OODIDA: On-board/Off-board Distributed Data Analytics for Connected Vehicles

Gregor Ulm
Fraunhofer-Chalmers Research Centre for Industrial Mathematics
Gothenburg, Sweden
gregor.ulm@fcc.chalmers.se

Emil Gustavsson
Fraunhofer-Chalmers Research Centre for Industrial Mathematics
Gothenburg, Sweden
emil.gustavsson@fcc.chalmers.se

Mats Jirstrand
Fraunhofer-Chalmers Research Centre for Industrial Mathematics
Gothenburg, Sweden
mats.jirstrand@fcc.chalmers.se

Abstract
Connected vehicles may produce gigabytes of data per hour, which makes centralized data processing impractical at the fleet level. In addition, there are the problems of distributing tasks to edge devices and processing them efficiently. Our solution to this problem is OODIDA (On-board/off-board Distributed Data Analytics), which is a platform that tackles both task distribution to connected vehicles as well as concurrent execution of large-scale tasks on arbitrary subsets of clients. Its message-passing infrastructure has been implemented in Erlang/OTP, while the end points are language-agnostic. OODIDA is highly scalable and able to process a significant volume of data on resource-constrained clients.

CCS Concepts • Computer systems organization → Cloud computing;

Keywords Distributed systems, Concurrent computing, Erlang

ACM Reference Format:
Gregor Ulm, Emil Gustavsson, and Mats Jirstrand. 2019. OODIDA: On-board/Off-board Distributed Data Analytics for Connected Vehicles. In Proceedings of (XX ’19). ACM, New York, NY, USA, 13 pages.

1 Introduction
Big data in the automotive industry is of increasing concern, considering that connected vehicles may produce several gigabytes of data per hour. Centrally processing this deluge of data at the fleet level is impractical. However, with OODIDA (On-board/Off-board Distributed Data Analytics), which is a platform that facilitates the distribution and concurrent execution of data analytics tasks in a heterogeneous system, we can process a substantial amount of data. This is a major step towards facilitating efficient big data analytics in vehicular networks. The majority of the computational work is carried out on client devices (on-board), so-called edge devices, and only a supplementary part on the server (off-board). OODIDA is a distributed system that connects analysts with a large number of on-board units (OBUs). In principle, it could be used for general distributed data processing tasks. However, it has been designed for the automotive domain and is intended for data exploration and rapid prototyping on a fleet of reference vehicles. A schematic overview of OODIDA is provided in Fig. 1: m users interact with a central cloud application b (bridge) that connects them to n clients, which are OBUs. The main feature of our system is distributing and concurrently processing decentralized data analytics tasks of arbitrarily duration. We provide a JSON-based language-agnostic interface for user and client applications. In addition, error handling mechanisms address the issue of intermittent availability of client devices.

This paper presents the architecture of OODIDA as well as use cases that illustrate how this framework solves real-world problems. We do not present a fully fledged-out solution, partly for the sake of brevity, and partly due to legal agreements that prevent us from publicly sharing source code artifacts. In summary, the contributions of this paper are:

- Describing the design and architecture of OODIDA
- Illustrating its use for concurrent distributed tasks
- Describing its assignment specification format
- Describing real-world use-cases
- Evaluating the performance of the system
The remainder of this paper starts off with some relevant background in Section 2. Subsequently, we provide a detailed description of OODIDA in Section 3, which includes a gradual presentation of our framework, moving from a very simple example all the way to fully distributed concurrent execution of multiple assignments. We substantiate the practical usefulness of OODIDA with an empirical evaluation in Section 4, before we conclude with a brief overview of related work in Section 5 as well as potential future work in Section 6.

2 Background
In this section, we provide some background on automotive big data. First, we summarize challenges commonly associated with big data, covering volume, velocity, and variety of data but also the issue of data privacy (2.1). Second, we look at ways of taming big data, covering real-world use cases in our domain (2.2). Third, we highlight scalability, based on transparent distribution and concurrency on clients and the server, as a key to solving automotive big data problems at the fleet level (2.3). Fourth, we clarify some relevant terms (2.4).

2.1 Big Data Challenges
The most immediate issue with data generated by connected vehicles is its immense volume and velocity, which could exceed 20 gigabytes per hour and vehicle [3]. It is impractical to transfer such volumes of data to a central server for processing, particularly at the fleet level. A related problem is that information may be time-critical, which implies that transferring data to a central server for subsequent processing is not viable. Consequently, processing data close to its source is a necessity for real-time analytics. Of course, there may be a need to perform subsequent centralized data processing of aggregated results. There is also the issue of the variety of big data, which is due to collecting a multitude of signals. Thus, the heterogeneity of data rules out a one-size-fits-all approach. An additional challenge, but one that is externally imposed, is data privacy [1, 20]. A recent example is the General Data Protection Regulation (GDPR) of the European Union [6], which places heavy restrictions on data handling. We will not discuss data privacy further, but merely point out that on-board processing sidesteps data privacy issues have to be addressed with central data collection.

2.2 Taming Big Data
The primary motivation of OODIDA is to make automotive big data easier to handle by processing it closer to its source, with the goal of limiting the amount of data that needs to be processed on a central server. To illustrate this, we present categories of use cases that occur in practice. Note that it would be desirable to perform tasks in those categories concurrently and with increasing numbers of vehicles.

We start with use cases where the central server only collects results from clients, without further off-board processing. A straightforward example is filtering on the client, e.g., monitoring or anomaly detection. We only keep values that fulfill a given predicate, which may allow us to discard most data. Another common approach in this vein is sampling, where a randomly chosen subset of data is retained. However, such tasks may require additional processing. A related example is MapReduce [4]. Consider the standard MapReduce problem of determining word frequency: clients could be tasked with first mapping each word w of the input to the tuple (w, 1). Afterwards, clients reduce all those tuples to their respective count c, i.e., (w, c). This is the data that is sent to the server, which, in turn, reduces all incoming data to the final value (w, c'), where c' specifies the total count of w in the entirety of the data that has been processed. While large-scale MapReduce is hardly the goal of OODIDA, there are use cases that fit this paradigm very well (cf. Section 3.5).

