

针对交叉路口场景的安全车队管理协议 (针对交叉路口场景的车队安全协同系统)

所在赛道与赛项：A-ICV

目录

1.目标问题与意义价值	4
2.设计思路与方案	8
2.1 系统总体结构	8
2.2 基础机动操作安全设计	9
2. 2. 1 合并与分裂	9
2. 2. 2 车队成员离队与加入	14
2. 2. 3 变道	16
2.3 针对红绿灯路口场景的车队管理协议	16
2. 3. 1 红绿灯路口场景描述	16
2. 3. 2 最优化车队速度轨迹	18
2. 3. 3 最优化车队长度	22
2.4 基于国密的群密钥车队通信方案	25
2. 4. 1 车队群加密方案概述	25
2. 4. 2 车辆实体向 CA 申请证书	27
2. 4. 3 群密钥分发与管理	27
2. 4. 4 安全性分析	30
3.实现方案	32
3.1 总体框架	32
3. 1. 1 VENTOS	32
3. 1. 2 OMNeT++端	34
3. 1. 3 SUMO 端	38
3.2 车队机动的实现	40
3. 2. 1 车辆节点应用层的实现	40
3. 2. 2 新增消息 PlatoonMsg	41
3. 2. 3 算法具体实现	42
3.3 红绿灯路口处协议实现	44
3. 3. 1 新增消息 PltInfo 与 PltCtrl	45
3. 3. 2 vehicle 应用层	45
3. 3. 3 RSU 应用层	47
3.4 加密与认证的实现	49
3. 4. 1 生成密钥与证书	49
3. 4. 2 新增消息 KeyMsg 与信号 Signal	50
3. 4. 3 SM2 身份认证与加解密	53
3. 4. 4 群密钥分发与加密的测试	55
4.运行效果与性能分析	57
4.1 基础机动操作例子展示	57
4.2 红绿灯路口场景例子展示	57
4. 2. 1 go_stage 中的绿灯到达例子展示	57
4. 2. 2 wait_stage 中的绿灯到达例子 1 展示	59
4. 2. 3 wait_stage 中的绿灯到达例子 2 展示	61
4. 2. 4 go_stage 中的红灯到达例子展示	63
4.3 车队群密钥方案演示	64
4.4 模拟结果性能分析	66

4.4.1 车队在路口处行为分析	67
4.4.2 车队在路口处性能与指标分析	69
5.创新与特色	73

1. 目标问题与意义价值

我国经济在持续高速发展，机动车保有量继续保持着快速的增长态势。公安部最新数据现实，截至到 2023 年 3 月底，全国汽车保有量达到 4.2 亿辆。全国一共有 84 个城市的汽车保有量超过百万辆，其中北京、成都、重庆、上海超过 500 万辆，苏州、郑州、西安、武汉超过 400 万辆。不断增加的汽车保有量一方面给我国城市带来了严重的交通拥堵和交通事故，另一方面使得我国石油对外依存度进一步攀升，因汽车尾气导致的环境污染进一步加剧。而特别地，信号控制交叉口也是城市交通系统的瓶颈，对于信号控制的交叉口，更容易出现拥堵、高延迟、高排放等问题。

问题 1：如何缓解城市交通拥堵与交通事故问题？如何减少汽车尾气排放量？

在这样的大背景下，人们亟需寻找解决方案。最直接地想要解决这个问题的方法就是使用自动驾驶技术。然而，目前的大部分自动驾驶方案都聚焦于单一车辆的自动行驶。不可否认单一车辆的自动驾驶在一定程度上带来了便利性，但是它并不能有效地解决现阶段交通的主要问题，如交通拥堵、行驶安全和环保污染等。因为单车行驶无法充分利用道路资源，交通吞吐量受到严重限制，行驶安全性也相对降低。此外，车辆行驶过程中频繁的速度变化会增加燃油消耗量和尾气排放量，从而进一步导致对环境造成负面影响。

本作品将利用车辆编队行驶来解决这一问题。将单一车辆汇总为一个车队 (Platoon)，整个车队进行集体地行驶、停靠、加速、减速、变道等等。与此同时，车联网 (V2X) 技术、车对车通信技术 (V2V) 和车辆与路边基础设施通信技术 (V2I)，协同车辆基础设施系统 (CVIS) 等技术的高速发展也为车辆编队行驶带来了发展的可行性支撑。总结来说，车辆编队行驶就是将公路中的一些智能网联车利用 V2X、V2V、V2I、CVIS 等技术将单一车辆组织成一组紧密跟随的车队，编队行驶使车辆能够比具有相同速度的普通车辆保持更小的车间距离，即图 1-1 中的 “d”，从而提高交通吞吐量和同质性，此外，由于碰撞中速度变化小且冲击速度相对较低，安全性得到提高。

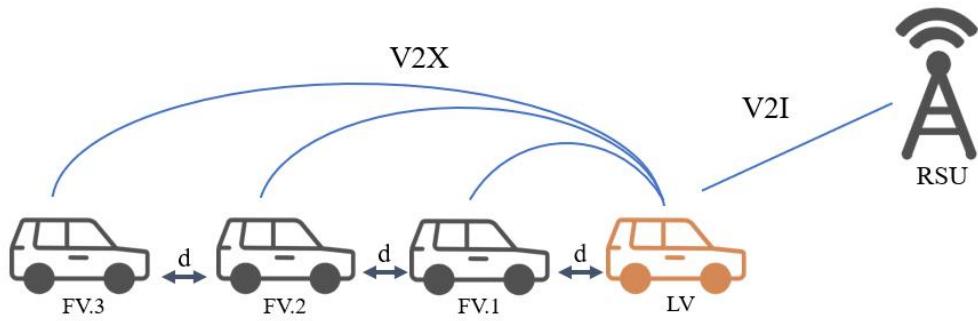


图 1-1 车辆编队行驶示意图

问题 2：如何设计安全的车队机动方法？如何完成车队的合并、分裂、成员离开或加入等？

然而车辆编队行驶需要解决的不仅仅是保持一个较小的车间距离进行编队行驶，车队还需要能够处理一些机动操作，如合并、分裂、变道、加入、离开车队等，以确保在更加复杂的交通情况中，如交叉路口的场景（如图 1-2），车队能够做出正确安全的行为。

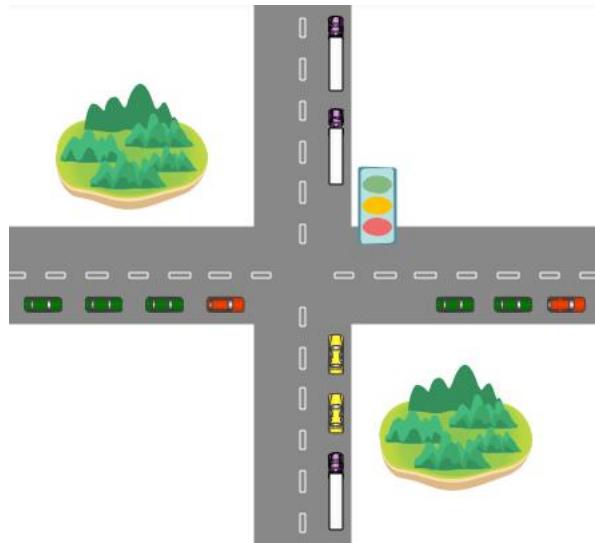


图 1-2 车队经过交叉路口

我们团队完成了对基础机动操作的安全设计，一共分为合并、分裂、变道、车队车头离开、车队成员离开、成员入队六个基础机动，对每个机动的执行步骤进行了完整的方案设计，包括如何在两个或者多个车队之间进行机动请求、机动响应以及机动执行等操作。在完成方案设计之后，在仿真平台上进行实验，以确保这六个基础机动能够使得车队编排行驶更为完整、智能。

问题 3：如何针对信号灯控制的交叉路口交通场景设计车队管理协议？来实

现车队安全、高效地通过红绿灯交叉路口？

在完成基础机动操作之后，车队已经可以作为一个初级智能模块化整体，能够解决一些初步的交通拥堵、交通事故问题。然而为了能够在城市道路行驶中更为智能，解决一些城市交通系统的瓶颈问题，例如信号控制交叉口场景问题，我们仍然需要设计一些基于场景的安全管理协议。

对于信号控制的交叉口，提高交叉口效率的方法有两种：第一类方法是根据交通估计结果，通过调整红绿灯来提高交叉口效率，一些现有的系统，如悉尼协调自适应交通都属于这一类。与第一种方法不同，第二种方法称为车队的协同驾驶，通过改变车队中车辆的移动计划来提高交通效率。在车队协同驾驶过程中，可以使得车辆在最短的时间内通过交叉口。

本作品采用 GLOSA (Green Light Optimal Speed Advisory) 算法为车队在交叉路口处进行速度轨迹规划，同时也计算出车队通过红绿灯的最佳长度，避免车队过长无法全部通过以及多个短车队在路口等待的情况出现。具体地，加入红绿灯车队通信方式以及车队协同驾驶，在靠近交叉路口场景时，为车头寻找推荐速度与推荐加速度以及最佳长度，使车辆在绿灯阶段以较小的延迟进入交叉口。速度轨迹规划可以尽量减少车头到达路口的时间，同时避免在路口长时间停车，减少气体排放，从而提高交通效率。由于车头前方附近没有其他车辆，所有的车队跟随者都应该跟随车头的轨迹，因此将 GLOSA 算法应用于车头轨迹优化是合理的。通过轨迹优化，车头可以带领整个车队从绿灯阶段开始以最小的延迟进入路口，同时避免在路口停车，减少气体排放。

问题 4：如何在上述的车队管理协议中保证车队内部通信的安全性与隐私性？如何设计一个适用于车队的安全通信方案？

最后，车队协同中数据的安全性和隐私保护无疑是至关重要的环节。然而，由于无线信道的开放和广播特性，更新车队中驾驶状态的车辆通信可能会遭受各类攻击，如窃听、重放攻击，以及恶意车辆节点的入侵。

车队是一个动态变化的结构，因此，针对单车通信的 V2V、V2I 的安全通信方案并不能直接适用于车队的场景。这样的状况导致了车辆的行驶信息、位置数据等敏感信息在无线通信过程中没有得到充分保护，极有可能被第三方无良收集并用于不正当用途，这对于用户的隐私权利无疑是一大威胁。因此，我们需要一

种更为有效、针对性强的方案，来确保车队中的数据通信安全，保护用户的隐私。

本作品利用基于国密的群加密方法来保护车队的通信安全问题。在车队场景中，群加密能够提供重要的安全保障和高效的通信管理。通过使用群密钥来确保只有车队的成员能够解读车队内部的通信，这可以防止敏感信息（例如路线、速度、位置等）被未授权的第三方获取，从而提高了车队的安全性。另外，通过中心化管理群密钥，我们可以在车队成员变动时及时更新群密钥，当车辆加入或离开车队时，由管理员更新并分发新的群密钥，这大大简化了车队管理的复杂性。

综上所述，在现实世界中，交通拥堵、交通事故、有害气体排放等问题为交通通行效率、人员安全、能源消耗等方面带来了很大的压力，同时给车辆运行性能、多车交互模式、交通流态势认知等方面带来了新的挑战。本系统拟在解决上述所提到的四个问题，利用车队解决交通运行效率低，安全性差以及能源消耗量高等问题；利用基础机动的设计完成车队编排行驶的纵向控制功能，构成一个完整的车队编排系统；制定合理的安全管理协议来解决交叉路口场景的车辆冗余、效率优化问题；利用国密算法完成车队在数据加密、身份认证、数据完整性和密钥交换等方面的安全通信，有效抵御窃听、篡改、伪装和重放攻击等威胁，确保车队通信的安全性。

2.设计思路与方案

2.1 系统总体结构

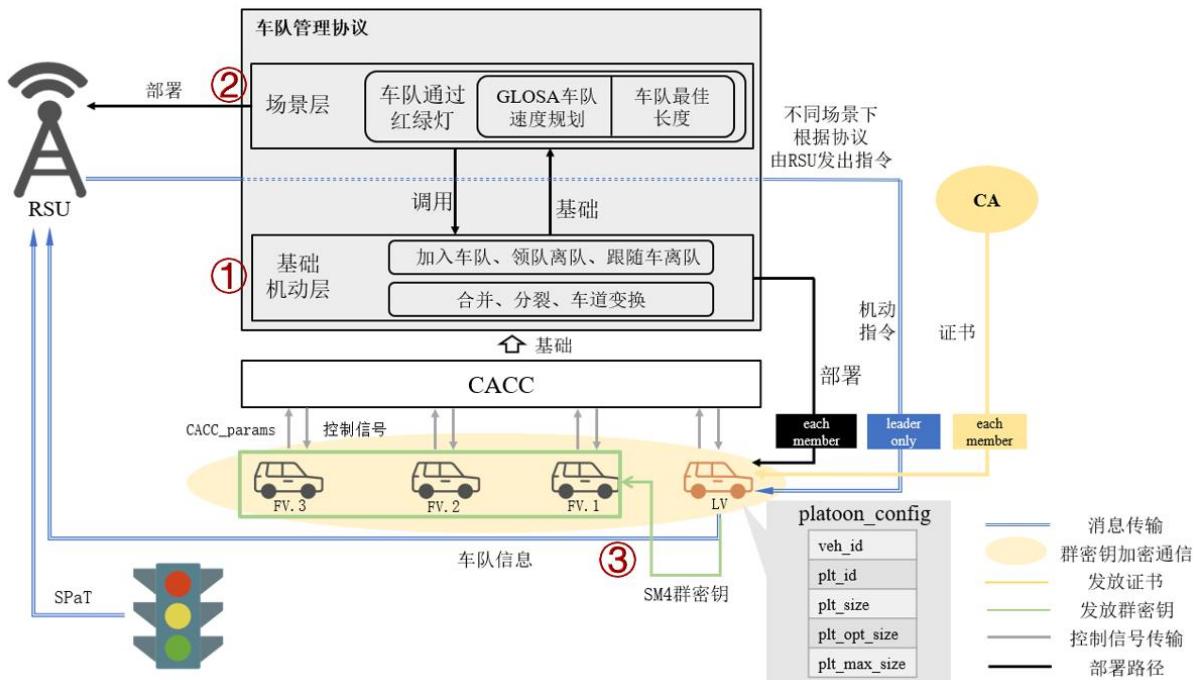


图 2-1 系统结构图

本作品拟按如下系统结构进行实现：

拟实现**针对交叉路口场景的车队安全协同系统**。如图 2-1 所示，该系统主要包含了三个部分，分别由红色圆圈数字标注。车队管理协议包括了两层，①基础机动层设计了车队的基础机动，负责安全的队形变换，②场景层针对具体的红绿灯路口场景完成了速度规划与最佳长度规划两个方面内容，③系统中的安全通信部分采用群密钥的方案解决车队内的安全通信问题。

车队的纵向控制是维护编队行驶的基础，我们采用了较为成熟的 CACC 控制策略，其基于 VANET 通信根据整个车队中其他车辆的移动状态信息动态调整车辆的速度、加速度以保持一定的队内车间距。**基础机动层**基于 CACC 实现了车队的基础机动，包括分裂、合并、换道以及基于以上三种机动的单车入队和单车离队机动，该层部署在智能网联车上，通过定义多种机动消息，基于 V2V 的通信方式实现安全的机动操作。**场景层**是基于基础机动层的，该层部署在路侧设备 RSU 上，针对常见的城市交通场景：车队通过红绿灯，RSU 收集车队的基础

信息，主要包括车队的速度与加速度信息、与路口的距离等，结合红绿灯的 SPaT(Signal Phase and Time)信息，计算车队的推荐速度轨迹和最佳车队长度，将其封装在消息中发送给车队队长，实现车队的管理，最终达到车队安全、高效地通过红绿灯目的，尽量减少车队在路口处的停车频率，增加城市交通的吞吐量。系统的安全通信部分，设计了基于国密的群密钥方案，车辆向 CA 申请 SM2 公钥证书，由群密钥管理员，也就是每个车队的队长，根据证书进行身份验证，验证通过后安全地为车队成员分发 SM4 群密钥，车队成员可以使用群密钥进行安全通信，该方案需要支持车队成员改变后群密钥的更新，以确保车队内通信的完整性与安全性。

2.2 基础机动操作安全设计

本节将阐述车队管理协议中执行的六种基本机动，即合并、分裂、变道车头离开、车队成员离开以及成员入队。我们设计的车队是中心化的管理方式，即车队队长拥有车队的基本信息，如队员信息、最佳长度、最大长度等，而队员是没有这些信息的，且一般由车队队长发起一个机动。每个车队机动都是通过相邻车辆或车队队长之间的结构化信息交换来协调的。这些消息交换可以使用限定状态机(FSM)精确地定义，该限定状态机在面对发生的不同事件时会经历一系列状态变换，在状态机的响应上一次只允许一次机动。

2.2.1 合并与分裂

合并机动指的是在同一车道上行驶的两个车队合并成一个车队。在预备进行合并操作时，首先判断车队的现有规模是否小于车队的最优规模，如果符合这个条件，我们规定合并机动总是由后车队的队长发起。假设我们有 A 车队和 B 车队，如图 2-2 所示。车队 B 就是发起机动请求的后排车队，由 4 辆车组成，而车队 A 是被发起机动请求的前排车队，由 3 辆车组成。

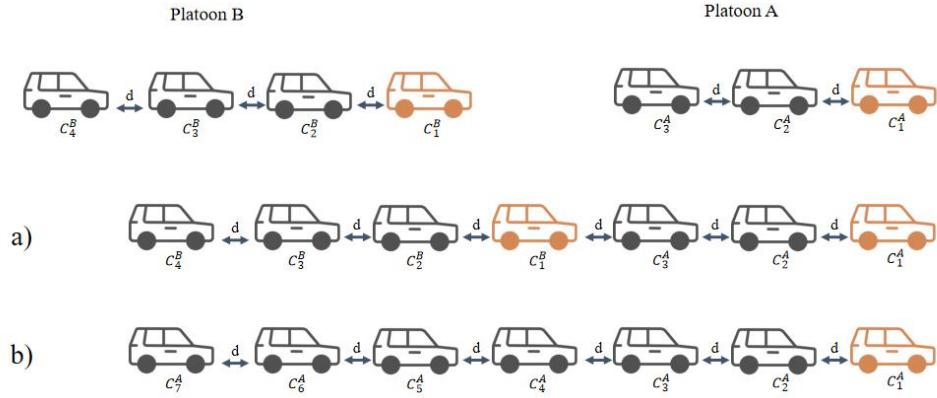


图 2-2 合并演绎图

为了便于阐述，我们设车队的最佳长度为 8。以下步骤显示如何执行合并操作：

(1) 合并请求: C_1^B 从前面的一个 A 车队成员那里收到信标信息，因为 C_1^B 的规模小于最优规模，所以可以发起合并机动。从 Beacon(信标信息)中提取前方车队的车队队长号(即 C_1^A 的车号)，并向前方队长(C_1^A)发送单向传播信号 MERGE_REQ。

(2) 合并响应:前排队长 C_1^A 可以选择“接受”或“拒绝”合并请求。如果 C_1^A 忙于执行其他的操作，则可以发送 MERGE_REJECT 来拒绝或延迟合并操作。 C_1^B 可以选择稍后再次发送 MERGE_REQ。如果 C_1^A 现在可以接受请求，并且满足最终车队的规模不超过最优队列规模的条件，此时 C_1^A 接受合并请求。

(3) 合并执行:在接收到 MERGE_ACCEPT 后， C_1^B 将其“time-gap”时间间隔缩小到“intra-platoon spacing”车内时距，以便加速赶上前面的车队(图 2-2a)。 C_1^B 向其所有车队成员发送 CHANGE_PL，将所有车队成员的队长(Platoon leader)替换为 C_1^A (图 2-2b)。至此所有车队成员开始听从 C_1^A 。最后， C_1^B 发送一个 MERGE_DONE 给 C_1^A ，并将其状态从 leader 变为 follower。

合并机动的状态机如图 2-3、2-4 所示，图 2-3 表示的是车队 B 在执行合并操作时，需要完成的机动流程图，图 2-4 表示的是车队 A 在执行合并操作后需要完成的机动流程图。

图中角上有一个黑色小三角形的状态是暂时状态，此时状态机花费的时间为零。

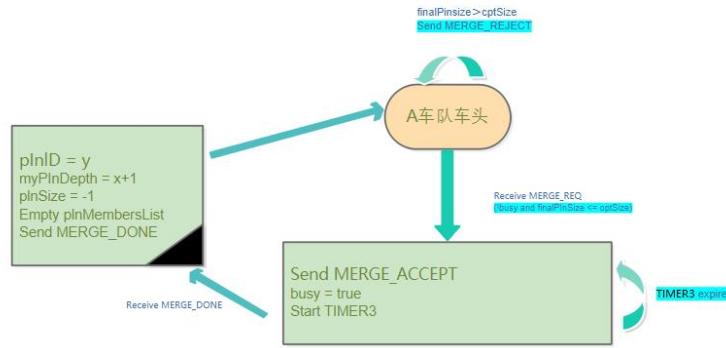


图 2-3 合并机动_车队 A 图

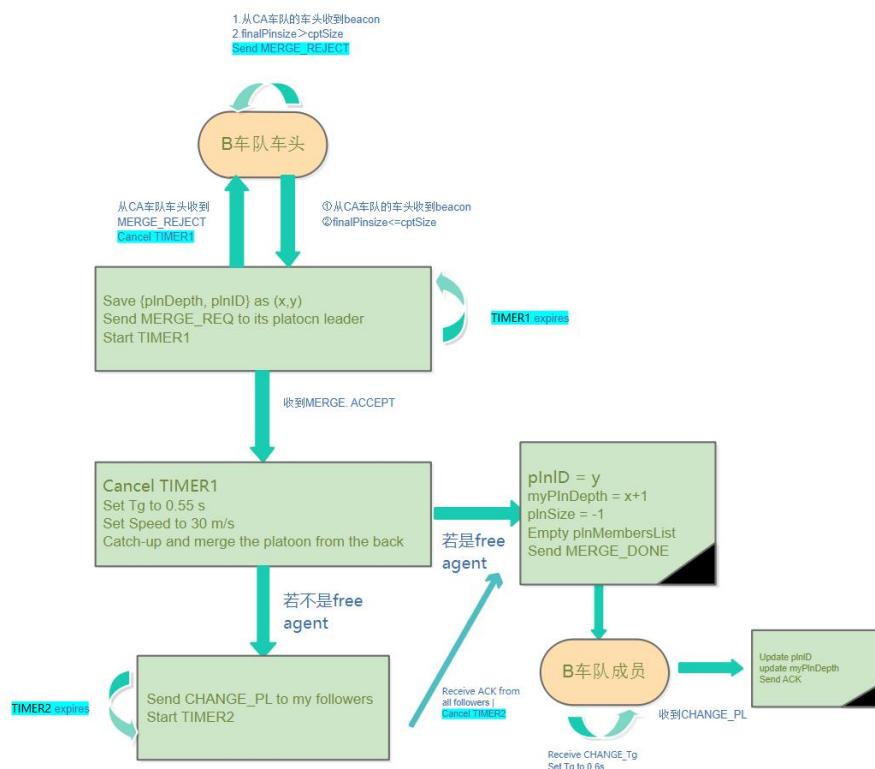


图 2-4 合并机动_车队 B 图

分裂机动指的是一个车队(至少有两辆车)在特定位置进行分开，形成两个较小的车队。与合并机动类似，分裂机动总是由车队队长 (Platoon leader)发起。当车队的规模超过最优队列规模时，可以使用分裂机动将车队分成两个较小的部分已完成后续的动作。假设我们有一个规模为 7 的车队，我们想把它分成规模为 4 的 A 车队和规模为 3 的 B 车队，如图 2-5 所示。以下步骤显示了如何进行车队分割机动：

