

专业硕士学位论文



自动驾驶运行时通信系统 的设计与实现

|  |  |
| --- | --- |
| 作者姓名： | 葛润泽 |
| 专业领域： | 软件工程 |
| 校内导师： | 李春杰 副研究员 |
| 企业导师： | 伊威宇 |
| 完成时间： | 二〇二二年七月二十日 |

University of Science and Technology of China A dissertation for master’s degree



**Design** **and** **implementation** **of**

**autonomous** **driving** **runtime**

**communication** **system**

Author: GE RUNZE

Speciality: Software Engineering

Supervisors: Prof. XXX, Prof. XXX

Advisor: Weiyu Yi

Finished time: July 20, 2022

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文， 是本人在导师指导下进行研究工作所取得的 成果。除已特别加以标注和致谢的地方外， 论文中不包含任何他人已经发表或撰 写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了 明确的说明。

作者签名： 签字日期：

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一， 学位论文著作权拥有者授权中国科学技术大学 拥有学位论文的部分使用权， 即：学校有权按有关规定向国家有关部门或机构送 交论文的复印件和电子版， 允许论文被查阅和借阅， 可以将学位论文编入《中国 学位论文全文数据库》等有关数据库进行检索， 可以采用影印、缩印或扫描等复 制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一 致。

控阅的学位论文在解密后也遵守此规定。

 公开 □ 控阅( 年)

作者签名：

签字日期：

导师签名：

签字日期：

摘 要

草泥马摘要是论文内容的总结概括， 应简要说明论文的研究目的、基本研究 内容、研究方法或过程、结果和结论， 突出论文的创新之处。摘要中不宜使用公 式、图表， 不引用文献。博士论文中文摘要一般 800~1000 个汉字， 硕士论文中 文摘要一般 600 个汉字。英文摘要的篇幅参照中文摘要。

关键词另起一行并隔写在摘要下方， 一般 3~8 个词， 中文关键词间空一字 或用分号“；”隔开。英文摘要的关键词与中文摘要的关键词应完全一致， 中间 用逗号“, ”或分号“;”隔开。

关键词：中国科学技术大学 学位论文 LATEX 模板 学士 硕士 博士

**ABSTRACT**

This is a sample document of USTC thesis LATEX template for bachelor, master and doctor. The template is created by zepinglee and seisman, which orignate from the template created by ywg. The template meets the equirements of USTC theiss writing standards.

This document will show the usage of basic commands provided by LATEX and some features provided by the template. For more information, please refer to the tem- plate document ustcthesis.pdf.

**Key** **Words**: University of Science and Technology of China (USTC), Thesis, LATEX Template, Bachelor, Master, PhD

目 录

第 1 章 [绪论](#_bookmark1) [1](#_bookmark1)

1.1 [选题背景及意义](#_bookmark2) [1](#_bookmark2)

1.2 [国内外研究现状](#_bookmark3) [2](#_bookmark3)

1.2.1 [国内研究现状](#_bookmark4) [3](#_bookmark4)

1.2.2 [国外研究现状](#_bookmark5) [4](#_bookmark5)

1.3 [论文主要工作内容](#_bookmark6) [6](#_bookmark6)

1.4 [章节安排](#_bookmark7) [6](#_bookmark7)

第 2 章 [相关技术](#_bookmark8) [8](#_bookmark8)

2.1 [ZeroMQ](#_bookmark9) [8](#_bookmark9)

2.1.1 [ZeroMQ 通信模式](#_bookmark10) [9](#_bookmark10)

2.1.2 [ZeroMQ 性能表现](#_bookmark11) [13](#_bookmark11)

2.2 [序列化技术](#_bookmark12) [13](#_bookmark12)

2.3 [远程服务调用](#_bookmark13) [15](#_bookmark13)

2.4 [共享内存](#_bookmark14) [16](#_bookmark14)

2.5 [本章小结](#_bookmark15) [18](#_bookmark15)

第 3 章 [自动驾驶运行时通信系统需求分析](#_bookmark16) [19](#_bookmark16)

3.1 [业务需求分析](#_bookmark17) [19](#_bookmark17)

3.1.1 [自动驾驶系统通信特点分析](#_bookmark18) [19](#_bookmark18)

3.1.2 [业务流程描述](#_bookmark19) [21](#_bookmark19)

3.2 [功能性需求](#_bookmark20) [22](#_bookmark20)

3.2.1 [通信单元管理](#_bookmark21) [22](#_bookmark21)

3.2.2 [服务管理](#_bookmark22) [24](#_bookmark22)

3.2.3 [通信方式自适应选择](#_bookmark23) [26](#_bookmark23)

3.2.4 [调度策略管理](#_bookmark24) [26](#_bookmark24)

3.3 [非功能性需求](#_bookmark25) [27](#_bookmark25)

3.3.1 [实时性](#_bookmark26) [27](#_bookmark26)

3.3.2 [安全性](#_bookmark27) [27](#_bookmark27)

3.3.3 [可靠性](#_bookmark28) [27](#_bookmark28)

3.3.4 [可扩展性](#_bookmark29) [27](#_bookmark29)

3.4 [本章小结](#_bookmark30) [28](#_bookmark30)

第 4 章 [自动驾驶运行时通信系统概要设计](#_bookmark31) [29](#_bookmark31)

4.1 [系统总体设计](#_bookmark32) [29](#_bookmark32)

4.1.1 [系统架构设计](#_bookmark33) [29](#_bookmark33)

4.1.2 [自动驾驶运行时通信组网设计](#_bookmark34) [30](#_bookmark34)

4.1.3 [系统功能模块设计](#_bookmark35) [31](#_bookmark35)

4.2 [通信单元模块概要设计](#_bookmark36) [33](#_bookmark36)

4.2.1 [订阅者模块概要设计](#_bookmark37) [33](#_bookmark37)

4.2.2 [发布者模块概要设计](#_bookmark38) [36](#_bookmark38)

4.3 [服务模块概要设计](#_bookmark39) [37](#_bookmark39)

4.3.1 [服务客户端设计](#_bookmark40) [37](#_bookmark40)

4.3.2 [服务服务端设计](#_bookmark41) [39](#_bookmark41)

4.4 [通信抽象模块概要设计](#_bookmark42) [41](#_bookmark42)

4.4.1 [通信抽象类概要设计](#_bookmark43) [41](#_bookmark43)

4.4.2 [抽象发布类概要设计](#_bookmark44) [42](#_bookmark44)

4.4.3 [抽象订阅类概要设计](#_bookmark45) [43](#_bookmark45)

4.4.4 [抽象服务类概要设计](#_bookmark46) [43](#_bookmark46)

4.5 [通信传输模块概要设计](#_bookmark47) [44](#_bookmark47)

4.5.1 [进程内通信模块设计](#_bookmark48) [44](#_bookmark48)

4.5.2 [进程间通信模块概要设计](#_bookmark49) [45](#_bookmark49)

4.5.3 [网络通信模块概要设计](#_bookmark50) [46](#_bookmark50)

4.5.4 [消息模块概要设计](#_bookmark51) [47](#_bookmark51)

4.6 [服务发现模块概要设计](#_bookmark52) [47](#_bookmark52)

4.7 [任务模块概要设计](#_bookmark53) [49](#_bookmark53)

4.8 [调度模块概要设计](#_bookmark54) [50](#_bookmark54)

4.9 [中心节点概要设计](#_bookmark55) [52](#_bookmark55)

4.10 [本章小节](#_bookmark56) [53](#_bookmark56)

第 5 章 [自动驾驶运行时通信系统详细设计与实现](#_bookmark57) [54](#_bookmark57)

5.1 [系统软件架构设计](#_bookmark58) [54](#_bookmark58)

5.2 [通信单元模块详细设计与实现](#_bookmark59) [54](#_bookmark59)

5.2.1 [发布者模块详细设计与实现](#_bookmark60) [55](#_bookmark60)

5.2.2 [订阅者模块详细设计与实现](#_bookmark61) [57](#_bookmark61)

5.3 [服务模块详细设计与实现](#_bookmark62) [59](#_bookmark62)

5.3.1 [服务端详细设计与实现](#_bookmark63) [60](#_bookmark63)

5.3.2 [客户端详细设计与实现](#_bookmark64) [61](#_bookmark64)

5.4 [通信抽象模块详细设计与实现](#_bookmark65) [63](#_bookmark65)

5.4.1 [通信方式自适应选择算法实现](#_bookmark66) [64](#_bookmark66)

5.4.2 [抽象服务类详细设计与实现](#_bookmark67) [66](#_bookmark67)

5.5 [通信传输模块详细设计与实现](#_bookmark68) [67](#_bookmark68)

5.5.1 [进程内通信模块详细设计与实现](#_bookmark69) [67](#_bookmark69)

5.5.2 [进程间通信模块详细设计与实现](#_bookmark70) [69](#_bookmark70)

5.5.3 [网络通信模块详细设计与实现](#_bookmark71) [75](#_bookmark71)

5.5.4 [消息模块详细设计与实现](#_bookmark72) [75](#_bookmark72)

5.6 [服务发现模块详细设计与实现](#_bookmark73) [75](#_bookmark73)

5.6.1 [服务发现模块类设计](#_bookmark74) [75](#_bookmark74)

5.6.2 [发现机制设计](#_bookmark75) [76](#_bookmark75)

5.7 [任务模块详细设计与实现](#_bookmark76) [78](#_bookmark76)

5.8 [调度模块详细设计与实现](#_bookmark77) [78](#_bookmark77)

5.8.1 [基于消息触发的调度详细设计与实现](#_bookmark78) [79](#_bookmark78)

5.8.2 [基于时间触发的调度详细设计与实现](#_bookmark79) [79](#_bookmark79)

5.9 [中心节点详细设计与实现](#_bookmark80) [80](#_bookmark80)

第 6 章 [自动驾驶运行时通信系统测试与分析](#_bookmark81) [81](#_bookmark81)

6.1 [测试环境](#_bookmark82) [81](#_bookmark82)

6.2 [功能性测试](#_bookmark83) [81](#_bookmark83)

6.3 [非功能性测试](#_bookmark84) [81](#_bookmark84)

[参考文献](#_bookmark85) [82](#_bookmark85)

[致谢](#_bookmark86) [85](#_bookmark86)

第 **1** 章 绪 论

1.1 选题背景及意义

随着近几年国内外自动驾驶相关技术的迅速发展， 越来越多的公司开始从 事自动驾驶行业的研究和商业应用探索。目前从事于自动驾驶技术研发的企业 主要分为三种： 首先是以通用、大众等传统的车企， 这类车企对于车辆的改装拥

有丰富的经验但对自动驾驶需要的相关技术比较欠缺； 其次是谷歌、百度等互联 网企业， 这类互联网企业擅长于自动驾驶架构研发并且工程经验相当丰富， 同时 积累了大量的机器学习相关的技术沉淀； 另外一类是以蔚来、小马智行等新兴的 自动驾驶企业， 这一类企业以视觉感知算法、规划控制算法为突破口[[1](#_bookmark87)]。鉴于 车企对车辆本身的研发和改装有丰富的技术沉淀而以纯自动驾驶技术为导向的 新兴科技企业缺乏对车辆的技术积累， 而自动驾驶技术的落地的前提就是自动 驾驶车辆的大规模量产， 所以传统车企与新兴技术企业合作的模式孕育而生[[2](#_bookmark88)] 。 到目前为止， 传统车企与新兴科技企业合作项目大多集中在 RoboTaxi 上， 并且

RoboTaxi 已经取得了从有到无的突破。国外的 Waymo 和国内的小马智行、文远 知行已经在不同的城市开启了 RoboTaxi 的相关业务， 实现了初步商用的目标[[3](#_bookmark89)] 。

虽然自动驾驶的量产和商业化取得了突破性的进展， 有关自动驾驶商业化 应用落地的目标也初步实现， 但是现有的自动驾驶各模块的通信框架并不能满 足性能上的要求以及扩展性的需求。上汽集团旗下分公司享道与 Momenta(魔门 塔) 合作开发出享道 Robotaxi 并且已经实现了上海和苏州两地的运营， 但其自动 驾驶内部通信架构仍然采用 ROS(机器人操作系统) 作为各个功能模块的通信方

法[[4](#_bookmark90)] 。ROS 是一款主要面向与机器人研究领域的开源的元操作系统， 为开发者 提供了进程间通信的相关功能并且支持分布式通信， 大大简化了开发过程和开 发量。 ROS 存在一些无法忽视的缺点， 主要缺点分为通信性能问题和扩展性问 题[[5](#_bookmark91)] 。

一方面， 自动驾驶系统中使用了种类多、数量大的传感器为自动驾驶算法模 块提供车辆周围的动态信息与静态信息， 数据量达到了 Gbps 级别且对通信延迟 非常敏感。而 ROS1 提供了 TCPROS 和 UDPROS 两种通信方式实现不同进程间 的通信， 虽然这种通信方式在各个平台上的兼容性最好， 但此通信方式并未对同 一物理机和不同物理机的进程间通信两种情况进行区分。在同一物理机进程间 通信的情况下， 由于操作系统对 TCP/IP 协议栈的实现， 发送方发送一份数据会 从用户态拷贝到内核态， 而接收方需要从逆向地从内核态拷贝到用户态， 通信过 程中存在数据的额外拷贝的问题， 在传输图像等数据量较大的数据时通信延迟 会随数据量大小呈线性增长趋势。特别地， 当存在 1-N 通信模式(一发多收) 时

第 1 章 绪 论

通信延迟会进一步上升[[6](#_bookmark92)-[7](#_bookmark93)] 。ROS2 引入了 DDS(Data Distributed Service, 数据分 发服务) 作为其通信中间件[[8](#_bookmark94)] ，DDS 支持共享内存方式的进程间通信， 有效减少 了因数据拷贝带来的通信延迟。但 DDS 自身对数据序列化格式单独做出了定义， 与 ROS2 的序列化格式并不相同，通信过程中存在多次序列化的问题[[7](#_bookmark93)] 。

另一方面， 自动驾驶系统的软件架构设计正迈向 SOA(Service Oriented Archi- tecture) 架构风格， 即将下载高精度地图、与云端服务器交互等功能以微服务的 形式进行数据传输。而 ROS 消息的发布-订阅和服务通过 ROS 内置的.msg 和.srv 文件定义并且传输过程中都会被系统内部序列化成二进制序列的形式。 ROS 实 现了特有的序列化方法， 且此方法不能做到向前和向后兼容。自动驾驶量产和商 业化应用后自动驾驶系统需要符合 SOA 的架构风格， 这表明自动驾驶系统需要 具备与各类型终端交互的能力， 而序列化方法的兼容性成为关键， ROS 的序列 化方法并不适合这种场景。

因此， 大多数自动驾驶公司在初期都优先采用 ROS 作为自己的通信架构， 但这种方法只能作为前期对产品功能开发和验证环节上的解决办法， 并不适合 作为最终量产和商业化应用的技术方案。开发一款可以优化或解决上述 ROS 缺 点的新通信方案是必经之路[[9](#_bookmark95)] 。

综上所述， 实现一个通信性能高、扩展性及兼容性强的自动驾驶通信架构不 仅仅是对自动驾驶通信架构本身的优化， 还可以为将来自动驾驶系统与用户或 其他终端进行通信互动打下基础。因此， 本文旨在研究一个具有分布式通信能 力， 并且能够根据场景自适应选择通信方式的自动驾驶运行时通信系统， 优化自 动驾驶通信系统通信性能， 提高其多终端数据交互能力，，具有一定的研究意义。

1.2 国内外研究现状

在上一节提到的自动驾驶系统中使用 ROS 作为通信方案的弊端在国内外从 事自动驾驶的企业已经注意并且已经开始寻找替代方案。尽管 ROS2 的发布修 复了 ROS1 的一些缺点并且优化了整个通信链路的性能和稳定性， 但是自研通信 架构已经成为自动驾驶行业的主流。百度、博世、大陆、 Autoware 基金会等都对 通信架构开始了研究， 在产品上用自研架构取代了 ROS。虽然自研架构的路线 纷纷被各家企业认可， 但是这并不意味着自动驾驶行业完全抛弃了 ROS，从各 大企业发布的自研通信架构可以看出其中有部分思想和实现方法依然借鉴或沿 用了 ROS 的相关思路， 大部分自研的通信框架保留了 ROS 中发布-订阅的通信 模式并且也都提供类似于 ROS 中服务 (ROS Service) 的相关功能。

1.2.1 国内研究现状

2019 年， 国内的百度公司为旗下的自动驾驶业务线 Apollo 研究并发布了一 套自研的分布式通信框架， 即 CyberRT 。CyberRT 是一套以 ROS 为参考并且致 力于优化 ROS 缺陷， 做到高并发、低延迟和高吞吐的通信系统， 提供了类似于 ROS 中的订阅者 (Subscriber)、发布者 (Publisher) 和服务 (Service) 的通信角色。 CyberRT 在设计中采用了分层次开发方法， 涵盖了从最上层提供给用户的相关接 口到最下层的通信方法， 用户只需要关注最上层 API 的应用而无需关注框架的 内部实现。同时， 分层次的设计方法使得整个框架的功能耦合度很低， 其架构如 图 1.1 所示[[10](#_bookmark96)] ：

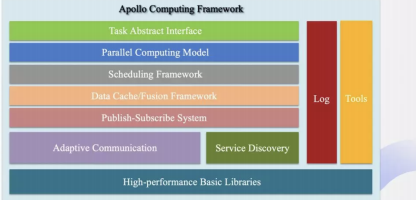


图 **1.1** **CyberRT** 架构图

根据系统的架构图可以得出 CyberRT 将整个通信系统分成了 6 层， 下文将 选取任务抽象层、调度层、数据通信层和服务发现层进行论述：

1. 任务抽象层: 该层的主要功能为向用户提供注册订阅者、发布者以及服务 的接口， 用户通过该接口可以快速将需要接收和发布的话题在通信节点内 进行注册， 如果需要注册供其他通信节点调用的服务， 也可以一起注册在 节点中。当用户注册完所有的通信功能后， 一个通信节点就被调度层加载 到系统内与其他节点进行数据交互和服务调用。在注册时， API 内部会自 动获取该节点的实际 IP 地址、端口号和进程号， 下层根据这些数据选择通 信方式。

2. 调度层： 该层的功能是将任务抽象层中用户自定义的通信节点加载到调度 队列中， 调度策略默认是基于数据驱动， 当某个通信节点收到了某条信息 或需要发送某条信息则将唤醒该节点并执行程序。该层也支持基于优先级 的调度策略， 当通信节点需要更高优先级来进行任务调度时用户可以使用 该种调度策略。

3. 数据通信层： 该层的功能是通过注册在系统中的话题， 找到匹配的发布者 和订阅者并根据实际情况选择使用共享内存或网络来进行数据通信。具体

方法是： 根据某一个话题匹配到发布或订阅该话题的数个发布者和订阅者， 这些发布者或订阅者内部都携带用户注册时程序所在机器上的 IP 地址、进 程号和端口号。通过比较这三个数据可以知道节点之间是否在同一台物理 机上或处于不同的物理机上， 如果处于同一台物理机上则选择共享内存的 通信方式，如果不在同一台物理机上则选择网络通信。

4. 服务发现层： CyberRT 支持分布式通信， 分布式通信代表着系统通信节点 的拓扑网络会动态改变， 而每个节点的服务也会动态变化， 此情况下保持 所有通信节点能正确接收到网络拓扑的变化并且更新自身的通信网络至最 新状态。该层最重要的功能是满足了分布式通信的最基本需求根据动态的 通信拓扑去更新通信网络中每一个节点的通信状态。

1.2.2 国外研究现状

Edwin Olson 等人在 2010 年提出了名为 LCM[[11](#_bookmark97)]的自动驾驶通信系统， 该通 信系统基于 UDP 多播方式实现了发布-订阅的通信模式。该系统中发布者发送的 数据会通过 UDP 多播传输至每一个订阅者， 无论订阅者是否订阅了对应的话题。 订阅者需要丢弃不属于自身订阅的数据，造成了大量的资源消耗。

Neil Dantam 等人在 2012 年弃用了基于网络的进程间通信方式， 将共享内存 作为进程间通信方式， 并将该方式集成到名为 Ach[[12](#_bookmark98)]的通信系统并运用于自动 驾驶等复杂机器人应用领域。 Ach 系统通过使用共享内存的进程间通信方式避 免了使用 TCP 通信带来的队头阻塞问题 (Head-of-Line Blocking Problem)，有效 减少了 TCP 有序传输机制下数据传输过程中出现乱序而带来的阻塞延迟[[13](#_bookmark99)] 。

Wei Liu 等人在 2020 年结合自动驾驶系统各节点信息流传递特点给出了 ROS 的通信延迟实验结果， 并提出了基于共享内存的 Z-framework[[14](#_bookmark100)]。该框架 使用共享内存作为进程间通信手段， 有效地降低了 ROS 在进程间通信延迟方面 的缺陷。但该框架存在读者或写者崩溃时整个共享内存块将无法被再次使用最 终无法完成通信。

罗伯特 ·博世公司自动驾驶部门开发了一款专注于基于共享内存实现进程 间通信的自动驾驶通信中间件 iceoryx(冰羚)，该中间件由 Eclipse 在 2021 年开源 并面向行业内提供自动驾驶系统通信方案。 iceoryx 和CyberRT 非常类似， 都采 用了发布-订阅的通信模型和服务发现机制。该中间件的特点是对共享内存的同 步方法做出了改进。传统的共享内存数据同步的方法是利用操作系统提供的信 号量或锁来保证临界资源的安全性， 但这种方式的缺点是信号量和锁的粒度太 大， 不利于提高程序的并发性。而 iceoryx 通过原子操作来实现了基于无锁队列 的共享内存数据结构， 这使得中间件的并发性和吞吐量相比于传统的使用锁来 实现数据同步的通信方法获得了比较大的提升[[15](#_bookmark101)] 。

虽然 iceoryx 的进程间通信性能非常好， 但缺点是该通信中间件不支持分布 式通信， 这就意味着使用此中间件就无法完成跨机器的通信。但作为进程间通信 中间件本身而言， 它的优点也是不容忽视的。德国大陆公司开源的 eCAL 自动驾 驶通信中间件就将 iceoyrx 作为进程间通信的子模块， 而 eCAL 自身实现了网络 通信的功能。不仅 eCAL 使用了iceoryx 作为进程间通信的实现， ROS2、Cyclone DDS 等分布式通信系统也都使用了 iceoryx[[16](#_bookmark102)] 。

