Design and Analysis of Delay-Tolerant Intelligent Intersection Management

BOWEN ZHENG, University of California, Riverside
CHUNG-WEI LIN, National Taiwan University
SHINICHI SHIRAISHI, TOYOTA InfoTechnology Center
QI ZHU, Northwestern University and University of California, Riverside

3

The rapid development of vehicular network and autonomous driving technologies provides opportunities to significantly improve transportation safety and efficiency. One promising application is centralized intelligent intersection management, where an intersection manager accepts requests from approaching vehicles (via vehicle-to-infrastructure communication messages) and schedules the order for those vehicles to safely crossing the intersection. However, communication delays and packet losses may occur due to the unreliable nature of wireless communication or malicious security attacks (e.g., jamming and flooding), and could cause deadlocks and unsafe situations. In our previous work, we considered these issues and proposed a delay-tolerant intersection management protocol for intersections with a single lane in each direction. In this work, we address key challenges in efficiency and deadlock when there are multiple lanes from each direction, and propose a delay-tolerant protocol for general multi-lane intersection management. We prove that this protocol is deadlock free, safe, and satisfies the liveness property. Furthermore, we extend the traffic simulation suite SUMO with communication modules, implement our protocol in the extended simulator, and quantitatively analyze its performance with the consideration of communication delays. Finally, we also model systems that use smart traffic lights with various back-pressure scheduling methods in SUMO, including the basic back-pressure control, the capacity-aware back-pressure control, and the adaptive max-pressure control. We then compare our delay-tolerant intelligent intersection protocol with smart traffic lights that use the three back-pressure scheduling methods, in the case of a network of interconnected intersections. Simulation results demonstrate that our approach significant outperforms the smart traffic lights under normal operation (i.e., when the communication delay is not too large).

CCS Concepts: • **Security and privacy → Domain-specific security and privacy architectures**; • **Applied computing → Transportation**; • **Computer systems organization → Embedded and cyber-physical systems**;

Additional Key Words and Phrases: Cyber-physical security, connected and autonomous vehicles, vehicular network, V2X, autonomous intersection, timing attack, back-pressure control, smart traffic lights

The authors gratefully acknowledge support from the U.S. National Science Foundation (awards 1834701, 1834324, and 1839511), Moxa Inc., MediaTek Inc., Ministry of Education in Taiwan (grant NTU-107V0901), and Ministry of Science and Technology in Taiwan (grant MOST-108-2636-E-002-011).

Authors' addresses: B. Zheng is with University of California, Riverside and Pony.AI, Inc., 3501 Gateway Blvd, Fremont, CA, 94538; email: bzhen003@ucr.edu; C.-W. Lin is with National Taiwan University, No. 1, Sec. 4, Roosevelt Rd., Taipei, Taiwan 10617; email: cwlin@csie.ntu.edu.tw; S. Shiraishi is with Toyota Research Institute - Advanced Development, Inc., Nihonbashi Muromachi Mitsui Tower 18F, 3-2-1 Nihonbashi muromachi, Chuo-ku, Tokyo, Japan 103-0022; email: shinichi.shiraishi@tri-ad.global; Q. Zhu is with Northwestern University and University of California, Riverside, 2145 Sheridan Rd., Evanston, IL, 60208; email: qzhu@northwestern.edu.

2378-962X/2019/11-ART3 $15.00
https://doi.org/10.1145/3300184

ACM Reference format:
Bowen Zheng, Chung-Wei Lin, Shinichi Shiraishi, and Qi Zhu. 2019. Design and Analysis of Delay-Tolerant Intelligent Intersection Management. *ACM Trans. Cyber-Phys. Syst.* 4, 1, Article 3 (November 2019), 27 pages. https://doi.org/10.1145/3300184

1 INTRODUCTION

Intersection management is one of the most important and challenging problems in transportation, as it plays a critical role in both traffic safety and efficiency [10, 23]. Traditionally, an intersection is controlled by either traffic lights with a pre-defined schedule or by stop signs. It is difficult for such systems to adapt to real-time traffic [10]. To make the traffic lights "smarter," several works try to adapt traffic signals to real-time situations by estimating traffic conditions [18, 22, 25] or gathering congestion information (e.g., queue length) from neighboring intersections [15, 16, 27, 28, 30, 32]. However, these smart traffic light systems still face performance bottlenecks, as the scheduling is not fine grained to vehicle level and it is difficult to adjust the scheduling period under different traffic patterns.

With the development of autonomous driving and vehicular communication technologies, intelligent intersections leveraging vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communications have shown great promise. More specifically, in an intelligent intersection based on vehicular network communication, traffic lights are replaced with wireless messages exchanged among vehicles and infrastructures. Based on these messages, vehicles can take wiser actions to improve traffic safety and efficiency. Such systems can be either centralized or distributed. In a centralized system, an intersection manager accepts requests from vehicles and decides the order for them to cross the intersection [5, 6, 12, 19, 20, 35]. In a distributed system, vehicles negotiate among themselves to decide the crossing order [3, 4, 24].

The previous works on vehicular network based intelligent intersections assume perfect communication among vehicles and infrastructures, and do not explicitly address communication delays. However, significant packet delays and losses could happen in a vehicular network under dense traffic [9, 13, 31]. The packet delay can be as long as several hundreds of milliseconds in the worst case [31]. Furthermore, the issues could be even more severe when the network is under malicious jamming or flooding attacks [8, 34]. In these cases, the previous works that do not consider communication delays or losses may lead to system deadlock or unsafe situations.

Therefore, we believe that it is essential to consider communication delays and losses in designing vehicular network based intelligent intersections, and to ensure that the system design satisfies the following properties:

- *Deadlock free*: The system should not have any deadlocks, even when there are communication delays or packet losses.
- *Liveness*: Every vehicle that sends requests should eventually cross the intersection, as long as the communication delays can be bounded (taking into consideration resending messages after packet losses), and the intersection manager will try to schedule vehicles from any lane eventually (i.e., no starving in scheduling policy).
- *Safety*: Vehicles with conflicting routes (i.e., routes that may cross each other within the intersection) may never enter the intersection at the same time.[1]

[1]It should be noted that the vehicles are assumed to have autonomous driving capabilities and may detect or even avoid incoming collisions in most cases. Nevertheless, conflicting routes could still lead to unsafe situations given the limitations of autonomous driving and are likely to cause deadlocks even without accidents. Furthermore, we also make this assumption based on practical consideration of vehicle passengers' mental acceptance.

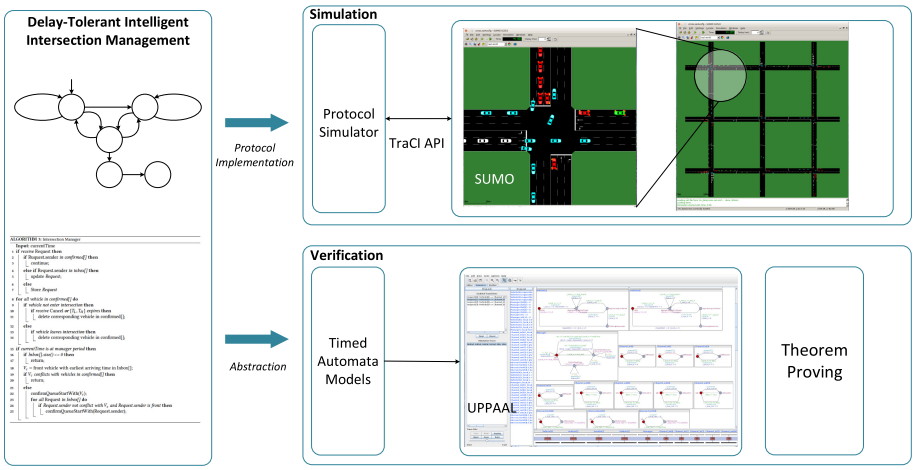


Fig. 1. Design, simulation, and verification of delay-tolerant multi-lane intelligent intersection management.

In our previous work [33], we proposed a delay-tolerant protocol for a four-way intersection with a *single lane in each direction*. We verified its deadlock-free, liveness, and safety properties, and analyzed its performance based on simulations.

In this article, we present a delay-tolerant intelligent intersection management protocol for *general multi-lane intersections*, develop modeling, simulation, and verification techniques for analyzing the protocol's properties and performance (Figure 1), and also implement a smart traffic light system for comparison. More specifically, the main contributions of this article include the following:

- We design a general delay-tolerant protocol for intelligent intersection management. The protocol addresses key efficiency and deadlock challenges when there are multiple lanes from each direction.
- We extend the widely used traffic simulation suite SUMO [1] with communications modules and implement our protocol in the extended simulator. We analyze the protocol's performance for a single intersection and for a network of interconnected intersections, with the consideration of communication delays.
- We verify the deadlock-free, liveness, and safety properties of our protocol.
- We also implement a smart traffic light system with back-pressure scheduling in SUMO, and compare its performance with our vehicular network based protocol.

The rest of the article is organized as follows. In Section 2, we introduce background information, including smart traffic lights with back-pressure scheduling and basic ideas of vehicular network based intelligent intersections. In Section 3, we present our delay-tolerant protocol for general multi-lane intersections. We first present the system model and review of our previously proposed single-lane protocol, then we discuss the challenges in addressing multiple lanes, and finally we present the details of our multi-lane protocol. In Section 4, we verify the deadlock-free, liveness, and safety properties of our multi-lane protocol. In Section 5, we show the simulation results of our protocol and compare them with smart traffic lights. We conclude the article in Section 6.

2 BACKGROUND

Traffic lights have long been used to control traffic and ensure safety. Traditional traffic lights follow a pre-defined schedule that is sometimes optimized based on historic data [14]. However,

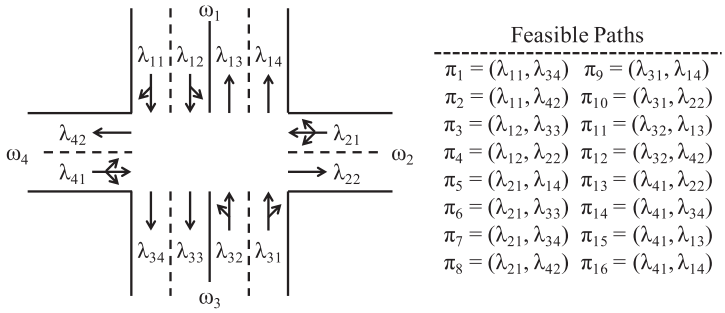


Fig. 2. An example intersection.

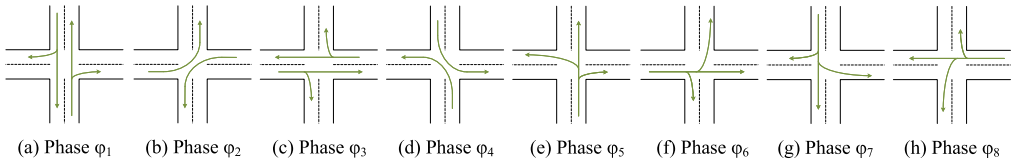


Fig. 3. Typical traffic light phases for a four-way intersection.

the pre-defined schedule cannot adapt well to varying traffic flows [32] and may not perform well under high traffic volumes and asymmetric traffic patterns. Recently, the concept of smart traffic lights has been proposed to improve traffic management performance [15, 16, 27, 28, 30, 32].

However, smart traffic lights still face bottlenecks, as the scheduling is not fine grained to vehicle level (microscopic traffic level) and it is not easy to adjust the scheduling period under different traffic patterns. The development of vehicular communication and autonomous driving provides a more fine-grained scheduling of the intersection and could achieve better performance. Both centralized and distributed intersection management have been proposed in the literature [3, 4, 12, 19, 20, 24, 35].