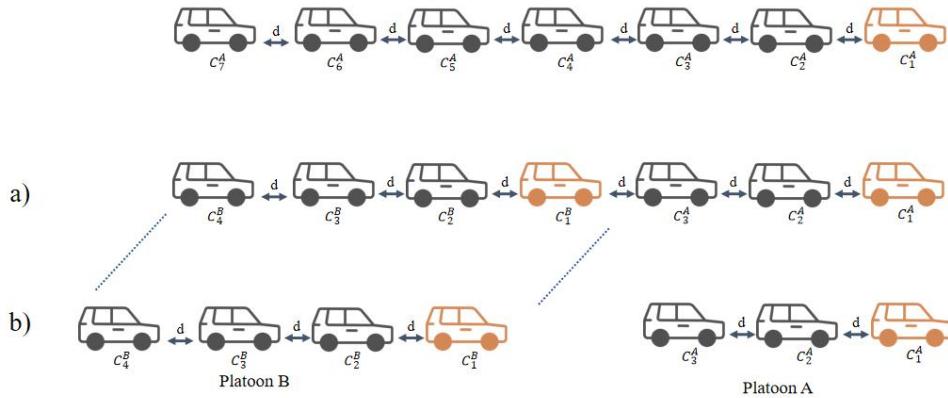


图 2-5 分裂演绎图

(1) 分裂请求:队长 C_1 通过向想要进行成员的成员车辆(上面示例中的 C_5)发送 SPLIT_REQ 消息来启动分裂机动。

(2) 分裂响应:在接收到分裂请求之后, C_5 可以选择接受或拒绝分割请求。若 C_5 选择拒绝, 则发送 PLIT_REJECT 消息, 并可以在其中包含拒绝分割操作的原因。若 C_5 选择接受, 这时候继续等待 leader 的后续命令, 而不允许擅自减速。如果 C_5 减速, 则会导致其跟随车(C_6 和 C_7)切换为避碰模式以防止追尾, 这是不可取的。

(3) 分裂执行:队长 C_1 向 C_5 发送单播消息 CHANGE_PL, 并使其成为暂时的自由代理车辆(没有加入任何车队的车辆被称为自由代理车辆)(图 2-5a)。现在我们拥有两个领导者: C_1^B 和 C_1^A 。 C_1^A 向 C_1^B 后面的所有追随成员发送一个多播消息 CHANGE_PL, 并要求他们将他们的车头更改为 C_1^B 。 C_1^A 通过发送 SPLIT_DONE 给 C_1^B 报告分裂完成。现在 C_1^B 可以安全减速并保持 Interplatoon_spacing 车队间距(图 2-5b)。

分裂机动的状态机如图 2-6、2-7 所示, 图 2-6 表示的是车队 A 在执行合并操作时, 需要完成的机动流程图, 图 2-7 表示的是车队 B 在执行合并操作后需要完成的机动流程图。

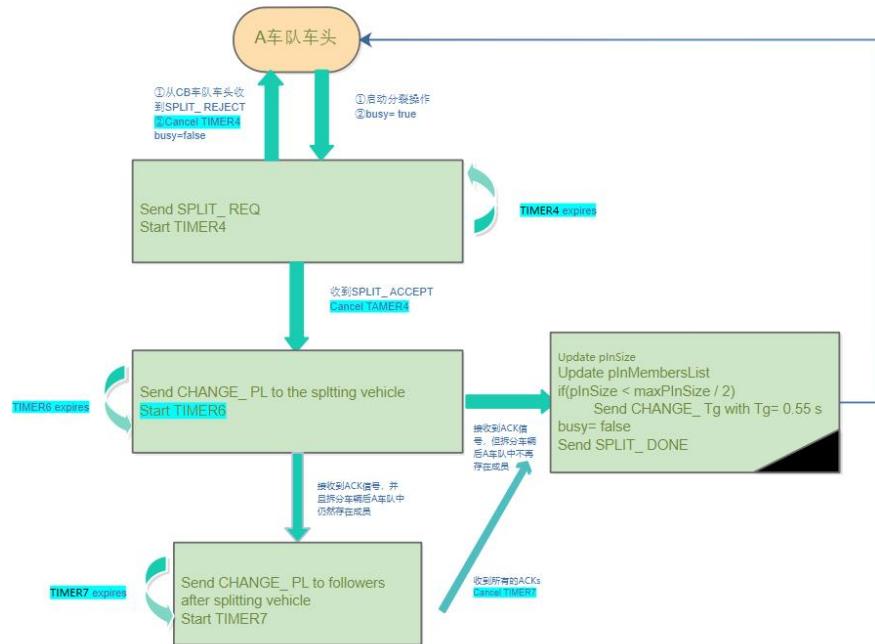


图 2-6 分裂机动_车队 A 图

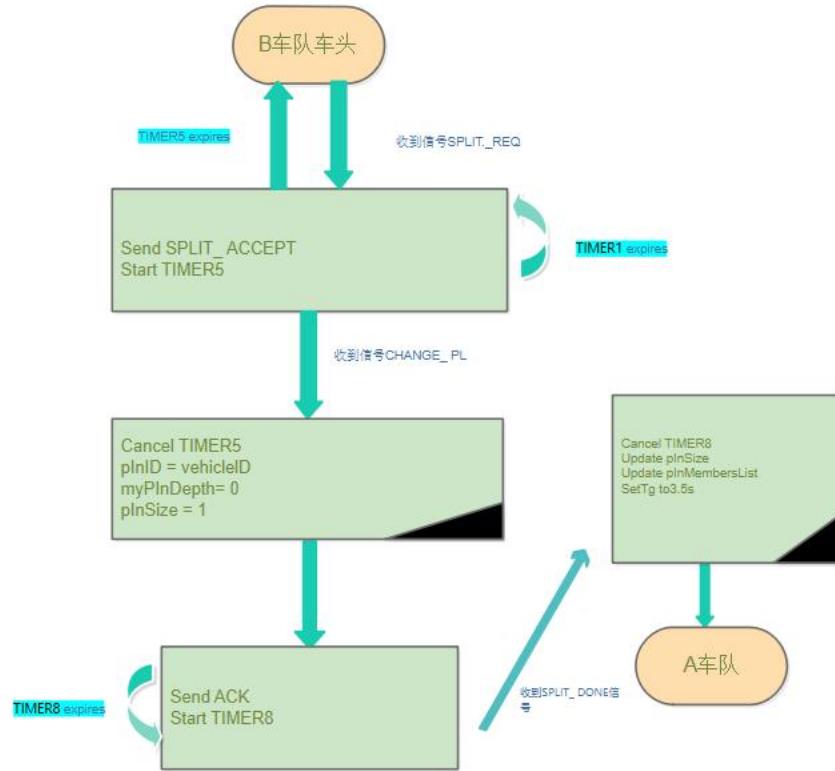


图 2-7 分裂机动_车队 B 图

2.2.2 车队成员离队与加入

I. 成员离队

车队成员离队包括了车队队长离队与跟随者离队两种情况，需要用不同的状态机和流程进行完成。

当队长需要离开车队时，它启动车队队长离开机动。如果车队规模大小为1，那么队长可以自由离开。在其他情况下，在队长离开车队之前，车队队长应该联系一个接班人来接替作为新队长。下面的步骤展示了车队队长离开机动是如何执行的：

(1) 离开公告:VOTE_LEADER 消息由队长发送给其所有车队成员，让其投票决定新的队长角色。车队成员们可以通过运行分布式领导选择算法投票选出新的车队队长。新当选的队长使用 ELECTED_LEADER 消息向现任队长宣布这一消息。至少有一个车队成员应该回应，否则旧队长将重新发送 VOTE_LEADER。为简单起见，我们假设车队中的第二辆车将接管领队角色。

(2) 离开执行:队长发起一个分裂机动，使自己成为一个自由代理车辆，从而安全退出车队。

当车队跟随者需要离开车队时(例如，车辆马上接近其目的地)，它启动车队成员离开机动，然后将控制权交给驾驶员以改变车道。每次只允许一辆车离开车队。车队成员离开车队的主要问题是需要其在跟随车辆的前面(或后面)创造足够的空间，以便使变道成为可能。车队成员离开机动可以使用一系列分裂和合并机动，如下所示:

(1) 离开请求/响应:车队成员 C_i 通过发送 LEAVE_REQ 通知车队队长。车队队长可以发送 LEAVE_REJECT 来拒绝这个请求，也可以在下一步执行一到两次分裂机动，使 C_i 成为自由代理车辆。

(2) 离开执行:如果 C_i 是车队中最后一辆车，一个分裂机动足以使 C_i 成为自由代理车辆。现在 C_i 可以进行车道变换和退出车队了。另一方面，如果 C_i 不是最后一辆车，则执行如图 2-8 所示的两次拆分加一次合并机动。

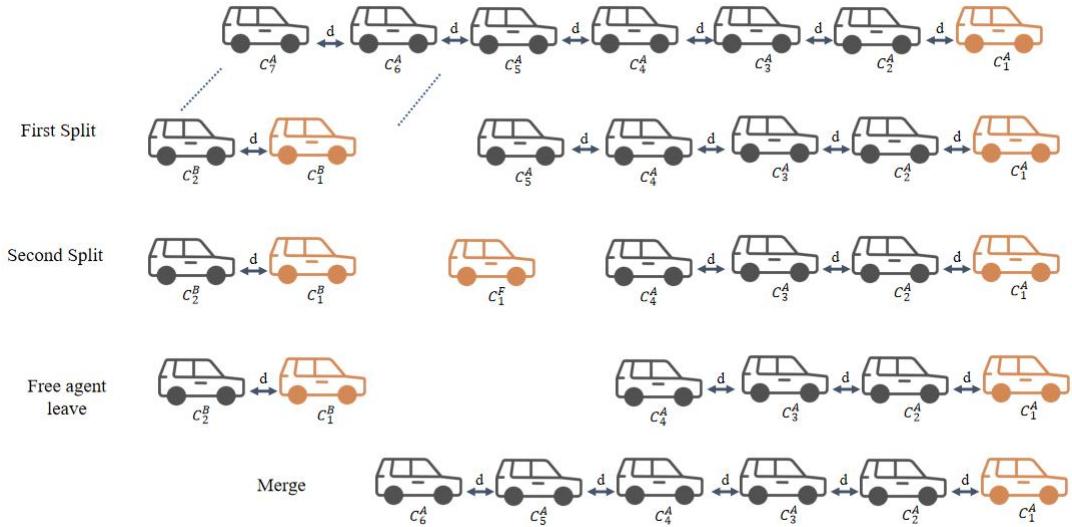


图 2-8 车队成员离开机动图

II. 成员加入

成员入队机动的操作应该是安全和平稳的。在成员入队操作上，有两种方法，第一种是将车辆驶入专用车道的操作留给驾驶员，然后将控制权交给车队管理协议。这种方法依赖于驾驶员安全地将车辆驶入专用车道。第二种方法是将这个负担放在车队管理协议上，驾驶员只需按下按钮就可以触发进入机动。显然，第二种方法是一种更安全的方法，因为协议对周围环境有更好的看法，并且可以决定何时是执行进入机动的好点。我们将使用第二种方法，并详细说明在车道 0 上行驶的车辆 C_5 如何执行进入机动：

(1) 选择目标车队:第一步首先是要找到更为合适的目标车队。目标车队的队伍长度必须小于最大队伍规模，否则新的加入请求将被车队队长拒绝。RSU 可以按照一些偏好排序将目标车队的列表呈现给驾驶员进行选择。选择车队的两个可能的标准是基于路线和基于成本。在基于路线的选择中，司机加入了在她为自己的旅行选择的路线上行驶的队列。在基于成本的选择中，可以采用一定的定价来进行车队成本分析，选择成本最低的车队。

(2) 换道: C_5 开始监听信号，检查目标车队的方位，计算到队长的距离，并测量预计到达时间 ξ 。如果 ξ 足够大，那么车队管理协议将引导 C_5 进入车道 1。 C_5 切换到 CACC 模式，车队时距 $T_g = 3.5$ s，作为自由车辆。随后， C_5 可以并入目标车队。在本作品中，我们只实现从后面进入车队。

2.2.3 变道

在我们的协议中，变道被认为是一种基本的队列机动，是与周围交通交互的重要组成部分。车辆采用变道方式进入预留队列车道，车队成员离开，车队长离开采用变道方式退出预留车道。在本作品中，我们将变道视为手动驾驶行为，并采用 SUMO 中默认的变道模型 LC2013 作为横向控制逻辑。具有变道合作的协同驾驶不在本作品的研究范围之内。

2.3 针对红绿灯路口场景的车队管理协议

2.3.1 红绿灯路口场景描述

信号控制交叉口是城市交通系统的瓶颈，为了减少车队在十字路口的停车次数，提高交通效率，减少车辆在路口的通行时间，降低油耗和有害气体排放量，我们设计针对红绿灯路口场景的车队管理协议，在该场景下基于 GLOSA(Green Light Optimal Speed Advisory)规划车队的速度轨迹，并且根据实时情况对车队长度进行调整。

我们的主要目标是最大化交叉口吞吐量，减少车队车辆在路口处的停车或者急刹车的情况，使车队尽可能快速地通过交叉口，因此我们的问题可以用下面的式子表示，其中 T 是交叉口吞吐量， X 是所有车辆轨迹的集合。

$$\max T(X)$$

路口吞吐量是一段时间内进入路口的车辆数量，因此对于在一个信号周期内以队列形式行驶的车辆，如果前车在绿灯阶段较早进入路口，则在该绿灯阶段将有更多时间留给后车通过路口，此外，如果车队中的队内间距更小，则更多车辆将有机会通过十字路口。因此，上述优化问题可以转化为最小化绿灯阶段车辆到达十字路口的时间：

$$\min_{x_i \in X} t_{arrival}(x_i)$$

对于车队队长和跟随者， $t_{arrival}$ 的计算是不同的。对于队长，由于其前方没有其他车辆，它的到达仅取决于其自身的驾驶状态；而对于跟随者，到达时间取

决于前车的到达时间 t_{pre} 和与前车之间的车头时距 hw , 而前车的到达时间又可以递归地表示, 最终跟随者的到达时间可以用队长的到达时间与车头时距 hw 表示:

$$t_{arrival} = \begin{cases} t_{leader_arrival} & leaders \\ t_{pre_arrival} + hw & followers \end{cases}$$

$$= \begin{cases} t_{leader_arrival} & leaders \\ t_{leader_arrival} + \sum_{j=1}^i hw_j & followers \end{cases}$$

由于车头时距 hw 的值是在车队的控制算法中定义的, 在本作品中即为 CACC 控制算法中的 “TG(Time Gap)” 参数, CACC 控制算法的目的就是使车队中保持规定的车头时距, 因此, 我们在后续的计算中认为 hw 为 CACC 中定义的 TG 值。因此本算法的重点在于最小化队长的 $t_{arrival}$, 对于队长来说, 由于其前方没有车辆, 队长的到达仅取决于其自身的驾驶状态, 可以通过 GLOSA 算法进行优化。

除了速度优化问题之外, 车队在红绿灯路口处可能出现队形变换的问题。有以下两种不可避免的情况出现: 1、在绿灯阶段, 车队想要以最快的速度通过路口, 但车队长度太大导致无法让所有车辆都通过, 这时车队必须要从某个位置断开; 2、在红灯阶段, 已有一个小规模的车队在路口处等待, 此时又进入了一个小规模车队, 两个车队可以合并为一个合适规模的车队在绿灯时一起通过路口。因此还需要一个灵活且安全的车队队形变换的规则集, 以确保车队在正确的时间点进行适合的机动操作。

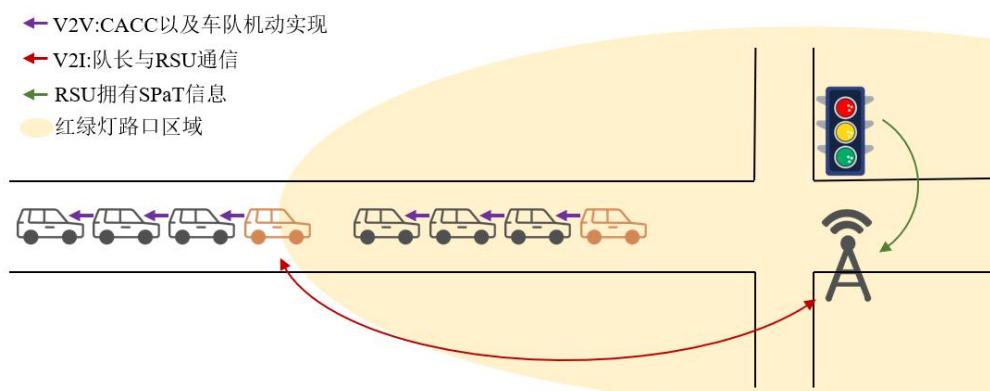


图 2-9 红绿灯路口场景信息交互图

路口处的 V2I 与 V2V 的信息交互模式以及指令下达情况图 2-9 所示，RSU 部署在红绿灯路口处，拥有红绿灯的 SPaT 信息，在 RSU 的通信范围内及定义为红绿灯路口区域。整个协议按照如下的步骤进行：首先，我们讲 RSU 的通信范围定义为 **路口区域**，车队的队长在到达路口区域时，通过 V2I 通信将车队的信息，包括自身的速度、加速度以及车队的基础信息发送给 RSU，如图中的红线所示；RSU 利用 SPaT 信息、车队的基础信息以及队长的速度信息，计算出两个关键量：该车队的最佳速度轨迹、最佳长度，发送给车队队长，同样属于图中红线的 V2I 通信；车队中车辆通过 V2V 通信，如图中紫色箭头所示，在 CACC 控制策略的控制下，根据 RSU 提供的速度轨迹以及最佳长度，在车队中进行速度的变化和相应的机动操作。

2.3.2 最优化车队速度轨迹

本小节的目标是最小化队长到达十字路口的时间 $t_{leader_arrival}$ ，同时尽量避免在十字路口停车，减少气体排放。由于队长前方附近没有其他车辆，同一车队中所有的跟随者都应该与队长拥有相同的轨迹，因此将 GLOSA 算法应用于队长的轨迹优化是合理的。通过利用轨迹优化，队长可以带领整个车队从绿灯阶段开始以最小延迟进入十字路口，同时避免在十字路口停车并减少气体排放。

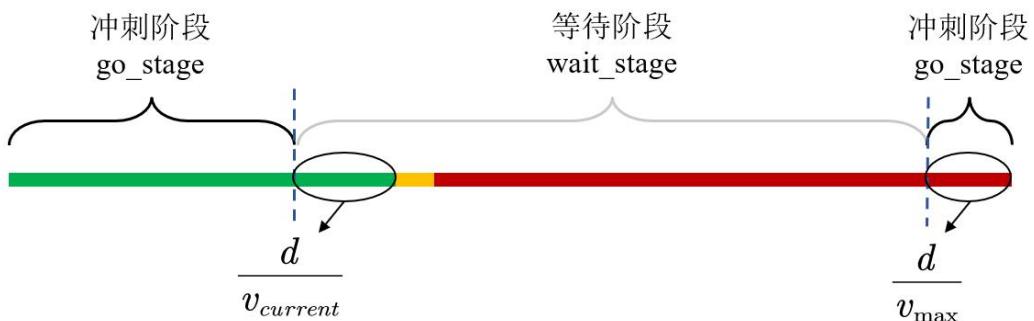


图 2-10 阶段分割示意图

路口处的交通信号灯的信息如图 2-10 所示，注意交通信号灯中是有黄灯的存在的，但是根据最新的交通规则，黄灯在信号灯中起到提醒、警示的作用，车辆在黄灯期间是不可以通过路口的，交通规则效应上与红灯相同，因此我们直接把黄灯阶段合并在红灯中。

我们将红绿灯分为了两个阶段，分别是黑色括号中的 **冲刺阶段**（go_stage）

和灰色括号中的等待阶段（wait_stage）。一般的 GLOSA 算法会将推荐速度 V_{ref} 分为绿灯阶段与红灯阶段，本作品中我们在绿灯与红灯期间分别设置两个临界值，因此划分出了冲刺阶段与等待阶段，与绿灯阶段和红灯阶段没有完全重合，而是有一定的偏移量。这是因为我们考虑到车队进入到路口处 RSU 的通信范围时（一般为 100m 到 500m，本作品中设置为 200m），会接收到 RSU 关于车队管理的消息，也就是推荐速度以及最优长度，由于此时车队的队长与路口有一定的距离，在绿灯即将结束某个时刻即使以最大速度冲刺也无法通过路口，对应的在红灯即将结束的某个时刻即使以某一最小速度也可以在变绿时通过路口，而无需像其他红灯的等待阶段一样减速。

这两个临界值的设置及原因如表 2-1 所示，其中 t_{green}/t_{red} 分别表示当前是红灯/绿灯，距离信号灯变为绿/红的时间，也就是 SPaT 信息；d 表示当前车辆与路口的距离，也就是车辆进入 RSU 通信范围时与路口的距离； V_{max} 是道路最大限速， $V_{current}$ 是车队队长进入路口区域时的速度。

