STTR: A System for Tracking All Vehicles All the Time At the Edge of the Network

Zhuangdi Xu
Georgia Institute of Technology
Atlanta, GA
xzdandy@gatech.edu

Harshit Gupta
Georgia Institute of Technology
Atlanta, GA
harshitg@gatech.edu

Umakishore Ramachandran
Georgia Institute of Technology
Atlanta, GA
rama@gatech.edu

ABSTRACT

To fully exploit the capabilities of sensors in real life, especially cameras, smart camera surveillance requires the cooperation from both domain experts in computer vision and systems. Existing alert-based smart surveillance is only capable of tracking a limited number of suspicious objects, while in most real-life applications, we often do not know the perpetrator ahead of time for tracking their activities in advance. In this work, we propose a radically different approach to smart surveillance for vehicle tracking. Specifically, we explore a smart camera surveillance system aimed at tracking *all* vehicles in *real time*. The insight is *not* to store the raw videos, but to store the space-time trajectories of the vehicles. Since vehicle tracking is a continuous and geo-distributed task, we assume a geo-distributed Fog computing infrastructure as the execution platform for our system. To bound the storage space for storing the trajectories on each Fog node (serving the computational needs of a camera), we focus on the *activities* of vehicles in the vicinity of a given camera in a specific geographic region instead of the time dimension, and the fact that every vehicle has a "finite" lifetime. To bound the computational and network communication requirements for detection, re-identification, and inter-node communication, we propose novel techniques, namely, *forward* and *backward* propagation that reduces the latency for the operations and the communication overhead. STTR is a system for smart surveillance that we have built embodying these ideas. For evaluation, we develop a toolkit upon SUMO to emulate camera detections from traffic flow and adopt MaxiNet to emulate the fog computing infrastructure on Microsoft Azure.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; **Sensor networks**; *Real-time systems*; • **Information systems** → *Storage management*; • **Networks** → Network types;

KEYWORDS

multi-target multi-camera tracking, smart camera surveillance, fog computing, trajectory management

ACM Reference Format:
Zhuangdi Xu, Harshit Gupta, and Umakishore Ramachandran. 2018. STTR: A System for Tracking All Vehicles All the Time At the Edge of the Network. In *DEBS '18: The 12th ACM International Conference on Distributed and Event-based Systems, June 25–29, 2018, Hamilton, New Zealand.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3210284.3210291

1 INTRODUCTION

A smart camera surveillance system [8] has the potential for simultaneously reducing manual labor (which could be error-prone) and achieving better efficiency and effectiveness for surveillance tasks. Specifically, due to increase in urban terrorism there is a crying need to bring more automation to bear on the task of suspicious vehicle tracking. Without such automation help, security personnel have to watch tons of archived videos in order to track a suspicious vehicle, which is extremely demanding and error-prone due to lapses in attention [18]. Existing smart camera surveillance systems, such as IBM S3 [20], are largely alert-based. In other words, users need to register the suspicious vehicles in advance and then once the system detects them, it will start tracking and sending alerts. However, in reality, it is not always the case that we know which vehicle we want to track in advance. For example, after a traffic accident, the police would want to find where the perpetrators escaped, which there is no way to know in advance. Alert-based smart surveillance cannot help a lot in these situations. The state-of-the-art is to store raw video streams from cameras and if a situation warrants it, to analyze the recorded videos to extract the track of a suspicious vehicle postmortem.

In this work, we propose a radically different approach to smart surveillance for vehicle tracking. Specifically, we explore a smart camera surveillance system aimed at tracking *all* vehicles in *real time*. The insight is *not* to store the raw videos, but store the space-time trajectories of the vehicles. Queries could then be answered directly from such recorded space-time tracks of all the vehicles. In this work, we do not focus on the querying side of the problem but on demonstrating that it is feasible to construct a system that would store the trajectories of all vehicles for their lifetime.

We adopt Fog/edge computing [3] as the platform for smart camera-based vehicle tracking. Processing sensor streams (especially cameras) at the edge of the network is advantageous for three reasons: (a) reducing the latency for processing the streams, (b) reducing the backhaul bandwidth needed to send raw sensor streams to the Cloud; and (c) preserving privacy concerns for the collected sensor data. Specifically, for our work in building a smart surveillance system for 24x7 vehicle tracking, Fog computing offers the following advantages:

- For a large-scale smart camera surveillance system, sending thousands of camera streams to the Cloud is not practical and a waste of network resources.
- Vehicle tracking is a continuous and local task, and geo-distributed Fog infrastructure is an ideal fit for such a task.
- Storing the vehicle trajectories at the edge of the network would enable fast query processing.

The biggest challenge for our approach is the limited storage on each fog node, since the vision in this work is to store the space-time track of all vehicles for their *lifetime*. Considering the infinite time dimension, it appears very hard (even impossible) to construct such a system, since cameras are continuously generating new vehicle detections. Based on this impression, the solution usually lies in discarding old trajectories or pushing the heavy lifting to the Cloud (under the assumption of unlimited resources in the Cloud) and reducing the role of the Fog to simply serve as a cache for recent trajectories. In our work, we look at the problem of trajectory generation differently. Instead of the time dimension, we focus on *activity* of a vehicle in a geographical region and formulate an upper bound for the storage space needed per vehicle for recording its activity in its entire lifetime. While the time dimension is infinite, the number of simultaneous activities (number of vehicles) of a finite geographical region is finite and the activity (lifetime) of *given vehicle* is also finite, which implies the storage required for a given finite geographical region to store *all* its activities is also finite. The assumption is that after the useful life of a vehicle (i.e., it is no longer being used in the roadways either because it has been totaled and/or has been sent to a junkyard), the tracks will no longer be hosted on the Fog nodes. Most likely it will be archived in the Cloud for legal reasons should such information be needed at some future date for law enforcement purposes. Considering the tag/title registrations for our vehicles every year, the task of tracking all vehicles is like an advanced vehicle registration system, and instead of on a yearly basis, the granularity is finer (say every second). And each camera (and its associated Fog node) is like a tag office, which is responsible of recording all activities occurring in its region.

The second challenge for large-scale camera-based tracking is the size of computation, where the system should make its best effort to detect and re-identify the vehicles as they are moving from cameras to cameras in real time. In other words, we want to reduce the size of computation or search search space such that we can minimize the latency of processing each vehicle detection. The smart camera system that we propose in this paper exploits the locality of vehicle tracking problem, and restricts the space to nearby cameras that vehicles must pass through. Specifically we propose two strategies to bound the computation that each camera has to do: *forward propagation* which progressively multicasts the signature of a detected vehicle to the downstream cameras in the forward path of the vehicle (using the topology of camera deployment in a given neighborhood) so that vehicle *re-identification* can be immediately triggered at a camera when a vehicle is sighted; and *backward propagation* to reach back to the upstream cameras upon detecting a new vehicle that was not informed by the forward propagation.

Based on the above two strategies, we propose our smart camera surveillance system dubbed, "STTR" (short for *Space Time Trajectory Registration*). STTR is aimed at solving the system side challenge, i.e., distributed computation, communication, and storage, for real-time vehicle tracking using camera networks. We rely on domain expertise from computer vision for multi-camera tracking algorithms including vehicle detection and re-identification. Also, in this work, we only focus on demonstrating the generation and storage of the space-time tracks of vehicles in real time. Efficient indexing structure of the space-time tracks for fast query processing is outside the scope of this work and will be explored in the future.

We summarize the contribution of this work as follows:

- We present details of the activity-based tracking of all the vehicles all the time in a given geographical area that is at the intellectual core of STTR, which enables bounding the storage space requirements at each Fog node.
- We present the details of *forward* and *backward* propagation that enable bounding the computation and communication requirements of STTR.
- We implement STTR using ZeroMQ for inter-camera communication and Redis persistent key-value store for recording the space-time trajectories.
- We build a toolkit on top of SUMO [13], with which we are able to import OpenStreetMap [21], generate traffic flows and detectors, simulate vehicle movements, and the corresponding camera streams.
- We evaluate STTR with the above tool kit and MaxiNet [24] on Microsoft Azure to experimentally verify the theoretical assertions about finite storage space requirements for activity-based space-time tracking of vehicles.