A more complex use case is federated learning [13], which is an example of distributed machine learning. In this case, the server maintains a global model, which is sent to clients. Each client trains the model with available local data and subsequently sends the updated model to the server, which computes an average of the received updated local models. In practice, there is likely a limited number of iterations, based on the total error of the global model. Once it falls below a specified value \(\epsilon\), training is considered complete. Subsequently, the final global model is sent to the user. Federated learning is an active area of research, with a particular interest from the mathematical optimization community. Examples include work on general distributed optimization [10] and the alternating direction of multipliers (ADMM) [11]. Large-scale optimization on edge devices using deep learning has also been shown as feasible [7].

Data processing tasks could be one-off with a fixed duration or they may run indefinitely long until a particular event has been detected. For some scenarios, it makes sense to repeatedly perform narrowly defined tasks, for instance getting a status update every \(n\) seconds, which can be modeled as an assignment with multiple iterations. Federated learning is merely a more complicated example of this approach as the results of iteration \(i\) are used as input for iteration \(i + 1\). Another useful addition is the emulation of stream processing by iteratively processing batches of data. The shorter the iterations are, the closer we get to pseudo-real time stream processing. A practical use case is updating records once per iteration, which could be used for creating incremental historical records.

2.3 Scalability at the Fleet Level
In order to handle big data problems at the fleet level, we need an effective means for task distribution. Of course, we also have to have the ability to issue multiple assignments to overlapping subsets of clients, which necessitates that client
devices are able to concurrently execute tasks. Meanwhile, the central server has to remain responsive even as the workload on the system increases. That being said, there is a clear limitation to the amount of work the system would need to perform as we are not targeting a large fleet of production cars. Instead, the goal is to execute an analytics platform on a private cloud that connects to OBUs in test vehicles, so-called reference vehicle computational nodes (RVCNs). This enables rapid prototyping of data analytics methods which may eventually be executed on OBUs in production vehicles.

2.4 Some Terminology

As this paper does not exclusively address a computer science audience, we would like to clarify a few relevant terms. The actor model is a mathematical model for describing concurrent computations, developed by Hewitt [9]. In it, independent actors send and receive messages, which makes it straightforward to model concurrent computations. This advantage also transfers to programming languages that are based on it. The most prominent language in that niche is Erlang, which is the language in which we have implemented the message-passing infrastructure of OODIDA. There is some conceptual confusion surrounding the terms concurrent and parallel, however. While even computer scientists may use them interchangeably, we follow, for instance, Harper [8] and Marlow [12], who view concurrency as the simultaneous execution of nondeterministic computations and parallelism as the simultaneous execution of deterministic computations.

As we are not going to share Erlang source code artifacts, there is little need to explain syntactic details. That being said, we do frequently use the terms processes and process identifiers (PIDs). Erlang processes communicate with each other by exchanging messages. In order to address processes, PIDs are used. By knowing the PID of a process, another process is able to send messages to it.

3 Solution

In this section we describe our solution to distributed data analytics in detail, starting with an overview of the OODIDA platform (3.1). Afterwards, we look at execution scenarios (3.2). The main part is the discussion of the central cloud application (3.3). A brief note addresses error handling (3.4), while our discussion of use cases shows how real-world data analytics problems map to OODIDA (3.5). To round off our description, we cover the back-end on client devices (3.6).

3.1 Overview of the OODIDA Platform

OODIDA is a distributed system for task distribution and management for automotive big data problems for the purpose of rapid prototyping of data analytics methods in a fleet of test vehicles. In short, its purpose is to take as its input the specification of an assignment, divide it into tasks, and distribute those tasks to a user-specified number of client devices. Users normally specify assignments with the help of an external library that facilitates their creation and verification. On client devices, an external application takes over and performs analytics tasks. OODIDA is flexible enough to execute multiple applications on clients. For instance, tasks of one category could be executed in Python while a different category of tasks is performed in a more performant language such as Go. The number of users, clients, and tasks is arbitrary and only limited by the computational power of the hardware the system is executed on.

While Fig. 1 presents a top-level view, Fig. 2 reveals the underlying message-passing infrastructure of OODIDA. The latter shows one possible instance of it, where one user has issued two assignments whose tasks are processed concurrently by three clients. In Fig. 1, temporary handlers are not shown. Thus, cloud nodes b and b’ in 2 correspond to cloud node b in Fig. 1, while client nodes, e.g. x and its task handler x’ in Fig. 2 correspond to a client c in Fig. 1. The red nodes represent permanent handlers and the blue nodes temporary ones. The user node u interacts with an external Python library, referred to as the front-end application f in Fig. 1. The nodes f_i and u_i run on the workstation of user i. After receiving an assignment specification from a user, the central cloud application b, which runs on an internal private cloud system, spawns an assignment handler that divides the assignment into tasks and distributes them to clients. For instance, one client device executes client node x, which spawns the task handler x’ on the same device. That task handler communicates the task specification to the corresponding external application a_x that is also executed on the client device. An example of such a task is running a multivariate regression analysis on a given number of samples of values read from a set of sensors.
After the external client application has finished its task, which could be one iteration of a task consisting of multiple iterations, the corresponding task handler receives the results and forwards it from the client device to its assignment handler $b$, which is executed in the cloud. Afterwards, the task handler terminates. Assuming the current assignment consists of only one iteration, the assignment handler awaits results from all members of the chosen subset of clients. The next step consists of node $b'$ aggregating those results, with optional off-board processing, and sending them to the node $b$. With its job being done, the assignment handler also terminates. Lastly, $b$ sends the results to the user node $u$, which, in turn, makes it accessible to the corresponding user application $f_i$. By default results are turned into a Python object that is accessible for further processing.