通过上文对国内外研究现状的阐述可以发现， 目前国内外自动驾驶通信系 统都采用共享内存作为进程间通信的方式。但共享内存使用起来有一定的难度， 需要考虑到数据同步的问题以及如何进行异常处理。常用的组织共享内存的数 据结构为缓行缓冲区 (RingBuffer)。该数据结构将固定大小的共享内存区切分成 n 个固定大小的数据块， 将其从 0 到 n- 1 进行编号， 同时维护写入序号和读取序 号。数据块是循环使用的， 向共享内存写入数据时会使用满足条件的最小编号的 数据块， 并且写入完成后会将下一个可以使用的编号自增， 当编号是 n- 1 的数据 块被写入数据后， 会再次在编号为 0 的数据块写入数据。该数据结构为了实现数 据的同步， 将每个数据块看作为一个临界资源， 当有发布者写或订阅者读都会尝

试请求锁并获得某个数据块的访问权限进行读或写， 确保不会发生脏读和脏写 的情况[[17](#_bookmark103)]。如图 1.2 所示，但此数据同步方法并不能保证程序在异常退出时能

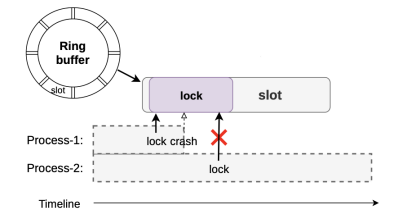


图 **1.2** 环形共享内存缓冲区数据结构示意图

够及时地释放资源， 当数据发布者在获得了某个数据块的锁并且正在写入数据 时由于异常提前终止了进程， 并不会释放已经获得的锁， 导致其他的发布者无法 获得数据块的写入权限从而发生死锁， 系统将无法完成数据通信。同样地， 当数 据订阅者在读取数据时异常终止进程导致读取序号无法正常更新， 发布者在写 入数据时最终触发缓冲区已满的情况，出现无法写入数据的情况。

另一个不容忽视的问题是服务发现机制对系统性能影响。对于自动驾驶系 统使用场景来说， 通信节点存在动态增减的情况， 所以需要要服务发现保证通信 节点间及时发现新增节点或被删除的节点。 ROS1 提供了中心化的服务发现机制

以同步网络内的节点和话题并且控制节点的生命周期， 称 Master 节点。该设计 在 Master 节点异常终止情况下无法保证正常工作； ROS2 选用了去中心化的服务 发现机制， 每个通信节点保存所有的节点和话题信息。该设计下， 通信节点的运 行状态由心跳包判定，当节点数量达到数百个时会消耗大量处理器资源[[18](#_bookmark104)] 。

1.3 论文主要工作内容

本论文主要研究自动驾驶运行时通信系统， 根据自动驾驶通信系统的研究现 状， 将进程间通信技术和网络通信技术结合起来， 提出基于共享内存、 ZeroMQ、 RPC 和序列化技术的自动驾驶运行时通信系统设计方案并进行具体实现。与现 有的自动驾驶系统通信系统不同的是， 本文将实现一个支持网络通信 (ZeroMQ)、 进程间通信 (共享内存) 和进程内通信的发布-订阅通信模式和支持服务端-客户 端的通信模式的自动驾驶运行时通信系统， 同时对共享内存通信的鲁棒性和服 务发现机制的性能做出改进。本论文主要工作如下：

1. 需求分析： 通过对自动驾驶通信特点的分析与归纳， 导出本文自动驾驶运 行时通信系统应具有的基本功能和基本业务流程， 总结出业务需求。在业 务需求的基础上， 使用面向对象的建模方式， 对自动驾驶运行时通信系统 进行功能性需求分析，通过 UML 和用例说明表对用例进行说明。

2. 系统概要设计： 基于功能性需求分析和分层次软件设计思想， 对系统整体 架构、组网方案、功能模块依赖关系进行设计， 结合 UML 类图和活动图 对功能模块进行概要性设计，明确功能性模块运行流程。

3. 系统详细设计与实现： 结合关键性数据结构说明表及时序图对功能模块重 要功能进行详细说明， 通过伪代码描述关键算法。重点从共享内存数据结 构的鲁棒性和服务发现机制性能的改进方法进行设计与实现展开论述。

4. 系统测试： 基于功能性需求分析， 对系统功能完整性进行功能性测试， 从 通信延迟、系统资源消耗等维度对系统进行性能测试。

1.4 章节安排

本论文的章节结构如图 1.3 所示，章节安排如下：

第一章， 绪论。本章首先对自动驾驶中通信系统现有的应用场景和缺陷进行 了相关介绍， 具体包括本论文的研究背景介绍以及对研究现状的介绍与分析， 其 次说明了本论文的研究意义以及本文的主要工作内容。

第二章， 相关技术介绍。本章主要介绍了网络通信中间件 ZeroMQ、序列化 技术 Protocol Buffer、远程服务调用技术 RPC 以及共享内存， 阐述并论证为何选

用上述技术，以及如何将上述技术应用到本论文的研究对象中。

第三章， 自动驾驶运行时通信系统需求分析。本章针对自动驾驶系统运行时 通信系统的特点， 导出本文自动驾驶运行时通信系统的功能性需求和非功能性 需求。

第四章， 自动驾驶运行时通信系统概要设计。本章结合架构图和活动图对自 动驾驶系统中自动驾驶运行时通信系统进行整体的设计， 结合流程图、时序图对 通信系统中各个模块的设计进行阐述。

第五章， 自动驾驶运行时通信系统详细设计与实现。本章结合类图与时序图 详细叙述本文如何实现自动驾驶运行时通信系统， 并着重阐述前文提出的共享 内存数据结构以及服务发现机制的不足之处的改进方法。

第六章， 测试与分析。本章通过功能测试来检查系统是否达到本文提出的功 能性需求； 通过数据大小、数据发布频率、通信参与者数量等多个维度对展开对 本文设计的通信系统进行性能测试， 将通信延迟等性能指标与同类产品进行比 较，并对测试结果进行详细分析。

第七章， 总结与展望。总结本文完成的研究内容并分析系统中存在的不足之

处以及对本系统的展望。

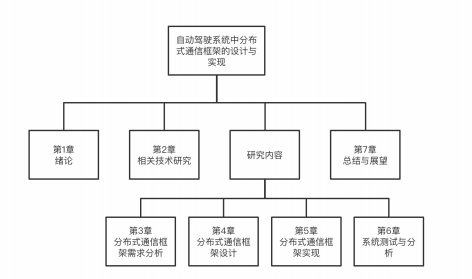


图 **1.3** 论文章节结构图

第 **2** 章 相 关 技 术

本文提出的自动驾驶运行时通信系统考虑到系统性能以及对成熟技术的复 用，在系统设计与实现中都考虑到了如何将优秀的开源技术应用到本系统中。本 系统使用了网络通信中间件 ZeroMQ、序列化技术 Protocol Buffer、远程服务调 用 (RPC) 开源库rest\_rpc，并且基于操作系统提供的共享内存相关系统调用实现 鲁棒性高和性能优秀的共享内存数据结构。本章将介绍上述的技术的特点并和 同类型的开源技术对比，给出本文选用上述开源技术的理由。

2.1 ZeroMQ

ZeroMQ(又称 ∅MQ 、0MQ 、ZMQ) 从其提供的库函数名看是基于操作系统 提供的套接字函数的二次封装， 但在实际使用中是一个高并发性能的网络通信 框架。 ZeroMQ 提供的套接字封装函数可以支持在多种场景下进行通信， 如线程 间、进程间、 TCP/IP 和广播等， 同时 ZeroMQ 提供的套接字可以提供多对多的 连接模式， 如发布-订阅、请求-应答等模式[[19](#_bookmark105)] 。ZeroMQ 会在后台线程使用异步 的方式去处理数据的 I/O 并且使用无锁化的数据结构来保证数据的一致性， 极大 降低了锁的争用而导致的系统延迟[[20](#_bookmark106)] 。

ZeroMQ 套接字与 TCP 套接字并不是同一个概念， 二者在传输方面的区别 如下:

1. ZeroMQ 传输的是消息， 并不是字节 (TCP) 或帧 (UDP)。消息在 ZeroMQ 中 的定义是一段固定长度的二进制数据块。

2. ZeroMQ 套接字在后台进行 I/O 操作， 无论接收消息还是发送消息， 消息都 会被暂存到本地的缓冲队列，缓冲队列的大小可以进行设置。

3. TCP 协议只能进行点对点的连接， ZeroMQ 套接字可以支持和多个套接字 的连接。在套接字类型符合要求是， ZeroMQ 套接字可以支持一对多、多 对一、多对多和一对一的连接， 因此 ZeroMQ 可以支持一个端点发送消息 到多个端点或一个端点接收多个端点的信息， 即扇出模型和扇入模型。图

2.1 直观地展示了这两个模型的具体结构：

第 2 章 相 关 技 术

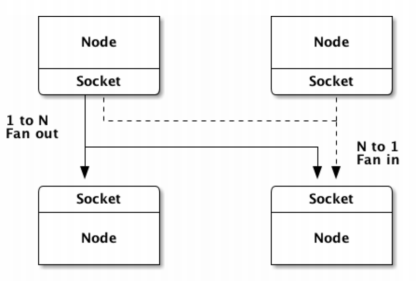


图 **2.1** **ZeroMQ** 的扇入和扇出模型

2.1.1 ZeroMQ 通信模式

ZeroMQ 共有四种通信模式： 请求-应答模式、发布-订阅模式、管道模式和 排他对接模式[[21](#_bookmark107)]，下文将对请求-应答模式和发布-订阅模式进行详细介绍。

1. 请求-应答模式

请求-应答模式将通信的双方角色划分为客户端和服务端， 用于远程过 程调用或任务分发。 ZeroMQ 提供了一共提供了四种套接字实现此模式的需 求： ZMQ\_REQ、ZMQ\_REP、ZMQ\_DEALER 和 ZMQ\_ROUTER。ZMQ\_REP 和 ZMQ\_REQ 配对使用时， 必须相应请求必须有相应的应答方， 否则请求发送后将 一直阻塞至相应的应答方出现位置。同时， 请求和应答这两个动作必须有先后顺 序， 不能出现先应答后请求的情况[[22](#_bookmark108)]。图 2.2 展示了最简单的一对一的请求-应 答模式的过程：

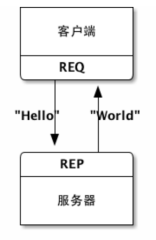


图 **2.2** 一对一的请求**-**应答模式过程

客户端创建 ZMQ\_REQ 套接字后， 填写请求的地址以及请求体的内容， 发送 后进入阻塞状态直到收到应答才能解除阻塞重新调度队列进行后续的作业； 服 务端创建 ZMQ\_REP 套接字并持续监听某一个地址， 当监听到某个请求消息到 来后对请求消息进行处理，处理完成后返回处理的结果给发出此请求的客户端。

ZMQ\_REQ 套接字和 ZMQ\_REP 套接字的连接关系既可以是一对一的， 也 可以是多对多的。当一个 ZMQ\_REQ 套接字连接多个 ZMQ\_REQ 套接字时， Ze- roMQ 会将客户端的多个请求负载均衡到不同的服务端， 如图 2.3 所示。当多个 ZMQ\_REQ 套接字连接至一个 ZMQ\_REP 套接字时， 服务端会公平地接收并处 理不同的请求。

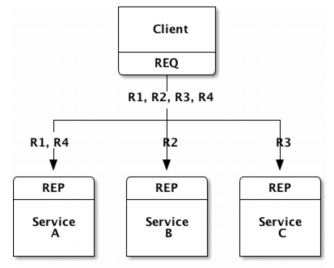


图 **2.3** 多请求负载均衡

考虑到本文提出的自动驾驶运行时通信系统的各通信节点并不是固定不变 的，节点之间的通信关系会随着时间动态地发生变化， 通信节点的绝对数量也会 发生变化。通信系统中可能出现一个客户端请求多个服务端， 也可能出现多个服 务端应答一个客户端。前一种情况下， 添加一个新的客户端是容易实现的， 只需 要将负载均衡模块的服务端地址列表告知新的客户端即可。但在后一种情况下， 添加一个新的服务端是不容易实现的， 需要将新的服务端地址通知到每一个存 在的客户端， 并且这些客户端可能分布在不同的网络中， 实现服务端与客户端的 地址同步就会产生大量的性能开销。

鉴于此类情况， ZeroMQ 提供了 ZMQ\_DEALER 套接字和 ZMQ\_ROUTER 套 接字来解决客户端服务端同步的问题。用户可通过将多个 ZMQ\_REQ 套接字连 接至 ZMQ\_ROUTER 套接字， 所有的请求会通过公平排队转发给 ZMQ\_DEALER

套接字， ZMQ\_DEALER 套接字通过负载均衡的方式将所有请求发送给与其相连 接的多个 ZMQ\_REP 套接字[[23](#_bookmark109)] 。ZeroMQ 称这种方法为请求-应答代理， 图 2.4 展示了使用代理的请求-应答模式的架构：

第 2 章 相 关 技 术

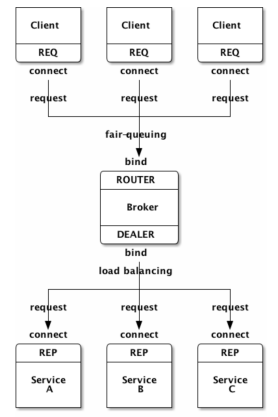


图 **2.4** 请求**-**应答代理架构

使用代理实现服务端与客户端动态变化的请求-应答模式的情况下， 所有的 客户端会在待发送的消息前插入空格后发送， ROUTER 为了标识多个客户端发 送的请求， 会将收到的所有请求加上 UUID(通用唯一识别码) 对不同的客户端请 求进行区分，图 2.5 展示了使用代理后 ZeroMQ 数据帧的变化：

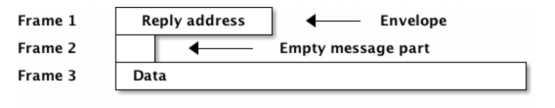


图 **2.5** 代理模式下的数据帧

其中帧 3 代表客户端实际的请求消息体， 帧 2 是客户端发出请求前加上的 空白消息， 帧 1 是 ROUTER 在收到客户端请求后加上的 UUID 。ROUTER 收到 请求消息并加上 UUID 后会交给 DEALER 通过负载均衡发送到不同的服务端进 行处理。服务端收到经过处理的请求消息后会直接取出帧 3 处理实际的请求体， 处理完成后会将帧 3 替换为处理后的响应消息返回给 DEALER，最后 ROUTER 会检验数据帧， 如果数据帧缺少 UUID 则直接丢弃本数据帧， 检验无误后根据 UUID 将响应消息返回给与 UUID 关联的客户端。

2. 发布-订阅模式

发布-订阅模式将通信的双方角色划分为发布者与订阅者。 ZeroMQ 提供 了四种套接字实现该模式的相关需求： ZMQ\_SUB 、ZMQ\_PUB 、ZMQ\_XSUB 和 ZMQ\_XPUB 。ZMQ\_SUB 与 ZMQ\_XSUB 套接字作为订阅者， ZMQ\_PUB 与 ZMQ\_XPUB 作为发布者。同请求应答模式一样， 不同角色的套接字连接关系既 可以是一对一的也可以是多对多的。图 2.6 是一个发布者连接多个订阅者的模

式：

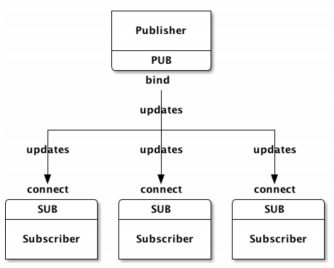


图 **2.6** 一对多发布**-**订阅模式

实际运用中， 可能会出现发布者先发送了数据但订阅者还未初始化完成的 情况， 一般会先由使用请求-应答模式发送请求， 等待订阅者应答后再发送数据。 本系统中， 会使用 Latch Message(锁消息) 的方式来保证这种情况下数据接收的 完整性。

使用发布-订阅模式同样面临发布者与订阅者动态变化的问题， 此问题的解 决思路同请求-应答模式相似。通过使用 ZMQ\_XSUB 与 ZMQ\_XPUB 套接字实 现发布-订阅代理， 这两个套接字既可以发送消息也可以接收消息。图 2.7 展示了 使用代理解决发布者与订阅者动态变化问题的方法。所有的发布者与订阅者不 直接连接而是连接到代理， 代理通过 ZMQ\_XSUB 获取发布的消息， 随后通过代 理将这些消息转发到 ZMQ\_XPUB，从而实现消息的间接发布。代理往往变动的 频率很低， 使用代理可以灵活应对分布式通信系统中频繁新增或减少的发布者 与订阅者。

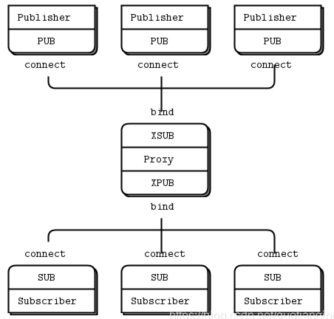


图 **2.7** 发布**-**订阅代理架构

2.1.2 ZeroMQ 性能表现

国外文章[[24](#_bookmark110)]比较了 MSMQ 、ActiveMQ 、RabbitMQ 和 ZeroMQ 每秒钟可以

发送和接受的消息数量，从表 2.1 可以看出 ZeroMQ 拥有比较明显的优势。

表 **2.1** 常见消息队列每秒收发消息数量对比**[**[**24**](#_bookmark110)**]**

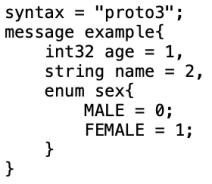
|  |  |  |  |
| --- | --- | --- | --- |
| ZeroMQ | RabbitMQ | ActiveMQ | MSMQ |
| 接收 88699  发送 242659 | 12281  12278 | 6453  6452 | 7021  11406 |

通过本小节与前小节对 ZeroMQ 的介绍， 可以得出 ZeroMQ 使用起来非常方 便， 并且其内部实现的代理可以灵活应对分布式通信系统动态变化的通信节点， 同时其性能也较其他同类产品有比较明显的优势， 最终本文统选用 ZeroMQ 作 为网络通信的工具。

2.2 序列化技术

序列化技术包括序列化和反序列化两部分，序列化是将某一种格式 (如 C++ 结构体、 Java 对象等) 进行一系列的处理并最终生成二进制序列的过程， 反序列 化则是序列化的相反过程[[25](#_bookmark111)] 。本文使用序列化技术主要有以下几点理由：

1. 本文研究的通信系统是具备分布式通信能力的， 这代表不同网络中的通信 节点需要使用网络通信来传输相关的信息。而基于 TCP/IP 的网络通信不 支持直接传输某个编程语言中的类或结构体， 必须要将相关数据类型序列

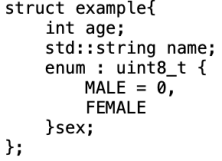
化成二进制序列才可以在网络中进行传输。而同一台物理机的进程间通信 也需要将数据类型序列化成二进制才能被共享内存或 Domain socket 进行 传输。

2. 序列化技术可以将某种语言的数据类型转换成与语言和平台无关的二进制 序列， 这个功能代表了使用序列化来进行数据的传输可以跨平台跨语言。 只要在序列化技术支持的平台和语言列表中进行开发就不需要为不同平台 和语言进行额外的适配工作，大大简化了开发工作量。

3. 序列化技术可以大大减少原数据的大小。从网络通信方面来看， 减少通信 需要的带宽并加快了数据传输的速度。从基于共享内存的进程间通信来看， 减少了需要向系统申请的共享内存空间。这种优势在传输小数据时不是很 明显， 但在自动驾驶中摄像机的图像数据大小会达到 MB 级别， 在这种情 况下序列化对原始数据的压缩就显得尤为重要。

本文选用 Google 开源的 Protocol Buffer 作为序列化工具。 Protocol Buffer 是一种 高性能、易于使用且扩展性较强的数据存储格式， 支持 Python 、C++ 和Java 语 言[[26](#_bookmark112)] 。

使用 Protocol Buffer 最重要的一步就是编写 IDL(交互式数据语言) 文件， 通 常 IDL 的编写是依据使用者已有的数据类型进行对照进行的， 即参照已有的 C+ 结构体或 Java 对象的内部数据成员进行编写。图 2.8 展示了如何根据 C++ 结构 体编写 IDL：



(a) C++ 结构体 (b) 对应 IDL

图 **2.8** 根据 **C++** 结构体编写对应 **IDL**

通过比较图 2.8(a) 与图 2.8(b) 可以看出， IDL 的编写基本和 C++ 等主流语言 保持了一致规则， 学习 IDL 编写的成本并不大。 IDL 提供了一系列的关键字供使 用者实现更为复杂的序列化规则， 不止包括定义一个数据类型， 还可以定义服务

的接口等高级功能，本文就不再赘述了。

常见的序列化技术并不只有 Protocol Buffer 一种， XML 和 JSON 也是应用非 常广泛的序列化技术方案， 但本文选用 Protocol Buffer 而不使用其他序列化技术

主要有性能上和序列化技术本身的数据表达能力的考虑。表 2.2 展示了 Protocol

Buffer 、XML 以及 JSON 表达相同一个数据类型的格式：

表 **2.2** 三种序列化技术描述数据的格式对比**[**[**27**](#_bookmark113)**]**

序列化技术

格式

Protocol Buffer message example{int32 age = 1;string name = 2;}

JSON {”age” : 1, ”name” : ”example”}

XML <example><age>1</age><name>example</name></example>

从表中可以看出 Protocol Buffer 相较于其他两种序列化方案， 它对描述一个 数据类型的描述格式非常直观， 每一个成员变量的类型在编写 IDL 时就直接确 定。而 XML 和 JSON 对一个数据类型描述的格式并不能让使用者直观地看出每 一个变量的具体类型， 当数据类型本身非常复杂时， 他们的描述格式阅读起来更

为困难。

除了易于阅读的优点， Protocol Buffer 在性能方面也较 XML 和 JSON 有明 显的优势。表 2.3 列出了这三种序列化方案序列化同一份数据的耗时、反序列化 同一份数据的耗时以及序列化后的字节大小[[28](#_bookmark114)] ：

表 **2.3** 三种序列化技术性能对比

|  |  |  |  |
| --- | --- | --- | --- |
| 序列化技术 | 序列化耗时 (ms) | 序列化后字节数 (Byte) | 反序列化耗时 (ms) |
| Protocol Buffer | 241 | 117 | 190 |
| JSON | 871 | 255 | 786 |
| XML | 1383 | 474 | 1781 |

本小节阐述了为什么在本文的自动驾驶运行时通信系统中需要使用序列化 技术， 通过对比 Protocol Buffer、XML 以及 JSON 描述数据类型格式的可阅读性， 以及列出实际数据对比了三种序列化技术的性能参数， 最终本文选用 Protocol Buffer 作为序列化工具。

2.3 远程服务调用

远程服务调用 (Remote Proceduce Calls,RPC) 是由Nelson 于 1982 年提出的一 种跨进程消息同步的一种通信机制[[29](#_bookmark115)] 。RPC 屏蔽了底层的通信细节和传输错误 并为分布式系统提供了灵活便捷的通信功能。 RPC 的具体实现包括 RPC-JSON 和 RPC-XML[[30](#_bookmark116)-[31](#_bookmark117)]，此外还有许多不同类型的实现， 但 RPC 的通信方法是类似 的，不同的实现方法仅在传输的数据格式本身有差异。

