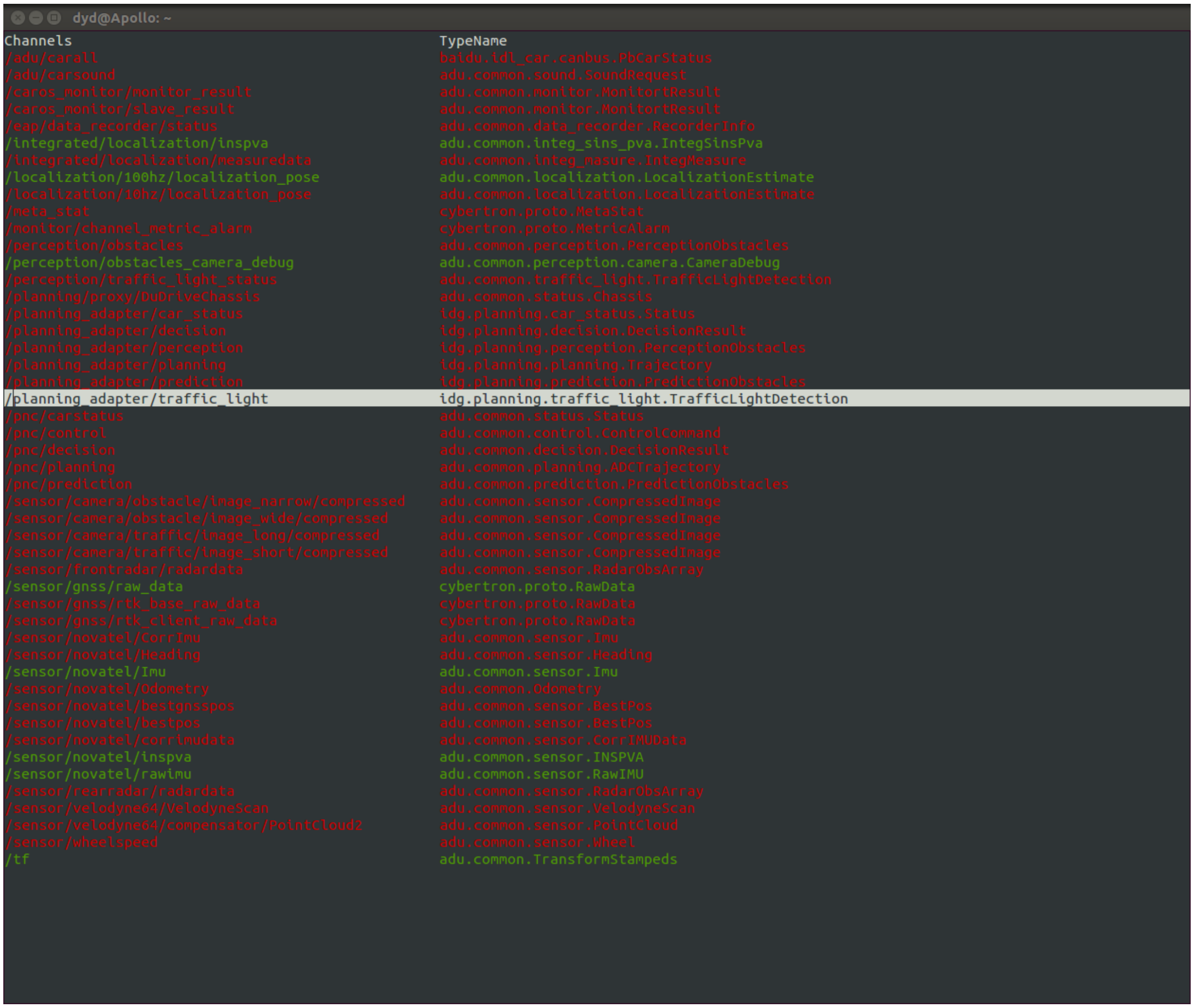
With the -c option, you can have cyber_monitor to monitor only a specified channel, such as:

```
username@computername:~$: cyber_monitor -c ChannelName
```

6.3 Get familiar with UI of cyber_monitor

After launching the command line tool, you will notice it is similar to cyber_visualizer. It automatically collects the information of all the channels through the topology and displays them in two columns (channel name, channel data type).

The default display for channel information is in red. However, if there is data flowing through the a channel, the corresponding line of the channel is displayed in green. As shown in the image below:



6.3.1 Interacting with cyber_monitor

Common commands

```
ESC | q key ---- Exit
Backspace ---- Back
h | H ---- Show help page
```

Common command for topology and channel

```
PageDown | Ctrl+d --- Next
PageUp | Ctrl+u --- Previous
Up, down or w, s keys ---- Move the current highlight line up and down
Right arrow or d key ---- Enter highlight line, display highlighted line data in
↪detail
Left arrow or a key ------ Return to the previous layer from the current
Enter key ----- Same as d key
```

Commands only for topology

```
f | F ----- Display frame rate
t | T ----- Display channel message type
Space ----- Close|Open channel (only valid for channels with data arrival; yellow␣
↪color after channel is closed)
```

Commands only for channel

```
i | I ----- Display channel Reader and Writer information
b | B ------ Display channel message content
```

View the repeated data field in a channel

```
n | N ---- Repeat the next data in the domain
m | M ---- Repeat one data on the domain
```

6.4 Cyber_recorder

`cyber_recorder` is a record/playback tool provided by Apollo Cyber RT. It provides many useful functions, including recording a record file, playing back a record file, splitting a record file, checking the information of record file and etc.

6.4.1 Install and run

Launch cyber_recorder:

```
$ source /your-path-to-apollo-install-dir/cyber/setup.bash
$ cyber_recorder
usage: cyber_recorder <command>> [<args>]
The cyber_recorder commands are:
    info                                Show information of an exist record.
    play                                Play an exist record.
    record                              Record same topic.
    split                               Split an exist record.
    recover                             Recover an exist record.
```

6.4.2 Commands of cyber_recorder

- To view the information of a record file:

```
$ cyber_recorder info -h
usage: cyber_recorder info [options]
        -h, --help                                  show help message
```

- To record a record file

```
$ cyber_recorder record -h
usage: cyber_recorder record [options]
    -o, --output <file>                output record file
    -a, --all                          all channels
    -c, --channel <name>               channel name
    -i, --segment-interval <seconds>   record segmented every n second(s)
    -m, --segment-size <MB>            record segmented every n megabyte(s)
    -h, --help                         show help message
```

- To play back a record file:

```
$ cyber_recorder play -h
usage: cyber_recorder play [options]
    -f, --file <file>                        input record file
    -a, --all                                play all
    -c, --white-channel <name>                only play the specified channel
    -k, --black-channel <name>                not play the specified channel
    -l, --loop                                loop play
    -r, --rate <1.0>                        multiply the play rate by FACTOR
    -b, --begin <2018-07-01 00:00:00>        play the record begin at
    -e, --end <2018-07-01 00:01:00>        play the record end at
    -s, --start <seconds>                play started at n seconds
    -d, --delay <seconds>                play delayed n seconds
    -p, --preload <seconds>                play after trying to preload n second(s)
    -h, --help                                show help message
```

- To split a record file:

```
$ cyber_recorder split -h
usage: cyber_recorder split [options]
    -f, --file <file>                  input record file
    -o, --output <file>                output record file
    -a, --all                          all channels
    -c, --channel <name>               channel name
    -b, --begin <2018-07-01 00:00:00>  begin at assigned time
    -e, --end <2018-07-01 01:00:00>    end at assigned time
```

- To repair a record file:

```
$ cyber_recorder recover -h
usage: cyber_recorder recover [options]
    -f, --file <file>                  input record file
    -o, --output <file>                output record file
```

6.4.3 Examples of using cyber_recorder

Check the details of a record file

```
$ cyber_recorder info demo.record
record_file:    demo.record
version:        1.0
duration:       19.995227 Seconds
begin_time:     2018-04-17 06:25:36
end_time:       2018-04-17 06:25:55
size:           28275479 Bytes (26.965598 MB)
```

(continues on next page)

(continued from previous page)

```
is_complete:    true
message_number: 15379
channel_number: 16
channel_info:   /apollo/localization/pose                             2000 messages :␣
↪apollo.localization.LocalizationEstimate
                /tf                                                   4000 messages :␣
↪apollo.transform.TransformStampeds
                /apollo/control                                       2000 messages :␣
↪apollo.control.ControlCommand
                /apollo/sensor/gnss/odometry                          2000 messages :␣
↪apollo.localization.Gps
                /apollo/canbus/chassis                                2000 messages :␣
↪apollo.canbus.Chassis
                /apollo/sensor/gnss/imu                               1999 messages :␣
↪apollo.drivers.gnss.Imu
                /apollo/sensor/gnss/rtk_obs                             41 messages :␣
↪apollo.drivers.gnss.EpochObservation
                /apollo/sensor/gnss/ins_stat                            20 messages :␣
↪apollo.drivers.gnss.InsStat
                /apollo/sensor/gnss/best_pose                           20 messages :␣
↪apollo.drivers.gnss.GnssBestPose
                /apollo/perception/obstacles                           400 messages :␣
↪apollo.perception.PerceptionObstacles
                /apollo/prediction                                     400 messages :␣
↪apollo.prediction.PredictionObstacles
                /apollo/sensor/conti_radar                             270 messages :␣
↪apollo.drivers.ContiRadar
                /apollo/planning                                       200 messages :␣
↪apollo.planning.ADCTrajectory
                /apollo/monitor/static_info                              1 messages :␣
↪apollo.data.StaticInfo
                /apollo/sensor/gnss/rtk_eph                             25 messages :␣
↪apollo.drivers.gnss.GnssEphemeris
                /apollo/monitor                                          3 messages :␣
↪apollo.common.monitor.MonitorMessage
```

Record a record file

```
$ cyber_recorder record -a
[RUNNING]  Record :     total channel num : 1  total msg num : 5
...
```

Replay a record file

```
$ cyber_recorder play -f 20180720202307.record
file: 20180720202307.record, chunk_number: 1, begin_time: 1532089398663399667, end_
↪time: 1532089404688079759, message_number: 75
please wait for loading and playing back record...
Hit Ctrl+C to stop replay, or Space to pause.
[RUNNING]  Record Time: 1532089404.688080    Progress: 6.024680 / 6.024680
play finished. file: 20180720202307.record
```

6.5 rosbag_to_record

`rosbag_to_record` is a tool which can convert rosbag to recorder file provided by Apollo Cyber RT. Now the tool support following channel:

```
/apollo/perception/obstacles
/apollo/planning
/apollo/prediction
/apollo/canbus/chassis
/apollo/control
/apollo/guardian
/apollo/localization/pose
/apollo/perception/traffic_light
/apollo/drive_event
/apollo/sensor/gnss/odometry
/apollo/monitor/static_info
/apollo/monitor
/apollo/canbus/chassis_detail
/apollo/control/pad
/apollo/navigation
/apollo/routing_request
/apollo/routing_response
/tf
/tf_static
/apollo/sensor/conti_radar
/apollo/sensor/delphi_esr
/apollo/sensor/gnss/best_pose
/apollo/sensor/gnss/imu
/apollo/sensor/gnss/ins_stat
/apollo/sensor/gnss/rtk_eph
/apollo/sensor/gnss/rtk_obs
/apollo/sensor/velodyne64/compensator/PointCloud2
```

6.5.1 Install and run

Launch rosbag_to_record:

```
$ source /your-path-to-apollo-install-dir/cyber/setup.bash
$ rosbag_to_record
Usage:
  rosbag_to_record input.bag output.record
```

6.5.2 Example

We can convert Apollo2.5 demo bag to record file.

```
$ rosbag_to_record demo_2.5.bag demo.record
record_file:    demo.record
version:        1.0
duration:       19.995227 Seconds
begin_time:     2018-04-17 06:25:36
end_time:       2018-04-17 06:25:55
size:           28275479 Bytes (26.965598 MB)
is_complete:    true
```

(continues on next page)

(continued from previous page)

```
message_number: 15379
channel_number: 16
channel_info:   /apollo/localization/pose                                2000 messages :␣
↪apollo.localization.LocalizationEstimate
                /tf                                                      4000 messages :␣
↪apollo.transform.TransformStampeds
                /apollo/control                                          2000 messages :␣
↪apollo.control.ControlCommand
                /apollo/sensor/gnss/odometry                             2000 messages :␣
↪apollo.localization.Gps
                /apollo/canbus/chassis                                   2000 messages :␣
↪apollo.canbus.Chassis
                /apollo/sensor/gnss/imu                                  1999 messages :␣
↪apollo.drivers.gnss.Imu
                /apollo/sensor/gnss/rtk_obs                                41 messages :␣
↪apollo.drivers.gnss.EpochObservation
                /apollo/sensor/gnss/ins_stat                               20 messages :␣
↪apollo.drivers.gnss.InsStat
                /apollo/sensor/gnss/best_pose                              20 messages :␣
↪apollo.drivers.gnss.GnssBestPose
                /apollo/perception/obstacles                              400 messages :␣
↪apollo.perception.PerceptionObstacles
                /apollo/prediction                                        400 messages :␣
↪apollo.prediction.PredictionObstacles
                /apollo/sensor/conti_radar                                270 messages :␣
↪apollo.drivers.ContiRadar
                /apollo/planning                                          200 messages :␣
↪apollo.planning.ADCTrajectory
                /apollo/monitor/static_info                                 1 messages :␣
↪apollo.data.StaticInfo
                /apollo/sensor/gnss/rtk_eph                                25 messages :␣
↪apollo.drivers.gnss.GnssEphemeris
                /apollo/monitor                                             3 messages :␣
↪apollo.common.monitor.MonitorMessage
Conversion finished! Took 0.505623051 seconds in total.
```

CHAPTER

SEVEN

DEVELOP INSIDE DOCKER ENVIRONMENT

To make life easier, Apollo Cyber RT has released a docker image and a number of scripts to help developers to build and play with Cyber RT.