### 3.2 Execution Scenarios

The overview initially presented in Fig. 2 may look overwhelming, so we will gradually describe various usage scenarios. We do not focus on specific assignments in this subsection, but on their distribution and execution instead. We will look at three scenarios: one finite task, multiple finite tasks, and multiple indefinitely long tasks. For simplicity, we consider only one user and three client devices, although there could be arbitrarily many of both. Similarly, OODIDA does not restrict the number of concurrently executed assignments.

The most basic scenario consists of submitting one finite task to clients, either (a) all of them or (b) a subset thereof, which are shown in Figs. 3a and 3b, respectively. In Fig. 3a, the cloud process $b$ has spawned an assignment handler $b'$, which divides the assignment into tasks and sends a corresponding message to each of the client nodes $x$, $y$, and $z$. Each client node runs on a separate OBU. Upon receiving a task description from $b'$, the client process spawns a task handler, e.g., client $x$ spawns task handler $x'$. An external process takes over at that point. Upon completion of that task, each task handler sends the result to $b'$, which performs optional off-board computations. Afterwards, results are sent to the cloud process $b$ and, finally, to the user process $u$. Figure 3b is a variation that stresses a key aspect of OODIDA: Instead of broadcasting assignments to all clients, we only send targeted information, which may temporarily isolate some clients. There is no task handler associated with node $y$ in that case because they only spawn if there is a concrete task to perform.

Executing just one finite task at a time on some or all of the available clients has some practical utility, but in a real-world setting as well as due to the fact that on-board computers already have multiple CPU cores, there are many use cases that necessitate concurrent execution of multiple assignments. For instance, an OBU that is training a machine learning model with local data could concurrently monitor a range of sensors for critical values for the purpose of performing anomaly detection tasks. In Fig. 4 we see two cases of two assignments being concurrently executed per client. In both, the cloud process has spawned two instances of the assignment handler $b'$, i.e., one handler per assignment. In Fig. 4a, each of the three client processes has spawned one task handler per received task, which means that all client nodes are processing both tasks. In comparison, in Fig. 4b two tasks are assigned to overlapping subsets of the set of available clients. Client $x$ only executes task 1, client $z$ only executes task 2, and client $y$ concurrently executes both task 1 and 2. We skip the trivial case of assigning multiple assignments to non-overlapping subsets.

Lastly, indefinitely long assignments do not warrant separate diagrams. They are a variation as tasks could as well run for an unspecified duration. This is the case when execution is not tied to a fixed number of iterations but to a stopping criterion, such as reaching a desired threshold value for the validation error of an artificial neural network. Tasks can also be set up to run until an interrupt has been received, which is issued by the user. It is also possible to set up an
OODIDA: On-board/Off-board Distributed Data Analytics

Figure 4. OODIDA is able to concurrently execute an arbitrary number of assignments. In (a), the cloud node $b$ spawned two assignment handlers $c'$, which both communicate to all clients $x$, $y$, and $z$, while in (b) two assignments are handled by overlapping subsets of clients. Red nodes indicate permanent processes, while blue nodes indicate that these processes are only temporary. Clients spawn task handlers ($x'$, $y'$, $z'$); node $u$ stands for the user.

The latter mode of operation generates a stream of data as results are processed as part of each iteration.

3.3 Implementation

OODIDA consists of one central cloud node $b$ as well as $m$ user nodes and $n$ client nodes. The central node thus provides a connection between the user, i.e. a data analyst specifying assignments on one end and a multitude of client devices on the other. The underlying message passing infrastructure has been implemented in the actor-model-based functional programming language Erlang. Our presentation will mainly focus on this system. We address the external applications interacting with the peripheral nodes on user workstations only in passing, but provide some details on the client backend (cf. Section 3.6). In a production system, each OBU is supposed to execute only one instance of the client process. This limit is not enforced, which facilitates an alternative operational mode, i.e. simulating a large number of client devices on a workstation. In the following, we focus on an instance with one user for the sake of brevity, although our system can accommodate an arbitrary number of users.

Main loop. On user workstations, a front-end $f$ is available that interfaces with a Python library, which generates and validates assignment specifications in the JSON format. The data analyst can optionally use a Python notebook, which provides a greater degree of interactivity. Once an assignment specification has been generated, it is processed via the user process $u$ and sent via the network to $b$, which runs on an internal private cloud. There, the execution of a single assignment resembles the following main loop:

1. Await assignment specification
2. Read assignment specification
3. Spawn assignment handler
4. Divide assignment into tasks
5. Distribute tasks to chosen subset of client devices
6. Monitor completion of tasks on client devices (completion entails that clients send their results to the cloud)
7. Upon completion, aggregate results of client devices
8. Perform optional off-board processing
9. Send assignment results to user process
10. Go to step 1

This loop is a stark simplification. In reality, the main server process is ready to process incoming assignment specifications (steps 1 to 3) or forward results from the assignment handler to the user (step 9), regardless of the status of any other active assignment as this is outsourced to assignment handlers, which take care of steps 4 to 8. Furthermore, the cloud process can spawn an arbitrary number of assignment handlers, without being blocked due to having to waiting for result.

OODIDA can also handle tasks that run for an unspecified amount of time, which can be manifested in multiple ways. For instance, an assignment could consist of one iteration, which terminates when a particular stopping criterion is met. Alternatively, it is possible to define an assignment consisting of $k$ rounds. These repeat steps 4 to 8 $k$ times. The end point may be left unspecified, in which case the assignment runs until a stopping criterion is met, e.g. an error rate below a set value $\epsilon$ in a federated learning task, or until the assignment is stopped through user intervention.