表 2-1 临界值设置

临界值大小	绿灯减速临界值	红灯加速临界值
	t_{red}	t_{green}
原因	$\frac{d}{v_{current}}$	$\frac{d}{v_{max}}$
车队的队长在进入红绿灯路口区域时，RSU 给出针对两个不同阶段车队队长不同的建议速度 V_{ref} 和建议加速度 A_{ref} 。当处在 go_stage 部分时，车队旨在以	绿灯最后阶段，车辆有无法通过的可能性。 $d/V_{current}$ 为当前车队以进入区域时的速度匀速行驶至等待线的时间，只有剩余的绿灯时间大于此值，才能保证车辆能够通过	红灯最后阶段，车辆可以加速通过。 通过确保速度最大的车辆能够在变绿时通过，来确保以更小的初速度进入区域的车辆能通过

最大速度冲刺, A_{ref} 取 a_{accel_max} , 以保证能够在红绿灯线变红之前通过路口, 减小因等停带来的吞吐量损失; 当处在 **wait_stage** 部分时, 车队以较小的加速度来**减速**, 其中较小的加速度确保车辆不做剧烈的制动来达到降低油耗的目的, 减速速度则保证在下次信号灯变绿之前不做停留, 减少停车次数。

针对 **wait_stage**, 已知的参数为距离下一次信号灯变绿的时间 t_{green} , 在这段时间内需要完成的位移距离 d (也即车队队长向 RSU 发送消息时与路口等待线的距离), 当前车速 $v_{current}$, 以及车辆加速度的范围(a_{min}, a_{max}), 其中 $a_{min} < 0, a_{max} > 0$ 。我们的目的是使**车辆在这段距离内的行驶过程有更小的油耗和尾气排放量、以更小的延迟进入路口**, 即到达路口时的速度尽量大, 这样就可以保证车队队长带领整个车队能够在信号灯变绿时刚好到达停车线处, 速度的变化尽量平稳以达到节省能耗的目的, 且更大的到达速度使得车辆在通过路口时花费更少的时间, 从而降低通过路口的延迟。

因此这个问题可以建模为以下的一个带约束的多目标优化问题:

$$\min \lambda_1 \int_0^{t_{green}} MOE_e(t) dt - \lambda_2 V_{ref} \quad (1)$$

$$s.t. \quad t_{changeV} + t_{uniformV} = t_{green} \quad (2)$$

$$d_{changeV} + d_{uniformV} = d \quad (3)$$

$$V_{ref} \leq v_{max} \quad (4)$$

$$a_{min} \leq A_{ref} \leq a_{max} \quad (5)$$

优化目标 (1) 中, 权重参数 λ_1 和 λ_2 分别代表了对降低油耗、增加到达路口时的速度两个目标的权重, 本作品中对两个目标相同追求, 因此取两个权重值均为 0.5。而 MOE_e 则是尾气排放的有效性的度量, 它代表瞬时气体排放率 (mg/s)。 e 表示气体排放类型, 例如 CH、CO、NOx、CO2, 这里我们只研究 CO2 气体作为代表, MOE_e 是关于一个时刻的瞬时速度与加速度的函数, 表示为:

$$MOE_{CO_2} \begin{cases} \sum_{m=0}^2 \sum_{n=0}^2 (L_{m,n}^{CO_2} \cdot v^m \cdot a^n), & a \geq 0 \\ \sum_{m=0}^2 \sum_{n=0}^2 (M_{m,n}^{CO_2} \cdot v^m \cdot a^n), & a < 0 \end{cases}$$

由于在模拟程序中是以每个时间步来进行的离散数据，一个时间步 Δt 为 0.1s，因此对 MOE 的积分可以写成求和的形式：

$$\sum_0^{t_{green}} MOE_{CO_2}(t_i) \cdot \Delta t$$

式子（2）中的 $t_{changeV}$ 指加速或者减速所花费的时间， $t_{uniformV}$ 是指车队队长加/减速后以推荐速度匀速行驶的阶段，是一个非负值，式子（3）中的 $d_{changeV}$ 为加速或者减速所行驶的距离， $d_{uniformV}$ 为以推荐速度匀速行驶的距离。这两个式子约束了车辆在 t_{green} 的时间内，需要先以推荐加速度变速行驶至推荐速度然后匀速行驶，行驶的距离为 d 。四个变量都是关于目标变量 V_{ref} 和 A_{ref} 的函数，它们的计算方法如下：

$$t_{changeV} = \frac{V_{ref} - v_{current}}{A_{ref}}$$

$$d_{changeV} = \frac{V_{ref}^2 - v_{current}^2}{2A_{ref}}$$

对于上述最优化问题的求解，使用梯度下降法，首先定义损失函数为：

$$L = \sum_0^{t_{green}} MOE_{CO_2}(t_i) \cdot \Delta t - V_{ref}$$

定义罚项 P ，其形式如下：

$$P = \gamma_1 \max \left(0, \frac{V_{ref} - v_{current}}{A_{ref}} - t_{green} \right)^2 + \gamma_2 \max \left(0, \frac{V_{ref}^2 - v_{current}^2}{2A_{ref}} - d \right)^2 + \gamma_3 \max (0, V_{ref} - v_{max})^2 + \gamma_4 \max (0, a_{min} - A_{ref})^2 + \gamma_5 \max (0, A_{ref} - a_{max})^2$$

其中， γ_i 是正则化参数，决定了对违反约束的惩罚力度。因此新的损失函数为：

$$L_{new} = \sum_0^{t_{green}} MOE_{CO_2}(t_i) \cdot \Delta t - V_{ref} + P$$

然后计算出新的损失函数 L_{new} 相对于 V_{ref} 和 A_{ref} 的梯度分别为 $\frac{\partial L_{new}}{\partial V_{ref}}$ 和 $\frac{\partial L_{new}}{\partial A_{ref}}$ ，使用学习率 η 迭代更新，其中 k 是迭代次数：

$$V_{ref}^{k+1} = V_{ref}^k - \eta \frac{\partial L_{new}}{\partial V_{ref}}, \quad A_{ref}^{k+1} = A_{ref}^k - \eta \frac{\partial L_{new}}{\partial A_{ref}}$$

接着使用梯度下降法对新的损失函数来进行约束。

2.3.3 最优化车队长度

如果对车队的长度没有限制，那么这个车队可能会很长，这就可能会被红灯分成两个车队，而后一个车队必然会停在路口，如果将要被分开的跟随者在到达路口时才紧急制动，就可能会造成燃油的浪费和尾气排放的增加，更严重的是紧急制动会降低车队的串稳定性，可能还会造成追尾等交通事故。此外，在红灯的等待期间，可能会有多个长度较小的车队都进入了红绿灯路口区域，这期间车速较小，因此车间距也较小，是符合车队合并的条件的，如果它们在等待期间合并，就可以以一个车队的形式通过绿灯，由此减小燃油量与尾气排放量。因此，本作品提出了一种计算车队的适当长度的方法，并开发了相应的协议来管理车队。

车队中跟随者的到达时间 $t_{follower_arrival}$ 在 3.3.1 小节中给出了计算公式，如果超过剩余的绿灯阶段时间，这个跟随者将不可避免地停在路口。考虑到 CACC 控制策略的目的是使跟随者与前车保持适当的预设车头时距，预设车头时距与实时车头时距之间的车头时距误差可以控制在很小的范围内，因此我们使用简化的方式来计算车队的适当长度，其中使用 hw_{set} 而不是实时的 hw ，使得计算简单易行。因此最优化车队长度可以计算为下列公式：

$$len = \min_{i \in L} i$$

$$L = \left\{ i \mid \max \left(t_{red} - t_{leader_arrival} - \sum_{k=1}^i hw_{set,k}, 0 \right) == 0, i \in I \right\}$$

其中 len 表示一个车队想要完整地在绿灯阶段通过路口可以容纳的最大车辆数， t_{red} 是下一个红色阶段的开始时间， I 是在该车队行驶的车辆的集合， $hw_{set,k}$ 表示车辆之间的预设车头时距，对于不同类型的车辆预设的车头时距是不同的。本作品中车队中的车辆类型是相同的，都设置为汽车，因此所有的跟随者具有相等的车头时距，所以最佳车队长度可以计算为如下的公式：

$$optSize = len = \begin{cases} \left\lfloor \frac{t_{red} - t_{leader_arrival}}{hw_{set}} \right\rfloor + 1, & go_stage \\ \frac{t'_{red} - t_{red}}{hw_{set}} + 1, & wait_stage \end{cases} (*)$$

在此之前，先要计算 $t_{leader_arrival}$ 的值。

针对 wait_stage，很明显有以下式子：

$$t'_{red} - t_{red} = t_{green_phase_duration}$$

针对 go_stage，由于考虑到车队加速到最大速度需要时间，所以，在这里本文将 $t_{leader_arrival}$ 的计算分为两种可能。第一种可能是当速度大于 $V_{standard}$ ，只需要经历加速阶段就能够完成 d 距离的行驶，第二种情况是当速度小于 $V_{standard}$ ，需要经历加速阶段和匀速阶段才能完成 d 距离的行驶，v-t 示意图如下图所示：

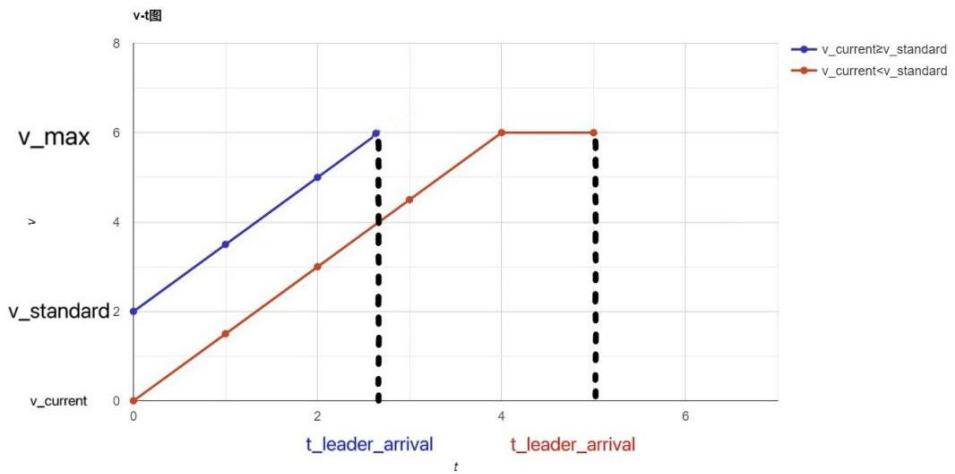


图 2-12 车队完成距离为 d 的 v-t 图

随后，定义 $V_{standard}$ 的大小，根据运动学公式求得：

$$V_{standard} = \sqrt{v_{max}^2 - 2ad}$$

将最后 $t_{leader_arrival}$ 的表达式根据当前速度分情况进行计算：

$$t_{leader_arrival} = \begin{cases} -\frac{v_{current}}{a} + \frac{\sqrt{v_{current}^2 + 2ad}}{a}, & v_{current} \geq V_{standard} \\ \frac{d}{v_{max}} + \frac{(v_{max} - v_{current})^2}{2a}, & v_{current} < V_{standard} \end{cases}$$

车队队长使用 3.3.2 小节所述的最优化速度轨迹方案，在冲刺阶段，无论队长以多大的初速度 $V_{current}$ 进入路口区域，建议速度 V_{ref} 都为 V_{max} ，队长会在路口区域从 $V_{current}$ 加速至 v_{max} 来通过路口，由于 $V_{current} < V_{max}$ ， $t_{leader_arrival}$ 的计算结果一定大于真实的队长到达时间 $t_{true_leader_arrival}$ ，因此计算的最佳长度 len 是小于按照真实情况的计算结果的，这样做是为了保证车队在 optSize 的长度内

一定能通过路口。而对于等待阶段，队长按照最优化速度轨迹行驶就会在绿灯开始时或者开始之后到达路口， $t_{red_prime} - t_{red}$ 的值即为此阶段绿灯持续时间，车队的最优长度就取决于绿灯持续的时间 $t_{green_phase_duration}$ ，与车头时距 hw_{set} ，绿灯持续时间一定小于真实留给车队通行的时间，同样也是为了保证车队在 $optSize$ 的长度内能通过路口。

车队管理协议中关于路口处车队最佳长度部分的算法如下所示，根据 3.2 中关于车队机动的设计，RSU 给车队队长下达了 $optSize$ 的指令后，在前后两个车队的通信范围内会根据 $optSize$ 进行对应的合并或者分裂的机动。

算法 1：路口处车队最佳长度算法

输入：SPaT 信息、车队信息

输出：RSU 发送包含 $optSize$ 的消息

1. 车队 P1 进入路口区域，send(RSU, platoonInfo)
 2. RSU 根据 platoonInfo 和 SPaT，按照(*)公式计算 $optSize$
 3. // 车队需要分裂的情况
 4. if $Pi \rightarrow pltSize > optSize$:
 5. $P1, P2 = Pi \rightarrow split(optSize)$ // 车队从 $optSize$ 处分裂，得到前方车队 P1 和后方车队 P2
 6. P1 根据 RSU 提供的最佳速度行驶，加速通过红绿灯
 7. P2 后续进入路口区域，根据 RSU 提供的最佳速度行驶
 8. end if
 9. // 车队需要合并的情况
 10. if $P1 \rightarrow pltSize < optSize$:
 11. P1 根据 RSU 提供的最佳速度行驶，等待绿灯阶段
 12. if $P2$ 进入路口区域 and $P2 \rightarrow pltSize + P1 \rightarrow pltSize < optSize$:
 13. $P1 = merge(P1, P2)$ // 前后两个车队合并为 P1
 14. P1 根据 RSU 提供的最佳速度行驶，后续通过红绿灯
 15. end if
 16. end if
-

综上，针对红绿灯路口的车队管理协议中，相对于 3.2 中车队机动的情景而言，增加了车队队长与 RSU 之间的通信，队长与 RSU 之间通信使用的消息的重要字段设置如图 2-13 所示。

Vehicle Message	RSU Message
SenderId; Position; Velocity; MaxAcceleration; MaxDeceleration; Headway;	ReceiverID; ReferenceVelocity; ReferenceAcc; OptSize;

图 2-13 消息设置

2.4 基于国密的群密钥车队通信方案

2.4.1 车队群加密方案概述

车队内的成员通过无线通信传递消息，进而完成机动以及达到其他通信目的，然而，由于无线信道的广播性质，为了更新在车队中驾驶状态的车辆通信容易受到窃听、重放攻击以及恶意车辆节点的攻击；且车队是一个动态变化的结构，针对单车通信的 V2V、V2I 的安全通信方案并不能直接适用于车队的场景。

在车队场景中，群加密能够提供重要的安全保障和高效的通信管理。通过使用群密钥，我们能够确保只有车队的成员能够解读车队内部的通信，这可以防止敏感信息（例如路线、速度、位置以及成员间的其他敏感信息）被未授权的第三方获取，从而提高了车队的安全性。另外，通过管理群密钥，我们可以有效地控制车队成员的变动，当车辆加入或离开车队时，只需要更新群密钥，而无需改变其他成员的密钥，这大大简化了车队管理的复杂性。此外，群加密也可以支持安全的组播通信，提高了通信效率。

本作品的群密钥方案中，针对身份认证的加密算法选用 **SM2 国密算法**，针对消息加密部分采用 **SM4 密钥作为群密钥**。国密算法是由中国国家密码管理局发布的一种国家商用密码算法，是为了保护中国政府和商业机构的敏感信息而设计的。**SM2** 是一种基于椭圆曲线的公钥密码算法，由于该算法基于 ECC，故其签名速度与秘钥生成速度都快于 RSA，其与 RSA 的对比如表 2-2 所示；**SM4 算法**采用分组密码结构，每个分组的大小为 128 位，属于无线局域网标准的分组数据对称加密算法，密钥长度和分组长度均为 128 位。

表 2-2 各加密算法的对比

算法名称	SM2	RSA	SM4	AES
算法结构	基本椭圆曲线 (ECC)	基于特殊的可逆模幂运算	非线性函数和线性函数	轮函数的组合
加解密类型	非对称加密算法	非对称加密算法	对称加密算法	对称加密算法
计算复杂度	完全指数级	亚指数级	亚指数级	亚指数级
存储空间	192–256bit	2048–4096bit	128bit	128–256bit
密钥生成速度	较 RSA 算法快百倍以上	慢	快	慢
解密加密速度	较快	一般	快	一般

本作品中设计了针对车队的群加密方案，如图 2-14 所示。

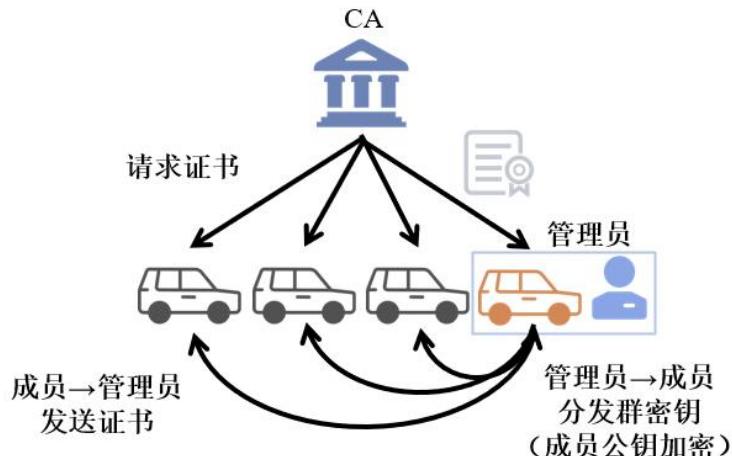


图 2-14 车队群加密方案

采取管理员群密钥算法，管理员拥有生成和分发群密钥的特权，能够将密钥安全地分发给群组成员，在本作品的车队场景中，根据 3.1 的设计，车队队长拥有车队的成员信息，并且能够与车队的所有成员通信，因此由车队的队长担任管理员，在车队形成时生成并分发 **SM4 群密钥**。群密钥需要在公共信道上被安全地分发给车队内的成员，因此由车队成员向管理员（队长）提供 CA 签发的 SM2 公钥证书，管理员验证之后使用 SM2 证书中的公钥将 SM4 群密钥加密后安全地

分发给成员。而 RSU 则担任 CA，车队中所有成员在经过 RSU 时检查自己的证书有效期，如果证书即将过期或者已经过期，则向 RSU 申请新的证书或者请求延长证书有效期。

当车队的成员发生变动时，如发生了车队合并、分裂以及成员的加入、离开，管理员需要相应地进行群密钥的更新与发放，以达到前向安全性和后向安全性。有管理员的中心化群密钥方案的优势在于可以快速地添加和删除成员，而不需要所有成员参与，以保证车队内通信的完整性与机密性。

下面介绍具体的步骤。

2.4.2 车辆实体向 CA 申请证书

I . 生成 SM2 私钥。车辆实体 Ui 使用 SM2 算法基于 VIN (Vehicle Identification Number, 车辆唯一标识) 生成 SM2 私钥 di 。SM2 是一种基于椭圆曲线的公钥密码算法，因此在密钥生成的过程中需要选取合适的椭圆曲线参数来初始化椭圆系统。

II . 证书请求。车辆 Ui 基于私钥 di 签名，创建证书请求 (Certificate Signing Request, CSR)，请求中包括车辆唯一标识信息 VIN 以及主题信息。签名后的证书请求为： $\text{Sign}(di, CSR)$ 。接下来车辆将签名后的证书请求发送给 CA。

III . 验证请求。CA 收到证书请求后，使用车辆公钥验证签名的有效性 $\text{Verify}(Pi, \text{Sign}(di, CSR))$ 。如果验证通过，CA 进一步验证车辆的标识信息。

IV . 签发证书。验证通过后 CA 生成证书 Certificate，证书中包括车辆的公钥 Pi 、标识信息 VIN、证书的有效期以及其他必要的信息。然后，CA 使用自己的根私钥 root_CA_prv 对证书信息进行签名 $\text{Sign}(\text{root_CA_prv}, \text{Certificate})$ 。接下来将签名后的证书发送给车辆 Ui 。

V . 验证证书。车辆 Ui 收到证书后，使用 CA 的公钥 CA_pub 验证证书的有效性 $\text{Verify}(\text{CA_pub}, \text{Sign}(\text{CA_prv}, \text{Certificate}))$ 。如果验证通过，车辆即可使用此证书进行安全通信。

2.4.3 群密钥分发与管理

(1) 针对车队组建与解散群密钥的分发与删除

车队组建的群密钥分发步骤如下：

I . 生成群密钥。当车队 P_j 形成时，管理员（队长） U_0 利用伪随机数生成器生成一个 128 位的随机数作为 SM4 的密钥 K_j 。

II . 发送证书请求。管理员向车队内所有成员广播证书请求 CERT_REQ 消息，请求成员发送由 CA 签发的证书；成员车辆 $U_x, U_y, U_z \dots$ 收到请求后，将证书 Certificate 封装在消息 CERT_MSG 中，由单播的形式发送给管理员。

III. 加密群密钥。管理员每次收到 CERT_MSG 之后，使用存储的 CA 证书验证签名的有效性 Verify(CA_pub, Certificate)，验证通过后提取出证书中的对应的公钥 $P_x, P_y, P_z \dots$ ，用这些公钥分别加密上一步骤中生成的对应的车队的群密钥 K_j ，加密后的群密钥为 $E(P_x, K_j)、E(P_y, K_j)、E(P_z, K_j) \dots$ 。

IV. 分发群密钥。管理员将加密后的群密钥 $E(P_x, K_j)、E(P_y, K_j)、E(P_z, K_j)$ 发送给对应的车辆 U_x, U_y, U_z 。

V. 解密群密钥。如车队中的车辆 U_x ，使用自己的私钥 d_x 对收到的加密群密钥 $E(P_x, K_j)$ 进行解密 $D(d_x, E(P_x, K_j))$ ，得到 SM4 密钥 K_j 。

车队解散时需要进行群密钥的删除，由于车队解散这个机动也是由队长发起的，因此在解散之前，首先由队长向所有车队成员发送删除群密钥的 DEL_KEY 消息；成员删除群密钥后发送 DEL_ACK 给队长，管理员在收到所有的 DEL_ACK 后就删除自己的 SM4 密钥，随后完成正常的车队解散机动。

