Time Constraints and Fault Tolerance in Autonomous Driving Systems

Yujia Luo

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-39
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-39.html

May 14, 2019

Acknowledgement

I would like to thank Professor Joseph Gonzalez for advising me during my candidacy. He provided support and advice above and beyond the requirements of an advisor. I would also like to thank the rest of the CarOS project: Ionel Gog, Sukrit Kalra, Peter Schafhalter, Raluca Ada Popa and Ion Stoica.

Time Constraints and Fault Tolerance in Autonomous Driving Systems

Yujia Luo *

*Based on paper draft "CarOS: A Data-flow Operating System for AutonomousVehicles" written with Ionel Gog, Sukrit Kalra, Peter Schafhalter, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica

Abstract

Safety is one of the biggest concerns for autonomous vehicles. Safety depends on accurate driving algorithms, reliable hardware, secure networks, and real-time software systems, the last of which is what we focus on. It is critical for software systems on autonomous vehicles to respond in real time and be robust to potential system failures.

After analyzing the distinct characteristics of autonomous vehicles and the related system solutions, we believe that a specialized software platform for autonomous vehicles is in need. The goal of such a platform is to achieve strict latency and availability requirements of autonomous vehicles. We propose a more flexible time constraint mechanism that covers a wide range of application needs, as well as a hybrid fault tolerance mechanism that fits into different availability needs of different driving applications. Lastly, we present the prototype of CarOS, an open source platform for developing time-aware and highly-available autonomous vehicles.

Contents

1 Introduction

Autonomous driving is one of the biggest unsolved problems in the domain of artificial intelligence. Both academia and industry have devoted considerable amount of efforts into the field for two major benefits: improving transportation efficiency and reducing traffic accidents. On average, a person wastes $960 and 42 hours per year because of traffic jams [32]. An average urban commuter can spend up to 82 hours per year on traffic delays [55]. Besides wasting resources, approximately 1.2 million lives are lost in traffic accidents each year around the world, where 94% of the accidents are caused by human error [18; 23]. These accidents are estimated to cost $836 billion annually [62]. By replacing human drivers, autonomous vehicles are expected to address both of the problems.

However, with strict regulations, autonomous driving still remains an unsolved domain. In order for self-driving cars to be as reliable as human-driven cars, they must be driven with no disengagement for (*i*) 291 million miles without causing a fatality, and (*ii*) 5.4 million miles without being involved in an accident that causes personal injury [27]. The top self-driving car company Waymo, which has logged the most miles, has only driven 7 million miles [38; 66]. Most importantly, the disengagement rate is very high across companies. Table 1 shows the most recently reported disengagement frequency for known self-driving car companies.

Company	**Miles driven**	**Miles between disengagements**
Mercedes-Benz	1,088	1.3
Bosch	1,454	2.4
Delphi	1,810	22.1
Baidu	1,971	41.0
Drive.ai	6,572	43.5
Nissan	5,007	208.6
Cruise	125,000	1,190.4
Waymo	352,545	5,595.9

Table 1: Average self-driven miles between disengagements [12].

According to the company reports collected by California Department of Motor Vehicles [8], there are six types of disengagement causes in general:

1. Perception discrepancy
2. Unwanted maneuver of the vehicle
3. Incorrect behavior prediction of other traffic participants
4. Reckless road user behavior
5. Software discrepancy
6. Hardware discrepancy

Cause 1-3 are all related to the reliability and accuracy of the driving algorithms, which is what most autonomous driving research focuses on but not our focus. Cause 4 concerns the unexpected incidents that happen in the physical world; cause 5 indicates the delay of reaction in software systems; cause 6 describes the uncontrolled failures that occur in hardware. Causes 4-6 are all related to key *real-time systems* problems – how can autonomous vehicles engage with the driving environment in a timely manner, even with potential delays in software delay, failures in hardware, and unpredictability in the physical world?

This thesis approaches these real-time problems from a software perspective, specifically, by studying the system platform that sits between the Real-time Operating System (RTOS) and the autonomous driving applications. We will address such a platform as **Autonomous Driving System (ADS)**. We believe that

designing a time-aware and fault tolerant ADS will fundamentally solve the real-time problems, advancing system research while also providing development supports for algorithm research.

The unique characteristics of autonomous vehicles impose challenging requirements on ADS, which must support complex workloads requiring gigabyte-level throughput and millisecond-level latency. Moreover, ADS should be elastic in the face of high-variation and unpredictable environments. Meanwhile, they must be available at all times. We claim that no existing systems can meet all of these requirements.

In this thesis, we make the following contributions:

- Identify the challenges in supporting autonomous driving applications
- Propose a comprehensive time constraint mechanism for ADS to meet tight latency requirements
- Propose a hybrid fault tolerance mechanism for ADS to ensure reliability and availability
- Propose the prototype of CarOS, an open-source ADS that supports the development of real-time autonomous vehicles

The rest of the thesis is organized as follows: §2 defines the problem space. §3 discusses related systems, analyzing which aspects make them insufficient to be deployed in autonomous vehicles yet which aspects can be leveraged. §4 focuses on designing time-related mechanisms for ADS based on existing concepts. §5 analyzes the existing fault tolerance solutions and proposes a hybrid solution. §6 demonstrates the solution prototype.

2 Problem Statement

Autonomous vehicles have rigorous real-time requirements. On average, it takes a human 0.96 seconds to release the accelerator, 2.2 seconds to achieve maximum braking, and 1.64 seconds to begin steering [44]. Moreover, it takes a human 100-150 milliseconds to react to a new signal in the best scenario [49]; in autonomous driving, this is equivalent to the time from receiving a sensor input to making a control decision based on this input. In order to achieve human-level safety, autonomous vehicles should react in less than 100 ms, which is perceived as an end-to-end processing latency constraint [39; 68]. Any delay in taking actions can have catastrophic consequences. For example, if the vehicle travels at 70 miles per hour on the highway, an additional 100 ms latency can increase stopping distance by 3 meters. The concept of the time constraint is never explicit in ADS. Instead, developers try to impose "virtual time constraints" through conducting extensive testing. Certification approaches used in the automotive industry consider all possible system behaviors in the testing and certification stages, which is impossible to do for autonomous vehicles that learn and adapt online in an unpredictable physical world [40]. In addition to tight time constraints, system failures may lead to catastrophic consequences. System failures are not corner cases as the average occurrence of software and hardware faults ranges from 400 to 16,000 miles across leading companies [8].

The distinct characteristics of autonomous vehicles make it challenging to achieve such tight real-time constraints. As an evolving field, autonomous vehicles feature increasingly complicated application pipeline and diverse sensor inputs. Figure 1 shows a simplified dataflow graph of the state-of-the-art autonomous driving pipeline [1; 9; 10; 64]. We summarize the properties of an autonomous vehicle into four aspects: data forgetfulness, computation, runtime and hardware. In the following, we discuss the details of each and explain how it influences meeting real-time requirements:

- **Data is forgotten fast.** An autonomous vehicle should only operate on fresh data. Historical data are irrelevant as they will not contribute to making correct control decisions at the present. Autonomous vehicles compute over much smaller time windows (milliseconds to seconds level) than other time-critical applications do (e.g. streaming applications' time windows are usually minutes, hours or even days in size). Therefore, any stateful operation in ADS only contains a small state, and we consider such an operation as *quasi-stateless*. Containing only stateless or quasi-stateless operations, ADS need to reconsider their fault tolerance strategy.
- **Computation is high-load and diverse.** In practice, an autonomous driving pipeline can contain hundreds of processing modules [1; 65]. For example, NVIDIA's DRIVE AV runs more than 10 deep neural network models simultaneously, together with a large number of computer vision and planning algorithms, and requires approximately 250 trillion operations per second [56]. On another end, an autonomous vehicle is equipped with a large number of sensors (e.g., a dozen of cameras, several LIDARs and radars), which help improve algorithmic accuracy and increase sensor availability in case of hardware failures. These sensors are required to publish data at a minimum frequency of 10 frames per second to ensure human-level safety [39]. In practice, each sensor can generate 1-2 GB/s of data (see Table 2), which then get processed through multiple algorithms and neural networks in real-time, leading to computation requirements that are at least 100× higher than those of the vehicles in production today [1]. In such a complex dataflow graph, defining time constraints is already a problem: what exactly should these constraints restrict? A time constraint should address multiple factors such as graph dependencies and individual module's performance, which are hard for developers to monitor. The frequency divergence and convergence of data paths in the graph can also add ambiguity. For example, if a data path is influenced by another (e.g. due to synchronization), we must address how its time constraint will be affected. In addition, the diversity of computation makes it hard to find a unified solution to keep every component available at all times.

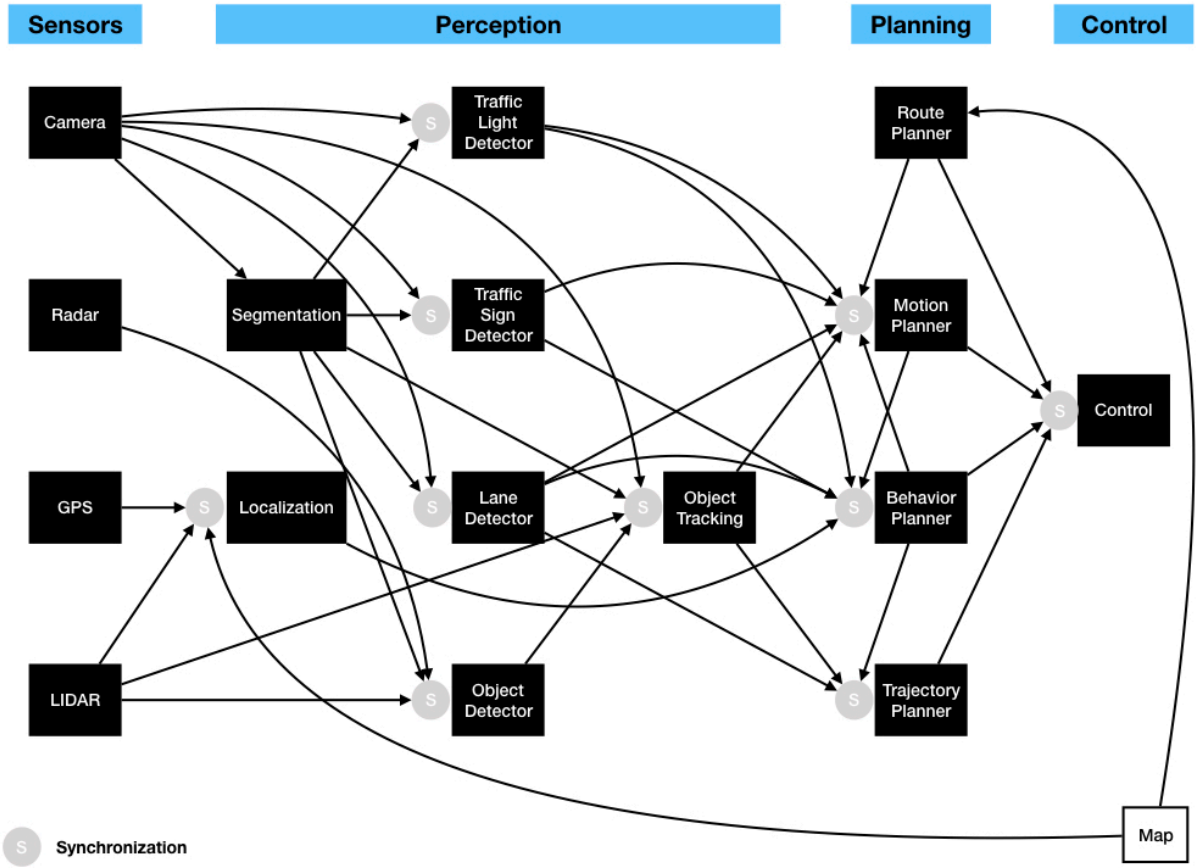


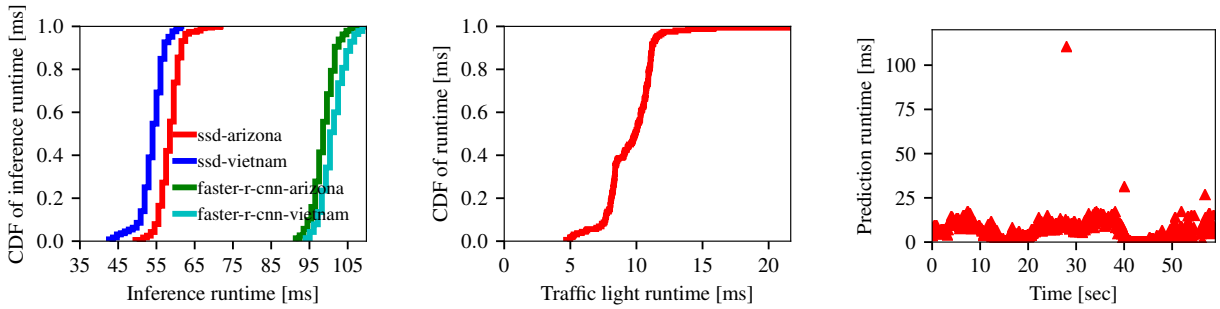
Figure 1: Simplified Autonomous Driving Dataflow Graph. There are four main parts of an autonomous driving pipeline: (*i*) sensors, which receive environment states, (*ii*) perception, which uses sensor data to detect, recognizes and localizes objects around the vehicle, (*iii*) planning, which generates vehicle's future trajectories based on perception results, and (*iv*) control, which physically operates the vehicle based on the planning results [53].