RPC 解决的问题是实现分布式系统中各个通信单元调用不同服务的过程像 本地调用一样方便。 RPC 为了保证调用远程的服务的透明性， 也引入了代理的 概念[[32](#_bookmark118)]。图 2.9 是一次 RPC 调用的流程图：

第 2 章 相 关 技 术

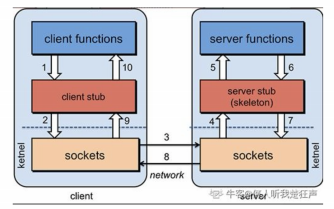


图 **2.9** 发布**-**订阅代理架构

从图 2.9 可以看出， RPC 调用过程主要分为以下几个步骤：

1. 客户端发起调用之前， 首先要确定需要调用什么服务。服务端为了将自身 提供的不同服务做区分， 会将每一个提供的服务加上标识 (Call ID)，以此 让客户端能够指定调用某一个服务。客户端确定调用的服务后， 需要将参 数一并发送给本地的 RPC 代理。在上一节提到过， 网络传输的内容只能是 二进制的序列， 所以客户端如果需要传输字符串之外的参数必须要将参数 序列化为二进制序列。

2. 客户端的 RPC 代理收到 Call ID 以及相关参数后， 找到提供此服务的服务 端并将调用信息以网络的形式传输给服务端。

3. 服务端的 RPC 代理收到服务调用请求后， 首先会对调用信息反序列化取得 Call ID 以及参数， 根据 Call ID 找到本地对应的服务结合参数进行处理， 处 理完成后将结果序列化并通过 RPC 代理返回客户端代理。

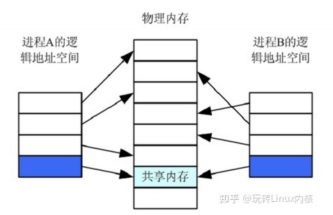
4. 客户端通过代理取得本次服务调用的结果， 反序列化完成后就可以继续之 后的作业。

RPC 解决的问题契合了本文提出系统的关键问题， 即如何保证分布式通信 系统在动态场景下所有通信节点都有方法获取最新的网络拓扑。本文基于 RPC 的思想以及实际提供的技术方案并且融合了注册中心的概念， 在主节点中向通 信节点提供服务调用， 包括查询网络拓扑以及根据新增节点发送的 RPC 请求向 所有有关的节点发送网络拓扑信息。本文已经使用了 Protocol Buffer 作为序列化 工具，能够在使用 RPC 请求的中需用该工具。

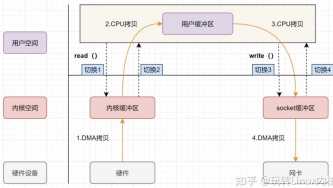
2.4 共享内存

共享内存是操作系统提供的进程间通信方式之一， 也是最重要的一种进程 间通信方式， 其通信效率相较于其他通信方式有明显的优势。共享内存的原理是

将两个或多个独立的进程的一段虚拟地址映射到同一块物理地址， 这些进程共 同使用物理地址的某一个区域。进程对共享内存的修改直接作用在物理内存上， 不会发生用户态与内核态之间的数据拷贝， 所以共享内存并没有多余的数据拷 贝， 称零拷贝 (零拷贝并不代表一次数据拷贝都没有发生， 零的概念是相对于拷 贝次数更多的通信方式而言)。图 2.10 对比了共享内存与 TCP socket 读写数据过 程的对比：



(a) C++ 共享内存



(b) TCP socket

图 **2.10** 共享内存与 **TCP** **socket** 读写过程对比

从图 2. 10(a) 和图 2. 10(b) 可以看出， 使用共享内存读写数据过程中只会发生 两次数据拷贝， 一次是将数据写入共享内存中一次是从共享内存中读取数据； 而 使用 TCP socket 在读写数据过程中会发生四次数据拷贝。使用进程间通信传输 本身就比较小的数据时， 拷贝次数对于系统性能及延迟的影响可以忽略不计， 但 传输大数据时会明显造成系统通信延迟的上升以及性能的下降。不仅如此， 在本 系统内会发生一个发布者向多个订阅者发送消息， 这会造成同一份数据被拷贝 数十次。表 2.4 总结了常见的自动驾驶通信系统使用进程间通信传输一次数据期 间发生的拷贝次数。表 2.5 是常用几种进程间通信方式吞吐量性能对比。

表 **2.4** 自动驾驶通信系统数据拷贝次数总结**[**[**33**](#_bookmark119)**]**

|  |  |  |
| --- | --- | --- |
| 进程间通信方法 | 通信系统 | 拷贝次数 |
|  | ROS1 | 1 + 3 × sub |
| Socket  共享内存 | ROS2  LCM [[11](#_bookmark97)]  Ach [[34](#_bookmark120)]  ETHZ-ASL | 3 + 3 × sub  1 + 3 × sub  1 + 1 × sub  1 + 1 × sub |

注： sub 代表同一个数据 (即话题) 被订阅的次数。

表 **2.5** 不同进程间通信方式吞吐量对比

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 数据量  通信方式 | | 128×1000 512×1000 2048×1000 4096×1000 | | | |
| SHM | 774MB/s | | 2072MB/s | 5391MB/s | 7480MB/s |
| TCP socket | 249MB/s | | 760MB/s | 1971MB/s | 2815MB/s |
| PIPE | 153MB/s | | 515MB/s | 1670MB/s | 2287MB/s |
| FIFO | 144MB/s | | 481MB/s | 1566MB/s | 2343MB/s |

注：表格中数据量单位为 (Byte × 次数)；SHM 为共享内存， PIPE 为匿名管道， FIFO 为命名管道。吞吐量测试均未对临界数据进行加锁保护。不同的测试环境数据可 能出现不同。

从表 2.4 和 2.5 可以看出， 共享内存的吞吐量相较于其他进程间通信方式在 吞吐量和减少消息拷贝方面有明显的优势。虽然共享内存的优点很多， 但也有不 可忽视的缺点： 共享内存没有提供数据同步的功能， 这代表如果要使用共享内存 来进行数据的读写需要使用者自身保证对临界资源的保护。本文选用共享内存 作为进程间通信手段的同时， 通过重新设计共享内存数据结构改进了第一章提 到的现有共享内存数据结构的缺陷，

2.5 本章小结

本章介绍了 ZeroMQ 和序列化技术的相关理论知识和应用， 并结合本文系 统的特点选用特定的技术框架， 并通过与选用技术框架同类型产品进行性能分 析来论述本文为什么要选用这些技术框架。此外着重对进程间通信进行深入介 绍并分析， 指出使用 TCP 或 UDP 实现进程间通信的缺点以及使用共享内存的优 点。

第 **3** 章 自动驾驶运行时通信系统需求分析

本章将结合自动驾驶系统内部与外部数据通信的模式与特点对本文的自动 驾驶运行时通信系统进行需求分析， 并在需求分析的基础上提出本系统的功能 需求与非功能性需求。

3.1 业务需求分析

本节阐述自动驾驶系统在单车内的数据通信以及与云端平台通信的模式与 特点，作为自动驾驶运行时通信系统的业务需求分析的出发点。

3.1.1 自动驾驶系统通信特点分析

1. 单车内通信特点

自动驾驶系统单车内的通信是指部署在车端计算平台各个功能模块相互通 信的过程。通常自动驾驶系统会将整个系统按照功能划分成若干模块， 如传感 器、规划、决策、控制、感知及地图引擎等。这些模块之间会进行接收发送相关 数据从而控制整个自动驾驶系统的性能表现。图 3.1 是自动驾驶系统单车内各功

能模块消息传递的拓扑网络。

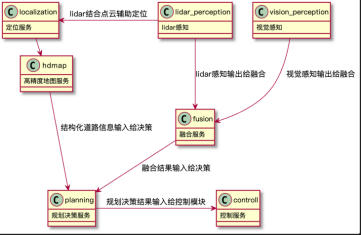


图 **3.1** 自动驾驶单车内模块通信拓扑图

从图 3.1 可以分析出， 模块之间通信主要是依次从上游原始数据传递到算法 处理模块最终由下游的模块直接使用并输出车辆的行为结果， 此外还存在两个 模块互相发送接收信息的模式。当系统内参与通信模块数量持续增加时， 通信的 拓扑网络将变得复杂。针对自动驾驶系统中通信网络拓扑复杂的问题， 采取基于 话题的发布-订阅通信模式是解决此问题的一个较好方法。

同时注意到一个模块发送的信息可以被多个模块接收， 这代表同一份数据

需要被传输到不同的模块中， 结合第二章论述的利用 TCP/IP 通信方式的特点， 同一份数据需要被拷贝多次到不同的模块中， 而单模块传输数据本身也需要多

次拷贝，如图 3.2 所示：

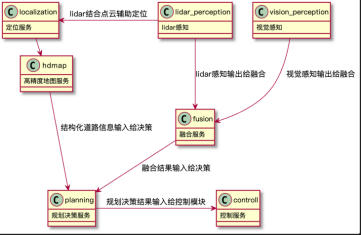


图 **3.2** 自动驾驶单车内模块通信拓扑图

从图中可以看出当传输摄像机拍摄的图像等较大的数据时， 数据的多次额 外拷贝就会成为通信系统本身的性能瓶颈。

2. 单车与云端平台通信特点

目前自动驾驶系统通信并不局限于车内各功能模块的通信，而是正迈向 SOA 架构风格。即自动驾驶系统运行时需要的元数据、配置文件甚至路线的规划都逐 渐从本地保存转变为与云端服务器进行数据的交互。大部分自动驾驶系统是基 于高精度地图来提供结构化道路信息供算法使用。高精度地图是一种精度更高， 纬度更多， 兼有动态数据和静态数据的电子地图。高精度地图的生产的原则之一 是反映实际道路的形态， 当道路形态改变时， 高精度地图将不再可靠。在这种技 术方案下， 自动驾驶系统需要更新最新的高精度地图数据且频率并不低， 所以自 动驾驶系统需要频繁地去更新地图数据。针对此问题， 自动驾驶系统会在每次运 行时都向云端的数据平台下载最新的地图数据， 流程如图 3.3 所示： 从此实际例 子可以看出， 自动驾驶系统通信已经不能局限于车内本地的通信， 需要更多地考 虑与云端或更多终端进行通信的能力。更早地布局自动驾驶车辆与不同终端的 通信能力可以实时地将数据同步到车内实现更好的自动驾驶性能表现， 同时未 来也可以更快速地接入智能路网等应用。

综合本节阐述的自动驾驶单车内通信及单车与云端平台通信特点， 可以得 出自动驾驶系统对通信系统的要求是多样的。

在业务形态方面， 传统的通信系统通常只提供单一的通信模式， 限制了用户 根据使用场景灵活调整通信模式的需求。而自动驾驶要求通信系统具备多样化 的通信模式， 既要求通信系统具备进程间通信能力， 还要求其具备网络通信能

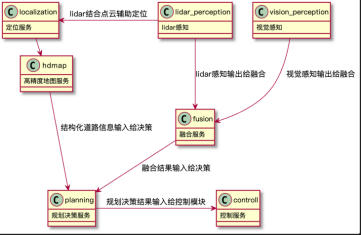


图 **3.3** 自动驾驶单车内模块通信拓扑图

力，甚至要求具备进程内通信能力。

3.1.2 业务流程描述

虽然自动驾驶运行时通信系统业务流程复杂， 但从更抽象的角度来看， 业务 流程可大致分为如下四步：

1. 申请通信： 通信需求方提交通信目标及通信方式到通信系统等待系统返回 申请结果。

2. 通信校验： 通信系统收到通信需求后， 校验申请方的申请是否合法， 若合 法则进入通信匹配流程，否则直接终止流程。

3. 通信匹配： 通信系统寻找通信需求方的通信目标， 若成功匹配到通信目标 则进入到建立通信信道流程， 否则进行等待通信目标出现直至超时终止流 程。

4. 建立通信信道：通信系统向通信双方通知通信信道建立，双方开始通信。 图 3.4 是对建立通信业务流程的描述说明。

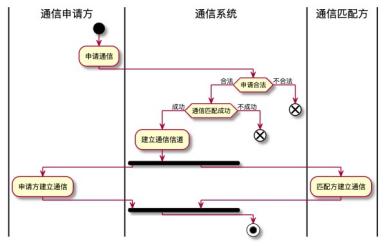


图 **3.4** 建立通信业务流程图

3.2 功能性需求

根据上一节对自动驾驶系统通信的模式与特点的概述与分析， 本文实现的 自动驾驶运行时通信系统的功能需求变得清晰， 本节将导出本系统的功能性需 求，并通过用例图对各个需求进行详细分析。

3.2.1 通信单元管理

注册通信单元功能是通信系统基本功能， 自动驾驶系统的大部分通信功能 都依托于数据发布者 (publisher) 发布数据和数据订阅者 (subscriber) 订阅数据对 通信单元实现准确、完备的管理不仅可以实现用户对管理通信单元多样的需求， 也是整个通信系统的基础。通信单元管理包括注册通信单元、删除通信单元、修 改通信单元通信域和查询通信单元信息。

注册通信单元， 分为注册数据发布者与注册数据订阅者， 而数据发布者与数 据订阅者又分为本机域通信和全域通信。本机域通信规定通信单元只能在单机 内通信；全域通信规定通信单元可以与任何通信单元进行通信。

删除通信单元， 分为删除发布者与删除订阅者， 其中删除订阅者分为立即删 除与惰性删除两种模式。立即删除模式下， 订阅者会被直接删除， 消息队列未接 收的数据将被直接丢弃； 惰性删除模式下， 会首先将消息队列中未接收的数据保 存再删除订阅者。

修改通信单元通信域， 分为修改发布者通信域与修改订阅者通信域， 用户可 以根据需要修改已经创建的发布者或订阅者的通信域。

查询通信单元信息， 分为查询发布者信息与查询订阅者信息， 查询信息可以 根据某一话题名称查询， 也可以查询所有存在的信息。同样地， 用户可以根据实 际需要查询本机内和全域内的通信单元信息。

通信单元管理用例图如图 3.5 所示：

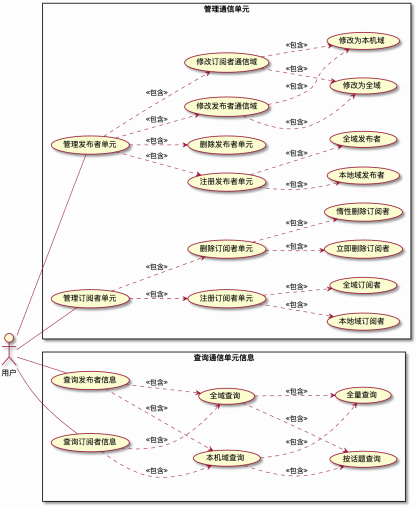


图 **3.5** 通信单元管理用例图

考虑到篇幅原因， 本节仅以注册订阅者单元用例和删除订阅者单元用例为

例说明，如表 3.1 和 3.2 所示：

表 **3.1** 注册订阅者单元用例说明

|  |  |
| --- | --- |
| 用例名称 | 注册订阅者单元 |
| 行为角色 | 数据订阅者 |
| 用例说明 | 注册订阅者单元，获得订阅数据能力； |
| 前置条件 | 发布者正确设置注册订阅者需要的参数； |
| 后置条件 | 注册订阅者单元成功，返回成功状态码；  1. 设置订阅者订阅话题名称； |
| 基本流 | 2. 设置订阅者发布数据的数据类型；  3. 设置订阅者订阅数据队列长度；  4. 设置订阅者通信域；  5. 提交参数到系统，注册成功，系统返回成功状态码； 1a. 数据订阅者未设置订阅话题名称，注册失败； |
| 异常流 | 2a. 数据订阅者未设置数据类型，注册失败；  3a. 订阅者设置队列长度为非法大小，如负数等，注册失败； 4a. 订阅者未设置通信域，注册失败； |

表 **3.2** 删除订阅者单元用例说明

|  |  |
| --- | --- |
| 用例名称 | 删除订阅者单元 |
| 行为角色 | 数据订阅者 |
| 用例说明 | 删除订阅者单元，停止订阅数据； |
| 前置条件 | 发布者正确设置删除发布者需要的参数； |
| 后置条件 | 删除订阅者单元成功，返回成功状态码； |
| 基本流 | 1. 设置删除模式；  2. 提交参数到系统，删除成功，系统返回成功状态码； |
| 异常流 | 1a. 订阅者未设置删除模式，删除失败，系统返回失败状态码； |

3.2.2 服务管理

在自动驾驶系统中， 不仅仅需要给予发布-订阅模式的异步通信功能， 还需 要具备同步通信功能。当用户注册了发布者和订阅者后， 仅获得了最基本的异步 消息收发能力， 而服务的概念可以为用户提供更为高级的同步消息请求能力， 更 契合自动驾驶系统 SOA 架构风格。在 3.1 节提到自动驾驶系统需要向云端数据 平台请求高精度地图等数据， 在这种情况下， 车端请求数据的动作的时间是不确 定的，需要数据平台提供请求数据的服务，等待车端发起数据请求。

根据自动驾驶系统通信特点及用户的实际需要， 服务管理包括创建服务、删 除服务、修改服务通信域和查询服务信息。

创建服务， 分为创建服务端与创建客户端， 同通信单元的发布者和订阅者一 样，服务端与客户端可以自由选择通信域。

删除服务， 分为删除服务端与删除客户端， 与通信单元不同的是， 删除模式 只有立即删除一种模式。

修改服务通信域， 分为修改服务端通信域与修改客户端通信域， 用户可以根 据需要更改已经创建的服务客户端或服务端的通信域， 改变提供服务和请求服 务的通信范围。

查询服务信息仅提供查询服务端相关信息， 用户最关心的是系统中存在哪 些服务端及其信息。查询可以通过服务端名称进行查询， 也可以进行全量查询， 并且用户可根据需要确定在哪种通信域内进行查询。服务管理用例图如图 3.6 所 示。

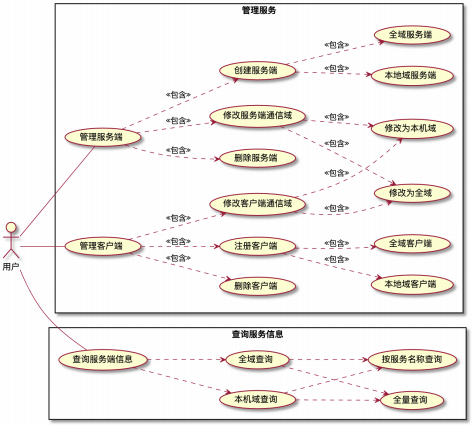


图 **3.6** 服务管理用例图

本节以创建服务端和查询服务信息用例为例进行说明， 如图表 3.3 和 3.4 所

示：

表 **3.3** 创建服务端用例说明

用例名称 创建服务端

行为角色 服务提供者

用例说明 创建服务，供服务使用者请求使用；

前置条件 提供者正确设置相关参数；

后置条件 创建服务成功，系统返回状态码；

1. 设置服务名称；

2. 设置服务通信域；

基本流

3. 设置服务接收请求的数据类型；

4. 提交参数到系统，创建成功，系统返回成功状态码； 1a. 服务提供者未设置服务名称，注册失败；

异常流 2a. 服务提供者未设置服务通信域，注册失败；

3a. 服务提供者未设置接收请求的数据类型，注册失败；

表 **3.4** 查询服务信息用例说明

|  |  |
| --- | --- |
| 用例名称 | 查询服务信息 |
| 行为角色 | 服务使用者 |
| 用例说明 | 获取系统内服务端信息； |
| 前置条件 | 服务使用者正确设置相关参数； |
| 后置条件 | 删除发布者单元成功，返回成功状态码； |
| 基本流 | 1. 设置查询域；  2. 设置查询模式； |
| 异常流 | 1a. 服务使用者未设置查询域，查询失败； 2a. 服务使用者未设置查询模式，查询失败； |

3.2.3 通信方式自适应选择

通信方式是自动驾驶运行时通信系统核心功能之一， 根据实际情况合理地 选择通信方式是降低系统通信延迟、提高消息吞吐量的关键。为了避免 ROS 系 统单一通信方式对通信性能的影响， 并根据 3.1 节对自动驾驶通信特点的分析， 本系统需要支持网络通信、进程间通信及进程内通信。本系统在支持多通信方式 的同时， 需要根据通信双方是否处于同一台物理机和是否处于同一个进程自动 选择网络通信、进程间通信或进程内通信方式并协助通信双方按照通信方式建 立通信链路。

3.2.4 调度策略管理

自动驾驶对通信系统需求不仅是对通信功能的需求， 还包含对任务调度方 式的需求。同通信性能本身对整个通信系统性能表现的影响程度， 调度策略的选

择一样重要。

在自动驾驶系统中， 会存在多种传感器如摄像头、LiDAR(激光雷达)、IMU(惯 性测量单元) 等。自动驾驶对传感器发送数据的频率有着较为严格的要求， 通常 会限定某个传感器在某个时间段内的平均频率为稳定在系统可接受范围内。针 对此类定频任务的实际需求， 采用基于时间驱动的调度策略即每秒固定调度次

数是较好的解决方法； 传感器作为自动驾驶最上游的数据， 最终要经过处理供下 游业务模块使用， 而大多下游模块并不需要像传感器一样要求较为稳定的频率 发送信息。此类业务模块只需要关注何时收到了数据， 并将数据运用到实际业务 中， 所以基于数据驱动的调度策适合该类任务的调度需求。由此得出， 系统需要 提供调度策略管理功能给用户， 调度策略管理包括设置调度策略及更改调度策 略。

本节以设置调度策略模式用例为例进行说明，如表 3.6 所示：

表 **3.5** 设置基于时间的调度策略用例说明

|  |  |  |
| --- | --- | --- |
| 用例名称 | 设置基于时间的调度策略 |  |
| 行为角色 用例说明 前置条件 后置条件  基本流  异常流 | 发布者或订阅者  该用例实现对任务调度策略的设置；  已经注册通信单元；  设置调度策略成功，系统返回成功状态码 1. 设置调度策略为基于时间;  2. 设置调度频率;  3. 设置成功，调度策略生效  1a. 未设置调度策略，设置失败；  2a. 未设置调度频率或调度频率为非法频率， | 设置失败； |

3.3 非功能性需求

自动驾驶运行时通信系统在满足功能性需求的基础之上， 还需要根据本系 统实际应用场景满足非功能性需求。本系统应用在算法模块数量庞大的自动驾 驶系统中，非功能需求需要满足实时性、安全性、可靠性和可扩展性。