车队组建与解散是整个车队群密钥通信方案的基础，群密钥还需要考虑的是车队成员的动态变化，在车队成员进入或者离开时，群密钥需要进行更新，以保证前向与后向的安全性，而在车队的场景中，这些变化主要指自由车辆的加入、成员的离开以及车队的合并与分裂。下面详细说明这些进行车队机动时群密钥的更新。

（2）针对车队机动的群密钥更新

本作品中的车队为中心化的车队，即由队长储存车队的成员信息，且车队的机动由队长进行发起，跟随者们在某些情况下可以发起机动，然后需要通知队长进行后续操作，因此在 **车队发生变化时管理员掌握车队成员变化的信息**。

针对**自由车加入与成员离开**的情况，当这两种机动完成后，管理员的车队成员表会进行更新，此时管理员首先生成新的 SM4 密钥，然后执行（1）所描述的剩下的 j 车队组建密钥分发流程，即向车队内的所有成员发送证书请求

CERT_REQ 消息，收到成员的证书后进行验证，验证通过后使用证书中的 SM2 公钥对新的 SM4 密钥进行加密，然后对应分发。

针对车队的合并与分裂的情况，可以看作是车队组建、自由车辆加入、离开的拓展情况，具体的密钥更新方案如表 2-3 所述。

表 2-3 车队机动密钥更新表示

车队 机动	合并	分裂
图示		
前方队 长行为 (LV1)	成员增加，执行（1）生成新的群密钥并分发	成员减少，执行（1）生成新的群密钥并分发
后方队 长行为 (LV2)	无	车队组建，执行（1）生成群密钥并分发

车队合并时，原车队 P1 和 P2 的队长分别为 LV1 和 LV2，合并为 LV1 为队长的 P1，当合并机动完成时，LV1 的车队队员信息得到更新，LV2 的身份由队长变为队员，此时由 LV1 执行（1）来更新 SM4 群密钥，对包括合并对象的所有成员在内的新成员进行证书验证后加密分发。

车队分裂时，原车队 P1 从第四辆车的位置分裂为了两个车队：P1 和 P2，队长分别为 LV1 和 LV2，其中 LV2 是新增加的队长，其原来的身份是跟随者，分裂后 LV1 所存储的车队信息发生改变，等同于前面的成员离开情况，此时 LV1 执行（1）来更新 SM4 群密钥，进行证书验证后加密分发；对于新出现的车队 P2，则同样执行（1）中描述的车队组建算法，此时由 LV2 来生成新的 SM4 群密钥并分发给其成员，此时 P2 中所有成员的群密钥相当于用新车队的群密钥进行覆盖。

(3) RSU 充当 CA 更新证书

而 RSU 则担任 CA, 车队中所有成员在经过 RSU 时检查自己的证书有效期, 如果证书即将过期或者已经过期, 则向 RSU 申请新的证书或者请求延长证书有效期。

2.4.4 安全性分析

对于本作品提出的基于国密的群密钥车队通信方案进行安全性分析, 我们主要从身份认证、完整性、前向安全性和后向安全性和抵抗密钥控制这五个方面进行讨论, 表描述了在密钥管理领域内这五个重要的方案评判标准的含义。我们的方案主要考虑了外部对手, 也就是车队以外的恶意节点, 由内部节点发起的其他攻击, 如加入车辆和离开车辆之间的勾结等不在本方案的讨论范围之内。

表 2-4 车队的安全特性及含义

特性	含义
身份认证 (Authentication)	身份认证保证数据的来源
完整性 (Integrity)	数据完整性确保车辆发送的数据未被攻击者更改
前向安全性 (Forward Security)	车辆成员离开一个车队后, 它无法计算任何后续的车队群密钥。此属性保证一旦车辆离开, 它就无法破解车队的加密通信
后向安全性 (Backward Security)	车辆成员加入一个车队后, 它不能计算任何以前的车队群密钥。此属性可以保证新成员无法获得车队内的早期信息
抵抗密钥控制 (Resistance to Key Control)	方案需要能够防止未授权的实体获得或控制加密密钥, 如攻击者或恶意用户

1、身份认证: 每个车辆都有一个由可信任的 CA 签发的与其 SM2 公钥相对

应的证书，提供了有效的身份验证。当车辆要加入车队时，必须提供其证书给群密钥管理员，管理员会向 CA 验证该证书的有效性。只有在证书被验证为有效时，车辆才能加入车队并获得群密钥。

2、完整性：群密钥由管理员生成并使用每个车队成员的公钥进行加密，然后发送给相应的车辆。这确保了只有对应的车辆能用其私钥解密出群密钥。由于 SM2 和 SM4 都是我国完全自主研发的国产密码算法，所以在缺少密钥信息的情况下，很难对密文进行修改而不被发现。

3、前向安全性：当车辆离开车队时，管理员会生成一个新的群密钥并重新分发给剩余的车辆，因此，即使一个车辆保留了旧的群密钥，它也不能用这个旧密钥解密新的通信内容；而当车队解散时，管理员在确保所有的车队成员都删除了自己持有的群密钥后，才会删除所管理的对应车队的群密钥，以确保在车队解散后成员不再拥有之前的群密钥。

4、后向安全性：由于每个车辆都使用其 SM2 私钥解密得到群密钥，所以即使一个新的车辆加入并得到了新的群密钥，它也不能用这个新密钥解密旧的通信内容。

5、抵抗密钥控制：在本方案中，只有车队管理员能生成和分发群密钥，其他的车辆不能控制群密钥。即使管理员被攻击，攻击者也无法控制群密钥，因为群密钥是用每个车辆的公钥加密的，只有相应的车辆才能用其私钥解密。

3.实现方案

实验的具体环境配置如下所示：

表 3-1 环境配置表

操作系统	Ubuntu 18.04.6 LTS
网络模拟器	OMNeT++ 5.4.1
交通模拟器	SUMO 1.8.0
密码学库	gmssl3.1.1

3.1 总体框架

我们的系统是基于开源的车联网模拟器 VENTOS 实现的。VENTOS 实际上也是基于更为广泛使用的车联网模拟器 Veins (Vehicles in Network Simulation) 开发的，两者都有两个重要的组成部分：OMNeT++ 和 SUMO，接下来将对他们进行详细的介绍。

3.1.1 VENTOS

VENTOS 是一个开源模拟车联网 (Vehicular Ad hoc Network, 简称 VANET) C++ 模拟器，用于通过支持 DSRC 的无线通信功能研究车辆交通流、协同驾驶以及车辆与基础设施之间的交互。VENTOS 支持机动车（汽车、公交车、卡车、摩托车等）、自行车和行人等多模态交通仿真，在其中可以设计和验证很多 DSRC 的应用，并监控安装在机动车辆或自行车上的车载单元 (OBU)、路边单元 (RSU) 和行人之间的消息交换。VENTOS 支持绝大多数类型的 V2X 无线通信，包括 V2V（车辆到车辆）、V2I（车辆到基础设施）和 V2P（车辆到行人），基于此，在 VENTOS 的基础上实现 CACC 车队的控制与相关协议的开发是可行且有效的选择。

VENTOS 实际上是一个集成模拟器，通过提供基于 IEEE 802.11p 和定制信道模型的完整车辆通信堆栈，以及基于道路交通模拟器 SUMO 的真实节点移动性建模方法，扩展了 OMNeT++ 网络模拟器，因此具体来说 VENTOS 集成了两

个模拟器：SUMO 和 OMNeT++。SUMO（Simulation of Urban Mobility，城市交通模拟）是一种开源、微观、连续空间、离散时间的 C++ 道路交通模拟器，由德国航空航天中心交通系统研究所开发，并被用作 VENTOS 的车辆交通模拟器，VENTOS 通过 TraCI（Traffic Control Interface）与 SUMO 紧密耦合，利用节点（包括汽车、自行车和行人）的移动信息进行模拟。OMNET++ 是一个开源、可扩展、模块化、基于组件的 C++ 仿真包，用于捕获无线通信仿真。IEEE 802.11p 物理层建模和 IEEE 1609.4 在 Veins 框架中实现，用于不同模块之间的无线 V2X 通信。图 3-1 展示了 VENTOS 架构。

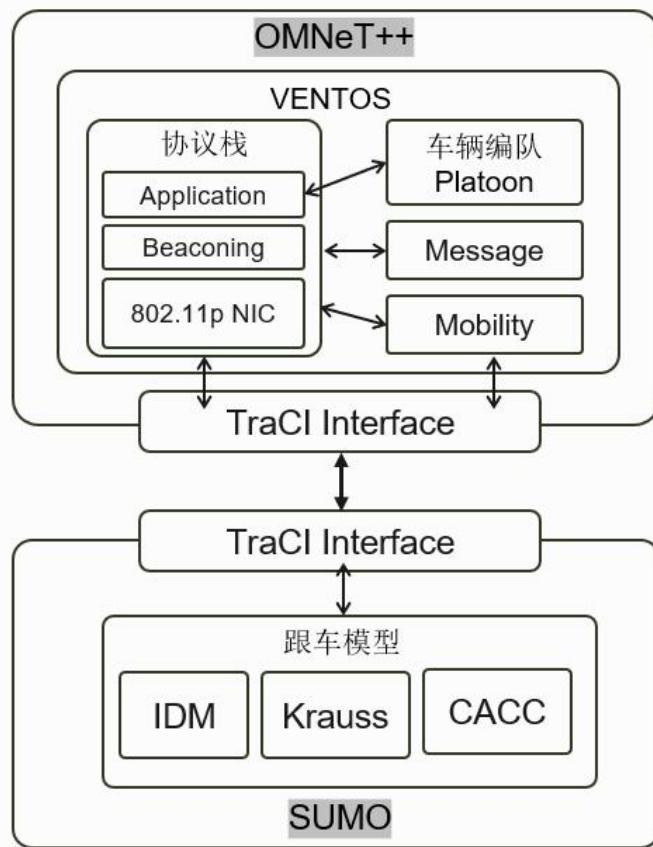


图 3-1 VENTOS 架构

VENTOS_SUMO 是 VENTOS 在 SUMO 中为 ACC/CACC 车辆实施额外的跟车模型，并扩展将 SUMO 连接到外部应用程序的 TraCI 命令。而 **VENTOS core** 即 VENTOS 的核心模块，负责编排所有内部模块，除此之外它还负责启动与 SUMO 的 TraCI 接口，并将模拟结果记录到文本文件中。这些输出文件可以输入到 Matlab 脚本中进行后处理。VENTOS 的内部模块提供了各种功能，如 TSC（信号灯控制）模块实现了在交叉路口使用的许多交通信号控制（TSC）算

法，例如固定时间和交通驱动，或研究社区开发的算法，例如自适应韦伯斯特、最长队列优先(LQF)、最早的工作优先 (OJF)、最大权重匹配 (MWM)；CRL 分发模块实现了 Vehicular PKI 架构中许多著名的 CRL 分发算法，例如 RSU-only、MostPieces Broadcast (MPB)、Intelligent CRL Exchange (ICE) 等…

本作品主要用到的是 VENTOS 项目中关于车队的算法模块：Platoon Management Protocol，以及针对 VENTOS 全局的 addNode 和 trafficControl 模块，它们分别负责通过 xml 文件添加 OMNeT++、SUMO 节点和控制所有节点的动作行为。

3. 1. 2 OMNeT++端

OMNeT++端可以认为是与 OMNeT++无线通信仿真模拟器相关的代码端。

(1) 项目文件结构

图 3-2 展示了我们的项目文件中的根目录，这里的内容分类为 OMNET++端。其中 **src** 存放了项目的所有 C++源代码，也就是各个模块（module）的定义与实现的部分；**examples** 存放了我们作品所展示的例子，主要包括运行一个例子所需要的配置文件，如模拟启动文件 omnet.ini 以及 SUMO 端的路网配置文件等等；**libs** 包含了 VENTOS 所需的仅标头库；**scripts** 包含用于模拟输出后处理的脚本文件。

```
jeremy@ubuntu [01:17:58] [~/IntersectionPlatoon] [master]
-> % tree -L 1
.
├── examples
└── libs
├── license.txt
├── Makefile
├── Makefile.inc
├── out
├── README.md
├── README.MiXiM.txt
└── README.Veins.txt
runme
scripts
src
```

图 3-2 项目根目录文件夹分布

src 文件夹存放了所有模块的 C++源代码，因此我们作品的所有算法实现均在 **src** 中，将在后续的几个小节中详细介绍算法的实现。图 3-3 是 **src** 文件夹的结构，表 3-2 解释了目录中与本作品相关的文件夹的作用。

```
jeremy@ubuntu [01:18:54] [~/I]
-> % tree src -L 1
src
├── addNode
├── baseAppl
├── gettingStarted
├── global
├── libVENTOS.so
├── logging
└── loggingWindow
├── Makefile
├── Makefile.inc
├── makefrag
├── MIXIM_veins
├── mobility
├── msg
├── nodes
├── package.ned
├── router
└── traci
    └── trafficLight
```

图 3-3 src 文件夹

表 3-2 src 目录中文件夹及其作用

addNode	包含用于添加节点（车辆、自行车、行人）的代码
baseAppl	包含所有节点通用的基础应用层
global	包含在项目中全局使用的代码
logging	包含负责记录所有活动的代码
loggingWindow	用于在窗口中显示日志的独立 C++ 应用程序
MIXIM_veins	包含实现 IEEE 802.11p 的静脉项目代码
mobility	包含负责在 OMNET++ 中节点移动的代码
msg	包含定义消息帧结构的代码
nodes	包含车辆、rsu、行人、自行车等的代码
traci	包含能够连接到 SUMO 的交通控制接口代码

example 目录中以不同文件夹的形式包含了不同的案例，下图就展示了两个案例文件夹 intersectionPlatoon 和 platoon_management，在 example 路径下还有一些重要的模块化网络模拟的重要配置文件，如 Network.ned 文件以 OMNET++ NED 格式描述网络拓扑，其中包括了一个案例使用的网络文件所包含的模块如.addNode、global、TraCI 等等；而具体的案例目录中则包含了更加具体化的內容，如.addNode.xml 配置文件描述了该案例的车辆/车队节点的设置，trafficControl 配置文件描述了该案例中车辆/车队节点的运动行为。

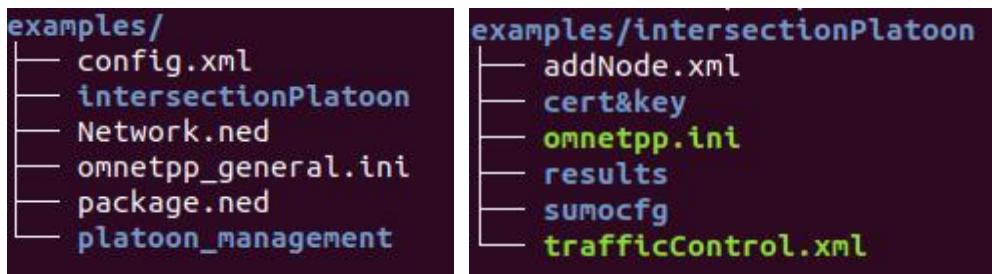


图 3-4 example 文件夹及其中的内容

(2) addNode 模块与 trafficControl 模块

addNode 模块是由 VENTOS 提供来帮助将不同的节点类型添加到模拟中，通过 addNode 模块可以添加 RSU、障碍物等固定节点以及车辆、自行车和行人等移动节点，所有节点都在 “**addNode.xml**”文件中定义，该文件位于与 omnetpp.ini 配置文件相同的文件夹中，addNode 模块读取此文件并决定应在模拟开始时插入哪些节点。

```

<?xml version="1.0" encoding="UTF-8"?>
<.addNode id="example_1">
    <!-- RSU -->
    <rsu id="RSU" pos="-11.5,-11.5,0" />
    <!-- platoon-->
    <vehicle_platoon id="veh" type="TypeCACC1" size="8" route="route1" departPos="825" departLane="1"
        platoonMaxSpeed="0" pltMgmtProt="true" optSize="8" maxSize="10" interGap="3"/>
</.addNode>

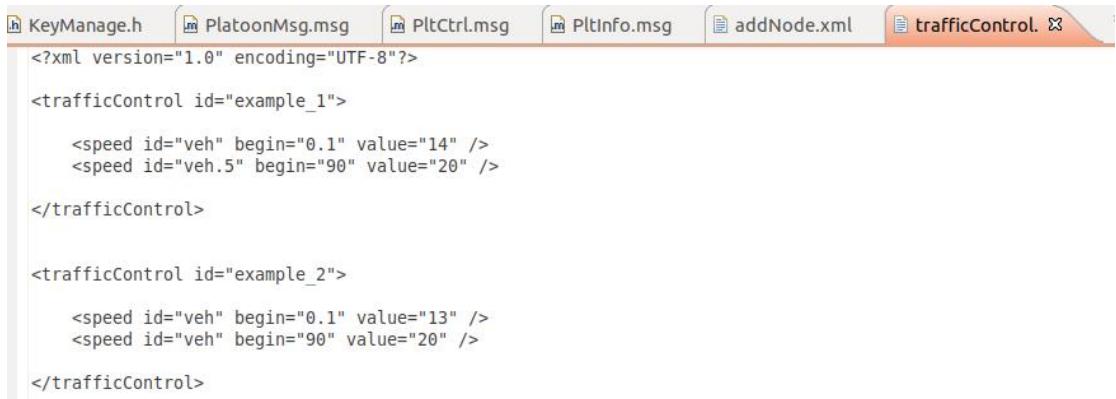
<.addNode id="example_2">
    <!-- RSU -->
    <rsu id="RSU" pos="-11.5,-11.5,0" />
    <!-- platoon-->
    <vehicle_platoon id="veh" type="TypeCACC1" size="8" route="route1" departPos="790" departLane="1"
        platoonMaxSpeed="0" pltMgmtProt="true" optSize="8" maxSize="10" interGap="3"/>
</.addNode>

```

图 3-5 intersectionPlatoon 中的 addNode.xml

如上图是案例 intersectionPlatoon 的 addNode.xml 文件，由于这个案例中包含了几个不同的子案例，针对每一个子案例都可以定义不同的 addNode，比如针对 example_1 的 addNode，添加了一个 RSU 节点，该节点在 SUMO 地图中的位置是(-11.5, -11.5, 0)，添加了一个车队 (vehicle_platoon) 类型的节点，该车队的 id 为 veh，根据 addNode 定义的车队命名规则，该车队的第一辆车（队长）的 SUMOID 为 veh，后面的车辆的 ID 依次为 veh.1、veh.2...，该车队采用 CACC1 控制策略，选取 SUMO 中定义的 route1 线路，出发位置为 825，出发车道为 1；

接下来还定义了车队的一些特殊参数，包括车队管理协议的开启、车队的最佳长度 optSize、最大长度 maxSize 以及车队内的车头时距（hw），也叫做 interGap。



```

KeyManage.h  PlatoonMsg.msg  PltCtrl.msg  PltInfo.msg  addNode.xml  trafficControl.xml

<?xml version="1.0" encoding="UTF-8"?>
<trafficControl id="example_1">
    <speed id="veh" begin="0.1" value="14" />
    <speed id="veh.5" begin="90" value="20" />
</trafficControl>

<trafficControl id="example_2">
    <speed id="veh" begin="0.1" value="13" />
    <speed id="veh" begin="90" value="20" />
</trafficControl>

```

图 3-6 intersectionPlatoon 中的 trafficControl.xml

上图则是该案例的 trafficControl.xml 文件，同样包含了几个不同的子案例， trafficControl 文件主要定义了 SUMOID 车辆在某个模拟时刻或者某段模拟时间内的动作行为。

(3) 消息类型

针对本作品的三个部分：基础机动层、场景层、群密钥方案，分别为其定义了若干种不同的消息类型来实现各部分的功能，见表 3-3。

表 3-3 不同功能的消息

使用部分	消息类型	作用
基础机动层	PlatoonMsg	负载机动相关微指令和数据
场景层	PltCtrl、PltInfo	车辆与 RSU 通信数据
群密钥方案	KeyMsg	负载指令以及证书、密钥等数据

每一种消息类型都有特定的作用和功能，基础机动层的消息 PlatoonMsg 负载了机动相关的微指令和执行机动时所需要的数据；场景层的消息用于车辆与 RSU 进行通信的；而群密钥方案也定义了新的消息类型 KeyMsg，用于负载指令以及证书、密钥等数据。详细的消息定义和使用在后面的小节中会展开描述。

3.1.3 SUMO 端

VENTOS 使用 SUMO 进行道路交通模拟，SUMO 提供了①各种道路、交叉口和车道结构的路网拓扑；②车辆行为的建模，其中主要指驾驶模型，包括 IDM（Intelligent Driver Model）、Kruass 跟车模型、ACC（Adaptive Cruise Control）、CACC（Cooperative Adaptive Cruise Control）；③数据收集和分析，主要指各种类型的仿真数据，包括车辆位置、速度、交通流量、通信消息等，这些数据可以用于分析和评估车辆网络的性能、路况改善策略、交通流量优化等。VENTOS 通过为 ACC/CACC 车辆添加新的跟车模型以及定义新的 TraCI 命令来扩展 SUMO 的功能。

(1) TraCI 接口

OMNeT++网络和 SUMO 交通模拟框架之间的耦合是通过 SUMO 公开的 TraCI 接口完成的，通过使用 TraCI 接口，VENTOS 向 SUMO 查询当前的交通状态，如车辆数量、位置和速度等，并且能够修改交通动态，包括车辆的速度以及 SUMO 中定义的车头时距（SUMO 端中使用参数 tau 表示）。

VENTOS 框架重写了 C++ TraCI API，并通过提供新方法扩展了官方的 SUMO TraCI API，traci/TraCICommands.h 文件包含 VENTOS 支持的所有 TraCI API。在本作品中的 TraCI API 调用是通过 TraCI 指针实现的，下面的代码创建一个 TraCI 指针对象的实例，然后用该对象调用了 TraCI API 的函数 vehicleGetSpeed()，返回了 SUMOID 车辆的当前速度。

```
auto TraCI = VENTOS::TraCI_Commands::getTraCI();
double speed = TraCI->vehicleGetSpeed(SUMOID);
```

通过这个“TraCI”对象可以调用 TraCI 接口函数，实现交通信息的获取和控制交通。表 3-4 列出了本作品主要用到的 TraCI 函数以及他们的返回值类型和作用。