Assignment specification. Assignments are JSON objects which contain key-value pairs. Among others, they indicate
the chosen on-board and off-board computations and the signals to process. Depending on the chosen computations there may be contextual parameters to set. The assignment specification also indicates duration and whether intermediate results are required. The latter is possible by setting a number of iterations. Of course, we also need to choose clients. There is a separate keyword for whole-fleet assignments. For sub-fleet assignments, the user can either choose a number of clients or a list of client IDs.

**User node (incoming).** Every instance of the user node $u$ has two main components, a process user that communicates with node $b$ as well as a process await that primarily reacts to input sent from the front-end $f$, which could be assignment specifications or status requests. We ignore the latter in our presentation, however. When an assignment specification arrives, await decodes it, extracts all relevant information, turns it into an Erlang tuple tagged as assignment, and sends it to user for further processing. Having a separate process await, which is stuck in a block on receive until information arrives, allows user to remain responsive throughout, which is relevant later, when we discuss how incoming results are handled.

**Cloud node (incoming).** The user process forwards the relevant content from an assignment specification to the Cloud process. Its state contains a list of currently active clients, pairing their IDs and PIDs. The goal is to send task specifications to each of the designated clients for the given assignment. However, we cannot block the execution of Cloud until we have received a result because the user may want to issue another assignment in the meantime, which, of course, has to be concurrently executed. This problem is tackled as follows: The Cloud process awaits assignments from the user process. If there is currently no client process available, which is the case when no vehicles have been registered in the system, the assignment is dropped (and the data analyst is informed of that outcome). Otherwise, the Cloud process spawns an assignment_handler process for the current assignment that sends task specifications to clients and monitors their completion.

The assignment_handler is a key component for enabling concurrent assignments on the OODIDA platform. It receives as arguments the list of currently active clients, the subset of clients specified by the assignment, the assignment configuration, and the PID of the Cloud process. It first determines the PIDs of the clients that were specified in the assignment configuration and afterwards sends a specification of the on-board task that has to be performed to each client. The next step is to block until we have received results from all clients (error handling is covered in Section 3.4), which is possible due to the cloud process spawning a new assignment handler process for each incoming assignment. We will continue at this point after discussing the client node.

**Client node (incoming and outgoing).** The client node waits for task descriptions to arrive. There is one client process per client device, which is active when the client device is available. Whenever this process receives a tuple that is tagged as a task, a task specification is extracted and turned into a JSON data type. This is followed by spawning a task_handler process, which communicates with an external application that has to complete the specified task. The task handler awaits completion. For each new incoming task, a new task handler is spawned. This mirrors spawning a separate assignment handler for each new assignment on the cloud node and it likewise ensures that the client remains responsive. The initial objective of the task handler is to forward the task specification to an external application. It is possible to have different applications process different tasks, but we will not discuss this further and instead only consider the case where there is one external application on the client that performs the specified task. Most tasks consist of reading values from a set of sensors for a given interval and performing a computation on it. The result of this computation is sent to task_handler.

We have implemented the client application in Python as well as Go, which both facilitate concurrency. Thus, clients can process multiple tasks concurrently. We previously mentioned that an assignment may consist of multiple iterations. However, this is completely hidden from the client as it treats each iteration in isolation. In other words, the client is both oblivious to whether it performs an assignment with multiple iterations or just one and also which iteration it is currently working on. Furthermore, the task_handler process does not communicate with the client process at all. Instead, it directly sends results to the assignment_handler of the current task. This reduces the communication and computation overhead of the client process even further. After sending the final result of the task to the assignment handler, the task_handler process terminates.

**Cloud node (outgoing).** Once the assignment handler has received all results from the task handlers, it may perform off-board computations. If the current assignment consists of multiple iterations, the assignment handler will call itself recursively, which leads to a repetition of sending out tasks to clients and waiting for their results, which are tagged with the PID of the originating process. On the other hand, if the current assignment has no iterations remaining, the final result is sent to the process Cloud. Afterwards, the assignment handler terminates. This is the end of a cascade effect: first, all task handlers terminated after the results of the respective on-board tasks they monitored have completed, which is followed by the assignment handler terminating after completing the final off-board computation. The last step of the

---

1Client IDs are static, while PIDs may change when a client reconnects. The user addresses clients with their IDs, not their PIDs.
process cloud is to send the results of the assignment to the user node it originated from.

**User node (incoming).** On the user node, the default approach is that the process user sends a JSON object containing the result of the issued assignment to the front-end application \( f \), which is most commonly an instance of a Python notebook. The data analyst interacting with it will receive a notification that the desired results have arrived. These can be further manipulated in that application, for instance by integrating them into computational workflows where the results of one assignment are used as input for the next; we will see an example in Section 3.5. Alternatively, results can be dumped to a file and manipulated in a third-party application for further statistical processing.

### 3.4 Error Handling

So far, we have made the assumption that all assignments complete and that all client nodes are available all the time. Of course, it is overly optimistic to assume that the network is always stable. To account for real-world disturbances, we use various Erlang/OTP mechanisms for fault-tolerance such as links and monitors. If the connection to a client has been lost, it is given a set duration of time to reconnect. As long as a disconnected client is alive, it will actively try to reconnect with the cloud node. Should this fail, that client’s results will be excluded from the current iteration of the assignment. For all remaining iterations of the assignment, OODIDA will attempt to send a task description also to clients that were lost during previous iterations, which means that whenever a client becomes available again, it will contribute to the current iteration of that assignment.

### 3.5 Practical Use Cases

In the following, we will look at a few use cases that illustrate how OODIDA can be used for practical data analytics tasks. We will cover anomaly detection and random sampling as examples of relatively straightforward tasks. More complex use cases involve MapReduce-like computations and federated learning. Lastly, we show how a data analyst interacting with our system can define control flows in which results of an assignment are used as input for subsequent ones.