The rest of the paper is organized as follows: Section 2 covers related work, Section 3 presents the problem definition and notations, Section 4 summarizes the theoretical upper bounds of computation and storage for the problem, Section 5 presents the details of the forward and backward strategies, Section 6 gives the architecture of STTR, Section 7 gives the implementation details of STTR, Section 8 presents the experimental setup for emulating the Fog computing infrastructure and the camera streams on Microsoft Azure using MaxiNet, Section 9 summarizes the evaluation of STTR, and Section 10 concludes with directions for future work.

2 RELATED WORK

Multi-target multi-camera tracking (MTMC).
Wu, et al. [26] present a new evaluation measure to isolate the track-based multi-camera tracking (T-MCT) errors from single camera tracking (SCT) errors. Gou, et al. [6] introduce DukeMTMC4ReID, a new large-scale real-world person re-identification dataset, which uses 8 disjoint surveillance camera views covering parts of the Duke University campus. Similarly, VeRi [16] is a dataset for vehicle re-identification in urban surveillance scenario, which contains over 40,000 bounding boxes of 619 vehicles captured by 20 cameras. These datasets are more specialized for computer vision algorithms, while in our work, traffic flow and the road network are more influential. That's why we adopt SUMO [13] for simulation and evaluation. Wang [22] discusses topics related to intelligent multi-camera video surveillance such as multi-camera calibration, and computing

the topology of camera networks. We believe that our work can be integrated with these ideas as long as they are local feature or detector based. Particularly we leave the detection and matching modules to domain experts, to allow integration of such algorithms into our system. Kawanishi, et al. [12] study failures caused by appearance change or environmental illumination if matching only happens across adjacent cameras and introduce “random camera drop" and “trajectory ensemble" to integrate incomplete trajectory result. This work is a very good complement for extending our system when detections are missed due to occlusion (e.g., a truck hiding a vehicle from the camera’s field of view).

Smart video surveillance.
Arth’s [2] work optimizes the object re-acquisition and tracking when there are limited resources. Alsmirat [1] utilizes the cloud and mobile edge computing (MEC) to optimize the network bandwidth for wireless surveillance system. Fan [5] presents an event-driven visualization mechanism fusing multi-modal information for a large-scale intelligent video surveillance system. Mobile fog [11] proposes a programming model for developing large-scale distributed situation awareness applications, launching the application components on Fog nodes at the edge of the network. Vehicle tracking by cameras is used as an example application for evaluation in mobile fog, as the intent is to showcase the features of the programming model. Arun [9] proposes the use of smart surveillance to shift from “investigation of incidents" to “prevention of potentially catastrophic incidents". Their system architecture involves object detection, multi-object tracking, object classification and real time alert. Their work is a pioneering effort in smart surveillance. IBM s3 [20] smart surveillance system involves a smart engine for video/image analysis and middleware for large scale surveillance management. Their work focuses more on openness and extensibility of the framework, and cross-indexed data model for correlation across multiple sensors and event types.

Trajectory management.
There exists many research efforts on trajectory compression or simplification [15][4][17]. While our work does not involve any trajectory simplification, these technologies can be very helpful if used with our system to further reduce actual storage usage. Another major topic on trajectory management is efficient query support [14][7][19], which usually balances the trade-off between spatio-temporal range query and retrieval by distance or time interval. At the current stage of our work, our focus is on building the large-scale smart camera surveillance system and storing all the trajectories. Efficient query support (e.g., spatial and temporal indexing into the stored trajectories) is in our future work.

3 PROBLEM DEFINITION AND NOTATIONS

This section defines the problem addressed in this work and all the notations used in the rest of the paper which we summarize in Table 1.

Figure 1[1] shows a pictorial representation of the road network and the computational infrastructure assumed in this work. We assume that vehicles are continuously moving on the road network, and there are no shortcuts hidden from the cameras for the vehicles to “disappear” from camera observation. We expect that most cameras are at road intersections and a few along the road. We assume a non-overlapping camera network in this paper, although there is nothing inherent in the system that we have built that will preclude it being used to a situation where the field of view (FOV) of the cameras overlap. We do not assume that cameras are at each road intersection, and in Sections 5 and 9, we will show how our system reacts to different density of camera distributions. Cameras are connected to nearby fog nodes, so camera streams can be processed there. For simplicity of illustration, we will assume

Table 1: Notation Table

$c_i \backslash f_i$	Camera i and its corresponding fog node. They are used interchangeably.
v_x	Vehicle x.
$v_x \in c_i \| [t_r, t_p]$	Vehicle x was active under camera i during time interval $[t_r, t_p]$.
$v_x \in c_i$	Vehicle x is active under camera i now.
$size(c_i)$	Size of camera i’s activity region.
s_x	The signature of vehicle x, such as license plate number or a vector of features returned by the vehicle detection algorithm.
d_x	The detected object of vehicle x, which usually includes the signature and other helpful information.
$d(v_x, c_i, t)$	The detected object of vehicle x at camera i and time stamp t.
$V(d_x)$, $C(d_x)$, $T(d_x)$	Reverse functions. $V(d_x) = v_x$, $C(d_x) = c_i$ and $T(d_x) = t$.
$reid(d_x, p)$	The vehicle re-identification algorithm. Given d_x and a candidate pool (list) p of vehicles’ detected objects, it returns i such that $V(d_x) = V(p[i])$.
$head(d_x)$	Getting the vehicle’s heading information from d_x.
$forward(c_i)$	The candidate pool maintained for forward propagation on camera i, which is composed of detected objects received from other cameras’ forward action.
$backward(c_i)$	The candidate pool maintained for backward propagation on camera i. Detected objects in $backward(c_i)$ meet the following requirement: $\forall d_x \in backward(c_i), V(d_x) \in c_i$.
$g_k : t_k$	One trajectory record generated from one detected object. In this paper, we use location and time stamp as an example.
$u(g_{k_1} : t_{k_1}, g_{k_2} : t_{k_2}, \ldots) \| c_i$	One trajectory vertex is the unit for vehicles’ trajectory stored on one camera. It can consist of many successive trajectory records.
$u_x \| c_i$	Same as the above. We emit the trajectory records for simplicity.
$u_{x_t} \| c_i \rightarrow u_{x_{t+1}} \| c_j$	The edge connected the trajectory vertices for the same vehicle on two different cameras. The first trajectory record of $u_{x_{t+1}} \| c_j$ should be the successor of the last one of $u_{x_t} \| c_i$.

[1]Icons made by Smashicons, Freepik from www.flaticon.com is licensed by CC 3.0 BY

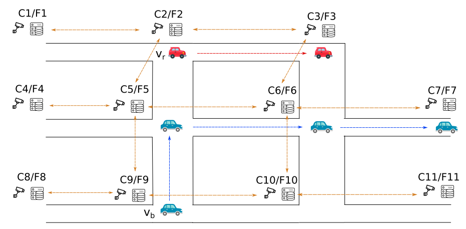

Figure 1: Conceptual Picture: This figure shows an example road network and camera surveillance system, where c_i represent cameras, f_i represent Fog nodes and orange lines represent bidirectional network communication between Fog nodes. Multiple vehicles are moving in this region.

each camera is connected to a unique fog node and refer to c_i and f_i interchangeably. In other words, in this paper we will use the word "camera" generally to include not only the capability of videoing but also computation, storage and network. In reality, it is more common that multiple cameras are connected to one nearby fog node, which can be done by running the application instances in the docker containers. We assume nearby cameras are connected by cables/fibers for network communication such that a low-latency local area network is set up among cameras.

With this setup, the smart camera surveillance system will launch application instances running on each camera, which cooperate with each other to track all passing vehicles in real time and store the trajectories such that the system is able to answer track-related queries immediately such as *where did the red vehicle at c_2 come from?* or *where does the blue vehicle at c_9 head to?* In the rest of the paper, we assume a perfect vehicle detection and re-identification algorithm and our work focuses on designing distributed computation, communication, and storage in this system.

4 ACTIVITY AND STORAGE UPPER BOUND

In this section, we will elaborate how we use physical restrictions to find the upper bound of storage required on each camera to store vehicles' trajectories. To start with, we first define two terminologies, *vehicle activity* and *camera's activity region*.