Next, we will first briefly describe the intersection model to help us introduce smart traffic lights (more details of the intersection model will be presented in Section 3 with our delay-tolerant protocol). We will then introduce the basic ideas of vehicular network based intelligent intersections.

2.1 Brief Intersection Model

In this article, we use the following definitions to represent an intersection (illustrated in Figure 2). An intersection $\mathcal{I} = (\mathcal{W}, \mathcal{P})$ consists of a set of ways $\mathcal{W} = \{\omega_1, \omega_2, \ldots, \omega_{|\mathcal{W}|}\}$ and a set of feasible paths $\mathcal{P} = \{\pi_1, \pi_2, \ldots, \pi_{|\mathcal{P}|}\}$. For a typical four-way intersection, the number of ways $|\mathcal{W}|$ equals to 4. Each way ω_i contains a set of lanes, denoted by $\mathcal{L}_i = \{\lambda_{i1}, \lambda_{i2}, \ldots, \lambda_{i|\mathcal{L}_i|}\}$. Then a feasible path π_k is defined as an ordered pair of lanes (i.e., $\pi_k = (\lambda_{ij}, \lambda_{i'j'})$). For each intersection, we can define a conflict table indicating whether two feasible paths conflict with each other (i.e., the two paths cross each other in the intersection). Taking the example in Figure 2, π_1 and π_9 do not conflict with each other, whereas π_3 and π_{12} conflict with each other.

2.2 Smart Traffic Lights

Smart traffic lights dynamically change phases based on current traffic condition (e.g., queue length of each lane). The set of possible phases is denoted as $\oplus = \{\phi_1, \phi_2, \ldots, \phi_{|\oplus|}\}$. Typical phases for a four-way intersection are depicted in Figure 3. The arrows in each phase denote the allowed paths based on the traffic light configuration in that phase. For example, in phase ϕ_1, only the traffic

signals for northbound and southbound traffic are green (going straight and turning right). In phase ϕ_2, the left turn signals for eastbound and westbound traffic are green. At each time slot (step), a smart traffic light system chooses a phase $\phi_i \in \oplus$ that can best reduce congestion based on current traffic conditions.[2]

Recent studies on smart traffic lights are mostly based on max-pressure and back-pressure control [15, 16, 27, 28, 30, 32]. In particular, back-pressure control is a greedy algorithm performed at each intersection locally. The main idea is to measure pressure based on queue lengths, and to release the pressures from high-pressure directions to low-pressure directions [16]. The hidden assumption is that queue lengths can be estimated by loop detectors and cameras, and then sent to the traffic light controller.

The *basic back-pressure control*, as introduced in other works [16, 27, 30], uses a linear pressure function $Pr(\lambda_i) = Q(\lambda_i)$, in which $Pr(\lambda_i)$ denotes the pressure on lane λ_i and $Q(\lambda_i)$ denotes the queue length of lane λ_i. If we use $\pi_{a,b} = (\lambda_a, \lambda_b)$ to denote a feasible path from upstream lane λ_a to downstream lane λ_b, we can compute the pressure difference $\Delta Pr_{\pi_{a,b}}$ as in Equation (1). $d_{\pi_{a,b}} \in \{0, 1\}$ is a binary variable indicating whether there are vehicles waiting at lane λ_a to leave from lane λ_a for lane λ_b.

$$\Delta Pr(\pi_{a,b}) = d_{\pi_{a,b}} max(Pr(\lambda_a) - Pr(\lambda_b), 0) \tag{1}$$

The basic algorithm of back-pressure control for smart traffic lights is illustrated in Algorithm 1, which is performed at every time slot (scheduling period). The algorithm first measures the pressure based on the linear function as described earlier (lines 2 and 3), then computes the pressure difference for each feasible path (lines 4 and 5). After that, the algorithm selects the phase ϕ_{out} that can maximize the objective function, which is the weighted sum of the pressure release from all feasible paths (lines 6 and 7). The weight $\mu_{\pi_{a,b}}$ for each path is the maximum number of vehicles that can go from lane λ_a to lane λ_b during the time slot. Take the example in Figure 2. In phase ϕ_1, no vehicle can move from lane λ_{31} to lane λ_{42} (left turn), and therefore the weight $\mu_{\pi_{31,42}}$ is 0. However, for path $\pi_{31,12}$, a non-zero weight $\mu_{\pi_{31,12}}$ denotes the maximum number of vehicles that can go from lane λ_{31} to λ_{12} (going straight) during the time slot.

ALGORITHM 1: Back-pressure control for intersection $\mathcal{I} = \{\mathcal{W}, \mathcal{P}\}$

Input: Queue lengths $Q(\lambda_i)$, $\forall \lambda_i \in \mathcal{L}$
Output: Traffic light phase ϕ_{out} for current time slot

1 **for** *time slot k* **do**
2 **for** *all lanes* $\lambda_i \in \mathcal{L}$ **do**
3 $Pr(\lambda_i) \leftarrow Q(\lambda_i)$;
4 **for** *all feasible paths* $\pi_{a,b} = (\lambda_a, \lambda_b) \in \mathcal{P}$ **do**
5 $\Delta Pr(\pi_{a,b}) \leftarrow d_{\pi_{a,b}} max(Pr(\lambda_a) - Pr(\lambda_b), 0)$;
6 $\phi_{out} \leftarrow \arg\max_{\phi_i \in \oplus} \sum_{\pi_{a,b} \in \mathcal{P}} \mu_{\pi_{a,b}} \Delta Pr(\pi_{a,b})$;
7 **return** Phase ϕ_{out} for time slot k;

Some extensions of the basic back-pressure control have also been presented in the literature [16, 21, 32]. These extensions are mainly proposed to solve the problems caused by the capacity of the road [16] and the turning rate estimation [21, 32]. Among variations, we consider the following two algorithms:

[2]Note that the term *phase* is commonly used in the literature (e.g., [15, 16, 28, 30]). There is not necessarily an order among phases—that is, any ϕ_i can be chosen at a time slot.

- *Capacity-aware back-pressure control*: Observing the problems caused by assuming infinite capacity in the basic back-pressure control, this method proposed in Gregoire et al. [16] normalizes the pressure equation to guarantee work conservation and mitigate congestion propagation. It can also improve the fairness for low-density traffic. The normalized pressure from Gregoire et al. [16] is shown in Equation (2), where $Q(\lambda_i)$ denotes the actual queue length of lane λ_i and C_{λ_i} denotes the capacity of the lane. The parameters m and C_∞ can be tuned to shape the pressure function. When traffic is sparse, the pressure function is approximately a linear function, which guarantees fairness and stability. As the function is convex, the slop of the pressure function grows with the increase of occupancy. The algorithm remains the same as the basic back-pressure control except for the change of the pressure function.

$$Pr(\lambda_i) = min\left(1, \frac{\frac{Q(\lambda_i)}{C_\infty} + \left(2 - \frac{Q(\lambda_i)}{C_\infty}\right)\left(\frac{Q(\lambda_i)}{C_{\lambda_i}}\right)^m}{1 + \left(\frac{Q(\lambda_i)}{C_{\lambda_i}}\right)^{m-1}}\right) \tag{2}$$

- *Adaptive max-pressure control*: This method is based on the series of works from Varaiya, in which in early works [27–29], the demand to the network is modeled with a constant average rate, the average turning ratio is pre-specified, and the vehicles are modeled as a queuing network with store and forward features. Network calculous or stochastic methods are used to analyze the models and guarantee stability and performance. In Lioris et al. [21], to reduce the assumption on the knowledge of the turning ratio in the traffic network, the authors further estimate the turning probabilities during the pressure calculation. The equation for estimating the turning rate $r(\lambda_l, \lambda_m)$ is shown in (3), where $a(\lambda_l, \lambda_m)$ represents the number of vehicles arriving at lane λ_l with destination λ_m during period t. λ_k denotes all possible destinations. With this turning rate estimation, the equation to calculate the pressure difference (namely Equation (1)) will be updated as Equation (4), where $Q(\lambda_a, \lambda_b)$ denotes the number of vehicles queued at link λ_a and waiting to go to link λ_b, and $O(\lambda_b)$ denotes the set of the out links from λ_b. The main algorithm is similar to the basic back-pressure control.

$$r(\lambda_l, \lambda_m) = \frac{\sum_t a(\lambda_l, \lambda_m)(t)}{\sum_k \sum_t a(\lambda_l, \lambda_k)(t)} \tag{3}$$

$$\Delta Pr(\pi_{a,b}) = Q(\lambda_a, \lambda_b) - \sum_{\lambda_p \in O(\lambda_b)} r(\lambda_b, \lambda_p) Q(\lambda_b, \lambda_p) \tag{4}$$

2.3 Vehicular Network Based Intelligent Intersection Management

As stated earlier, intelligent intersection management based on vehicular network communication can be classified into two major categories: centralized management and distributed management.

In centralized management, an intersection manager schedules the order for vehicles to cross the intersection [3, 12, 17, 19, 20, 24, 35]. In many of those approaches (e.g., [12, 17, 19]), the intersection is discretized to small grids. Even when two vehicles have conflicting routes, they may be allowed to both enter the intersection, as long as they do not share the same grid at the same time. The authors in Dresner and Stone [12] propose a protocol to address both autonomous and regular vehicles through V2I communication and virtual traffic lights. In Hausknecht et al. [17], the authors further optimize their approach for multiple intersections. The work in Jin et al. [19] focuses more on the fuel consumption and emission using a similar fine-grained scheduling strategy based on

grids. The work in Kowshik et al. [20] proves safety and liveness properties, without considering communication delays. In Zhu and Ukkusuri [35], linear programming is used to model vehicles as traffic flows and avoid conflicting routes by setting constraints.

The main idea for distributed intersection management is to require every vehicle broadcasting enter, cross, and exit packets with the identification of its current grid [4–6]. The authors in Azimi et al. [4–6] use wait-for graph and mathematical reasoning to prove their protocol is deadlock free. In Naumann et al. [24], the authors use Petri net models to prove the system to be deadlock-free. The authors in Ahmane et al. [3] also consider traffic smoothness.

These approaches in the literature do not explicitly address communication delays or losses. However, as discussed in Section 1, significant packet delays and losses may occur in dense traffic [9, 13, 31] or due to malicious jamming or flooding attacks [8, 34]. In such cases, these previous approaches may lead to system deadlock or unsafe situations.

Next, we will introduce the design, verification, and simulation of our delay-tolerant intelligent intersection management approach.

3 DELAY-TOLERANT INTELLIGENT INTERSECTION MANAGEMENT PROTOCOL

3.1 System Model

Our centralized intelligent intersection system includes four major components: intersection network, intersection manager, vehicles, and communication messages exchanged between vehicles and the intersection manager.

3.1.1 Intersection Network Model. A traffic network $\mathcal{N}$ contains a set of intersections $\mathcal{I} = \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_{|\mathcal{I}|}\}$ and a set of connections between the intersections $\mathcal{C} = \{\mathcal{C}_{\mathcal{I}_1,\mathcal{I}_2}, \mathcal{C}_{\mathcal{I}_1,\mathcal{I}_3}, \ldots\}$. An intersection $\mathcal{I}_k = (\mathcal{W}_k, \mathcal{P}_k)$ consists of a set of ways $\mathcal{W}_k = \{\omega_1^k, \omega_2^k, \ldots, \omega_{|\mathcal{W}|}^k\}$ and a set of feasible paths $\mathcal{P}_k = \{\pi_1^k, \pi_2^k, \ldots, \pi_{|\mathcal{P}|}^k\}$, as described in Section 2. Each way ω_i^k contains a set of lanes, denoted as $\mathcal{L}_i^k = \{\lambda_{i1}^k, \lambda_{i2}^k, \ldots, \lambda_{i|\mathcal{L}_i|}^k\}$. A feasible path π_l is defined as an ordered pair of lanes as $\pi_l = (\lambda_{ij}, \lambda_{i'j'})$. Each connection $\mathcal{C}_{\mathcal{I}_i,\mathcal{I}_j}$ is a set containing ordered pairs of lanes, with the first element denoting the lane from the source intersection and the second element denoting the lane to the destination intersection. For example, in Figure 4, the connection between $\mathcal{I}_1$ and $\mathcal{I}_2$ is represented by $\mathcal{C}_{\mathcal{I}_1,\mathcal{I}_2} = \{(\lambda_{22}^2, \lambda_{41}^1), (\lambda_{42}^1, \lambda_{21}^2)\}$.