**Sampling.** One of the challenges of big data is its immense volume. A straightforward approach to handling it is to collect a random sample, which amounts to only a fraction of the original data. The assumption is that the random sample has properties similar to the original data. In order to do this with OODIDA, the data analyst needs to set up a sampling task with a list of signals and specify the amount of data in percent that should be collected on the client. This is the on-board task. A fitting corresponding off-board task is the collection of values, paired with the ID of the client they originated from.

**Anomaly detection.** The goal of anomaly detection is finding outliers in data. For instance, we can determine when a given data point is far outside the normal range the chosen machine learning model predicts. Concretely, assume we have a probability distribution of our data and want to detect values that are considered implausible. For this, a threshold \( \epsilon \) is specified. Thus, the data analyst selects a model, sets the desired parameters, and the list of signals to monitor. This defines the on-board task that is performed on the client. A fitting off-board task would be to collect the device IDs of the affected OBUs and their corresponding values.

**Predictive maintenance.** Using an existing machine learning model that is tied to a list of signals, it is possible to detect components that may be in need of maintenance. As a simplified example, consider a model that, based on the speed of the vehicle, vehicle weight, current precipitation, force of braking and road condition — this is a lookup based on current GPS coordinates of the vehicle — is able to identify if its braking efficiency deviates from expected values. A task like this is suitable for whole-fleet assignment. Normally components do not degrade rapidly, so it is sufficient to run a task like this in a set interval and not permanently. The corresponding off-board task is the collection of values from vehicles that deviate from the model.

**MapReduce-like tasks.** While running large-scale MapReduce jobs on a fleet of connected vehicles may sound outlandish, there are nonetheless use cases where a MapReduce-like approach is very helpful. For instance, consider driverspecific speed-distribution histograms. The corresponding on-board task consists of mapping velocity signal values to a bin value, resulting in \((bin_i, 1)\) and subsequently reducing these pairs to their sums \( (bin_i, \sum) \) as well as normalizing them. The fitting off-board task would either be a mere collection of those histograms, or averaging them to one histogram of the entire fleet.

**Federated learning.** A much more complex use case is federated learning, an approach to machine learning where client devices train a model on local data. One round consists of sending a global model to all clients. Afterwards, client devices train the model with local data and send their updated local model to a central computer, which averages the received local models to a new global model. As long as its error exceeds the provided threshold, this process is repeated. The assignment_handler process that is spawned by the cloud process initializes the model with arbitrary parameter values. The state of the assignment_handler furthermore contains a value \( \epsilon \), which is the desired maximal validation error of the machine learning model. For each round of the training process, the assignment_handler sends the current model to every client process. In turn, clients train the model with local data. Then, they send the updated local model to the assignment_handler. The subsequent task
of the assignment handler is to create a new global model, which is based on the updated local models that were received. Afterwards, the assignment handler evaluates the new global model with a validation data set. If the resulting error exceeds the desired value $\epsilon$, training continues by sending the new global model to all client processes, and so on. Otherwise, training has been completed and the model is sent to the user.

**Combined work flows.** So far, we have discussed isolated tasks. However, the user front-end also facilitates combined workflows, where the results of one assignment serve as the input for a subsequent one. For example, consider the case of predictive maintenance, for which the user issues a whole-fleet assignment. Some of the clients report that there is no issue, and others that there is potential need for maintenance, so the IDs of the latter are filtered out. We then take the set difference of the set of all active client IDs and the set of client IDs for which we have received results, and send out a sub-fleet assignment targeting the missing client IDs. This loop continues until we have received reports from all active vehicles.

### 3.6 Client Back-End

The external client application has been implemented in Python, but we also have a Go prototype as a proof-of-concept that implements a subset of the functionality of the Python client. These applications consume tasks specifications, perform computational work, and send its results as JSON objects to their task handler. In principle, OODIDA could be run with multiple applications on the client. An example of this is delegating stream processing tasks to an external stream processing engine like Apache Edgent.\(^2\) — we have implemented this as a call from the existing Python application to Edgent, but this could as well be handled with a stand-alone application. While the client, just like the front-end, are in principle language-agnostic, one requirement is that tasks can be executed concurrently. In Python, we use the multiprocessing library for that purpose, while in Go we use goroutines.

### 4 Evaluation

In order to evaluate OODIDA, we looked at a number of sample workloads that are representative of real-world use cases. Our experiments focus on the performance of client devices as well as system-wide performance. While we look at total throughput on clients as well, the primary focus is on quantifying the ability of the system to remain responsive while concurrently executing substantial workloads. Below, we describe our hardware and software environment (4.1), the experiments we conducted (4.2) as well as their results (4.3), and discuss them (4.4).

---

\(^2\)The homepage of the Apache Edgent project is [https://edgent.apache.org/](https://edgent.apache.org/).

### 4.1 Hardware and Software Setup

We used three quad-core PC workstations as well as Raspberry Pi client devices. Workstation 1 contains an Intel Core i7-6700k CPU (4.00GHz), workstation 2 and 3 an Intel Core i7-7700k CPU (4.2GHz). All three workstations are equipped with 32 GB RAM each, of which 23.5 GB were made available to the hosted Linux operating system. They run on Windows 10 Pro (build 1803), executing OODIDA on a hosted Linux Ubuntu 16.04 LTS in VirtualBox 6.0 (workstation 1) or VirtualBox 5.2 (workstations 2 and 3). For the client stress test, we used a Raspberry Pi 3 Model B with a quad-core Broadcom ARMv7 CPU (1.2 GHz) and 1 GB RAM. It uses the Linux distribution Raspbian 9 with kernel 4.14.52. The CPU of the Raspberry Pi is similar to the CPU of the RVCNs our industrial collaborators use. Our internal prototype runs on edge devices that have been connected wirelessly. However, for the purpose of establishing the upper bound for system performance, we used an Ethernet connection.