表 3-4 部分 TraCI API 函数

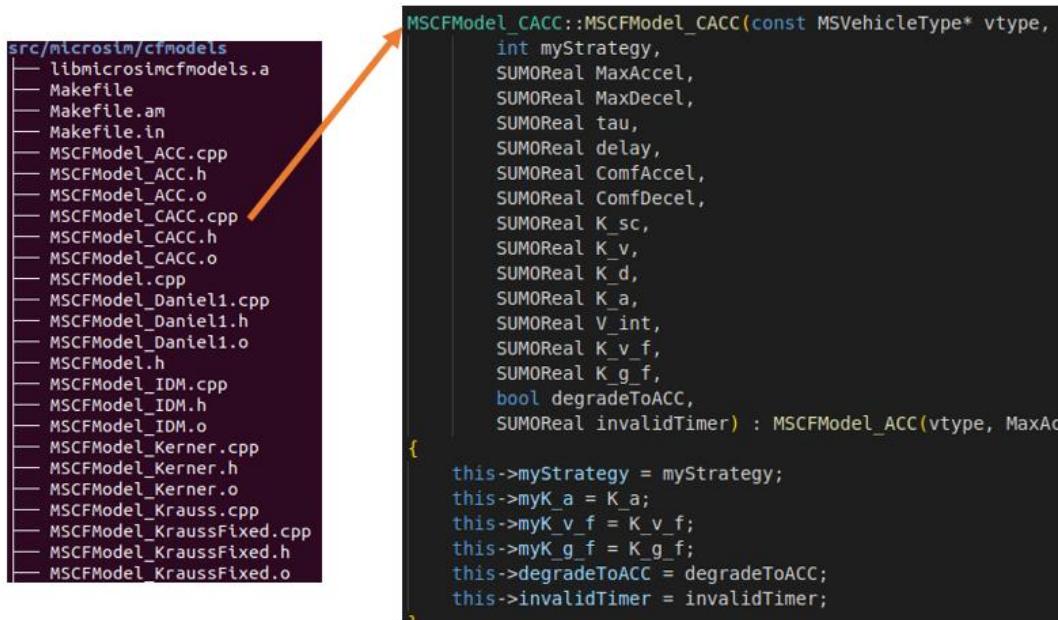
TraCI API 函数	返回值类型	作用
vehicleGetSpeed	double	获取车辆当前速度
vehicleSetSpeed	void	设置车辆速度
vehicleGetPosition	TraCICoord	获取车辆当前位置坐标

TLGetPhase	uint32_t	获取信号灯的当前的 phase
TLGetState	std::string	获取信号灯当前的 state
TLGetNextSwitchTime	uint32_t	获取信号灯到下一个 phase 的模拟时间

(2) CACC 实现

车辆在 VENTOS 中接收到的数据会馈送到 SUMO 中的 CACC，队列协议和应用逻辑在 OMNeT++ 框架中实现，而应用决策的驱动和部分应用逻辑在 SUMO 中实现。

VENTOS 通过修改 SUMO 源码以实现车辆编队控制器。所有 SUMO 端的源文件位于 SUMO 根文件夹下的 sumo/src/microsim/ 文件夹中，其中最重要的是 cfmodels 子文件夹，在该文件夹中实现了多种不同的应用于车队的控制策略。图 3-7 是编译之后的 SUMO 文件夹中的 src/microsim/ 路径中的部分文件情况，这里可以看到定义 ACC、CACC、Daniel1、IDM 控制策略的.cpp 和.h 文件，在 MSCFModel_CACC 的源文件中，定义一个 CACC 控制策略的对象，其初始化参数即为在 SUMO 的路由文件(.rou.xml)中定义车队类型所需要的参数。



```

src/microsim(cfmodels
├── libmicrosimcfmodels.a
├── Makefile
├── Makefile.am
├── Makefile.in
├── MSCFModel_ACC.cpp
├── MSCFModel_ACC.h
├── MSCFModel_ACC.o
├── MSCFModel_CACC.cpp
├── MSCFModel_CACC.h
├── MSCFModel_CACC.o
├── MSCFModel.cpp
├── MSCFModel_Daniel1.cpp
├── MSCFModel_Daniel1.h
├── MSCFModel_Daniel1.o
├── MSCFModel.h
├── MSCFModel_IDM.cpp
├── MSCFModel_IDM.h
├── MSCFModel_IDM.o
├── MSCFModel_Kerner.cpp
├── MSCFModel_Kerner.h
├── MSCFModel_Kerner.o
├── MSCFModel_Krauss.cpp
└── MSCFModel_KraussFixed.cpp
      └── MSCFModel_KraussFixed.h
      └── MSCFModel_KraussFixed.o

```

```

MSCFModel_CACC::MSCFModel_CACC(const MSVehicleType* vtype,
                                int myStrategy,
                                SUMORReal MaxAccel,
                                SUMORReal MaxDecel,
                                SUMORReal tau,
                                SUMORReal delay,
                                SUMORReal ComfAccel,
                                SUMORReal ComfDecel,
                                SUMORReal K_sc,
                                SUMORReal K_v,
                                SUMORReal K_d,
                                SUMORReal K_a,
                                SUMORReal V_int,
                                SUMORReal K_v_f,
                                SUMORReal K_g_f,
                                bool degradeToACC,
                                SUMORReal invalidTimer) : MSCFModel_ACC(vtype, MaxAccel)
{
    this->myStrategy = myStrategy;
    this->myK_a = K_a;
    this->myK_v_f = K_v_f;
    this->myK_g_f = K_g_f;
    this->degradeToACC = degradeToACC;
    this->invalidTimer = invalidTimer;
}

```

图 3-7 cfmodels 中的 CACC 控制策略的实现

3.2 车队机动的实现

3.2.1 车辆节点应用层的实现

在 VENTOS 中添加节点可以使用 addNode 模块来完成，对应的节点类型在文件夹/src/node 中，其中车辆对应的文件夹为 vehicle，图 3-8 是 vehicle 目录中的内容。

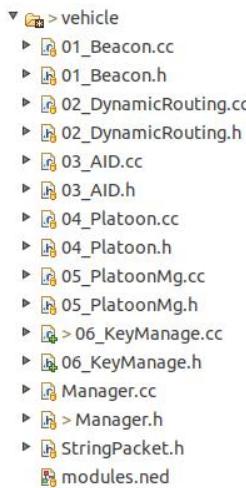


图 3-8 vehicle 文件夹

vehicle 中关于应用层的文件夹是使用分层的方式组织的，01_Beacon 是最底层的应用，接着 02_DynamicRouting 中定义并实现了继承自 01_Beacon 的类，相似的，后面的类都是从上一层继承而来，而最底层的 Beacon 类则是继承自 /src/BaseAppl 目录中定义的 BaseWaveApplLayer，因为车辆的应用层建立在车间的 WSM（Wave Short Message）通信协议的基础之上。

```

class ApplVBeacon : public BaseWaveApplLayer
{
private:
    typedef BaseWaveApplLayer super;
protected:
    double sonarDist;
  
```

图 3-9 Beacon 层的定义

本节主要针对车队机动的实现，具体代码位于 **04_Platoon.h**、**04_Platoon.cc**、**05_PlatoonMg.h**、**05_PlatoonMg.cc** 中，04 部分实现了车队的基本通信，05 部分则基于 04 的基本车队通信，首先按照 3.2 节所设计的算法实现了车队的 4 种机动：合并、分裂、基于此的成员离队和队长离队，因此本节主要解释 05_PlatoonMg 关于车队机动的具体实现。

车辆节点的实现是通过继承关系的多个应用层中定义的实现应用层的类来实现的，在 05_PlatoonMg.h 中，在其他层的类也同理，首先定义了该类的父类，在保护类函数（protected）或者公有类（public）中首先调用父类的该函数，执行完成后再执行子类的函数，以保证车辆节点中基于继承关系的类的正确实现。比较典型的共有函数：有初始化和结束处理函数 initialize()、finish()，保护类函数：处理不同来源的 Beacon 消息的函数 onBeaconVehicle()、onBeaconRSU()，在图 3-10 中可以看到在这些保护类函数的实现中都首先调用了其父类的相应函数，再执行本身的代码。

```

public:
    ~AppLVPlatoonMg();
    virtual void initialize(int stage);
    virtual void finish();

protected:
    virtual void handleSelfMsg(omnetpp::cMessage*); 
    virtual void handlePositionUpdate(cobject*); 
    virtual void onBeaconVehicle(BeaconVehicle*); 
    virtual void onBeaconRSU(BeaconRSU*); 
}

void AppLVPlatoonMg::onBeaconVehicle(BeaconVehicle* wsm)
{
    // pass it down!
    super::onBeaconVehicle(wsm);

    if(plnMode == platoonManagement)
    {
        merge_BeaconFSM(wsm);
        split_BeaconFSM(wsm);
        common_BeaconFSM(wsm);
        entry_BeaconFSM(wsm);
        leaderLeave_BeaconFSM(wsm);
        followerLeave_BeaconFSM(wsm);
        dissolve_BeaconFSM(wsm);
    }
}

```

图 3-10 protected 和一些 public 函数中的继承操作

3.2.2 新增消息 PlatoonMsg

Beacon 信标的发送频率一般设置为 0.1s 或者相近的短频发送频率，在车队中用于车间通信维护纵向的 CACC 控制策略。而车队机动需要专门用于车队的消息来进行状态转移的消息递送，beacon 不能用于此目的，因此新增了消息类型 PlatoonMsg.msg 来用于车队中的 V2V 通信，此消息类型负载车队机动相关的微指令（MicroCommands）。消息的定义如图 3-11 所示，

```

typedef struct value
{
    // values used by CHANGE_Tg
    double newTG = -1;

    // values used by CHANGE_PL
    double newPltDepth = -1;
    std::string newPltLeader;

    // values used by MERGE_REQ
    stringQueue myPltMembers;
    bool manualMerge = false;

    // values used by SPLIT_DONE
    double caller = -1;
    stringQueue myNewPltMembers;
    int maxSize = -1;
    int optPlnSize = -1;
    bool manualSplit = false;

    // values used by LEAVE_REQ and ELECTED_LEADER
    double myPltDepth = -1;

    // values used by LEAVE_ACCEPT
    bool lastFollower = false;
} value_t;

```

图 3-11 PlatoonMsg.msg

PlatoonMsg 也是有 WaveShortMessage 派生而来的消息类型，说明车队中微指令消息的传递也是基于 WSM 而实现的。消息中定义了发送者 senderID、接收者 receiverID、微指令类型 uCommandType、收发方的 platoonID 以及一个自定义结构体的字段 value，value 中存放一些针对不同微指令的不同类型的附加值，比如队长向所有成员发送指令更改队长指定“CHANGE_PL”时，就需要指定需要更改的队长在车队中的位置及其 SUMOID。

通过 receiverID 与 receiverPlatoonID 字段的结合使用，还可以实现消息在车队内的单播与广播。当 receiverID 字段设置为“multicast”，receiverPlatoonID 字段设置为该车队的 ID 时，接收者只需要检车这两个字段是否为对应的值，如果是的话则接收并进行下一步的处理，就此实现车队内广播。

3.2.3 算法具体实现

(1) 类中成员变量的设置

本小节的目的是实现 3.2 所设计的车队机动操作，在 05_PlatoonMg 层（车队管理应用层）中定义了不同类型的成员变量以服务于这个目的。

➤ 用于 selfMessage 的 plnTIMER 计时器

根据 3.2 中车队机动的设计，车队中的成员在很多状态下会开启一个计时器，当计时器时间到就会转换为另一种状态，或者维持当前状态，如在合并时，后方车队 leader 首先向前方车队 leader 发送 MERGE_REQ 合并请求

消息，然后会进行一段等待时间，如果等待时间过去之后仍未收到回应，则会重新发送，这种类似的情形在状态机十分常见，解决方法就是 omnetpp 的 CMessage 类型的消息，延时发送一个消息 plnTIMER，当模块收到这个 plnTIMER 消息时说明倒计时已经结束，就可以进行后续的操作。

```
// merge maneuver
// -----
plnTIMER1 = new omnetpp::cMessage("wait for merge reply");
plnTIMER1a = new omnetpp::cMessage("wait to catchup");
plnTIMER2 = new omnetpp::cMessage("wait for followers ack");
plnTIMER3 = new omnetpp::cMessage("wait for merge done");
plnTIMER3a = new omnetpp::cMessage("wait for front platoon beacon");

    ...
else if(vehicleState == state_mergeAccepted)
{
    cancelEvent(plnTIMER1);
    busy = true;

    TraCI->vehicleSetTimeGap(SUMOID, TG);
    TraCI->vehicleSetSpeed(SUMOID, 30.); // catch-up

    // now we should wait until we catch-up completely
    setVehicleState(state_waitForCatchup);

    // MyCircularBufferMerge.clear();
    scheduleAt(omnetpp::simTime() + updateInterval, plnTIMER1a);
}
```

图 3-12 plnTIMER 变量 (.cc 文件中)

➤ 定义 state 和 uCommand 类型的枚举变量

uCommand 类型的枚举变量用于车队内关于机动的微指令消息的传输，需要一个类型变量，指定消息的类型，在实现中我们定义了枚举类的类型变量 uCommand 来实现消息类型。状态机的转换必然需要状态的表示，这里我们选择定义枚举类型的 state 变量来实现状态表征。

```
typedef enum uCommand
{
    MERGE_REQ,
    MERGE_ACCEPT,
    MERGE_REJECT,
    MERGE_DONE,
    ...
};

typedef enum states_num
{
    state_idle,           // 0
    state_platoonLeader, // 1
    state_platoonFollower, // 2
    state_waitForLaneChange, // 3
    ...
};
```

图 3-13 state 和 uCommand 枚举变量

(2) 状态机的实现

一个车队在发生机动之前是不会启动机动的状态机的，队长处于 state_leader 状态，但车间通信的 Beacon 是一直在以一个高频率进行发送的，如合并这样的

机动就可以由后方车队接受前方车队的 Beacon 来判断是否可以合并，可以合并时就会启动合并的状态机，因此通过接收 Beacon 开启状态机，图 3-14 是合并机动的 Beacon 状态机。

```
void ApplVPlatoonMg::merge_BeaconFSM(BeaconVehicle* wsm)
{
    if(!mergeEnabled)
        return;

    if(vehicleState == state_platoonLeader)
    {
        // can we merge?
        if(!busy && plnSize < optPlnSize)
        {
            if(isBeaconFromFrontVehicle(wsm))
            {
                int finalPlnSize = wsm->getPlatoonDepth() + 1 + plnSize;
```

图 3-14 合并机动 Beacon 状态机

状态机的实现主要在于状态的表征、状态转换条件的判断以及状态转换，状态表征在前面已经用 state 枚举变量解决了，状态转换条件判断与状态转换，在 OMNeT++ 的 C++ 代码中，实现形式就是一个带有多重 if 语句的函数，每个 if 语句决定每个状态的接下来的行为。因此，针对每种机动都有一个状态机实现的函数（DataFSM 函数），如图 3-15 就展示了合并机动的状态机函数。

```
void ApplVPlatoonMg::merge_DataFSM(PlatoonMsg* wsm)
{
    if(!mergeEnabled)
        return;

    if(vehicleState == state_waitForMergeReply)
    {
        if (wsm->getUCommandType() == MERGE_REJECT && wsm->getSenderId() == leadingPlnID)
        {
            mergeReqAttempts = 0;
            cancelEvent(plnTIMER1);

            setVehicleState(state_platoonLeader, "Merge_Reject");
        }
        else if(wsm->getUCommandType() == MERGE_ACCEPT && wsm->getSenderId() == leadingPlnID)
```

图 3-15 合并机动状态机函数

而状态机需要形成闭环，前面定义的 plnTIMER 变量作为车辆模块自己的消息，模块可以通过 handleSelfMsg() 函数来处理计时器消息 plnTIMER。在收到不同机动的计时器消息后，程序跳转到相应的状态机中去即可。

3.3 红绿灯路口处协议实现

在 4.2 的基础上实现路口处的车队管理协议，主要是添加了 RSU，也就是添

加了车队与 RSU 之间的 V2I 通信，并且基于此类通信实现了车队最佳长度和最佳速度的计算与负载消息传输。因此本节的实现内容仍然在 05_PlatoonMg 层中。

3.3.1 新增消息 PltInfo 与 PltCtrl

根据 3.3 的设计，路口处的车队管理协议的实现的核心在于①车队在进入路口区域时给 RSU 发送车队的基本信息；②RSU 接收到车队发送的信息后，根据 SPaT 信息综合计算车队的最佳推荐速度与最佳长度，并将其发送给对应车队的队长。针对以上两点，添加两个新的消息类型：PltInfo 与 PltCtrl。

PltInfo 是车队队长发送给 RSU 的车队的基础信息，根据 3.3 中算法的设计，包括了车队的车头时距 TG、车队队长的当前位置 pos、队长的当前速度 speed、车队中成员车辆的最大加速度 maxAccel 与最大减速度 maxDecel。这些信息都是 RSU 在计算该车队的最佳推荐速度与最佳长度时的算法输入。

```
packet PltInfo extends WaveShortMessage
{
    string senderID;
    string receiverID; // myFullId instead of SUMOID
    string sendingPlatoonID;
    double TG = -1; // hw_preset
    TraCICoord pos; // current position
    double speed; // current speed
    double maxAccel; // max acceleration
    double maxDecel; // max deceleration
};
```

图 3-16 PltInfo 消息设置

PltCtrl 是 RSU 发送给车队队长的消息，包含了 RSU 计算出的车队的推荐速度与最佳长度。

```
packet PltCtrl extends WaveShortMessage
{
    string senderID;
    string receiverID;
    string receivingPlatoonID;
    double refSpeed; // reference speed
    int optSize; // reference optSize
};
```

图 3-17 PltCtrl 消息设置

3.3.2 vehicle 应用层

(1) 进入路口区域发送 PltInfo

路口区域在 3.3 中的定义是 RSU 的通信范围，为了达到车队的队长在进入

路口区域时就发送 PltInfo 消息，需要在队长第一次收到 RSU 广播的 Beacon 时就发送消息，因此在保护类函数 onBeaconRSU() 中实现车队基本消息的封装与发送。这里需要注意的是我们定义了布尔变量 haveSendPltInfo，当队长将 PltInfo 发送之后，就将该值设为 true，以免当下次收到 RSU 的 Beacon 时再次发送。

```
void ApplVPlatoonMg::onBeaconRSU(BeaconRSU* wsm)
{
    // pass it down!
    super::onBeaconRSU(wsm);

    // only leader responds beaconRSU
    if(vehicleState == state_platoonLeader)
    {
        //only send once
        if(!haveSendPltInfo)
        {
            // collect info
            const char * sender = wsm->getSender();
            TraCICoord pos = TraCI->vehicleGetPosition(SUMOID);
            double speed = TraCI->vehicleGetSpeed(SUMOID);
            double maxAccel = TraCI->vehicleGetMaxAccel(SUMOID);
            double maxDecel = TraCI->vehicleGetMaxDecel(SUMOID);
            LOG_INFO << boost::format("%s send PltInfo: receiverID: %s, TG:
                %SUMOID.c_str()
                %sender
                %TG
                %pos.x
                %pos.y
                %speed
                %maxAccel
                << std::flush;
            // call sendPltInfo
            // use TG instead of TraCI->vehicleGetTimeGap()(TP for leader)
            sendPltInfo(sender, TG, pos, speed, maxAccel, maxDecel);
            haveSendPltInfo = true;
        }
    }
}
```

图 3-18 onBeaconRSU 函数实现

(2) 收到 PltCtrl

当车队队长收到 RSU 计算之后发送的 PltCtrl 消息后，就应当根据 PltCtrl 消息中的推荐速度与最佳长度来进行速度的调整，以及 optSize 值的调整。根据 3.2 中机动的设置，车队会自动根据 optSize 的值进行对应的机动，就此就达到了车队管理协议的目的。

```

void ApplVPlatoonMg::onPltCtrl(PltCtrl* wsm)
{
    std::string receiverID = wsm->getReceiverID();
    std::string receivingPlatoonID = wsm->getReceivingPlatoonID();
    if(vehicleState == state_platoonLeader && SUMOID == receiverID && myPlnID == receivingPlatoonID)
    {
        optPlnSize = wsm->getOptSize();
        LOG_INFO << boost::format("%s receive PltCtrl\n optPlnSize: %d\n refSpeed: %.2f\n")
            %SUMOID
            %optPlnSize
            %wsm->getRefSpeed()
            << std::flush;
        TraCI->vehicleSetSpeed(SUMOID, wsm->getRefSpeed());
    }
}

```

图 3-19 onPltCtrl 函数实现

3.3.3 RSU 应用层

RSU 同样也 VENTOS 定义的节点中的一种，其源码在/src/nodes/RSU 中，同样可以通过 addNode 模块添加节点。由于 RSU 参与在车队管理协议中，作为一种不同于之前实现的 RSU 的应用，为其添加一个新的应用层：06_Intersection，与车队管理协议相关的源代码在 06_Intersection.cc 和对应的头文件中。

RSU 在收到队长发送的 PltInfo 后，需要将其中的车队基本信息的内容拿出来，再结合 SPaT 信息进行计算，然后发送计算结果，这些功能集成到函数 onPltInfo() 中，如图 3-20 所示。SPaT 信息的获取通过 TraCI 接口完成，相应的函数如下：

```

int nextSwitchTimeMs = TraCI->TLGetNextSwitchTime("2");
std::string state = TraCI->TLGetState("2");
char nowSignal = state[17];

```

其中第一行获取 ID 为“2”的信号灯下一次信号灯变换的时间，第二行获取信号灯的 state，注意一个路口的信号灯有多个 state，用于管理路口处不同车道的信号灯，根据在 SUMO 端的 Intersection.net.xml 中对信号灯的设置，我们可以知道我们关心的自西向东的 lane2 对应的 state 索引为 17，第三行就通过索引 17 获取当前信号灯的颜色。

```

void ApplRSUIntersection::onPltInfo(PltInfo* wsm)
{
    LOG_INFO << boost::format("%s receive PltInfo: senderID: %s, receiverID: %s, TG
                                %SUMOID.c.str()
                                %wsm->getSenderID()
                                %wsm->getReceiverID()
                                %wsm->getTG()
                                %wsm->getPos().x
                                %wsm->getPos().y
                                %wsm->getSpeed()
                                << std::flush;
    if(strcmp(wsm->getReceiverID(), myFullId) == 0)
    {
        // collect value from wsm
        std::string sender = wsm->getSenderID();
        std::string sendingPlatoonID = wsm->getSendingPlatoonID();
        double TG = wsm->getTG();
        TraCICoord pos = wsm->getPos();
        double speed = wsm->getSpeed();
        double maxAccel = wsm->getMaxAccel();
        double maxDecel = wsm->getMaxDecel();

        // get control value
        CtrlValue cValue = getCtrlValue(TG, pos, speed, maxAccel, maxDecel);

        // send PltCtrl.msg
        sendPltCtrl(sender, sendingPlatoonID, cValue.refVelocity, cValue.optSize);
    }
}