The official Cyber RT docker image is built based on Ubuntu 18.04. It comes with the full support for building Cyber RT and the drivers on top of it. So if you are interested in Cyber RT only, that would be the ideal point to start with.

Note: ARM platform support has added recently fully integrated with Apollo development environment. You will be able to develop Cyber RT on both x86 and ARM platform with the same set of scripts. However, since ARM platform has only been verified on Nvidia Drive PX, so if you are trying on some dev board other than Drive PX. Please let us know if you run into any issues

The following sections will show to how to start and play with Cyber RT docker and also how to build your own docker image from scratch.

7.1 Build and Test with Cyber RT in Docker

To start the official Cyber RT docker, you need to run the command below first:

Note: Running this command for the first time could take a while because you will be downloading the full docker image, depending on your network bandwidth.

Note: You will lose all your previous changes in the docker if you have ran this command before. Unless you would like to start a fresh docker environment.

```
./docker/scripts/cyber_start.sh
```

To enter the docker you just started:

Note: you can enter and exit multiple times whenever you like, the docker environment will stay there until the next time you start the docker again.

```
./docker/scripts/cyber_into.sh
```

To build Cyber RT only and test it:

```
./apollo.sh build_cyber
bazel test cyber/...
```

You should be able to see all the tests passed before developing your project.

To build drivers on Cyber RT only:

```
./apollo.sh build_drivers
```

Note: start of instructions for ARM platform only

Due to some limitation of docker on Drive PX platform, you need to follow the steps below on top of the procedure above.

For the first time after running cyber_into.sh to get into the Cyber RT container, please run the following two commands:

```
/apollo/scripts/docker_adduser.sh
su nvidia
```

To exit, please use `ctrl+p ctrl+q` instead of `exit`. Otherwise, you will lose your current running container.

Note: end instructions for ARM platform only

7.2 Build Cyber RT Docker Image

To build your owner docker image for Cyber RT, please run the following commands on x86 platform:

```
cd docker/build/
./build_cyber.sh cyber.x86_64.dockerfile
```

For ARM platform,

```
cd docker/build/
./build_cyber.sh cyber.aarch64.dockerfile
```

To save you some time due to the performance on ARM platform, you can add the following option to download some prebuilt packages.

```
cd docker/build/
./build_cyber.sh cyber.aarch64.dockerfile download
```

If you would like to save your docker image for long term, use the following commands as example to save it in your own docker registry, please use "docker images" to find the name of your own image.

```
docker push [YOUR_OWN_DOCKER_REGISTRY]:cyber-x86_64-18.04-20190531_1115
```

CHAPTER

EIGHT

MIGRATION GUIDE FROM APOLLO ROS

This article describes the essential changes for projects to migrate from Apollo ROS (Apollo 3.0 and before) to Apollo Cyber RT (Apollo 3.5 and after). We will be using the very first ROS project talker/listener as example to demostrate step by step migration instruction.

8.1 Build system

ROS use `CMake` as its build system but Cyber RT use `bazel`. In a ROS project, CmakeLists.txt and package.xml are required for defining build configs like build target, dependency, message files and so on. As for a Cyber RT component, a single bazel BUILD file covers. Some key build config mappings are listed below.

Cmake

```
project(pb_msgs_example)
add_proto_files(
  DIRECTORY proto
  FILES chatter.proto
)
## Declare a C++ executable
add_executable(pb_talker src/talker.cpp)
target_link_libraries(pb_talker ${catkin_LIBRARIES}pb_msgs_example_proto)
add_executable(pb_listener src/listener.cpp)
target_link_libraries(pb_listener ${catkin_LIBRARIES}  pb_msgs_example_proto)
```

Bazel

```
cc_binary(
  name = "talker",
  srcs = ["talker.cc"],
  deps = [
    "//cyber",
    "//cyber/examples/proto:examples_cc_proto",
    ],
  )
cc_binary(
  name = "listener",
  srcs = ["listener.cc"],
  deps = [
    "//cyber",
    "//cyber/examples/proto:examples_cc_proto",
    ],
  )
```

We can find the mapping easily from the 2 file snippets. For example, `pb_talker` and `src/talker.cpp` in cmake `add_executable` setting map to `name = "talker"` and `srcs = ["talker.cc"]` in BUILD file `cc_binary`.

8.1.1 Proto

Apollo ROS has customized to support proto message formate that a separate section `add_proto_files` and projectName_proto(`pb_msgs_example_proto`) in `target_link_libraries` are required to send message in proto formate. For config proto message in Cyber RT, it's as simple as adding the target proto file path concantenated with name of `cc_proto_library` in `deps` setting. The `cc_proto_library` is set up in BUILD file under proto folder.

```
cc_proto_library(
  name = "examples_cc_proto",
  deps = [
    ":examples_proto",
  ],
)
proto_library(
  name = "examples_proto",
  srcs = [
    "examples.proto",
  ],
)
```

The package definition has also changed in Cyber RT. In Apollo ROS a fixed package `package pb_msgs;` is used for proto files, but in Cyber RT, the proto file path `package apollo.cyber.examples.proto;` is used instead.

8.2 Folder structure

As shown below, Cyber RT remove the src folder and pull all source code in the same folder as BUILD file. BUILD file plays the same role as CMakeLists.txt plus package.xml. Both Cyber RT and Apollo ROS talker/listener example have a proto folder for message proto files but Cyber RT requires a separate BUILD file for proto folder to set up the proto library.

8.2.1 Apollo ROS

- CMakeLists.txt
- package.xml
- proto
  - chatter.proto
- src
  - listener.cpp
  - talker.cpp

8.2.2 Cyber RT

- BUILD
- listener.cc
- talker.cc
- proto
  - BUILD
  - examples.proto (with chatter message)

8.3 Update source code

8.3.1 Listener

Cyber RT

```
#include "cyber/cyber.h"
#include "cyber/examples/proto/examples.pb.h"

void MessageCallback(
    const std::shared_ptr<apollo::cyber::examples::proto::Chatter>& msg) {
  AINFO << "Received message seq-> " << msg->seq();
  AINFO << "msgcontent->" << msg->content();
}

int main(int argc, char* argv[]) {
  // init cyber framework
  apollo::cyber::Init(argv[0]);
  // create listener node
  auto listener_node = apollo::cyber::CreateNode("listener");
  // create listener
  auto listener =
      listener_node->CreateReader<apollo::cyber::examples::proto::Chatter>(
          "channel/chatter", MessageCallback);
  apollo::cyber::WaitForShutdown();
  return 0;
}
```

ROS

```
#include "ros/ros.h"
#include "chatter.pb.h"

void MessageCallback(const boost::shared_ptr<pb_msgs::Chatter>& msg) {
  ROS_INFO_STREAM("Time: " << msg->stamp().sec() << "." << msg->stamp().nsec());
  ROS_INFO("I heard pb Chatter message: [%s]", msg->content().c_str());
}

int main(int argc, char** argv) {
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber pb_sub = n.subscribe("chatter", 1000, MessageCallback);
  ros::spin();
```

(continues on next page)

(continued from previous page)

```
  return 0;
}
```

You can see easily from the two listener code above that Cyber RT provides very similar API to for developers to migrate from ROS.

- `ros::init(argc, argv, "listener");` –> `apollo::cyber::Init(argv[0]);`
- `ros::NodeHandle n;` –> `auto listener_node = apollo::cyber::CreateNode("listener");`
- `ros::Subscriber pb_sub = n.subscribe("chatter", 1000, MessageCallback);` –> `auto listener = listener_node->CreateReader("channel/chatter", MessageCallback);`
- `ros::spin();` –> `apollo::cyber::WaitForShutdown();`

Note: for Cyber RT, a listener node has to use `node->CreateReader<messageType>(channelName, callback)` to read data from channel.

8.3.2 Talker

Cyber RT

```
#include "cyber/cyber.h"
#include "cyber/examples/proto/examples.pb.h"

using apollo::cyber::examples::proto::Chatter;

int main(int argc, char *argv[]) {
  // init cyber framework
  apollo::cyber::Init(argv[0]);
  // create talker node
  auto talker_node = apollo::cyber::CreateNode("talker");
  // create talker
  auto talker = talker_node->CreateWriter<Chatter>("channel/chatter");
  Rate rate(1.0);
  while (apollo::cyber::OK()) {
    static uint64_t seq = 0;
    auto msg = std::make_shared<Chatter>();
    msg->set_timestamp(Time::Now().ToNanosecond());
    msg->set_lidar_timestamp(Time::Now().ToNanosecond());
    msg->set_seq(seq++);
    msg->set_content("Hello, apollo!");
    talker->Write(msg);
    AINFO << "talker sent a message!";
    rate.Sleep();
  }
  return 0;
}
```

ROS

```
#include "ros/ros.h"
#include "chatter.pb.h"

#include <sstream>
```

(continues on next page)

(continued from previous page)

```
int main(int argc, char** argv) {
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<pb_msgs::Chatter>("chatter", 1000);
  ros::Rate loop_rate(10);
  int count = 0;
  while (ros::ok()) {
    pb_msgs::Chatter msg;
    ros::Time now = ros::Time::now();
    msg.mutable_stamp()->set_sec(now.sec);
    msg.mutable_stamp()->set_nsec(now.nsec);
    std::stringstream ss;
    ss << "Hello world " << count;
    msg.set_content(ss.str());
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

Most of the mappings are illustrated in listener code above, the rest are listed here.

- `ros::Publisher chatter_pub = n.advertise<pb_msgs::Chatter>("chatter", 1000);` -> `auto talker = talker_node->CreateWriter<Chatter>("channel/chatter");`
- `chatter_pub.publish(msg);` -> `talker->Write(msg);`

8.4 Tools mapping

ROS | Cyber RT | Note :————- | :————- | :————– rosbag | cyber_recorder | data file scripts/diagnostics.sh | cyber_monitor | channel debug offline_lidar_visualizer_tool | cyber_visualizer |point cloud visualizer

8.5 ROS bag data migration

The data file changed from ROS bag to Cyber record in Cyber RT. Cyber RT has a data migration tool `rosbag_to_record` for users to easily migrate data files before Apollo 3.0 (ROS) to Cyber RT like the sample usage below.