- **Runtime has high-variation and is unpredictable.** Complex computation logic and data content can lead to high variation in runtime, and such a variation can be unpredictable. There can be occasional bursts of computation load: a more complicated data (e.g. an image with more pedestrians to track) can require much longer computation time, and it is impossible to learn the true "worst case" during profiling. Figure 2 uses components in the Apollo pipeline [10] to demonstrate this runtime variation. In addition, the involvement of external systems can increase unpredictability. For example, the vehicle uses Tensorflow for modelling and communicates with external systems to retrieve mapping and traffic information. As a result, runtime profiling is no longer sufficient to ensure real-time in the physical world.
- **Hardware is constrained.** Unlike big data systems that can leverage an entire cluster of machines, autonomous vehicles only have limited amount of hardware. Such a restriction makes it harder for ADS to achieve resource elasticity by scaling up or achieve availability by running copies of all

computation.

Hardware	#	f (Hz)	BW (MB/s)
Backfly Camera	8	22	365
IMX390CQV Camera	8	60	960
VLS-128 LIDAR	5	10	75
Radar	21	40	8

Table 2: A self-driving car has several types of sensors that generate around 1 GB/s [1; 6; 65].



(a) Object detector inference runtime varies by: (*i*) up to 50% across frames from the same environment, (*ii*) up to 10% across environments for the same model, and (*iii*) up to 100% when using different models.

(b) CDF of runtime of the Apollo traffic light detector. The runtime is unpredictable, and thus Apollo either run the component at a low frequency (e.g., every 25 milliseconds) or risk missing deadlines.

(c) Timeline of the Apollo prediction component runtime. The component could cause deadline misses because the max runtime is an order of magnitude greater than the 90th percentile runtime.

Figure 2: The runtime of different Apollo pipeline components varies greatly.

Because of these characteristics, autonomous vehicles cannot avoid potential real-time constraint violations in the physical world. However, the current policies to deal with these violations are not sufficient. When the system disengages, an autonomous vehicle will execute one of the three actions: (*i*) signals humans to take over, (*ii*) stops itself and (*iii*) crashes. (*i*) is the most common, but in high (level 4) and full (level 5) automation, human intervention is not even an option [39]. Even in situations where human drivers are presented, they might fail to react due to various reasons (e.g. lack of attention [50; 51]). In event of system failures, some companies enable action (*ii*) by employing redundant hardware (e.g., deploying critical components on independent power systems) and using backup computing to safely stop the vehicle [1; 2; 66]. Nevertheless, in a safe autonomous vehicle, we should not take any of these actions and make the system robust to any potential delays or failures. We believe that no existing systems can sustain such a workflow while meeting the real-time requirements, which will be explained in the next section.

3 Related Systems

The most widely-used software platforms in developing robots and autonomous vehicles is the Robot Operating System (ROS) [9; 52; 64; 65]. However, ROS is far from satisfying our requirements for ADS, which not only have reliability, safety, and real-time requirements similar to Real-time Embedded Systems (RTES) but also have processing and throughput demands similar to Data Streaming Systems (DSS). In the following, we analyze these related systems, exploring how they are insufficient or can be leveraged to serve autonomous vehicles. Table-3 shows an overview of the system comparison between ADS and the related systems.

System	Purpose	Time-aware	Time Constraints	Dynamic Scheduling	Dynamic Dataflow	Resource Elasticity	Fault Tolerance	Message Delivery Guarantee	Modularity and Evolvability
ROS [52]	robots, autonomous driving	No	No	No	No	No	No	No	Yes
Ray [47]	RL models	No	No	Yes	Yes	Yes	Logging, Checkpoint	No	Yes
Tensorflow [3]	ML models	No	No	Yes	No	Yes	Checkpoint	No	Yes
RTES	time-critical device	Yes	Yes	No	No	No	Mostly redundancy	No	No
DSS	streaming	Yes	No	Yes	No	Yes	Checkpoint	Yes	Yes
ADS	autonomous driving	Yes	Yes	Yes	Yes	No	Hybrid	Yes	Yes

Table 3: System Property Comparison

3.1 AI Systems

ROS is originally designed for large-scale integrative robotics research. It is a flexible platform that allows developers to build evolvable and modular applications. Applications implemented using ROS consist of one or more ROS nodes (e.g, OS processes), which require users to manually deploy them on machines (e.g., fork a new process to start new ROS node). ROS nodes communicate using either an asynchronous publishers/subscriber pattern or a synchronous RPC service with no message delivery guarantee. We claim that such a design is sufficient for implementing research prototypes but not for real-time production applications. Fundamentally, ROS is just a message communication layer; it is not a real-time system that provides any form of time constraints or fault tolerance.

More recently, the prototype of a new version of ROS, ROS2 [54], was released to address the limitations in ROS but the actual system is still under heavy development. Based on its published prototype [43], ROS2 changes the original communication middleware from ROSTCP to Data Distribution Service (DDS), which provides a similar publisher-subscriber communication mechanism as ROS. DDS provides a distributed discovery system so that a master node is no longer required. It also adds shared-memory transport (ROS only uses TCP transport) and provides configurable QoS. Furthermore, Baidu's Apollo [10] has an open-source autonomous driving platform that was previously built on ROS but switched to their new open-source middleware solution, CyberRT [7], which is similar to ROS but has its own task-graph-based scheduler.

All of ROS, ROS2 and Apollo have the following limitations:

- **No Real Time:** because these systems do not provide their own time constraint mechanisms, they cannot monitor and regulate application runtime based on the constraints. ROS and ROS2 do not even have a resource manager. Each ROS node is fixed to be run in one OS process, so scheduling and

resource allocation are completely handled by the RTOS. Without overseeing the overall workflow, the system cannot make global decisions such as load shedding and message prioritization in order to ensure real-time, and message bursts cannot be mitigated with the help of back-pressure.

- **No Reliability:** these systems have no fault tolerance mechanisms. For example in ROS, the core master node which monitors publisher/subscriber topics is a single point of failure. Because they do not have state management, they can't recover a failed state.
- **No Adaptability:** it is hard to implement reactive applications that adjust when workflow (e.g. input rate) or computation duration changes in these systems. For example, ROS does not support back pressure, so messages will be dropped if a ROS subscriber node cannot keep up with the publisher node. Similarly, in Apollo, there are occasional message delay problems. Figure-3a evaluates the components built with Apollo and demonstrates the irregularity in their data sending.
- **No Message Delivery Guarantee:** these systems all use the publisher-subscriber communication mechanism, which has no order guarantees for message delivery. The publishers and subscribers maintain pre-configured fixed-size queues that drop messages when they are full. Moreover, there is no notion of progress in these systems, so that they cannot guarantee message delivery through progress checking.

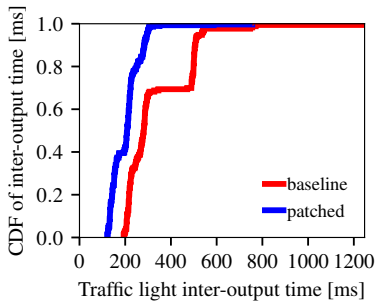
In addition to these execution platforms designed particularly for robots and autonomous vehicles, there are many other frameworks used in implementing AI algorithms such as Tensorflow and Caffe [3; 34]. However, these frameworks focus on providing fine-grain operators for machine learning model developers. They also focus on optimizing for batch processing since model training is their target application. Ray [47] is another dataflow graph execution system designed for reinforcement learning applications. In order to train and deploy learning policies, Ray needs to focus on facilitating stateful computation (e.g. simulator, model), therefore using global state storage and actor model to make state editing and management more efficiently, a functionality that is at odds with the needs of ADS. Moreover, Ray is not a real-time system so it is not time-aware and does not provide any time constraint mechanism, and its message delivery latency is higher than that in ROS.

3.2 Real-time Embedded Systems

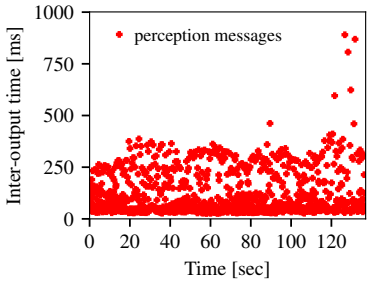
Real-time Embedded Systems (RTES) are systems that are built on Real-time Operating Systems (RTOS) and execute multiple tasks in the real world with strict time requirements and limited hardware resources. Such systems are used in a wide range of devices that assist humans in interacting with the real world, such as telephones, vehicle controls, military weapon devices and aircraft controls. Depending on the needs of the devices, RTES have different levels of strictness in terms of their time constraints: hard, soft and firm. Missing a hard time constraint is considered a system failure and will cause catastrophic consequences such as loss of human lives. Systems like aircraft controls usually have hard time constraints. Systems with soft time constraints try to reach deadlines but do not fail if a deadline is missed; instead, they degrade their performance metrics to signal the system to improve responsiveness. Systems like cruise control and entertainment video delivery software fall into this category. Furthermore, systems with firm time constraints treat information delivered or computations made after a deadline as invalid while degrading their performance metrics, which is closer to a soft real-time system. Systems like robots on the assembly lines could be considered to have firm time constraints. ADS are considered most similar to the hard RTES like aircraft control systems because they share a similar level of safety concern.

However, there are still multiple major aspects that hard RTES are at odds with ADS:

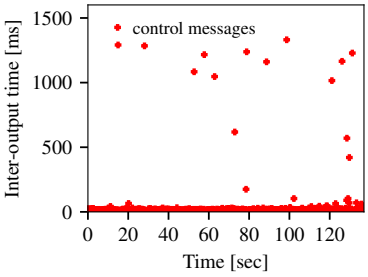
- **Strictness of Time Constraints:** although both systems can cause catastrophes when violating a time constraint, ADS should not consider it as a system failure; instead, they can initiate different strategies to deal with violations, such as stopping the vehicle or shedding loads to prevent future



(a) CDF of message inter-output time of the Apollo traffic light detector. Apollo's camera runs at 8 FPS (i.e., inter-output time should be 125 milliseconds plus traffic light detector runtime), but the traffic light detector sometimes does not output messages for over 1 second.