3.3.1 实时性

本系统的首要功能是向自动驾驶系统中各算法模块提供通信功能， 自动驾 驶系统本身是对实时性要求非常高的系统， 本系统需要保证消息传输延迟保证 在 10ms 之内并且不能出现较大波动。

3.3.2 安全性

本系统的应用场景是在实际道路上运行的车辆， 车辆的所有操作均由自动 驾驶系统操作线控底盘完成。此场景对安全性的要求极高。而本系统将承担车内 各模块的通信工作，系统需要有能力判断接收到的消息收否被人为篡改。

3.3.3 可靠性

自动驾驶系统是个长时间运行的系统， 本系统需要在长时间运行的实际场 景下保证内存、 CPU 等系统资源的占用都稳定在合理的范围内

3.3.4 可扩展性

本系统业务复杂， 主要涉及不同方式的通信业务及复杂的用户接口设计， 所 以代码结构也同样复杂。所以在实际的代码开发过程中需要尽可能将代码抽象 为一个个独立的模块， 模块之间通过预留的接口串联。同时， 本系统并没有使用 较为流行的通信框架如 DDS，模块化的设计思路可以快速地将更多的通信方式 集成到系统中。

3.4 本章小结

本章对自动驾驶运行时通信系统的应用场景进行深入分析， 并从中导出功 能性需求和非功能性需求。一方面， 功能性需求详细分析了本系统核心模块需要 具备的功能； 另一方面， 实际应用的角度出发分析了本系统需要保证的非功能性 需求。

第 **4** 章 自动驾驶运行时通信系统概要设计

本章基于需求分析的结果和软件开发规范， 对自动驾驶运行时通信系统进 行概要设计。系统的概要设计将从系统架构、通信组网方案、模块依赖关系、功 能架构及功能模块五个维度进行展开详细论述。基于本章的论述， 可以从总体上 了解自动驾驶运行时通信系统的设计思路，为后续详细实现奠定基础。

4.1 系统总体设计

系统总体设计导出了系统具体的层次架构和技术方案， 且明确了系统各模 块关系和系统功能。

4.1.1 系统架构设计

本文提出的自动驾驶运行时通信系统在采用分层次的系统架构设计方法。从 开发角度看， 分层次的架构设计不仅可以使系统的众多功能模块较为清晰地分 布在不同层次上， 还可以使分布在不同层次的功能模块之间减少耦合性。从用户 角度看， 分层次的系统架构中层次越高软件的抽象表达能力越强， 而层次越低抽 象表达能力就越低， 这使得用户无需关心系统底层复杂的实现， 只需要关心系统 最上层提供的 API(应用程序接口) 来实现自身业务。

如图 4.1 所示， 本文将自动驾驶运行时通信系统架构分为应用层、任务层、 调度层、数据通信层和通信抽象层。从图 4.1 可以看出， 在第三章需求分析中导 出的用户需求全部位于应用层， 整个系统将为应用层提供高度抽象的 API，通信 单元间通信链路的建立、自动驾驶运行时系统网络拓扑的同步等实现全部隐藏 在系统底层。下文按照自顶向下的顺序介绍系统的五个层次：

1. 应用层： 应用层的功能是根据需求分析导出的通信单元管理、服务管理、调 度策略管理和通信模式管理功能需求， 直接提供 API 供用户使用， 如创建 发布者、删除发布者等。

2. 任务层： 任务层将用户在应用层创建的所有通信单元和服务封装成一个整 体作为任务供调度层调度。

3. 调度层： 调度层将对任务层的每一个任务按照其设置的调度策略进行调度。 当调度策略被设置为基于数据驱动时， 只有当任务中的某一个通信单元或 收到数据时任务才会运行； 基于时间驱动时， 任务将按照设置的频率被周 期性调度运行。

4. 数据通信层： 数据通信层中会保存所有由用户通过 API 创建的通信单元和

服务， 即发布者、订阅者、服务的服务端和服务的客户端的实体， 但通信 链路并没有实际产生。该层的功能是将数据传输的操作抽象化， 调用通信 抽象层功能完成实际的数据传输操作。

5. 通信抽象层： 该层是整个自动驾驶运行时通信系统的核心层， 所有的通信 链路的建立、数据的传输、通信网络拓扑同步及自适应通信方式都在该层 实现。通信方式和服务发现紧密相关， 通信方式的选择由服务发现提供的 网络拓扑决定， 服务发现根据通信方式更新网络拓扑即通信单元列表和服

务列表。

外部工具包含本文系统使用的开源框架 ZeroMQ 、rest\_rpc 、protobuf。

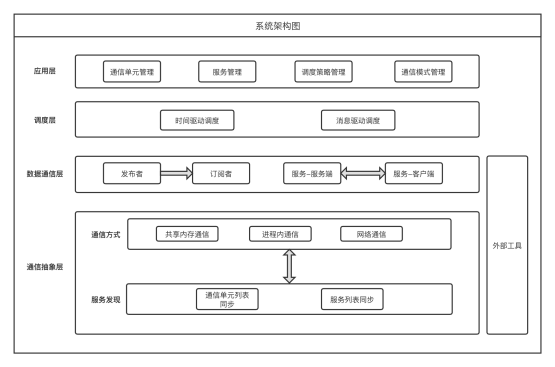


图 **4.1** 自动驾驶运行时通信系统架构图

4.1.2 自动驾驶运行时通信组网设计

针对通信系统网络拓扑动态变化的问题， 如何使各分布在不同网络中的通信 节点实时获取最新网络拓扑并相互发现是最为关键的问题， 即服务发现问题。本 文对服务发现问题的解决方法是实现一个弱中心节点的方案， 所有任务通过自 身的服务发现模块对中心节点发起 RPC 请求获取网络拓扑并完成通信的连接。 弱中心节点的含义是中心节点只提供网络拓扑信息的查询与更新， 并不参与实 际的通信。当任务内的通信单元或服务建立通信后消息通信是点对点的， 不需要 中心节点进行代理通信。

本文提出的自动驾驶运行时通信系统对于同一个局域网下的多个物理机不 处于同一个网络中。本系统提供的通信域对通信范围作出了划分， 本机域指的是

同一台物理机内的通信， 全域指的是在网络内所有物理机之间的通信。本系统组

网设计如图 4.2 所示。

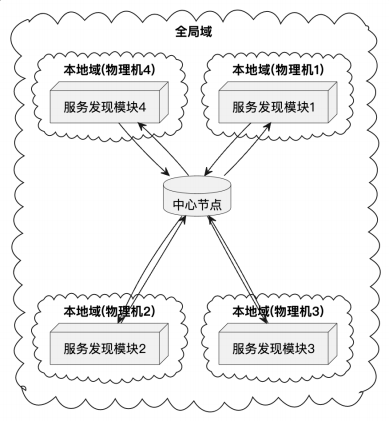


图 **4.2** 通信系统组网设计图

4.1.3 系统功能模块设计

根据需求分析结果及系统架构， 本文将系统分为如下七个模块： 通信单元模 块、服务模块、任务模块、调度模块、通信传输模块、通信抽象模块及服务发现 模块。本文根据业务与软件设计两个维度将系统模块层级分为一级模块与二级 子模块， 其中通信单元模块、任务模块、通信传输模块与通信抽象模块被分为多 个二级子模块， 系统功能模块设计如图 4.3 所示。根据功能模块图所示， 本文设 计的自动驾驶运行时通信系统各模块实现功能如下：

1. 通信单元模块： 包含发布者模块与订阅者模块两个二级子模块， 该模块实 现基于发布-订阅通信模式功能， 同时向用户提供接口。发布者模块实现发 布消息和管理发布者功能；订阅者模块实现订阅消息和管理订阅者功能。

2. 服务模块： 包含服务服务端和服务客户端两个二级子模块， 该模块实现基 于请求-响应的通信模式功能， 同时向用户提供接口。服务服务端实现注册 服务和管理服务端功能；服务客户端实现请求服务和管理客户端功能。

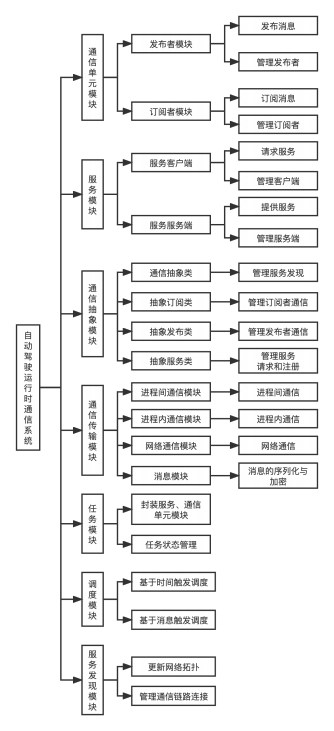


图 **4.3** 自动驾驶运行时通信系统功能模块图

3. 通信抽象模块： 包含通信抽象类、抽象订阅类、抽象发布类和抽象服务类 四个二级子模块， 该模块将通信单元模块和服务模块所有的操作进行统一 管理并且负责生成服务发现请求参数供服务发现模块完成服务发现功能。 抽象订阅类根据用户对通信单元模块与服务模块的管理动态创建抽象订阅 类、抽象发布类和抽象服务类， 并生成服务发现请求参数至服务发现模块；

抽象订阅类控制每个订阅者的通信链路， 并通过通信传输模块监听通信链 路中的消息； 抽象发布类控制每个发布者的通信链路， 并将待发布消息通 过通信传输模块发送消息； 抽象服务类控制所有服务端与客户端的注册服 务和请求服务操作。

4. 通信传输模块： 包含进程间通信模块、进程内通信模块、网络通信模块与 消息模块四个字模块， 该模块是通信系统发布-订阅通信模式的实际传输模 块， 发布者发布消息与订阅者订阅消息全部由该模块负责序列化、加密与 传输。进程间通信模块通过共享内存方式传输消息； 进程内通信模块通过 指针方式传输信息； 网络通信模块通过 ZeroMQ 传输消息； 消息模块负责 将进程间通信模块与网络通信模块待传输消息进行序列化与加密处理。

5. 任务模块： 该模块将所有的发布者、订阅者、服务服务端和服务客户端及 用户的逻辑代码封装为一个统一的任务， 同时提供对任务状态的管理功能。

6. 调度模块： 该模块将用户的任务进行调度， 调度策略分为基于消息触发的 调度策略和基于时间的调度策略。

7. 服务发现模块： 该模块一方面接收本进程通信抽象类的服务发现参数向中 心节点寻找通信匹配方并控制通信双方的通信链路建立； 另一方面接收中 心节点的更新通知，更改本进程内通信方的通信链路。

4.2 通信单元模块概要设计

自动驾驶运行时通信系统基于发布-订阅的异步通信功能依靠通信单元模块 完成， 通信单元模块由发布者模块与订阅者模块两个子模块共同完成此功能。通 信单元模块是最靠近用户的模块之一， 用户将直接使用该模块提供的接口管理 发布者与订阅者。该模块的设计思想是尽可能屏蔽系统最底层的复杂实现而提 供高度抽象且易用的接口给用户。

4.2.1 订阅者模块概要设计

订阅者模块用于实现基于发布-订阅模式异步通信中订阅者的基础功能， 并 向用户提供管理订阅者和订阅消息的功能接口。订阅者模块的管理接口包括创 建订阅者、删除订阅者与修改订阅者通信域， 订阅消息由订阅者模块内部通过消 息队列完成，用户无需调用显式接口完成订阅消息的操作。

订阅者模块内部保存订阅者配置文件， 该文件决定了订阅者需要订阅的话 题以及通信域， 配置文件如表 4.1 所示。配置文件中， Topic 与 Domain 需要由用 户显式给定， 字段不可为空； QueueSize 系统默认为 100，用户可以自由设置消 息队列的长度； DataCallback 为订阅者模块收到消息后执行的回调函数， 系统默

认的回调函数为将收到的消息压入消息队列， 但用户可以自定义收到消息后的

回调函数。

表 **4.1** 订阅者模块配置文件

字段名 字段解释

Topic 订阅者订阅消息的话题名称

Domain 订阅者所在的通信域

QueueSize 订阅者内部消息队列长度

DataCallback 订阅者接收到消息后执行的回调函数

订阅者模块中的管理接口由用户直接调用， 接口设计如表 4.2 所示。管 理接口中， create\_subscriber 接口需要配置文件作为参数并根据话题名称与 通信域供服务发现模块进行服务发现并与匹配的发布者进行点对点的连接； delete\_subscriber 接口不需要接收任何参数， 由接口内部向服务发现模块发出删 除请求； modify\_subscriber\_domain 接口需要由用户指定修改后的通信域作为参 数； query\_publishers\_info 与 query\_subscribers\_info 接口根据查询范围与查询条 件查询系统内的发布者信息或订阅者信息， 查询范围可以选择本地域或全域， 查 询条件可以选择按话题查询或全量查询。

表 **4.2** 订阅者模块接口

|  |  |
| --- | --- |
| 接口签名 | 接口解释 |
| create\_subscriber  delete\_subscriber  modify\_subscriber\_domain  query\_publishers\_info  query\_subscribers\_info | 创建订阅者  删除订阅者  修改订阅者者通信域 查询发布者信息 查询订阅者信息 |

订阅者模块中的消息队列保存自身订阅话题对应发布者发布的消息， 根据 消息队列大小动态地接收与丢弃消息队列中保存的消息。消息队列在基于 C++ 标准库提供的队列之上加入了互斥锁与条件变量以保证消息同步性。同样地， 消

息队列也向用户提供了操作接口，接口设计如表 4.3 所示。

表 **4.3** 消息队列相关接口

|  |  |
| --- | --- |
| 接口签名 | 接口解释 |
| size  empty  pop | 查询队列中消息的数量  查询队列是否为空  从队列的队头取出一条消息 |
| pop\_newest pop\_newest\_and\_clear | 从队列取出最新消息，即从队列的队尾取出一条消息 从队列取出最新消息并清空队列中所有消息 |

订阅者模块的时序设计如图 4.4 所示。

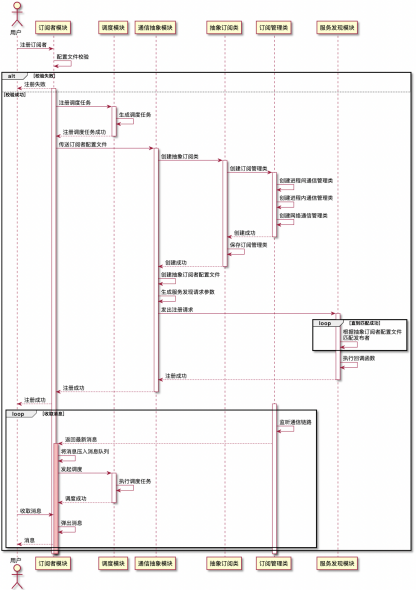


图 **4.4** 创建订阅者时序图

用户调用创建订阅者接口后， 订阅者模块首先校验配置文件中的必填字段 是按要求填写， 若配置文件校验失败则本次创建失败； 配置文件校验成功后， 将 其中的回调函数作为任务调度模块的调度任务； 调度任务设置完成后， 将配置文 件传至通信抽象模块， 通信抽象模块根据配置文件信息创建抽象订阅类， 在订阅 者配置文件的基础上加入 IP 地址、进程号以及订阅者更新通信链路的回调函数 形成抽象订阅者配置文件， 作为服务发现参数传至服务发现模块； 服务模块根据 抽象订阅者配置文件中的 IP 地址、进程号、话题名称以及通信域进入循环寻找 与订阅者匹配的发布者， 若寻找成功则调用抽象订阅者配置文件中的回调函数， 发布订阅双方完成点对点的连接并开始通信。

4.2.2 发布者模块概要设计

发布者模块用于实现基于发布-订阅模式异步通信中发布者的基础功能， 并 向用户管理订阅者和发布消息的功能接口。发布者模块的管理接口包括创建发 布者、删除发布者与修改发布者通信域，发布消息接口由发布者模块唯一提供。

同订阅者模块类似， 发布者模块内部保存了发布者配置文件， 该文件决定 了发布者发布消息的话题以及通信域， 配置文件如表 4.4 所示。由于发布者内部 并不需要维护消息队列， 其配置文件内容相比订阅者配置文件较少， 但 Topic 与 Domain 仍需要由用户显式给定。

表 **4.4** 订阅者配置文件

|  |  |
| --- | --- |
| 字段名 | 字段解释 |
| Topic Domain | 发布者发布消息的话题名称 发布者所在的通信域 |

发布者模块中的管理接口和发布消息接口由用户直接调用， 接口设计如表 4.5 所示。同订阅者模块的管理接口类似， create\_publisher 需要配置文件作为参 数并根据配置文件中的话题名称与通信域供服务发现模块寻找匹配的订阅者进 行点对点连接； delete\_publisher 接口同样不需要任何参数， 由接口内部向服务发 现模块发出删除请求； modify\_publisher\_domain 接口需要由用户指定修改后的通 信域作为参数； publish 接口是发布者发布消息的唯一接口； query\_publishers\_info 与 query\_subscribers\_info 接口功能与订阅者模块中的接口功能相同。

表 **4.5** 发布者模块接口

|  |  |
| --- | --- |
| 接口签名 | 接口解释 |
| create\_publisher  delete\_publisher  modify\_publisher\_domain  publish  query\_publishers\_info  query\_subscribers\_info | 创建发布者  删除发布者 修改发布者通信域  发布消息 查询发布者信息 查询订阅者信息 |

发布者模块的创建、删除与修改流程与订阅者模块大致相似， 二者唯一的不 同是通信抽象模块会返回与该发布者对应的抽象发布类。发布者模块保存该类 并通过该类中保存的发布管理类进行发布消息， 发布者创建及发布消息时序设 计如图 4.6 所示。

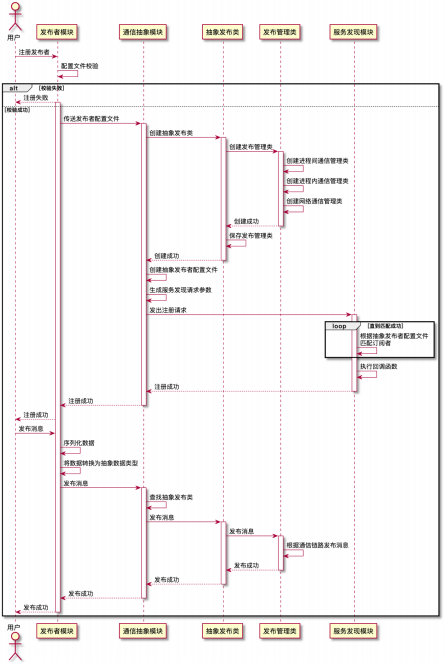


图 **4.5** 创建发布者时序图

4.3 服务模块概要设计

自动驾驶运行时通信系统基于 RPC 的同步数据通信功能由服务模块实现， 服务模块功能由服务服务端与服务客户端两个子模块共同完成。

4.3.1 服务客户端设计

服务客户端用于实现基于 RPC 同步数据通信中客户端即服务请求方的基础 功能，提供管理服务客户端接口及请求服务的功能接口。

服务客户端内部保存客户端配置文件， 该文件决定了客户端需要请求的服 务名称、通信域， 配置文件如表 4.6 所示。配置文件中， ServiceName 与 Domain

需要由显式给定，不可为空。

表 **4.6** 服务客户端配置文件

字段名

字段解释

ServiceName 服务客户端需要调用服务的名称

Domain 服务客户端请求服务的通信域范围

服务客户端接口设计如表 4.7 所示。 create\_service\_client 根据客户端配置 文件中的 ServiceName 与 Domain 初始化服务客户端； delete\_service\_client 不需 要任何参数， 被调用后直接删除客户端， 删除过程不需要由服务发现模块参与； modify\_service\_client\_domain 在接收用户指定的目标通信域后由服务发现模块完 成通信域的切换； is\_service\_available 接口实现查询客户端需要请求服务可用性； call\_service 接口实现发起向服务服务端发起请求； query\_service\_list 接口根据查 询范围与查询条件查询系统内现有服务列表， 查询范围可以选择本地域或全域， 查询条件可以选择按服务名称查询或全量查询。

表 **4.7** 服务客户端接口

|  |  |
| --- | --- |
| 接口签名 | 接口解释 |
| create\_service\_client  delete\_service\_client  modify\_service\_client\_domain  is\_service\_available  call\_service  query\_service\_list | 创建服务客户端  删除服务客户端  修改服务客户端通信域  查询服务可用性  发起服务请求  查询现有服务列表 |

服务客户端的创建、删除与修改通信域与通信单元模块中的订阅者模块流 程有较大区别， 服务客户端的信息并不需服务发现模块更新至通信网络拓扑中， 只需要服务模块帮助查询对应服务的服务端 IP 地址即可由RPC 完成通信。

用户调用创建服务客户端接口后， 服务客户端保存配置文件并将配置文件 并将配置文件作为参数传至通信抽象模块； 通信抽象模块根据配置文件字段创 建抽象服务类作为实际发起 RPC 调用的辅助类， 创建完成后将配置文件传至服 务发现模块； 服务发现模块将配置文件保存即结束创建工作， 服务客户端保存抽 象服务类实例后向用户返回创建成功信息。

用户调用发起服务请求接口后， 服务客户端通过调用保存的抽象服务类发 起实际的 RPC 调用， 调用完成后向用户返回结果。服务客户端的时序设计如图

4.7 所示。

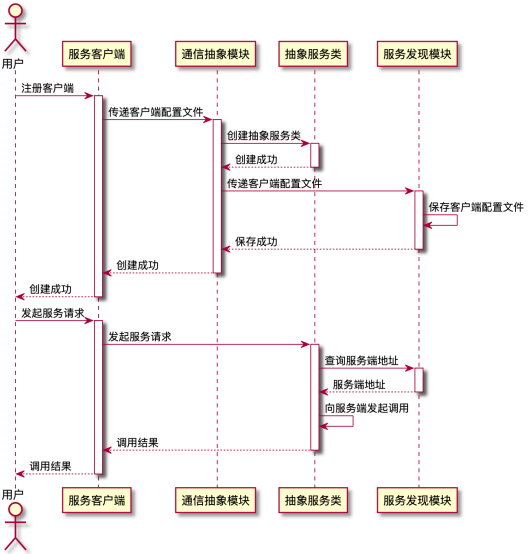


图 **4.6** 服务客户端时序图

4.3.2 服务服务端设计

服务服务端用于实现基于 RPC 同步数据通信中服务端即服务提供方的基础 功能，提供管理服务端接口及创建服务的功能接口。

服务服务端内部保存服务端配置文件， 该文件决定了服务端提供的服务名 称、服务功能以及通信域， 配置文件如表 4.8 所示。配置文件中的所有字段全部

由用户显式给定，不能为空。

表 **4.8** 服务服务端配置文件

|  |  |
| --- | --- |
| 字段名 | 字段解释 |
| ServiceName  Domain  ServiceCallback | 服务客户端需要调用服务的名称。  服务客户端请求服务的通信域范围。  服务端提供的处理函数。 |

服务服务端接口设计如表 4.9 所示。服务服务端提供的接口与服务客户端大 致相同。根据服务端的特点， 服务端接口删除了客户端中的发起服务请求接口与

查询服务可用性接口。

表 **4.9** 服务服务端接口

|  |  |
| --- | --- |
| 接口签名 | 接口解释 |
| create\_service\_server  delete\_service\_server  modify\_service\_server  query\_service\_list | 创建服务端  删除服务端 修改服务端通信域 查询现有服务列表 |

不同于服务客户端， 服务端的创建、删除、修改需要服务发现模块参与更新

网络拓扑。

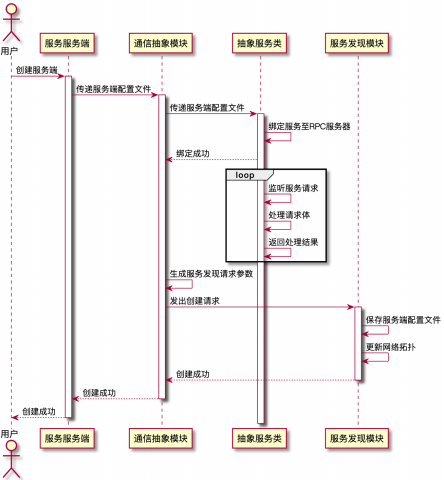


图 **4.7** 服务服务端时序图

服务服务端时序设计如图 4.8 所示。用户调用创建创建服务端接口后， 服 务端保存配置文件并将配置文件作为参数传至通信抽象模块； 通信抽象模块将 配置文件作为参数传至抽象服务类， 抽象服务类根据配置文件绑定服务至本地 RPC 服务器中； 通信抽象模块在配置文件基础上添加IP 地址和端口号并作为服

务发现请求参数传至服务发现模块； 服务发现模块根据请求参数更新网络拓扑， 随后向用户返回创建成功信息； 服务端创建成功后， 由服务抽象类负责监听服务 请求并根据用户提供的回调函数对请求体进行处理.