```
rosbag_to_record demo_3.0.bag demo_3.5.record
```

CHAPTER

NINE

C++ API

Topological communication APIs is actually implemented in `node` and `reader/writer` and `client/service`.

9.1 cyber/node/node.h

Defined in `cyber/node/node.h`

class Node

Node is the fundamental building block of Cyber RT.

every module contains and communicates through the node. A module can have different types of communication by defining read/write and/or service/client in a node.

Warning Duplicate name is not allowed in topo objects, such as node, reader/writer, service/clinet in the topo.

Public Functions

const std::string &**Name**() **const**

Get node's name.

Warning duplicate node name is not allowed in the topo.

template<typename **MessageT**>
auto **CreateWriter**(**const** proto::RoleAttributes &*role_attr*)

Create a *Writer* with specific message type.

Return std::shared_ptr<Writer<MessageT>> result *Writer* Object

Template Parameters

- `MessageT`: Message Type

Parameters

- `role_attr`: is a protobuf message RoleAttributes, which includes the channel name and other info.

template<typename **MessageT**>
auto **CreateWriter**(**const** std::string &*channel_name*)

Create a *Writer* with specific message type.

Return std::shared_ptr<Writer<MessageT>> result *Writer* Object

Template Parameters

- `MessageT`: Message Type

Parameters

- `channel_name`: the channel name to be published.

template<typename **MessageT**>
auto **CreateReader**(**const** std::string &*channel_name*, **const** CallbackFunc<*MessageT*> &*reader_func* = nullptr)
Create a *Reader* with specific message type with channel name qos and other configs used will be default.

Return std::shared_ptr<cyber::Reader<MessageT>> result *Reader* Object

Template Parameters

- `MessageT`: Message Type

Parameters

- `channel_name`: the channel of the reader subscribed.
- `reader_func`: invoked when message receive invoked when the message is received.

template<typename **MessageT**>
auto **CreateReader**(**const** *ReaderConfig* &*config*, **const** CallbackFunc<*MessageT*> &*reader_func* = nullptr)
Create a *Reader* with specific message type with reader config.

Return std::shared_ptr<cyber::Reader<MessageT>> result *Reader* Object

Template Parameters

- `MessageT`: Message Type

Parameters

- `config`: instance of *`ReaderConfig`*, include channel name, qos and pending queue size
- `reader_func`: invoked when message receive

template<typename **MessageT**>
auto **CreateReader**(**const** proto::RoleAttributes &*role_attr*, **const** CallbackFunc<*MessageT*> &*reader_func* = nullptr)
Create a *Reader* object with `RoleAttributes`

Return std::shared_ptr<cyber::Reader<MessageT>> result *Reader* Object

Template Parameters

- `MessageT`: Message Type

Parameters

- `role_attr`: instance of `RoleAttributes`, includes channel name, qos, etc.
- `reader_func`: invoked when message receive

template<typename **Request**, typename **Response**>
auto **CreateService**(**const** std::string &*service_name*, **const typename** Service<*Request*, *Response*>::ServiceCallback &*service_callback*)
Create a *Service* object with specific `service_name`

Return std::shared_ptr<Service<Request, Response>> result `Service`

Template Parameters

- `Request`: Message Type of the Request
- `Response`: Message Type of the Response

Parameters

- `service_name`: specific service name to a serve
- `service_callback`: invoked when a service is called

template<typename **Request**, typename **Response**>
auto **CreateClient**(**const** std::string &*service_name*)

Create a *Client* object to request *Service* with `service_name`

Return std::shared_ptr<Client<Request, Response>> result `Client`

Template Parameters

- `Request`: Message Type of the Request
- `Response`: Message Type of the Response

Parameters

- `service_name`: specific service name to a *Service*

void **Observe**()

Observe all readers' data.

void **ClearData**()

clear all readers' data

template<typename **MessageT**>
auto **GetReader**(**const** std::string &*channel_name*)

Get the *Reader* object that subscribe `channel_name`

Return std::shared_ptr<Reader<MessageT>> result reader

Template Parameters

- `MessageT`: Message Type

Parameters

- `channel_name`: channel name

9.2 cyber/node/reader_base.h

Defined in `cyber/node/reader_base.h`

class ReaderBase

Base Class for *Reader* *Reader* is identified by one apollo::cyber::proto::RoleAttribute, it contains the channel_name, channel_id that we subscribe, and host_name, process_id and node that we are located, and qos that describes our transportation quality.

Subclassed by *apollo::cyber::Reader< MessageT >*

Public Functions

virtual bool **Init**() = 0
Init the *Reader* object.

Return true if init successfully

Return false if init failed

virtual void **Shutdown**() = 0
Shutdown the *Reader* object.

virtual void **ClearData**() = 0
Clear local data.

virtual void **Observe**() = 0
Get stored data.

virtual bool **Empty**() **const** = 0
Query whether the *Reader* has data to be handled.

Return true if data container is empty

Return false if data container has data

virtual bool **HasReceived**() **const** = 0
Query whether we have received data since last clear.

Return true if the reader has received data

Return false if the reader has not received data

virtual double **GetDelaySec**() **const** = 0
Get time interval of since last receive message.

Return double seconds delay

virtual uint32_t **PendingQueueSize**() **const** = 0
Get the value of pending queue size.

Return uint32_t result value

virtual bool **HasWriter**()
Query is there any writer that publish the subscribed channel.

Return true if there is at least one *Writer* publish the channel

Return false if there is no *Writer* publish the channel

virtual void **GetWriters**(std::vector<proto::RoleAttributes> **writers*)
Get all writers pushlish the channel we subscribes.

Parameters

- `writers`: result RoleAttributes vector

const std::string &**GetChannelName**() **const**
Get *Reader*'s Channel name.

Return const std::string& channel name

uint64_t **ChannelId**() **const**
Get *Reader*'s Channel id.

Return uint64_t channel id

const proto::QosProfile &**QosProfile**() **const**
Get qos profile.

You can see qos description

Return const proto::QosProfile& result qos

bool **IsInit**() **const**
Query whether the *Reader* is initialized.

Return true if the *Reader* has been inited

Return false if the *Reader* has not been inited

9.3 cyber/node/reader.h

Defined in `cyber/node/reader.h`

template<typename **MessageT**>
class Reader : **public** apollo::cyber::*ReaderBase*
Reader subscribes a channel, it has two main functions:

1. You can pass a `CallbackFunc` to handle the message then it arrived
2. You can Observe messages that Blocker cached. *Reader* automatically push the message to Blocker's `PublishQueue`, and we can use `Observe` to fetch messages from `PublishQueue` to `ObserveQueue`. But, if you have set CallbackFunc, you can ignore this. One *Reader* uses one `ChannelBuffer`, the message we are handling is stored in ChannelBuffer *Reader* will Join the topology when init and Leave the topology when shutdown

Warning To save resource, `ChannelBuffer` has limited length, it's passed through the `pending_queue_size` param. pending_queue_size is default set to 1, So, If you handle slower than writer sending, older messages that are not handled will be lost. You can increase `pending_queue_size` to resolve this problem.

Public Functions

Reader(**const** proto::RoleAttributes &*role_attr*, **const** CallbackFunc<MessageT> &*reader_func* = nullptr, uint32_t *pending_queue_size* = DEFAULT_PENDING_QUEUE_SIZE)
Constructor a *Reader* object.

Warning the received messages is enqueue a queue,the queue's depth is pending_queue_size

Parameters

- `role_attr`: is a protobuf message RoleAttributes, which includes the channel name and other info.
- `reader_func`: is the callback function, when the message is received.
- `pending_queue_size`: is the max depth of message cache queue.

bool **Init**()

Init *Reader*.

Return true if init successfully

Return false if init failed

void **Shutdown**()

Shutdown *Reader*.

void **Observe**()

Get All data that `Blocker` stores.

void **ClearData**()

Clear `Blocker`'s data.

bool **HasReceived**() **const**

Query whether we have received data since last clear.

Return true if the reader has received data

Return false if the reader has not received data

bool **Empty**() **const**

Query whether the *Reader* has data to be handled.

Return true if blocker is empty

Return false if blocker has data

double **GetDelaySec**() **const**

Get time interval of since last receive message.

Return double seconds delay

uint32_t **PendingQueueSize**() **const**

Get pending_queue_size configuration.

Return uint32_t the value of pending queue size

void **Enqueue**(**const** std::shared_ptr<MessageT> &*msg*)

Push `msg` to Blocker's `PublishQueue`

Parameters

- `msg`: message ptr to be pushed

void **SetHistoryDepth**(**const** uint32_t &*depth*)

Set Blocker's `PublishQueue`'s capacity to `depth`

Parameters

- `depth`: the value you want to set

uint32_t **GetHistoryDepth**() **const**

Get Blocker's `PublishQueue`'s capacity.

Return uint32_t depth of the history

std::shared_ptr<MessageT> **GetLatestObserved**() **const**

Get the latest message we `Observe`

Return std::shared_ptr<MessageT> the latest message

std::shared_ptr<MessageT> **GetOldestObserved**() **const**

Get the oldest message we `Observe`

Return std::shared_ptr<MessageT> the oldest message

virtual Iterator **Begin**() **const**

Get the begin iterator of `ObserveQueue`, used to traverse.

Return Iterator begin iterator

virtual Iterator **End**() **const**

Get the end iterator of `ObserveQueue`, used to traverse.

Return Iterator begin iterator

bool **HasWriter**()

Is there is at least one writer publish the channel that we subscribes?

Return true if the channel has writer

Return false if the channel has no writer

void **GetWriters**(std::vector<proto::RoleAttributes> **writers*)

Get all writers pushlish the channel we subscribes.

Parameters

- `writers`: result vector of RoleAttributes

9.4 cyber/node/writer_base.h

Defined in `cyber/node/writer_base.h`

class WriterBase

Base class for a *Writer*.

A *Writer* is an object to send messages through a 'Channel'

Warning One *Writer* can only write one channel. But different writers can write through the same channel

Subclassed by *apollo::cyber::Writer< MessageT >*

Public Functions

WriterBase (**const** proto::RoleAttributes &*role_attr*)
Construct a new *Writer* Base object.

Parameters

- `role_attr`: role attributes for this *Writer*

virtual bool **Init** () = 0
Init the *Writer*.

Return true if init success

Return false if init failed

virtual void **Shutdown** () = 0
Shutdown the *Writer*.

virtual bool **HasReader** ()
Is there any *Reader* that subscribes our Channel? You can publish message when this return true.

Return true if the channel has reader

Return false if the channel has no reader

virtual void **GetReaders** (std::vector<proto::RoleAttributes> **readers*)
Get all Readers that subscriber our writing channel.

Parameters

- `readers`: result vector of RoleAttributes

const std::string &**GetChannelName** () **const**
Get *Writer*'s Channel name.

Return const std::string& const reference to the channel name

bool **IsInit** () **const**
Is *Writer* initialized?

Return true if the *Writer* is inited

Return false if the Write is not inited

9.5 cyber/node/writer.h

Defined in `cyber/node/writer.h`

template<typename **MessageT**>
class Writer : **public** apollo::cyber::*WriterBase*

Public Functions

Writer (**const** proto::RoleAttributes &*role_attr*)
Construct a new *Writer* object.

Parameters

- `role_attr`: we use RoleAttributes to identify a *Writer*

bool **Init** ()
Init the *Writer*.

Return true if init successfully

Return false if init failed

void **Shutdown** ()
Shutdown the *Writer*.

bool **Write** (**const** MessageT &*msg*)
Write a MessageT instance.

Return true if write successfully

Return false if write failed

Parameters

- `msg`: the message we want to write

bool **Write** (**const** std::shared_ptr<MessageT> &*msg_ptr*)
Write a shared ptr of MessageT.

Return true if write successfully

Return false if write failed

Parameters

- `msg_ptr`: the message shared ptr we want to write

bool **HasReader** ()
Is there any *Reader* that subscribes our Channel? You can publish message when this return true.

Return true if the channel has reader

Return false if the channel has no reader

void **GetReaders** (std::vector<proto::RoleAttributes> **readers*)
Get all Readers that subscriber our writing channel.

Parameters

- `readers`: vector result of RoleAttributes

9.6 cyber/node/node_channel_impl.h

Defined in `cyber/node/node_channel_impl.h`

struct ReaderConfig

Public Functions

ReaderConfig()

< configurations for a *Reader*

Public Members

uint32_t **pending_queue_size**

configuration for responding ChannelBuffer.

Older messages will dropped if you have no time to handle

class NodeChannelImpl

The implementation for *Node* to create Objects connected by Channels.

e.g. Channel *Reader* and *Writer*

Public Functions

NodeChannelImpl(**const** std::string &*node_name*)

Construct a new *Node* Channel Impl object.

Parameters

- `node_name`: node name

virtual ~NodeChannelImpl()

Destroy the *Node* Channel Impl object.

const std::string &**NodeName**() **const**

get name of this node

Return const std::string& actual node name

9.7 cyber/node/node_service_impl.h

Defined in `cyber/node/node_service_impl.h`

class NodeServiceImpl

The implementation for *Node* to create Objects connected by Param.

e.g. Param Server and *Client*

Public Functions

NodeServiceImpl(**const** std::string &*node_name*)
Construct a new *Node Service* Impl object.

Parameters

- `node_name`: node name

NodeServiceImpl()
Forbid default-constructor.

~NodeServiceImpl()
Destroy the *Node Service* Impl object.

9.8 cyber/parameter/parameter.h

Defined in `cyber/parameter/parameter.h`

class Parameter
A *Parameter* holds an apollo::cyber::proto::Param, It's more human-readable, you can use basic-value type and Protobuf values to construct a paramter.

Parameter is identified by their `name`, and you can get *Parameter* content by call *value()*

Public Functions

Parameter()
Empty constructor.

Parameter(**const** *Parameter* &*parameter*)
copy constructor

Parameter(**const** std::string &*name*)
construct with paramter's name

Parameters

- `name`: *Parameter* name

Parameter(**const** std::string &*name*, **const** bool *bool_value*)
construct with paramter's name and bool value type

Parameters

- `name`: *Parameter* name
- `bool_value`: bool value

Parameter(**const** std::string &*name*, **const** int *int_value*)
construct with paramter's name and int value type

Parameters

- `name`: *Parameter* name

- `int_value`: int value

Parameter(**const** std::string &*name*, **const** int64_t *int_value*)
construct with paramter's name and int value type

Parameters

- `name`: *Parameter* name
- `int_value`: int value

Parameter(**const** std::string &*name*, **const** float *float_value*)
construct with paramter's name and float value type

Parameters

- `name`: *Parameter* name
- `float_value`: float value

Parameter(**const** std::string &*name*, **const** double *double_value*)
construct with paramter's name and double value type

Parameters

- `name`: *Parameter* name
- `double_value`: double value

Parameter(**const** std::string &*name*, **const** std::string &*string_value*)
construct with paramter's name and string value type

Parameters

- `name`: *Parameter* name
- `string_value`: string value

Parameter(**const** std::string &*name*, **const** char **string_value*)
construct with paramter's name and char* value type

Parameters

- `name`: *Parameter* name
- `string_value`: char* value

Parameter(**const** std::string &*name*, **const** std::string &*msg_str*, **const** std::string &*full_name*, **const** std::string &*proto_desc*)
use a protobuf type value to construct the parameter

Parameters

- `name`: *Parameter* name
- `msg_str`: protobuf contents
- `full_name`: the proto full name
- `proto_desc`: the proto's description

Parameter(**const** std::string &*name*, **const** google::protobuf::Message &*msg*)
use a google::protobuf::Message type value to construct the parameter

Parameters

- `name`: *Parameter* name
- `msg`: protobuf message

void **FromProtoParam**(**const** Param &*param*)
Parse a cyber::proto::Param object to cyber::parameter::Parameter object.

Return True if parse ok, otherwise False

Parameters

- `param`: The cyber::proto::Param object parse from A pointer to the target *Parameter* object

Param **ToProtoParam**() **const**
Parse a cyber::parameter::Parameter object to cyber::proto::Param object.

Return The target cyber::proto::Param object

ParamType **Type**() **const**
Get the cyber:parameter::ParamType of this object.

Return result cyber:parameter::ParameterType

std::string **TypeName**() **const**
Get Paramter's type name, i.e.

INT,DOUBLE,STRING or protobuf message's fullname

Return std::string the *Parameter*'s type name

std::string **Descriptor**() **const**
Get Paramter's descriptor, only work on protobuf types.

Return std::string the *Parameter*'s type name

const std::string **Name**() **const**
Get the *Parameter* name.