3.1.2 Intersection Manager Model. In this article, we assume that every intersection is equipped with an intersection manager to schedule the crossing of vehicles. The manager for intersection $\mathcal{I}_i$ contains the following components:

- A buffer $\mathcal{I}_i.inbox[]$ to store the messages from vehicles within the range of intersection $\mathcal{I}_i$, including *Request* and *Cancel* messages (more details about the messages are described later).
- A buffer $\mathcal{I}_i.confirmed[]$ to store the vehicles to which the $Confirm$ message had been sent from the intersection manager.
- A scheduler to process the messages sent from vehicles and decides their order of crossing.
- A conflict table that stores the feasible paths of intersection $\mathcal{I}_i$ and whether the paths conflict with each other. An example of the conflict table for intersection $\mathcal{I}_2$ in Figure 4 is shown in Table 1.
- Additional sensors (e.g., cameras, loop detectors) that can help the intersection detect whether the vehicles have entered or left the intersection.

3.1.3 Vehicle Model. The vehicles in this work are assumed to be autonomous. A vehicle consists of the following properties:

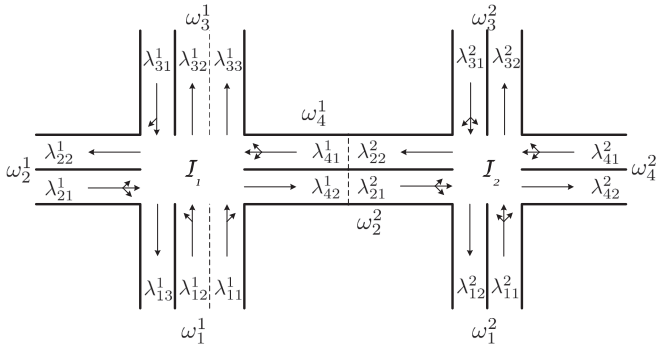


Fig. 4. An example intersection network.

Table 1. Conflict Table for $\mathcal{I}_2$

Paths	$\pi_{\lambda_{11}^2,\lambda_{22}^2}$	$\pi_{\lambda_{11}^2,\lambda_{32}^2}$	…
$\pi_{\lambda_{11}^2,\lambda_{22}^2}$	N	N	…
$\pi_{\lambda_{11}^2,\lambda_{32}^2}$	N	N	…
$\pi_{\lambda_{11}^2,\lambda_{42}^2}$	N	N	…
$\pi_{\lambda_{21}^2,\lambda_{32}^2}$	Y	Y	…
$\pi_{\lambda_{21}^2,\lambda_{42}^2}$	Y	Y	…
$\pi_{\lambda_{21}^2,\lambda_{12}^2}$	N	N	…
…	…	…	…

- Vehicle dynamics information, including acceleration, deceleration, length, and maximum speed, among others.
- Vehicle moving models, including the car-following model and lane-changing model, among others. The details of these models are beyond the scope of this article and do not affect the correctness of our protocol. In our simulation-based performance study, we directly use the models provided by SUMO.
- Transportation information, including
  - —*Departure lane*: The lane into which the vehicle will enter the transportation network (the term might be a bit counter-intuitive—we directly borrow it from SUMO, same as the following two terms).
  - —*Departure position*: The position of the lane to which the vehicle will enter the network.
  - —*Departure speed*: The initial speed the vehicle has when entering the network.
  - —*Route*: The route for a vehicle is defined as an ordered sequence of feasible paths (i.e., $V_i.route = \{\pi_1, \pi_2, \pi_3, \ldots\}$). A *valid* route is defined as follows.

Definition 3.1. A route is valid if for all adjacent pairs of feasible paths in the route (i.e., $\forall \pi_i^a, \pi_{i+1}^b$ of intersections $\mathcal{I}_a$ and $\mathcal{I}_b$, respectively), the pair $(\pi_i^a.second, \pi_{i+1}^b.first)$ is in the defined connection between $\mathcal{I}_a$ and $\mathcal{I}_b$ (i.e., $(\pi_i^a.second, \pi_{i+1}^b.first) \in C_{\mathcal{I}_a,\mathcal{I}_b}$).[3]

3.1.4 Messages. We define three types of messages between the intersection manager and the vehicles, similarly as in our previous work [33]:

- *Request*: A request message is sent by a vehicle to acquire permission for entering the intersection. It contains *requestID*, *roundID*, *sender*, *sending time*, *current road*, *destination road*, *isFront,* and *estimated arriving time* (t_{exp}). In particular, *isFront* denotes whether the vehicle is the front vehicle in its lane (i.e., no other vehicle between it and the intersection). This information is assumed to be detected by sensors. The *estimated arriving time* is used by the intersection manager to schedule the time window for each vehicle to enter the intersection and can be calculated based on vehicle location, speed, and acceleration information (all collected from sensors). The *roundID* is used to distinguish different rounds of crossing, as the vehicle may cross the intersection several times and each round should use a unique ID.
- *Confirm*: A confirm message is sent by the intersection manager to give permission to a vehicle for entering the intersection. It contains *confirmID*, *roundID*, *sending time*, and

[3]It should be noted that this definition can be relaxed if the lane-changing function exists.

arriving time window ($[T_L, T_H]$). If the vehicle enters the intersection during the arriving time window, it is guaranteed to be safe according to our protocol. A vehicle cannot enter the intersection if no *Confirm* is received. If the vehicle cannot enter the intersection within the time window, it must not enter the intersection. Instead, the vehicle can send a *Cancel* message as discussed later.

- *Cancel*: A cancel message can be sent by a vehicle to notify the intersection manager that a previous *Confirm* is "canceled" and the vehicle will not enter the intersection. The *Cancel* message can be used for improving performance and is optional. Once receiving the *Cancel* message, the intersection manager can schedule other vehicles immediately and does not have to wait for the vehicle to cross the intersection. Without receiving the *Cancel* message, the intersection manager will wait for the corresponding timing window $[T_L, T_H]$ to expire, before scheduling another vehicle (note that the intersection manager knows whether the vehicle has entered the intersection through sensors). The fields in the *Cancel* message include *cancelID*, *corresponding confirmID*, and *sending time*.

3.2 Single-Lane Delay-Tolerant Protocol

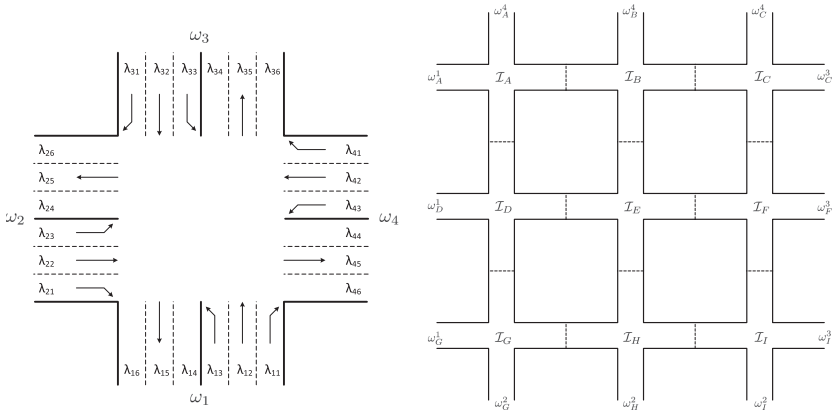
The protocol for a single-lane intersection was introduced in our previous work [33]. In that protocol, we define three types of timeouts: (1) timeout for message transmission (i.e., a message becomes invalid after the defined amount of time), (2) timeout for resending (i.e., the amount of time a vehicle has to wait before resending *Request* if *Confirm* is not received), and (3) timeout for wait (i.e., the amount of time the intersection manager has to wait for a currently scheduled vehicle before scheduling other vehicles). The protocol makes the following assumptions:

- There is an intersection manager at the intersection. It receives *Requests* and *Cancels* from vehicles within its communication range, decides the order for those vehicles to cross the intersection, and then sends corresponding *Confirms*.
- All vehicles and the intersection manager are connected through a vehicular network.
- Each message has a living period and becomes invalid after that. This is the same as message timeout and is removed in the general multi-lane protocol described later.
- A *Confirm* message should correspond to the latest *Request* message that a vehicle has sent. If the vehicle has later sent a newer *Request*, it will not take the *Confirm* corresponding to the old *Request*. This is removed in the general multi-lane protocol.
- All vehicles are equipped with basic safety functions, such as collision avoidance, as the bottom line for safety concerns.

The intersection manager contains a buffer $\mathcal{I}_i.inbox[]$ to store the *Request* and *Cancel* from the vehicles within the range of intersection $\mathcal{I}_i$. As our protocol is mostly based on first come first served (FCFS) scheduling, the intersection manager considers the *Request* messages sent from the front vehicles from different directions (at most one front vehicle from each direction in a single-lane intersection) and chooses the one with the earliest estimated arriving time t_{exp}. If its route does not conflict with the currently confirmed vehicles, the intersection manager will send *Confirm* to that vehicle. Once the *Comfirm* is sent, the intersection will be reserved for the vehicle until it enters the intersection or will wait for a timeout before sending *Confirm* to other vehicles.

In Zheng et al. [33], the vehicle model and the intersection manager model are captured in state machines. Then the liveness and safety properties of the intersection are verified by converting those state machines to timed automata and using the UPPAAL verification tool [2]. Please refer to Zheng et al. [33] for more details of the single-lane delay-tolerant protocol.

Next, we will discuss the challenges in addressing multiple lanes from each direction and then introduce our multi-lane delay-tolerant protocol.



(a) Single intersection (b) Multiple intersections

Fig. 5. Examples of multi-lane intersections.

3.3 Challenges in Addressing Multiple Lanes

To help explain the challenges and solutions in multi-lane intersections, we assume that an intersection has dedicated lanes for left turns, right turns, and going straight, as shown in Figure 5(a). We use the same assumption in our simulations for measuring system performance, but our protocol guarantees safety, liveness, and deadlock-free properties for the more general model defined in Section 3.1. We also assume that a vehicle goes to the dedicated lane corresponding to its path automatically. For a traffic network with multiple intersections, as shown in Figure 5(b), the connection of adjacent intersections is to connect the corresponding lanes together—that is, connecting left lane with left lane, middle lane with middle lane, and right lane with right lane.