4.4 通信抽象模块概要设计

通信抽象模块是整个自动驾驶运行时通信系统中最为重要的模块之一， 该 模块将通信单元和服务的所有操作如创建、删除、修改、发布消息等在通信抽象 类中进行统一管理， 并负责创建抽象发布类、抽象订阅类与抽象服务类完成消息 的发布-订阅功能与服务的请求-应答功能。此外， 该模块将发布者、订阅者、服 务服务端与服务客户端的信息同步至服务发现模块其完成服务发现功能。

4.4.1 通信抽象类概要设计

通信抽象类负责处理通信单元模块和服务模块的所有请求， 控制抽象发布 类、抽象订阅类与抽象服务类的创建与删除， 协助服务发现模块完成服务发现功

能，该类唯一存在，其接口设计如表 4.10 所示。

表 **4.10** 通信抽象模块接口设计

|  |  |
| --- | --- |
| 接口签名 | 接口解释 |
| create\_publisher  delete\_publisher  modify\_publisher\_domain  publish  create\_subscriber  delete\_subscriber  modify\_subscriber\_domain  change\_communication\_mode  create\_service\_server  delete\_service\_server  modify\_service\_server\_domain  create\_service\_client  delete\_service\_client  modify\_service\_client\_domain  query\_subscribers\_info  query\_publishers\_info  query\_service\_server\_info | 处理创建发布者请求  处理删除发布者请求 处理发布者更改通信域请求  处理发布者发布消息请求 处理创建订阅者请求 处理删除订阅者请求  处理订阅者更改通信域请求  更改订阅者进程间通信方式  处理创建服务服务端请求  处理删除服务服务端请求  处理修改服务服务端通信域请求 处理创建服务客户端请求 处理删除服务客户端请求  处理修改服务客户度啊内通信域请求  查询系统内订阅者信息  查询系统内发布者信息  查询系统内服务端信息 |

通信抽象类在处理发布者或订阅者的创建请求时， 会生成与其对应的抽象 发布类或抽象订阅类， 并在类中为其绑定更新通信链路的回调函数供服务发现 模块收到网络拓扑变化时调用， 并在配置文件的基础上加入本机 IP 地址以及本 进程的进程号生成服务发现请求参数。处理服务端或客户端的创建请求时， 会 创建抽象服务类并由该类完成基于 RPC 的通信功能， 并不需要在抽象服务类中

绑定更新通信链路的回调函数。通信抽象类在服务端的配置文件基础上加入本 机 IP 地址， 而对客户端的配置文件不加入任何信息， 随后生成对应服务发现请

求参数。服务发现请求参数如表 4.11 所示。

表 **4.11** 服务发现请求参数

|  |  |  |
| --- | --- | --- |
| 请求参数 | 字段 | 字段解释 |
| PublisherInfo  SubscriberInfo  ServiceServerInfo  ServiceClientInfo | Topic  Domain  Address  Callback | 发布者发布消息的话题名称  发布者所在的通信域  发布者的 IP 地址和进程号  根据订阅者信息更新通信链路的回调函数 |
| Topic  Domain  Address  Callback | 订阅者订阅消息的话题名称  订阅者所在的通信域  订阅者的 IP 地址和进程号  根据发布者信息更新通信链路的回调函数 |
| ServiceName Domain  IP  port | 服务端提供服务的名称  服务端提供服务的通信域  服务端的 IP 地址  服务端的端口 |
| ServiceName Domain | 客户端请求服务的名称  客户端请求服务的通信域 |

4.4.2 抽象发布类概要设计

抽象发布类内部创建通信传输模块实现建立与订阅者通信链路的建立以及 消息的发布， 本系统通信方式自适应选择功能也在该类中实现， 每个发布者都唯 一对应了一个抽象发布类。当有匹配的订阅者时， 服务发现模块将订阅者的 IP 地址与进程号作为参数传入抽象发布类的回调函数中， 回调函数中抽象发布类 通过对自身的 IP 地址和进程号与参数做判断完成通信方式的自适应选择， 并返

回连接地址。通信方式自适应选择流程如图 4.9 所示。

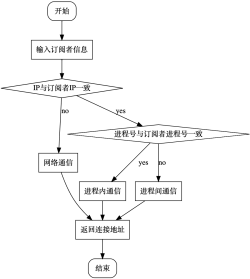


图 **4.8** 通信方式自适应选择流程图

4.4.3 抽象订阅类概要设计

抽象订阅类内部创建通信传输模块实现建立与发布者通信链路的建立以及 消息的订阅， 每个订阅者都唯一对应了一个抽象发布类当有匹配的发布者时， 服 务发现模块将抽象发布类返回的连接地址作为参数传入订阅抽象类的回调函数 中， 回调函数中通过对地址前缀的判断并连接至对应的通信链路， 建立与发布者

点对点的通信链路。抽象订阅类与发布者建立通信链路流程图如图 4.10 所示。

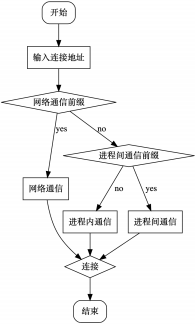


图 **4.9** 通信方式自适应选择流程图

4.4.4 抽象服务类概要设计

抽象服务类使用开源 RPC 框架rest\_rpc 实现服务模块中服务端注册服务与 客户端请求服务的功能， 该类由通信抽象类创建并唯一存在， 即多个服务端和客 户端共用一个抽象服务类。抽象服务类内部创建两个线程， 一个线程用于接收客 户端请求， 另一个线程用于发起服务请求和接收用户对客户端和服务的管理， 两 个线程处于异步运行状态。同一个进程中， 抽象服务类和服务发现模块的 RPC 服务共用一个端口。抽象服务类时序设计如图 4.11 所示。

第 4 章 自动驾驶运行时通信系统概要设计

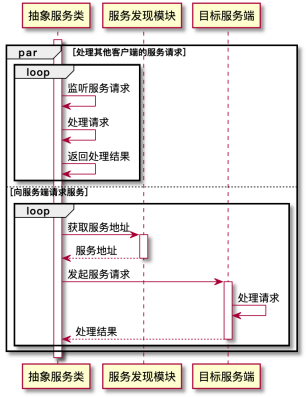


图 **4.10** 抽象服务类时序设计

4.5 通信传输模块概要设计

通信传输模块实现了本系统基于发布-订阅模式的通信功能， 所有发布者与 通信者的通信全部由本模块完成。根据本系统提供的通信方式， 通信传输模块分 为进程间通信、进程内通信和网络通信三个子模块。

4.5.1 进程内通信模块设计

进程内通信模块的功能是实现同一个进程中的发布者与订阅者的通信。进 程内通信模块的创建与删除由抽象发布类和抽象订阅类管理， 二者共用一个进 程内通信模块实例。抽象发布类通过该模块管理进程内发布者， 抽象订阅类通过 该模块管理进程内订阅者。

进程内通信模块包含一个进程内通信管理类、一个进程内通信发布者类和 一个进程内通信订阅者类。进程内通信管理类负责创建、删除进程内通信下的发 布者与订阅者， 并开辟线程轮询消息队列是否有新消息的到来； 进程内通信发布 者类负责将待发布消息压入消息对垒中； 进程内订阅者类负责从消息队列弹出 订阅消息， 并调用用户传入的回调函数将消息压入订阅者模块中的消息队列中 供用户使用。

由于发布者与订阅者处于同一个进程内， 二者间的通信不需要使用网络通

信或共享内存， 只需要传递消息的指针即可完成通信， 不需要对消息进行额外拷 贝。进程内通信模式下， 使用同一个话题通信的发布者与订阅者共用一个消息队 列， 发布者将消息的指针存入消息队列， 而订阅者将消息的指针从消息队列中取 出， 进程内通信模块消息传输设计如图 4.12 所示。从图 4.12 可以看出， 进程内 通信模块根据话题创建不同的消息队列， 使用同一个话题通信的发布者与订阅 者通过向消息队列压入或弹出消息的方式进行通信。同时， 本模块为了保证消息 队列在发布者之行压入消息和订阅者之行弹出消息操作时的原子性， 使用了互

斥锁机制对临界资源即消息队列进行保护。

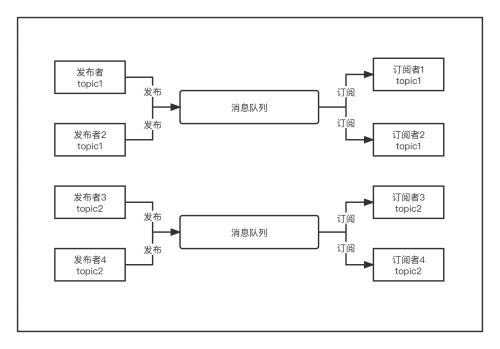


图 **4.11** 进程内通信消息传输

4.5.2 进程间通信模块概要设计

进程间通信模块的功能是使用共享内存实现同一台物理机不同进程中的发 布者与订阅者的通信。进程间通信模块的创建与删除由抽象发布类和抽象订阅 类管理，二者共用一个网络通信模块实例。

进程间通信模块包含一个进程间通信管理类、一个进程间通信发布者类和 一个进程间通信订阅者类。进程间通信管理类负责创建、删除进程间通信下的发 布者类与订阅者类， 并开辟线程轮询是否有新的消息到来； 进程间通信发布者类 负责将待发布消息进行序列化处理， 然后将序列化后的消息写入对应共享内存 文件， 通过 ZeroMQ PUB 套接字通知对应订阅者接收消息； 进程间通信订阅者 类通过 ZeroMQ SUB 套接字接收发布者的通知并从共享内存文件中取出对应的 消息并进行反序列化操作， 最后调用用户传入的回调函数将消息压入订阅者模

块中的消息队列中供用户使用。进程间通信模块传输设计如图 4.13 所示。

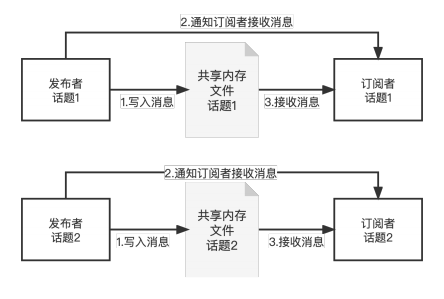


图 **4.12** 进程内通信消息传输

4.5.3 网络通信模块概要设计

网络通信模块的功能是使用 ZeroMQ 实现不同物理机建间的发布者与订阅 者的通信。网络通信模块的创建与删除由抽象发布类和抽象订阅类管理， 二者共

用一个网络通信模块实例。

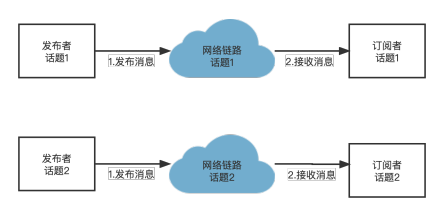


图 **4.13** 进程内通信消息传输

网络通信模块消息传输设计如图 4.14 所示。网络通信模块基于 ZeroMQ 中 的 PUB-SUB 套接字完成， 传输协议使用ZeroMQTCP 协议。网络通信模块包含 一个网络通信管理类、一个网络通信发布者类和一个网络通信订阅者类。网络通 信管理类负责创建、删除网络通信下的发布者与订阅者， 并开辟线程轮询网络链 路是否有新的消息到来； 网络通信发布者类负责将待发布消息进行序列化处理， 然后通过 ZeroMQ PUB 套接字发送； 网络通信订阅者类通过 ZeroMQ SUB 套接

字从网络链路中取出消息并对消息进行反序列化， 最后调用用户传入的回调函 数将消息压入订阅者模块中的消息队列中供用户使用。

4.5.4 消息模块概要设计

消息模块向网络通信模块和进程间通信模块提供消息的序列化、反序列化 以及端到端通信保护功能。

1. 序列化与反序列化概要设计

本系统使用 protobuf 作为消息的序列化与反序列化的工具。 protobuf 需要 由用户根据业务需要的数据结构编写后缀名为.proto 的 IDL 文件并使用protobuf 提供的 protoc 工具生成与.proto 文件对应的.pb.h 文件和.pb.cc 文件， 用户通过引 用.pb.h 文件中的 C++ 接口实现对消息的序列化与反序列化， 但.pb.h 中的接口与 C++ 原生数据结构(int，vector 等) 在赋值等场景下的使用方法并不相同， 本系 统基于 protobuf 生成的.pb.h 与.pb.cc 文件二次生成一套 C++ 代码。二次生成的 代码向用户提供与.proto 文件对应的原生 C++ 数据结构， 用户在发布者类与订阅 者类使用原生 C++ 数据结构进行对消息的处理。但在网络通信模块和进程间通 信模块发布消息时， 二次生成代码通过 protobuf 接口将原生的 C++ 数据结构转 换为 protobuf 接口并调用protobuf 提供的序列化接口将转换后的结构序列化后 发布消息； 在网络通信模块和进程间通信模块接收消息时， 二次生成代码通过 protobuf 提供的反序列化接口先将消息反序列化为 protobuf 结构， 再将 protobuf 结构转换为原生 C++ 结构。

2. 端到端通信保护概要设计

4.6 服务发现模块概要设计

服务发现模块是实现本系统实现动态网络下通信功能的重要模块， 服务发 现模块唯一存在于每个进程。

服务发现模块根据抽象服务类以及中心发出的服务发现请求参数实现服务 发现功能， 服务发现请求参数如表 4.11 所示。服务发现模块服务发现机制分为 两类： 主动发现和被动发现。一方面， 通信抽象类在本机内新增、删除或修改发 布者、订阅者以及服务服务端时导致网络拓扑变化时向服务发现模块发出请求， 服务发现模块主动向中心节点发出发现请求获取匹配通信方信息； 另一方面， 在 其他进程中中网络拓扑发生变化时其自身的服务发现模块会向中心节点发出发 现请求获取匹配通信方信息， 中心节点会自动寻找相关联的服务发现模块并发 出更新信息查询请求， 关联的服务发现模块根据更信息更新通信链路。服务发现 模块的服务发现功能的工作流如下：

1. 本进程内新增、删除或修改发布者， 服务发现模块将通信抽象类发出的请

求参数通过 RPC 更新该发布者信息至中心节点， 并将参数保存至本模块。 中心节点按服务发现模块查询匹配的订阅者信息， 若查询到则将所有关联 的其他进程中的服务发现模块的 RPC 地址和订阅者信息返回至本进程服 务发现模块。本进程服务发现模块将每个订阅者网络信息作为参数调用抽 象发布类的更新通信链路回调函数完成自适应通信方式连接后，通过 RPC 将连接地址和订阅者信息发送给每一个其他进程的服务发现模块。其他进 程的服务发现模块根据连接地址调用抽象订阅类中的更新通信链路回调函 数完成自适应通信连接。匹配的发布者与订阅者连接至同一个连接地址， 双方开始通信。

2. 本进程内新增、删除或修改订阅者， 服务发现模块将通信抽象类发出的请 求参数通过 RPC 更新该订阅者信息至中心节点， 并将参数保存至本模块。 中心节点按服务发现模块查询匹配的发布者信息， 若查询到则将本进程服 务发现模块的 RPC 地址和订阅者网络信息通过 RPC 发送至所有关联的其 他进程中的服务发现模块。其他进程的服务发现模块根据订阅者网络信息 完成自适应通信方式连接后将连接地址和订阅者网络信息通过 RPC 发送 至本进程的服务发现模块。本进程服务发现模块根据连接地址完成自适应 通信连接，双方开始通信。

3. 本进程内新增、删除或修改服务服务端， 服务发现模块将通信抽象类发出 的请求参数通过 RPC 更新该服务服务端信息至中心节点， 并将参数保存至 本模块。更新完成后， 不需要与关联的客户端建立通信链路。其他进程的 客户端仅在向请求本进程中的服务时通过其服务发现模块向中心节点请求 服务所在 IP 地址。

4. 本机进程内新增、删除或修改服务客户端， 服务发现模块仅保存参数， 并 不向中心节点发起 RPC 请求。

5. 其他进程内新增、删除或修改发布者， 其他进程的服务发现模块会将订阅 者网络信息和连接地址通过 RPC 的方式通知本进程的服务发现模块。本进 程服务发现模块根据订阅者信息查询到与其对应的抽象订阅类， 将连接地 址作为参数调用抽象订阅类的更新通信链路回调函数完成自适应通信方式 连接，双方开始通信。

6. 其他进程内新增、删除或修改订阅者， 中心节点会将其他进程的服务发现 模块的 RPC 地址和订阅者网络信息通过 RPC 通知本进程的服务发现模块。 本进程服务发现模块根据订阅者网络信息查询到对应的抽象发布类， 将订 阅者网络信息作为参数调用抽象发布类的更新通信链路回调函数完成自适 应通信方式连接， 最后将连接地址通过 RPC 返回至其他进程。其他进程根 据连接地址完成自适应通信方式连接后，双方开始通信。

服务发现模块的服务发现工作流程图如图 4.14 所示。

服务发现模块除了提供服务发现的功能外， 还提供查询网络拓扑信息的功 能。服务发现模块的查询功能的工作流如下：

1. 用户查询网络内订阅者或发布者信息， 服务发现模块根据用户设置的查询 模式以及通信域构造 RPC 请求体向中心节点发出查询请求。中心节点根据 服务发现模块的请求参数将符合查询条件的信息返回至服务发现模块， 服 务发现模块继续将查询结果返回至用户。

2. 用户查询网络内服务端信息， 服务发现模块根据用户设置的查询模式以及 通信域构造 RPC 请求体向中心节点发出查询请求。中心节点根据服务发现 模块的请求参数将符合查询条件的信息返回至服务发现模块， 服务发现模 块继续将查询结果返回至用户。

服务发现模块的查询网络拓扑信息工作流程图如图 4.15 所示。

4.7 任务模块概要设计

任务模块的功能是将进程内一个个分散的发布者、订阅者、服务客户端和服 务端统一包装为一个完整的任务， 这样设计的原因是采用任务的设计不仅可以 契合自动驾驶系统模块化的思想，还可以管理每个任务的运行状态。

本模块采用类的方式定义一个抽象任务类， 抽象任务类将所有接口都设计 为虚函数， 由用户根据实际业务需要实现具体逻辑， 抽象任务类的接口设计如表

4.11 所示。

表 **4.12** 任务模块接口设计

接口签名 接口解释

init 任务初始化

pause 任务暂停

resume 任务恢复运行

running 任务运行

destroy 任务销毁

从表 4.13 可以看出， 本模块将一个任务的接口定义为五种： 初始化、暂停、 恢复、运行及销毁。其中， 初始化接口向用户提供该任务所需要的变量、发布者、 订阅者、服务端或客户端的初始化工作； 暂停接口向用户提供基于判断条件的暂 停任务运行功能； 恢复运行接口向用户提供基于判断条件的将处于暂停状态的 任务重新恢复运行的功能； 运行接口向用户提供任务运行时的数据处理逻辑等 功能； 销毁接口向用户提供销毁该任务回收所有资源的功能。其中运行接口是任 务模块中最为重要的接口， 该接口不仅是用户实现核心业务逻辑的接口， 还是调 度模块调度任务的入口。

一个任务共存在四种状态： 初始化、运行、暂停和销毁， 任务的状态管理由

状态机管理，任务模块的状态机设计如图 4.13 所示。

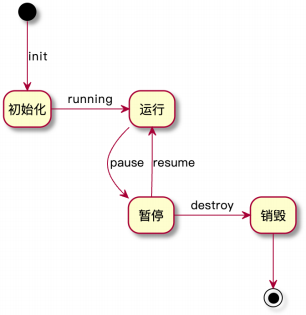


图 **4.14** 任务模块状态机

4.8 调度模块概要设计

调度模块的功能是根据调度策略控制任务模块的运行， 调度模块唯一存在 于每个不同的进程。调度模块根据用户提供的调度配置文件创建调度任务并执 行调度任务， 调度配置文件如表 4.13 所示。 taks\_name 是不同调度任务的名称， 不同的调度任务以任务名称区分； task\_callback 是调度任务对应的回调函数， 当 任务被调度时将执行该函数。配置文件中的字段全部由用户指定。

表 **4.13** 调度配置文件

|  |  |
| --- | --- |
| 字段名 | 字段解释 |
| task\_name task\_callback | 调度任务名称  调度任务对应的回调函数 |

调度模块的接口设计如表 4.14 所示。 create\_message\_task 接口用于创建基于 消息触发的调度任务， create\_time\_task 接口用于创建基于时间触发的调度任务， avtivate\_task 接口用于激活指定的调度任务。

表 **4.14** 调度模块接口设计

|  |  |
| --- | --- |
| 接口签名 | 接口解释 |
| create\_message\_task  create\_time\_task  activate\_task | 创建基于消息触发的调度任务  创建基于时间触发的调度任务  激活指定调度任务 |