Return const std::string the *Parameter*'s name

bool **AsBool**() **const**
Get Paramter as a bool value.

Return true result

Return false result

int64_t **AsInt64**() **const**
Get Paramter as an int64_t value.

Return int64_t int64 type result

double **AsDouble**() **const**
: et Paramter as a double value

  Return double type result

const std::string **AsString**() **const**
: Get Paramter as a string value.

  Return const std::string *Parameter*'s string expression

std::string **DebugString**() **const**
: show debug string

  Return std::string *Parameter*'s debug string

template<typename **ValueType**>
std::enable_if<std::is_same<*ValueType*, bool>::value, bool>::type **value**() **const**
: Translate paramter value as a protobuf::Message.

  Return std::enable_if< std::is_base_of<google::protobuf::Message, ValueType>::value, ValueType>::type protobuf::Message type result

  Template Parameters

  - `ValueType`: type of the value

template<typename **ValueType**>
std::enable_if<std::is_integral<*ValueType*>::value && !std::is_same<*ValueType*, bool>::value, *ValueType*>::type **value**() **const**
: Translate paramter value to int type.

  Return std::enable_if<std::is_integral<ValueType>::value && !std::is_same<ValueType, bool>::value, ValueType>::type int type result

  Template Parameters

  - `ValueType`: type of the value

template<typename **ValueType**>
std::enable_if<std::is_floating_point<*ValueType*>::value, *ValueType*>::type **value**() **const**
: Translate paramter value to bool type.

  Return std::enable_if<std::is_floating_point<ValueType>::value, ValueType>::type floating type result

  Template Parameters

  - `ValueType`: type of the value

template<typename **ValueType**>
std::enable_if<std::is_convertible<*ValueType*, std::string>::value, **const** std::string&>::type **value**() **const**
: Translate paramter value to string type.

Return std::enable_if<std::is_convertible<ValueType, std::string>::value, const std::string&>::type string type result

Template Parameters

- `ValueType`: type of the value

template<typename **ValueType**>
std::enable_if<std::is_same<*ValueType*, bool>::value, bool>::type **value**() **const**
Translate paramter value to bool type.

Return std::enable_if<std::is_same<ValueType, bool>::value, bool>::type bool type result

Template Parameters

- `ValueType`: type of the value

9.9 cyber/parameter/parameter_server.h

Defined in `cyber/parameter/parameter_server.h`

class ParameterServer
Parameter Service is a very important function of auto-driving.

If you want to set a key-value, and hope other nodes to get the value, Routing, sensor internal/external references are set by *Parameter Service ParameterServer* can set a parameter, and then you can get/list paramter(s) by start a *ParameterClient* to send responding request

Warning You should only have one *ParameterServer* works

Public Functions

ParameterServer(**const** std::shared_ptr<*Node*> &*node*)
Construct a new *ParameterServer* object.

Parameters

- `node`: shared_ptr of the node handler

void **SetParameter**(**const** *Parameter* &*parmeter*)
Set the *Parameter* object.

Parameters

- `parmeter`: parameter to be set

bool **GetParameter**(**const** std::string &*parameter_name*, *Parameter* **parameter*)
Get the *Parameter* object.

Return true get parameter success

Return false parameter not exists

Parameters

- `parameter_name`: name of the parameer want to get

- `parameter`: pointer to store parameter want to get

void **ListParameters**(std::vector<*Parameter*> **parameters*)
get all the parameters

Parameters

- `parameters`: result Paramter vector

9.10 cyber/parameter/parameter_client.h

Defined in `cyber/parameter/parameter_client.h`

class ParameterClient
Parameter Client is used to set/get/list parameter(s) by sending a request to *ParameterServer*.

Public Functions

ParameterClient(**const** std::shared_ptr<*Node*> &*node*, **const** std::string &*service_node_name*)
Construct a new *ParameterClient* object.

Parameters

- `node`: shared_ptr of the node handler
- `service_node_name`: node name which provide a param services

bool **GetParameter**(**const** std::string &*param_name*, *Parameter* **parameter*)
Get the *Parameter* object.

Return true

Return false call service fail or timeout

Parameters

- `param_name`:
- `parameter`: the pointer to store

bool **SetParameter**(**const** *Parameter* &*parameter*)
Set the *Parameter* object.

Return true set parameter succues

Return false 1. call service timeout

1. parameter not exists The corresponding log will be recorded at the same time

Parameters

- `parameter`: parameter to be set

bool **ListParameters**(std::vector<*Parameter*> **parameters*)
Get all the *Parameter* objects.

Return true

Return false call service fail or timeout

Parameters

- `parameters`: pointer of vector to store all the parameters

9.11 cyber/service/service_base.h

Defined in `cyber/service/service_base.h`

class ServiceBase
Base class for *Service*.

Subclassed by *apollo::cyber::Service< Request, Response >*

Public Functions

ServiceBase(**const** std::string &*service_name*)
Construct a new *Service* Base object.

Parameters

- `service_name`: name of this *Service*

const std::string &**service_name**() **const**
Get the service name.

9.12 cyber/service/service.h

Defined in `cyber/service/service.h`

template<typename **Request**, typename **Response**>
class Service : **public** apollo::cyber::*ServiceBase*
Service handles `Request` from the *Client*, and send a `Response` to it.

Template Parameters

- `Request`: the request type
- `Response`: the response type

Public Functions

Service(**const** std::string &*node_name*, **const** std::string &*service_name*, **const** ServiceCallback &*service_callback*)
Construct a new *Service* object.

Parameters

- `node_name`: used to fill RoleAttribute when join the topology
- `service_name`: the service name we provide

- `service_callback`: reference of `ServiceCallback` object

Service (**const** std::string &*node_name*, **const** std::string &*service_name*, ServiceCallback &&*service_callback*)
Construct a new *Service* object.

Parameters

- `node_name`: used to fill RoleAttribute when join the topology
- `service_name`: the service name we provide
- `service_callback`: rvalue reference of `ServiceCallback` object

Service ()
Forbid default constructing.

bool **Init** ()
Init the *Service*.

void **destroy** ()
Destroy the *Service*.

9.13 cyber/service/client_base.h

Defined in `cyber/service/client_base.h`

class ClientBase
Base class of *Client*.

Subclassed by *apollo::cyber::Client< Request, Response >*

Public Functions

ClientBase (**const** std::string &*service_name*)
Construct a new *Client* Base object.

Parameters

- `service_name`: the service we can request

virtual void **Destroy** () = 0
Destroy the *Client*.

const std::string &**ServiceName** () **const**
Get the service name.

virtual bool **ServiceIsReady** () **const** = 0
Ensure whether there is any *Service* named `service_name_`

9.14 cyber/service/client.h

Defined in `cyber/service/client.h`

template<typename **Request**, typename **Response**>
class Client : **public** apollo::cyber::*ClientBase*

Client get `Response` from a responding `Service` by sending a Request.

Warning One *Client* can only request one *Service*

Template Parameters

- `Request`: the `Service` request type
- `Response`: the `Service` response type

Public Functions

Client (**const** std::string &*node_name*, **const** std::string &*service_name*)

Construct a new *Client* object.

Parameters

- `node_name`: used to fill RoleAttribute
- `service_name`: service name the *Client* can request

Client ()

forbid Constructing a new *Client* object with empty params

bool **Init** ()

Init the *Client*.

Return true if init successfully

Return false if init failed

Client<Request, Response>::SharedResponse **SendRequest** (SharedRequest *request*, **const** std::chrono::seconds &*timeout_s* = std::chrono::seconds(5))

Request the *Service* with a shared ptr Request type.

Return SharedResponse result of this request

Parameters

- `request`: shared ptr of Request type
- `timeout_s`: request timeout, if timeout, response will be empty

Client<Request, Response>::SharedResponse **SendRequest** (**const** Request &*request*, **const** std::chrono::seconds &*timeout_s* = std::chrono::seconds(5))

Request the *Service* with a Request object.

Return SharedResponse result of this request

Parameters

- `request`: Request object
- `timeout_s`: request timeout, if timeout, response will be empty

Client<Request, Response>::SharedFuture **AsyncSendRequest**(SharedRequest *request*)
Send Request shared ptr asynchronously.

Client<Request, Response>::SharedFuture **AsyncSendRequest**(**const** Request &*request*)
Send Request object asynchronously.

Client<Request, Response>::SharedFuture **AsyncSendRequest**(SharedRequest *request*, CallbackType &&*cb*)
Send Request shared ptr asynchronously and invoke `cb` after we get response.

Return SharedFuture a `std::future` shared ptr

Parameters

- `request`: Request shared ptr
- `cb`: callback function after we get response

bool **ServiceIsReady**() **const**
Is the *Service* is ready?

void **Destroy**()
destroy this *Client*

template<typename **RatioT** = std::milli>
bool **WaitForService**(std::chrono::duration<int64_t, *RatioT*> *timeout* = std::chrono::duration<int64_t, *RatioT*>(-1))
wait for the connection with the *Service* established

Return true if the connection established

Return false if timeout

Template Parameters

- `RatioT`: timeout unit, default is std::milli

Parameters

- `timeout`: wait time in unit of `RatioT`

9.15 cyber/service_discovery/specific_manager/manager.h

Defined in `cyber/service_discovery/specific_manager/channel_namager.h`

class Manager
Base class for management of Topology elements.

Manager can Join/Leave the Topology, and Listen the topology change

Subclassed by *apollo::cyber::service_discovery::ChannelManager*, *apollo::cyber::service_discovery::NodeManager*, *apollo::cyber::service_discovery::ServiceManager*

Public Functions

Manager()

Construct a new *Manager* object.

virtual ~Manager()

Destroy the *Manager* object.

bool **StartDiscovery**(RtpsParticipant **participant*)

Startup topology discovery.

Return true if start successfully

Return false if start fail

Parameters

- `participant`: is used to create rtps Publisher and Subscriber

void **StopDiscovery**()

Stop topology discovery.

virtual void **Shutdown**()

Shutdown module.

bool **Join**(**const** RoleAttributes &*attr*, RoleType *role*, bool *need_publish* = true)

Join the topology.

Return true if Join topology successfully

Return false if Join topology failed

Parameters

- `attr`: is the attributes that will be sent to other *Manager*(include ourselves)
- `role`: is one of RoleType enum

bool **Leave**(**const** RoleAttributes &*attr*, RoleType *role*)

Leave the topology.

Return true if Leave topology successfully

Return false if Leave topology failed

Parameters

- `attr`: is the attributes that will be sent to other *Manager*(include ourselves)
- `role`: if one of RoleType enum.

ChangeConnection **AddChangeListener**(**const** ChangeFunc &*func*)

Add topology change listener, when topology changed, func will be called.

Return ChangeConnection Store it to use when you want to stop listening.

Parameters

- `func`: the callback function

void **RemoveChangeListener**(**const** ChangeConnection &*conn*)
Remove our listener for topology change.

Parameters

- `conn`: is the return value of `AddChangeListener`

virtual void **OnTopoModuleLeave**(**const** std::string &*host_name*, int *process_id*) = 0
Called when a process' topology manager instance leave.

Parameters

- `host_name`: is the process's host's name
- `process_id`: is the process' id

9.16 cyber/service_discovery/specific_manager/channel_manager.h

Defined in `cyber/service_discovery/specific_manager/channel_manager.h`

class ChannelManager : **public** apollo::cyber::service_discovery::*Manager*
Topology *Manager* of *Service* related.

Public Functions

ChannelManager()
Construct a new Channel *Manager* object.

virtual ~ChannelManager()
Destroy the Channel *Manager* object.

void **GetChannelNames**(std::vector<std::string> **channels*)
Get all channel names in the topology.

Parameters

- `channels`: result vector

void **GetProtoDesc**(**const** std::string &*channel_name*, std::string **proto_desc*)
Get the Protocol Desc of `channel_name`

Parameters

- `channel_name`: channel name we want to inquire
- `proto_desc`: result string, empty if inquire failed

void **GetMsgType**(**const** std::string &*channel_name*, std::string **msg_type*)
Get the Msg Type of `channel_name`

Parameters

- `channel_name`: channel name we want to inquire
- `msg_type`: result string, empty if inquire failed

bool **`HasWriter`** (**`const`** std::string &*channel_name*)
Inquire if there is at least one *Writer* that publishes `channel_name`

Return true if there is at least one *Writer*

Return false if there are no Writers

Parameters

- `channel_name`: channel name we want to inquire

void **`GetWriters`** (RoleAttrVec **writers*)
Get All Writers object.

Parameters

- `writers`: result RoleAttr vector

void **`GetWritersOfNode`** (**`const`** std::string &*node_name*, RoleAttrVec **writers*)
Get the Writers Of *Node* object.

Parameters

- `node_name`: node's name we want to inquire
- `writers`: result RoleAttribute vector

void **`GetWritersOfChannel`** (**`const`** std::string &*channel_name*, RoleAttrVec **writers*)
Get the Writers Of Channel object.

Parameters

- `channel_name`: channel's name we want to inquire
- `writers`: result RoleAttribute vector

bool **`HasReader`** (**`const`** std::string &*channel_name*)
Inquire if there is at least one *Reader* that publishes `channel_name`

Return true if there is at least one *Reader*

Return false if there are no *Reader*

Parameters

- `channel_name`: channel name we want to inquire

void **`GetReaders`** (RoleAttrVec **readers*)
Get All Readers object.

Parameters

- `readers`: result RoleAttr vector

void **`GetReadersOfNode`** (**`const`** std::string &*node_name*, RoleAttrVec **readers*)
Get the Readers Of *Node* object.

Parameters

- `node_name`: node's name we want to inquire
- `readers`: result RoleAttribute vector

void **GetReadersOfChannel**(**const** std::string &*channel_name*, RoleAttrVec **readers*)

Get the Readers Of Channel object.

Parameters

- `channel_name`: channel's name we want to inquire
- `readers`: result RoleAttribute vector

void **GetUpstreamOfNode**(**const** std::string &*node_name*, RoleAttrVec **upstream_nodes*)

Get the Upstream Of *Node* object.

If *Node* A has writer that publishes channel-1, and *Node* B has reader that subscribes channel-1 then A is B's Upstream node, and B is A's Downstream node

Parameters

- `node_name`: node's name we want to inquire
- `upstream_nodes`: result RoleAttribute vector

void **GetDownstreamOfNode**(**const** std::string &*node_name*, RoleAttrVec **downstream_nodes*)

Get the Downstream Of *Node* object.

If *Node* A has writer that publishes channel-1, and *Node* B has reader that subscribes channel-1 then A is B's Upstream node, and B is A's Downstream node

Parameters

- `node_name`: node's name we want to inquire
- `downstream_nodes`: result RoleAttribute vector

FlowDirection **GetFlowDirection**(**const** std::string &*lhs_node_name*, **const** std::string &*rhs_node_name*)

Get the Flow Direction from `lhs_node_node` to `rhs_node_name` You can see FlowDirection's description for more information.

Return FlowDirection result direction

bool **IsMessageTypeMatching**(**const** std::string &*lhs*, **const** std::string &*rhs*)

Is `lhs` and `rhs` have same MessageType.

Return true if type matches

Return false if type does not matches

Parameters

- `lhs`: the left message type to compare
- `rhs`: the right message type to compare

9.17 cyber/service_discovery/specific_manager/node_manager.h

Defined in `cyber/service_discovery/specific_manager/node_manager.h`

class NodeManager : **public** apollo::cyber::service_discovery::*Manager*
Topology *Manager* of *Node* related.

Public Functions

NodeManager()
Construct a new *Node Manager* object.

virtual ~NodeManager()
Destroy the *Node Manager* object.

bool **HasNode**(**const** std::string &*node_name*)
Checkout whether we have `node_name` in topology.

Return true if this node found

Return false if this node not exits

Parameters

- `node_name`: *Node*'s name we want to inquire

void **GetNodes**(RoleAttrVec **nodes*)
Get the Nodes object.

Parameters

- `nodes`: result RoleAttr vector

9.18 cyber/service_discovery/specific_manager/service_manager.h

Defined in `cyber/service_discovery/specific_manager/service_manager.h`

class ServiceManager : **public** apollo::cyber::service_discovery::*Manager*
Topology *Manager* of *Service* related.

Public Functions

ServiceManager()
Construct a new *Service Manager* object.

virtual ~ServiceManager()
Destroy the *Service Manager* object.

bool **HasService**(**const** std::string &*service_name*)
Inquire whether `service_name` exists in topology.

Return true if service exists

Return false if service not exists

Parameters

- `service_name`: the name we inquire

void **GetServers** (RoleAttrVec **servers*)
Get the All Server in the topology.

Parameters

- `servers`: result RoleAttr vector