### 4.2 Experiments

OODIDA is designed to run in a heterogeneous system, where a powerful central server orchestrates computationaly bound edge devices. Our experiments explored the limits of (1) a single client device and (2) the entire system. For the first kind of experiments, the mode of connection to the system is irrelevant because the client does not communicate with the cloud process while performing its work. For the second kind, however, the difference between Ethernet and wireless connectivity affects the maximum throughput via the network. While Ethernet connectivity is not indicative of real-world usage of OODIDA, it nonetheless allows us to estimate the theoretical peak performance of our system. We used synthetic velocity data for our experiments that has been modeled after data that was generated in research vehicles. This puts fewer restrictions on us due to sidestepping issues of data anonymity and confidentiality. For the purpose of evaluating the performance of our system, synthetic data is just as useful as real-world data as the system is agnostic to properties of the data.

**Clients.** In order to determine the maximal sustainable workload of the chosen client hardware, we executed OODIDA with one user and the cloud process on workstation 1, while a Raspberry Pi device executed one client process. We measured the amount of velocity data that can be processed in a sampling task in five minutes. For simplicity, sensor data was read as fast as possible. Data was provided as either single- or double-precision floating-points numbers. In a separate experiment, we measured CPU and RAM utilization as the number of concurrently executed assignments increased. We used a lightweight sampling task, with a sampling rate of 1 Hz. The used version of the Python interpreter was 3.5.3; Go was compiled with version 1.11.3.
System. The goal of the whole-system experiments is to determine how OODIDA reacts to surges of activity, with a focus on how large concurrent workloads affect the response time of the cloud process. We looked at this issue from two sides: (1) incoming assignments from the user side as well as (2) incoming results from clients.

In (1) the user and cloud processes were executed on workstation 1. We issued a batch of 100 assignments and measured how the response time, which is the difference between assignment creation and processing, is affected as subsequent assignments are generated by an automated process. A surge of such a dimension is unrealistically large, so these results help to explore the theoretical limits of our system.

In (2), workstations 2 and 3 simulated 50 clients each, for a total of 100, which are tasked with indefinitely long assignments. In three different scenarios, they execute either 1, 5, or 10 tasks concurrently per client. We collected 100,000 processing times of a full task iteration per scenario. A full task iteration start when the assignment handler splits the assignment into tasks for clients and ends when it has collected the results of that tasks. The round-trip time between cloud and client workstations is included as well. In order to put more demand on the cloud process, we used dummy client processes in Erlang that produce a result immediately after receiving a task specification, leading to a sustained surge of incoming results. Using 100 clients may seem low, but this number is a multiple of the number of RVCNs OODIDA is targeting. Also, the number of clients is not nearly as relevant as the number of incoming results files. Running 10 tasks concurrently on one client is arguably excessive. Yet, the workload of 100 clients running ten tasks is numerically equivalent to, for instance, 500 clients concurrently executing two tasks. Of course, it is also the case that real-world tasks would not immediately produce a result. Thus, again, the stress we are putting our system under far exceeds real-world demands.

4.3 Results

Client. As the results of the throughput test in Fig. 5 show, there is a large gap between the performance of the Python client and the Go client. It amounts to one order of magnitude. The OODIDA client application in Go processes around 45.4 (47.9) million floats (doubles) in five minutes. The corresponding numbers for Python are virtually identical with 6.78 (6.77) million floats (doubles). These are the averages of five runs. The corresponding standard deviations are minimal and amount to 0.100 (0.099) in Go versus 0.031 (0.036) in Python. Extrapolating these measurements to one hour, we arrive at 2.18 (4.60) GB in Go versus 0.33 (0.65) GB in Python.

In addition, we determined RAM and CPU utilization with increasing numbers of concurrently processed assignments, the results of which are shown in Fig. 6. These plots start with zero assignments, which shows that the base utilization for the Python client is less than 1% CPU and 20.5 MB of RAM. With the Go client, base CPU utilization is likewise close to zero, while base RAM utilization amounts to 18.8 MB. From that point onward, RAM consumption increases linearly in Python, while CPU utilization fluctuates, but increases mostly linearly. Past around 250 assignments, the Python client became less responsive. At around 90% of RAM utilization, CPU utilization spiked due to memory swapping. CPU utilization is close to 20% at 300 concurrent tasks. In contrast, the Go client showed a very modest increase of RAM consumption to around 20 MB, while the CPU load increased a lot more slowly to around 10%.

System. The results of the evaluation of the system are shown in Fig. 7. The median and standard deviation of five runs in Fig. 7a indicates that congestion due to incoming assignment is a non-issue. Even when submitting an excessively large batch of 100 assignments, the difference between assignment creation by the user and assignment processing on the cloud is less than 40 ms on average, with a slight upward trend.

Incoming workload from clients and its effect on the response time of the cloud process is shown in Fig. 7b. Here we see that the system remains highly responsive even as the workload scales up dramatically. In all three scenarios the measurements approximate a log-normal distribution. As the number of incoming results increases, due a greater concurrent workload on clients, iteration completion times exhibit a modest deterioration. To single out the most extreme case: even with 1000 incoming results per iteration — 100 clients executing 10 tasks concurrently for 100 iterations — the 95th percentile is at only 20 ms.

4.4 Discussion

Client. The experiments that focused on determining the performance of client devices revealed a significant performance gap between Python and Go. This is easy to explain.

\[ We explored JIT compilation with Numba and PyPy. The former does not support the Python module os, which our client application requires. The 

![Figure 5. Throughput on the client in number of values in five minutes, based on single and double-precision floating point values. The Go implementation is around seven times faster. Extrapolated to one hour, these results show that we can process around 2.18 (4.60) GB vs. 0.33 (0.65) GB of floats (doubles).](image)
Figure 6. Concurrency stress test of the Python and Go clients, showing CPU and RAM utilization while increasing the number of concurrently executed sampling tasks at a sampling rate of 1 Hz. The plots show the mean and standard deviation of five runs. Over 250 tasks can be processed concurrently in Python before hitting hardware limitations. The spike in CPU utilization at 90% memory is due to memory swapping. In contrast, the Go client uses up 20% of the available RAM with the same workload. At 300 concurrent tasks, CPU utilization is 18% with Python and 10% with Go.

considering that Python is a slow interpreted programming language with a rudimentary type system, while Go is statically typed and compiled. Type information facilitates compiler optimizations, leading to more efficient binary code. We consider Go a more attractive development target for real-world use cases, and may at one point port the entire client application to that language. However, one intended feature of OODIDA, namely the desired ability to execute user-defined Python functions, necessitates a Python installation on the client.