3.3.1 Direct Extension of the Single-Lane Protocol. In the single-lane protocol, only the front vehicles are allowed to send *Requests* and receive *Confirms*. Furthermore, the intersection manager can send only one *Confirm* at a time, which may lead to significant loss of performance for vehicles that have lined up and can enter the intersection together. Such restrictions are especially problematic when we consider multi-lane intersections with more vehicles from each direction. To address this and enable sending confirmations to multiple vehicles, a direct extension of the single-lane protocol to multiple-lane intersections can be as follows:

- *Vehicles*: A vehicle sends a *Request* once it enters the communication range of the intersection.
- *Intersection manager*:
  - —Different from the single-lane protocol, the intersection manager now periodically checks traffic conditions and in each controlling period decides *a set of vehicles* with non-conflicting routes that can be sent *Confirms* simultaneously.
  - —The intersection manager may simultaneously confirm multiple vehicles aligned in a queue from the same lane and send consecutive time windows ($[T_L, T_H]$) to those vehicles according to their locations. For example, if three vehicles (V_1, V_2 and V_3) are aligned in a queue, and V_1 is the front vehicle with estimated arriving time $t_{exp} = 10.0$, the time window for V_1, V_2, and V_3 can be $[10.0, 13.0)$, $[13.0, 16.0)$, and $[16.0, 19.0)$, respectively.

However, when considering communication delays or losses, the direct extension may lead to inefficient scenarios and possible deadlocks, as shown in Sections 3.3.2 and 3.3.3, respectively.

3.3.2 Efficiency Problems in Direct Extension. In earlier direct extension, when simultaneously sending *Confirms* to multiple vehicles in a queue (from the same lane), the intersection manager needs to assign consecutive time windows to those vehicles based on their locations. To do this, one idea is to first ask each vehicle to include its location in the *Request* message. Then when sending *Confirms*, the intersection manager can try to estimate the current location of each vehicle based on its location at the request time and the time tag (i.e., sending time) of the *Request* message. However, an accurate estimation of vehicle physical locations is difficult, given that the intersection manager does not know the detailed vehicle dynamics. Even the estimation of relative locations (i.e., ordering) of vehicles could be difficult since the vehicles may not send the *Requests* at the same time. Even when the vehicles do send the *Requests* simultaneously, some of those request messages could be significantly delayed or lost, in which case the intersection manager may miss sending *Confirms* to some of the vehicles in the queue. For instance, assume that vehicles A, B, and C are in a queue (A is ahead of B, and B is ahead of C), and the *Requests* from A and C are received by the intersection manager, whereas the *Request* from B is significantly delayed or lost. Then the intersection manager only sends *Confirms* to A and C, with consecutive time windows. A may be able to enter the intersection, whereas C cannot move because B is still waiting ahead of it. In this case, the system has to wait until C's time window expires and reschedule vehicles—an inefficient scenario with significant performance loss.

Another idea is for the intersection manager to leverage the estimated arriving time in *Requests* to determine vehicle locations (or at least their ordering) and assign consecutive time windows. However, except for the front vehicle in a lane, it is often difficult for a vehicle to estimate its arriving time, as its arrival at the intersection depends on when the vehicles in front of it can pass through the intersection. Furthermore, even when vehicles can somehow accurately estimate their arriving time, their *Request* messages may still be significantly delayed or lost, in which case the intersection manager may miss sending *Confirms* to some of the vehicles in a queue and cause performance loss, similarly as explained earlier with the example.

In fact, there will be a loss of performance whenever the intersection manager confirms the vehicles that are not at the front of a queue while not confirming the front vehicle. If this keeps happening (e.g., due to design bugs in the manager), the system may even be deadlocked.

In the direct extension, the assumption that each message has a living period may also cause problems, particularly for *Request* messages. The original intention to introduce a message living period is to prevent the intersection manager from using old messages that contain outdated information. However, throwing out outdated messages makes it difficult for the intersection manager to determine all vehicles in the queue. For example, in Figure 6, vehicles A, B, and C are waiting before the intersection. We assume that the message living period is 1.5. At time 0, vehicle B sends *Request* R2 to the intersection manager. The manager may be scheduling other vehicles from other directions and therefore put R2 in the incoming inbox. At time 1, vehicle A and vehicle C send *Requests* R1 and R3 to the intersection manager, respectively. At time 2, vehicles from other directions have passed through the intersection, and the manager decides to schedule the queue starting from A. However, at this moment, R2 had already become invalid (since the message living period is 1.5) and been thrown out. The intersection manager therefore only sends *Confirms* to A and C. Although A will pass the intersection now, C cannot move because B is still waiting. In this case, the system has to wait until C's window expires and re-schedule, and thus cause performance loss. In some cases, the message living period may also lead to deadlocks, as explained in Section 3.3.3.

3.3.3 Possible Deadlocks in Direct Extension. In the direct extension, a controlling period is defined for the intersection manager to collect *Requests* during each period and send *Confirms* to a set of vehicles with non-conflicting routes. When considering such a period together with the

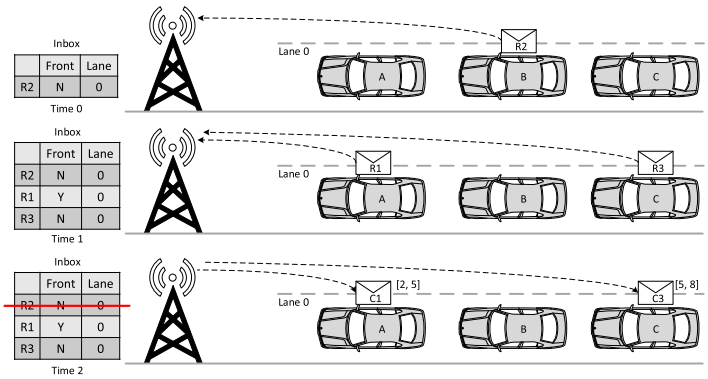


Fig. 6. Problem of the living period of messages.

IM period	1	V period	1	Msg delay	0.1

Time	Vehicle (V)	Manager(IM)	
k	Send Req1	Send Con0	Inbox[]
k + 0.1	Receive Con0 invalid	Receive Req1	Inbox[Req1]
k + 1	Send Req2	Send Con1	Inbox[]
k + 1.1	Receive Con1 invalid	Receive Req2	Inbox[Req2]

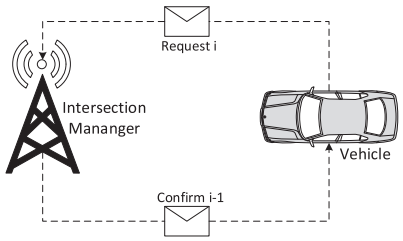


Fig. 7. Deadlock example: sending a new *Request* message before receiving a valid *Confirm* message.

resending timeout of vehicles, as well as the assumption that a *Confirm* message should only be taken by a vehicle if it corresponds to the latest *Request* from that vehicle, the direct extension may cause deadlocks, as shown later.

The first scenario is shown in Figure 7. In this example, there is only one vehicle. The controlling period of the intersection manager is 1, and the resending timeout for the vehicle is also 1. The message delay is 0.1. We assume that the vehicle had already sent a *Request* Req0 to the intersection manager before time $k-1$, but the manager was scheduling other vehicles. At time k, the manager sends a *Confirm* Con0 to the vehicle. However, due to message delay, the vehicle does not know that a *Confirm* is sent, while the resending timeout has been reached. So the vehicle sends another *Request* Req1 at time k. At time $k+0.1$, Req1 arrives at the intersection manager and is stored in the inbox. Meanwhile, the *Confirm* message Con0 arrives at the vehicle. However, it does not correspond to the latest *Request* message Req1 and thus is treated as invalid. Then at time $k+1$, the vehicle sends another *Request* Req2 to the intersection manager, while the intersection manager checks the inbox and sends another *Confirm* Con1 to the vehicle. This process will keep repeating, and the system is deadlocked. More generally, such deadlock occurs when a vehicle sends a new *Request* before receiving a valid *Confirm* from the last *Request*, which could happen in the following instances:

- The round trip communication time is greater than the resending timeout of the vehicle, or
- The time window length in the *Confirm* $[T_L, T_H]$ is smaller than the message delay *msgDelay* (i.e., $T_H - T_L < msgDelay$).

Another type of deadlock occurs due to the usage of the message living period. For instance, vehicle A could already be in the front of a queue and sends a *Request*. The intersection manager might be scheduling vehicles from other directions and put the *Request* in the inbox. Then when

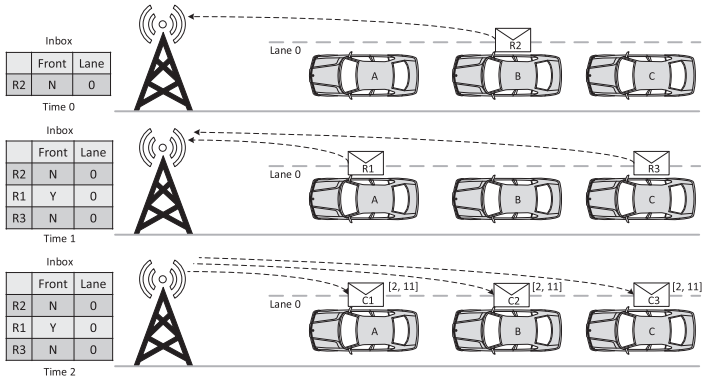


Fig. 8. Modified protocol that removes the message living period and sends a single time window to all confirmed vehicles in a queue from the same lane.

the intersection manager is ready to schedule vehicles from A's lane, that *Request* might have already expired. If this keeps happening, the system will get into a deadlock situation.

3.4 Multi-Lane Delay-Tolerant Protocol

In the design of our multi-lane delay-tolerant protocol, we first address the preceding efficiency and deadlock challenges from the direct extension.

3.4.1 Solution to Efficiency Problems. The solution lies in a few aspects. First, there is in fact no need to figure out the ordering of vehicles in a queue. Instead, the vehicles can themselves maintain the order to enter the intersection, as they are assumed to have autonomous driving capabilities (with features like adaptive cruise control and collision avoidance). Therefore, instead of assigning multiple consecutive time windows to vehicles in a queue, the intersection manager assigns a single time window that is estimated as long enough for all confirmed vehicles in a queue to enter the intersection one by one. If later vehicles cannot enter the intersection when the single time windows expires, they will stop and resend *Requests*.

Second, as shown in Sections 3.3.2 and 3.3.3, the message living period (i.e., timeout) should not be used for *Request* messages. In fact, we remove the usage of timeout for all messages, as it is not essential for *Confirms* and *Cancels* to begin with, especially with the usage of time windows in *Confirms*. To avoid using outdated information, the intersection manager only needs to update the *Request* from the same vehicle—that is, it stores every *Request* in its inbox until the corresponding *Confirm* is sent. A *Request* will not expire due to timeout and can only be overwritten by a new one from the same vehicle.

We use the example in Figure 6 to illustrate the modified protocol, as shown in Figure 8. In this case, all *Requests* are stored in the inbox, including R2 from vehicle B at time 0, R1 from vehicle A at time 1, and R3 from vehicle C at time 1. None of these messages will expire due to timeout. At time 2, based on all three *Requests*, the intersection manager assigns a single time window of 9 time units (i.e., [2, 11]) to all three vehicles. The vehicles will then proceed to enter the intersections in order.

Finally, it is required that non-front vehicles can only be confirmed together with the front vehicle in the same lane, to prevent the performance loss or even deadlock as discussed in Section 3.3.2.

3.4.2 Solution to Deadlock Problems. In Figure 7, even though the time window in the *Confirm* is still valid, it is not taken by the vehicle because the *Confirm* does not correspond to the latest

Request. This requirement, however, is not necessary and is the main reason for the deadlock. In the modified protocol, we allow a vehicle to take a *Confirm* message as long as its estimated arriving time is within the time window in the *Confirm*. Such modification will avoid the deadlock in Figure 7, although there is a side effect where one more possible *Request* could be left in the inbox.