Definition 4.1. For vehicle x, if we have $\exists d(v_x, c_i, t) \wedge (\forall c_{i'} \nexists t'(t' > t) \wedge d(v_x, c_{i'}, t'))$, then we call vehicle x is active under camera i from time t, shortened as $v_x \in c_i|[t, now]$. Similarly, if we have $\exists d(v_x, c_i, t_r) \wedge (\exists c_j t_p(t_p > t_r) \wedge d(v_x, c_j, t_p)) \wedge (\forall c_{i'} \nexists t_q(t_p > t_q > t_r) \wedge d(v_x, c_{i'}, t_q))$, then we call vehicle x was active under camera i between t_r and t_p, shortened as $v_x \in c_i|[t_r, t_p]$.

In other words, at any given time point, each vehicle is active under the camera that last detected it. As the last camera detecting the vehicle changes with the movement of the vehicle, so we have a chain of vehicle activities. For example, in figure 1, we have $v_r \in c_2 \rightarrow v_r \in c_3$ for the red vehicle and $v_b \in c_9 \rightarrow v_b \in c_5 \rightarrow v_b \in c_6 \rightarrow v_b \in c_7$ for the blue vehicle[2]. Let's consider a specific stage of chain $v_x \in c_i|[t_r, t_p] \rightarrow v_x \in c_j|[t_p, t_q]$. We know vehicle x is at camera i at t_r and at camera j at t_p, but we have no information where v_x is exactly between t_r and t_p. For example, it could be moving on the road, or it could have come to a stop (say in a parking garage, or on the roadside). But we do know it must be at some place that cannot be detected by any other cameras during $[t_r, t_p]$. The collection of these places are called camera i's activity region.

[2]Time interval is omitted for simplicity.

Definition 4.2. For camera i, its activity region is defined as:

$$\bigcup_{\forall v_x, c_j \; v_x \in c_i|[t_r, t_p] \rightarrow v_x \in c_j|[t_p, t_q]} location(v_x)$$

For example, in Figure 1, the activity region of c_5 is the 4 roads towards c_2, c_4, c_6 and c_9. In reality, camera's activity region can be more diverse including roads, garage, plaza, estate. But what matters is the size of activity region and whether it is finite. With reference to Figure 1, there is a practical upper bound for the size of the activity regions owned by cameras $c_2, c_3, c_5, c_6, c_9, c_{10}$; while it is unbounded for the cameras $c_1, c_4, c_8, c_7, c_{11}$, since these cameras are at the periphery of a geographical region of interest.

Definition 4.3. For a given geographical region and camera surveillance system, cameras that have a finite activity region are regarded as *interior* cameras, while cameras that have an infinite activity region are regarded as *boundary* cameras, and camera i's activity region is shortened as $size(c_i)$.

In the rest of the paper, we only focus on *interior* cameras unless stated otherwise. For a closed geographical region, almost all cameras are interior cameras, and for a large-scale camera surveillance system, majority of the cameras are interior. In Section 7, we will revisit boundary cameras and discuss options therein for bounding the storage requirement.

The finite size of the camera's activity region gives us a very good property, because at any given time, the number of vehicles that are active under this camera has to be finite, for vehicles need to occupy space. Meanwhile, each vehicle's life is also finite which indicates there exists trajectory upper bound for a given camera. Putting these facts together, we have the following simple idea — **At any time point, each camera stores the trajectory of vehicles that are active under its region.** And we have the storage space required on a camera i at time t is:

$$disksize(c_i, t) = \sum_{\forall v_x, v_x \in c_i|[t,t]} sizeof(traj(v_x))$$

where *traj* is the trajectory of a given vehicle and *sizeof* is the number of bytes to store them. By applying the activity region and the trajectory upper bound, we have the following theorem:

THEOREM 4.4. *For each camera i, its storage space over time $disksize(c_i, t)$ has the following upper bound:*

$$\forall t, disksize(c_i, t) \leq \rho_i size(c_i) \# maximum\ size\ of\ trajectory$$

where $\rho_i size(c_i)$ is the maximum number of vehicles that can be simultaneously active under c_i.

Since this upper bound is conditioned by $\rho_i size(c_i)$, which is governed by the physical environment near the camera and is unlikely to change frequently, this also gives a good reference for system administrators to know the storage capacity needed for each camera.

We show the storage required on each camera is finite, but is it small enough to be practical? We make the discussion concrete with a hypothetical but realistic example. Considering that the average life of a vehicle is 150000 miles[23], with a deployment of 5 cameras every mile, using two 64-bit longitude- latitude to represent each car detection, and assuming 100 (maximum number of simultaneous vehicles) activities for a given camera, the upper bound for storage space needed on each camera can be calculated as $150000 * 5 * 2 * 64 * 100$ which is around 1.12GB[3]. In reality, we will have to store more meta data besides longitude-latitude, but at the same time it is also unlikely that ALL the vehicles in one camera's activity region are close to the end of their active life.

5 PROPAGATING VEHICLE DETECTIONS

In this section, we are looking into the detection propagation models to limit the storage and computational needs for vehicle re-identification. Upon vehicle detection by a camera, the detections have to be propagated for re-identification by neighboring cameras. While domain experts provide detection and re-identification algorithms, our system will handle the underlying detection propagation to reduce the resource requirements. Broadcasting to all the cameras in the region of interest is not only wasteful of networking resources but also burdening ALL the cameras with unnecessary additional computational work (leading to latency for re-identification). Instead, we propose two detection propagation models, detection *forward* and *backward*, which exploit the topology of camera deployment in the roadways.

5.1 Forward Propagation

Upon detection of a vehicle, the camera propagates the detected vehicle's signature to the set of cameras that are likely candidates for this vehicle to pass through next. We call this *forward propagation*. This way the downstream cameras are already primed to run the re-identification procedure when the vehicle is sighted without any additional communication. For example, in Figure 1, when the blue vehicle is detected by c_5, c_6 is the only candidate camera downstream in the direction in which the blue car is headed, so we can safely forward $d(v_b, c_5, t)$ to c_6. For a more complicated case, let us remove c_6 from the camera surveillance system. In this case, since the direction of travel of the vehicle cannot be predicted, we have to forward $d(v_b, c_5, t)$ to a set of cameras (c_3, c_7, c_{10}). Eventually one camera will detect the vehicle again, which is c_7 in this case, and c_7 will send a confirmation to (c_3, c_{10}), so they can safely discard $d(v_b, c_5, t)$. In general, until such confirmation is received from one of the downstream cameras that are in the plausible set for the next sighting of the vehicle, all cameras have to hold on to the objects received by this forward propagation.

Theorem 5.1. *The resource for each camera to store all the detection objects received by forward propagation has the following upper bound:*

$$\forall c_i, sizeof(forward(c_i)) \leq \rho_i size(c_i) * sizeof(d_x)$$

where $forward(c_i)$ is the candidate pool for detection objects received by forward propagation and we assume the format of detected object is fixed, so $sizeof(d_x)$ is constant.

The rationale behind theorem 5.1 has similarity with how we discuss the activity and storage upper bound. When the detected object $d(v_x, c_i, t_r)$ stays in the $forward(c_j)$ at time t_p, it means $v_x \in c_i|[t_r, t_p]$. In other words, v_x must be somewhere between the c_i and c_j, which we call the shared activity region between camera i and j, shortened as $c_i \cap c_j$. By summing up all cameras that have shared activity region with c_j, we have $size(\sum_{\forall c_i \neq c_j} c_i \cap c_j) \leq size(c_j)$. And we also have $\forall d_x \in forward(c_j) \Rightarrow V(d_x) \in \bigcup_{\forall c_i \neq c_j} c_i \cap c_j$. Combining these terms together, if a detection object is in the camera's forward propagation candidate pool, then the corresponding vehicle must occupy the space in the camera's activity region. So the storage resource of the camera's candidate pool is capped by the product of maximum number of vehicles in the camera's activity region and size of each detected object.