(b) Timeline of message inter-output time of the Apollo perception component. Despite frames arriving every 125 milliseconds, the component often does not output any messages for more than 250 milliseconds, indicating that frames are dropped.



(c) Timeline of message inter-output time of the Apollo control component. The mean inter-output time of control messages is 10 milliseconds, but the controller sometimes does not output commands for over one second.

Figure 3: Apollo's components do not output data regularly. They unpredictably, fail to meet deadlines and to output data.

violations, but these strategies are not options for an aircraft. Therefore, ADS should not enforce hard time constraints.

- **Predictability and Determinism:** a key requirement for tasks running on hard RTES is predictability - the timing behavior of all tasks must always be within an acceptable range [58], but this is infeasible for autonomous vehicles to achieve (explained in §2). On another end, hard RTES usually deal with deterministic workflow. Therefore, most research efforts in real-time systems target RTOS and programming languages [46], aiming at avoiding non-deterministic operations in the execution paths (e.g., page faults, dynamic memory allocation, synchronization primitives that block indefinitely). However, since ADS ensure real-time by having time constraints influenced by wall-clock time and dynamic dataflow strategies to meet time constraints (see details in §4), they are dealing with non-deterministic workflow in the application level. As a result, studying RTOS does not solve our problem.
- **Isolation:** the key design principle in hard RTES is isolation - different processes have different priorities, and high priority processes run on separate machines. For example, flight control is physically separated from cabin environment control. However, ADS are not self-contained as they depend on external large codebases (e.g. Tensorflow) and communicate with external systems (see details in §2)
- **Throughput:** ADS deal with much higher throughput than most hard RTES do. For example, a modern luxury vehicle has 60× more code than an F-22 fighter jet [40] as the land has much more information to be processed than the sky. Moreover, these aircraft usually have a straighter and clearer route to follow while having less external objects to interact with.
- **Testing:** Given the multitude of situations they can encounter, autonomous vehicles are validated on public infrastructure instead of in-house simulation. Testing on the road exposes the vehicles to significant risks.
- **Modularity and Evolvability:** autonomous vehicles are required to be updated much more frequently

than hard RTES because autonomous driving is still an unsolved domain undergoing rapid innovation. RTES typically target problems whose solutions are well understood and stable. As a result, the developers of RTES assume they rarely change and often design them as monolithic systems that tightly integrate hardware and software.

All the above points besides the last one show how RTES and ADS are designed for very different purposes and that RTES is insufficient for building autonomous vehicles. The last point on modularity and evolvability reveals the key reason we should not use RTES as a backbone for ADS. However, the reliability requirements of RTES is very similar to that of ADS. Hardware redundancy and software diversity are the most common fault tolerance mechanisms implemented in RTES, while some RTES also leverage a hybrid solution that incorporates a diverse set of mechanisms [4], which could potentially be leveraged by ADS.

3.3 Data Streaming Systems

Distributed data processing system is a well-studied domain in system research. It is divided into two major areas: batch processing and stream processing. ADS have fewer overlaps with batch processing systems [20; 31; 69], which target big data analytic applications and interact with finite data stored in the database. Their primary goal is to optimize throughput. They ignore the continuous and real-time nature of data. On the contrary, Data Streaming Systems (DSS) [5; 14; 48; 63] share some properties with ADS as they both deal with infinite and continuous streams of data (e.g. webpage monitoring logs, stock market trading data, bank transactions) and these streams are required to be processed immediately.

Similar to ADS, DSS are time-aware, but they are still different from ADS in the following aspects:

- **Control of Data Source:** DSS may interact with external remote applications (e.g., webpage running on a different machine, cloud databases) to get data source. Therefore, they do not have control over the arrival time and order of the data because the data may be lost or delayed. In contrast, ADS rely on in-house sensors which communicate directly with the system on the same machine.
- **Latency:** DSS are still not fast enough given autonomous vehicles' high processing load and millisecond-level execution latency requirement.
- **Time Constraints:** DSS have no notion of deadlines. Their notion of latency requirement is more of a Quality of Service (QoS) measurement component running on the side to monitor the overall performance over a certain time span. ADS require time constraint specification and monitoring on a per-message-level granularity.
- **Data Forgetfulness:** sometimes DSS require large time windows (state) to compute over historical data. Quasi-stateless operations are much more common in autonomous vehicles than streaming applications.
- **Fault Tolerance:** rollback is a common strategy to ensure reliability in streaming systems. However, because of the quasi-stateless property of autonomous vehicles, recovering historical data is not a concern for ADS.
- **Resource Elasticity:** DSS usually run in large-scale cluster environments, which allow them to achieve better resource elasticity. Resource elasticity is referring to dynamically allocating computation resources among tasks during runtime so as to meet performance metrics and improve resource utilization. DSS leverage mechanisms such as resource scale-up [5; 15; 41; 67]. However, ADS are unable to achieve this level of elasticity due to their hardware constraints.

Regardless of the differences in application requirements, the dataflow model of DSS is applicable to ADS. Decades of research in dataflow systems have demonstrated their ability to simultaneously support high-performance computation and fast-evolving applications. Modern dataflow systems usually follow a master-worker architecture pattern and use a dataflow graph at execution time to oversea the tasks and the communication between them. Such a mechanism can not only provide application developers with a more

unified programming model, but also facilitate system developers to implement time constraint and fault tolerance mechanisms based on global decisions. Therefore, we claim that ADS should be designed using a dataflow approach.

4 Time Constraints

We describe a list of time-related mechanisms that are important to make ADS real-time. For each mechanism, we analyze their existing solutions and whether these solutions are applicable to ADS.

4.1 Timestamped Dataflow

Before addressing time constraints, we should first discuss the notion of time. On a high level, there are two types of time: physical (wall-clock) time and logical time. Physical time makes a system non-deterministic, so the system usually has its own clock system that creates logical timestamps to indicate time-related properties of the data. For systems that are time-critical and message-order-aware (e.g., DSS), their dataflow usually carries timestamps. These timestamps serve the purpose of (*i*) maintaining processing order of data, (*ii*) tracking progress at asynchronously running operators, (*iii*) measuring time-related metrics of the system (e.g. QoS) and (*iv*) realizing time constraints.

To satisfy (*i*) and (*ii*), a timestamp needs to serve as an order number for the message. For example, a timestamp can indicate the creation time of the message; it is created at the source operator of the dataflow graph and can be defined differently based on the application need (e.g., event time, ingestion time) [5; 14; 63]. Naiad [48] uses more advanced logical timestamps to achieve (*i*) and (*ii*) for cyclic graphs. Its logical timestamps include iteration counters that indicate data's position within a certain iteration. In systems that needs to meet QoS requirement [42], timestamps are only created in certain time interval to measure QoS metrics. Other systems like PTIDES [21] use timestamps to implement deadline mechanism (see details in 4.4). ADS should at least achieve (*i*), (*ii*) and (*iv*).

4.2 Clock Synchronization

If logical time is used in a distributed system, clock synchronization can be an issue. If the system has operations that require a synchronized clock, there are two solutions: (*i*) ensure single system clock and (*ii*) synchronize when necessary. (*i*) is rigid and not robust as all components need to progress in lockstep, and (*ii*) can create high communication overhead. Some systems [22; 35; 45] choose to loosen such a synchronization criteria by allowing an components to be synchronized for a bounder time error combined with user-provided upper time bounds on processing and network latencies.

However, clock synchronization is a less severe problem for ADS. Unlike data centers where fine grained clock synchronization is difficult to achieve since data is received from external sources, ADS can easily achieve synchronized clock because

- Data are generated by local sensors
- Hardwares are not shared across entities
- Network traffic is generated and controlled by a single entity

ADS can leverage PTPd on the top network priority [26] to synchronize clocks across devices with high precision.

4.3 Progress Tracking

Progress tracking is concept created in DSS to better ensure the order of processing data that may arrive out-of-order. The traditional ways to deal with orderly dataflow include rollback and sending null messages. Some systems speculatively process events even when there is no assurance that these events are in order, and roll back if necessary [33], while some send null messages in order to provide lower bounds on the time stamps of future messages [17]. To avoid sending redundant null messages, PTIDES [21; 70] uses minimum model-time delay (static analysis) to avoid null messages while capturing dependency.

In modern DSS [5; 14; 48; 63], three alternatives of progress tracking approaches were proposed to avoid sending null messages: (*i*) watermarks, (*ii*) out-of-band progress notifications and (*iii*) clock synchronization

combined with user-provided upper time bounds on processing and network latency.

In method (*i*), watermarks are generated in the source operators of a dataflow graph, passed to the downstream operators, and flow as part of the dataflow [5; 14]. Watermarks help different components in an application share progress with each other, therefore preventing them from waiting for data sources that are outdated or do not exist. With watermarks, each component can make progress-based decisions locally. However, given the tight latency requirement in ADS (e.g., 100ms), Figure-4 shows the end-to-end latency limitation of Flink's watermark delivery. There are also several other limitations of watermark mechanisms:

- **P1: Slow Path Blocking.** In an intermediate synchronization operator, progress depends on all input streams and the system users cannot configure progress dependency. As a result, the operator can get blocked by a slow input stream because it requires watermarks from all inputs to send downstream progress. In the best case when there are no synchronization in the dataflow, the system can still be blocked as it progresses at the frequency of the lowest frequency source operator.
- **P2: Loop.** In a cyclic graph, if there is no way for an operator to identify watermark coming from a backtracking input stream, the operator will receive backward progress, thus causing errors.
- **P3: Future Progress Blocking.** It is not possible to issue future progress at intermediary operators. Here is a sample case in autonomous driving: a prediction operator receives object bounding boxes every n camera frames received by its upstream detection operator, and predicts locations of the boxes for $N-1$ future frames. If $n = 4$ and the prediction operator receives a frame with event time t, it will predict and issue bounding boxes with event time $t+1, t+2, t+3$, pass along a watermark with event time t, and wait for the next frame $t+4$ to arrive. Ideally, we want it to pass a watermark with event time $t+3$ so that it does not block downstream progress unnecessarily, but these watermarks, once generated by source operators, cannot be modified by the intermediary operators.
- **P4: Network Overload.** To keep the system up-to-date, source operators should continuously generate watermarks (ideally one watermark after each message), yet this may lead to network overload.

Method (*ii*) uses a global progress tracker to receive progress from individual operators and broadcast out-of-band notifications to all [48]. Such a mechanism executes a distributed algorithm, which does not provide low latency progress tracking. It not only does not solve P1 and P3 problems in (*i*), but also makes it hard for the dataflow to be reconfigured at runtime (e.g., operator scale-up). Method (*iii*) requires users to provide upper time bounds for each operator and the network, which is not feasible in the rapidly changing environment of autonomous vehicles development.

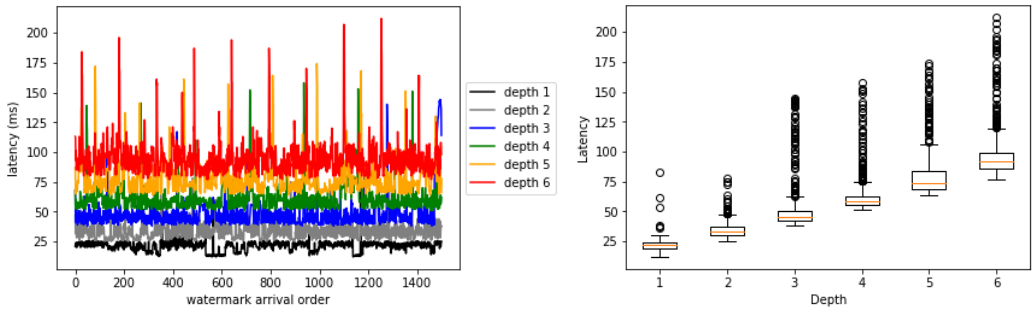
We claim that (*i*) can potentially be a suitable progress tracking mechanism for ADS if we add extra logic to allow watermark modification at intermediate operators. While we can eliminate P1-P3 by doing so, we cannot avoid P4.