void **GetClients** (**const** std::string &*service_name*, RoleAttrVec **clients*)
Get the Clients object that subscribes `service_name`

Parameters

- `service_name`: Name of service you want to get
- `clients`: result vector

9.19 cyber/service_discovery/topology_manager.h

Defined in `cyber/service_discovery/topology_manager.h`

class TopologyManager
elements in Cyber *Node*, Channel, *Service*, *Writer*, *Reader*, *Client* and Server's relationship is presented by Topology.

You can Imagine that a directed graph *Node* is the container of Server/Client/Writer/Reader, and they are the vertice of the graph and Channel is the Edge from *Writer* flow to the *Reader*, *Service* is the Edge from Server to *Client*. Thus we call *Writer* and Server `Upstream`, *Reader* and *Client* `Downstream` To generate this graph, we use *TopologyManager*, it has three sub managers *NodeManager*: You can find Nodes in this topology *ChannelManager*: You can find Channels in this topology, and their Writers and Readers *ServiceManager*: You can find Services in this topology, and their Servers and Clients *TopologyManager* use fast-rtps' Participant to communicate. It can broadcast Join or Leave messages of those elements. Also, you can register you own `ChangeFunc` to monitor topology change

Public Functions

void **Shutdown** ()
Shutdown the *TopologyManager*.

ChangeConnection **AddChangeListener** (**const** ChangeFunc &*func*)
To observe the topology change, you can register a `ChangeFunc`

Return ChangeConnection is the connection that connected to `change_signal_`. Used to Remove your observe function

Parameters

- `func`: is the observe function

void **RemoveChangeListener** (**const** ChangeConnection &*conn*)
Remove the observe function connect to `change_signal_` by `conn`

NodeManagerPtr &**node_manager**()
Get shared_ptr for *NodeManager*.

ChannelManagerPtr &**channel_manager**()
Get shared_ptr for *ChannelManager*.

ServiceManagerPtr &**service_manager**()
Get shared_ptr for *ServiceManager*.

9.20 cyber/component/component.h

Defined in `cyber/component/component.h`

template<typename **M0** = NullType, typename **M1** = NullType, typename **M2** = NullType, typename **M3** = NullType>
class Component : **public** apollo::cyber::ComponentBase

The *Component* can process up to four channels of messages. The message type is specified when the component is created. The *Component* is inherited from ComponentBase. Your component can inherit from *Component*, and implement Init() & Proc(…), They are picked up by the CyberRT. There are 4 specialization implementations.

Warning The Init & Proc functions need to be overloaded, but don't want to be called. They are called by the CyberRT Frame.

Template Parameters

- `M0`: the first message.
- `M1`: the second message.
- `M2`: the third message.
- `M3`: the fourth message.

Public Functions

bool **Initialize**(**const** ComponentConfig &*config*)
init the component by protobuf object.

Return returns true if successful, otherwise returns false

Parameters

- `config`: which is defined in 'cyber/proto/component_conf.proto'

9.21 cyber/component/timer_component.h

Defined in `cyber/component/timer_component.h`

class TimerComponent : **public** apollo::cyber::ComponentBase

TimerComponent is a timer component. Your component can inherit from *Component*, and implement Init() & Proc(), They are called by the CyberRT frame.

Public Functions

bool **Initialize** (**const** TimerComponentConfig &*config*)
init the component by protobuf object.

Return returns true if successful, otherwise returns false

Parameters

- `config`: which is define in 'cyber/proto/component_conf.proto'

9.22 cyber/logger/async_logger.h

Defined in `cyber/logger/async_logger.h`

class AsyncLogger : **public** Logger
Wrapper for a glog Logger which asynchronously writes log messages. This class starts a new thread responsible for forwarding the messages to the logger, and performs double buffering. Writers append to the current buffer and then wake up the logger thread. The logger swaps in a new buffer and writes any accumulated messages to the wrapped Logger.

This double-buffering design dramatically improves performance, especially for logging messages which require flushing the underlying file (i.e WARNING and above for default). The flush can take a couple of milliseconds, and in some cases can even block for hundreds of milliseconds or more. With the double-buffered approach, threads can proceed with useful work while the IO thread blocks.

The semantics provided by this wrapper are slightly weaker than the default glog semantics. By default, glog will immediately (synchronously) flush WARNING and above to the underlying file, whereas here we are deferring that flush to a separate thread. This means that a crash just after a 'LOG_WARN' would may be missing the message in the logs, but the perf benefit is probably worth it. We do take care that a glog FATAL message flushes all buffered log messages before exiting.

Warning The logger limits the total amount of buffer space, so if the underlying log blocks for too long, eventually the threads generating the log messages will block as well. This prevents runaway memory usage.

Public Functions

void **Start** ()
start the async logger

void **Stop** ()
Stop the thread.

Flush() and *Write()* must not be called after this. NOTE: this is currently only used in tests: in real life, we enable async logging once when the program starts and then never disable it. REQUIRES: *Start()* must have been called.

void **Write** (bool *force_flush*, time_t *timestamp*, **const** char **message*, int *message_len*)
Write a message to the log.

Start() must have been called.

Parameters

- `force_flush`: is set by the GLog library based on the configured 'logbuflevel' flag. Any messages logged at the configured level or higher result in 'force_flush' being set to true, indicating that the message should be immediately written to the log rather than buffered in memory.
- `timestamp`: is the time of write a message
- `message`: is the info to be written
- `message_len`: is the length of message

void **Flush**()
Flush any buffered messages.

uint32_t **LogSize**()
Get the current LOG file size.

The return value is an approximate value since some logged data may not have been flushed to disk yet.

Return the log file size

const std::thread ***LogThread**() **const**
get the log thead

Return the pointer of log thread

9.23 cyber/timer/timer.h

Defined in `cyber/timer/timer.h`

struct TimerOption
The options of timer.

Public Functions

TimerOption(uint32_t *period*, std::function<void)
> *callback*bool *oneshot*Construct a new *Timer* Option object.

Parameters

- `period`: The period of the timer, unit is ms
- `callback`: The task that the timer needs to perform
- `oneshot`: Oneshot or period

TimerOption()
Default onstructor for initializer list.

Public Members

uint32_t **period** = 0
The period of the timer, unit is ms max: 512 * 64 min: 1.

std::function<void ()> **callback**
The task that the timer needs to perform.

bool **oneshot**
True: perform the callback only after the first timing cycle False: perform the callback every timed period.

class Timer
Used to perform oneshot or periodic timing tasks.

Public Functions

Timer(*TimerOption opt*)
Construct a new *Timer* object.

Parameters

- `opt`: *Timer* option

Timer(uint32_t *period*, std::function<void)
> *callback*bool *oneshot*Construct a new *Timer* object.

Parameters

- `period`: The period of the timer, unit is ms
- `callback`: The tasks that the timer needs to perform
- `oneshot`: True: perform the callback only after the first timing cycle False: perform the callback every timed period

void **SetTimerOption**(*TimerOption opt*)
Set the *Timer* Option object.

Parameters

- `opt`: The timer option will be set

void **Start**()
Start the timer.

void **Stop**()
Stop the timer.

9.24 cyber/time/time.h

Defined in `cyber/time/time.h`

class Time
Cyber has builtin time type *Time*.

Public Functions

double **ToSecond**() **const**
convert time to second.

Return return a double value unit is second.

uint64_t **ToNanosecond**() **const**
convert time to nanosecond.

Return return a unit64_t value unit is nanosecond.

std::string **ToString**() **const**
convert time to a string.

Return return a string.

bool **IsZero**() **const**
determine if time is 0

Return return true if time is 0

Public Static Functions

static *Time* **Now**()
get the current time.

Return return the current time.

static void **SleepUntil**(**const** *Time* &*time*)
Sleep Until time.

Parameters

- `time`: the *Time* object.

9.25 cyber/record/header_builder.h

Defined in `cyber/record/header_builder.h`

class HeaderBuilder
The builder of record header.

Public Static Functions

static Header **GetHeaderWithSegmentParams**(**const** uint64_t *segment_interval*, **const** uint64_t *segment_raw_size*)
Build a record header with customized max interval time (ns) and max raw size (byte) for segment.

Return A customized record header.

Parameters

- `segment_interval`:
- `segment_raw_size`:

static Header **GetHeaderWithChunkParams**(**const** uint64_t *chunk_interval*, **const** uint64_t *chunk_raw_size*)
Build a record header with customized max interval time (ns) and max raw size (byte) for chunk.

Return A customized record header.

Parameters

- `chunk_interval`:
- `chunk_raw_size`:

static Header **GetHeader**()

Build a default record header.

Return A default record header.

9.26 cyber/record/record_base.h

Defined in `cyber/record/record_base.h`

class RecordBase

Base class for record reader and writer.

Subclassed by *apollo::cyber::record::RecordReader*, *apollo::cyber::record::RecordWriter*

Public Functions

virtual ~RecordBase()

Destructor.

virtual uint64_t **GetMessageNumber**(**const** std::string &*channel_name*) **const** = 0

Get message number by channel name.

Return Message number.

Parameters

- `channel_name`:

virtual const std::string &**GetMessageType**(**const** std::string &*channel_name*) **const** = 0

Get message type by channel name.

Return Message type.

Parameters

- `channel_name`:

virtual const std::string &**GetProtoDesc**(**const** std::string &*channel_name*) **const** = 0

Get proto descriptor string by channel name.

Return Proto descriptor string by channel name.

Parameters

- `channel_name`:

virtual std::set<std::string> **GetChannelList**() **const** = 0

Get channel list.

Return List container with all channel name string.

const proto::Header &**GetHeader**() **const**
Get record header.

Return Record header.

const std::string **GetFile**() **const**
Get record file path.

Return Record file path.

9.27 cyber/record/record_message.h

Defined in `cyber/record/record_message.h`

struct RecordMessage
Basic data struct of record message.

Public Functions

RecordMessage()
The constructor.

RecordMessage(**const** std::string &*name*, **const** std::string &*message*, uint64_t *msg_time*)
The constructor.

Parameters

- `name`:
- `message`:
- `msg_time`:

Public Members

std::string **channel_name**
The channel name of the message.

std::string **content**
The content of the message.

uint64_t **time**
The time (nanosecond) of the message.

9.28 cyber/record/record_reader.h

Defined in `cyber/record/record_reader.h`

class RecordReader : **public** apollo::cyber::record::*RecordBase*
The record reader.

Public Functions

RecordReader(**const** std::string &*file*)
The constructor with record file path as parameter.

Parameters

- `file`:

virtual ~RecordReader()
The destructor.

bool **IsValid**() **const**
Is this record reader is valid.

Return True for valid, false for not.

bool **ReadMessage**(*RecordMessage* **message*, uint64_t *begin_time* = 0, uint64_t *end_time* = UINT64_MAX)
Read one message from reader.

Return True for success, flase for not.

Parameters

- `message`:
- `begin_time`:
- `end_time`:

void **Reset**()
Reset the message index of record reader.

uint64_t **GetMessageNumber**(**const** std::string &*channel_name*) **const**
Get message number by channel name.

Return Message number.

Parameters

- `channel_name`:

const std::string &**GetMessageType**(**const** std::string &*channel_name*) **const**
Get message type by channel name.

Return Message type.

Parameters

- `channel_name`:

const std::string &**GetProtoDesc**(**const** std::string &*channel_name*) **const**
Get proto descriptor string by channel name.

Return Proto descriptor string by channel name.

Parameters

- `channel_name`:

std::set<std::string> **GetChannelList**() **const**
Get channel list.

Return List container with all channel name string.

9.29 cyber/record/record_writer.h

Defined in `cyber/record/record_writer.h`

class RecordWriter : **public** apollo::cyber::record::*RecordBase*
The record writer.

Public Functions

RecordWriter()
The default constructor.

RecordWriter(**const** proto::Header &*header*)
The constructor with record header as parameter.

Parameters

- `header`:

virtual ~RecordWriter()
Virtual Destructor.

bool **Open**(**const** std::string &*file*)
Open a record to write.

Return True for success, false for fail.

Parameters

- `file`:

void **Close**()
Clean the record.

bool **WriteChannel**(**const** std::string &*channel_name*, **const** std::string &*message_type*, **const** std::string &*proto_desc*)
Write a channel to record.

Return True for success, false for fail.

Parameters

- `channel_name`:
- `message_type`:
- `proto_desc`:

template<typename **MessageT**>
bool **WriteMessage**(**const** std::string &*channel_name*, **const** *MessageT* &*message*, **const** uint64_t *time_nanosec*, **const** std::string &*proto_desc* = "")

Write a message to record.

Return True for success, false for fail.

Template Parameters

- `MessageT`:

Parameters

- `channel_name`:
- `message`:
- `time_nanosec`:
- `proto_desc`:

bool **SetSizeOfFileSegmentation**(uint64_t *size_kilobytes*)

Set max size (KB) to segment record file.

Return True for success, false for fail.

Parameters

- `size_kilobytes`:

bool **SetIntervalOfFileSegmentation**(uint64_t *time_sec*)

Set max interval (Second) to segment record file.

Return True for success, false for fail.

Parameters

- `time_sec`:

uint64_t **GetMessageNumber**(**const** std::string &*channel_name*) **const**

Get message number by channel name.

Return Message number.

Parameters

- `channel_name`:

const std::string &**GetMessageType**(**const** std::string &*channel_name*) **const**

Get message type by channel name.

Return Message type.

Parameters

- `channel_name`:

const std::string &**GetProtoDesc**(**const** std::string &*channel_name*) **const**

Get proto descriptor string by channel name.

Return Proto descriptor string by channel name.

Parameters

- `channel_name`:

std::set<std::string> **`GetChannelList`**() **`const`**
Get channel list.

Return List container with all channel name string.

bool **`IsNewChannel`**(**`const`** std::string &*channel_name*) **`const`**
Is a new channel recording or not.

Return True for yes, false for no.

9.30 cyber/record/record_viewer.h

Defined in `cyber/record/record_viewer.h`

`class RecordViewer`
The record viewer.

Public Functions

`RecordViewer`(**`const`** RecordReaderPtr &*reader*, uint64_t *begin_time* = 0, uint64_t *end_time* = UINT64_MAX, **`const`** std::set<std::string> &*channels* = std::set<std::string>())
The constructor with single reader.

Parameters

- `reader`:
- `begin_time`:
- `end_time`:
- `channels`:

`RecordViewer`(**`const`** std::vector<RecordReaderPtr> &*readers*, uint64_t *begin_time* = 0, uint64_t *end_time* = UINT64_MAX, **`const`** std::set<std::string> &*channels* = std::set<std::string>())
The constructor with multiple readers.

Parameters

- `readers`:
- `begin_time`:
- `end_time`:
- `channels`:

bool **`IsValid`**() **`const`**
Is this record reader is valid.

Return True for valid, false for not.

uint64_t **begin_time**() **const**
Get begin time.

Return Begin time (nanoseconds).

uint64_t **end_time**() **const**
Get end time.

Return end time (nanoseconds).

std::set<std::string> **GetChannelList**() **const**
Get channel list.

Return List container with all channel name string.

Iterator **begin**()
Get the begin iterator.

Return The begin iterator.

Iterator **end**()
Get the end iterator.

Return The end iterator.

class Iterator : **public** std::iterator<std::input_iterator_tag, *RecordMessage*, int, *RecordMessage* *, *RecordMessage*&>
The iterator.

Public Functions

Iterator(*RecordViewer* **viewer*, bool *end* = false)
The constructor of iterator with viewer.

Parameters
- `viewer`:
- `end`:

Iterator()
The default constructor of iterator.

virtual ~Iterator()
The default destructor of iterator.

bool **operator==**(*Iterator* **const** &*other*) **const**
Overloading operator ==.

Return The result.
Parameters
- `other`:

bool **operator!=**(**const** *Iterator* &*rhs*) **const**
Overloading operator !=.

Return The result.

Parameters

- `other`:

Iterator &**`operator++`**`()`

Overloading operator ++.

Return The result.

pointer **`operator->`**`()`

Overloading operator ->.

Return The pointer.

reference **`operator*`**`()`

Overloading operator *.

Return The reference.

CHAPTER

TEN

PYTHON API

Cyber RT provides the python interfaces for developers.

10.1 python/cyber_py/cyber.py

Defined in python/cyber_py/cyber.py

`cyber_py.cyber.`**`init`**(*module_name module_name = "cyber_py"*)

init cyber environment.

Return Success is True, otherwise False.