调度模块支持基于基于消息触发的调度策略和基于时间触发的调度策略。基 于消息触发的调度策略是侵入式的， 即该调度策略并不需要由用户手动去激活 调度任务， 而是由通信抽象模块监听到新消息时调用订阅者模块中的回调函数， 并最终在订阅者模块的回调函数中激活对应调度任务。在同一个任务中， 基于消 息触发的调度任务数量等于订阅者数量。基于消息触发的调度策略时序设计如

图 4.14 所示。

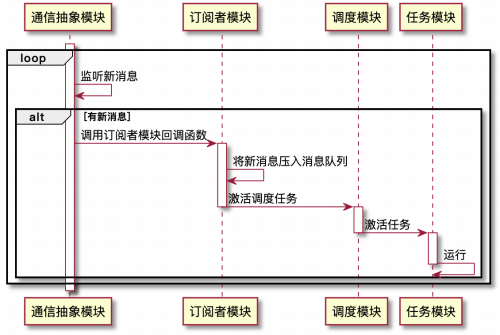


图 **4.15** 基于消息触发的调度策略时序图

基于时间触发的调度策略相较于基于消息触发调度策略不再是侵入式的， 需 要用户显式指定任务的调度频率。基于时间触发的调度策略并不根据是否有新 消息到来判断是否激活调度任务， 而是从创建调度线程时就以固定的频率调用 任务用户任务中的运行接口。该调度策略下， 调度模块记录首先计算任务在调度 频率下的执行时间间隔， 每当任务调度完成后继续计算当前时间与执行时间间 隔计算调度剩余时间， 若调度剩余时间大于零， 则在线程内睡眠剩余时间后再进 行下一次调度， 若调度剩余时间小于零， 则立即进行下一次调度。基于时间触发 的调度策略时序设计如图 4.15 所示。

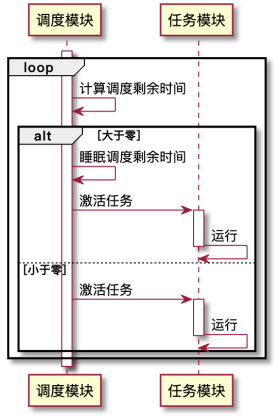


图 **4.16** 基于时间触发的调度策略时序图

4.9 中心节点概要设计

中心节点是本系统实现服务发现功能的核心模块， 中心节点接受所有服务 发现模块的 RPC 调用保存整个通信系统的网络拓扑信息并协助各服务发现模块

完成通信链路的建立。

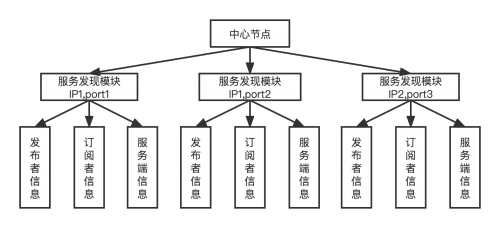


图 **4.17** 网络拓扑信息存储模型

中心节点根据 IP 地址以及端口号将服务发现模块进行区分， 并且将每个服 务发现模块中的网络拓扑信息进行单独存储， 存储模型示意图如图 4.17 所示。从

图中可以看出， 中心节点将 IP 地址与端口号两个信息中任意一个不相同的服务 发现模块都进行区分。中心节点存储信息包括发布者信息、订阅者信息以及服务 端信息， 三种信息都来自各服务发现模块进行主动发现时使用的请求参数。请求 参数如表 4.11 中的 PublisherInfo 、SubscriberInfo 和 ServiceServerInfo。

为了防止中心节点异常退出导致网络拓扑信息丢失， 中心节点会定期将自 身保存的通信系统网络拓扑信息转换为 json 文件并保存至本地。当中心节点异 常退出并重启时首先加载 json 文件中的历史网络拓扑信息， 可以有效恢复异常 退出前的网络拓扑信息并继续工作。

中心节点提供了一系列供服务发现模块请求的服务， 其中最主要的服务包 括对发布者、订阅者以及服务端的注册、删除和修改。除此之外， 中心节点还提 供网络拓扑信息查询服务。

4.10 本章小节

本章从自动驾驶运行时系统的整体结构为切入点， 详细论述了该系统的整 体结构； 依次介绍系统各功能模块功能及设计思路、技术重点和难点， 为后续的 详细设计与实现做铺垫。

第 **5** 章 自动驾驶运行时通信系统详细设计与实现

本章基于概要设计的基础之上， 对自动驾驶运行时通信系统各功能模块进 行详细设计与实现，主要内容包括系统线程设计、类设计以及关键算法的描述。

5.1 系统软件架构设计

在概要设计中为了方便本系统向用户提供高抽象能力的接口对本系统进行 了层次划分， 同时为了便于开发对模块也进行了划分。本节从代码开发的角度对 本系统软件架构的各层次、各模块关系进行明确定义。本系统的组件图如图 5.1

所示。

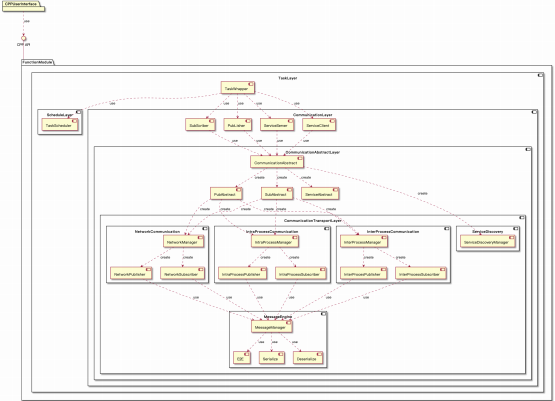


图 **5.1** 自动驾驶运行时通信系统组件图

从图 5.1 可以看出， 本系统在开发阶段严格遵守概要设计阶段对本系统的层 次划分以及模块划分规则。

5.2 通信单元模块详细设计与实现

通信单元模块向用户实现了用户管理基于发布-订阅的异步通信的应用程序 接口， 根据概要设计中提出的设计方案， 通信单元模块由发布者模块和订阅者模

块两个二级子模块共同实现功能。发布者模块实现管理发布者以及发布消息的 功能，订阅和模块实现管理订阅者以及订阅消息的功能。

5.2.1 发布者模块详细设计与实现

本模块使用 C++ 语言开发并融合其面向对象的编程思想， 向用户提供高度 抽象的应用程序接口。根据概要设计中的接口设计方案， 发布者模块相关类设计

如图 5.2 所示。

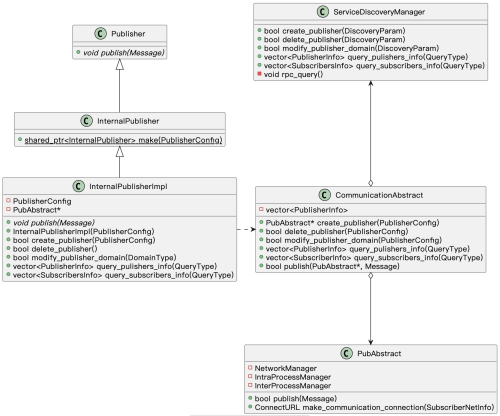


图 **5.2** 发布者模块类图

如图 5.2 所示， 发布者模块主要由 Publisher 、InternalPublisher 和 InteralPub- lisher 三个类构成。 InternalPublisher 类根据用户传入的发布者配置文件 Publish- erConfig 使用工厂函数 make() 创建 InternalPublisherImpl 类， 并通过 C++ 多态的 特性将 Publisher 类的指针指向 InternalPublisherImpl 类， 用户最终使用 Publisher 类完成管理、发布消息等操作。 InternalPublisherImpl 类中保存发布者配置文件 PublisherConfig 和抽象发布类 PubAbstract 的指针。 CommunicationAbstract(通信 抽象类)、PubAbstract(抽象发布类) 和 ServiceDiscoveryManager(服务发现类) 三个 类共同服务于 Publisher 类的创建、修改、修改和发布消息功能。

发布者模块的时序图如图 5.3 所示。用户创建发布者时， 首先调用 Inter- nalPublisherImpl 类中的 create\_publisher 函数，函数体内继续调用 Communication-

第 5 章 自动驾驶运行时通信系统详细设计与实现

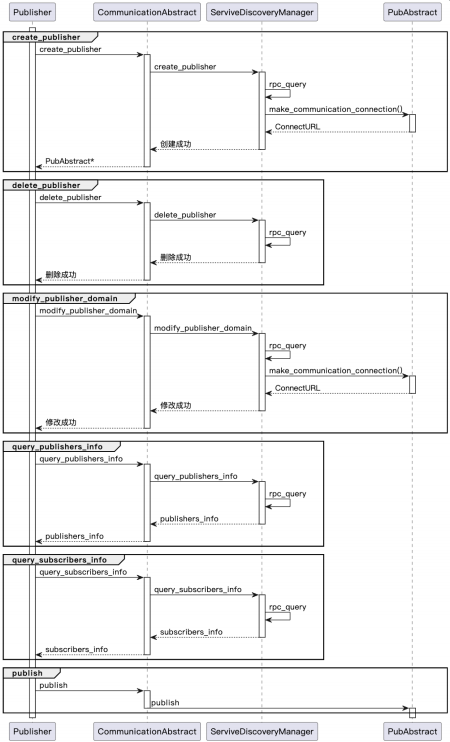


图 **5.3** 发布者模块时序图

Abstract 类的 create\_publisher 函数，CommunicationAbstract 类创建 PubAbstract 类 并在 PublisherConfig 基础上加入 IP 和进程号生成服务发现参数继续 Discovery- Param 调用 ServiceDiscoveryManager 类中的 create\_publisher 函数，ServiceDiscov- eryManager 类在函数内调用 rpc\_query 向中心节点发出创建请求完成服务发现工 作得到匹配订阅者的网络信息， 并调用 CommunicationAbstract 类绑定的回调函

数在 PubAbstract 类完成通信链路的建立， 最后 CommunicationAbstract 类返回之 前创建的 PubAbstract 类的指针供 Publisher 类调用 publish 函数发布消息； 删除发 布者时， 调用 InternalPublisherImpl 类中的 delete\_publisher 函数，在函数内继续调 用 CommunicationAbstract 类的 delete\_publisher 函数， CommunicationAbstract 类 首先删除 Publisher 类对应的 PubAbstract 类并调用 ServiceDiscoveryManager 类完 成网络拓扑的更新； 修改发布者通信域时， 调用 InternalPublisherImpl 类中的 mod- ify\_publisher\_domain 函数， 在函数内继续调用 CommunicationAbstract 类的 mod- ify\_publisher\_domain 函数，CommunicationAbstract 类根据传入的通信域类型生成 服务发现请求参数调用 ServiceDiscoveryManager 类的 modify\_publisher\_domain 函数， ServiceDiscoveryManager 类向中心节点发出更改请求完成服务发现工作并 调用 CommunicationAbstract 类绑定的回调函数在 PubAbstract 类重新建立通信链 路。

当用户查询通信系统内的订阅者信息或发布者信息时， 首先调用 In- ternalPublisherImpl 类 中 的 query\_publishers\_info 或 query\_subscribers\_info 函 数， 随 后 继 续 调 用 CommunicationAbstract 类 中 的 query\_publishers\_info 或 query\_subscribers\_info 函数， 函数 内根据用户传入 的 QueryType 生成 Dis- coveryParam 调 用 ServiceDiscoveryManager 类 中 的 query\_publishers\_info 或 query\_subscribers\_info 函数， ServiceDiscoveryManager 类根据传入的 QueryType 向中心节点查询订阅者或发布者的信息，返回结果。

当用户使用 Publisher 类中的 publish 函数发布消息时， publish 函数内直接使 用 InternalPublisherImpl 类中保存的抽象发布类指针 PubAbstract\* 作为参数调用 CommunicationAbstract 类中的 publish 函数， CommunicationAbstract 类通过指针 调用 PubAbstract 类中的 publish 函数， 由该类完成对消息的序列化并通过建立的 通信链路将消息发布。

5.2.2 订阅者模块详细设计与实现

本模块与发布者模块设计思想一致， 向用户提供高度抽象的应用程序接口。 根据概要设计中的接口设计方案，订阅者模块相关类设计如图 5.4 所示。

如图 5.4 所示， 订阅者模块类图设计同发布者模块基本一致， 主要由 Sub- scriber、InternalSubscriber 和InternalSubscriberImpl 三个类构成。InternalSubscriber- Impl 内部保存订阅者配置文件和调度配置文件， 除了创建、删除和修改通信域三 个函数， 该类新增了修改通信方式的函数 change\_communication\_mode 和一系列 操作消息队列的函数。 CommunicationAbstract(通信抽象类) 、SubAbstract(抽象订 阅类) 和 ServiceDiscoveryManager(服务发现类) 三个类共同服务于 Subscriber 类 的创建、修改、修改和发布消息功能。不同的是， InternalSubscriberImpl 类需要

使用 Dequeue(双向队列类) 作为消息队列存放订阅到的消息，并使用 TaskSched- uler(调度类) 侵入式地注册基于消息触发的调度策略调度任务。 Dequeue 类内部 存放了互斥锁保证用户从队列中取出消息和底层通信抽象模块向队列压入消息 这两个操作是互斥的，保证数据的同步性。

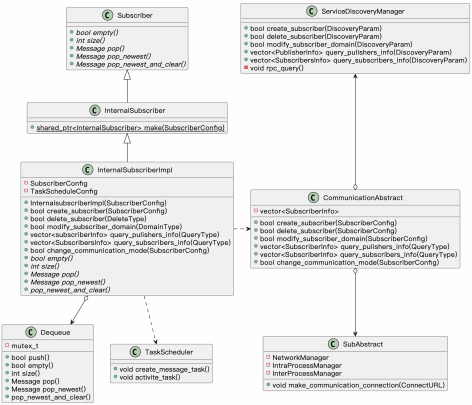


图 **5.4** 订阅者模块类图

订阅者模块时序图如图 5.5 所示。

用户创建订阅者时， 首先调用 InternalSubscriberImpl 类的 create\_subscriber 函数， 函数体内设置调度配置文件 ScheduleConfig 并调用 TaskScheduler 类的 cre- ate\_message\_task() 函数创建调度任务并生成相应的回调函数供 SubAbstract 类进 行回调， 回调函数的功能是将新消息压入消息队列并唤醒调度任务， 随后继续 调用 CommunicationAbstract 类的 create\_subscriber 函数， CommunicationAbstract 类在函数中生成服务发现请求参数 DiscoveryParam 后调用 ServiceDiscoveryMan- ager 类的 create\_subscriber 函数， ServiceDiscoveryManager 类调用 rpc\_query 函数 后得到匹配发布者返回的通信地址后， 调用 CommunicationAbstract 类绑定的回 调函数在 SubAbstract 类中完成通信链路的建立。 SubAbstract 类在完成通信链路 的建立后开辟轮询新消息的新线程， 有新消息到来时调用 InternalSubscriberImpl 类生成的回调函数将新消息压入消息队列并唤醒调度任务。

用 户 修 改 订 阅 者 通 信 方 式 时， 首 先 调 用 InternalSubscriberImpl 类 的 change\_communication\_mode 函数， 随后继续调用 CommunicationAbstract 类的

第 5 章 自动驾驶运行时通信系统详细设计与实现

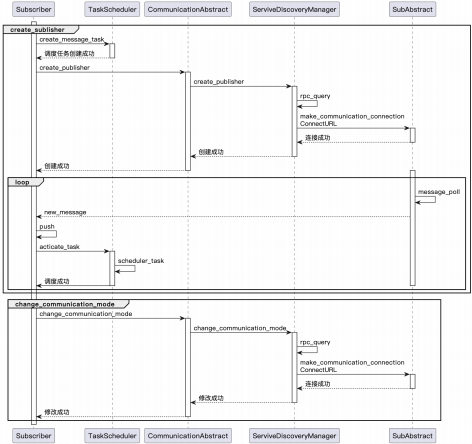


图 **5.5** 订阅者模块时序图

change\_communication\_mode 函数， CommunicationAbstract 类生成服务发现请 求参数后调用 ServiceDiscoveryManager 类的 change\_communication\_mode 函数， ServiceCommunicationManager 类调用rpc\_query 函数后得到匹配发布者返回的通 信地址后，最终在 SubAbstract 类中完成通信链路的建立。

用户删除订阅者时， InternalSubscriberImpl 类根据用户传入的删除模式 DeleteType 判断是否需要将消息队列中的消息保存至用户空间， 随后删除过程 同删除发布者过程一致。此外修改订阅者通信域、查询发布者信息和查询订阅者 信息这三个操作同发布者模块的过程并无区别， 本小节及下一节不再赘述这三 种操作的过程。

5.3 服务模块详细设计与实现

服务模块实现了基于 RPC 请求-应答的同步通信功能， 根据概要设计中提出 的设计方案， 服务模块由客户端和服务端两个二级子模块共同实现功能。服务端 实现管理服务端以及向客户端提供服务的功能， 客户端实现管理客户端以及向 服务端请求服务的功能。服务模块设计思想同通信单元模块一致， 都采用了面向 对象的编程方法，向用户提供高度抽象的应用程序接口。

5.3.1 服务端详细设计与实现

根据概要设计中的接口设计方案，服务端相关类设计如图 5.6 所示。



图 **5.6** 服务端类图

如图 5.6 所示， ServiceServer 类通过 make() 函数根据传入的服务端配置文件 构造实际工作类 InternalServiceServerImpl 类。InternalServiceServerImpl 类实现了 ServiceServer 类的所有虚函数， 用户基于多态特性通过 ServiceServer 类即可调 用 InternalServiceServerImpl 的所有函数。 CommunicationAbstract、ServiceDiscov- eryManager 和 SubAbstract 三个类共同完成服务于 ServiceServer 类的提供服务功 能。

服 务 端 时 序 图 如 图 5.7 所 示。 用 户 通 过 ServiceServer 类 中 的 cre- ate\_service\_server 函数创建服务端， 函数体内继续调用 CommunicationAbstract 类中的 create\_service\_server 函数， CommunicationAbstract 类根据用户传入的服 务端配置文件 ServiceServerConfig 调用 ServiceAbstract 类中的 bind\_service 函数， ServiceAbstract 类根据配置文件中的服务名、回调函数绑定服务至 rpc 服务器， 随后 CommunicationAbstract 类生成服务发现请求参数 DiscoveryParam 调用 Ser- viceDiscoveryManager 类的 create\_service\_server 函数， ServiceDiscoveryManager 类调用 rpc\_query 函数完成向中心节点的注册，返回注册成功。

用户通过 ServiceServer 类中的 delete\_service\_server 函数删除服务端， 函数 体内继续调用 CommunicationAbstract 类中的 delete\_service\_server 函数， Commu- nicationAbstract 类首先调用 ServiceDiscoveryManager 类在网络拓扑中删除该服 务， 随后调用 ServiceAbstract 类中的unbind\_service 函数解除服务的绑定， 最后 返回删除成功。

第 5 章 自动驾驶运行时通信系统详细设计与实现

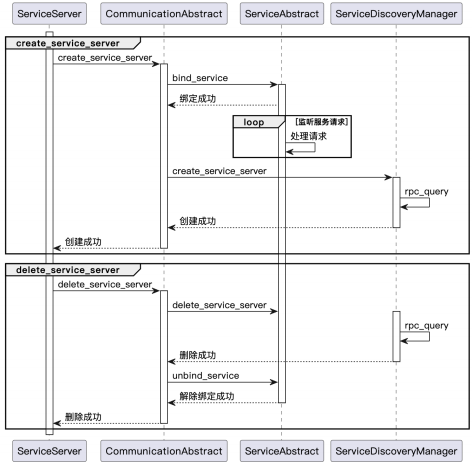


图 **5.7** 服务端时序图

5.3.2 客户端详细设计与实现

根据概要设计中的接口方案， 客户端相关类设计如图 5.8 所示。客户端的设 计思想同服务端一致， ServiceClient 类通过 make() 函数根据传入的客户端配置文 件构造实际工作类 InternalServiceClientImpl 类。 InternalServiceClientImpl 类实现 了 ServiceClient 类所有的虚函数。与服务端不同， 客户端的注册、删除和修改通 信域并不需要 ServiceDiscoveryManager 类向中心节点发起请求， ServiceDiscov- eryManager 类只将客户端配置文件保存至本地。只有在客户端发起实际服务请 求时 ServiceDiscoveryManager 类才向中心节点发起请求。 ServiceAbstract 采用单 例设计模式，同一个进程中只存在唯一的 ServiceAbstract 类实例。

客 户 端 时 序 图 如 图 5.9 所 示。 用 户 通 过 ServiceClient 类 中 的 cre- ate\_service\_client 函数创建客户端， 函数体内继续调用 CommunicationAbstract 类 中的 create\_service\_client ，CommunicationAbstract 类将客户端配置文件 Service- ClientConfig 保存， 随后返回创建成功， 创建过程并不需要 ServiceDiscoveryMan- ager 类参与。此外， 用户调用 ServiceClient 类中的 delete\_service\_client 函数和 modify\_service\_client\_domain 函数同样不需要 ServiceDiscoveryManager 类参与。

用户通过 ServiceClient 类中的 call\_service 函数发出对服务的请求， 函数内

第 5 章 自动驾驶运行时通信系统详细设计与实现



图 **5.8** 客户端类图

调用 CommunicationAbstract 类的 call\_service 函数并最终在 ServiceAbstract 类中 call\_service 函数执行实际的服务请求。 ServiceAbstract 类通过调用 ServiceDiscov- eryManager 类中的 lookup\_for\_service 函数得到服务的 IP 地址， 随后使用本地的 rpc 服务器发起服务请求。

用户可以通过 ServiceClient 类中的 is\_service\_available 函数查询需要请求的 服务是否可用， 函数最终在 ServiceDiscoveryManager 类的 is\_service\_available 函 数中完成对服务的查询并返回结果。

第 5 章 自动驾驶运行时通信系统详细设计与实现

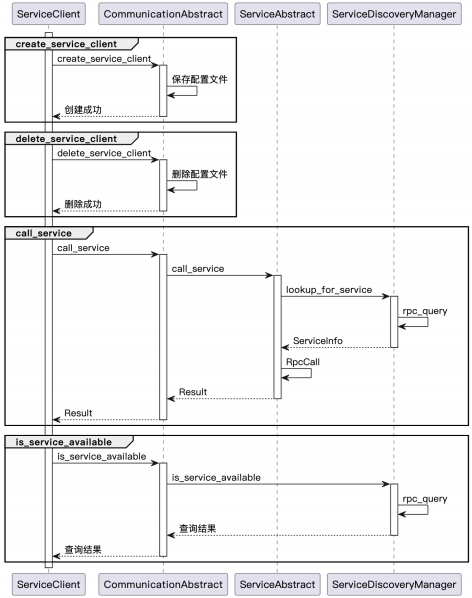


图 **5.9** 客户端时序图

5.4 通信抽象模块详细设计与实现

通信抽象模块是整个通信系统承上启下的模块， 一方面本模块接受通信单 元模块及服务模块提供给用户的创建、删除等应用程序接口的调用， 另一方面本 模块需要将用户所使用的高度抽象的应用程序接口进行处理， 根据配置文件创 建抽象订阅类、抽象发布类和抽象服务类并且生成服务发现请求参数通过服务 发现模块将本进程内发布者、订阅者、服务端信息更新至网络拓扑中。本模块将 用户层高度抽象的接口与底层复杂的接口进行了转换。