5.2 Backward Propagation

Backward Propagation would be warranted if a vehicle is detected by a camera for which no forward detection notification was received from some upstream camera in the direction of travel of the vehicle[4]. In this case, the camera has to guess whence from (i.e., which upstream camera) the vehicle came. Therefore, it sends the detected object "backwards" to a candidate set of upstream cameras once again taking into account the deployment topology for possible re-identification by one of those cameras. For example, in Figure 1, considering removing c_6 from the camera surveillance system, if the blue vehicle is detected by c_7, (without any prior forwarding information), we will backward $d(v_b, c_7, t)$ to a set of cameras (c_3, c_5, c_{10}). One camera should confirm it saw v_b before, which is c_5 in this case. Different from forward propagation, cameras do not need to keep the detected objects received from backward propagation. Instead, if the camera cannot re-identify the detected object in the backward propagation, it simply discards the object. On the other hand, the candidate pool for backward propagation is composed of detected objects that have not been re-identified by any other upstream cameras, which is $\forall d_x \in backward(c_i) \Rightarrow V(d_x) \in c_i$ where $backward(c_i)$ is the backward candidate pool of camera i.

Theorem 5.2. *The resource for each camera to store all the detected objects for backward propagation has the following upper bound:*

$$\forall c_i, sizeof(backward(c_i)) \leq \rho_i size(c_i) * sizeof(d_x)$$

Again the storage space is capped by the product of the maximum number of vehicles in the camera's activity region and size of each detected object, as $backward(c_i)$ is actually the collection of detected objects of vehicles that are active under camera i.

6 STTR SYSTEM ARCHITECTURE

STTR (short for Space Time Trajectories Registration) is the system that embodies the ideas presented thus far for registering the space-time tracks of all the vehicles all the time. Figure 2 shows the STTR

[3]This upper bound is predicated on clearing out the trajectories of "destroyed vehicles" from the fog nodes either by archiving them in the Cloud or deleting them permanently as discussed in Section 7.2.3.

[4]In the current implementation of STTR, failure of forward propagation is caused by the optimizations in section 7.1, while in reality it could also be due to the fact that an upstream camera failed to detect the vehicle.

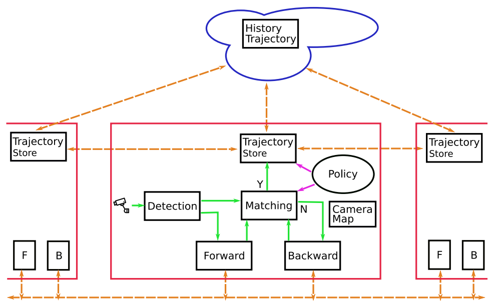

Figure 2: STTR System Architecture. The red rectangle represents the application instances running on each camera. The orange lines represent the fog-fog or fog-cloud network communication. The green lines represent the flow of processing one vehicle's detected object from camera stream to trajectory storage.

system architecture. Each application instance running on the camera contains 7 modules: Detection, Matching, Forward, Backward, TrajectoryStore, CameraMap and Policy. Among them, Detection and Matching provide interfaces for domain experts to plug in unique algorithms. Specifically, we expect the input of Detection will be the raw camera stream (frames) and the output of Detection will be the stream of vehicle's detected objects. Each detected object is a JSON object that at least includes *time stamp*, *unique signature* and *direction of travel* information. Matching implements the re-identification algorithm $reid(d_x, p)$, which inputs a detected object and a candidate pool and outputs the object from the candidate pool. For the remaining 5 modules, CameraMap maintains the geographical relationship between cameras, which is mainly used with the direction of travel to form the set of cameras for forward and backward propagation; Forward/Backward implements the vehicle detection propagation in Section 5, and real-time requirement is also taken into account by dynamically monitoring the traffic flow and truncating/flushing the forward candidate pool; TrajectoryStore distributes the trajectory according to vehicle activities as discussed in Section 4 and optimizes network consumption; Policy module allows each camera to configure parameters and disable specific modules, for example only a subset of cameras are permitted to archive the trajectories at the cloud. There could be services running in the cloud to reach into the space-time trajectories stored in the camera nodes. For example, we show the *HistoryTrajectory* service in the cloud in Figure 2. This service may provide the ability to archive some of the trajectories (e.g., vehicles no longer in active service, trajectories that are older than some delta, etc.) from the camera nodes to the cloud. Similarly, we envision other services, e.g., *QueryEngine* that may embody our future work with supporting query processing of the stored trajectories in the cameras.

The flow of processing each detected objects is presented as green lines in Figure 2. After getting detected objects d_x from Detection, d_x is given to the Forward to multicast it to candidate cameras, and Matching to $reid(d_x, forward(c_i))$. If a re-identification is successfully found, d_x and the previous detected object is given to TrajectoryStore for writing. Otherwise, d_x will be given to the Backward to multicast to backward candidate cameras. Meanwhile, Backward will re-identify the detected objects it received with $backward(c_i)$ through Matching, and similarly if a re-identification is found, TrajectoryStore will be used for recording.

7 IMPLEMENTATION DETAILS

STTR is written in Python, communication between cameras is based on ZeroMQ, and persistent key-value store is based on Redis. In this section, we go into some implementation details.

7.1 Matching

The Matching module brings in the re-identification algorithm provided by domain experts, which accepts a detected object and a candidate pool. In this section, we show some tricks in STTR to help further reducing the latency of re-identification.

7.1.1 Forward candidate pool flush. Consider a vehicle stream $v_1v_2 \dots v_{2n}$ from camera c_a to c_b, where the odd sequence $v_1v_3 \dots v_{2n-1}$ stops in a plaza in the middle and the even sequence $v_2v_4 \dots v_{2n}$ directly heads to c_b. However, by forward propagation, the detected objects of the odd sequence, which is $d_1d_3 \dots d_{2n-1}$, will stay in the $forward(c_b)$. So for vehicle v_{2k} from the even sequence, the re-identification on c_b has to go through first k detected objects in $forward(c_b)$, and this phenomenon will accumulate over time. In other words, long-term stopped vehicle will produce an unwelcome detected object in the top of following cameras' forward candidate pool, and we want to discard it (when the corresponding vehicle moves again later, its re-identification can be found through backward propagation) to reduce the latency for majority moving vehicles.

Algorithm 1 Forward_candidate_pool_flush

```
1: $k \leftarrow reid(d_x, forward(c_i))$
2: **if** $k > \alpha * len(forward(c_i))$ **then**
3:     $forward(c_i) \leftarrow forward(c_i)[k+1 : end]$
4: **else**
5:     $forward(c_i) \leftarrow forward(c_i)[0 : k-1, k+1 : end]$
6: **end if**
```

Algorithm 1 illustrates the idea of flush operation of camera's forward candidate pool. After we get the re-identification result for every detected object, we check the position of the result in the candidate pool, where α is a parameter between 0-1 for tolerance of short-term stopped vehicles and overtaking between vehicles. If we find the result is after first $\alpha * len(forward(c_i))$ objects, we flush the candidate pool by removing all detected objects before the re-identification result. Otherwise, we only remove the result itself.

7.1.2 Adaptive truncated candidate pool. Considering a long-term stopped vehicle moves again, when it gets detected by the following camera, unfortunately its forwarded detected object has been discarded. However, there is no way for Matching module to know this, so it will try to re-identify the vehicle with the whole forward candidate pool, fail and fall back to Backward module. When the re-identification algorithm is efficient and rate of traffic flow is low, we can afford doing this, but this is not often the case. Instead we want to consider the rate of traffic flow, efficiency

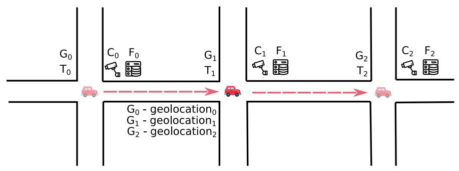

Figure 3: Trajectory aggregation example. The red vehicle starts at intersection G_0 at time T_0, then moves through intersection G_1 at time T_1 and finally ends at intersection G_2 at time T_2. Correspondingly we have camera C_i and fog node F_i at each road intersection.

of re-identification algorithm and then decide whether to give a truncated candidate pool to the Matching module. Intuitively, the slower the algorithm is, the more we need to truncate the candidate pool. Similarly, if the traffic flow is high, we have to spend less time on each detected object, so the candidate pool needs to include few objects. By these observations, Algorithm 2 shows how we monitor the rate of traffic flow, time spent on re-identifying the last detected object, and then update the size of the candidate pool given to the re-identification function.