More specifically, using the same example in Figure 7, we assume that the vehicle had already sent a *Request* Req0 to the intersection manager before time $k-1$, while the vehicle was scheduling other vehicles. At time k, the manager sends a *Confirm* Con0 to the vehicle, and we assume that the time window in Con0 is set as $[k, k+1]$. At time $k+0.1$, the *Confirm* Con0 received by the vehicle now is within the time window $[k, k+1]$ and therefore is valid. However, at time k, the vehicle had sent another *Request* Req1, and this *Request* is received by the intersection manager at time $k+0.1$. As the vehicle had already received a *Confirm*, *Request* Req1 becomes a redundant message. We will show that this one more *Request* is harmless in any case. If the *Request* comes within the time window of Con0, the intersection manager knows that it had already sent a *Confirm* to the vehicle and thus it will simply delete the new *Request*. If the *Request* comes beyond the time window, the intersection manager cannot tell whether the vehicle comes to the intersection again or is the redundant message caused by delay. In this case, the intersection manager will send a corresponding *Confirm* to the vehicle again. If the vehicle had already left the intersection, the vehicle will not take this latest *Confirm*. If the vehicle is in fact coming to the intersection again, it will not take the latest *Confirm* as well (the round ID is different).

3.4.3 Multi-Lane Protocol. Based on the preceding solutions to the efficiency and deadlock challenges, we propose our multi-lane delay-tolerant protocol. We first summarize the changes we made to the original single-lane protocol:

- *Vehicles*:
  - —A vehicle sends *Request* once it enters the communication range of the intersection.
  - —Once a vehicle receives a *Confirm* message, it can proceed as long as its estimated arriving time is within the time window in *Confirm*. The *Confirm* does not have to correspond to the latest *Request* the vehicle sent, but the *roundID* should be the same.
- *Intersection manager*:
  - —Following a controlling period, the manager periodically checks traffic conditions and decides a set of vehicles with non-conflicting routes to send *Confirms*.
  - —The manager stores every *Request* in its inbox until a corresponding *Confirm* is sent. A new *Request* from the same vehicle overwrites the old one.
  - —Non-front vehicles are only confirmed with the front vehicle in the same lane.
  - —The manager confirms multiple vehicles aligned in a queue by sending a single time window ($[T_L, T_H]$) to them. Note that not all vehicles in that lane have to be confirmed, but the front vehicle has to be.
- *Timeouts*:
  - —Instead of the three timeouts in the single-lane protocol, we now only set a resending timeout for vehicles, denoted as t^r_{out}. A vehicle will resend *Request* if *Confirm* is not received after t^r_{out}. We remove the message timeout (i.e., living period). We also remove the timeout for the intersection manager to wait for the currently scheduled vehicle. Instead, T_H in the time window denotes the longest time the intersection manager will wait.

Now we describe the details of our multi-lane delay-tolerant protocol.

Vehicle protocol. The pseudocode of the vehicle protocol is shown in Algorithm 2. Similar to the case in our single-lane protocol [33], vehicle behavior is captured by a state machine. The initial

state of a vehicle is *ApproachingNotConfirmed*. The behavior of the state is defined in the run() procedure—that is, the vehicle keeps its speed and follows front vehicles if there is any (lines 3 and 4).

The procedure next() is used to determine the next state according to the situations at the current timestep. In this procedure, the estimated arriving time to the intersection t_{exp} is calculated based on the vehicle location, speed, and acceleration (line 6). The vehicle will send *Request*[4] with a period equal to the *resendingTimeout* until it receives *Confirm* (lines 7 and 8). Once it receives *Confirm*, it will move to the *ApproachingConfirmed* state and check the time window in that state (lines 9 and 10). However, if no *Confirm* is received and the distance to the intersection (or the last vehicle waiting at the intersection) is less than a safe distance ($disToIntersection \leq safeValue$), the vehicle is required to decelerate and stop before the intersection waiting line, which is the behavior of the state *DecelerationNotConfirmed* (lines 12 and 13). Otherwise, the vehicle stays in the same state *ApproachingNotConfirmed* (lines 14 and 15).

For the state *DecelerationNotConfirmed*, its behavior is to decelerate and guarantee full stop before the intersection if there is no confirmation, which is defined in its run() procedure (lines 17 and 18). Its next() procedure is similar to that of the state *ApproachingNotConfirmed*, where the vehicle will periodically send *Request* until it receives *Confirm* (lines 20 through 26). In this state, if the vehicle has already arrived at the intersection, the field t_{exp} will be replaced with the actual arrival time (line 22).

In the *ApproachingConfirmed* state, the vehicle has received a *Confirm* from the intersection manager with a time window $[T_L, T_H]$ to enter the intersection. The behavior of this state is to keep its speed and follow possible front vehicles (line 29). If the vehicle cannot arrive at the intersection within the time window or the destination lane is full (line 32), it needs to stop before the waiting line. If its distance is less than the safe value, it should enter the *DecelerationNotConfirmed* state, which guarantees that the vehicle can stop before the waiting line or the last waiting vehicle (lines 34 and 35). If the vehicle is still far from the intersection, it should enter *ApproachingNotConfirmed* (lines 36 and 37). For both cases, the vehicle could send *Cancel* to improve efficiency (line 33).

Finally, if the vehicle arrives at the intersection within the time window, it will switch to the state *EnteringIntersection*. In the *EnteringIntersection* state, the vehicle will turn to the destination road (lines 42 and 43). Once it arrives in the range of the next intersection, it re-enters the *ApproachingNotConfirmed* state (lines 48 and 49).

Intersection manager protocol. The pseudocode for the intersection manager is shown in Algorithm 3. Just as we described in Section 3.1.2, the intersection manager maintains a buffer inbox[] for storing the *Requests* and a buffer confirmed[] for storing the vehicles currently being sent *Confirms*. Lines 1 through 7 show how the intersection manager stores the received *Requests* in the inbox[] and maintain the inbox[] buffer. Once a *Request* is received, if the vehicle that sends the *Request* is already in the confirmed[] buffer, the intersection manager will simply delete the message (lines 2 and 3), as we have sent *Confirm* to the vehicle that might be already on the way. If the vehicle that sends the *Request* has sent a *Request* before that is stored in the inbox[], the intersection will update the *Request* with the latest estimated arriving time (lines 4 and 5). For other situations, the *Request* will be stored in the inbox (lines 6 and 7). Lines 8 through 14 show how the confirmed[] buffer is maintained. For each vehicle in the confirmed[] inbox, the intersection manager will first check whether it has entered the intersection (line 9). If it has not entered the intersection while a *Cancel* is received or the time window expires, the intersection manager will

[4]For all fields inside the *Request*, please refer to Section 3.1.4.

ALGORITHM 2: Vehicle

```
Input: currentTime
Switch State do
    case ApproachingNotConfirmed do
        Procedure run()
            keep speed, follow front vehicles if any;
        Procedure next(currentTimeStep)
            calculate t_exp;
            if currentTime is at resendingTimeout and within the communication range then
                send Request with t_exp;
            if receive Confirm then
                return ApproachingConfirmed;
            else
                if disToIntersection ≤ safeValue then
                    return DecelerationNotConfirmed;
                else
                    return ApproachingNotConfirmed;
    case DecelerationNotConfirmed do
        Procedure run()
            decelerate, follow front vehicles and stops before the waiting line or the last waiting vehicle;
        Procedure next(currentTime)
            calculate t_exp;
            if currentTime is at resending timeout and within the communication range then
                send Request with t_exp or actual arrive time if arrived;
            if receive Confirm then
                return ApproachingConfirmed;
            else
                return ApproachingNotConfirmed;
    case ApproachingConfirmed do
        Procedure run()
            keep speed and follow front vehicles if any;
        Procedure next(currentTime)
            calculate t_exp;
            if currentTime ≥ T_H or t_exp ≤ T_L or destination lane full then
                send Cancel;
                if disToIntersection ≤ safeValue then
                    return DecelerationNotConfirmed;
                else
                    return ApproachingNotConfirmed;
            else if arriveTime within [T_L, T_H] then
                return EnteringIntersection;
            else
                return ApproachingConfirmed;
    case EnteringIntersection
        Procedure run()
            turn to destination lane;
```

```
45
46
47      Procedure next()
48          if enters the range of next manager then
49              return ApproachingNotConfirmed;
50          else
51              return EnteringIntersection;
```

directly remove the corresponding vehicle from the confirmed[] buffer (lines 10 and 11). If it has entered the intersection, the intersection manager will keep the vehicle in the confirmed[] buffer until it leaves the intersection (line 12). Therefore, if we revisit lines 2 and 3, the new *Request* will be stored if the corresponding vehicle is removed from the confirmed[] buffer.

Lines 15 through 25 illustrate the procedure in each controlling period. If the inbox[] is not empty (lines 16 and 17), the intersection manager will first check the *Requests* from the front vehicles. Since our policy is mostly based on FCFS, the *Request* with the earliest estimated arriving time (it becomes the actual arrive time if the vehicle has arrived at the intersection) is the candidate for scheduling. The reason for considering the front vehicle with the earliest arrival time is to guarantee liveness, which is proved in Section 4.2. The vehicle sending this *Request* is denoted as V_c as in line 18. Then the intersection manager checks the routes between V_c and the vehicles in the confirmed[] buffer. If conflict is detected, the intersection manager will skip this round of scheduling (lines 19 and 20). If no conflict is detected, the intersection manager will send *Confirm* to V_c and the vehicles aligned behind it with the same destination lane as V_c[5] (line 22). This is done by calling the subroutine confirmQueueStartWith(). Meanwhile, the intersection manager will check for another front vehicle that does not have conflicting routes with the vehicles in the confirmed[] buffer. If found, the intersection manager will also send *Confirm* to this vehicle and all vehicles lined up with the same destination lane (lines 23 through 25).

The subroutine for sending *Confirms* to a front vehicle and all vehicles behind it with the same destination lane is shown in Algorithm 4, named confirmQueueStartWith(). The input to this subroutine is a front vehicle V_f. The buffer vehiclesToConfirm[] is used to store all vehicles that will receive *Confirms* during this round of scheduling. In line 1, the front vehicle V_f is appended to the vehiclesToConfirm[] buffer. From line 2 to line 3, the algorithm searches all vehicles behind V_f and appends the vehicle with the same destination lane into vehiclesToConfirm[]. As described in Section 3.4.3, the time window is for all vehicles in the buffer vehiclesToConfirm[] and is sent to every vehicle. Lines 6 through Line 10 are used for calculating the time window $[T_L, T_H]$ for all vehicles in vehiclesToConfirm[]. The lower bound T_L is set as the current time, and therefore the intersection is reserved for the vehicle.[6] The upper bound T_H depends on the situation. If the estimated arriving time of V_f is less than currentTime, the front vehicle may have already arrived. In that case, the upper bound should start with the current time, plus the bound on maximum communication delay and the traveling time for all vehicles in the set vehiclesToConfirm[]. As shown in line 8, the

[5]As described in Section 3.3, we assume that the intersection has dedicated lanes for left turns, right turns, and going straight. Therefore, the vehicles behind a front vehicle will all have the same destination lane as the front vehicle. Our protocol will also work without this assumption but with a loss of performance.

[6]It is possible that after a vehicle is confirmed the intersection manager receives another request with an earlier estimated arrival time (such a request probably got delayed by a bad communication condition). To mitigate (but not fully prevent) such a scenario, the intersection manager can put constraints such as only confirming a vehicle's request if its arrival time is within a bound of the current time (which was in fact implemented in our simulator).