通信抽象模块相关类设计如图 5.10 所示。通信抽象模块包含 Communica- tionAbstract、ServiceDiscoveryManager、PubAbstract、SubAbstract、ServiceAbstract

和 Util 六个类。 Util 类作为工具类供 CommunicationAbstract 类获取本机的 IP 地 址和本进程的进程号。 ServiceDiscoveryManager 类提供服务发现功能。 PubAb- stract、SubAbstract 和 ServiceAbsract 三个类是本模块的核心类， 三个类的生命周 期 CommunicationAbstract 类负责。NetworkManager (网络通信管理类)、IntraPro- cessManager (进程内通信管理类) 和 InterProcessManager (进程间通信管理类) 三个类由 PubAbstract 类和SubAbstract 类创建并负责管理其生命周期， 三个属于 通信传输模块的入口类。 PubAbstract 类和SubAbstract 类本身不实现通信的功能， 而是由内部的三个通信管理类进行实际的基于发布-订阅的通信。 ServiceAbstract 类与 PubAbstract 类和 SubAbstract 类不同， 内部完全实现了基于服务的通信， 不

需要创建额外的通信管理类辅助通信。



图 **5.10** 通信抽象模块类图

本节从通信方式自适应选择算法的实现和抽象服务类的设计与实现两方面 进行阐述。

5.4.1 通信方式自适应选择算法实现

本系统的核心功能之一通信方式自适应选择在本模块中的抽象发布类和抽 象订阅类中的 make\_communication\_connection 函数实现。根据 4.4 节对本模块的 概要设计， 通信方式自适应选择分为两部分： 一部分是发布者根据自身和订阅者

网络信息选择通信方式并返回连接地址， 下文简称为创建算法； 另一部分是订阅 者根据返回的连接地址选择通信方式并连接，下文简称为连接算法。

如算法[5.1](#_bookmark121)所示， 当发布者自身 IP 地址与订阅者 IP 不一致时返回网络通信 连接地址； 若 IP 一致但进程号不一致返回进程间通信连接地址； 若进程号一致 则返回进程内通信连接地址。在返回网络地址前， 使用对应的通信管理类创建发 布者并将其绑定到连接地址上。对于进程内通信和进程间通信方式， connectURL 由地址前缀 intraprocess 或 ipc 加上发布者自身的话题名称组成。对于网络通信 方式， zeroMQ 使用 tcp 为前缀的地址进行跨物理机的通信， 地址中需要指定 IP 地址和端口号。本系统将发布者所在物理机 IP 作为 zeroMQ 中的 IP 地址， 端口 号则设置为通配符 \* 由 zeroMQ 自动寻找可用端口号。

|  |
| --- |
| 算法 **5.1** 创建算法 |
| **Data:** 订阅者网络信息 SubscriberNetInfo，发布者自身网络信息 PublisherNetInfo **Result:** 自适应通信连接地址， connectURL  **1** **if** *SubscriberNetInfo.ip* *==* *PublisherNetInfo.ip* **then**  **2** **if** *SubscriberNetInfo.pid* *==* *PublisherNetInfo.pid* **then**  **3** connectURL = intraprocess://topic\_name;  **4** publisher = IntraProcessManager.create\_publisher(connectURL);  **5** publisher.bind(connectURL);  **6** **else**  **7** connectURL = ipc://topic\_name;  **8** publisher = 和 InterProcessManager.create\_publisher(connectURL);  **9** publisher.bind(connectURL);  **10** **end**  **11** **else**  **12** connectURL = tcp://ip\_address:\*;  **13** publisher = NetworkManager.create\_publisher(connectURL);  **14** publisher.bind(connectURL);  **15** **end**  **16** return connectURL |

如算法[5.2](#_bookmark122)所示， 订阅者得到发布者返回的连接地址后， 将连接地址的前缀 取出并和各通信方式连接地址的前缀比较， 选择对应的通信方式， 通过对应的 通信管理类创建订阅者。创建完成后调用 set\_callback 函数将 Subscriber 类中的 回调函数 DataCallback 注册到对应的通信管理类中并将注册的订阅者连接至连 接地址上， 最后通信管理类开启轮询接收消息。 在算法 5.1 和算法 5.2 中， 抽象 发布类和抽象订阅类通过三个通信管理类创建对应通信方式的发布者和订阅者， 随后由通信传输模块进行实际的消息传输， 详细传输设计及算法将在下一节展 开阐述，这里不做赘述。

|  |
| --- |
| 算法 **5.2** 连接算法 |
| **Data:** 连接地址 connectURL  **1** prefix = getPrefix(connectURL);  **2** **if** *prefix* *==* *tcp://***then**  **3** subscriber = NetWorkProcessManager.create\_subscriber(connectURL);  **4** NetWorkProcessManager.set\_callback(DataCallback);  **5** subscriber.connect(connectURL);  **6** NetWorkProcessManager.message\_poll();  **7** **else** **if** *prefix* *==* *interprocess://***then**  **8** InterProcessManager.create\_subscriber(connectURL);  **9** InterProcessManager.set\_callback(DataCallback);  **10** subscriber.connect(connectURL);  **11** NetWorkProcessManager.message\_poll();  **12** **else** **if** *prefix* *==* *intraprocess://***then**  **13** IntraProcessManager.create\_subscriber(connectURL);  **14** IntraProcessManager.set\_callback(connectURL,DataCallback);  **15** subscriber.connect(connectURL);  **16** NetWorkProcessManager.message\_poll();  **17** return |

5.4.2 抽象服务类详细设计与实现

抽象服务类内部基于 rest\_rpc 框架实现了基于服务的通信方式， 与抽象发布 类和抽象订阅类不同，该类并不需要通信抽象模块的参与即可完成通信。

从图 5.10 中可以看出， ServiceAbstract 类存在 bind\_service 、unbind\_service、 call\_service 和 lookup\_for\_service 四个函数。 bind\_service 函数负责绑定用户服务 端提供的服务至本地 rpc 服务器 rpc\_server 中；unbind\_service 函数负责将用户需 要删除的服务从本地 rpc 服务器中解除绑定； call\_service 函数接收用户发出的服 务请求并通过 rpc\_client 请求服务； lookup\_for\_service 函数向服务发现模块查找 需要请求的服务的 IP 地址； rpc\_server 和 rpc\_client 在 ServiceAbstract 类中唯一 存在。

由于同一个端口只能被一个 RPC 服务器所监听， 而服务发现模块同样需要 rest\_rpc 实现服务发现功能。针对此问题， 本系统在设计时将同一个进程内的 ServiceAbstract 类和 ServiceDiscoverManager 类需要提供的 RPC 服务绑定至同一 个端口。 ServiceAbstract 类内部分为两个线程， 彼此之间异步地运行。一个线程 用户 rpc\_server 接受其他客户端的服务请求， 另一个线程用于 rpc\_client 发起服 务请求和执行 bind\_service、unbind\_service 、call\_service 和 lookup\_for\_service 四 个函数。 ServiceAbstract 类时序图如图 5.11 所示。

第 5 章 自动驾驶运行时通信系统详细设计与实现

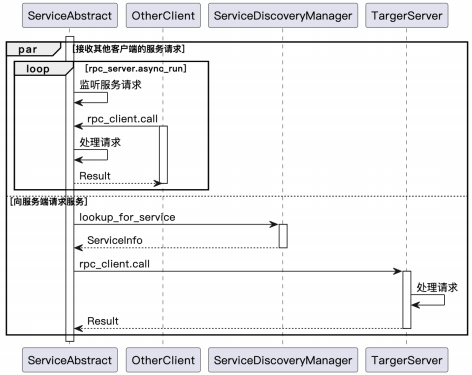


图 **5.11** 抽象服务类时序图

5.5 通信传输模块详细设计与实现

通信传输模块是本系统的核心模块， 基于发布-订阅的异步通信的实际消息 传输全部由本模块实现。根据概要设计的设计方案， 本节将进程内通信模块、进 程间通信模块、网络通信模块和消息模块进行详细设计与实现。

5.5.1 进程内通信模块详细设计与实现

根据概要设计中对本模块提出的设计方案， 本模块相关类设计如图 5.12 所 示。本模块包含进程内通信管理类 IntraProcessManager、进程内通信发布者类 IntraProcessPublisher 和进程内通信订阅者类 IntraProcessSubscriber 类。 IntraPro- cessManager 类控制 IntraProcessPublisher 类和 IntraProcessSubscriber 类的生命周 期。

第 5 章 自动驾驶运行时通信系统详细设计与实现

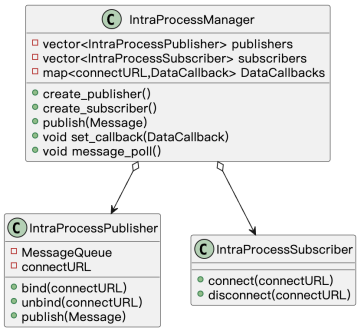


图 **5.12** 进程内通信模块类图

考虑到进程内通信并不需要严格将不同话题的发布者和订阅者区分， In- traProcessManager 类采用单例设计， 每个进程中只存在一个实例， 所有的进程 内通信发布者和进程内通信订阅者全部保存在 IntraProcessManager 类中。 Data- Callbacks 存放所有订阅者的回调函数， 根据 PubAbstract 类返回的 connectURL 完成对不同 Subscriber 类中的 DataCallback 进行区分。 create\_publisher 和 cre- ate\_subscriber 函数用于创建 IntraProcessPublisher 或 IntraProcessSubscriber 类。 publish 函数被对应的 PubAbstract 类调用发布消息。 set\_callback 函数被对应的 SubAbstract 类调用，IntraProcessManger 类将 Subscriber 类DataCallback 与 connec- tURL 保存至哈希表 DataCallbacks 中。 message\_poll 函数对每个 IntraProcessPub- lisher 类中的消息队列进行轮询， 若有新消息则根据 connectURL 找到对应 Sub- scriber 类的 DataCallback 执行回调函数， IntraProcessManager 类中开辟新线程执 行此函数。

IntraProcessPublisher 类中的 bind 函数将 IntraProcessPublisher 类中的 connec- tURL 更新为 PubAbstract 类返回的连接地址 connectURL；unbind 函数将类中的 connectURL 清空；publish 函数将待发布消息压入类中的消息队列MessageQueue。 IntraProcessSubscriber 类中的 connect 和 disconnect 函数不实现任何功能， 设计这 两个接口的目的是与 IntraProcessPublisher 类中的 bind 和unbind 函数相呼应。

IntraProcessManager 类中 message\_poll 函数对消息轮询的实现如算法[5.3](#_bookmark123)所 示。如上文所述， 该函数运行在新的线程中并以死循环的方式对消息进行轮询。 循环首先获取各 publisher 的 connectURL (第 3 行)，随后将 publisher 中消息队列

MessageQueue 中的消息存入临时队列 message (第 5 行)，随后将该 publisher 的 MessageQueue 清空(第 7 行)，接着在哈希表 DataCallbacks 中查询以 connectURL 键的值是否存在(第 8 行)，若存在根据键获取 DataCallback 并将临时队列所有 消息作为参数调用 DataCallback (第 9- 12 行)。

|  |
| --- |
| 算法 **5.3** 进程内通信消息轮询算法 |
| **Data:** 回调函数哈希表 DataCallbacks, 进程内通信发布者publishers  **1** **while** *true* **do**  **2** **for** *publisher* *in* *publishers* **do**  **3** connectURL = publisher.connectURL;  **4** **for** *message* *in* *publisher.MessageQueue* **do** **5**  messages.add(message);  **6** **end**  **7** publisher.MessageQueue.clear();  **8** bool find = DataCallbacks.find(connectURL);  **9** **if***find* *==* *true* **then**  **10** DataCallback = DataCallbacks[connectURL];  **11** **for** *message* *in* *messages* **do** **12**  DataCallback(message);  **13** **end**  **14** **end**  **15** **end**  **16** **end**  **17** return |

5.5.2 进程间通信模块详细设计与实现

为了改进本文第一章提到的现有自动驾驶通信系统的缺陷， 进程间通信模 块是本通信系统中最为复杂的一个模块。本模块对共享内存操作使用 POSIX 接 口(mmap 、shm\_open 等) 实现。本小节依次从共享内存区数据结构设计、进程 间通信模块类设计以及共享内存通信机制展开论述。

1. 共享内存区数据结构设计

本系统对共享内存区数据结构设计如图 5.13 所示， 共享内存区由控制和数 据两部分组成。控制部分保存了该共享内存区相关信息， 而数据部分存储实际的 通信消息。

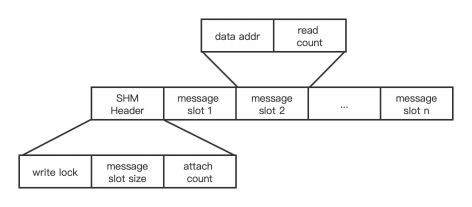


图 **5.13** 共享内存区数据结构

在共享内存区的控制部分 SHM Header 中， 由 write lock 、message slot size、 attach count 三个字段对共享内存区进行实际的控制。表 5.1 对 SHM Header 中三 个字段的定义和作用作出了说明。

表 **5.1** 控制部分参数说明

|  |  |  |
| --- | --- | --- |
| 字段名称 | 字段类型 | 字段作用 |
| write lock  message slot size  attach count | pthread\_mutex\_t  int  atomic\_int | 保护共享内存区同一时间只能被一个发布者写入消息  数据部分 message slot 的个数  连接至共享内存区订阅者的个数 |

共享内存区数据部分由多个 message slot 组成， message slot 内部由数据地址 data addr 和引用计数 read count 组成。表 5.2 对 message slot 中字段的定义和作用 作出了说明。

表 **5.2** 数据部分参数说明

|  |  |  |
| --- | --- | --- |
| 字段名称 | 字段类型 | 字段作用 |
| data addr read count | uint8\_t\* atomic\_int | 消息起始内存地址  当前正在读取消息的订阅者个数 |

2. 进程间通信模块类设计

根据概要设计中对本模块提出的设计方案， 本模块相关类设计如图 5.14 所 示。本模块包含进程间通信管理类 InterProcessManager、进程间通信发布者类 InterProcessPublisher、进程间通信订阅者类 InterProcessSubscriber、共享内存写 入类 ShmWriter 、共享内存读取类 ShmReader、共享内存区控制部分 SHMReader 类和共享内存区数据部分 MessageSlot 类。

第 5 章 自动驾驶运行时通信系统详细设计与实现

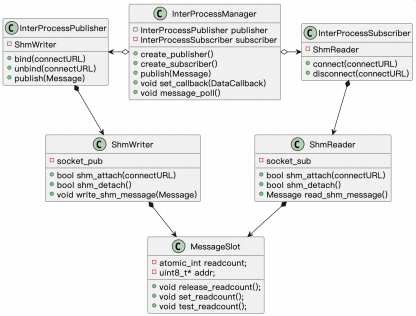


图 **5.14** 进程间通信模块类图

InterProcessManager 类的设计与进程内通信模块中 IntraProcessManager 类的 设计相同， InterProcessManager 类将不同话题下的发布者和通信严格区分， 该类 不再采用单例模设计， 而是根据实际不同话题下发布者和订阅者的数量进行实 例化。同时， InterProcessManager 类也不需要将 Subscriber 类中的 DataCallback 函数与 connectURL 一一对应并保存。

ShmWriter 和ShmReader 是本模块最重要的两个类， 实现了向共享内存写入 消息和从共享内存读取消息的功能。如算法[5.1](#_bookmark121)和算法[5.2](#_bookmark122)所示， 当抽象发布类和 抽象订阅类建立建立通信链路时， 会分别调用 InterProcessPublisher 中的 bind 函 数和 InterProcessSubscriber 中的 connect 函数，随后 ShmWriter 与 ShmReader 类被 创建并初始化。ShmWriter 类在shm\_attach 函数中根据 connectURL 创建共享内存 文件并在共享内存文件中初始化共享内存区数据结构， 将共享内存文件地址映射 至本进程的地址空间； ShmReader 类在 shm\_attach 函数中根据 connectURL 打开 ShmWriter 类创建的共享内存文件并将共享内存文件地址映射至本进程的地址空 间。ubind 和 disconnect 函数将 ShmWriter 与 ShmReader 删除， ShmWriter 和 Shm- Reader 通过 shm\_detach 将共享内存文件与本进程解除映射。 write\_shm\_message 函数用于 ShmWriter 向共享内存中写入用户需要发布的消息并使用网络通知各 订阅者读取消息； read\_shm\_message 用于 ShmReader 从共享内存中读取消息并 通过 Subscriber 类中的 DataCallback 函数将消息压入 Subscriber 类中的消息队 列供用户使用。 InterProcessManager 通过 message\_poll 函数开辟新线程并轮询 ShmWriter 的读取通知，调用 ShmReader 进行消息的读取。

SHMHeader 类是共享内存区控制部分的实现， 内部包含互斥锁 write\_lock、

message\_slot\_size 和 attach\_count 。write\_lock 用于保护共享内存区同一时间只能 被发布者占用写入消息， message\_slot\_size 为数据部分 message slot 的个数， at- tach\_count 用于记录连接至共享内存区订阅者的个数。

MessageSlot 类是共享内存区数据部分的实现， 内部包含一个指向共享内 存区域的数据首地址 addr 和控制message slot 中正在读取消息的订阅者个数的 readcount。该类中的 release\_readcount 函数在 ShmReader 类读取消息后被调用， 函数内将 readcount 原子地执行减 1 操作。 set\_readcount 和 test\_readcount 函数被 ShmWriter 类调用， set\_readcount 原子地将 MessageSlot 类中的 readcount 数值置 为 SHMReader 类中的 attach\_count，test\_readcount 用于检测是否有订阅者异常退 出导致 readcount 异常。

3. 共享内存地址规划

由于共享内存将被映射至多个进程的地址空间中， 共享内存数据结构中不能 含有指向某个进程的指针， 所以共享内存数据结构不能使用 C++ STL (标准模板 库) 中的容器是无法存放至共享内存中的， 因为 STL 的容器会在本进程的堆空间 中申请内存， 对于其他进程来说这些内存地址无法访问的。 C++ Boost 库(标准库 的扩展) 则提供了专为共享内存设计的容器， 重新设计了容器的空间分配器， 容 器可以被跨进程访问。但 Boost 库的语法比较复杂且源码体积高达 1G 以上， 程 序编译时间会明显增加， 本系统不采用 Boost 作为共享内存通信的实现手段。本 系统采用 C++ 中 placement new 的方式为共享内存数据结构分配内存， placement new 的作用是在某个指定的内存地址上创建相关变量。如图 5.15 所示， 共享内存 数据结构各字段从 mmap 返回的首地址开始紧密排放， 并且 message\_slot 中的消 息起始内存地址 addr 也全部为共享内存中的地址。不同进程只需要在shm\_attach 函数中通过 mmap 返回的共享内存首地址 addr 即可访问 SHM Header 和 message slot 中各变量。

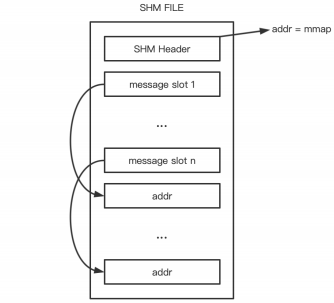


图 **5.15** 共享内存地址规划

4. 共享内存通信机制设计

发布者首次创建共享内存区过程如算法[5.4](#_bookmark124)所示。首先通过通信话题名称 topic\_name 获取用于共享内存通信的文件名称 shm\_file，使用 shm\_open 以及 mmap 函数创建共享内存文件(第 1-3 行)；随后使用上文提到的 C++ plancement new 依次将 SHMHeader、MessageSlot 和 data\_addr 创建(第 4-9 行)。对于 SHM- Header 类中类型为 pthread\_mutex\_t 的互斥锁write\_lock，创建过程中 SHMWriter 使用 pthread\_mutexattr\_setpshared 函数将互斥锁 write\_lock 的进程共享属性设置 为 PTHREAD\_PROCESS\_SHARED，使用pthread\_mutexattr\_setrobust 函数将互斥 锁 write\_lock 的健壮属性设置为 PTHREAD\_MUTEX\_ROBUST。设置后， 互斥锁 write\_lock 可以被跨进程使用并且可以保证健壮性。创建完成后， 共享内存中的 各部分地址排列如图 5.15 所示。

|  |  |
| --- | --- |
| 算法 **5.4** 创建共享内存区过程 | |
| **1** shm\_file = shm\_ipc+topic\_name; | |
| **2** fd = shm\_open(shm\_file); | |
| **3** addr = mmap(MAP\_SHARED,fd); | |
| **4** SHMHeader\* shm\_header = Create(addr,SHMHeader); | |
| **5** MessageSlot\* message\_slot =  Create(addr+Size(SHMHeader),MessageSlot[message\_slot\_size]); | |
| **6** **for** i ← 0 **to** *message\_slot\_size* **do** | |
| **7** | uint8\_t\* data\_addr = Create(addr+Size(SHMHeader)+Size(MessageSlot)\*message\_slot\_size |
| **8** | +data\_size\*i,data\_size); |
| **9** | MessageSlot[i].data\_addr = data\_addr; |
| **10** **end** | |

发布者向共享内存中写入消息过程如算法[5.5](#_bookmark125)所示。首先发布者请求控制部

分 SHM Reader 中的互斥锁 write\_lock 取得对共享内存区写入的权限(第 1 行)； 随后遍历数据部分 MessageSlot 数组查看是否存在某个 MessageSlot 的 readcount 为 0，若存在则对待发布消息 Message 进行序列化操作， 将 MessageSlot 中的addr 和 readcount 进行设置并获取序列化后消息的大小 size (第 2-9 行)；最后通过网 络方式将对应的 MessageSlot 下标 index 和消息大小 size 发送给订阅者通知其读 取消息，释放互斥锁 write\_lock (第 10- 11 行)。

|  |
| --- |
| 算法 **5.5** 进程间通信发布消息过程 |
| **Data:** 待发布消息 Message  **1** lock(write\_lock);  **2** **for** index ← 0 **to** *message\_slot\_size* **do**  **3** **if** *MessageSlot[index].readcount* *==* 0 **then**  **4** MessageSlot[index].addr = Serialize(Messsage);  **5** size = Size(Message);  **6** set\_readcount(MessageSlot[index].readcount);  **7** break;  **8** **end**  **9** **end**  **10** socket.send(index,size);  **11** unlock(write\_lock); |
| 订阅者从共享内存中读取消息过程如算法[5.6](#_bookmark126)所示。首先订阅者根据下标 in- dex 获取对应MessageSlot 中的数据首地址 addr (第 1 行)；随后通过首地址 addr 和数据大小 size 将序列化后的消息存入 buf 并进行反序列化(第 2-3 行)； 序列 化完成后将 readcount 原子地减 1 (第 4 行)；最后调用DataCallback 函数将消息 压入 Subscriber 类中的消息队列并激活调度任务(第 5 行)。 |
| 算法 **5.6** 进程间通信订阅消息过程 |
| **Data:** Socket Message(index,size)  **1** addr = MessageSlot[index].addr;  **2** buf = copy(addr,size);  **3** Message = DeSerialize(buf);  **4** release\_readcount();  **5** DataCallback(Message); |