The Python implementation exhibits a constant increase in RAM consumed that outpaces CPU utilization. New processes in the Python application are forks that do share some memory libraries. We also tried spawning processes, but this led to an even more dramatic increase in RAM utilization as they do not share any memory. (We also wanted to avoid programming with Python threads in general, ruling out those alternatives.) Despite this rather severe drawback, it is nonetheless the case that the amount of (simple) assignments that can be executed concurrently is quite high, far exceeding 200. Thus, even a device as resource-constrained as the ones we used are powerful enough for complex real-world scenarios. In fact, it seems implausible that someone would want to run such a high number of concurrent assignments on any one device.

The ongoing evolution of ARM-based CPUs will benefit OODIDA in the future as performance increases will come due to technological progress in that area. Also, it is possible to put multiple on-board units into one vehicle to increase the maximum throughput. The expectation is that this would enable linear scalability of the amount of data that can be processed, assuming the respective hardware units are comparable. While using two OBUs that execute the Python edge software application in one vehicle would only increase the maximal throughput of double-precision floating-point numbers to around 1.3 GB/hour, the comparable number with a Go-based system would be 9.2 GB/hour.

System. The main takeaway of our system tests is that OODIDA handles concurrent execution very well. We have found that congestion due to incoming assignments from users is a non-issue. Even when a batch of 100 assignments arrives at once, which is a figure that is far above real-world requirements, the cloud process keeps up very well. The slight upwards trend observed in the data as the number of assignments increases is practically irrelevant with a response time of around 50 milliseconds in the worst case.

When simulating the workload of an, as of yet, unrealistically large number of clients, the system remains highly responsive. The bottleneck will instead lie with client hardware and its ability to produce real-time results. On the other hand, the message-passing infrastructure of our system has been shown to already be able to handle unrealistically high workloads. Furthermore, the central cloud hardware is a regular workstation with a rather modest amount of CPU cores. Thus, we are quite far from reaching hardware limits on that front even if system demands increase by orders of magnitude. We would also like to reiterate that these results hold regardless of the work that is performed on the client as it was excluded from our measurements. Consequently, OODIDA is able to process its intended workload with ease.
Figure 7. The stress test of the cloud process showed that (a) the system can easily process even a surge of incoming assignments. Processing times, based on five runs, which are illustrated by mean and standard deviation, only marginally deteriorate. Furthermore, as seen in (b), OODIDA comfortably handles large numbers of concurrent assignments. The histograms show the processing times of 100,000 results per scenario, measured from assignment creation on the cloud until the reception of results from the client.

5 Related Work

The problem of using networked devices for data analytics has been tackled from different angles. Very well-known is Google’s MapReduce [4], which consists of using clusters of commodity PCs for batch-processing of large-scale finite tasks. A related implementation of MapReduce in Erlang/OTP and Python was described by Mundkur et. al [14]. With MapReduce, a master process keeps track of task completion. While the master can supervise multiple assignments, worker machines are only assigned one specific task along with a corresponding batch of data. In OODIDA, data is generated on the client. Clients furthermore concurrently execute a multitude of assignments, which could be based on data batches as well as data streams. Apart from those conceptual differences, MapReduce was designed to work with an entirely different kind of hardware in mind, namely commodity PCs in a data center instead of computationally-bound edge devices. However, due to its inherent flexibility, OODIDA can perform MapReduce-like tasks as well. This is detailed in Section 3.5.

Some of our use cases require the ability to process infinite data streams. Thus, our work could be seen as related to the many stream processing platforms that emerged as a response to limitations of MapReduce. Of course, the focus should not necessarily be on MapReduce, as there are decades of research behind stream processing, which Stephens summarizes [19]. One could use OODIDA to perform complex stream processing tasks, similar to use cases that are tackled in systems like Apache Storm [21] or Apache Spark [23]. A significant caveat would be that OODIDA is intended to orchestrate CPUs of rather limited computational power instead of data centers housing industry-grade server hardware. However, using OODIDA for stream processing would be a rather limiting use case. While one could write, for instance, an application for filtering values out of an infinite data stream on an edge device, it would be more practical to call an existing stream processing engine on edge devices, but preferably one that has been designed with limitations of that kind of hardware in mind, such as Apache Edgent.

More generally, there is the field of distributed data processing, which has a long history. An early discussion piece was contributed by Enslow in the 1970s [5]. For a later overview article, see Scherr [18]. On a superficial level, this could be seen as highly related to OODIDA. However, one of the main motivations behind distributed data processing, which is also the case for MapReduce and related systems, is increased reliability and availability. Our focus is on efficiently making use of available hardware resources that would otherwise remain largely unused, and data that would otherwise largely remain unprocessed. We take into account that not all vehicles in a fleet will be available all the time, but if they are, their on-board units can be utilized.

OODIDA emerged from ff1-erl, a framework for federated learning in a functional programming language [22]. The main conclusion of that project was the general feasibility of using Erlang/OTP for large-scale task distribution. However, it also confirmed that it is not prudent to rely entirely on Erlang/OTP as the performance for numerical computations may not be favorable. In addition, there are practical limitations, such as the limited number of skilled Erlang/OTP programmers on the labor market or the fact that extensive machine learning libraries exist in other programming languages, such as the Python libraries scikit-learn [15].
and Keras [2]. Both aspects, but in particular the latter one, point to the importance of focusing on interoperability and the resulting benefits due to making the user front-end as well as the client back-end of OODIDA language-agnostic.