ALGORITHM 3: Intersection Manager

```
Input: currentTime
if receive Request then
    if Request.sender in confirmed[] then
        continue;
    else if Request.sender in inbox[] then
        update Request;
    else
        Store Request
for all vehicle in confirmed[] do
    if vehicle not enter intersection then
        if receive Cancel or [T_L, T_H] expires then
            delete corresponding vehicle in confirmed[];
    else
        if vehicle leaves intersection then
            delete corresponding vehicle in confirmed[];
if currentTime is at manager period then
    if Inbox[].size() == 0 then
        return;
    V_c = front vehicle with earliest arriving time in Inbox[];
    if V_c conflicts with vehicles in confirmed[] then
        return;
    else
        confirmQueueStartWith(V_c);
        for all Request in Inbox[] do
            if Request.sender not conflict with confirmed[] and Request.sender is front then
                confirmQueueStartWith(Request.sender);
```

ALGORITHM 4: confirmQueueStartWith(V_f)

```
Input: V_f: a front vehicle
vehiclesToConfirm[].append(V_f);
for all Request in Inbox[] do
    if Request.sender.currentLane == V_f.currentLane then
        if Request.sender.destinationLane == V_f.destinationLane then
            vehiclesToConfirm[].append(Request.sender);
T_L = currentTime;
if currentTime ≥ V_f.estArrTime then
    T_H = currentTime + msgDelay^MAX + vehiclesToConfirm[].size() * timeGap;
else
    T_H = V_f.estArrTime + msgDelay^MAX + vehiclesToConfirm[].size() * timeGap;
for all vehicle in vehiclesToConfirm[] do
    send Confirm to vehicle with window [T_L, T_H];
    confirmed[].append(vehicle);
    delete corresponding Request in inbox[];
```

bound that we set for the maximum communication delay is denoted as $msgDelay^{MAX}$, and the traveling time for all vehicles in vehiclesToConfirm[] is calculated as vehiclesToConfirm.size() * timeGap, where timeGap is the estimated time for one vehicle to cross the intersection.[7] If the estimated arriving time of V_f is greater than the current time, the vehicle has not arrived yet. In that case, the upper bound T_H should start with its estimated arriving time $V_f.estArrTime$, plus $msgDelay^{MAX}$ and the traveling time for all vehicles in the set vehiclesToConfirm[], as shown in line 10. Finally, for every vehicle in vehiclesToConfirm[], the intersection manager sends *Confirm* to it with time window $[T_L, T_H]$. Meanwhile, the corresponding *Request* is removed from the inbox[], as shown in lines 11 through 14.

4 PROVING DEADLOCK-FREE, LIVENESS, AND SAFETY PROPERTIES

We verified the deadlock-free, liveness, and safety properties for the single-lane protocol in Zheng et al. [33] by leveraging the UPPAAL model checking tool [2]. We could apply a similar approach to our new multi-lane protocol for intersections that have only a single lane from each direction (i.e., for the single-lane special case of the multi-lane protocol). For the general multi-lane case, however, we have developed new methods to prove these properties, as explained in the following.

4.1 Deadlock Free

Deadlocks are classified into two categories: resource deadlocks and communication deadlocks [26]. Resource deadlocks can happen in a system where threads share resources. Communication deadlocks can happen if a thread A is waiting for a message from another thread B while the other thread B is waiting for a message from thread A [7, 26].

We first consider resource deadlocks. In our problem, the intersection and the entrances to the intersection can be viewed as resources, whereas the vehicles can be viewed as threads. The intersection cannot be shared by vehicles with conflicting routes, and the entrance to the intersection can only be accessed by front vehicles. A deadlock on a resource can occur if and only if all of the following four conditions are met simultaneously, known as the Coffman conditions [11]:

- *Mutual exclusion*: The resources involved must be unshareable. This is true for our case, as the resource (intersection) cannot be shared by vehicles with conflicting routes.
- *No preemption*: The resource can only be released by the thread holding it. This is true for our case, as the resource (intersection) cannot be released for other vehicles if the confirmed vehicles are still inside the intersection.
- *Hold and wait*: A thread is currently holding at least one resource and requesting other resources being held by other processes. This could happen in our case, as one non-front vehicle might hold one resource (e.g., the intersection for time window $[T_L, T_H]$) and requesting for another resource (e.g., the entrance that is currently occupied by another (front) vehicle).
- *Circular wait*: A thread is waiting for a resource being held by another thread, whereas the other thread in turn is waiting for the resource being held by the current thread.

The first three conditions could be met simultaneously. However, as shown in the following, we prove that the fourth condition, circular wait, will never happen in our protocol.

Lemma 4.1. *There is no circular wait between a front vehicle V_f and any vehicle V_w behind it.*

Proof. To prove that there is no circular wait, we only have to prove that the front vehicle V_f does not depend on vehicle V_w. According to the policy in Section 3.4.3, which is also shown in

[7] Accurate estimation of the traveling time can lead to better performance.

lines 22 through 25 in Algorithm 3, the intersection manager will only send *Confirm* to a non-front vehicle V_w if its front vehicle V_f is sent a *Confirm* together. There can be two situations: (1) $V_f.route$ conflicts with $V_w.route$, in which case the *Confirm* will only be sent to V_f and thus V_f does not depend on V_w, and (2) $V_f.route$ does not conflict with $V_w.route$, in which case even if V_w gets the *Confirm* earlier, it cannot block V_f as the intersection is shareable for non-conflicting vehicles. Therefore, V_f will not wait for V_w in both cases, and thus there is no circular wait between V_f and V_w. □

Lemma 4.2. *There is no circular wait between any two front vehicles V_f^i and V_f^j.*

Proof. We prove this by contradiction and assume that the lemma is false (i.e., V_f^i and V_f^j wait for each other). V_f^i waiting for V_f^j indicates that V_f^j is in the confirmed[] buffer and $V_f^i.route$ conflicts with $V_f^j.route$. Meanwhile, V_f^j waiting for V_f^i indicates that V_f^i is in the confirmed[] buffer and $V_f^i.route$ conflicts with $V_f^j.route$. Thus, V_f^i and V_f^j are both in the confirmed[] buffer with conflicting routes. However, this is contradictory to the policy that only vehicles with non-conflicting routes can be put into the confirmed[] buffer (shown in lines 19 and 20 of Algorithm 3). Therefore, there is no circular wait between any two front vehicles V_f^i and V_f^j. □

Theorem 4.3. *There is no circular wait between any two vehicles V_i and V_j.*

Proof. We prove this based on two different situations:

- If V_i and V_j are on the same lane:
  - —If one of the vehicles is a front vehicle, according to Lemma 4.1, there is no circular wait.
  - —If both are non-front vehicles and both are confirmed, they have to be confirmed with the front vehicle. Thus, they have the same routes and there is no circular wait.
- If V_i and V_j are on different lanes:
  - —We prove this by contradiction. Assume that V_i and V_j wait for each other. V_i waiting for V_j indicates that either (1) the front vehicle in V_i's lane, denoted as V_f^i, waits for V_j, or (2) V_i directly waits for V_j. In both cases, V_j is in the confirmed[] buffer and $V_j.route$ conflicts with $V_i.route$. Similarly, V_j waiting for V_i indicates that either (1) the front vehicle in V_j's lane, denoted as V_f^j, waits for V_i, or (2) V_j directly waits for V_i. In both cases, V_i is in the confirmed[] buffer and $V_j.route$ conflicts with $V_i.route$. Any one of those cases is contradictory to the policy that only vehicles with non-conflicting routes can be put into the confirmed[] buffer (lines 19 and 20 in Algorithm 3). Therefore, there is no circular wait for vehicles on different lanes either. □

According to Theorem 4.3, there is no circular wait, and therefore there is no resource deadlock. The following theorem shows that there is no communication deadlock, as long as there is a bound $msgDelay^{MAX}$ on the maximum communication delay and such bound is known. We assume that $msgDelay^{MAX}$ has taken into account possible message losses and resends (i.e., a message could be lost and re-sent, but it will eventually reach its receiver within $msgDelay^{MAX}$).

Theorem 4.4. *There is no circular wait between the intersection manager and any vehicle V_i during communication if the communication delay is bounded by a known bound $msgDelay^{MAX}$.*

Proof. In our protocol, once vehicle V_i sends a *Request* to the intersection manager, it will wait until it receives a *Confirm*. First, as the communication delay is bounded, a *Request* from the vehicle will eventually reach the intersection manager (taking into account of message losses and resends), and a *Confirm* from the manager will eventually reach the vehicle. Furthermore, the

intersection manager does not need a reply from the vehicle indicating whether it receives the *Confirm*. Instead, no matter whether the message is delivered or not, the intersection manager will release the intersection after the time window $[T_L, T_H]$ expires. This means that even a blocked intersection manager will automatically release after the time window expires. Therefore, there is no circular wait in communication. □

4.2 Liveness

The liveness property specifies that any vehicle that sends *Requests* will eventually cross the intersection as long as the communication delay is bounded and the intersection manager will try to schedule vehicles from any lane eventually (i.e., no starving in scheduling policy).

Theorem 4.5. *Any vehicle V_i that sends* Requests *will eventually cross the intersection if the communication delay is bounded by an known bound $msgDelay^{MAX}$ and the intersection manager will try to schedule vehicles from any lane eventually.*

Proof. First, assume that V_i is a front vehicle. A *Request* from V_i will eventually reach the intersection manager (taking into account message losses and resends). The intersection manager will eventually try to schedule vehicle V_i (based on the order of arrival time), and send a *Confirm* with the time window $T_H - T_L \geq msgDelay^{MAX}$ and T_L as the current time. This time window will be valid when V_i receives the *Confirm* and then V_i can enter the intersection.

Since a front vehicle can enter the intersection eventually, if a vehicle is not a front vehicle, it will eventually become one and enter the intersection as well. □

4.3 Safety

Safety is naturally guaranteed in our protocol. According to Algorithm 3, all vehicles in the confirmed[] buffer will not have conflicting routes. Therefore, no vehicles with conflicting routes may enter the intersection at the same time.

5 SIMULATION RESULTS

We use simulation results to demonstrate the effectiveness of our multi-lane delay-tolerant protocol in improving traffic efficiency. The simulations are conducted with (1) the widely used traffic simulation suite SUMO [1] to control the vehicle movement through TraCI APIs, (2) our extension of SUMO to model messages and communication delays, and (3) implementation of our protocol in the extended simulator as well as the smart traffic light system for comparison.

5.1 Simulator Implementation and Experiment Setup

In extending SUMO, we define message class, channel class, intersection manager class, and vehicle class. The message class defines three types of messages and their fields. The channel class defines communication interfaces for sending and receiving messages, and can also add delay to messages based on certain distribution. The intersection manager class defines its behavior according to Algorithm 3, and the sending and receiving of messages are done by calling the communication interfaces provided by the channel class. The vehicle class models the vehicle behavior according to Algorithm 2. The vehicle physical movement is controlled by calling TraCI APIs provided by SUMO. At each timestep, information such as current speed, acceleration, and location can be collected through TraCI APIs, and the vehicle class can use such information to adjust vehicle speed and acceleration. The extended SUMO simulator is shown in the upper right of Figure 1.