4.4 Time Constraints

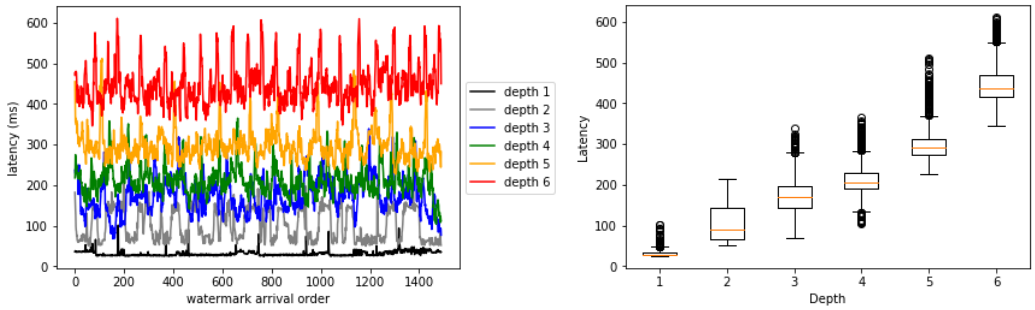
Time constraint is a common concept in real-time systems. A time constraint restricts a certain property of the system. There are three questions to ask when designing a time constraint:

- Q1: What does it restrict?
- Q2: On what granularity level does it restrict?
- Q3: Does it restrict a self-contained process in the system or a process that interacts with the physical world?

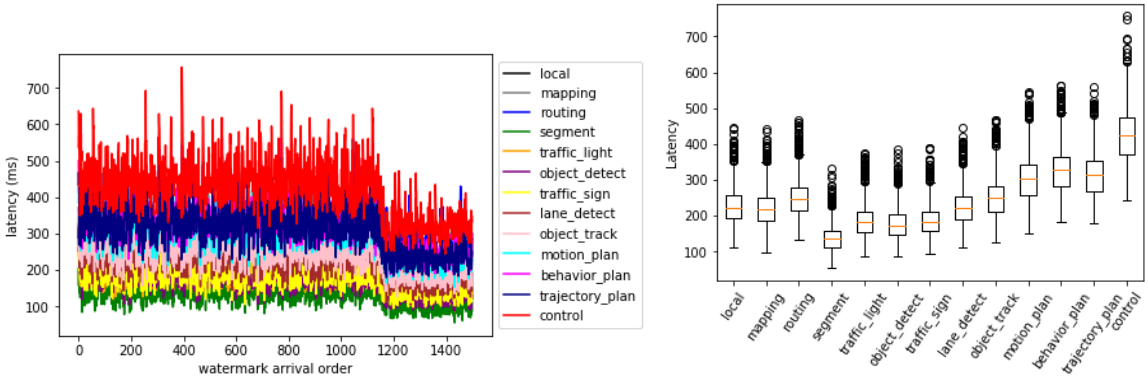
From one perspective to address Q1, a time constraint can either restrict the performance of the system (e.g. latency) or regulate the behavior of the system (e.g. response rate) [61]. From another perspective, a time constraint restricts the time elapse between the occurrence of two events and it serves as an upper bound, a lower bound, or a fixed duration [19]. These events can be either stimuli from the environment (e.g. camera sends in an image) or responses from the system (e.g. control outputs an action). Existing



(a) End-to-end latency timeline of no-window pipelines: a no-window pipeline chains a sequence of dummy operators with no computation latency. The depth of the pipline denotes the number of the chained operators.



(b) End-to-end latency timeline of window pipelines: a window pipeline chains a sequence of window (synchronization) operators. Each window waits for one message from each of the two input streams and joins them.



(c) Latency breakdown of synthetic pipeline: the pipeline follows the dataflow graph in Figure 10 but with either dummy or window operators.

Figure 4: Flink Progress Tracking Latency Evaluation. The latency denotes the watermark travel time. At the source operator, we send one watermark after each message, which is a 1024x768x3 (2.25MB) image frame published at 30 frames per second frequency. The latency spikes in the graphs are caused by Java's garbage collection. The experiment is setup on a 12-core machine and configured to have one task manager with one task slot. We eliminate any buffering time during message delivery.

systems use different methods to implement these mechanisms: MillWheel [5] allows users to define timers within individual computation unit to be triggered upon either wall-clock time or watermark; Nephele [42] provides QoS constraints, which are considered loose because they only impose descriptive statistics (e.g., mean latency) constraint on data items within a specified time span.

For Q2, a more granular time constraint means it exists in the lower level computing of the system. For example, ROS2 [54] uses DDS as a communication layer to achieve real-time in the sense that it allows developers to configure the deadline in order to guarantee QoS. Such a deadline only exists on the communication level, ensuring that the messages can be transported within a certain time frame and will not be dropped as in ROS. A time constraint with higher level of granularity lives in the application level. For example, Nephele [42] allows users to provide latency constraints on the job graph. Such a higher granularity time constraint can restrict more properties in the system, such as the latency of individual task running, data transporting in a channel or a sequence of tasks and channels.

For Q3, the involvement of physical world requires the system to use physical time to check time constraint violations, and thus make the system non-deterministic. On the contrary, a self-contained time constraint only uses logical time to measure an internal property of the system (e.g. data transporting latency), so it makes the system deterministic. The latter is the assumption in the domain of traditional real-time computing research, but the assumption does not hold true for ADS.

ADS require application-level time constraints that interact with the physical world. They should restrict per-message-level latency rather than the QoS descriptive statistics. They should restrict the latency of a sequence of tasks and regulate the pace of the dataflow. For a dataflow-graph-based ADS, granularity level can be further broken down into operator-level (local) and graph-level (global). In the following, we discuss Q1 in details for each granularity level.

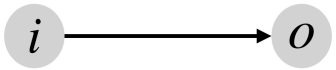
4.4.1 Operator-level Time Constraints

The purpose of operator-level time constraints is to restrict latency of individual tasks. We divide them into four categories:

- **TC1: Message Processing Deadline**: the deadline for processing input messages in an operator. An operator can have multiple inputs and outputs, resulting in multiple data paths formed by different input-output pairs. Figure 5 demonstrates how message processing deadlines are defined in different scenarios.
- **TC2: Timestamp Processing Deadline**: the deadline for processing all input messages tagged with the same timestamp, or timestamps within a specified time window.
- **TC3: Maximum Inter-output Time**: the maximum time elapsed from the last time the operator outputs a message to the time it outputs a new message. This is similar to an output frequency with an upper bound on output intervals instead of a fixed interval.
- **TC4: Loop Processing Time**: the maximum time elapsed from the operator receives an input x to it receives another input that is associated with input x. Such an association can be identified using logical timestamps [48].

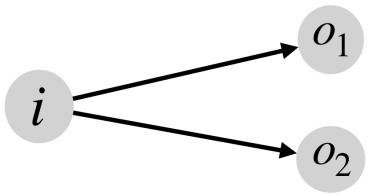
TC1, TC2 and TC4 ensure timely processing, while TC3 ensures availability. Autonomous vehicles require both guarantees: timely processing ensures that new sensor data are processed in a timely manner, while availability ensures that data progress even when they are delayed or absent (e.g., control operators must regularly output control commands). In an ideal setting where we could accurately identify which input message leads to which output message, we would not need TC3.

Having only operator-level time constraints is not sufficient because they are irrelevant if the operator does not receive input messages on time. Therefore, we need time constraints on a higher granularity level.



$p(i, o)$

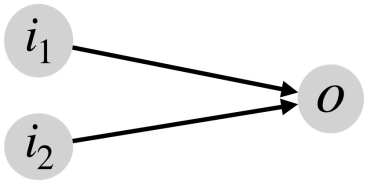
(a) 1-to-1 mapping: TC1 restricts the time elapse from input i receives a message to output o sends it.



All critical: $p(i_1, (o_1 \wedge o_2))$

1 critical: $p(i_1, (o_1 \vee o_2))$

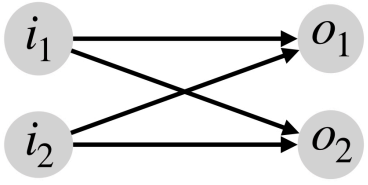
(b) 1-to-many mapping: we can either restrict one path or all paths to be critical. Restricting one path is equivalent to 1-to-1 mapping, while restricting all paths is equivalent to the processing time of the slowest path.



All critical: $p((i_1 \wedge i_2), o)$

1 critical: $p((i_1 \vee i_2), o)$

(c) Many-to-1 mapping: only happens in two types of operators: join and synchronization. Both operator types allow defining all paths as critical, in which case TC1 restricts the processing time of the slowest path. However, only join operators can deal with a mix of critical and non-critical paths: a non-critical path is behind progress, the operator will wait for it until its deadline is due, at which point the operator will just join critical stream message with either a null message or the last message received on the non-critical stream. on the other hand, synchronization operators have to wait on all paths unless we loosen the synchronization.



All critical: $p((i_1 \wedge i_2), (o_1 \wedge o_2))$

1-input critical: $p((i_1 \vee i_2), (o_1 \wedge o_2))$

1-output critical: $p((i_1 \wedge i_2), (o_1 \vee o_2))$

(d) Many-to-many mapping: a combination of 1-to-many and many-to-1 mapping.

Figure 5: Specify TC1 for Operators with Different Number of Inputs/Outputs, where p denotes the processing time.

4.4.2 Graph-level Time Constraints

The purpose of graph-level time constraints is to monitor the latency of a sequence of tasks so that we can define end-to-end deadlines. The types of time constraints are equivalent to those described in the operator-level as we can simply replace single task processing with task sequence processing. However, Figure 6 shows several challenging cases that we should consider when designing time constraint mechanisms for ADS.

4.5 Time Constraints Violation

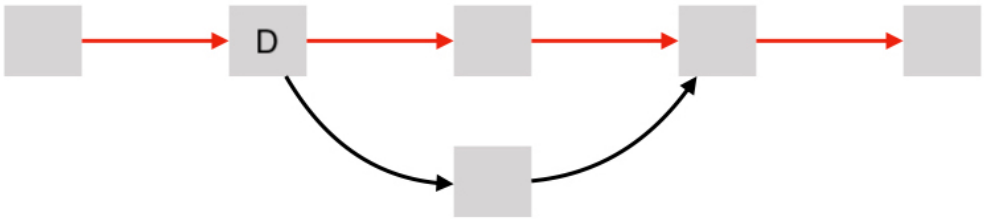
For time-critical applications, profiling is commonly used to estimate runtime for different components beforehand. However, profiling is only sufficient if either the application workflow is predictable or their performance metrics do not concern the corner cases (e.g., mean latency in QoS). For autonomous driving, neither of these two assumptions holds true. Besides software and hardware failures (addressed in §5), there are three other potential reasons that time constraints can be violated at runtime after extensive profiling: (*i*) data delay and (*ii*) bursts of computation load. (*i*) refers to the case when sensor data are not produced reliably, while (*ii*) describes the scenario when an operator receives complicated data (e.g. an image with more pedestrians to track) and requires much longer computation time. It is impossible to learn the true "worst case" during profiling.

Time constraint violations can be prevented on the operator level. In the following discussion, we assume that the system knows the estimated runtime distribution for each component in the application. With such an assumption, an operator, upon receiving a message with a time constraint, can know how much time it has left until the time constraint is violated and what the estimated time is to flow through the rest of the pipeline. With these information, the operator can make local decisions to perform preventive strategies.