6 Future Work

OODIDA is a fully functional prototype that is both able to simulate clients in software as well as interface with edge devices that are connected either wirelessly or via Ethernet. In order to provide a more fully fledged out solution for the automotive industry, we intend to implement further distributed data analytics algorithms or, if an third-party solution exists, provide access to it through our cloud and client applications. Given that one focus of the OODIDA project is rapid prototyping, we have also been working on fully automating deployment of the system to clients via the network. Ease of use is also one of our concerns. We have addressed this by providing a library for the data analyst that can be used with a Python notebook, which is particularly useful for creating combined workflows. However, it may be worth creating a more user-friendly graphical interface.

OODIDA already scales very well as the number of clients increases. However, when deployed on real hardware, there are situations where the performance bottleneck of Python is an issue. Thus, we intend to fully port the client application to Go, with the exception of an experimental feature for rapid prototyping, namely the ability of the user to define and execute custom Python code on the cloud as well as clients without having to restart any part of the system. This is relevant for exploratory experiments as it sidesteps the time-consuming and disruptive deployment process. We intend to describe this feature as well as the front-end application in an experience report.

Security is a potential weakness of Erlang/OTP and an area that deserves paying close attention to [16]. As our system is deployed on a private cloud as well as clients that are located in reference vehicles, this is less of an issue for us. That being said, we are well-aware of that issue and intend to devote resources to code hardening. On that note, we also intend to makes further use of the rich Erlang/OTP ecosystem. While we already use Dialyzer [17] for static analysis, we intend to also use the type checker Gradualizer, which would be a good match since we already use type specification in our entire code base.

Acknowledgments

This research was supported by the project On-board/Off-board Distributed Data Analytics (OODIDA) in the funding program FFI: Strategic Vehicle Research and Innovation (DNR 2016-04260), which is administered by VINNOVA, the Swedish Government Agency for Innovation Systems. We furthermore acknowledge the help of Adrian Nilsson and Simon Smith who conducted the evaluation of OODIDA.

References

[1] Deyan Chen and Hong Zhao. 2012. Data Security and Privacy Protection Issues in Cloud Computing. In Computer Science and Electronics Engineering (ICCEE), 2012 International Conference on, Vol. 1. IEEE, 647–651.
[2] François Chollet et al. 2015. Keras: Deep Learning Library for Theano and Tensorflow. URL: https://keras.io/.
[3] Riccardo Coppola and Maurizio Morisio. 2016. Connected Car: Technologies, Issues, Future Trends. ACM Computing Surveys (CSUR) 49, 3 (2016), 46.
[4] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM 51, 1 (2008), 107–113.
[5] Philip Harrison Enslow. 1978. What is a 'Distributed' Data Processing System? Computer 11, 1 (1978), 13–21.
[6] Regulation (EU). 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). Official Journal of the European Union L119 (2016), 1–88.
[7] Corentin Hardy, Erwan Le Merrer, and Bruno Sericola. 2017. Distributed Deep Learning on Edge-Devices: Feasibility via Adaptive Compression. arXiv preprint arXiv:1702.04683 (2017).
[8] Robert Harper. 2016. Practical Foundations for Programming Languages. Cambridge University Press.
[9] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. Session 8 Formalisms for Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence. In Advance Papers of the Conference, Vol. 3. Stanford Research Institute, 235.
[10] Jakub Konecný, H Brendan McMahan, Daniel Ramage, and Peter Richtárik. 2016. Federated Optimization: Distributed Machine Learning on Device Intelligence. arXiv preprint arXiv:1610.02327 (2016).
[11] Qunwei Li, Bhavya Kailkhura, Ryan Goldhahn, Priyadip Ray, and Pramod K Varshney. 2017. Robust Federated Learning Using ADMM in the Presence of Data Falsifying Byzantines. arXiv preprint arXiv:1710.05241 (2017).
[12] Simon Marlow. 2012. Parallel and Concurrent Programming in Haskell. In Central European Functional Programming School. Springer, 339–401.
[13] H Brendan McMahan, Eider Moore, Daniel Ramage, and Blaise Agueray Arcas. 2016. Federated Learning of Deep Networks Using Model Averaging. (2016).
[14] Prashanth Mundkur, Ville Tuulos, and Jared Flatow. 2011. Disco: a Computing Platform for Large-Scale Data Analytics. In Proceedings of the 10th ACM SIGPLAN workshop on Erlang. ACM, 84–89.
[15] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research 12, 10 (2011), 2825–2830.
[16] Alexandre Jorge Barbosa Rodrigues and Viktória Fordós. 2018. Towards Secure Erlang Systems. In Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang. ACM, 67–70.
[17] Konstantinos Sagonas. 2005. Experience from Developing the Dialyzer: A Static Analysis Tool Detecting Defects in Erlang Applications. In Proceedings of the ACM SIGPLAN Workshop on the Evaluation of Software Defect Detection Tools.
[18] Allan L. Scherr. 1999. Distributed Data Processing. IBM Systems Journal 38, 2/3 (1999), 354.

Gradualizer is available at: https://github.com/josefs/Gradualizer.
[19] Robert Stephens. 1997. A Survey of Stream Processing. *Acta Informatica* 34, 7 (1997), 491–541.

[20] Omer Tene and Jules Polonetsky. 2011. Privacy in the Age of Big Data: a Time for Big Decisions. *Stan. L. Rev. Online* 64 (2011), 63–69.

[21] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 147–156.

[22] Gregor Ulm, Emil Gustavsson, and Mats Jirstrand. 2018. Functional Federated Learning in Erlang (ffl-erl). *CoRR* abs/1808.08143 (2018). arXiv:1808.08143 http://arxiv.org/abs/1808.08143

[23] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing With Working Sets. *HotCloud* 10, 10-10 (2010), 95.