```
init cyber environment.
```

Parameters

- `module_name`: Used as the log file name.

`cyber_py.cyber.`**`ok`**()

```
is cyber envi ok.
```

`cyber_py.cyber.`**`shutdown`**()

```
shutdown cyber envi.
```

`cyber_py.cyber.`**`is_shutdown`**()

```
is cyber shutdown.
```

`cyber_py.cyber.`**`waitforshutdown`**()

```
wait until the cyber is shutdown.
```

`class Node`

```
class for cyber Node wrapper.
```

Public Functions

`register_message`(*self self, file_desc file_desc*)

register proto message by proto descriptor file.

Parameters

- `file_desc`: object about datatype.DESCRIPTOR.file .

```
register proto message desc file.
```

create_writer (*self self*, *name name*, *data_type data_type*, *qos_depth qos_depth = 1*)
create a channel writer for send message to another channel.

Return return the writer object.

```
create a channel writer for send message to another channel.
```

Parameters

- `name`: is the channel name.
- `data_type`: is message class for serialization
- `qos_depth`: is a queue size, which defines the size of the cache.

create_reader (*self self*, *name name*, *data_type data_type*, *callback callback*, *args args = None*)
create a channel reader for receive message from another channel.

Return return the writer object.

```
create a channel reader for receive message from another channel.
```

Parameters

- `name`: the channel name to read.
- `data_type`: message class for serialization
- `callback`: function to call (fn(data)) when data is received. If args is set, the function must accept the args as a second argument, i.e. fn(data, args)
- `args`: additional arguments to pass to the callback

create_rawdata_reader (*self self*, *name name*, *callback callback*, *args args = None*)

```
Create RawData reader:listen RawMessage
```

create_client (*self self*, *name name*, *request_data_type request_data_type*, *response_data_type response_data_type*)
create client for the c/s.

Return the client object.

Parameters

- `name`: the service name.
- `request_data_type`: the request message type.
- `response_data_type`: the response message type.

create_service (*self self*, *name name*, *req_data_type req_data_type*, *res_data_type res_data_type*, *callback callback*, *args args = None*)
create client for the c/s.

Return return the service object.

Parameters

- `name`: the service name.
- `req_data_type`: the request message type.
- `res_data_type`: the response message type.

- `callback`: function to call (fn(data)) when data is received. If args is set, the function must accept the args as a second argument, i.e. fn(data, args)
- `args`: additional arguments to pass to the callback.

spin (*self self*)

```
spin every 0.002s.
```

class Writer

```
Class for cyber writer wrapper.
```

Public Functions

write (*self self, data data*)
write message.

Return Success is 0, otherwise False.

```
writer message string
```

Parameters

- `data`: is a message type.

class Reader

```
Class for cyber reader wrapper.
```

class Client

```
Class for cyber service client wrapper.
```

Public Functions

send_request (*self self, data data*)
send request message to service.

Return None or response from service.

```
send request to service
```

Parameters

- `data`: is a message type.

class ChannelUtils

Public Static Functions

get_debugstring_rawmsgdata (*msg_type msg_type, rawmsgdata rawmsgdata*)
Parse rawmsg from rawmsg data by message type.

Return a human readable form of this message. For debugging and other purposes.

Parameters

- `msg_type`: message type.
- `rawmsgdata`: rawmsg data.

get_msgtype (*channel_name channel_name*, *sleep_s sleep_s = 2*)
Parse rawmsg from channel name.

Return return the messsage type of this channel.

Parameters

- `channel_name`: channel name.
- `sleep_s`: wait time for topo discovery.

get_channels (*sleep_s sleep_s = 2*)
Get all active channel names.

Return all active channel names.

Parameters

- `sleep_s`: wait time for topo discovery.

get_channels_info (*sleep_s sleep_s = 2*)
Get the active channel info.

Return all active channels info. {'channel1':[], 'channel2':[]} .

Parameters

- `sleep_s`: wait time for topo discovery.

class NodeUtils

Public Static Functions

get_nodes (*sleep_s sleep_s = 2*)
Get all active node names.

Return all active node names.

Parameters

- `sleep_s`: wait time for topo discovery.

get_node_attr (*node_name node_name*, *sleep_s sleep_s = 2*)
Get node attribute by the node name.

Return the node's attribute.

Parameters

- `node_name`: node name.
- `sleep_s`: wait time for topo discovery.

get_readersofnode (*node_name node_name*, *sleep_s sleep_s = 2*)
Get node's reader channel names.

Return node's reader channel names.

Parameters

- `node_name`: the node name.
- `sleep_s`: wait time for topo discovery.

get_writersofnode (*node_name node_name*, *sleep_s sleep_s = 2*)
Get node's writer channel names.

Return node's writer channel names.

Parameters

- `node_name`: the node name.
- `sleep_s`: wait time for topo discovery.

10.2 python/cyber_py/record.py

Defined in python/cyber_py/record.py

class RecordReader

```
Class for python RecordReader wrapper.
```

Public Functions

__init__ (*self self*, *file_name file_name*)
the constructor function.

Parameters

- `file_name`: the record file name.

read_messages (*self self*, *start_time start_time = 0*, *end_time end_time = 18446744073709551615*)
Read message from bag file.

Return return (channnel, data, data_type, timestamp)

Parameters

- `start_time`: the start time to read.
- `end_time`: the end time to read.

get_messagenumber (*self self*, *channel_name channel_name*)
Return message count of the channel in current record file.

Return return the message count.

Parameters

- `channel_name`: the channel name.

get_messagetype (*self self*, *channel_name channel_name*)
Get the corresponding message type of channel.

Return return the name of ther string type.

Parameters

- `channel_name`: channel name.

get_protodesc(*self self*, *channel_name channel_name*)

```
Return message protodesc.
```

get_headerstring(*self self*)

```
Return message header string.
```

reset(*self self*)

```
Return reset.
```

get_channellist(*self self*)

```
Return current channel names list.
```

class RecordWriter

```
Class for python RecordWriter wrapper.
```

Public Functions

__init__(*self self*, *file_segmentation_size_kb file_segmentation_size_kb = 0*, *file_segmentation_interval_sec file_segmentation_interval_sec = 0*)

the constructor function.

Parameters

- `file_segmentation_size_kb`: size to segment the file, 0 is no segmentation.
- `file_segmentation_interval_sec`: size to segment the file, 0 is no segmentation.

open(*self self*, *path path*)

Open record file for write.

Return Success is Ture, other False.

Parameters

- `path`: the file path.

close(*self self*)

Close record file.

```
Close record file.
```

write_channel(*self self*, *channel_name channel_name*, *type_name type_name*, *proto_desc proto_desc*)

Writer channel by channelname, typename, protodesc.

Return Success is Ture, other False.

```
Writer channel by channelname,typename,protodesc
```

Parameters

- `channel_name`: the channel name to write
- `type_name`: a string of message type name.
- `proto_desc`: the message descriptor.

write_message(*self self*, *channel_name channel_name*, *data data*, *time time*, *raw raw = True*)

Writer msg: channelname, data, writer time.

Return Success is Ture, other False.