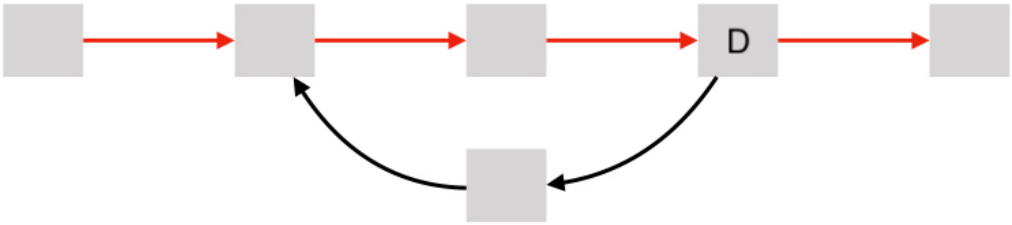
There is an important category of preventive strategies that ADS should leverage: *Dynamic Dataflow* – dynamically changing dataflow at runtime to meet time constraints. We list out a few properties that we can change about the dataflow at runtime:

- **Dataflow Content:** drop or prioritize data. Strategies like this include load shedding and message prioritization [36; 59; 60].
- **Graph Configuration:** if the system supports dynamic graph, it can scale up certain operators and execute them in parallel to deal with workload bursts [47]. For example, developers may want to spawn more object trackers in response to an object detector that detects many objects in a sequence of frames.
- **Dataflow Routing:** for critical computation, users can design a backup component or path to run in parallel. This backup should be more time efficient yet may degrade accuracy as a trade-off. Upon immediate time constraint violation, the system can take the output from the backup component. For example, a state-of-the-art object detection component in an autonomous vehicle has a relatively high process time. The user can run a faster yet less accurate detection component in parallel and have the downstream component listen to both. The downstream component only processes outputs from the more accurate detector during normal operation but will switch to the faster detector when necessary.

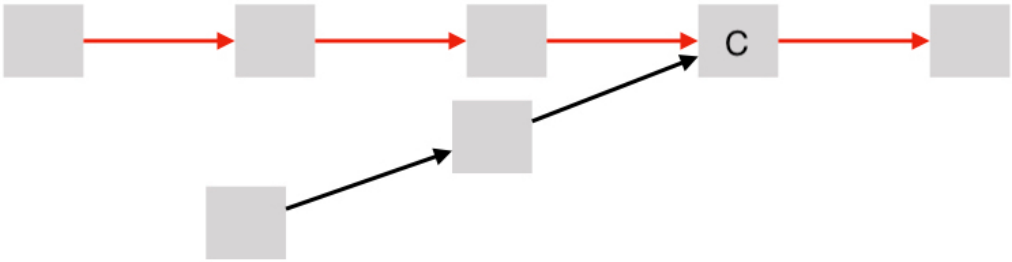
Above strategies are more effective if they can be executed on the graph level by a centralized or distributed controller, which can incorporate time constraint statuses of all critical operators to make global decisions. However, preventing time constraint violations on the graph level is out of the scope of this thesis.



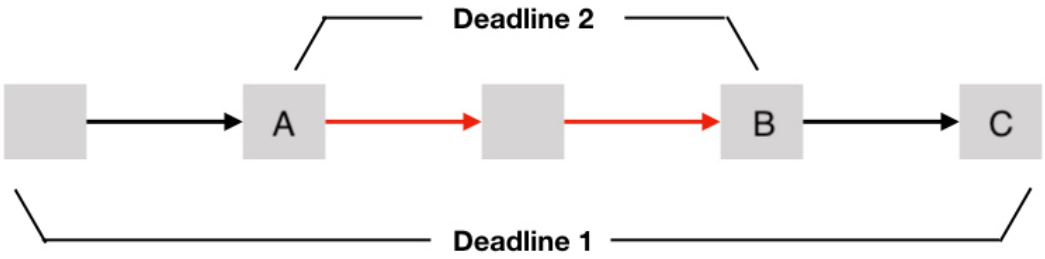
(a) Path with a Byroad: in this case a critical path cannot be defined merely by an input-output pair. Additional logic needs to be executed at the diverging point of the graph (the operator denoted D) to determine which downstream path is affected by time constraint.



(b) Path with a Cycle: this case is similar to (a) yet with a different diverging point.



(c) Path with a Merge: the converging problem at operator C is fundamentally the same as (c) in Figure 5, therefore no additional handling is required.



(d) Path with Overlapping Deadlines: part of the path is affected by two deadlines. Locally, an individual operator can use the most immediate deadline as the target constraint, but ideally the system should be able to make dynamic global decisions based on both deadlines. For example, if operator A receives a message that has 1 second left until violating deadline 2 and 2 seconds left until violating deadline 1, operator A will locally decide deadline 1 as immediate deadline. However, if the processing time between B and C is 1.5 seconds, this message may miss deadline 2. Therefore, it is important to incorporate both deadlines and make global decisions.

Figure 6: Challenging Cases for Graph-level Time Constraints

5 Fault Tolerance

As mentioned in §1, software and hardware failures are two of the key causes of disengagement in autonomous vehicles. These failures include sensor failures, computing hardware failures, and software bugs, which are hard to be eliminated through extensive testing. They can be catastrophic if the system does not handle them either proactively or reactively. For example, the vehicle can suddenly stop in flowing traffic if the control component fails and the system does not have a backup. Or, the system can make a wrong decision if the pedestrian detector fails and the system misses a few critical camera frames.

Fault tolerance is a well-studied topic in system research. A variety of solutions have been proposed for applications that are time critical and process unbounded data. However, the fault tolerance problem of autonomous vehicles is distinct from others due to four aspects:

- **Strict Time Constraints**: autonomous vehicles are time critical as control actions are often required to be delivered within a strict time constraint. Such a requirement also restricts the time overhead induced by running a fault tolerance mechanism during normal operations of the system and by recovering from failures.
- **Exactly-once Message Delivery**: exactly-once delivery includes processing messages in timely order, ensuring no messages are missed and no duplicated messages are delivered. Any violations could be catastrophic: missing messages or processing disordered messages can cause significant decrease in accuracy (e.g. pedestrians would not be detected if the object tracker does not receive all bounding boxes from the detector), and duplicate messages not only waste computation resources but also produce unwanted outputs (e.g. duplicate control commands can cause vehicle to over-steer).
- **Quasi-stateless**: autonomous vehicles 'forget' about data fast, so the operators only needs to keep track of a small amount of data. This property puts an upper bound on the time it takes to recover a failed operator.
- **Resource Constraints**: a combination of the limited machine resources we can place on a vehicle and the complexity of its computation graph, we are unable to make copies of every computation unit and every data path to run them in parallel as backups.

There are existing fault tolerance evaluation metrics [30] that we can use to describe the above characteristics in a more concrete way:

- Normal Operation Overhead: the additional latency caused by running fault tolerance mechanism during normal operation when no failure occurs.
- Recovery Overhead: the additional latency caused by a failure incident.
- Recovery Precision: how different the recovered state and message delivery are from those before the failure. [30] categories recovery precision as precise, rollback and gap. *Precise* reconstruct the exact same state as before the failure and achieves exactly-once message delivery. *Rollback* reconstructs an equivalent state while duplicated messages are possible, therefore only achieving at-least-once message delivery. `Gap` spins up failed task from scratch without reconstruction of the state, achieving at-most-once message delivery.
- Resource Efficiency: how much additional resources the system needs to execute the fault tolerance mechanism.

With these metrics, we can impose four requirements on an ideal fault tolerance mechanism for ADS: (*i*) low normal operation overhead, (*ii*) low recovery overhead, (*iii*) high recovery precision, and (*iv*) high resource efficiency.

To simplify the fault tolerance problem, we assume that the operators are fail-stop – they fail by stopping execution and losing the operator state, and the failures are detectable by other operators. Furthermore, we assume that operators communicate through reliable and in-order communication protocols across the

network (e.g., TCP), and within a machine (e.g., message queues). Finally, we assume that operators are internally deterministic (i.e., they do not contain logic that depends on time or on processing time message arrival, and do not use randomization). However, we cannot assume determinism on the overall dataflow because operators may depend on the arrival order of messages on different input streams. For instance, if the user implements dynamic dataflow mechanisms (see details in §4.5), an operator may first execute data received from a faster input stream before it executes on the ones received from a slower input.

In the following sections, we discuss in details about some of the existing fault tolerance mechanisms and determine if they have potential usability in ADS. We place these mechanisms into three general categories: (*i*) amnesia, (*ii*) active replication and (*iii*) rollback. Amnesia strategies simply detect and restart failed operators, disregarding the missed messages during the failure. Active replication strategies fully replicate operators and run replicas in parallel with the primary one. Rollback strategies reset dataflow to a past time point and resumes processing from then. We discuss the trade-offs each of these solutions makes with the evaluation metrics and how it fits into the application needs of autonomous vehicles. Table-4 shows a summary of the analysis.

Mechanism	**Normal Operation Overhead**	**Recovery Overhead**	**Recovery Precision**	**Resource Efficiency**
Amnesia	Low	Low	Gap	High
Active Replication	Low	Low	Precise	Low
Logging & Replay	Low	High	Rollback/Precise	High
Checkpointing	High	Medium	Rollback/Precise	High

Table 4: Fault Tolerance Mechanisms Evaluation [30]

5.1 Amnesia

Amnesia is a straightforward mechanism as it simply starts up a new copy of the failed operator and resumes processing data received from the point of recovery without fully recovering the damage incurred from the failure. The damage can lead to (*i*) the loss of messages during failure and (*ii*) the loss of operator state. Amnesia is usually applied to stateless operators which are not affected by (*ii*). For stateful operators, Amnesia does not rebuild the original state but rebuilds states using new input messages.

Amnesia is a low-cost mechanism with low recovery precision as a trade-off. During normal operation, the system does nothing besides monitoring statuses of the operators to detect failure, so the fault tolerance overhead during normal operation is negligible and the additional resource required is low. Upon failure, the system restarts the failed operator, which also induces negligible overhead if the operator is stateless. If the operator is stateful, the recovery overhead depends on how long it takes to build a new state. Amnesia features gap recovery precision as it only guarantees at-most-once message delivery.

In ADS, Amnesia is suitable for stateless and quasi-stateless operators that are not safety critical. For example, operators that do high-level routing, path planning and GPS can miss several messages without affecting safety.

5.2 Active Replication

Active Replication achieves fault tolerance through redundancy, running multiple copies of a task/path in parallel on different machines. When designing an active replication strategy, we should consider (*i*) the trade-off between consistency and availability and (*ii*) the coordination granularity. The consistency in (*i*) refers to whether different replicas have the same the messages flowing through and whether they maintain the same states. Consistency requires extra coordination mechanism to insert synchronization (lockstep)

among replicas, making sure that all replicas run deterministicaly and receive input messages in the same order. The trade-off of having high consistency is replica coordination, which decreases the availability of the system as it will not continue processing until replicas are coordinated. A less consistent yet more available mechanism would allow replicas to run out-of-sync while fixing the out-of-sync replica later when needed [11].

Aspect (*ii*) refers to the granularity at which replicas are coordinated, which influences the number of operators affected by a failure. For example, if replicas are coordinated only at the beginning and the end of the replicated pipelines, a single operator failure brings down the whole replica, and the cost of recovering the whole replica will grow linearly with the state size of the whole replica. And before the replica is recovered, the primary running pipeline must either stop to help recover the replica (copy state over to reconstruct the state) or continue executing with decreased fault tolerance.