Algorithm 2 Adaptive_truncated_candidate_pool

```
1: $rate \leftarrow Detection(c_i)$
2: $L \leftarrow \frac{L}{rate*time}$
3: $(k, time) \leftarrow reid(d_x, forward(c_i)[0:L])$
```

7.2 TrajectoryStore

In this subsection, we go into the detail of trajectory implementation in STTR and how to bring the activity and storage upper bound in Section 4 into life and optimize it. We implement the trajectories as vertices and edges linking two vertices on the key-value store. Each trajectory vertex is a list of trajectory records of when and which camera detected the corresponding vehicle. Each edge links trajectory vertices stored on different fog nodes and heads to the vertex with large time stamp. For simplicity, we will use the following notation $u(g_1 : t_1, g_2 : t_2)|c_2 \rightarrow u(g_3 : t_3)|c_3$ to represent one trajectory vertex on camera 2 storing two trajectory records (g_1, t_1) and (g_2, t_2), while another trajectory vertex on camera 3 stores the trajectory record (g_3, t_3) and a directed edge links them together.

7.2.1 *Greedy trajectory aggregation.*

The most intuitive way to implement the idea — **At any time point, each camera stores the trajectory of vehicles that are active under its region** – is to aggregate the trajectories as vehicle moves from one camera to another. Take the red vehicle in Figure 3[5] as an example:

- When the red vehicle is detected by C_0, we generate the trajectory vertex $u(G_0 : T_0)|F_0$.
- When the red vehicle is then detected by C_1, we get and remove the previous trajectory vertex $u(G_0 : T_0)|F_0$ from F_0, and aggregate the new trajectory vertex at F_1, so we have $u(G_0 : T_0, G_1 : T_1)|F_1$.
- Similarly, when the red vehicle is finally detected by C_2, we get and remove the previous trajectory vertex $u(G_0 : T_0, G_1 : T_1)|F_1$ from F_1, and aggregate the new trajectory vertex at F_2, so we have $u(G_0 : T_0, G_1 : T_1, G_2 : T_2)|F_2$.

Algorithm 3 Greedy_trajectory_aggregation

```
1: $d_x \leftarrow Detection(c_i)$
2: $d'_x \leftarrow Matching(d_x, c_i)$
3: $u'_x \leftarrow pullAndRemoveTrajectory(d'_x)$
4: $store(append(u'_x, c_i : T(d_i))|c_i)$
```

Algorithm 3 summarizes the procedure of greedy trajectory aggregation. Upon receiving the new detected object d_x and its re-identification result d'_x, we pull and remove the trajectory vertex u'_x from the camera $C(d'_x)$, which is the previous camera that detected the vehicle. Then we store and append the new trajectory record to u'_x. The greedy trajectory aggregation is a direct translation from the activity upper bound idea, but it suffers from the increasing network consumption over time. Because the trajectories of vehicle will become larger and larger over the life of the vehicle, and it will be not practical to keep pulling the whole trajectories from one camera to another.

7.2.2 *Lazy trajectory aggregation.*

The idea is that each camera only aggregates the trajectory if its storage is under pressure and a new trajectory vertex is coming. We again use the example in Figure 3 and assume no camera's storage is under pressure at the beginning.

- When the red vehicle is detected by C_0, we generate the trajectory vertex $u(G_0 : T_0)|F_0$.
- When the red vehicle is then detected by C_1, we generate the new trajectory vertex and create an edge from the previous trajectory vertex, which is $u(G_0 : T_0)|F_0 \rightarrow u(G_1 : T_1)|F_1$.
- Similarly, when the red vehicle is finally detected by C_2, we will have $u(G_0 : T_0)|F_0 \rightarrow u(G_1 : T_1)|F_1 \rightarrow u(G_2 : T_2)|F_2$.
- After a period time, F_1's storage is under pressure and a new trajectory vertex is coming, so $u(G_1 : T_1)|F_1$ is chosen to be aggregated.
- $u(G_1 : T_1)|F_1$ will be aggregated to F_2, so we have $u(G_0 : T_0)|F_0 \rightarrow u(G_1 : T_1, G_2 : T_2)|F_2$ as the trajectory of the red vehicle.

Algorithm 4 Lazy_trajectory_aggregation

```
1: $d_x \leftarrow Detection(c_i)$
2: $d'_x \leftarrow Matching(d_x, c_i)$
3: $u_y \leftarrow randomPick(c_i)$
4: $aggregate(u_y)$
5: $store(trajectoryVertex(d'_x) -> u(c_i : T(d_x))|c_i)$
```

Algorithm 4 summarizes the procedure of lazy trajectory aggregation. Upon receiving the new detected object d_x and its re-identification result d'_x, if the storage of the camera is under pressure[6], a random victim trajectory vertex with outgoing edge is

[5]Icons made by Smashicons, Freepik from www.flaticon.com is licensed by CC 3.0 BY

[6]Idea of lazy trajectory aggregation also works without setting a threshold for storage pressure, and actually in evaluation, in order to see the lazy trajectory aggregation in

picked u_y. We aggregate u_y to the trajectory vertex which the outgoing edge points to. Then we create an edge from the trajectory vertex of d'_x to the new trajectory vertex generated from d_x. Under lazy trajectory aggregation, each camera also owns the activity upper bound and corresponding storage upper bound. It is because whenever a new trajectory vertex is coming, either the activity region of the camera hasn't been full of vehicles or one old trajectory vertex is available to be aggregated to another camera. In the worst case, for example, all cameras are under pressure, and then lazy trajectory aggregation will be eventually consistent with greedy aggregation as it will keep aggregating the trajectory vertex with outgoing edges such that trajectory data leaving on one camera will only belong to the vehicles that are still active under this camera. Moreover, lazy trajectory aggregation is network friendly for two reasons. Firstly, the rate of the trigger of lazy aggregation is the rate of incoming vehicles detected by the camera which is slow compared to the network speed[7]. Secondly, the minimum aggregation only needs to transmit one trajectory record as there is only one trajectory record written in. Yet, for network efficiency, it is usually better to pack multiple trajectory records into one transmission. On the other hand, the downside of lazy trajectory aggregation is the potential increased in query processing time, e.g., for a range search. Exploring indexing techniques for efficient query processing is part of our future work.

7.2.3 General discussion. Does the infinite time line contradict with the activity upper bound we give for each camera? The answer is no. Let's consider a closed region first. In a closed region, every camera is the interior camera except for those at the sink and the source. Source is where the vehicle is created, such as the vehicle factory, the entrance of the region, while the sink is where the vehicle is "destroyed" (e.g, vehicle is "totaled", or sent to the junkyard), the exit of the region. In other words, when the vehicle is created from the source, we start tracking the vehicle, and when the vehicle is destroyed at the sink, we stop tracking the vehicle. So more precisely, we are able to track all *alive* vehicles in the region over their lifetime with each camera owning limited resources. With time elapsing, there will be new vehicles born from the source and we start to allocate resource for them, old vehicles gone at the sink and we "free" their resources (either by deletion or archiving them permanently in the cloud). Notice we do not need explicit garbage collection, because the trajectory aggregation will eventually aggregate the whole trajectory to the sink, as the sink is the last camera detecting those retired vehicles. On the other hand, for a general non-closed region, boundary cameras are different from sink and source, because vehicles can leave from one boundary camera and enter the region from another. One interesting option we can try is to create virtual cameras that connect all boundary cameras together to manually force a closed region. Although all cameras turn into interior, these virtual cameras cover the infinite activity region in terms of storing activities. Theoretically it is possible *any* vehicle from around the world could frequently leave and enter a given region. However, in reality, we believe vehicles active in a specified geographic region are largely "return customers" to the STTR system every day. The example cloud service (HistoryTrajectory) shown in Figure 2 would come in handy to archive trajectories from the boundary cameras (or the virtual cameras) into the cloud.

the three-hour simulation, we omit the threshold and start the aggregation from the beginning.

[7]Physical world is slow compared to the cyber world!

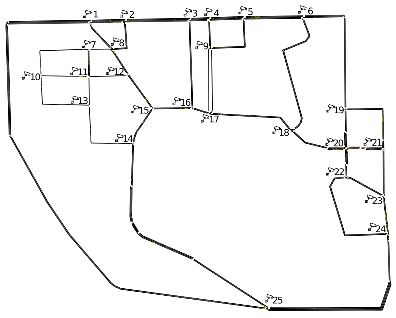

Figure 4: This figure shows the map used for traffic flow simulation and 25 cameras' deployment.