```

图 3-20 onPltInfo 函数实现

另外，信号灯的设置在 SUMO 端进行完成，修改 intersection.net.xml 文件，在其中添加信号灯的相关代码，如图 3-21 所示。该信号灯的 id 为 “2”，一共有 3 个 phase，持续时间分别是 30s，30s 和 3s，20 个 state，车队专用的车道的信号灯控制 state 的索引为 17。除了这里添加信号灯之外，还需要更改路口的类型、每个路口的 connection 的具体设置等。

```

<tlLogic id="2" type="static" programID="0" offset="0">
    <phase duration="30" state="GGGGgrrrrrGGGGgrrrrr"/>
    <phase duration="30" state="rrrrrGGGGgrrrrrGGGGg"/>
    <phase duration="3" state="rrrrryyyyyrrrrryyyy"/>
</tlLogic>

```

图 3-21 SUMO 端添加信号灯

RSU 获取了所有计算推荐速度和最佳长度所需要的参数时，调用函数 getCtrlValue()，函数的片段如图 3-22 所示，按照 3.3 描述的算法来计算推荐速度和最佳长度，最后再封装在 PltCtrl 消息中，以单播的形式发送给对应的车队队长。

```

ApplRSUIntersection::CtrlValue ApplRSUIntersection::getCtrlValue(double TG, T
{
    double distance = abs(pos.x - (-14.0)); // -14 is from sumo->net.xml file,
    double threshold;

    // var from TrafficLight
    enum Stage {
        GO_STAGE,
        WAIT_STAGE
    };
    Stage currentState;
    int nextSwitchTimeMs = TraCI->TLGetNextSwitchTime("2");
    double nextSwitchTime = nextSwitchTimeMs / 1000;
    double currentTime = omnetpp::simTime().dbl();
    double remainingTime = nextSwitchTime - currentTime;
    double greenDuration = 30.0,
           redDuration = 30.0,
           yellowDuration = 6.0; // from sumo->net.xml file
    ...
}

```

图 3-22 getCtrlValue 函数实现

3.4 加密与认证的实现

针对 3.4 节描述的基于国密的群密钥车队通信方案的实现，我们主要完成了车队内的群密钥生成、更新与分发，目前还没有实现方案中设计的 RSU 作为 CA，车队经过 RSU 时进行证书的更新。因此车队的加密通信实现范围在车辆节点之间，根据车辆节点分层式的应用层设计，我们针对加密通信新加入了一个 **06_KeyManage** 层，在该层中实现车队的群密钥方案。

使用 gmssl 3.1.1 库，gmssl 是一个开源的密码库，用于支持国密算法和相关密码应用。gmssl 3.1.1 库支持国密算法，包括 SM2、SM3、SM4 和 SM9 等；也提供了数字签名和验证功能，可以使用 SM2 算法进行数字签名和验证操作；gmssl 3.3.1 同时提供了 X.509 证书的创建、读取和验证功能，支持证书生成、证书签名、证书链验证等操作，用于建立信任关系和进行身份验证。除了库文件，gmssl 3.1.1 还提供了一些命令行工具，用于执行常见的密码学操作，如加密、解密、签名、摘要计算等，这些工具可以方便地在命令行界面进行使用。

3.4.1 生成密钥与证书

我们在/example/intersectionPlatoon 目录中创建了一个文件夹 “cert&key” ，用于存放所有车辆和 CA 的私钥和证书。由于本节的主要目的是 SM4 群密钥的生成、管理、分发，其中在身份认证环节涉及到了 SM2 私钥、申请的生成和证

书的分发，这一部分我们直接在模拟程序外事先使用 **gmssl** 命令行工具生成相应的私钥和证书。

具体方法为，在目录/example/intersectionPlatoon/cert&key 中，首先为每个车辆和 CA 创建一个文件夹，文件夹命名为车辆的 SUMOID 以及 CA，方便在模拟程序中直接通过 SUMOID 来定位到对应文件夹，这里根据 DSRC 通信范围设置为 200m，车队中的车头时距（time gap）和车队一般的速度，设置车队的最大长度为 8，因此只需要 veh、veh.1...veh.7 一共 8 个文件夹存放车队车辆的 SM2 私钥和证书的 pem 文件。而 CA 文件夹中则存放 CA 的 SM2 私钥的 pem 文件，用于生成根证书，以及根证书的 pem 文件，用于签发车辆的公钥证书 pem 文件。

(1) 生成 CA 的 SM2 私钥、自签发根证书。具体指令如下：

```
gmssl sm2keygen -pass 1 -out rootcakey.pem
gmssl certgen -CN ROOTCA -days 3650 -key rootcakey.pem -sm2_id ca -pass 1
-out rootcacert.pem
```

第一行生成了密码为“1”的 CA 的根密钥文件 rootcakey.pem。第二行输入了刚生成的 rootcakey.pem 私钥文件，设置 CN 为 ROOTCA、有效期为 10 年，sm2id 为 ca（证书验证时需要输入证书签发者的 sm2id），输出为 CA 的根证书文件 rootcacert.pem，用于给其他实体签发证书。

(2) 生成车辆的 SM2 私钥、证书请求 CSR，由 CA 根证书签发公钥证书。

```
gmssl sm2keygen -pass 1 -out private_key.pem
gmssl reqgen -CN "veh.$x" -key private_key.pem -pass 1 -out req.pem
gmssl reqsign -in req.pem -days 365 -cacert ../CA/rootcacert.pem
-key ../CA/rootcakey.pem -sm2_id ca -pass 1 -out certificate.pem
```

第一行生成车辆的 SM2 私钥文件 private_key.pem。第二行输入私钥文件，生成证书请求文件 req.pem。第三行输入为证书请求文件、CA 的根证书以及 CA 的私钥，输出为 CA 签名后的证书 certificate.pem。

相关的脚本文件位于目录/example/intersectionPlatoon/cert&key 中。

3.4.2 新增消息 KeyMsg 与信号 Signal

车队群密钥方案中，需要通过短波消息在车队之间传送微指令（与车队管理协议相似）以及其中可能包含的具体数据。其中微指令包括了 5 种类型：CERT_REQ、CERT_MSG、ENCRYPT_KEY、KEY_DELETE、DEL_ACK，在 06_KeyManage.h 中，定义了密钥管理层的类，类中定义成员变量 uCommand_k

枚举类型。需要通过短波消息传输的数据为车辆的公钥证书和管理员加密的SM4群密钥。因此新增消息类型 KeyMsg.msg，其定义如图 3-23 所示，其中的 uCommandType 字段存放刚刚说的枚举变量的消息类型，value_k 为在消息文件中定义的结构体，用于存放证书或者加密后的 SM4 密钥的具体数据，存放形式为无符号整型的 vector 变量，其定义如图 3-24 所示，value_k 的两个字段分别对应两种需要传输的数据类型。

```
packet KeyMsg extends WaveShortMessage
{
    string senderID;
    string receiverID;
    int uCommandType;
    string sendingPlatoonID;
    string receivingPlatoonID;
    value_k value; // some uCommands need this field to
                    // send extra information to the receiver
};
```

图 3-23 KeyMsg.msg

```
typedef std::vector<uint8_t> uintVector;

typedef struct
{
    // values used by CERT_MSG
    uintVector certificate;
    // values used by ENCRYPT_KEY
    uintVector encryptedKey;
} value_k;
```

图 3-24 value_k 类型

信号机制是 OMNeT++ 提供的一种用于模块间通信和事件触发的机制，它允许模块之间发送和接收消息，以实现模块之间的交互和协调。车队的群密钥方案涉及到车队的不同机动时的密钥管理，但密钥管理与车队机动位于不同的应用层中，因此使用信号机制来进行事件触发，在车队管理协议层中首先定义两种信号：newLeaderSignal、memberChangeSignal，其含义分别为有新的 leader 出现时的信号，表明新的 leader 出现就组建了一个新的车队，该车队需要执行车队组建的群密钥生成与分发；车队成员发生变化的信号，表明该车队成员发生变化，需要执行对应的密钥更新。

```
public:
    // signal for key management
    omnetpp::simsignal_t newLeaderSignal;
    omnetpp::simsignal_t memberChangeSignal;
```

图 3-25 05_PlatoonMg.h 定义的信号

在 05_PlatoonMg.cc 中，当车队发生合并、分裂等机动的某一特定时刻，队长会发送对应的信号，具体的发送时机如下。

- ① 合并完成时，前方车队队长发送 memberChangeSignal 信号，信号内容填充为队长的 SUMOID，方便正确的对象接收信号。

```
else if(vehicleState == state_mergeDone)
{
    cancelEvent(plnTIMER3);
    plnSize = plnSize + secondPlnMembersList.size();
    plnMembersList.insert(plnMembersList.end(), secondPlnMembersList.begin());

    // emit memberChangeSignal
    StringPacket *packet = new StringPacket();
    packet->setString(SUMOID);
    emit(memberChangeSignal, packet);
```

图 3-26 合并完成发送成员改变信号

- ② 分裂完成时，前方车队队长发送 memberChangeSignal 信号，信号内容填充为队长的 SUMOID，方便正确的对象接收信号。

```
else if(vehicleState == state_mergeDone)
{
    cancelEvent(plnTIMER3);
    plnSize = plnSize + secondPlnMembersList.size();
    plnMembersList.insert(plnMembersList.end(), secondPlnMembersList.begin(),

    // emit memberChangeSignal
    StringPacket *packet = new StringPacket();
    packet->setString(SUMOID);
    emit(memberChangeSignal, packet);
```

图 3-27 分裂完成发送成员改变信号

- ③ 分裂完成时，后方车队队长发送 newLeaderSignal 信号，信号内容填充为队长的 SUMOID，方便正确的对象接收信号。

```
else if(vehicleState == state_waitForSplitDone)
{
    if(wsm->getUCommandType() == SPLIT_DONE && wsm->getSenderId() == c
    {
        cancelEvent(plnTIMER8);

        plnMembersList.clear();
        plnMembersList = wsm->getValue().myNewPltMembers;
        plnSize = plnMembersList.size();

        // emit newLeaderSignal
        StringPacket *packet = new StringPacket();
        packet->setString(SUMOID);
        emit(newLeaderSignal, packet);
```

图 3-28 分裂完成发送新的队长信号

除了上面列举的合并、分裂机动以外，在车队成员离开或者有新成员加入车

队时，队长也要发送 memberChangeSignal。而车队队长离开的情况则如 3.2 所述的，是多个分裂机动的组合，因此不需要单独设置信号的发送。

3.4.3 SM2 身份认证与加解密

这一节主要描述实现 3.4 所设计的车队组群密钥分发算法以及成员改变时群密钥更新算法。不管是车队组建生成新的群密钥然后进行分发，还是成员改变后更新群密钥在进行分发，逻辑都是相同的，下面具体阐述逻辑与实现。

(1) 根据上一节所述的信号机制或者初始化模拟时插入的车队来生成 SM4 密钥。图 3-29 的两段代码分别为初始化时和接收到信号时的行为，都是首调用 **generateSM4Key()** 函数生成 SM4 密钥，然后向成员发送请求证书的消息。

```
void ApplVKeyManage::initialize(int stage)
{
    super::initialize(stage);

    if (stage == 0)
    {
        groupKeyEnabled = par("groupKeyEnabled").boolValue();

        // subscribe memberChangeSignal
        getSimulation()->getSystemModule()->subscribe("memberChangeSignal", this);
        getSimulation()->getSystemModule()->subscribe("newLeaderSignal", this);

        // if i am leader, play GKM
        // generate sm2 key, send
        if(myPlnDepth == 0)
        {
            generateSM4Key();
            sendKeyMsg("multicast", CERT_REQ, myPlnID);
        }
    }

    void ApplVKeyManage::receiveSignal(omnetpp::cComponent *source, omnetpp::simsignal
    {
        StringPacket *packet = dynamic_cast<StringPacket *>(obj);
        std::string senderSUMOID;
        if (packet != nullptr)
            senderSUMOID = packet->getString();
        else
            throw omnetpp::cRuntimeError("Wrong signal value");

        if(signalID == memberChangeSignal && senderSUMOID == SUMOID)
        {
            LOG_WARNING << boost::format("%s: front leader change key...\n")
                % SUMOID << std::flush;
            generateSM4Key();
            sendKeyMsg("multicast", CERT_REQ, myPlnID);
        }
        else if(signalID == newLeaderSignal && senderSUMOID == SUMOID)
        {
            LOG_WARNING << boost::format("%s: back leader generate key...\n")
                % SUMOID << std::flush;
            generateSM4Key();
            sendKeyMsg("multicast", CERT_REQ, myPlnID);
        }
    }
}
```

图 3-29 初始化与接收信号后的行为

(2) 成员接收到 CERT_REQ 信息后, 从文件系统中读取证书文件并封装在消息 CERT_MSG 中发送给管理员。

```
void ApplVKeyManage::onKeyMsg(KeyMsg *wsm)
{
    /* followers, send CERT_MSG to leader */
    if (wsm->getUCommandType() == CERT_REQ && wsm->getSenderId() == myPlnID)
    {
        std::string myCertFilePath = "/home/jeremy/IntersectionPlatoon/examples/intersectionPla
        LOG_INFO << boost::format("%s\n") % myCertFilePath << std::flush;
        BufferData myCert = readCertificateFromPath(myCertFilePath);
        // check
        std::string myCertStringP = uint8ArrayToHexString(myCert.content, myCert.len);
        LOG_INFO << boost::format("%s cert: %s\n") % SUMOID % myCertStringP << std::flush;

        value_k value;
        std::vector<uint8_t> myCertEncode = encodeBufferData(myCert);
        value.certificate = myCertEncode;
        sendKeyMsg(myPlnID, CERT_MSG, myPlnID, value);
        LOG_INFO << boost::format("%s send CERT_MSG to %s\n") % SUMOID % myPlnID << std::flush;
    }
}
```

图 3-30 成员发送证书给管理员

(3) 管理员接收到 CERT_MSG 后, 调用 `verifyCert()` 函数首先使用 CA 的根证书验证收到成员的证书的合法性, 验证通过后提取出证书中的 SM2 公钥, 调用 `encryptSM4Key()` 函数加密自己的 SM4 密钥, 然后封装在 ENCRYPT_KEY 消息中发送给对应成员。

```
/* leader receive certificate from followers, 1.verify it and 2.use pub key to encrypt sm4
if(wsm->getUCommandType() == CERT_MSG && wsm->getReceiverID() == SUMOID)
{
    std::vector<uint8_t> certEncode= wsm->getValue().certificate;
    BufferData cert = decodeBufferData(certEncode);
    std::string sender = wsm->getSenderId();
    std::string caCertFilePath = "/home/jeremy/IntersectionPlatoon/examples/intersectionPla
    BufferData caCert = readCertificateFromPath(caCertFilePath);
    // check
    std::string certStringP = uint8ArrayToHexString(cert.content, cert.len);
    LOG_INFO << boost::format("received\n%s cert: %s\n") % sender % certStringP << std::fl
    // verify
    const char* signerId = "ca";
    bool isCertValid = verifyCert(cert, caCert, signerId);
    if(isCertValid)
    {
        // encrypt and send
        LOG_INFO << boost::format("verified\n") << std::flush;
        BufferData encryptKey = encryptSM4Key(cert);
        std::vector<uint8_t> keyEncode = encodeBufferData(encryptKey);
        value_k value;
        value.encryptedKey = keyEncode;
        sendKeyMsg(sender, ENCRYPT_KEY, myPlnID, value);
    }
}
```

图 3-31 管理员进行证书验证和加密

(3) 成员接收到 ENCRYPT_KEY 消息后, 提出加密后的 SM4 密钥, 从文

件系统中读取出 SM2 私钥，调用 **decryptSM4Key()** 函数解密 SM4 群密钥。

```

/* follower receive encrypt key, decrypt and use it to communicate*/
if(wsm->getUCommandType() == ENCRYPT_KEY && wsm->getReceiverID() == SUMOID) //multicast fr
{
    LOG_INFO << boost::format("%s received sm4 key\n")% SUMOID << std::flush;

    std::vector<uint8_t> keyEncode = wsm->getValue().encryptedKey;
    BufferData ciphertext = decodeBufferData(keyEncode);
    std::string privateKeyPath = "/home/jeremy/IntersectionPlatoon/examples/intersectionPl
    const char* pass = "1";
    BufferData decryptKey = decryptSM4Key(privateKeyPath, pass, ciphertext);
    std::string stringSM4Key = uint8ArrayToHexString(decryptKey.content, decryptKey.len);

    LOG_INFO << boost::format("%s SM4 key: %s\n")% SUMOID %stringSM4Key << std::flush;

```

图 3-32 成员解密获取 SM4 群密钥

以上标蓝的函数是利用 gmssl3.1.1 库所实现的关于 SM2 加解密和证书验证的函数，由于篇幅的原因不在这里具体展示，在源代码中可以查看。除此之外，还有一些工具函数为整个方案进行服务，如从文件系统中读取证书 pem 文件的 **readCertificateFromPath()** 函数，将使用 gmssl3.1.1 直接读取的 pem 文件或加密后的 SM4 密钥转换为 KeyMsg 的 value 字段为无符号整数的 vector 形式的编码函数 **encodeBufferData()** 以及对应的解码函数，**decodeBufferData()**，同样可以在源代码中进行查看。

3.4.4 群密钥分发与加密的测试

为了测试车队中使用群密钥方案是否成功，需要在组建车队和车队机动之后，完成 SM4 密钥以加密的形式分发给车队成员，然后由车队中的一位成员发送一条用其自己的 SM4 密钥加密的消息，在其通信范围内的所有车辆都接收这条消息，然后使用自己的 SM4 密钥对其进行解密。这一类型的消息同样是有具体数值负载的，为了方便起见我们直接在 uCommand_k 枚举变量中添加一个枚举变量 TEST_MSG，即 KeyMsg 多了一种微指令的类型，为测试数据传输定义的一种类型。

我们选择让管理员也就是队长来发送这个加密之后的测试数据，但队长会收到所有成员的 CERT_MSG 消息，我们期望在队长收到所有的 CERT_MSG 消息之后，并且将 SM4 群密钥以各自的公钥加密发送给各个成员，并且成员也完成了解密获取了 SM4 群密钥之后，再发送测试消息 TEST_MSG。因此我们定义一个消息定时器变量 TIMER，定时长度为 3s，确保上述的过程都完成了之后，管

理员再进行 SM4 加密和发送加密信息，具体的代码如下。

```
if(wsm->getUCommandType() == CERT_MSG && wsm->getReceiverID() == SUMOID)
{
    .....
    if (!TIMER->isScheduled())
    {
        scheduleAt(omnetpp::simTime() + SEND_TEST_OFFSET, TIMER);
    }
}
```

因此在 handleSelfMsg()函数中实现使用 SM4 群密钥加密信息并发送，具体的加密实现如图 3-33 所示。

```
void ApplVKeyManage::handleSelfMsg(omnetpp::cMessage* msg)
{
    super::handleSelfMsg(msg);
    if(msg == TIMER)
    {
        cancelEvent(TIMER);
        uint8_t plainText[TEST_TEXT_LEN] = {0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38,
                                         0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38};
        SM4_KEY formedSM4Key;
        sm4_set_encrypt_key(&formedSM4Key, sm4key);
        uint8_t cbuf[TEST_TEXT_LEN];
        sm4_encrypt(&formedSM4Key, plainText, cbuf);

        BufferData bufCipherText;
        bufCipherText.content = cbuf;
        bufCipherText.len = TEST_TEXT_LEN;
        value_k value;
        std::vector<uint8_t> CipherTextEncode = encodeBufferData(bufCipherText);
        value.testMsg = CipherTextEncode;
        std::string stringPlainText = uint8ArrayToHexString(plainText, TEST_TEXT_LEN);
        LOG_INFO << boost::format("\n%s sent secret message: %s\n\n") % SUMOID % string
        sendKeyMsg("broadcast", TEST_MSG, "broadcast", value);
    }
}
```

图 3-33 SM4 加密信息

对应的解密部分的实现，在 onKeyMsg()也就是车辆收到 KeyMsg 后新增加一个分支，处理类型为 TEST_MSG 的情况，在其中使用自身的 SM4 群密钥进行解密和输出。

4.运行效果与性能分析

4.1 基础机动操作例子展示

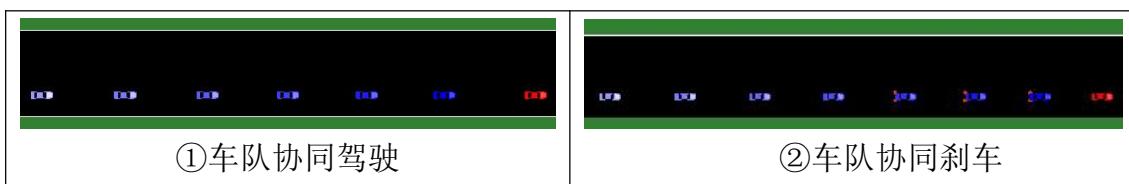


图 4-1 车队协同刹车过程图



图 4-2 车辆进入车队过程图

车队的合并、分裂基础机动将在后面的例子中展示。

4.2 红绿灯路口场景例子展示

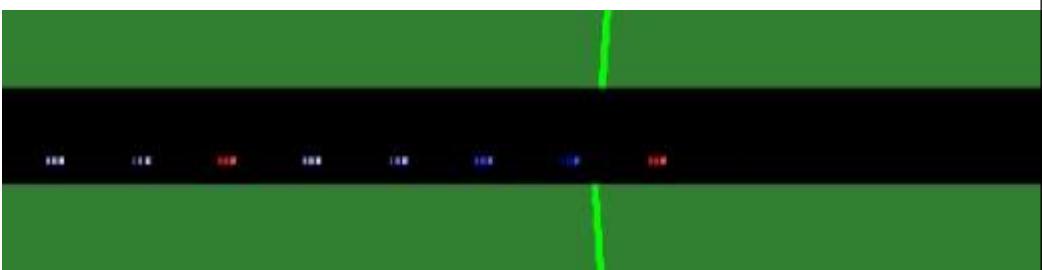
4.2.1 go_stage 中的绿灯到达例子展示

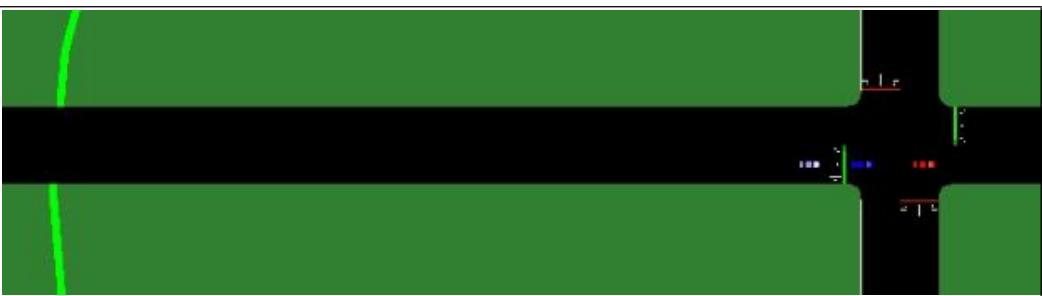
当车队在 go_stage 中的绿灯阶段进入 RSU 覆盖区域，由于得到的 Optsize 小于车队长度，所以要进行分裂操作，本例子分为两个车队，车队C_A可以在剩余的绿灯阶段顺利通过红绿灯路口，所以进行加速机动，旨在达到V_{ref}并顺利通过红绿灯路口，不做停留。车队C_B由于加速到最大速度V_{max}都不能顺利通过红绿灯路口，因而进行减速机动，旨在等待到下一个绿灯时刻刚好驾驶到红绿灯路口线，不做停留便能加速通过红绿灯路口。距离流程如表 4-1：