```
Writer msg:channelname,rawmsg,writer time
```

Parameters

- `channel_name`: channel name to write.
- `data`: when raw is True, data processed as a rawdata, other it needs to SerializeToString
- `time`: message time.
- `raw`: the flag implies data whether or not a rawdata.

set_size_fileseg(*self self, size_kilobytes size_kilobytes*)

```
Return filesegment size.
```

set_intervaltime_fileseg(*self self, time_sec time_sec*)

```
Return file interval time.
```

get_messagenumber(*self self, channel_name channel_name*)

```
Return message count.
```

get_messagetype(*self self, channel_name channel_name*)

```
Return message type.
```

get_protodesc(*self self, channel_name channel_name*)

```
Return message protodesc.
```

10.3 python/cyber_py/cyber_time.py

Defined in python/cyber_py/cyber_time.py

class Duration

```
class for cyber Duration wrapper.
```

Public Functions

sleep(*self self*)

```
sleep for the amount of time specified by the duration.
```

to_sec(*self self*)

```
convert to second.
```

to_nsec(*self self*)

```
convert to nanosecond.
```

class Time

```
class for cyber time wrapper.
```

Public Functions

__init__(*self self, other other*)

Constructor, creates a *Time*.

Parameters

- `other`: float means seconds unit. int or long means nanoseconds.

to_sec(*self self*)

```
convert to second.
```

to_nsec(*self self*)

```
convert to nanosecond.
```

sleep_until(*self self*, *time time*)

```
sleep until time
```

Public Static Functions

now()

```
return current time.
```

class Rate

```
class for cyber Rate wrapper. Help run loops at a desired frequency.
```

Public Functions

__init__(*self self*, *other other*)

Constructor, creates a *Rate*.

Parameters

- `other`: float means frequency the desired rate to run at in Hz. int or long means the expected_cycle_time.

sleep(*self self*)

```
sleep for any leftover time in a cycle.
```

reset(*self self*)

```
Sets the start time for the rate to now.
```

get_cycle_time(*self self*)

```
Get the actual time of a cycle from start to sleep.
```

get_expected_cycle_time(*self self*)

```
Get the expected cycle time.
```

10.4 python/cyber_py/cyber_timer.py

Defined in python/cyber_py/cyber_timer.py

class Timer

```
class for cyber timer wrapper.
```

Public Functions

__init__(*self self*, *period period = None*, *callback callback = None*, *oneshot oneshot = None*)

Used to perform oneshot or periodic timing tasks.

Parameters

- `period`: The period of the timer, unit is ms.
- `callback`: The tasks that the timer needs to perform.

- `oneshot`: 1:perform the callback only after the first timing cycle 0:perform the callback every timed period

set_option (*self self*, *period period*, *callback callback*, *oneshot oneshot = 0*)
set the option of timer.

Parameters

- `period`: The period of the timer, unit is ms.
- `callback`: The tasks that the timer needs to perform.
- `oneshot`: 1:perform the callback only after the first timing cycle 0:perform the callback every timed period

start (*self self*)
start the timer

stop (*self self*)
stop the timer

10.5 python/cyber_py/parameter.py

Defined in python/cyber_py/parameter.py

class Parameter

```
Class for Parameter wrapper.
```

Public Functions

type_name (*self self*)

```
return Parameter typename
```

descriptor (*self self*)

```
return Parameter descriptor
```

name (*self self*)

```
return Parameter name
```

debug_string (*self self*)

```
return Parameter debug string
```

as_string (*self self*)

```
return string value
```

as_double (*self self*)

```
return double value
```

as_int64 (*self self*)

```
return native value
```

class ParameterServer

```
Class for ParameterServer wrapper.
```

Public Functions

__init__ (*self self*, *node node*)
constructor the *ParameterServer* by the node object.

Parameters

- `node`: the node to support the parameter server.

set_parameter(*self self*, *param param*)

```
set parameter.
```

get_parameter(*self self*, *param_name param_name*)

```
get Parameter by param name param_name.
```

get_paramslist(*self self*)

```
get all params of this parameterserver.
```

class ParameterClient

```
Class for ParameterClient wrapper.
```

Public Functions

__init__(*self self*, *node node*, *server_node_name server_node_name*)

constructor the *ParameterClient* by a node and the parameter server node name.

Parameters

- `node`: a node to create client.
- `server_node_name`: the parameter server's node name.

set_parameter(*self self*, *param param*)

```
set parameter.
```

get_parameter(*self self*, *param_name param_name*)

```
get Parameter by param name param_name.
```

get_paramslist(*self self*)