7.3 Other implementation details

For performance and our best effort to guarantee the real-time property, we adopt a lock-free system design and manually handle the collisions between threads and cameras. For example, considering the trajectory $u_{x_t}|c_i \rightarrow u_{x_{t+1}}|c_j$, in lazy trajectory aggregation, it is possible that C_i and C_j will independently make the decision to aggregate $u_{x_t}|c_i$ and $u_{x_{t+1}}|c_j$ at the same time. When the $u_{x_t}|c_i$ aggregation request reaches C_j, $u_{x_{t+1}}|c_j$ may not exist anymore. A "fail-wait-retry" protocol is used, as c_j is responsible for updating the trajectory edge information correctly. The trajectory vertex will be chosen as the victim for lazy trajectory aggregation only if it has both incoming and outgoing edges, which indicates its role in re-identification has completed. Otherwise the trajectory vertex and corresponding detected object have to be kept on the birth camera for the re-identification purpose. Backward propagation is multi-thread supported. Instead of blocking and processing one vehicle detected object each time, multi-threading can be used to improve the overall throughput, which is very helpful when the re-identification is slow and we have redundant computational resources. A simple example cloud service is implemented, which can be used to achieve the trajectory based on *LIVEness* factor. For example, $LIVEness = 3600$ which means for each vehicle, the recent one-hour trajectory will be kept at the Fog while the trajectory older than one hour will be pushed to the Cloud. This is implemented by checking the trajectory's timestamp upon each trajectory aggregation, and establishing a trajectory edge from Cloud to the Fog to guarantee the integrity of each vehicle's trajectory.

8 EXPERIMENTAL SETUP

In this section, we present the experimental setup for performance evaluation of STTR.

8.1 Traffic flow and camera stream of detections

SUMO is an open source, highly portable, microscopic and continuous road traffic simulation package designed to handle large road

Table 2: Vehicle configuration

type	length	accel	decel[25]	sigma	probability
passenger	5	2.6	4.5	1.0	0.3
passenger	5	2.6	4.5	0.5	0.3
passenger	5	2.6	4.5	0.2	0.3
bus	15	1.2	2.5	0.1	0.1

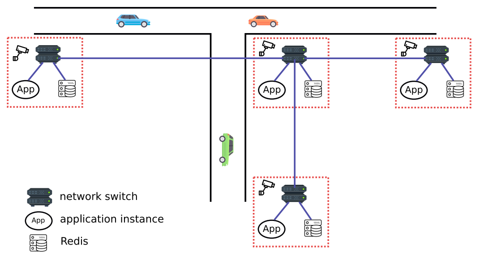

Figure 5: This figure shows the network topology we build using MaxiNet. Purple lines represent the network communication. Red dashed square represents one fog node.

networks [13]. We build a tool kit upon SUMO to generate camera stream of detections given the simulated traffic flow. Specifically, we import and simplify the campus map from OpenStreetMap [21] as shown in Figure 4. Vehicles are generated on each road following the distribution shown in the Table 2, where sigma is the driver’s imperfectness. Number of vehicles started on each road is proportional to the length of the roads. Rerouters which keep vehicles moving are generated at the end of each lane. And detectors which record the entering and leaving of vehicles are generated at beginning and end of each lane. Finally, we cluster the output of detectors by road intersections, extract the vehicle identifier, time stamp and direction of travel and publish them to cameras’ application instances as stream of detected objects.

8.2 Fog computing topology

We emulate the fog computing topology using MaxiNet [24] on a Microsoft Azure virtual machine with 32 cores, 128 GB memory. Figure 5[8] shows the example network topology. Each fog node owns two hosts (docker containers), one for STTR’s application instance and the other for the redis key-value store, and one network switch which connects these two components. Nearby fog nodes’ switches are connected such that the network topology is isomorphic with the road network in Figure 4.

9 EVALUATION

All experiments are based on a 10000-second traffic flow and camera stream emulation. Following metrics are collected during the emulation: storage usage from the Redis’s info command, network usage using Linux utilities tcpdump and tcpstat [10], computation latency for each detected object. Following variables are changed between experiments: set of cameras enabled to see how density of cameras

[8]Icons made by Smashicons, Freepik from www.flaticon.com is licensed by CC 3.0 BY

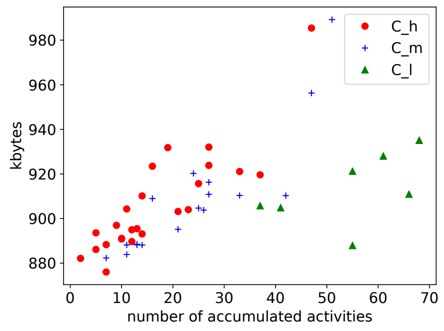

Figure 6: This figure shows the storage usage and number of activities accumulated on each camera at the end of the experiment. C_h, C_m and C_l are three different camera sets we choose above. This experiment is run under E_h and greedy trajectory aggregation.

affects the system, and synthetically inject latency to emulate varying efficiency of re-identification[9] to see how efficiency of matching affects the system. Specifically, we have three camera sets: $C_h = \{all\ cameras\}$, $C_m = \{c_1, c_2, c_3, c_4, c_5, c_6, c_{14}, c_{15}, c_{16}, c_{17}, c_{18}, c_{19}, c_{21}, c_{22}, c_{23}, c_{24}, c_{25}\}$ and $c_l = \{c_1, c_3, c_6, c_{15}, c_{18}, c_{21}, c_{25}\}$. And we have three latency settings: $E_h = 0ms$, $E_m = 100ms$ and $E_l = 250ms$, used when comparing the object signatures of the detected vehicle with the candidate pool during the forward and backward processing. The performance will be best for the experiment that uses all the cameras with negligible cost for re-identification (C_h, E_h) to use as the gold standard for all the other experimental settings.

9.1 Storage

In this subsection, we quantify the storage space usage on each camera over time. There are three factors that influence the storage usage: the road network and camera density, the traffic flow, and the trajectory aggregation. As shown in Section 4, the road network and camera density determine the maximum activity region and therefore the upper bound of storage usage on a particular camera. Meanwhile the actual traffic flow determines the distribution of routing storage usage across cameras. Finally, the trajectory aggregation will migrate data around cameras to guarantee that no camera is overloaded.

9.1.1 Activity vs. storage usage. Figure 6 shows the storage usage and number of activities accumulated on each camera at the end of the experiment. As mentioned earlier, an activity denotes that a particular vehicle is active under the camera. Hence the number of accumulated activities on a camera implies the number of vehicles whose journey ended in the camera’s activity region. Firstly, we can see that in general, the storage usage increases with the increase in the number of activities, because each camera needs to store all activities ending in its activity region. However, the storage usage on a camera does not increase linearly with the number of accumulated activities because of the size of each activity (trajectory) is not

[9]We consider the optimal case is comparing two license plate numbers for which the overhead is negligible.

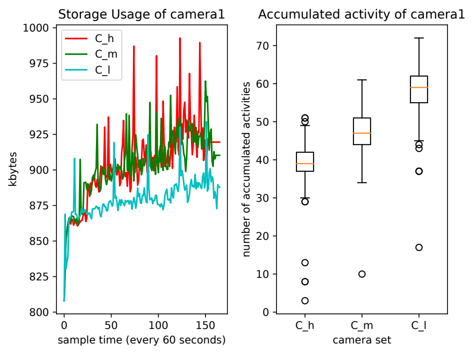

Figure 7: Storage use for a representative camera. We use Camera1 as the representative camera since it belongs to all three camera sets to show how density affects the storage usage and the number of activities. The left figure is storage usage over time, while the right figure is the box plot of the number of activities over time. This experiment is run under E_h and greedy trajectory aggregation.