表 4-1 go_stage 中的绿灯到达例子展示

1、初始化阶段：		
车队长度 (len)	8	
推荐速度 (V _{ref})	无	
Optsize	8	

①车队还未进入 RSU 覆盖区域之前进行协同驾驶

车队个数	1		
机动操作	协同驾驶		
2、车队进入 RSU 区域阶段:			
车队长度 (len)		8	
推荐速度 (V_ref)		20	
Optsize		5	
车队个数		2	
机动操作		分裂	②收到 RSU 信号中的 optsize、V_ref，进行车队分裂行为
3、车队 A 加速，车队 B 减速阶段:			
车队长度 (len)	C_A	5	
	C_B	3	
推荐速度 (V_ref)	C_A	20.0	
	C_B	4.10	
Optsize	5		
车队个数	2		
机动操作	C_A	加速	
	C_B	减速	③C_A、C_B 车队分别进行协同驾驶
4、车队 B 过红绿灯阶段			
车队长度 (len)	3		
推荐速度 (V_ref)	无		
Optsize	5		

车队个数	1	
机动操作	加速	

④完成整个车队的过红绿灯行为

4. 2. 2 wait_stage 中的绿灯到达例子 1 展示

当车队在 wait_stage 中的绿灯阶段进入 RSU 覆盖区域，由于加速到最大速度 V_{max} 都不能顺利通过红绿灯路口，因而进行减速机动，旨在等待到下一个绿灯时刻刚好驾驶到红绿灯路口线，不做停留便能加速通过红绿灯路口。并且当前 Optsize 等于车队长度，所以不需要进行分裂、合并等操作。

距离流程如表 4-2：

表 4-2 wait_stage 中的绿灯到达例子 1 展示

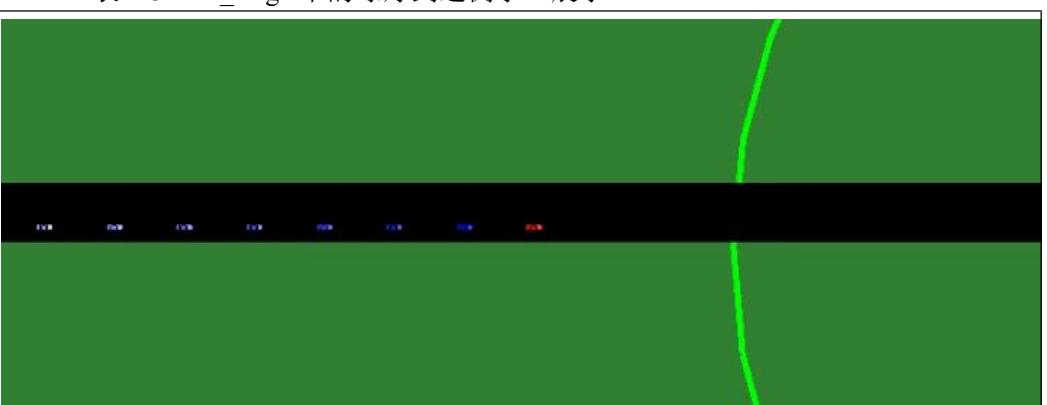
1、初始化阶段：		 ① 车队还未进入 RSU 覆盖区域之前进行协同驾驶	
车队长度 (len)	8		
推荐速度 (V_{ref})	无		
Optsize	8		
车队个数	1		
机动操作	协同驾驶	 ② 收到 RSU 信号中的 optsize、 V_{ref} ，进行车队减速行为	
2、车队进入 RSU 区域阶段：			
车队长度 (len)	8		
推荐速度 (V_{ref})	3.70		
Optsize	8		
车队个数	1		
机动操作	减速		

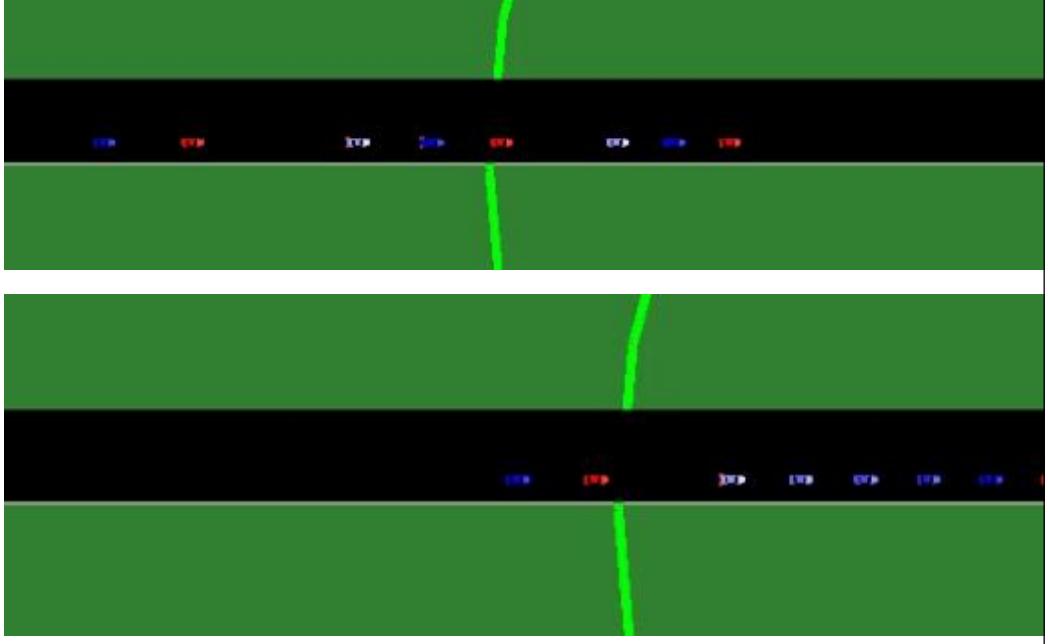
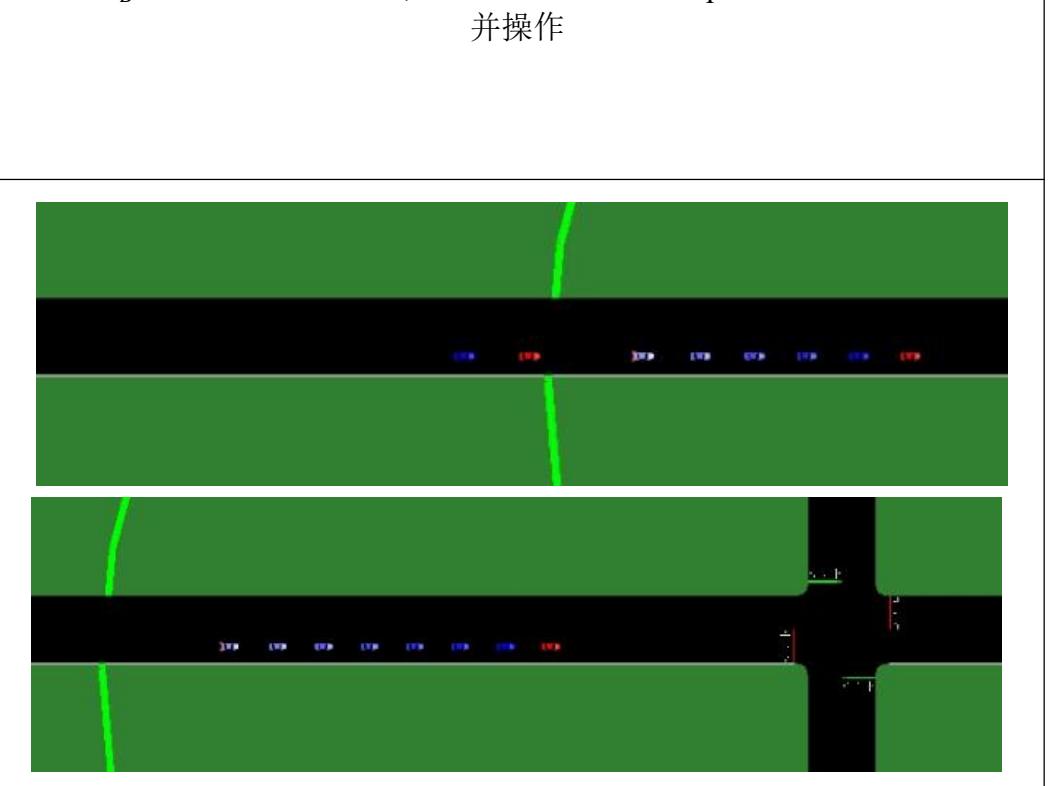
3、车队达到 V_{ref} , 匀速行驶:		
车队长度 (len)	8	
推荐速度 (V_{ref})	无	
Optsize	8	
车队个数	1	
机动操作	协同驾驶/ 匀速	③车队进行协同匀速驾驶
4、车队过红绿灯阶段		
车队长度 (len)	8	
推荐速度 (V_{ref})	无	
Optsize	8	
车队个数	1	
机动操作	加速	④完成整个车队的过红绿灯行为

4.2.3 wait_stage 中的绿灯到达例子 2 展示

当车队在 wait_stage 中的绿灯阶段进入 RSU 覆盖区域, 由于加速到最大速度 V_{max} 都不能顺利通过红绿灯路口, 因而进行减速机动, 旨在等待到下一个绿灯时刻刚好驾驶到红绿灯路口线, 不做停留便能加速通过红绿灯路口。但在这种情况下, Optsize 大于单个车队长度, 所以需要进行合并操作, 将多个小型车队合并成一个车队, 从而减少控制成本。距离流程如表 4-3:

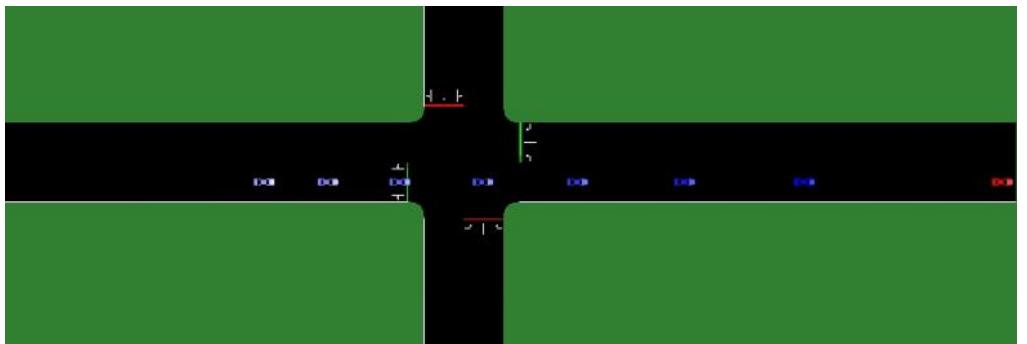
表 4-3 wait_stage 中的绿灯到达例子 2 展示

1、初始化阶段:		
车队长度 (len)	8	
推荐速度 (V_{ref})	无	
Optsize	3	
车队个数	3	
机动操作	分裂	①车队根据初始化 optsize 进行分裂, 分裂成 3 个车队
2、车队 C_A 进入 RSU 区域阶段:		
车队长度 (len)	C_A	3
	C_B	3
	C_C	2
推荐速度 (V_{ref})	C_A	3.79
	C_B	无
	C_C	无
Optsize	8	②车队进入 RSU 区域阶段, 收到 RSU 信号中的 optsize、 V_{ref} , 根据 V_{ref} 进行减速
车队个数	3	

机动操作	C_A	减速	
	C_B	匀速	
	C_C	匀速	
3、车队 C_B 进入 RSU 区域阶段，与 C_A 进行合并：			
车队长度 (len)	C_A	3	
	C_B	3	
	C_C	2	
推荐速度 (V_{ref})	C_A	3.79	
	C_B	5.37	
	C_C	无	
Optsize		8	
车队个数		3	
机动操作	C_A	匀速	
	C_B	合并	
	C_C	匀速	
4、车队 C_C 进入 RSU 区域阶段，同理与 C_A 进行合并：			
车队长度 (len)	C_A	6	
	C_C	3	
	C_A	3.79	
推荐速度 (V_{ref})	C_B	5.37	
	C_C	6.98	
	Optsize	8	
车队个数		2	
机动操作	C_A	匀速	
	C_C	合并	

③ 车队 C_B 进入 RSU 区域阶段, 收到 RSU 信号中的 optsize 为 8, 可以进行合并操作

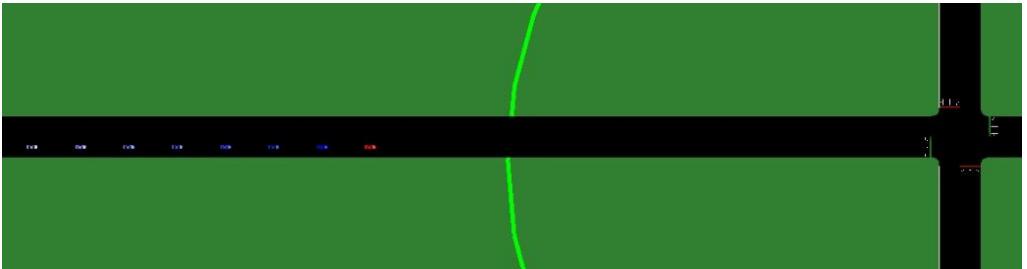
④ 车队 C_C 进入 RSU 区域阶段, 收到 RSU 信号中的 optsize 为 8, 可以进行合并操作

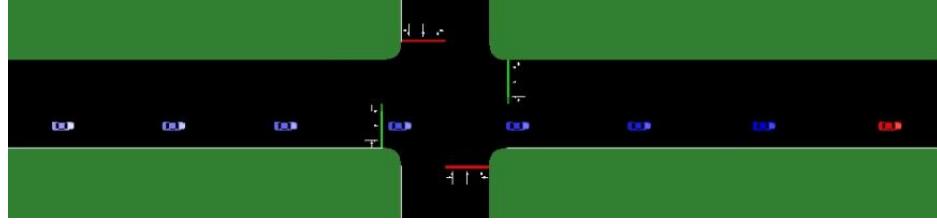
5、车队达到V_{ref}, 匀速行驶:		
车队长度 (len)	8	
推荐速度 (V_{ref})	3.79	
Optsize	8	
车队个数	1	
机动操作	协同驾驶/ 匀速	⑤车队进行协同匀速驾驶
6、车队过红绿灯阶段		
车队长度 (len)	8	
推荐速度 (V_{ref})	无	
Optsize	8	
车队个数	1	
机动操作	加速	⑥完成整个车队的过红绿灯行为

4.2.4 go_stage 中的红灯到达例子展示

当车队在 go_stage 中的红灯阶段进入 RSU 覆盖区域，由于加速到最大速度 V_{max} 能够顺利通过红绿灯路口，因而进行加速机动，旨在等待到下一个绿灯时刻刚好驾驶到红绿灯路口线，不做停留便能加速通过红绿灯路口。并且当前 Optsize 等于车队长度，所以不需要进行分裂、合并等操作。距离流程如表 4-4:

表 4-4 go_stage 中的红灯到达例子展示

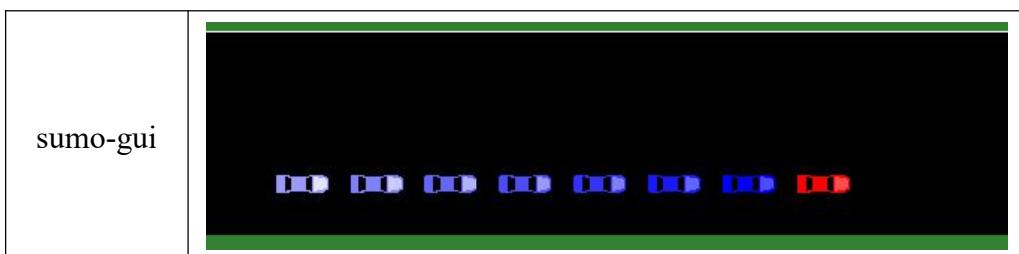
1、初始化阶段:		
车队长度 (len)	8	
推荐速度 (V_{ref})	无	
Optsize	8	
车队个数	1	
机动操作	协同驾驶	①车队还未进入 RSU 覆盖区域之前进行协同驾驶

2、车队进入 RSU 区域 阶段:		
车队长度 (len)	8	②收到 RSU 信号中的 optsize、 V_{ref} , 进行车队加速行为
推荐速度 (V_{ref})	20.00	
Optsize	8	
车队个数	1	
机动操作	加速	
3、车队达到 V_{ref} , 匀速行驶:		
车队长度 (len)	8	③车队达到 V_{ref} 之后, 进行协同匀速驾驶
推荐速度 (V_{ref})	无	
Optsize	8	
车队个数	1	
机动操作	协同驾驶/ 匀速	
4、车队过红绿灯阶段		
		④完成整个车队的过红绿灯行为

4.3 车队群密钥方案演示

车队群密钥方案的演示结合上一节的 5.4.1 例子进行, 由于群密钥的分发、使用群密钥加密主要是控制台的输出, 因此我们在控制台中进行查看。

1、车队组建时群密钥的分发



控制台输出	<pre>leader--veh SM4 key: 67c6697351ff4aec29cdbaabf2fbe346 veh.2 received sm4 key veh.2 SM4 key: 67c6697351ff4aec29cdbaabf2fbe346 veh.5 received sm4 key veh.5 SM4 key: 67c6697351ff4aec29cdbaabf2fbe346 veh.6 received sm4 key veh.6 SM4 key: 67c6697351ff4aec29cdbaabf2fbe346 veh.3 received sm4 key veh.3 SM4 key: 67c6697351ff4aec29cdbaabf2fbe346 veh.7 received sm4 key veh.7 SM4 key: 67c6697351ff4aec29cdbaabf2fbe346 veh.4 received sm4 key veh.4 SM4 key: 67c6697351ff4aec29cdbaabf2fbe346 veh.1 received sm4 key veh.1 SM4 key: 67c6697351ff4aec29cdbaabf2fbe346</pre>
-------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

模拟开始时在地图中插入长度为 8 的车队，此时由车队队长（SUMOID 为 veh）向队员们发送 CERT_REQ，收到队员们的 SM2 公钥证书后进行验证，验证通过后提取出 SM2 公钥，用其对自己生成的 SM4 密钥进行加密分发。控制台输出可以看到车队中成员车辆收到了 SM4 群密钥后进行解密后的结果都与管理员 veh 相同，说明车队共享了一个相同的 SM4 群密钥。

2、管理员发送 SM4 密钥加密后的测试信息

sumo-gui	
控制台输出	<pre>veh sent secret message: 31323334353637383132333435363738 veh.1 received secret message: 31323334353637383132333435363738 veh.2 received secret message: 31323334353637383132333435363738 veh.3 received secret message: 31323334353637383132333435363738 veh.4 received secret message: 31323334353637383132333435363738 veh.5 received secret message: 31323334353637383132333435363738 veh.6 received secret message: 31323334353637383132333435363738 veh.7 received secret message: 31323334353637383132333435363738</pre>

模拟时间在 3s 之后，管理员 veh 向所有通信范围内的车辆发送加密后的密文信息，其明文是“31323334353637383132333435363738”，在控制台输出可以看到车队中其他车辆收到信息后进行解密，得到相同的结果。

3、车队分裂后密钥更新

sumo-gui	
----------	--------------------------------------------------------------------------------------

控制台输出	<pre> platoon veh is splitting from 5 veh: front leader change key... veh.5: back leader generate key... veh.2 received sm4 key veh.2 SM4 key: 7cc254f81be8e78d765a2e63339fc99a veh.3 received sm4 key veh.3 SM4 key: 7cc254f81be8e78d765a2e63339fc99a veh.4 received sm4 key veh.4 SM4 key: 7cc254f81be8e78d765a2e63339fc99a veh.1 received sm4 key veh.1 SM4 key: 7cc254f81be8e78d765a2e63339fc99a veh.6 received sm4 key veh.6 SM4 key: 66320db73158a35a255d051758e95ed4 veh.7 received sm4 key veh.7 SM4 key: 66320db73158a35a255d051758e95ed4 </pre>
-------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

前方队长 veh 更新密钥，后方队长 veh.5 生成新的密钥，分别进行分发。结果可看出 veh.1、veh.2、veh.3、veh.4 拥有与 veh 相同的群密钥；veh.6、veh.7 拥有与 veh.5 相同的群密钥。

4、车队分裂后管理员 veh.5 发送测试信息

sumo-gui	
控制台 输出	<pre> veh.5 sent secret message: 31323334353637383132333435363738 veh.6 received secret message: 31323334353637383132333435363738 veh.4 received secret message: 041d5896bdb32b5997c049fdeee42e19 veh.7 received secret message: 31323334353637383132333435363738 veh.3 received secret message: 041d5896bdb32b5997c049fdeee42e19 veh.2 received secret message: 041d5896bdb32b5997c049fdeee42e19 veh.1 received secret message: 041d5896bdb32b5997c049fdeee42e19 veh received secret message: 041d5896bdb32b5997c049fdeee42e19 </pre>

veh.5 发送测试信息，明文同样是“31323334353637383132333435363738”其车队的成员 veh.6、veh.7 成功解密获得相同明文，另一个车队成员 veh、veh.1、veh.2、veh.3、veh.4 并不能解密得到相同明文，因为两个车队是不同的群密钥。