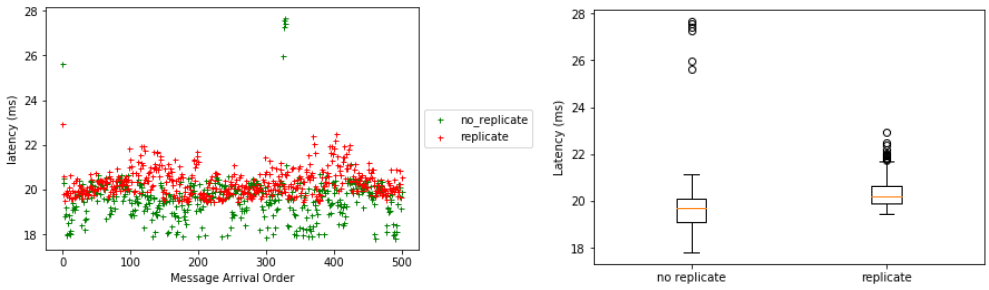
There are three existing techniques that leverage different trade-offs for (*i*). The most traditional technique is to use process-pairs [24; 25] to execute all replicas in lockstep. All replicas must be synchronized before they proceed, therefore achieving the highest consistency. Unlike process-pairs, techniques in systems like Borealis [11] provides greater flexibility by making the consistency trade-off configurable. It uses a special operator at the beginning of a fault tolerant data-flow as a coordinator that provides different extent of synchronization. If prioritizing availability, the operator allows tentative messages to be processed when any replica fails, passing down partial inputs and correcting them later when stable inputs are available again. However, in order to correct these partial messages, Borealis has to incorporate logging and replay or checkpointing mechanisms (see §5.3 for details) to re-run computation on stable inputs. On the other hand, techniques like Flux [57] offers a looser synchronization mechanism than lockstep while still maintaining high availability in most cases. It inserts special coordination operators both at the beginning and the end of a fault tolerant dataflow to record the progress difference among different replicas. Upon a replica failure, other replicas can take over and use the recorded progress to catch up with the progress right before the failure. Therefore, Flux ensures high consistency in all cases and high availability during normal operation, while sacrificing availability during failure. However, neither of Flux nor Borealis addresses fault tolerance of these special coordination operators and assumes they are always available.

Overall, active replication achieves precise recovery precision, low normal operation overhead and low recovery overhead, while sacrificing resource efficiency by doing redundant work. Its coordination mechanism ensures exactly-once message delivery but induces a small amount of overhead during normal operations. Figure 7 evaluates the coordination latency of Flux implemented on ROS. During failure and recovery, the process of recovering the failed replica might induce some latency due to state transfer, but the latency should be way lower than replaying messages. The amount of additional resources required depends on how much of the pipeline are replicated, but it will be much more than no-replication fault tolerance since it induces duplicate processing on all messages.

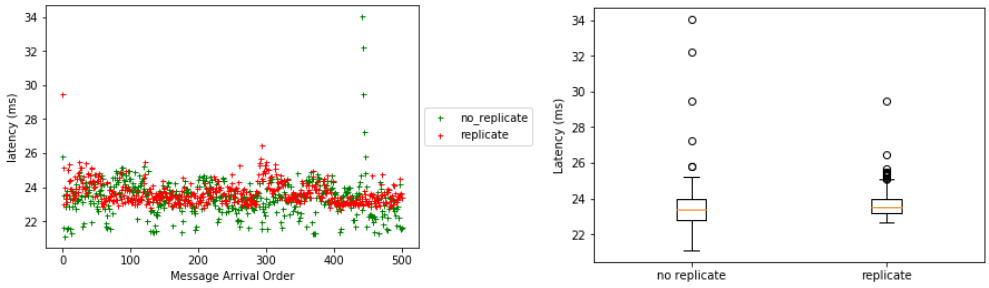
Active replication targets operations that have a low tolerance for failure and delay. In autonomous driving, operations like this include time-critical components such as the control and the object detector. These operations require high availability, high consistency within small time windows, and should not rollback to past messages. Therefore, among the three mechanisms discussed above, Flux seems to be the most suitable option. The strict synchronization in process-pairs might introduce interruption latency, and the rollback and correction mechanism in Borealis is not suitable for these critical components.

5.3 Rollback

Rollback refers to strategies that reset the system to a previous state and execute from then. Different rollback strategies involve storing different states and deploying different methods to store these states. There are two major rollback methods: (*i*) logging and replay and (*ii*) checkpointing. (*i*) only saves the



(a) Replication vs. No Replication End-to-end Latency Timeline in ROS Depth-5 Pipeline



(b) Replication vs. No Replication End-to-end Latency Timeline in ROS Depth-10 Pipeline

Figure 7: Flux Active Replication Coordination Latency Evaluation in ROS. End-to-end latency denotes the message travel time through the pipeline. The message is a 1024x768x3 (2.25MB) image frame published at 30 frames per second frequency. The experiment is set up on a 12-core machine.

changes of the states (e.g. message delivered within a certain time frame), while (*ii*) usually records the entire state of the dataflow at predetermined time points.

Logging and Replay: the mechanism features recording the change of states with different storage options. The most straightforward method is to log changes onto persistent storage (e.g. disk), from which the recovered operator can retrieve and re-execute the logs. If the state change required to log is small and can be fit into memory, we can use passive standby [30] that deploys a replicated operator to log the state changes of the primary operator in memory and replay the logs when the primary fails. Upstream backup [30] uses the upstream operator of the target operator to log the messages flowing between them. More specifically, the upstream operator logs its output messages and re-sends them when a failed downstream operator recovers. If the application wants to achieve precise recovery, both passive standby and upstream backup need to add extra coordination logic with downstream neighbors so as to drop state changes that no longer influence downstream. However, coordinating with direct downstream neighbors results in adding cascading back pressure edges from downstream neighbors all the way to the sink operators, which induces extra overhead.

If incorporating progress updates, the logging and replay mechanism will have high recovery precision (exactly-once message delivery). It has a low normal operation overhead and high resource efficiency because the operators are simply logging state changes of a single operator. However, it can potentially have a high recovery overhead since the operators need to replay the logs to reconstruct the state.

Checkpointing: it refers to saving the entire state (snapshot) of the dataflow graph with a certain frequency and rollback to one of these states (usually the most recent one) upon failure. There are two common check-

pointing strategies implemented in modern DSS: (*i*) persistent state updates and (*ii*) distributed snapshots.

DSS such as MillWheel [5] and Storm [63] require operators to only acknowledge receipt of a message after they have checkpointed the updated state and outputted messages to downstream, so upon failure and rollback, the system will only restore changes caused by the acknowledged messages. This eager checkpointing and logging strategy has several benefits: processes can dynamically join and leave the dataflow because the fault tolerance mechanism does not require to keep track of the data-flow graph, not-failed operators do not have to recover from checkpoints, and operators can choose locally when to checkpoint. However, the strategy has two main drawbacks that make it incompatible with the low latency and high throughput requirement of autonomous vehicles: it has low-throughput because all state mutations must be persisted, and has high-latency because output messages must be acknowledged by downstream operators before next incoming message can be acknowledged.

Other systems like Flink [14] and Naiad [48] optimize for overall throughput over latency and resource cost. They implement fault tolerance strategies based on Chandy-Lamport Snapshotting [16], which requires operators to checkpoint state upon the receipt of marker messages that are periodically propagated through the dataflow. The scheme guarantees that a consistent global state is checkpointed and implicitly guarantees that cascading rollbacks will not occur. However, it has several limitations. First, in order to guarantee exactly-once semantics, output operators (e.g., control) can only release outputs that are captured by the latest checkpoint. In other words, the maximum frequency at which checkpoints can be sent is coupled with the frequency at which they are taken (e.g., vehicles could only send control commands every 5 seconds if checkpoints are only taken every 5 seconds). Second, in case of a failure, all operators (including non-failed operators) must rollback to the latest checkpoint. Such rollbacks would have catastrophic effects on autonomous vehicles because they also affect safety operators, which have to restore state from checkpoints and replay old data. Third, source operators, which in Flink implicitly introduce marker messages via watermarks, control the low time bound at which checkpoints can be taken; operators can at most checkpoint after every watermark, but cannot checkpoint if they have not received a watermark. Finally, all operators must checkpoint at the same watermark frequency, regardless of the operator type and if it is a suitable time or not. Otherwise, checkpoints would not be globally consistent, and thus frameworks would have to keep a history of checkpoints, find a globally consistent view across operators, rollback to potentially older checkpoints, and garbage collect checkpoints.

Either form of checkpointing creates high latency overhead, leading to latency spikes during normal operation. Figure-8 shows this overhead by evaluating Flink, which uses a type (*ii*) checkpointing strategy called Asychronuous Barrier Snapshotting [13; 14].

Rollback recovery is suitable for operations that require either at-least-once or exactly-once message delivery. Logging and replay is suitable for quasi-stateless operators whose states depend only on a small number of recent input messages and therefore can recover quickly. Autonomous driving operations like this include the object tracker, which tracks a certain object in image frames for a few seconds and the tracking latency for each frame is very low. Moreover, checkpointing places an upper bound on how many messages must be replayed so it is suitable for operators that are quasi-stateless and cannot catch up quickly.

5.4 Fault Tolerance in ADS

Based on the above analysis, there is no single fault tolerance strategy that can satisfy all requirements for ADS. Different operations in autonomous driving require different trade-offs between latency overhead, recovery precision and resource efficiency. Table 5 illustrates fault-tolerance-related properties of each operation module in ADS and proposes the most suitable strategy.

We recommend active replication strategies for components that are on the critical path between sensors and control commands and rerunning them will potentially cause deadline misses. Any failure, delay or

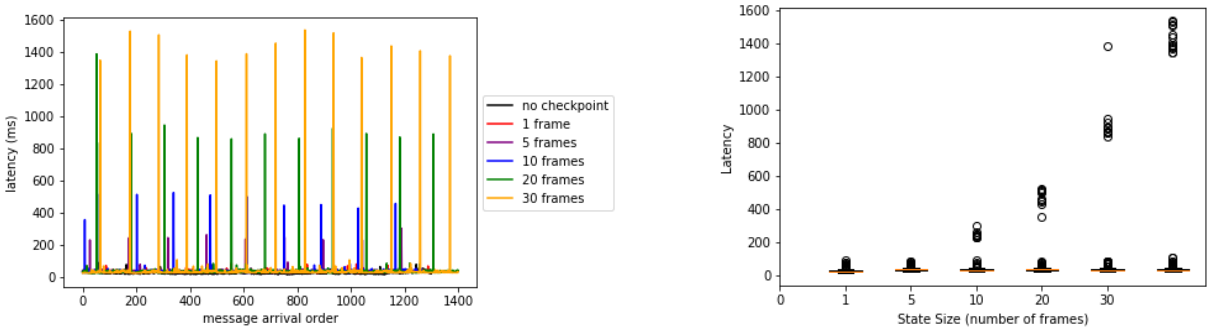


Figure 8: Evaluate Flink's checkpointing latency during normal operation (no failure) with varied total operator state sizes, which are denoted by the number of frames. The source operator publishes 1024x768x3 (2.25MB) image frames at a frequency of 30 frames per second. The system performs checkpointing every 5 seconds.

Operation	Critical	State Size	Catch Up Time if Rollback	Message Delivery Requirement	Recommended Strategy
Sensor Connector	yes	stateless	low	exactly-once	logging replay
Segmentation	yes	stateless	low	exactly-once	logging replay
Detector	yes	stateless	high	exactly-once	active replication
Object Tracker	no	quasi	medium	exactly-once	logging replay
Localization	no	quasi	medium	at-most-once	amnesia
Motion Planner	yes	stateless	low	exactly-once	logging replay
Route Planner	no	quasi	low	at-most-once	amnesia
Control	yes	stateless	low	exactly-once	active replication

Table 5: Fault Tolerance Needs of Different Components in ADS

inaccuracy occurred to them can be catastrophic. For example, recovering an object detector takes time, and potential pedestrian images might be lost during the recovery process. Active replication is suitable for this case as it ensures high availability and consistency. Checkpointing is not sufficient because it increases critical path latency and the state the operator rollbacks to would not be irrelevant by the time it completes recovery. Logging and replay is also not suitable because detector has high latency and the catch-up process will stall progress for too long.