当 多 个 发 布 者 中 的 一 个 或 多 个 由 于 异 常 退 出 无 法 正 常 释 放 互 斥 锁 write\_lock 导 致 其 他 发 布 者 无 法 再 次 取 得 互 斥 锁 的 情 况， 本 模 块 使 用 pthread\_mutexattr\_setrobust 函 数 将 互 斥 锁 write\_lock 设 置 为 PTHREAD\_MUTEX\_ROBUST 属性， 被多个发布者共享的互斥锁拥有健壮 性。 当某个发布者尚未释放获得的互斥锁异常退出后， 其他发布者使用 pthread\_mutex\_lock 函数再次尝试获取互斥锁时会返回 EOWNERDEAD 的错误 码， 只需要调用 pthread\_mutex\_consistent 函数恢复互斥锁的一致性即可正常使 用。

当订阅者发生异常退出无法正常释放 message slot 中的 readcount 时， 本模

块并不会立即采取恢复 readcount 一致性的操作。根据算法[5.5](#_bookmark125)发布者向共享内存 中写入消息会查找第一个 readcount 等于零的 message slot，当发布者无法找到 满足条件的 message slot 时， 会调用 test\_readcount 函数对每一个 message slot 中 的 readcount 进行检查。检查算法如算法[5.7](#_bookmark127) 所示。首先定义订阅者异常退出数 量 kdump\_count (第 1 行) ；随后遍历共享内存区数据部分中所有 MessageSlot 类， 使用原子操作 CAS 尝试将其中的 readcount 置 0，若尝试次数超过 10 次则 将该 readcount 与 kdump\_count 取较大值并赋值给 kdump\_dump (第 4- 10 行)；最 后使用原子操作将共享内存区控制部分 SHMHeader 中的 attach\_count 减去异常 退出的订阅者数量(第 12 行)。执行完test\_readcount 函数后， SHMHeader 中的 attach\_count 与实际连接至共享内存订阅者数量恢复一致。

|  |
| --- |
| 算法 **5.7** 订阅者崩溃检测 |
| **Data:** MessageSlot  **1** kdump\_count = 0;  **2** **for** index ← 0 **to** *message\_slot\_size* **do**  try\_times = 0;  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **while** *MessageSlot[index].readcount.CAS(0,0)* *not* *true* **do**  try\_times++;  **if** *try\_times* *>* *10* **then**  kdump\_count = max(kdump\_count,MessageSlot[index].readcount);  break;  **end**  **end**  **11** **end**  **12** SHMHeader.attach\_count.fetch\_sub(kdump\_count); |

5.5.3 网络通信模块详细设计与实现

5.5.4 消息模块详细设计与实现

5.6 服务发现模块详细设计与实现

服务发现模块是通信系统与中心节点交换网络信息的重要模块， 本模块与 中心节点通过 RPC 形式进行网络拓扑信息的交互控制通信链路并向用户提供查 询网络拓扑信息功能。

5.6.1 服务发现模块类设计

服务发现模块相关类如图 5.16 所示， 本模块类构成相对简单， 包含 Ser- viceDiscoveryManager 、rpc\_client 、rpc\_server 三个类。 rpc\_client 类和 rpc\_server 类由 C++ RPC 框架rest\_rpc 实现，提供最基本的服务绑定与请求功能。 ServiceDis- coveryManager 类采用单例设计模式， 使用rest\_rpc 框架完成与中心节点的网络 拓扑信息交互并控制通信链路， 所有函数均由 CommunicationAbstract 通信抽象

类和 ServiceAbstract 抽象服务类调用。 ServiceDiscoveryManager 类的内部保存本 进程内所有发布者、订阅者、服务客户端和服务服务端的信息， 即 PublisherInfo、

SubscriberInfo 、ServiceClientInfo 和 ServiceServerInfo。

ServiceDiscoveryManager 类被实例化时会调用自身内部的 start 函数将本模 块 RPC 通信地址(IP 地址与端口号) 通过 RPC 形式通知中心节点， 随后执行 init\_service 函数绑定服务接收来自中心节点的信息查询请求。初始化完成后 Ser- viceDiscoveryManager 类一方面接受来自 CommunicationAbstract 通信抽象类和 ServiceAbstract 抽象服务类的主动调用， 上传自身网络拓扑信息至中心节点； 另 一方面， ServiceDiscoveryManager 类接受来自中心节点的更新信息查询请求。

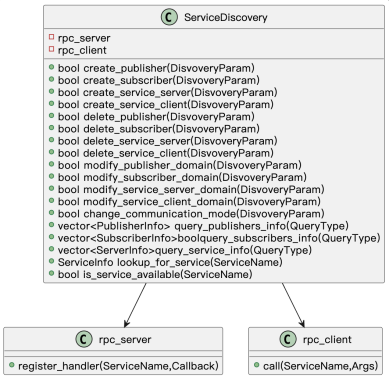


图 **5.16** 服务发现模块类图

5.6.2 发现机制设计

概要设计中， 服务发现模块发现机制分为主动发现与被动发现两类。主动发 现机制分为如下三情况：

1. 当注册、删除或修改发布者和订阅者时， ServiceDiscoveryManager 类构造 请求信息向中心节点发出 RPC 调用获取匹配通信方后， 根据自身保存的更 新通信链路回调函数 Callback，调用 PubAbstract 类和 SubAbstract 类中的 make\_communication\_connection 函数建立通信链路， 并将连接地址和订阅 者信息通过 RPC 方式发送至其他关联的服务发现模块。

2. 当注册、删除或修改服务服务端时， ServiceDiscoveryManager 类只向中心 节点更新服务端的 IP 和端口信息。

3. 当注册、删除或修改服务客户端时， ServiceDiscoveryManager 类不向中心节 点更新客户端的信息， 只有当客户端通过 ServiceAbstract 类发出实际请求 时 ServiceDiscoveryManager 类根据服务名向中心节点查询服务端的 IP 和 端口信息返回至 ServiceAbstract 类。

被动发现机制并不由用户的操作触发， 而是由 ServiceDiscoveryManager 类 在初始化中通过 rest\_rpc 注册供中心节点调用的服务。被动发现机制提供的服务 如下：

1. 更新发布者： 其他进程注册、删除或修改发布者时， 将创建的连接地址和订 阅者信息通过 RPC 方式发送本进程。 ServiceDiscoveryManager 类根据订阅 者信息中的话题名称 Topic 找到对应的 SubscriberInfo，通过其中的 Callback 调用 SubAbstract 类的 make\_communication\_connection 函数选择对应的通 信方式。

2. 更新订 阅者： 其他进程注册、 删 除或修改订 阅者 时， 中心节 点会将 其他进程服务发现模块的 RPC 地址和订阅者信息通过 RPC 方式发 送至本进程。 ServiceDiscoveryManager 类根据订阅者话题名称 Topic 找 到对应 的 PublisherInfo，通过其 中 的 Callback 调用 PubAbstract 类 的 make\_communication\_connection 函数选择对应的通信方式， 最后将连接地 址和订阅者信息返回至其他进程的服务发现模块。

以注册发布者为例， 服务发现模块与中心节点之间的信息流向如图 5.17 所

示。

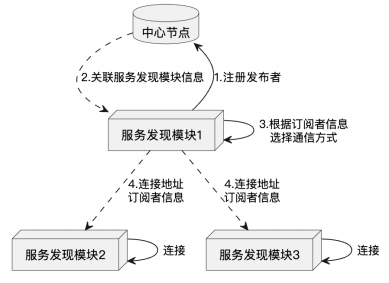


图 **5.17** 服务发现信息流向图

5.7 任务模块详细设计与实现

任务模块本身并不提供功能逻辑， 功能模块的作用是向用户提供一个统一 的基类， 用户需要继承基类并创建具体的任务类。根据概要设计中的设计方案，

任务模块类设计如图 5.12 所示。

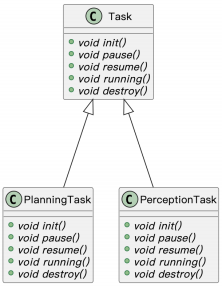


图 **5.18** 任务模块类图

任务模块的基类 Task 中的 init、pause、resume、running 和 destroy 五个函数 全部为纯虚函数， 内部不实现任何逻辑。用户需要定义自身的任务类如图 5.12 中的规划任务 (PlanningTask) 和感知任务 (PerceptionTask) 继承基类， 并实现五个 函数的逻辑功能。

5.8 调度模块详细设计与实现

调度模块使用基于消息触发和基于时间触发两种策略调度控制用户任务 的运行， 调度模块类图设计如图 5.13 所示。 Scheduler 类采用单例设计， 每个 进程中只存在一个实例。 TaskEntity 是调度实体， 由任务名 TaskName、任务回 调函数 TaskCallback 以及缓冲队列 Buffer 构成， TaskEntity 与调度任务一一对 应， 本系统规定 TaskEntity 中的回调函数必须是用户任务中的 running 函数。 cre- ate\_message\_task 与 create\_time\_task 函数分别向 Scheduler 类注册基于消息触发 的调度任务和基于时间触发的调度任务。 activate\_task 函数用于激活基于消息触 发的调度任务。 start\_schedule 函数用于开始任务的调度， 将 start 置为 true，所有 调度任务都需要 start 为true 时才会执行调度任务。

第 5 章 自动驾驶运行时通信系统详细设计与实现

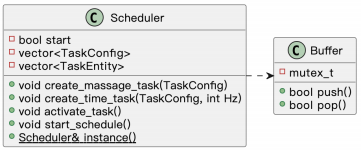


图 **5.19** 任务模块类图

5.8.1 基于消息触发的调度详细设计与实现

根据概要设计的设计， 基于消息触发的调度策略是侵入式的， 即该调度策 略并不需要用户手动地配置 TaskEntity 中的各字段， 而是在 Subsciber 类被实例 化时自动填充 TaskEntity 各字段并调用 create\_message\_task 函数注册调度任务。 注册调度任务成功后， 将 activate\_task 函数与 Subscriber 类的 DataCallback 回 调函数生成新的回调函数 DataCallback。当有新消息到来时通信新的回调函数 DataCallback 被执行， 函数中先将消息压入消息队列随后激活调度任务行用户任 务中的 running 函数。激活调度任务并不直接调用任务回调函数 TaskCallback，而 是将对应任务的缓冲队列 Buffer 中压入一个元素。

当调用 create\_message\_task 函数注册任务后， Scheduler 类会将每一个基于 消息触发调度策略的任务开辟一个新的线程， 线程中采用死循环判断缓冲队列 是否不为空， 若不为空则调用 TaskEntity 中的 TaskCallback 激活任务， 否则继续 判断直到缓冲队列不为空。每个调度任务运行在不同的线程中。

5.8.2 基于时间触发的调度详细设计与实现

基于时间触发的调度策略不再是侵入式， 而是需要用户手动设定 TaskEn-

tity 中各字段并设定调度频率， 调度频率单位是赫兹 (Hz)。当用户调用 cre- ate\_time\_task 函数后 Scheduler 类会开辟一个新的线程并开始以固定的频率执 行用户任务中的 running 函数。该调度策略最重要的是如何将用户任务中的 run- ning(TaskCallback) 函数尽可能贴合用户设置的调度频率。针对此问题， 本小节 给出的实现如算法[5.8](#_bookmark128)所示。 Scheduler 类的开辟新的线程后， 使用 C++ 中 ratio 库 计算调度的间隔(第 1 行)； 随后进入死循环， 获取执行 TaskCallback 函数之前 的时间点 start (第 3 行)； 当 TaskCallback 运行结束后再次获取时间点 end 并计 算运行耗时 cost (第 4-6 行)；随后计算需要等待的时间wait (第 7 行)；如果等 待时间小于 0 则直接进行下一次调度，否则线程睡眠(第 8- 11 行)。

|  |
| --- |
| 算法 **5.8** 时间触发调度算法 |
| **Data:** 调度频率 Hz  **13** **end**  **1** duration = ratio(1,Hz);  **2** **while** *true* **do**  clock start = now;  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  TaskCallback;  clock end = now;  clock cost = end - start;  clock wait = duration - cost;  **if** *wait* *<=* *0* **then**  continue;  **else**  sleep(wait);  **end** |

整个任务调度模块的线程运行如图 5.14 所示。消息调度线程只有当 Sub- scriber 类中的回调函数 DataCallback 被抽象订阅类调用是才会被激活并执行 TaskCallback，而时间调度线程以固定频率执行 TaskCallback。

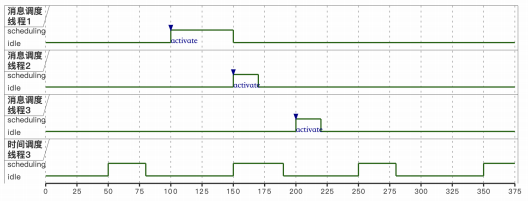


图 **5.20** 任务调度模块线程运行图

5.9 中心节点详细设计与实现

第 **6** 章 自动驾驶运行时通信系统测试与分析

6.1 测试环境

表 **6.1** 常见消息队列每秒收发消息数量对比**[**[**24**](#_bookmark110)**]**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 通信方式 | 消息大小 | 延迟统计 | | |
| 平均延迟 | 最小延迟 | 最大延迟 |
| 进程间通信通信 | 1KB  1  1  1 | 0.0513ms  1  1  1 | 0.0513ms  1  1  1 | 0.0513ms  1  1  1 |
| 进程内通信 | 1KB  1  1  1 | 0.0513ms  1  1  1 | 0.0513ms  1  1  1 | 0.0513ms  1  1  1 |
| 网络通信 | 1KB  1  1  1 | 0.0513ms  1  1  1 | 0.0513ms  1  1  1 | 0.0513ms  1  1  1 |

表 **6.2** **Comparison** **of** **different** **obfuscations** **in** **terms** **of** **their** **transformation** **capa-** **bilities**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| obfuscators | Transformations | | | | |
| Renaming | Dead code removal | control flow obfuscation | string encryption | code encryption |
| Proguard Allatori DashO Androcrypt | ✓  ✓  ✓  ✓ | ×  ×  ×  × | ×  ×  ×  × | ✓  ✓  ✓  ✓ | ×  ×  ×  × |

6.2 功能性测试

6.3 非功能性测试

参 考 文 献

[1] 李小喜. 无人驾驶汽车通信系统中的路由控制策略研究[D]. 安徽: 中国科学技术大学,

2019.

[2] 杨天悦. 自动驾驶商用化进程提速[N]. 北京日报, 2022-01-07(019).

[3] 徐名赫，朱雷. 自动驾驶出租车商业化现状及挑战[J]. 智能网联车, 2020(04): 40-47. [4] 张真齐. 让消费者尝鲜自动驾驶服务享道勾勒智慧出行蓝图[N]. 中国青年报, 2022-01-

27(T03).

[5] NISHIMURA K, ISHIKAWA T, SASAKI H, et al. Raplet: Demystifying publish/subscribe

latency for ros applications[C/OL]//2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA). 2021: 41-50. DOI: [10.1109/](https://doi.org/10.1109/RTCSA52859.2021.00013) [RTCSA52859.2021.00013](https://doi.org/10.1109/RTCSA52859.2021.00013).

[6] KRONAUER T, POHLMANN J, MATTHé M, et al. Latency analysis of ros2 multi-node

systems[C/OL]//2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI). 2021: 1-7. DOI: [10.1109/MFI52462.2021.9591166](https://doi.org/10.1109/MFI52462.2021.9591166).

[7] MARUYAMA Y, KATO S, AZUMI T. Exploring the performance of ros2[J]. 2016 Interna-

tional Conference on Embedded Software (EMSOFT), 2016: 1- 10.

[8] MORITA R, MATSUBARA K. Dynamic binding a proper dds implementation for optimizing

inter-node communication in ros2[C/OL]//2018 IEEE 24th International Conference on Em- bedded and Real-Time Computing Systems and Applications (RTCSA). 2018: 246-247. DOI: [10.1109/RTCSA.2018.00043](https://doi.org/10.1109/RTCSA.2018.00043).

[9] 奚美丽, 张远骏. 自动驾驶操作系统现状与发展趋势[J]. 汽车与配件, 2021(12): 8. [10] BAIDU. CyberRT Introduction[EB/OL]. 2019. <http://apollo.auto/cyber.html>.

[11] HUANG A S, OLSON E, MOORE D C. Lcm: Lightweight communications and marshalling

[C]//Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on. 2010.

[12] DANTAM N, STILMAN M. Robust and efficient communication for real-time multi-process

robot software[C/OL]//2012 12th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012). 2012: 316-322. DOI: [10.1109/HUMANOIDS.2012.6651538](https://doi.org/10.1109/HUMANOIDS.2012.6651538).

[13] AHMAD S, ARSHAD M J. Enhancing fast tcp’s performance using single tcp connection for parallel traffic flows to prevent head-of-line blocking[J/OL]. IEEE Access, 2019, 7: 148152- 148162. DOI: [10.1109/ACCESS.2019.2946527](https://doi.org/10.1109/ACCESS.2019.2946527).

[14] LIU W, GONG Y, WU H, et al. Memory-centric communication mechanism for real-time

autonomous navigation applications[C]//ICPP ’20: 49th International Conference on Parallel

Processing. 2020.

[15] BOSCH, ECLIPSE. Iceoryx[EB/OL]. 2019. <https://github.com/eclipse-iceoryx/iceoryx>. [16] CONTINENTAL. ecal[EB/OL]. 2021. <https://continental.github.io/ecal>.

[17] LIU W, WU H, JIANG Z, et al. A robotic communication middleware combining high per-

formance and high reliability[C/OL]//2020 IEEE 32nd International Symposium on Com- puter Architecture and High Performance Computing (SBAC-PAD). 2020: 217-224. DOI: [10.1109/SBAC-PAD49847.2020.00038](https://doi.org/10.1109/SBAC-PAD49847.2020.00038).

[18] GONG Y, WANG L, LIU W, et al. Poster: A light weight service discovery mechanism in

robot systems[C/OL]//2020 IEEE/ACM Symposium on Edge Computing (SEC). 2020: 178- 180. DOI: [10.1109/SEC50012.2020.00028](https://doi.org/10.1109/SEC50012.2020.00028).

[19] 蒲凤平, 陈建政. 基于 ZeroMQ 的分布式系统[J]. 电子测试, 2012(7): 7.

[20] 沙卫平, 芮挺, 高琦煜, 等. 基于 ZeroMQ 的多线程编程程序设计[J]. 江苏科技信息, 2017

(27): 3.

[21] 杨曜旗, 张磊. 基于 ZeroMQ 的分布式通信开发平台的设计与实现: 第 8 册[Z]. 2022.

[22] 王鹏, 邓芳. 网络视频监控系统消息中间件的研究与实现[Z]. 2015.

[23] ESTRADA N, ASTUDILLO H. Comparing scalability of message queue system: Zeromq

vs rabbitmq[C/OL]//2015 Latin American Computing Conference (CLEI). 2015: 1-6. DOI:

[10.1109/CLEI.2015.7360036](https://doi.org/10.1109/CLEI.2015.7360036).

[24] HADLOW M. Message queue shootout[EB/OL]. 2021. [http://mikehadlow.blogspot.com/20](http://mikehadlow.blogspot.com/2011/04/message-queue-shootout.html) [11/04/message-queue-shootout.html](http://mikehadlow.blogspot.com/2011/04/message-queue-shootout.html).

[25] 张杰. 基于 Protocol Buffers 的远程抄水表数据传输系统[D]. 华中科技大学, 2017.

[26] 隋心怡, 王瑞刚, 梁小江. 基于 Google Protocol Buffer 的即时通讯系统设计[J]. 电子科技,

2017, 30(1): 4.

[27] MAEDA K. Performance evaluation of object serialization libraries in xml, json and binary

formats[C/OL]//2012 Second International Conference on Digital Information and Communi- cation Technology and it’s Applications (DICTAP). 2012: 177- 182. DOI: [10.1109/DICTAP](https://doi.org/10.1109/DICTAP.2012.6215346) [.2012.6215346](https://doi.org/10.1109/DICTAP.2012.6215346).

[28] 音视频流媒体技术. XML ，JSON ，PROTOBUF 序列化协议[Z].

[29] ANDREW, D., BIRRELL, et al. Implementing remote procedure calls[J]. Acm Transactions

on Computer Systems, 1984.

[30] Toward the marriage of xml and mobile agents: Utilizing xml-rpc as a migration media[C]//

IASTED international conference on parallel and distributed computing and systems;PDCS 2009. 2011.

[31] SAMSEL C, GöKAY S, HEINIZ P, et al. Web service to json-rpc transformation[C]//8th

International Conference on Software Engineering and Applications. 2013.

[32] WEN X, MENG F, GAO L, et al. Research on distributed databse system based on remote

method call[C/OL]//2010 3rd International Conference on Information Management, Innova- tion Management and Industrial Engineering: Vol. 1. 2010: 62-65. DOI: [10.1109/ICIII.2010](https://doi.org/10.1109/ICIII.2010.20) [.20](https://doi.org/10.1109/ICIII.2010.20).

[33] WANG Y P, TAN W, HU X Q, et al. Tzc: Efficient inter-process communication for robotics

middleware with partial serialization[C/OL]//2019 IEEE/RSJ International Conference on In- telligent Robots and Systems (IROS). 2019: 7805-7812. DOI: [10.1109/IROS40897.2019.896](https://doi.org/10.1109/IROS40897.2019.8968462) [8462](https://doi.org/10.1109/IROS40897.2019.8968462).

[34] DANTAM N, STILMAN M. Robust and efficient communication for real-time multi-process

robot software[C]//IEEE-RAS International Conference on Humanoid Robots. 2012.

致 谢

在研究学习期间， 我有幸得到了三位老师的教导， 他们是： 我的导师， 中国 科大 XXX 研究员， 中科院 X 昆明动物所马老师以及美国犹他大学的 XXX 老师。 三位深厚的学术功底， 严谨的工作态度和敏锐的科学洞察力使我受益良多。衷心 感谢他们多年来给予我的悉心教导和热情帮助。

感谢 XXX 老师在实验方面的指导以及教授的帮助。科大的 XXX 同学和 XXX 同学参与了部分试验工作，在此深表谢意。