4.4 模拟结果性能分析

本节主要针对车队在采用交叉路口车队管理协议（Intersection Platoon Management, IPM）时具体的行为进行分析，统计了车队通过红绿灯路口时的指标：通行时间、尾气排放量，与自由车辆以及不采用 IIPM 方案的车队进行测试比较。

4.4.1 车队在路口处行为分析

模拟仿真程序中的系统参数见表 4-5。本作品提出的交叉路口的车队管理协议包含了 RSU 为交叉路口处的车队提供最佳速度与最佳长度的建议，相较于不使用车队形式或者是不使用 IPM，该方案的优势在于针对不同的信号灯情况以及车队情况提出了利于减少通行时间和尾气排放量的建议速度轨迹和车队长度。因此本小节为了更清晰的展示，主要针对在路口区域进行了车队机动的两个场景（4.4.2 中的场景 1 和场景 3）进行行为分析。

表 4-5 模拟程序的重要参数

参数	描述	值	单位
Vmax	道路最大限速	20	m/s
MaxAccel	CACC 控制策略 车辆最大加速度	3	m/s^2
MaxDecel	CACC 控制策略 车辆最大减速度	5	m/s^2
TP	车队间车头时距	3.5	s
TG	车队内车头时距	1.2	s
VehicleBeaconInterval	车辆发送信标时间间隔	0.1	s
RSUBeaconInterval	RSU 发送信标时间间隔	1	s
MaxIntfDist	信号覆盖距离	200	m

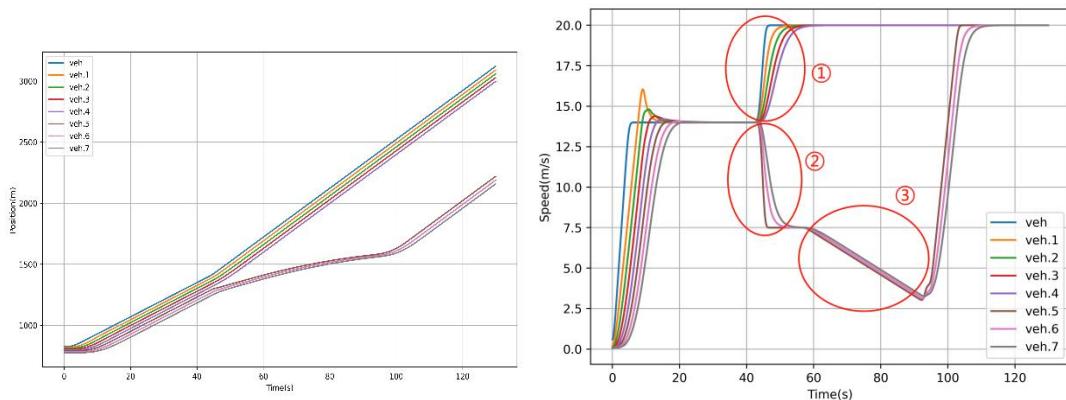
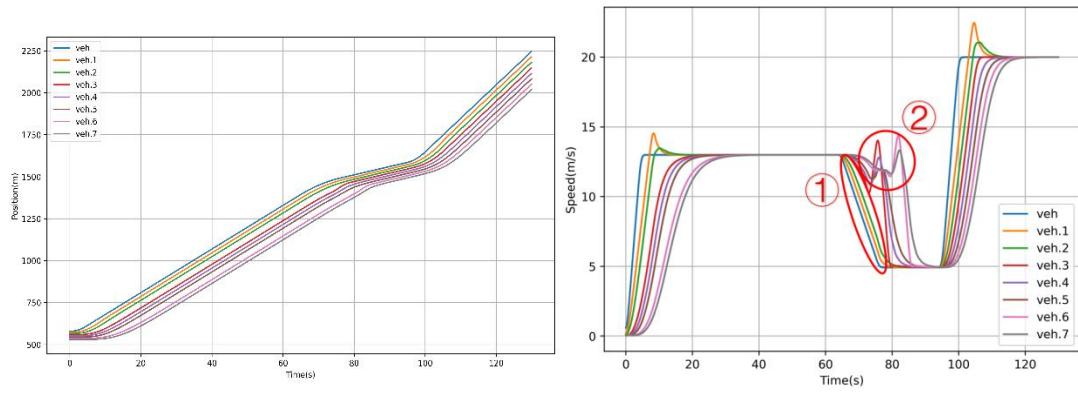


图 4-3 场景 1 轨迹图



(a) 位置-时间轨迹图

(b) 速度-时间轨迹图

图 4-4 场景 2 轨迹图

表 4-6 车队变换数据

(a) 场景 1

时间	vehID	状态变化	机动类型	备注
43.70	veh	leader→sendSplitReq	分裂开始/加速	veh 收到 RSUBeacon
45.55	veh.5	follower→leader	分裂结束	车队表示: *** *****
47.65	veh.5		减速	veh.5 收到 RSUBeacon

(b) 场景 3

时间	vehID	状态变化	机动类型	备注
67.54	veh		减速	veh 收到 RSUBeacon
73.88	veh.3	leader→sendMergeReq	合并开始	车队表示: ** *** ***
76.83	veh.3	leader→follower	合并结束	车队表示: ** *****
80.88	veh.6	leader→sendMergeReq	合并开始	车队表示: ** *****
83.45	veh.6	leader→follower	合并结束	车队表示: *****

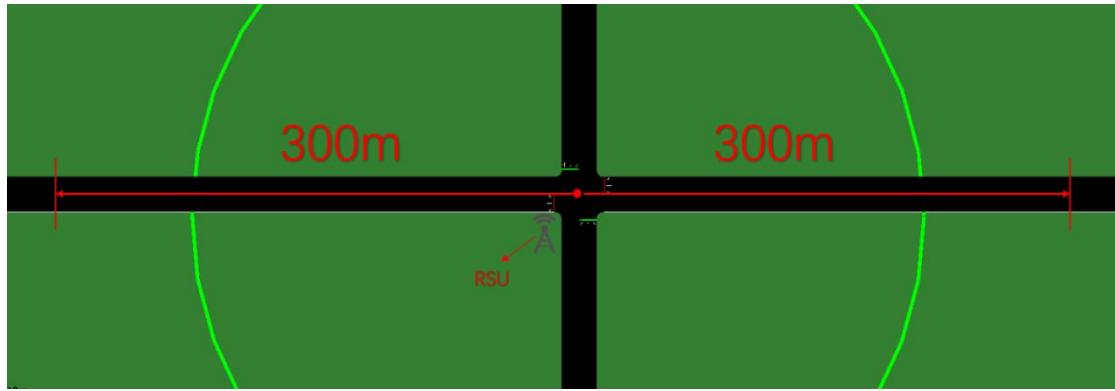
图 4-3 和图 4-4 分别是车队在场景 1（车队在绿灯时到达路口区域，阶段为 GO-STAGE，RSU 计算的最佳长度小于车队长度，进行分裂机动）与场景 3（车队在红灯时到达路口区域，阶段为 WAIT-STAGE，RSU 计算的最佳长度大于车队长度，多个车队连续到达，进行合并机动）的 8 辆车的位置-时间轨迹图以及速度-时间轨迹图。表 4-6 展示了这两个场景下的具体车队机动操作以及进行机动时状态变换情况。

图 4-3 (b) 中, ①处为 veh 为队长的车队在 RSU 给出的最佳长度为 5 进行分裂后的前方长度为 5 的车队的加速过程, veh 同时也收到了推荐速度 20m/s, 因此分裂的同时以最大加速度进行加速, 为了尽可能快速地通过路口。②处为分裂后 veh.5 为新的队长, 分裂机动后以最大加速度进行减速的过程, 这是因为在设计车队的基础机动时, 为了尽可能让机动快速安全地完成, 加速度通常取最大值, 需要注意在交叉路口处的分裂与普通的分裂机动有所不同, 分裂后的后方车队是无法在当前轮次通过绿灯的, 因此会进行减速操作, 从而不再为了保持 TP (车队间车头时距) 而再次加速, 而是直接减速至一个固定速度值 7.5, 等待与 RSU 的交互。③处为以 veh.5 为队长, 长度为 3 的车队在收到 RSU 的推荐速度后以一个较小的推荐加速度进行减速的过程, 较小的加速度是为了避免剧烈的速度变化, 达到降低油耗和尾气排放量的目的。

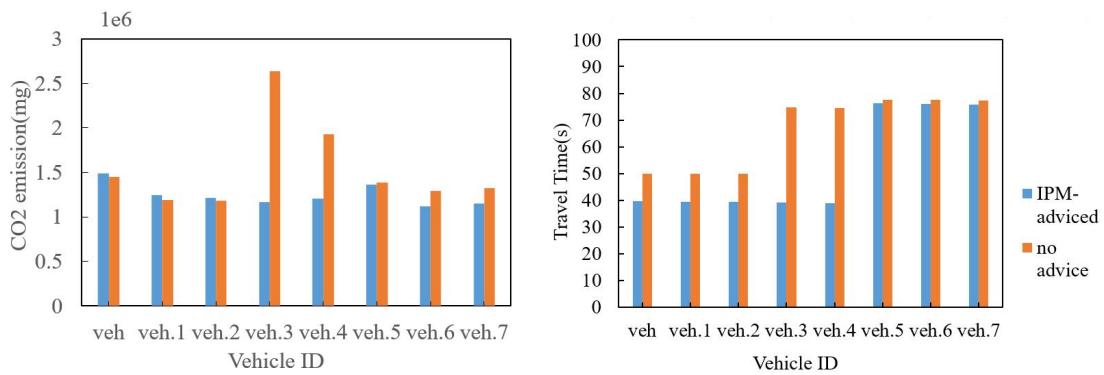
图 4-4 (b) 中, ①处为 veh 为队长、长度为 3 的车队在进入路口区域后, 收到 RSU 给出的最佳车队长度 25、推荐速度 5 后的减速过程。接下来后方的两个长度为 3 和 2 的车队分别前后进入路口区域, 表现为图中的②处, 两个尖刺为两个车队的合并过程, 为了使机动操作快速安全, 均是以最大加速度完成的合并机动: 首先加速减小车头时距到 TG 值, 然后减速到与前方合并车队相同的速度以保持 TG。注意 veh 队长车队以较小推荐加速度将速度减小至推荐速度, 再以推荐速度滑行至等待线, 绿灯后加速通过, 这同样也是出于节省油耗和减小尾气排放量的考虑。

4.4.2 车队在路口处性能与指标分析

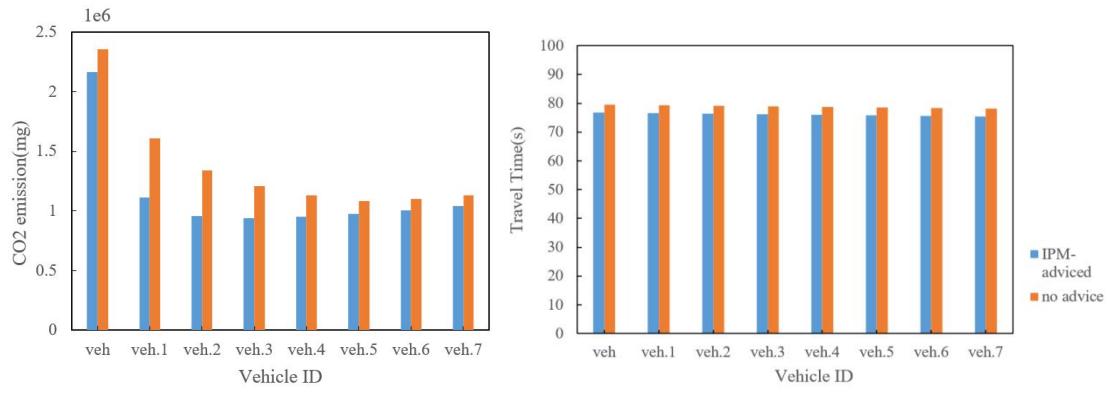
本节中测试了 (1) 自由车辆, 无车辆编队行驶时通过红绿灯交叉路口的通行时间和二氧化碳排放量; (2) 车辆编队行驶, 但不应用 IPM, 即不提供最佳速度轨迹和最佳长度, 通过交叉路口的通行时间和二氧化碳排放量; (3) 车队应用本作品提出的 IPM 通过交叉路口的通行时间和二氧化碳排放量。

图 4-5 记录通行时间与 CO₂ 排放量的距离

针对（2）和（3），由于均是车辆编队行驶，我们统一固定每次测试的车队长度为 8，车队的初速度在各个场景中的两种方法都相同，设置为 13m/s，在 4.2 节中的四个场景分别进行模拟仿真，最后记录每一辆车的路口通行时间和二氧化碳排放量。在第三章中定义了“路口区域”为 RSU 的信号覆盖范围（200m 通信距离），为了能够确保完整地收集行驶过程中的数据，由于车队的初速度等前置条件都是相同的，区别只在于是否启用了 IPM，因此在测试中定义路口通行时间为通行路口中心左右 300 米距离的时间，如图 4-5 所示，二氧化碳排放量同样也是每一辆车在这 600 米的距离范围内所排放的总二氧化碳量。



(a) 场景 1



(b) 场景 2

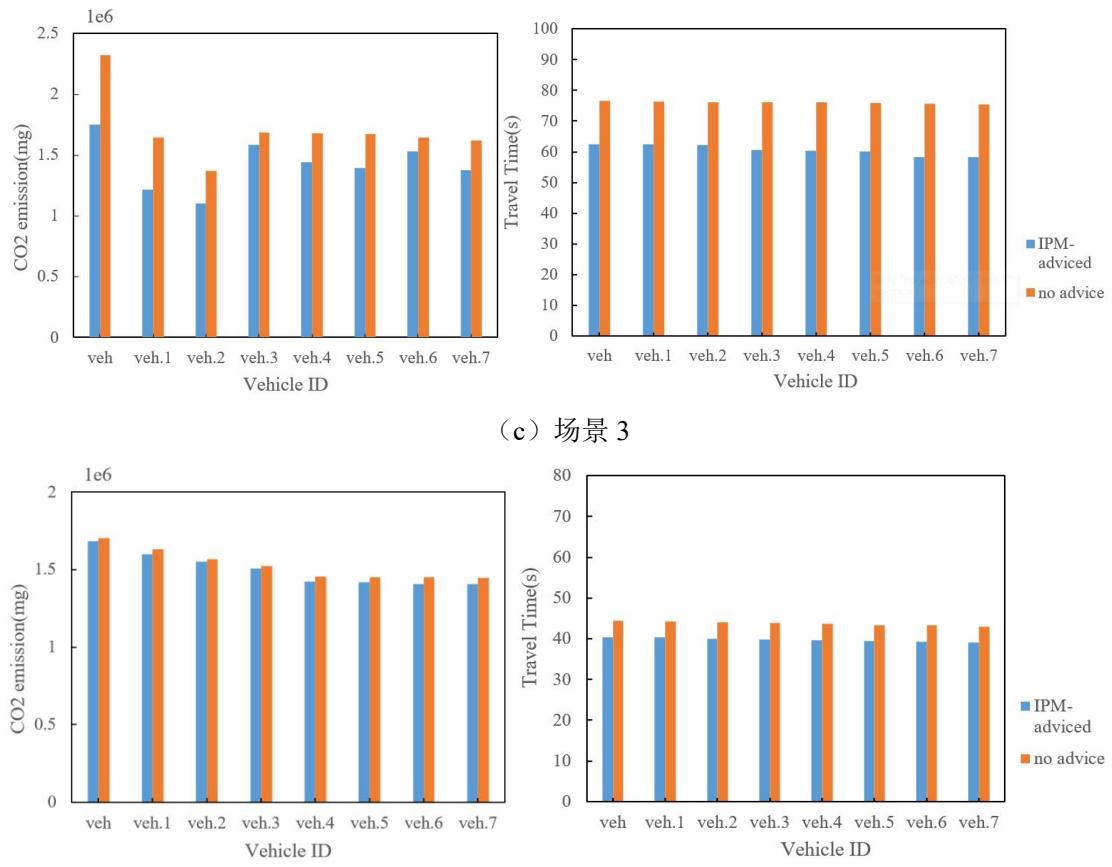


图 4-6 车队通过路口采用 IPM 方案与 no advice 方案通行时间与 CO2 排放对比

图 4-6 是四个场景中两种方法的通行时间和 CO2 排放量的对比结果图, 可以看出应用了 IPM 的车队在通过红绿灯路口时, 在通行时间变少的情况下, CO2 排放量也减少了。在场景 1 中, 不应用 IPM 会导致长度为 8 的车队在 veh.3 处遇到红灯无法通过路口, 第四辆车就会在路口处急剧减速停车等待, 因此它的 CO2 排放量也相较于其他车辆很高, 而应用了 IPM 的车队会根据最佳车队长度提前在 veh.5 处进行分裂, 后方车队再次根据 IPM 以较小的加速度减速, 这样会就可以达到减少 CO2 排放的目的, 并且由于到达等待线时车辆速度不会降为 0, 绿灯后通过时间也会有所减少。在场景 3 中实现的是等待时多个车队的合并, 根据结果可以发现这个操作可以显著降低通行的时间, 并且由于车队合并, 在车队中队员的身份可以更加省油地行驶, 因此油耗也会更加友好。

表 4-7 三种模式的数据对比

	CO2 排放量 (mg)	通行时间 (s)
自由车辆	2050621	98.2504

编队行驶无 IPM	1548952	66.1074
编队行驶 IPM	1326502	56.5623

表 4-7 展示了三种模式下车辆通过红绿灯路口的 CO2 排放量以及通行时间的统计结果。其中自由车辆方案是在距离交叉路口左侧 800 米处以 $1/2.35=0.426$ veh/s 的速率插入 200 辆车辆进入道路中, 0.426 veh/s 选取理由是 TP (3.5) 与 TG (1.2) 的平均值的倒数, 为了尽可能将初始条件设置为与编队行驶相同, 插入车辆的初速度分布为, 以编队车辆的初速度 13m/s 为均值, 1m/s 为标准差的正态分布, 模拟结束后计算出每一辆车在以路口中点为中心的 600m 内的 CO2 排放量以及通行时间。编队行驶的两个方案也是相似的配置, 但由于车队通信范围以及其他参数的限制, 这里的车队的最大长度为 8, 插入车队的数量为 25 个, 位置与初速度分布都与自由车辆相同。表中的结果显示采用本作品提出的 IPM 行驶通过红绿灯路口在 CO2 排放方面比自由车辆减少了 **35.2%**, 比无 IPM 的编队行驶减少了 **14.36%**; 在通行时间方面 IPM 比自由车辆行驶减少了 **42.46%**, 比无 IPM 的编队行驶减少了 **14.46%**。

5. 创新与特色

1. 实现车辆编队行驶

为了解决如何解决城市交通拥堵、交通事故问题以及汽车尾气排放问题，本项目提出了车队编排行驶。将单一车辆汇总为一个车队，结合车对车通信技术（V2V）和车辆与路边基础设施通信技术（V2I），协同车辆基础设施系统（CVIS）等技术完成车辆编队行驶，使得整个车队进行集体地行驶、停靠、加速、减速、变道等等。同时，解决了车队的纵向控制问题，如合并、分裂、变道、加入、离开车队六个基础机动操作的设置，对每个机动的执行步骤进行了完整的方案设计，包括如何在两个或者多个车队之间进行机动请求、机动响应以及机动执行等操作。以确保在更加复杂的交通情况中，如交叉路口的场景，车队能够做出正确、安全、高效的行为。

车队中的车辆比具有相同速度的普通车辆具有更小的车间距，因此可以提高交通吞吐量；由于车队中车辆的速度变化相对较小，且车间的速度差更小，达到更高的安全性；车队行驶由于更平稳的速度变化和减小空气阻力而减小了燃料消耗和尾气排放。**相比于普通的单个车辆行驶来说采用车辆编队行驶提高了交通吞吐量、行驶安全性以及降低了燃油消耗量和尾气排放量。**

2. 设计并实现了针对城市交叉路口场景的车队管理协议

本作品实现了车队在交叉路口场景中的安全管理协议，该协议为车队在交叉路口处进行速度轨迹规划以及最佳长度的计算，由此设计了合适的机动操作，使得车队进行正确、安全、高效的队形变换。具体来说，车队管理协议根据实时的车队速度、车队与路口距离进行计算，提供了车队在路口区域行驶的最佳速度轨迹，在必要时对车队进行机动操作执行，例如根据绿灯剩余秒数进行车队分裂，减少车队在路口停车的次数，实现车队高效地通过红绿灯路口。

车队管理协议减少了车队通过路口的时间，避免在路口长时间停车，减少气体排放，从而提高交通效率。车辆编队技术与中国减少碳排放和促进可持续交通发展的战略目标相一致，**针对国内交通压力巨大的情况，本作品提出的针对交叉路口场景的车队管理协议通过降低车辆的尾气排放量，进一步支持国家的碳中和**

目标，优化交通流，减缓交通拥堵，进一步提高道路使用率。

3. 利用 SM2、SM4 国密算法进行数字签名和加密

本作品中设计了基于国密的群加密方法来实现车队内的加密通信。在车队场景中，群加密能够提供重要的安全保障和高效的通信管理。通过使用群密钥，我们能够确保只有车队的成员能够解读车队内部的通信，这可以防止敏感信息（例如路线、速度、位置等）被未授权的第三方获取，从而提高车队的安全性。同时，车队中的群密钥方案实现了完整性、前行安全性、后向安全性、抵抗密钥控制性，使得车队编排行驶更加安全，免受第三方窃听攻击。

SM2、SM4 算法与 RSA、AES 等同类型密码算法相比具有加密强度更高、安全性能更强、传输速度更快、性能开销更小等优点，对于车队中的车间加密通信应用场景具有无可比拟的优势。此外，国密算法作为我国完全自主研发的国产密码算法，由我国掌握核心科技，**推广国密算法对于我国商用密码产业的发展将起到强有力的推进作用，能够鼓励我国商用密码科技创新能力和密码管理体系进一步提高和完善。**