For the quasi-stateless components that are not on the critical path and can recover states quickly, we recommend the logging and replay strategy. These components can start from scratch or rollback while still managing to recover such that the end-to-end deadline is satisfied. Such components include Kalman-filter-based object trackers which keeps buffers of sensor data to predict trajectories of objects around the vehicle.

Any non-critical components, which allow data missing yet have an upper time bound on how long the operators can miss inputs for, should adopt amnesia strategies. For example, route planners do not affect safety if they miss several inputs as long as the route is re-calculated regularly.

6 Solution Prototype

CarOS is an open-source ADS designed for building autonomous vehicles and robotics applications. The system is currently under heavy development[1], and this section provides an overview of the system prototype.

6.1 Design Goals

CarOS aims to provide users with the following properties:

- A easy-to-use programming model featuring a dataflow graph, which may contain cycles.
- Operations can have multiple inputs and outputs.
- A time constraint mechanism that can be defined on both operator level and graph level.
- A synchronization mechanism.
- A configurable hybrid fault tolerance strategy.
- A operation scale-up mechanism.
- A data logging and replaying mechanism.
- Dynamic dataflow strategies to mitigate time constraint violations.

6.2 Architecture

Figure 9 shows the architecture of CarOS. The underlying component of CarOS is an execution engine of directed dataflow graphs, which are consist of operators (nodes) and data streams (edges). Currently, we are using external frameworks, ROS and Ray, as the execution backend. CarOS provides a flexible dataflow graph model that can easily reconfigure the backend framework. However, none of these frameworks is sufficient for CarOS: as mentioned in §3, the subscribers and publishers in ROS maintain fixed-size queues that drop messages, while Ray's message delivery is too slow. Further down the road, CarOS will have its own execution backend so that we can have more control of message delivery and resource management to achieve reliable and low-latency execution.

6.3 Programming Model

In CarOS, applications are built as a graph of operators, which are connected via data streams. Specifically, an operator contains the computation logic of a task, while a data stream signifies a message delivery channel between operators. An autonomous driving graph can contain hundreds of operators and data streams, so we provide users with a easy-to-use programming model.

In CarOS, developing an application is consist of graph development and operator development. In graph development, user specifies an *Operator Dependency Graph* by connecting operator instances, without having to specify the exact data streams on which connected operators communicate. In operator development, users can select the input streams that the operator depends on and register callback functions on them. Users also define the output streams and implement connection logic of input and output streams in the callback functions. Combining these two processes will generate a *Dataflow Graph*, as illustrated in Figure 10.

```
# Declare Graph
graph = erdos.graph.get_current_graph()

# Declare Operators
camera_op = graph.add(CameraOp, name='camera')
detector_op = graph.add(DetectorOp, name='detector')

# Connect Operators
```

[1] CarOS codebase: `https://github.com/erdos-project/erdos`

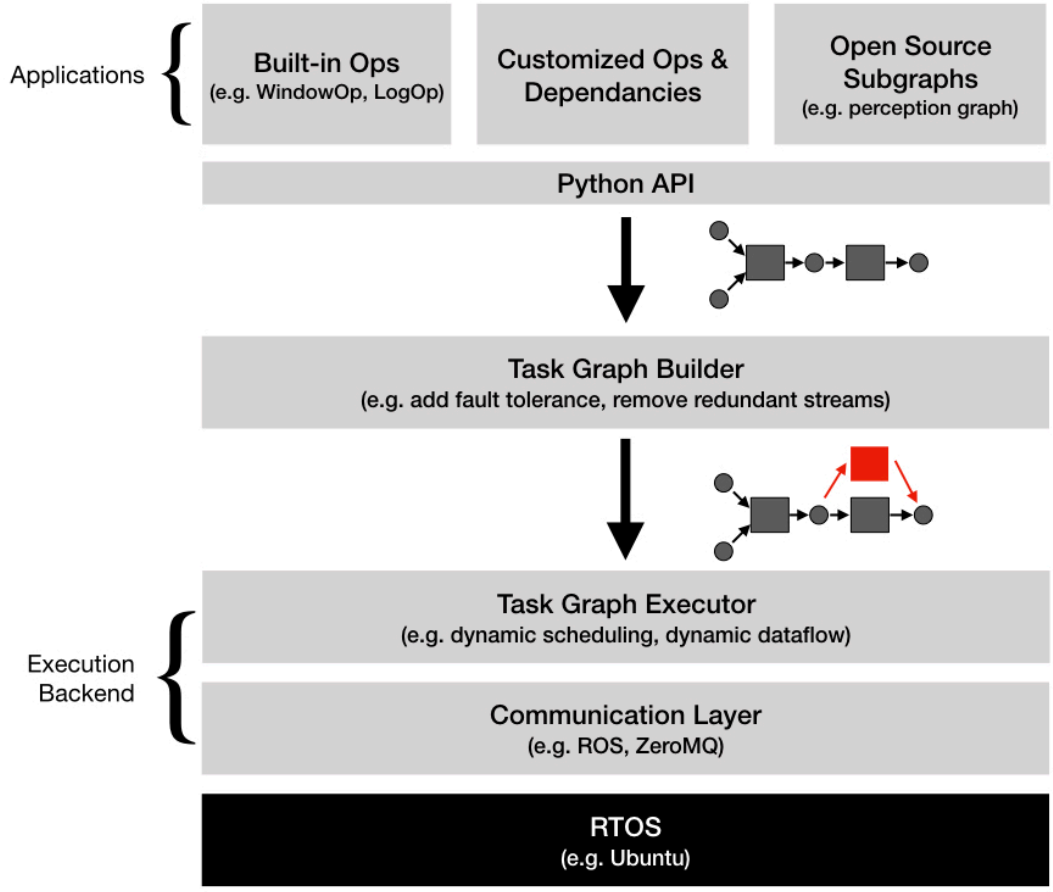


Figure 9: CarOS Architecture

```
graph.connect([camera_op], [detector_op])

# Execute Graph
graph.execute(framework)
```

Listing 1: Sample Code Example of Graph Development

```
class DetectorOp(Op):
    @staticmethod
    def setup_streams(input_streams):
        input_streams.filter(filter_function).add_callback(DetectorOp.on_msg)
        return [DataStream(name='bounding_box')]

    def filter_function(stream):
        # Select input streams that this operator depend on
          ...

    def on_msg(self, msg):
        # Callback function
```

```
13          ...
```

Listing 2: Sample Code Example of Operator Development

In traditional dataflow execution engines, streams are first-class citizens that are explicitly passed to the operators. We believe that these approaches are not ideal for applications consist of many operators that have multiple inputs and outputs. We shift the stream dependency implementation into operator development so that users can maintain the graph code more easily.

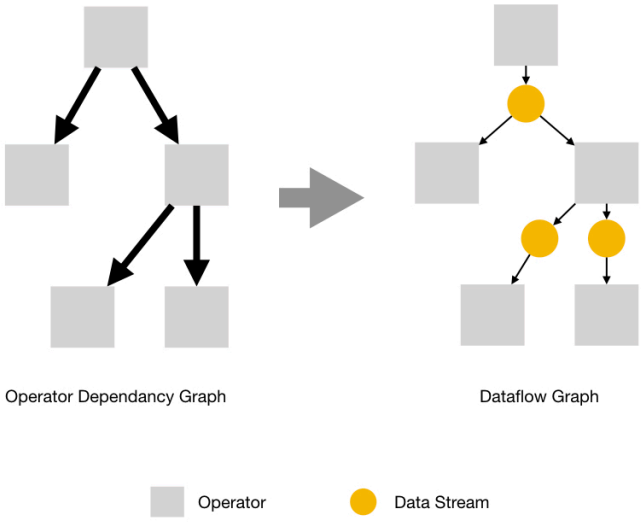


Figure 10: Transformation from high-level operator dependency graph to dataflow graph.

6.4 System Mechanisms

CarOS provides a set of built-in mechanisms that are essential to building autonomous driving applications. These mechanisms are all built upon our core programming model. In the following, we describe each of them and propose future improvements.

- **Time Constraints:** based on the assumption that all operators are executed under a synchronized clock, the system provides processing deadline and inter-output time constraints. The inter-output time constraints include fixed inter-output time (frequency) and maximum inter-output time (maximum frequency). On operator level, users can use function decorators (e.g., @deadline) to specify time bound value, callback function in case of time constraint violation and bounded input/output stream names. On graph level, users can use graph API (e.g., graph.add_deadline). This mechanism will be expanded further to cover all cases discussed in §4.
- **Fault Tolerance:** users can enable four fault tolerance strategies in CarOS: amnesia, active replication, upstream backup and checkpointing. Currently, users enable fault tolerance of an operator by strategies (e.g. replication) instead of operator properties (e.g. if time critical, state size). Ideally, users do not need to know specific fault tolerance strategies, and CarOS can provide automatic strategy matching given operator properties.

- **Progress Tracking:** CarOS adopts the watermark mechanism. Progress tracking is only enabled when user creates and sends watermarks at source operators. These watermarks are inherently message objects that flow with normal messages. Users can interact with these watermarks at intermediate operators by registering additional watermark callback functions on input streams. Moreover, users can control which part of the dataflow graph allows watermark to flow through by adding labels at an operator's output stream.
- **Synchronization:** CarOS adopts Flink's window mechanism [29] to achieve synchronization or join of multiple unbounded data streams.
- **Subgraph:** CarOS introduces the subgraph concept to better modularize the application and reduce code redundancy. Subgraphs are fundamentally graph objects, but they work like operators as users can directly connect them with operators during graph development. Figure 11 illustrate how to consolidate a graph of only operators with subgraphs. Subgraphs can also facilitate collaboration between application developers as they can easily import other open source graphs into their applications.
- **Data Logging and Replay:** data logging and replay is an important feature for research and debugging purpose. CarOS provides two options to log data: (*i*) tag operators and graphs as logged, and (*ii*) add logger graphs. (*i*) requires minimum code as it is just a switch to log all (e.g., tag log_input_streams=True for an operator logs all input stream messages). (*ii*) is provided for the purpose of selective logging. In customized logger graph, users can determine which streams and what messages of an stream to log.
- **Operator Scale-up:** users can dynamically scale-up an operator to handle increased workload (see Figure 12). Stateless operators can easily be scaled-up, but stateful operators must be handled with care. We can achieve stateful operator scale-up by leveraging the timestamps and progress tracking mechanisms [28; 37]. Users can scale-up operators by specifying provided partition functions on input streams, partitioning input streams in such a way that state can be sharded and all messages are routed to the operator copy that contains the state they change. For example, a stream of detected objects can be partition by object ID so that the bounding box of an object detected in several frames is sent to the same object tracker.

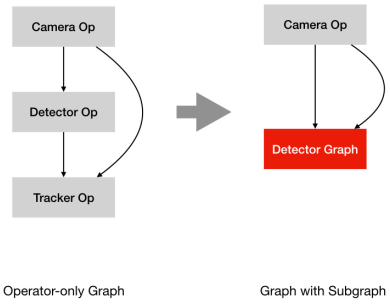


Figure 11: Subgraph

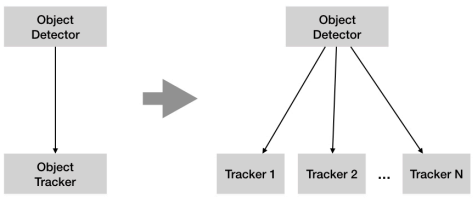


Figure 12: Operator Scale-up

7 Conclusion

In order to solve the safety problem of autonomous vehicles, we should not only improve accuracy in the driving algorithms but also design real-time software platforms that are robust to system failures and meet tight time constraints; this thesis focuses on the latter. We state that the complex, unpredictable and forgetful nature of autonomous vehicles distinct themselves from other time-critical applications, while the existing policies and systems are insufficient to support them. We then analyze related systems, drawing a conclusion that ADS should be modeled as data streaming systems, which allow data to be processed in an orderly and timely manner while providing a dataflow graph for the system to make safety-related global decisions. We discuss in details about time-related mechanisms such as dataflow timestamps, progress tracking, and time constraints, and briefly address the dynamic dataflow strategies for preventing time constraint violations. We also discuss several fault tolerance strategies and how they can be leveraged in different components of autonomous vehicles. Lastly, we describe a prototype of CarOS, an ADS that is designed particularly for meeting these real-time requirements of autonomous vehicles.