```
get all params of the server_node_name parameterserver.
```

INDEX

A

apollo::cyber::Client (*C++ class*), 83
apollo::cyber::Client::AsyncSendRequest (*C++ function*), 84
apollo::cyber::Client::Client (*C++ function*), 83
apollo::cyber::Client::Destroy (*C++ function*), 84
apollo::cyber::Client::Init (*C++ function*), 83
apollo::cyber::Client::SendRequest (*C++ function*), 83
apollo::cyber::Client::ServiceIsReady (*C++ function*), 84
apollo::cyber::Client::WaitForService (*C++ function*), 84
apollo::cyber::ClientBase (*C++ class*), 82
apollo::cyber::ClientBase::ClientBase (*C++ function*), 82
apollo::cyber::ClientBase::Destroy (*C++ function*), 82
apollo::cyber::ClientBase::ServiceIsReady (*C++ function*), 82
apollo::cyber::ClientBase::ServiceName (*C++ function*), 82
apollo::cyber::Component (*C++ class*), 91
apollo::cyber::Component::Initialize (*C++ function*), 91
apollo::cyber::logger::AsyncLogger (*C++ class*), 92
apollo::cyber::logger::AsyncLogger::Flush (*C++ function*), 93
apollo::cyber::logger::AsyncLogger::LogSize (*C++ function*), 93
apollo::cyber::logger::AsyncLogger::LogThread (*C++ function*), 93
apollo::cyber::logger::AsyncLogger::Start (*C++ function*), 92
apollo::cyber::logger::AsyncLogger::Stop (*C++ function*), 92
apollo::cyber::logger::AsyncLogger::Write (*C++ function*), 92
apollo::cyber::Node (*C++ class*), 65
apollo::cyber::Node::ClearData (*C++ function*), 67
apollo::cyber::Node::CreateClient (*C++ function*), 67
apollo::cyber::Node::CreateReader (*C++ function*), 66
apollo::cyber::Node::CreateService (*C++ function*), 66
apollo::cyber::Node::CreateWriter (*C++ function*), 65
apollo::cyber::Node::GetReader (*C++ function*), 67
apollo::cyber::Node::Name (*C++ function*), 65
apollo::cyber::Node::Observe (*C++ function*), 67
apollo::cyber::NodeChannelImpl (*C++ class*), 74
apollo::cyber::NodeChannelImpl::~NodeChannelImpl (*C++ function*), 74
apollo::cyber::NodeChannelImpl::NodeChannelImpl (*C++ function*), 74
apollo::cyber::NodeChannelImpl::NodeName (*C++ function*), 74
apollo::cyber::NodeServiceImpl (*C++ class*), 74
apollo::cyber::NodeServiceImpl::~NodeServiceImpl (*C++ function*), 75
apollo::cyber::NodeServiceImpl::NodeServiceImpl (*C++ function*), 75
apollo::cyber::Parameter (*C++ class*), 75
apollo::cyber::Parameter::AsBool (*C++ function*), 77
apollo::cyber::Parameter::AsDouble (*C++ function*), 78
apollo::cyber::Parameter::AsInt64 (*C++ function*), 77
apollo::cyber::Parameter::AsString (*C++ function*), 78
apollo::cyber::Parameter::DebugString (*C++ function*), 78
apollo::cyber::Parameter::Descriptor

(C++ function), 77
apollo::cyber::Parameter::FromProtoParam (C++ function), 77
apollo::cyber::Parameter::Name (C++ function), 77
apollo::cyber::Parameter::Parameter (C++ function), 75–77
apollo::cyber::Parameter::ToProtoParam (C++ function), 77
apollo::cyber::Parameter::Type (C++ function), 77
apollo::cyber::Parameter::TypeName (C++ function), 77
apollo::cyber::Parameter::value (C++ function), 78, 79
apollo::cyber::ParameterClient (C++ class), 80
apollo::cyber::ParameterClient::GetParameter (C++ function), 80
apollo::cyber::ParameterClient::ListParameters (C++ function), 80
apollo::cyber::ParameterClient::ParameterClient (C++ function), 80
apollo::cyber::ParameterClient::SetParameter (C++ function), 80
apollo::cyber::ParameterServer (C++ class), 79
apollo::cyber::ParameterServer::GetParameter (C++ function), 79
apollo::cyber::ParameterServer::ListParameters (C++ function), 80
apollo::cyber::ParameterServer::ParameterServer (C++ function), 79
apollo::cyber::ParameterServer::SetParameter (C++ function), 79
apollo::cyber::Reader (C++ class), 69
apollo::cyber::Reader::Begin (C++ function), 71
apollo::cyber::Reader::ClearData (C++ function), 70
apollo::cyber::Reader::Empty (C++ function), 70
apollo::cyber::Reader::End (C++ function), 71
apollo::cyber::Reader::Enqueue (C++ function), 70
apollo::cyber::Reader::GetDelaySec (C++ function), 70
apollo::cyber::Reader::GetHistoryDepth (C++ function), 71
apollo::cyber::Reader::GetLatestObserved (C++ function), 71
apollo::cyber::Reader::GetOldestObserved (C++ function), 71
apollo::cyber::Reader::GetWriters (C++ function), 71
apollo::cyber::Reader::HasReceived (C++ function), 70
apollo::cyber::Reader::HasWriter (C++ function), 71
apollo::cyber::Reader::Init (C++ function), 70
apollo::cyber::Reader::Observe (C++ function), 70
apollo::cyber::Reader::PendingQueueSize (C++ function), 70
apollo::cyber::Reader::Reader (C++ function), 69
apollo::cyber::Reader::SetHistoryDepth (C++ function), 70
apollo::cyber::Reader::Shutdown (C++ function), 70
apollo::cyber::ReaderBase (C++ class), 67
apollo::cyber::ReaderBase::ChannelId (C++ function), 69
apollo::cyber::ReaderBase::ClearData (C++ function), 68
apollo::cyber::ReaderBase::Empty (C++ function), 68
apollo::cyber::ReaderBase::GetChannelName (C++ function), 68
apollo::cyber::ReaderBase::GetDelaySec (C++ function), 68
apollo::cyber::ReaderBase::GetWriters (C++ function), 68
apollo::cyber::ReaderBase::HasReceived (C++ function), 68
apollo::cyber::ReaderBase::HasWriter (C++ function), 68
apollo::cyber::ReaderBase::Init (C++ function), 68
apollo::cyber::ReaderBase::IsInit (C++ function), 69
apollo::cyber::ReaderBase::Observe (C++ function), 68
apollo::cyber::ReaderBase::PendingQueueSize (C++ function), 68
apollo::cyber::ReaderBase::QosProfile (C++ function), 69
apollo::cyber::ReaderBase::Shutdown (C++ function), 68
apollo::cyber::ReaderConfig (C++ class), 74
apollo::cyber::ReaderConfig::pending_queue_size (C++ member), 74
apollo::cyber::ReaderConfig::ReaderConfig (C++ function), 74
apollo::cyber::record::HeaderBuilder (C++ class), 95

apollo::cyber::record::HeaderBuilder::GetHeader (*C++ function*), 96
apollo::cyber::record::HeaderBuilder::GetHeaderWithChunkParams (*C++ function*), 95
apollo::cyber::record::HeaderBuilder::GetHeaderWithSegmentParams (*C++ function*), 95
apollo::cyber::record::RecordBase (*C++ class*), 96
apollo::cyber::record::RecordBase::~RecordBase (*C++ function*), 96
apollo::cyber::record::RecordBase::GetChannelList (*C++ function*), 96
apollo::cyber::record::RecordBase::GetFile (*C++ function*), 97
apollo::cyber::record::RecordBase::GetHeader (*C++ function*), 97
apollo::cyber::record::RecordBase::GetMessageNumber (*C++ function*), 96
apollo::cyber::record::RecordBase::GetMessageType (*C++ function*), 96
apollo::cyber::record::RecordBase::GetProtoDesc (*C++ function*), 96
apollo::cyber::record::RecordMessage (*C++ class*), 97
apollo::cyber::record::RecordMessage::channel_name (*C++ member*), 97
apollo::cyber::record::RecordMessage::content (*C++ member*), 97
apollo::cyber::record::RecordMessage::RecordMessage (*C++ function*), 97
apollo::cyber::record::RecordMessage::time (*C++ member*), 97
apollo::cyber::record::RecordReader (*C++ class*), 97
apollo::cyber::record::RecordReader::~RecordReader (*C++ function*), 98
apollo::cyber::record::RecordReader::GetChannelList (*C++ function*), 99
apollo::cyber::record::RecordReader::GetMessageNumber (*C++ function*), 98
apollo::cyber::record::RecordReader::GetMessageType (*C++ function*), 98
apollo::cyber::record::RecordReader::GetProtoDesc (*C++ function*), 98
apollo::cyber::record::RecordReader::IsValid (*C++ function*), 98
apollo::cyber::record::RecordReader::ReadMessage (*C++ function*), 98
apollo::cyber::record::RecordReader::RecordReader (*C++ function*), 98
apollo::cyber::record::RecordReader::Reset (*C++ function*), 98
apollo::cyber::record::RecordViewer (*C++ class*), 101
apollo::cyber::record::RecordViewer::begin (*C++ function*), 102
apollo::cyber::record::RecordViewer::begin_time (*C++ function*), 102
apollo::cyber::record::RecordViewer::end (*C++ function*), 102
apollo::cyber::record::RecordViewer::end_time (*C++ function*), 102
apollo::cyber::record::RecordViewer::GetChannelList (*C++ function*), 102
apollo::cyber::record::RecordViewer::IsValid (*C++ function*), 101
apollo::cyber::record::RecordViewer::Iterator (*C++ class*), 102
apollo::cyber::record::RecordViewer::Iterator::~Ite (*C++ function*), 102
apollo::cyber::record::RecordViewer::Iterator::Iter (*C++ function*), 102
apollo::cyber::record::RecordViewer::Iterator::oper (*C++ function*), 102
apollo::cyber::record::RecordViewer::Iterator::oper (*C++ function*), 103
apollo::cyber::record::RecordViewer::Iterator::oper (*C++ function*), 103
apollo::cyber::record::RecordViewer::Iterator::oper (*C++ function*), 103
apollo::cyber::record::RecordViewer::Iterator::oper (*C++ function*), 102
apollo::cyber::record::RecordViewer::RecordViewer (*C++ function*), 101
apollo::cyber::record::RecordWriter (*C++ class*), 99
apollo::cyber::record::RecordWriter::~RecordWriter (*C++ function*), 99
apollo::cyber::record::RecordWriter::Close (*C++ function*), 99
apollo::cyber::record::RecordWriter::GetChannelList (*C++ function*), 101
apollo::cyber::record::RecordWriter::GetMessageNumb (*C++ function*), 100
apollo::cyber::record::RecordWriter::GetMessageType (*C++ function*), 100
apollo::cyber::record::RecordWriter::GetProtoDesc (*C++ function*), 100
apollo::cyber::record::RecordWriter::IsNewChannel (*C++ function*), 101
apollo::cyber::record::RecordWriter::Open (*C++ function*), 99
apollo::cyber::record::RecordWriter::RecordWriter (*C++ function*), 99
apollo::cyber::record::RecordWriter::SetIntervalOfF (*C++ function*), 100
apollo::cyber::record::RecordWriter::SetSizeOfFileS (*C++ function*), 100

apollo::cyber::record::RecordWriter::WriteChannel (*C++ function*), 99
apollo::cyber::record::RecordWriter::WriteMessage (*C++ function*), 99
apollo::cyber::Service (*C++ class*), 81
apollo::cyber::Service::destroy (*C++ function*), 82
apollo::cyber::Service::Init (*C++ function*), 82
apollo::cyber::Service::Service (*C++ function*), 81, 82
apollo::cyber::service_discovery::ChannelManager (*C++ class*), 86
apollo::cyber::service_discovery::ChannelManager::~ChannelManager (*C++ function*), 86
apollo::cyber::service_discovery::ChannelManager::ChannelManager (*C++ function*), 86
apollo::cyber::service_discovery::ChannelManager::GetChannelNames (*C++ function*), 86
apollo::cyber::service_discovery::ChannelManager::GetDownstreamOfNode (*C++ function*), 88
apollo::cyber::service_discovery::ChannelManager::GetFlowDirection (*C++ function*), 88
apollo::cyber::service_discovery::ChannelManager::GetMsgType (*C++ function*), 86
apollo::cyber::service_discovery::ChannelManager::GetProtoDesc (*C++ function*), 86
apollo::cyber::service_discovery::ChannelManager::GetReaders (*C++ function*), 87
apollo::cyber::service_discovery::ChannelManager::GetReadersOfChannel (*C++ function*), 88
apollo::cyber::service_discovery::ChannelManager::GetReadersOfNode (*C++ function*), 87
apollo::cyber::service_discovery::ChannelManager::GetUpstreamOfNode (*C++ function*), 88
apollo::cyber::service_discovery::ChannelManager::GetWriters (*C++ function*), 87
apollo::cyber::service_discovery::ChannelManager::GetWritersOfChannel (*C++ function*), 87
apollo::cyber::service_discovery::ChannelManager::GetWritersOfNode (*C++ function*), 87
apollo::cyber::service_discovery::ChannelManager::HasReader (*C++ function*), 87
apollo::cyber::service_discovery::ChannelManager::HasWriter (*C++ function*), 86
apollo::cyber::service_discovery::ChannelManager::IsMessageTypeMatching (*C++ function*), 88
apollo::cyber::service_discovery::Manager (*C++ class*), 84
apollo::cyber::service_discovery::Manager::~Manager (*C++ function*), 85
apollo::cyber::service_discovery::Manager::AddChangeListener (*C++ function*), 85
apollo::cyber::service_discovery::Manager::Join (*C++ function*), 85
apollo::cyber::service_discovery::Manager::Leave (*C++ function*), 85
apollo::cyber::service_discovery::Manager::Manager (*C++ function*), 85
apollo::cyber::service_discovery::Manager::OnTopoMo (*C++ function*), 86
apollo::cyber::service_discovery::Manager::RemoveCh (*C++ function*), 85
apollo::cyber::service_discovery::Manager::Shutdown (*C++ function*), 85
apollo::cyber::service_discovery::Manager::StartDis (*C++ function*), 85
apollo::cyber::service_discovery::Manager::StopDisc (*C++ function*), 85
apollo::cyber::service_discovery::NodeManager (*C++ class*), 89
apollo::cyber::service_discovery::NodeManager::~Noc (*C++ function*), 89
apollo::cyber::service_discovery::NodeManager::GetN (*C++ function*), 89
apollo::cyber::service_discovery::NodeManager::HasN (*C++ function*), 89
apollo::cyber::service_discovery::NodeManager::Node (*C++ function*), 89
apollo::cyber::service_discovery::ServiceManager (*C++ class*), 89
apollo::cyber::service_discovery::ServiceManager:: (*C++ function*), 89
apollo::cyber::service_discovery::ServiceManager:: (*C++ function*), 90
apollo::cyber::service_discovery::ServiceManager:: (*C++ function*), 90
apollo::cyber::service_discovery::ServiceManager:: (*C++ function*), 89
apollo::cyber::service_discovery::ServiceManager:: (*C++ function*), 89
apollo::cyber::service_discovery::TopologyManager (*C++ class*), 90
apollo::cyber::service_discovery::TopologyManager: (*C++ function*), 90
apollo::cyber::service_discovery::TopologyManager: (*C++ function*), 91
apollo::cyber::service_discovery::TopologyManager: (*C++ function*), 90
apollo::cyber::service_discovery::TopologyManager: (*C++ function*), 90
apollo::cyber::service_discovery::TopologyManager: (*C++ function*), 91
apollo::cyber::service_discovery::TopologyManager: (*C++ function*), 90
apollo::cyber::ServiceBase (*C++ class*), 81
apollo::cyber::ServiceBase::service_name (*C++ function*), 81

apollo::cyber::ServiceBase::ServiceBase (*C++ function*), 81
apollo::cyber::Time (*C++ class*), 94
apollo::cyber::Time::IsZero (*C++ function*), 95
apollo::cyber::Time::Now (*C++ function*), 95
apollo::cyber::Time::SleepUntil (*C++ function*), 95
apollo::cyber::Time::ToNanosecond (*C++ function*), 94
apollo::cyber::Time::ToSecond (*C++ function*), 94
apollo::cyber::Time::ToString (*C++ function*), 95
apollo::cyber::Timer (*C++ class*), 94
apollo::cyber::Timer::SetTimerOption (*C++ function*), 94
apollo::cyber::Timer::Start (*C++ function*), 94
apollo::cyber::Timer::Stop (*C++ function*), 94
apollo::cyber::Timer::Timer (*C++ function*), 94
apollo::cyber::TimerComponent (*C++ class*), 91
apollo::cyber::TimerComponent::Initialize (*C++ function*), 92
apollo::cyber::TimerOption (*C++ class*), 93
apollo::cyber::TimerOption::callback (*C++ member*), 93
apollo::cyber::TimerOption::oneshot (*C++ member*), 93
apollo::cyber::TimerOption::period (*C++ member*), 93
apollo::cyber::TimerOption::TimerOption (*C++ function*), 93
apollo::cyber::Writer (*C++ class*), 72
apollo::cyber::Writer::GetReaders (*C++ function*), 73
apollo::cyber::Writer::HasReader (*C++ function*), 73
apollo::cyber::Writer::Init (*C++ function*), 73
apollo::cyber::Writer::Shutdown (*C++ function*), 73
apollo::cyber::Writer::Write (*C++ function*), 73
apollo::cyber::Writer::Writer (*C++ function*), 73
apollo::cyber::WriterBase (*C++ class*), 71
apollo::cyber::WriterBase::GetChannelName (*C++ function*), 72
apollo::cyber::WriterBase::GetReaders (*C++ function*), 72
apollo::cyber::WriterBase::HasReader (*C++ function*), 72
apollo::cyber::WriterBase::Init (*C++ function*), 72
apollo::cyber::WriterBase::IsInit (*C++ function*), 72
apollo::cyber::WriterBase::Shutdown (*C++ function*), 72
apollo::cyber::WriterBase::WriterBase (*C++ function*), 72

C

cyber_py.cyber.ChannelUtils (*built-in class*), 107
cyber_py.cyber.ChannelUtils.get_channels() (*built-in function*), 108
cyber_py.cyber.ChannelUtils.get_channels_info() (*built-in function*), 108
cyber_py.cyber.ChannelUtils.get_debugstring_rawmsgdata() (*built-in function*), 107
cyber_py.cyber.ChannelUtils.get_msgtype() (*built-in function*), 107
cyber_py.cyber.Client (*built-in class*), 107
cyber_py.cyber.Client.send_request() (*built-in function*), 107
cyber_py.cyber.init() (*built-in function*), 105
cyber_py.cyber.is_shutdown() (*built-in function*), 105
cyber_py.cyber.Node (*built-in class*), 105
cyber_py.cyber.Node.create_client() (*built-in function*), 106
cyber_py.cyber.Node.create_rawdata_reader() (*built-in function*), 106
cyber_py.cyber.Node.create_reader() (*built-in function*), 106
cyber_py.cyber.Node.create_service() (*built-in function*), 106
cyber_py.cyber.Node.create_writer() (*built-in function*), 105
cyber_py.cyber.Node.register_message() (*built-in function*), 105
cyber_py.cyber.Node.spin() (*built-in function*), 107
cyber_py.cyber.NodeUtils (*built-in class*), 108
cyber_py.cyber.NodeUtils.get_node_attr() (*built-in function*), 108
cyber_py.cyber.NodeUtils.get_nodes() (*built-in function*), 108
cyber_py.cyber.NodeUtils.get_readersofnode() (*built-in function*), 108
cyber_py.cyber.NodeUtils.get_writersofnode() (*built-in function*), 109
cyber_py.cyber.ok() (*built-in function*), 105
cyber_py.cyber.Reader (*built-in class*), 107

cyber_py.cyber.shutdown() (*built-in function*), 105
cyber_py.cyber.waitforshutdown() (*built-in function*), 105
cyber_py.cyber.Writer (*built-in class*), 107
cyber_py.cyber.Writer.write() (*built-in function*), 107
cyber_py.cyber_time.Duration (*built-in class*), 111
cyber_py.cyber_time.Duration.sleep() (*built-in function*), 111
cyber_py.cyber_time.Duration.to_nsec() (*built-in function*), 111
cyber_py.cyber_time.Duration.to_sec() (*built-in function*), 111
cyber_py.cyber_time.Rate (*built-in class*), 112
cyber_py.cyber_time.Rate.__init__() (*built-in function*), 112
cyber_py.cyber_time.Rate.get_cycle_time() (*built-in function*), 112
cyber_py.cyber_time.Rate.get_expected_cycle_time() (*built-in function*), 112
cyber_py.cyber_time.Rate.reset() (*built-in function*), 112
cyber_py.cyber_time.Rate.sleep() (*built-in function*), 112
cyber_py.cyber_time.Time (*built-in class*), 111
cyber_py.cyber_time.Time.__init__() (*built-in function*), 111
cyber_py.cyber_time.Time.now() (*built-in function*), 112
cyber_py.cyber_time.Time.sleep_until() (*built-in function*), 112
cyber_py.cyber_time.Time.to_nsec() (*built-in function*), 112
cyber_py.cyber_time.Time.to_sec() (*built-in function*), 111
cyber_py.cyber_timer.Timer (*built-in class*), 112
cyber_py.cyber_timer.Timer.__init__() (*built-in function*), 112
cyber_py.cyber_timer.Timer.set_option() (*built-in function*), 113
cyber_py.cyber_timer.Timer.start() (*built-in function*), 113
cyber_py.cyber_timer.Timer.stop() (*built-in function*), 113
cyber_py.parameter.Parameter (*built-in class*), 113
cyber_py.parameter.Parameter.as_double() (*built-in function*), 113
cyber_py.parameter.Parameter.as_int64() (*built-in function*), 113
cyber_py.parameter.Parameter.as_string() (*built-in function*), 113
cyber_py.parameter.Parameter.debug_string() (*built-in function*), 113
cyber_py.parameter.Parameter.descriptor() (*built-in function*), 113
cyber_py.parameter.Parameter.name() (*built-in function*), 113
cyber_py.parameter.Parameter.type_name() (*built-in function*), 113
cyber_py.parameter.ParameterClient (*built-in class*), 114
cyber_py.parameter.ParameterClient.__init__() (*built-in function*), 114
cyber_py.parameter.ParameterClient.get_parameter() (*built-in function*), 114
cyber_py.parameter.ParameterClient.get_paramslist() (*built-in function*), 114
cyber_py.parameter.ParameterClient.set_parameter() (*built-in function*), 114
cyber_py.parameter.ParameterServer (*built-in class*), 113
cyber_py.parameter.ParameterServer.__init__() (*built-in function*), 113
cyber_py.parameter.ParameterServer.get_parameter() (*built-in function*), 114
cyber_py.parameter.ParameterServer.get_paramslist() (*built-in function*), 114
cyber_py.parameter.ParameterServer.set_parameter() (*built-in function*), 114
cyber_py.record.RecordReader (*built-in class*), 109
cyber_py.record.RecordReader.__init__() (*built-in function*), 109
cyber_py.record.RecordReader.get_channellist() (*built-in function*), 110
cyber_py.record.RecordReader.get_headerstring() (*built-in function*), 110
cyber_py.record.RecordReader.get_messagenumber() (*built-in function*), 109
cyber_py.record.RecordReader.get_messagetype() (*built-in function*), 109
cyber_py.record.RecordReader.get_protodesc() (*built-in function*), 110
cyber_py.record.RecordReader.read_messages() (*built-in function*), 109
cyber_py.record.RecordReader.reset() (*built-in function*), 110
cyber_py.record.RecordWriter (*built-in class*), 110
cyber_py.record.RecordWriter.__init__() (*built-in function*), 110
cyber_py.record.RecordWriter.close() (*built-in function*), 110
cyber_py.record.RecordWriter.get_messagenumber()

(*built-in function*), 111
`cyber_py.record.RecordWriter.get_messagetype()`
(*built-in function*), 111
`cyber_py.record.RecordWriter.get_protodesc()`
(*built-in function*), 111
`cyber_py.record.RecordWriter.open()`
(*built-in function*), 110
`cyber_py.record.RecordWriter.set_intervaltime_fileseg()`
(*built-in function*), 111
`cyber_py.record.RecordWriter.set_size_fileseg()`
(*built-in function*), 111
`cyber_py.record.RecordWriter.write_channel()`
(*built-in function*), 110
`cyber_py.record.RecordWriter.write_message()`
(*built-in function*), 110