In our simulations, we mainly consider a network of interconnected intersections. The model for each intersection is shown in Figure 5(a). It has four ways (ω_1, ω_2, ω_3, and ω_4), and each direction consists of three lanes for left turns, going straight, and right turns. The feasible

paths for the system are as follows: $\pi_1 = (\lambda_{11}, \lambda_{46})$, $\pi_2 = (\lambda_{12}, \lambda_{35})$, $\pi_3 = (\lambda_{13}, \lambda_{24})$, $\pi_4 = (\lambda_{21}, \lambda_{16})$, $\pi_5 = (\lambda_{22}, \lambda_{45})$, $\pi_6 = (\lambda_{23}, \lambda_{34})$, $\pi_7 = (\lambda_{31}, \lambda_{26})$, $\pi_8 = (\lambda_{32}, \lambda_{15})$, $\pi_9 = (\lambda_{33}, \lambda_{44})$, $\pi_{10} = (\lambda_{41}, \lambda_{36})$, $\pi_{11} = (\lambda_{42}, \lambda_{25})$, $\pi_{12} = (\lambda_{43}, \lambda_{14})$. The length of each lane is 100 m. The network of intersections is shown in Figure 5(b). The network consists of nine intersections: $\mathcal{I} = \mathcal{I}_A, \mathcal{I}_B, \ldots, \mathcal{I}_I$. Each intersection has the same setup as the single intersection shown in Figure 5(a). The connection of adjacent intersections is to connect the corresponding lanes together (i.e., connecting left lane with left lane, middle lane with middle lane, and right lane with right lane). There are 12 entrances for the intersection network: ω_A^1, ω_D^1, ω_G^1, ω_G^2, ω_H^2, ω_I^2, ω_C^3, ω_F^3, ω_I^3, ω_A^4, ω_B^4, ω_C^4.

In simulations, the length of a vehicle is set as 5 m. A vehicle has maximum acceleration of 0.8 m/s^2, deceleration of 4.5 m/s^2, and speed limit of 10 m/s. The routes of the vehicles are randomly generated, with the probability ratio for left turns, going straight, and right turns set as 0.25:0.5:0.25. The arriving time of the vehicles follows Poisson distribution in which the arriving rate represents how many vehicles will arrive per second on average. In our experiment, the arriving rate ranges from 0.1 vehicle/s to 0.5 vehicle/s, and a vehicle can only arrive at one of the entrances of the intersections. We use the ratio of the flow from north-south directions and the flow from west-east directions to represent traffic patterns. For example, traffic pattern "flow 0.5 : 0.1" denotes that the average traffic flow from north-south directions is 0.5 vehicle/s and the average traffic flow from west-east directions is 0.1 vehicle/s at each entrance. The performance of the traffic network is measured by the average traveling time of all vehicles during simulation.

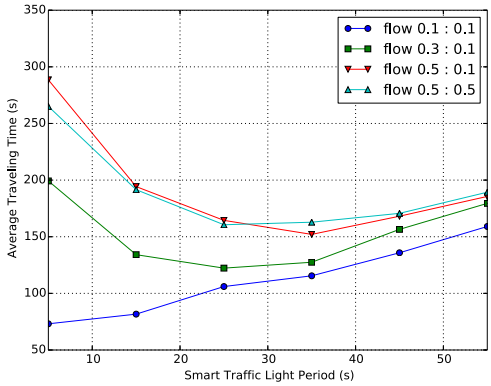
5.2 Comparison between Our Delay-Tolerant Intelligent Intersection Management and Smart Traffic Lights

In this section, we compare our multi-lane delay-tolerant intelligent intersection design with a smart traffic light system that uses the basic back-pressure control, the capacity-aware back-pressure control, and the adaptive max-pressure control, respectively, as described in Section 2.2. We consider the cases of a network of nine interconnected intersections. For comparison, we assume that our protocol works under normal conditions where the communication delay is negligible. For simplicity, we set the communication delay in our protocol as 0 (our protocol can also tolerate bounded delays with degradation of performance, as shown in the next section). We randomly generate traffic traces, and use the same traces in our system and in smart traffic lights with the three different algorithms. The total number of vehicles generated for the network of nine intersections is 1,200. For the capacity-aware back-pressure control, the parameters are set according to Gregoire et al. [16]—that is, m is set to 2.0, C_∞ is set to 200, and C_{λ_i} (capacity of each lane) is set to 15.

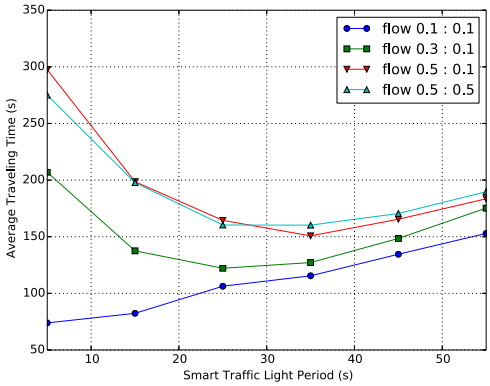
For a smart traffic light system with back-pressure scheduling, we explore its scheduling period from 5 to 55 seconds. The system adjusts its phase each period. We set the minimum period as 5 seconds to leave enough time for one vehicle to cross the intersection. Before switching phases, we also change the green lights to yellow for 3 seconds for safety purposes.

Our intersection manager also periodically processes data with a controlling period, although at a much finer granularity than the scheduling period of smart traffic lights. We explore its controlling period from 0.1 to 2 seconds, which is from the simulation precision to the largest mean of communication delay in this experiment.

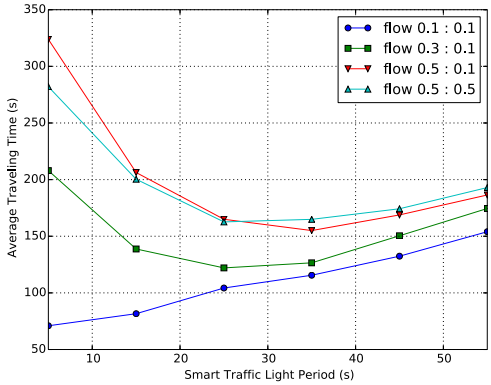
The simulation results for a network of nine interconnect intersections (as in Figure 5(b)) are shown in Figure 9. In these experiments, every intersection uses the same scheduling period in the smart traffic light system and uses the same controlling period in our intelligent intersection system. The performance of the smart traffic lights highly depends on the scheduling period under different traffic patterns, whereas the performance of the intersection manager does not change much with respect to the controlling period.



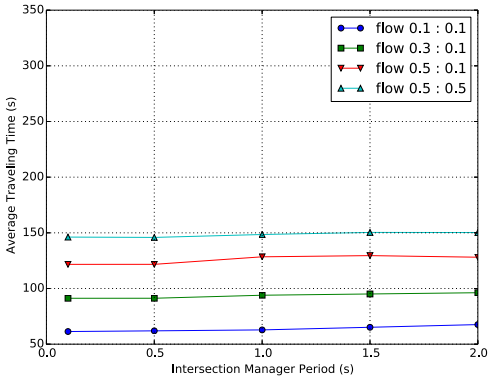
(a) Performance of basic back-pressure control



(b) Performance of capacity-aware back-pressure control



(c) Performance of adaptive max-pressure control



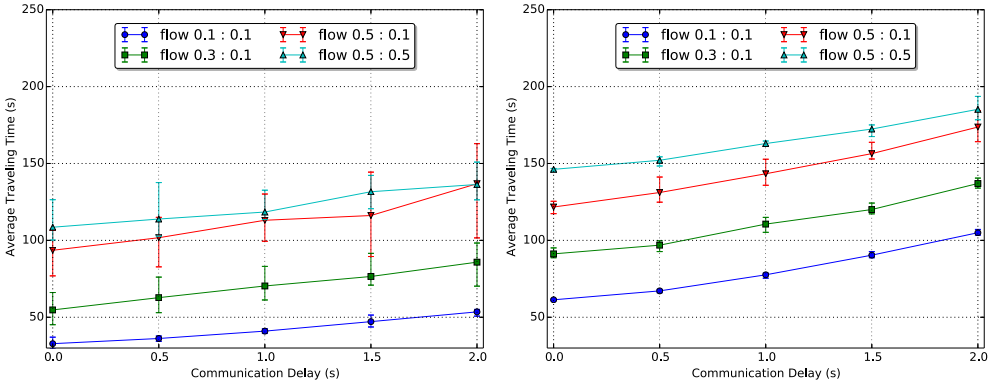
(d) Performance of our intelligent intersection design

Fig. 9. Comparison between smart traffic lights and our intelligent intersection design in the case of nine interconnected intersections with multiple lanes.

Table 2. Best Average Traveling Time for Smart Traffic Lights and for Our Intelligent Intersection Design in the Case of Nine Interconnected Intersections

	Smart Traffic Light			
Traffic Pattern	Basic Back Pressure	Capacity-aware Back Pressure	Adaptive Max Pressure	Our Intelligent Intersection
0.1 : 0.1	73.125	73.927	70.954	61.354
0.3 : 0.1	122.287	122.190	122.054	91.224
0.5 : 0.1	152.068	150.755	155.021	121.688
0.5 : 0.5	160.584	160.235	162.664	145.884

If we assume that the smart traffic lights can always adjust the scheduling period to achieve the best performance, we can compare the performance of our approach with the best performance of the three smart traffic light approaches in Table 2. We can see that our intelligent intersection significantly outperforms the smart traffic lights in all tested traffic patterns.



(a) Single intersection (b) Nine intersections

Fig. 10. Performance of our multi-lane delay-tolerant protocol under varying communication delays.

5.3 Performance of Our Intelligent Intersection Design under Communication Delays

In this section, we further study the performance of our delay-tolerant protocol under varying communication delays. We set the delays to follow Gaussian distribution with the mean ranging from 0 to 2.0 seconds. The relationship between the mean and the deviation of the communication delays can be complicated. According to Yao et al. [31], the deviation can be 1.5 times the mean for the IEEE 802.11p standard. For simplicity, we model the standard deviation equal to the mean.

The same delay distribution for every intersection. In this experiment, the delays at all intersections follow the same Gaussian distribution with the same mean and standard deviation. To deal with the delays, we set the bound on maximum communication delay $msgDelay^{MAX}$ as 4.1 seconds and the resending timeout of a vehicle t^r_{out} as 8.0 seconds. If in simulation a randomly generated delay exceeds $msgDelay^{MAX}$, it will be set as $msgDelay^{MAX}$; if it is smaller than 0, it will be set as 0. In this experiment, 300 vehicles are randomly generated for the single-intersection case, and a total of 1,200 vehicles are generated for the nine-intersection case. For every vehicle, we set its ratio for left turns, going straight, and right turns as 0.25:0.5:0.25.

The simulation results are shown in Figure 10. The x-axis denotes the mean of the communication delays, and the y-axis denotes the performance defined as the average traveling time. There is no deadlock or collision observed in all simulations, as we have already proved in Section 4.

In Figure 10(a) and (b), the average traveling time increases (i.e., the performance decreases) as the delay increases. For the same level of communication delay, the denser traffic (larger flow) will result in worse performance in general. For example, under the same communication delay, the dense traffic pattern "flow 0.5 : 0.5" has longer average traveling time than sparse traffic "flow 0.1 : 0.1." The bar on each point in the figure is the error bar, denoting the deviation of the average traveling time for all simulation samples. The deviation is very small for sparse traffic, and it increases when the traffic becomes dense and unbalanced.

Finally, we can see that the performance does not decrease much when the communication delay is within 0.5 second, which is the case in normal conditions (even with very dense traffic). This shows that our intelligent intersection design should outperform smart traffic lights in normal conditions (especially for a network of intersections). If the communication delay exceeds 0.5 second, it is very likely that the network is under malicious attacks. We conducted some initial studies on such security vulnerabilities, and the results are shown in the following.