However, this thesis only defines and analyzes the problem space of ADS without proposing a validated solution. Future work should focus on validating the proposed fault tolerance and time-related mechanisms. There is also great research potential in the problem space of meeting time constraints. This paper does not address problems such as the graph-level solutions to prevent time constraint violations and the trade-off between meeting time constraints and ensuring accuracy.

References

[1] *2018 Self-driving Safety Report*. General Motors, 2018.

[2] *A Matter of Trust: Ford's Approach to Developing Self-driving Vehicles*. https://media.ford.com/content/dam/fordmedia/pdf/Ford_AV_LLC_FINAL_HR_2.pdf. Ford.

[3] Martın Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 265–283.

[4] Francisco Afonso, Carlos Silva, Adriano Tavares, and Sergio Montenegro. "Application-level fault tolerance in real-time embedded systems". In: *2008 International Symposium on Industrial Embedded Systems*. IEEE. 2008, pp. 126–133.

[5] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, et al. "MillWheel: fault-tolerant stream processing at internet scale". In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1033–1044.

[6] Amara D. Angelica. "Google's self-driving car gathers nearly 1 GB/sec". In: (May 4, 2013).

[7] *Apollo Cyber RT*. https://github.com/ApolloAuto/apollo/tree/master/cyber. Baidu.

[8] *Autonomous Vehicle Disengagement Reports*. https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/disengagement_report_2017. State of California Department of Motor Vehicles, 2017.

[9] Autoware. *Autoware User's Manual - Document Version 1.1*. https://github.com/CPFL/Autoware-Manuals/blob/master/en/Autoware_UsersManual_v1.1.md.

[10] Baidu. *Apollo 3.0 Software Architecture*. https://github.com/ApolloAuto/apollo/blob/master/docs/specs/Apollo_3.0_Software_Architecture.md.

[11] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. "Fault-tolerance in the Borealis Distributed Stream Processing System". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Baltimore, Maryland, 2005, pp. 13–24.

[12] State of California Department of Motor Vehicles. *Autonomous Vehicle Disengagement Reports 2017*. https://www.dmv.ca.gov/portal/dmv/detail/vr/autonomous/disengagement_report_2017.

[13] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. "Lightweight asynchronous snapshots for distributed dataflows". In: *arXiv preprint arXiv:1506.08603* (2015).

[14] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache flink: Stream and batch processing in a single engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[15] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. "Integrating scale out and fault tolerance in stream processing using operator state management". In: *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. ACM. 2013, pp. 725–736.

[16] K Mani Chandy and Leslie Lamport. "Distributed snapshots: Determining global states of distributed systems". In: *ACM Transactions on Computer Systems (TOCS)* 3.1 (1985), pp. 63–75.

[17] K. Mani Chandy and Jayadev Misra. “Distributed simulation: A case study in design and verification of distributed programs”. In: *IEEE Transactions on software engineering* 5 (1979), pp. 440–452.

[18] *Critical Reasons for Crashes Investigated in the National Motor Vehicle Crash Causation Survey*. https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812115. NATIONAL HIGHWAY TRAFFIC SAFETY ADMINISTRATION, 2015.

[19] B Dasarathy. “Timing constraints of real-time systems: Constructs for expressing them, methods of validating them”. In: *IEEE transactions on Software Engineering* 1 (1985), pp. 80–86.

[20] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[21] Patricia Derler, Thomas H Feng, Edward A Lee, Slobodan Matic, Hiren D Patel, Yang Zheo, and Jia Zou. *PTIDES: A programming model for distributed real-time embedded systems*. Tech. rep. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING and COMPUTER SCIENCE, 2008.

[22] John Eidson and Kang Lee. “IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems”. In: *Sensors for Industry Conference, 2002. 2nd ISA/IEEE*. Ieee. 2002, pp. 98–105.

[23] *Global status report on road safety 2015*. https://www.who.int/violence_injury_prevention/road_safety_status/2015/en/. World Health Organization.

[24] Jim Gray. “Why do computers stop and what can be done about it?” In: *Symposium on reliability in distributed software and database systems*. Los Angeles, CA, USA. 1986, pp. 3–12.

[25] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[26] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. “Queues don’t matter when you can JUMP them!” In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Oakland, California, USA, May 2015.

[27] Hars, Alexander. *Top misconceptions of autonomous cars and self-driving vehicles*. http://www.inventivio.com/innovationbriefs/2016-09/Top-misconceptions-of-self-driving-cars.pdf.

[28] Moritz Hoffmann, Frank McSherry, and Andrea Lattuada. “Latency-conscious dataflow reconfiguration.” In: *BeyondMR@ SIGMOD*. 2018, pp. 1–1.

[29] Fabian Hueske. *Introducing Stream Windows in Apache Flink*. https://flink.apache.org/news/2015/12/04/Introducing-windows.html. 2015.

[30] J-H Hwang, Magdalena Balazinska, Alex Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. “High-availability algorithms for distributed stream processing”. In: *21st International Conference on Data Engineering (ICDE’05)*. IEEE. 2005, pp. 779–790.

[31] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. “Dryad: distributed data-parallel programs from sequential building blocks”. In: *ACM SIGOPS operating systems review*. Vol. 41. 3. ACM. 2007, pp. 59–72.

[32] *JAMMED: How Much Time Money Does Traffic Congestion Waste?* https://www.autoinsurancecenter.com/traffic-jammed.htm. Auto Insurance Center.

[33] David R Jefferson. "Virtual time". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7.3 (1985), pp. 404–425.

[34] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, et al. "Caffe: Convolutional architecture for fast feature embedding". In: *Proceedings of the 22nd ACM international conference on Multimedia*. ACM. 2014, pp. 675–678.

[35] Svein Johannessen. "Time synchronization in a local area network". In: *IEEE control systems Magazine* 24.2 (2004), pp. 61–69.

[36] Evangelia Kalyvianaki, Themistoklis Charalambous, Marco Fiscato, and Peter Pietzuch. "Overload management in data stream processing systems with latency guarantees". In: *7th IEEE International Workshop on Feedback Computing (Feedback Computing'12)*. 2012.

[37] Andrea Lattuada, Frank McSherry, and Zaheer Chothia. "Faucet: a user-level, modular technique for flow control in dataflow engines". In: *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*. ACM. 2016, p. 2.

[38] Lee, Timothy B. *Waymo announces 7 million miles of testing, putting it far ahead of rivals*. `https://arstechnica.com/cars/2018/06/waymo-announces-7-million-miles-of-testing-putting-it-far-ahead-of-rivals/`.

[39] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. "The architectural implications of autonomous driving: Constraints and acceleration". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2018, pp. 751–766.

[40] Todd Litman. *Autonomous Vehicle Implementation Predictions Implications for Transport Planning*. `https://www.vtpi.org/avip.pdf`. Victoria Transport Policy Institute, 2019.

[41] Björn Lohrmann, Peter Janacik, and Odej Kao. "Elastic stream processing with latency guarantees". In: *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE. 2015, pp. 399–410.

[42] Björn Lohrmann, Daniel Warneke, and Odej Kao. "Nephele streaming: stream processing under QoS constraints at scale". In: *Cluster computing* 17.1 (2014), pp. 61–78.

[43] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. "Exploring the performance of ROS2". In: *Proceedings of the 13th International Conference on Embedded Software*. ACM. 2016, p. 5.

[44] Daniel V McGehee, Elizabeth N Mazzae, and GH Scott Baldwin. "Driver reaction time in crash avoidance research: validation of a driving simulator study on a test track". In: *Proceedings of the human factors and ergonomics society annual meeting*. Vol. 44. 20. SAGE Publications Sage CA: Los Angeles, CA. 2000, pp. 3–320.

[45] David L Mills. "A brief history of NTP time: Memoirs of an Internet timekeeper". In: *ACM SIGCOMM Computer Communication Review* 33.2 (2003), pp. 9–21.

[46] Aloysius Ka-Lau Mok. "Fundamental design problems of distributed systems for the hard-real-time environment". PhD thesis. Massachusetts Institute of Technology, 1983.

[47] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, et al. "Ray: A distributed framework for emerging {AI} applications". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 561–577.

[48] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martın Abadi. "Naiad: a timely dataflow system". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 439–455.

[49] Allen Newell and Stuart K Card. "The prospects for psychological science in human-computer interaction". In: *Human-computer interaction* 1.3 (1985), pp. 209–242.

[50] "Preliminary Report Highway: HWY18FH011". In: *National Transportation Safety Board* ().

[51] "Preliminary Report Highway: HWY18MH010". In: *National Transportation Safety Board* (May 24, 2018).

[52] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[53] Taylor Raack. *Autonomous Vehicle Technology: System Integration with ROS*. https://taylor.raack.info/2018/08/autonomous-vehicle-technology-system-integration-with-ros/. 2018.

[54] *ROS 2*. https://index.ros.org/doc/ros2/.

[55] David Schrank, Bill Eisele, Tim Lomax, and Jim Bak. "2015 urban mobility scorecard". In: (2015).

[56] "Self-driving Safety Report 2018". In: (2018).

[57] Mehul A Shah, Joseph M Hellerstein, and Eric Brewer. "Highly available, fault-tolerant, parallel dataflows". In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM. 2004, pp. 827–838.

[58] Dave Stewart. *Introduction to Real Time*. https://www.embedded.com/electronics-blogs/beginner-s-corner/4023859/Introduction-to-Real-Time.

[59] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. "Staying fit: Efficient load shedding techniques for distributed stream processing". In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment. 2007, pp. 159–170.

[60] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. "Load shedding in a data stream manager". In: *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment. 2003, pp. 309–320.

[61] B Taylor. "Introducing real time constraints into requirements and high level design of operating systems". In: *Proc. 1980 Nat. Telecommunications Conf.* Vol. 1. 1980, pp. 18–5.

[62] *The Economic and Societal Impact Of Motor Vehicle Crashes, 2010 (Revised)*. https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812013. NATIONAL HIGHWAY TRAFFIC SAFETY ADMINISTRATION, 2015.

[63] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, et al. "Storm@ twitter". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 147–156.

[64] Udacity. *An Open Source Self-Driving Car*. https://www.udacity.com/self-driving-car.

[65] Nicolò Valigi. "Lessons learned building a self-driving car on ROS". In: (Sept. 29, 2018).

[66] *Waymo Safety Report: On the Road to Fully Self-Driving*. https://storage.googleapis.com/sdc-prod/v1/safety-report/SafetyReport2018.pdf. Waymo, 2018.

[67] Yingjun Wu and Kian-Lee Tan. "ChronoStream: Elastic stateful stream computation in the cloud". In: *2015 IEEE 31st International Conference on Data Engineering*. IEEE. 2015, pp. 723–734.

[68] Akihiro Yamaguchi, Yousuke Watanabe, Kenya Sato, Yukikazu Nakamoto, Yoshiharu Ishikawa, Shinya Honda, and Hiroaki Takada. "In-vehicle distributed time-critical data stream management system for advanced driver assistance". In: *Journal of Information Processing* 25 (2017), pp. 107–120.

[69] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95.

[70] Yang Zhao, Jie Liu, and Edward A Lee. "A programming model for time-synchronized distributed real-time systems". In: *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*. IEEE. 2007, pp. 259–268.