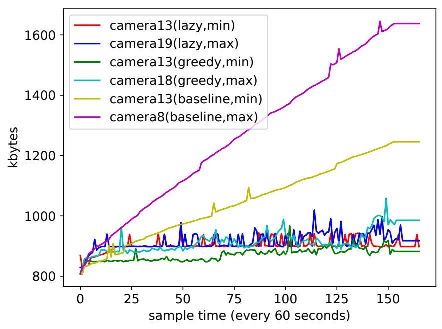

Figure 8: Lazy Vs. Greedy aggregation. Storage usage under greedy and lazy trajectory aggregation strategies are shown for representative cameras that display maximum and minimum storage use. A baseline with no aggregation is also plotted as yellow and purple line. The experiment is run under (C_h, E_h).

necessarily the same. Secondly, we observe that the number of accumulated activities on each camera increases with the decrease in the density of cameras. This is because a lower density of cameras leads to each camera being responsible for a large activity region, making them responsible for a higher number of activities. However, the storage usage does not necessarily increase, because with fewer cameras, we also have fewer detections and correspondingly the size of a trajectory is smaller. This demonstrates that the storage usage on each camera is determined by the number of accumulated vehicle activities and the size of vehicle's trajectory together.

9.1.2 Camera density vs. storage usage. To better see how density affects the storage usage, we pick camera1 and plot its storage usage and the number of activities over time for the three experimental settings (C_h, C_m and C_l) in Figure 7. Firstly, we can see the storage usage is slowly increasing, which is expected because each trajectory becomes longer over time. Meanwhile, there exists dramatic fluctuations on storage usage, which is caused by the greedy trajectory aggregation. When a large trajectory gets migrated and aggregated to another camera, the storage usage declines rapidly, and when a large trajectory gets migrated and aggregated in, the storage usage roars up. Secondly, we observe that the storage usage pattern with a medium density of cameras (green line) is lower than that of high camera density (red line). The storage usage pattern with low camera density (blue line) is further lesser than the medium density pattern. This means the storage usage is lower with fewer cameras, and the reason is that each trajectory is shorter due to fewer number of detection points. On the other hand, from the right parts of the figure, we can clearly see the general increase in the number of activities during the whole simulation when we choose fewer cameras.

9.1.3 Greedy/lazy aggregation vs. storage usage. Figure 8 compares the storage usage between greedy and lazy trajectory aggregation. We plot the cameras with the maximum and minimum average storage usage, so we can expect that other cameras' storage variation would fall between them. We also plot a baseline (yellow and purple line in the figure), which represent cases when no aggregation is triggered and every camera simply stores the detections in its activity region. The baseline, though not practical, serves as the worst case upper bound for storage need at each camera.

We can see that both greedy and lazy trajectory aggregation techniques incur much less storage usage over time than the baseline approach, which is because by aggregating a vehicle's trajectory and storing them on one camera, we save on the duplication of the vehicle meta data. Moreover, both greedy and lazy trajectory aggregation also balance the storage usage better across cameras than the baseline. With the baseline approach, where trajectory aggregation is not enabled, we see that the storage gap (between maximum and minimum storage use) keeps increasing over time because some cameras see more activities than others, which would eventually result in storage hot spots on some cameras. Meanwhile, with greedy or lazy trajectory aggregation, the storage usage will uniformly increase on each camera over time.

Next, we focus on the behavior of storage usage for greedy and lazy aggregation. Although the storage usage fluctuates a lot due to continuous data movement caused by trajectory aggregation, we still see that the distribution of storage usage under lazy and greedy trajectory do not differ significantly, which follows our claim that the lazy trajectory can also satisfy the storage upper bound in Section 4. If we compare the red line and green line, which are respectively camera13's storage usage under lazy and greedy aggregation, we do see that the lazy aggregation leads to more storage use. It is because lazy trajectory aggregation does need to maintain more meta data for the trajectory that is not fully aggregated on one camera, such as edges linking distributed trajectory vertices. Finally, we find that lazy trajectory aggregation's slow data movement can better balance the storage across cameras, as the storage usage pattern of least and most loaded cameras are quite close.

9.2 Network

In this set of evaluations, we look into the network usage of each camera over time. Similar to storage usage variation, the network

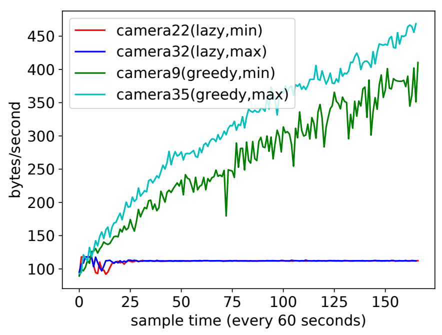

Figure 9: Network Usage for Greedy/Lazy aggregation. Results for representative cameras are shown that have the maximum and minimum network usage. The experiment is conducted under (C_h, E_h) setting.

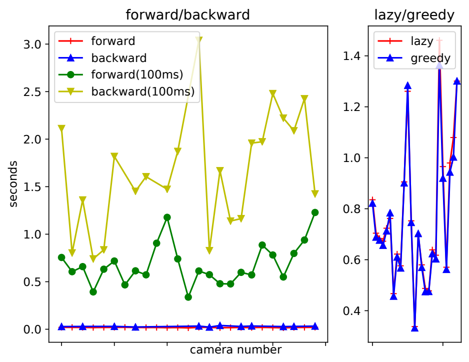

Figure 10: Latency for Forward/Backward Processing. The left subfigure shows the average latency for processing a detected object by forward/backward propagation. The green and yellow lines are under (C_h, E_m) setting, while the red and blue lines are under (C_h, E_h) setting. The right subfigure confirms that there is no latency difference between greedy and lazy aggregation. The x axis is the camera numbers 1 to 25 shown in the deployment Figure 4.

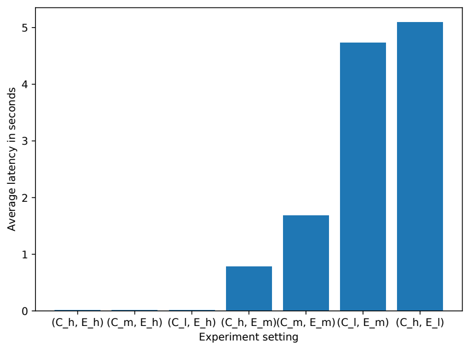

Figure 11: Latency under different experimental configurations. This figure summarizes the average latency for processing a detected object under different experimental settings.

usage will be affected by the road network, the traffic flow and the trajectory aggregation. However, the road network and actual traffic flow are accountable for network usage mainly due to the dependence of message flow patterns between cameras (forward propagation/backward) which is negligible over time compared to the data movement caused by trajectory aggregation. So we are going to focus on the difference in network usage between greedy and lazy trajectory aggregation.

9.2.1 Greedy/lazy aggregation vs. network usage. In Figure 9, we pick two cameras with maximum and minimums total network usage during the experiment from greedy and lazy trajectory aggregation, and we plot their network usage in kbytes/second over the whole simulation. As we can see the greedy aggregation will cause the network usage to increase continuously over time because of the growing size of the trajectory, while on the other hand, the lazy aggregation almost has a constant network usage over time. In addition, we observe that the maximum and minimum network usage for cameras under lazy aggregation are roughly the same. This result shows that all cameras place roughly the same load on the network infrastructure with lazy aggregation, which is because each lazy aggregation is composed of fixed number of trajectory records[10] due to the increased activity in the camera's region.

9.3 Latency

In this subsection, we are exploring the factors that affect the latency of processing each detected vehicle. This latency is defined as the time interval between receiving a "detection notification" and completion of the re-identification and writing the trajectory into the key-value store. Since this latency is unaffected by the choice of lazy Vs. greedy aggregation strategy, we show the results only for the lazy aggregation.

9.3.1 Forward/Backward vs. latency. A detected object can be processed either through forward propagation or backward propagation. Figure 10 shows the latency difference between these two strategies. We can see forward propagation always outperforms backward propagation and the difference increases with the increasing latency in re-identification algorithm which we inject synthetically. This is because the system is optimized towards forward propagation as a fast path trying to get an immediate re-identification result and backward propagation as a slow path to make sure not losing the track of vehicles.