Only one intersection (among the nine intersections) has communication delay. In this experiment, we use the nine-intersection model to study which intersections may have the most significant

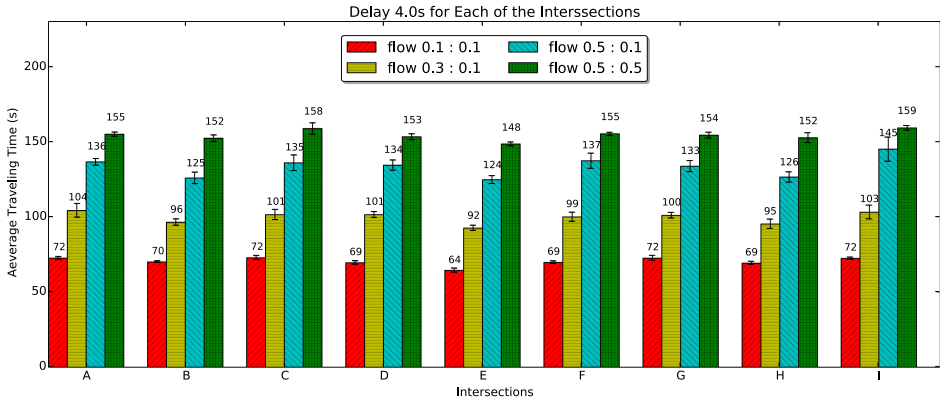


Fig. 11. Performance of a nine-intersection system when one intersection is delayed (being attacked).

impact on system performance when its communication delay is large (possibly due to malicious attacks). During each simulation, we set a communication delay with a mean of 4.0 seconds and a standard deviation of 1.0 second for only one of the intersections (the other intersections are assumed to have no communication delay). The delay bound $msgDelay^{MAX}$ is set as 5.1 seconds, and the resending timeout t^r_{out} is set as 8 seconds. The simulation results are shown in Figure 11. The x-axis denotes the nine intersections, as labeled in Figure 5(b). The y-axis denotes the average traveling time as performance. The bars with different colors denote different traffic patterns. We can see that the significance of certain intersections can be different under different traffic patterns. For sparse traffic, there is not much difference no matter which intersection is delayed (being attacked). However, for denser traffic, delaying some intersections can cause a more significant increase in average traveling time.

6 CONCLUSION

We present a delay-tolerant intelligent intersection management protocol for general multi-lane intersections. We prove that the protocol satisfies the deadlock-free, liveness, and safety properties. We implement the protocol in an extension of the SUMO simulator and compare it with smart traffic lights that use back-pressure scheduling. The simulation results demonstrate that our intelligent intersection design significantly outperforms the smart traffic lights for a network of intersections.

REFERENCES

[1] DLR. 2018. SUMO. Retrieved October 7, 2019 from http://www.dlr.de/ts/en/desktopdefault.aspx/tabid-9883/16931_read-41000/.

[2] UPPAAL. 2018. Home Page. Retrieved October 7, 2019 from http://uppaal.org/.

[3] Mourad Ahmane, Abdeljalil Abbas-Turki, Florent Perronnet, Jia Wu, Abdellah El Moudni, Jocelyn Buisson, and Renan Zeo. 2013. Modeling and controlling an isolated urban intersection based on cooperative vehicles. *Transportation Research Part C: Emerging Technologies* 28 (2013), 44–62.

[4] Reza Azimi, Gaurav Bhatia, Raj Rajkumar, and Priyantha Mudalige. 2012. *Intersection Management Using Vehicular Networks*. Technical Report.

[5] Reza Azimi, Gaurav Bhatia, Ragunathan Raj Rajkumar, and Priyantha Mudalige. 2014. STIP: Spatio-temporal intersection protocols for autonomous vehicles. In *Proceedings of the ACM/IEEE 5th International Conference on Cyber-Physical Systems (with CPS Week 2014) (ICCPS'14)*. IEEE, Los Alamitos, CA, 1–12. DOI: https://doi.org/10.1109/ICCPS.2014.6843706

[6] Seyed (Reza) Azimi, Gaurav Bhatia, Ragunathan (Raj) Rajkumar, and Priyantha Mudalige. 2013. Reliable intersection protocols using vehicular networks. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems (ICCPS'13)*. ACM, New York, NY, 1–10. DOI: https://doi.org/10.1145/2502524.2502526

[7] Valmir C. Barbosa. 1990. Strategies for the prevention of communication deadlocks in distributed parallel programs. *IEEE Transactions on Software Engineering* 16, 11 (1990), 1311–1316.

[8] Y. Ozan Basciftci, Fangzhou Chen, Joshua Weston, Ron Burton, and C. Emre Koksal. 2015. How vulnerable is vehicular communication to physical layer jamming attacks? In *Proceedings of the 2015 IEEE 82nd Vehicular Technology Conference (VTC'15-Fall)*. 1–5. DOI: https://doi.org/10.1109/VTCFall.2015.7390968

[9] Saeed Bastani, Bjorn Landfeldt, and Lavy Libman. 2011. On the reliability of safety message broadcast in urban vehicular ad hoc networks. In *Proceedings of the 14th ACM International Conference on Modeling, Analysis, and Simulation of Wireless and Mobile Systems (MSWiM'11)*. ACM, New York, NY, 307–316. DOI: https://doi.org/10.1145/2068897.2068951

[10] Lei Chen and Cristofer Englund. 2016. Cooperative intersection management: A survey. *IEEE Transactions on Intelligent Transportation Systems* 17, 2 (Feb. 2016), 570–586. DOI: https://doi.org/10.1109/TITS.2015.2471812

[11] Edward G. Coffman, Melanie Elphick, and Arie Shoshani. 1971. System deadlocks. *ACM Computing Surveys* 3, 2 (June 1971), 67–78. DOI: https://doi.org/10.1145/356586.356588

[12] Kurt Dresner and Peter Stone. 2008. A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research* 31 (2008), 591–656.

[13] Yaser P. Fallah and Masoumeh K. Khandani. 2015. Analysis of the coupling of communication network and safety application in cooperative collision warning systems. In *Proceedings of the ACM/IEEE 6th International Conference on Cyber-Physical Systems (ICCPS'15)*. ACM, New York, NY, 228–237. DOI: https://doi.org/10.1145/2735960.2735975

[14] Nathan H. Gartner, John D. C. Little, and Henry Gabbay. 1975. Optimization of traffic signal settings by mixed-integer linear programming: Part I: The network coordination problem. *Transportation Science* 9, 4 (1975), 321–343.

[15] Jean Gregoire, Emilio Frazzoli, Arnaud De La Fortelle, and Tichakorn Wongpiromsarn. 2014. Back-pressure traffic signal control with unknown routing rates. *IFAC Proceedings Volumes* 47, 3 (2014), 11332–11337.

[16] Jean Gregoire, Xiangjun Qian, Emilio Frazzoli, Arnaud De La Fortelle, and Tichakorn Wongpiromsarn. 2015. Capacity-aware backpressure traffic signal control. *IEEE Transactions on Control of Network Systems* 2, 2 (June 2015), 164–173. DOI: https://doi.org/10.1109/TCNS.2014.2378871

[17] Matthew Hausknecht, Tsz-Chiu Au, and Peter Stone. 2011. Autonomous intersection management: Multi-intersection optimization. In *Proceedings of the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 4581–4586. DOI: https://doi.org/10.1109/IROS.2011.6094668

[18] P. B. Hunt, D. I. Robertson, R. D. Bretherton, and M. Cr. Royle. 1982. The SCOOT on-line traffic signal optimisation technique. *Traffic Engineering & Control* 23, 4 (1982), 190–192.

[19] Qiu Jin, Guoyuan Wu, Kanok Boriboonsomsin, and Matthew Barth. 2012. Advanced intersection management for connected vehicles using a multi-agent systems approach. In *Proceedings of the 2012 IEEE Intelligent Vehicles Symposium*. 932–937. DOI: https://doi.org/10.1109/IVS.2012.6232287

[20] H. Kowshik, D. Caveney, and P. R. Kumar. 2011. Provable systemwide safety in intelligent intersections. *IEEE Transactions on Vehicular Technology* 60, 3 (March 2011), 804–818. DOI: https://doi.org/10.1109/TVT.2011.2107584

[21] Jennie Lioris, Alex Kurzhanskiy, and Pravin Varaiya. 2016. Adaptive max pressure control of network of signalized intersections. *IFAC-PapersOnLine* 49, 22 (2016), 19–24.

[22] Pitu Mirchandani and Larry Head. 2001. A real-time traffic signal control system: Architecture, algorithms, and analysis. *Transportation Research Part C: Emerging Technologies* 9, 6 (2001), 415–432.

[23] NHTSA National Highway Traffic Safety Administration. 2018. Fatality Analysis Reporting System (FARS). Retrieved October 7, 2019 from https://www.nhtsa.gov/research-data/fatality-analysis-reporting-system-fars.

[24] R. Naumann, R. Rasche, Jürgen Tacken, and C. Tahedi. 1997. Validation and simulation of a decentralized intersection collision avoidance algorithm. In *Proceedings of the Conference on Intelligent Transportation Systems*. 818–823. DOI: https://doi.org/10.1109/ITSC.1997.660579

[25] Arthur G. Sims and Kenneth W. Dobinson. 1980. The Sydney Coordinated Adaptive Traffic (SCAT) system philosophy and benefits. *IEEE Transactions on Vehicular Technology* 29, 2 (May 1980), 130–137. DOI: https://doi.org/10.1109/T-VT.1980.23833

[26] Mukesh Singhal. 1989. Deadlock detection in distributed systems. *Computer* 22, 11 (1989), 37–48.

[27] Pravin Varaiya. 2009. A universal feedback control policy for arbitrary networks of signalized intersections. EECS, UC Berkeley. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.308.5717.

[28] Pravin Varaiya. 2013. Max pressure control of a network of signalized intersections. *Transportation Research Part C: Emerging Technologies* 36 (2013), 177–195.

[29] Pravin Varaiya. 2013. The max-pressure controller for arbitrary networks of signalized intersections. In *Advances in Dynamic Network Modeling in Complex Transportation Systems*. Springer, 27–66.

[30] Tichakorn Wongpiromsarn, Tawit Uthaicharoenpong, Yu Wang, Emilio Frazzoli, and Danwei Wang. 2012. Distributed traffic signal control for maximum network throughput. In *Proceedings of the 2012 15th International IEEE Conference on Intelligent Transportation Systems*. 588–595. DOI: https://doi.org/10.1109/ITSC.2012.6338817

[31] Yuan Yao, Lei Rao, Xue Liu, and Xingshe Zhou. 2013. Delay analysis and study of IEEE 802.11p based DSRC safety communication in a highway environment. In *Proceedings of the 2013 Proceedings IEEE INFOCOM*. 1591–1599. DOI: https://doi.org/10.1109/INFCOM.2013.6566955

[32] Ali A. Zaidi, Balázs Kulcsár, and Henk Wymeersch. 2016. Back-pressure traffic signal control with fixed and adaptive routing for urban vehicular networks. *IEEE Transactions on Intelligent Transportation Systems* 17, 8 (Aug. 2016), 2134–2143. DOI: https://doi.org/10.1109/TITS.2016.2521424

[33] Bowen Zheng, Chung-Wei Lin, Hengyi Liang, Shinichi Shiraishi, Wenchao Li, and Qi Zhu. 2017. Delay-aware design, analysis and verification of intelligent intersection management. In *Proceedings of the 2017 IEEE International Conference on Smart Computing (SMARTCOMP'17)*. 1–8. DOI: https://doi.org/10.1109/SMARTCOMP.2017.7946999

[34] Bowen Zheng, Chung-Wei Lin, Huafeng Yu, Hengyi Liang, and Qi Zhu. 2016. CONVINCE: A cross-layer modeling, exploration and validation framework for next-generation connected vehicles. In *Proceedings of the 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'16)*. 1–8. DOI: https://doi.org/10.1145/2966986.2980078

[35] Feng Zhu and Satish V. Ukkusuri. 2015. A linear programming formulation for autonomous intersection control within a dynamic traffic assignment and connected vehicle environment. *Transportation Research Part C: Emerging Technologies* 55 (2015), 363–378.

Received December 2017; revised October 2018; accepted December 2018