9.3.2 Density of camera and efficiency of re-identification vs. latency. The optimal case for computing the similarity between two detected vehicles is comparing their license plates for which the overhead is negligible and represented by the E_h experimental setting. However realistically, the re-identification algorithm may use some appearance-base similarity as its basis for comparison (e.g., the L^2 distance between two large signature vectors), We use E_m and E_l to simulate this extra latency. Similarly, we might not be able to afford cameras at each road intersection, and we use C_m and C_l to simulate the sparse camera distribution. From Figure 11, we can see that the extra latency in comparing the signature dominates the latency of processing each detected object. There is a clear latency

[10]We aggregate 5 trajectory records in the implementation, although 1 also works!

increase between experiments under E_h to E_m and E_m to E_l. On the other hand, the density of cameras has much less influence, which is further declining when we have less extra latency. For example, we can see under $E_m = 100ms$, C_m takes 2.15x time to process a vehicle detection than C_h. However, under $E_h = 0ms$, the camera density does not affect the result. All these experimental results demonstrate that an efficient re-identification algorithm is critical for the smart camera surveillance's real-time property.

10 CONCLUSION

STTR is a distributed smart camera-based system for registering the trajectories of all the vehicles all the time built assuming a geo-distributed Fog infrastructure. The key insight is to focus on the activities of vehicles in the vicinity of a camera rather than the time dimension to bound the storage requirement for storing the aggregated trajectories of the vehicles. The other insight is to judiciously communicate the vehicle detections at a given camera (forwards and backwards) exploiting the deployment knowledge of the cameras on the roadways, reducing the latency for detection and re-identification tasks, and bounding the inter-node communication requirements. We have built a toolkit on top of SUMO to generate traffic flows and detectors, simulate vehicle movements, and the corresponding camera streams. We implement and evaluate STTR with the above tool kit and Maxinet [24] on Microsoft Azure to experimentally verify the theoretical assertions about finite storage space requirement for activity-based space-time tracking of vehicles. There are two areas of future work. Our current system focuses purely on generating the space-time tracks with low latency and storing them with finite storage space on the Fog nodes. An immediate future work is creating efficient spatial and temporal index structures for answering queries on the stored trajectories. Another avenue for future work is increasing the confidence in the accuracy of the space-time tracks from the systems side. While the detection and re-identification algorithms are the forte of domain experts (computer vision), there are opportunities from the system side to bound errors in detection and re-identification by adopting a probabilistic approach to trajectory generation and maintenance. We are also planning to work with the campus police department to perform *in situ* studies using STTR.

ACKNOWLEDGEMENTS

This work was funded in part by an NSF Award (NSF-CPS- 1446801) and a grant from Microsoft Corp. We thank members of Georgia Tech's Embedded Pervasive Lab and the anonymous reviews for helping to improve the presentation.

REFERENCES

[1] Mohammad A Alsmirat, Yaser Jararweh, Islam Obaidat, and Brij B Gupta. 2017. Internet of surveillance: a cloud supported large-scale wireless surveillance system. *The Journal of Supercomputing* 73, 3 (2017), 973–992.

[2] Clemens Arth, Christian Leistner, and Horst Bischof. 2007. Object reacquisition and tracking in large-scale smart camera networks. In *Distributed Smart Cameras, 2007. ICDSC'07. First ACM/IEEE International Conference on.* IEEE, 156–163.

[3] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing.* ACM, 13–16.

[4] Hu Cao, Ouri Wolfson, and Goce Trajcevski. 2006. Spatio-temporal data reduction with deterministic error bounds. *The VLDB Journal-The International Journal on Very Large Data Bases* 15, 3 (2006), 211–228.

[5] Ching-Tang Fan, Yuan-Kai Wang, and Cai-Ren Huang. 2017. Heterogeneous information fusion and visualization for a large-scale intelligent video surveillance system. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 47, 4 (2017), 593–604.

[6] Mengran Gou, Srikrishna Karanam, Wenqian Liu, Octavia Camps, and Richard J Radke. 2017. Dukemtmc4reid: A large-scale multi-camera person re-identification dataset. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops.*

[7] Marios Hadjieleftheriou, George Kollios, Vassilis J Tsotras, and Dimitrios Gunopulos. 2006. Indexing spatiotemporal archives. *The VLDB Journal* 15, 2 (2006), 143–164.

[8] Arun Hampapur, Lisa Brown, Jonathan Connell, Ahmet Ekin, Norman Haas, Max Lu, Hans Merkl, and Sharath Pankanti. 2005. Smart video surveillance: exploring the concept of multiscale spatiotemporal tracking. *IEEE signal processing magazine* 22, 2 (2005), 38–51.

[9] Arun Hampapur, Lisa Brown, Jonathan Connell, Sharat Pankanti, Andrew Senior, and Yingli Tian. 2003. Smart surveillance: applications, technologies and implications. In *Information, Communications and Signal Processing, 2003 and Fourth Pacific Rim Conference on Multimedia. Proceedings of the 2003 Joint Conference of the Fourth International Conference on*, Vol. 2. IEEE, 1133–1138.

[10] Paul Herman. 2009. *TCPSTAT.* https://frenchfries.net/paul/tcpstat/.

[11] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. 2013. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing.* ACM, 15–20.

[12] Yasutomo Kawanishi, Daisuke Deguchi, Ichiro Ide, and Hiroshi Murase. 2017. Trajectory Ensemble: Multiple Persons Consensus Tracking across Non-overlapping Multiple Cameras over Randomly Dropped Camera Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops.* 56–62.

[13] Daniel Krajzewicz, Jakob Erdmann, Michael Behrisch, and Laura Bieker. 2012. Recent Development and Applications of SUMO - Simulation of Urban MObility. *International Journal On Advances in Systems and Measurements* 5, 3&4 (December 2012), 128–138. http://elib.dlr.de/80483/

[14] Ralph Lange, Frank Dürr, and Kurt Rothermel. 2008. Scalable processing of trajectory-based queries in space-partitioned moving objects databases. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems.* ACM, 31.

[15] Ralph Lange, Frank Dürr, and Kurt Rothermel. 2011. Efficient real-time trajectory tracking. *The VLDB Journal-The International Journal on Very Large Data Bases* 20, 5 (2011), 671–694.

[16] Xinchen Liu, Wu Liu, Huadong Ma, and Huiyuan Fu. 2016. Large-scale vehicle re-identification in urban surveillance videos. In *Multimedia and Expo (ICME), 2016 IEEE International Conference on.* IEEE, 1–6.

[17] Nirvana Meratnia and A Rolf. 2004. Spatiotemporal compression techniques for moving point objects. In *International Conference on Extending Database Technology.* Springer, 765–782.

[18] James C Miller, Matthew L Smith, and Michael E McCauley. 1998. *CREW FATIGUE AND PERFORMANCE ON US COAST GUARD CUTTERS.* Technical Report.

[19] Jinfeng Ni and Chinya V Ravishankar. 2007. Indexing spatio-temporal trajectories with efficient polynomial approximations. *IEEE Transactions on Knowledge and Data Engineering* 19, 5 (2007), 663–678.

[20] Chiao-Fe Shu, Arun Hampapur, Max Lu, Lisa Brown, Jonathan Connell, Andrew Senior, and Yingli Tian. 2005. Ibm smart surveillance system (s3): a open and extensible framework for event based surveillance. In *Advanced Video and Signal Based Surveillance, 2005. AVSS 2005. IEEE Conference on.* IEEE, 318–323.

[21] ©OpenStreetMap contributors. [n. d.]. *OpenStreetMap.* https://www.openstreetmap.org/.

[22] Xiaogang Wang. 2013. Intelligent multi-camera video surveillance: A review. *Pattern recognition letters* 34, 1 (2013), 3–19.

[23] Herb Weisbaum. 2006. *What's the life expectancy of my car?* http://www.nbcnews.com/id/12040753/ns/business-consumer_news/t/whats-life-expectancy-my-car/.

[24] Philip Wette, Martin Draxler, Arne Schwabe, Felix Wallaschek, Mohammad Hassan Zahraee, and Holger Karl. 2014. Maxinet: Distributed emulation of software-defined networks. In *Networking Conference, 2014 IFIP.* IEEE, 1–9.

[25] SUMO wiki. 2017. SUMO vehicle types. (January 2017). http://sumo.dlr.de/wiki/Tutorials/SUMOlympics.

[26] Chih-Wei Wu, Meng-Ting Zhong, Yu Tsao, Shao-Wen Yang, Yen-Kuang Chen, and Shao-Yi Chien. 2017. Track-clustering Error Evaluation for Track-based Multi-Camera Tracking System Employing Human Re-identification. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2017 IEEE Conference on.* IEEE, 1416–1424.