Simulation as a Service for Cooperative Vehicles

Jörg Christian Kirchhof, Evgeny Kusmenko, Bernhard Rumpe, Hengwen Zhang
Chair of Software Engineering, RWTH Aachen University, Aachen, Germany, {kirchhof,kusmenko}@se-rwth.de

***Abstract*—Simulating connected vehicles in realistic environments is a computationally expensive task, particularly when large numbers of traffic participants are involved. To cope with exploding hardware requirements it can become indispensable to tackle such simulations in a divide and conquer manner. A promising approach breaking down a simulation scenario in a set of small tasks is the spatial subdivision of the simulated area. Thereby, each spatial sector is simulated by a dedicated sub-simulator enabling efficient distribution of the overall simulation across multiple machines. However, a distributable and easy-to-use simulation solution providing automated scalability and hiding the distribution complexity from the simulation engineer requires a service-based simulator architecture equipped with interfaces for simulation control, integration, and resource management. In this work, building upon previous results we propose a cloud-ready simulation infrastructure providing automotive engineers with the ability to analyze large scale connected traffic scenarios without caring about the execution environment. As a consequence, the proposed solution can be integrated seamlessly into existing continuous integration environments supporting agile research and development processes for cooperatively interacting vehicles.**

***Index Terms*—simulation as a service, cooperative vehicles, model-driven engineering, testing**

I. Introduction

In both research and development of Cyber-Physical Systems (CPSs), particularly of autonomous vehicles, simulation has always been an indispensable tool to validate a system's behavior before it can be deployed in a real-world scenario. In the last years, we have been experiencing a continuous vehicle automation process leading to an increasingly rapid softwarization of vehicle systems. Consequently, vehicle behavior becomes more and more difficult to grasp. The more intelligence finds its way into our transportation systems, the more important simulation tasks become in the development process. Autonomous vehicle functions need to be tested, and hence simulated, in a vast range of driving scenarios and under an inexhaustible number of environmental conditions [1]. And if that wasn't enough, the next logical step is to leverage autonomous vehicles to interactively cooperating agents redefining our traffic systems as highly-reactive, self-optimizing cyber-physical networks. The implication is obvious: automated driving behavior seizes to be an isolated function provided by the host vehicle and its sensor signals, but suddenly depends on collaborative decisions. This, in turn, imposes new kinds of requirements towards simulation technology: a cooperative vehicle simulator needs to take into account the decision making of numerous traffic participants, provide a meaningful V2X communication infrastructure, and possibly cover large urban areas, thereby producing an unprecedented computational load. The latter is a serious issue in many projects and has led to the need for simulator efficiency research [1]–[3]. To tame this complexity, we divide computationally expensive simulations into a a set of smaller simulations. These smaller simulations are executed by a set of *sub-simulators* that are combined to carry out a larger simulation.

The main research question of this work is: how can a simulation platform satisfying the performance requirements of the cooperative driving domain be integrated into an agile engineering (or research) process where resources need to be provided on demand?

To answer this question we amalgamate our experiences in agile software engineering processes and simulation technology. The concept of *Continuous Integration (CI)* is vital in agile software development projects. The basic idea is that whenever a developer commits his or her work, the continuous integration pipeline assesses the whole updated, integrated code-base with regard to a set of specified project requirements. The assessment usually consists of compiling the project and performing unit and integration tests, but can also include the evaluation of non-functional metrics. The point is that developers neither need to know how the CI pipeline works nor should they be able to alter it. The only thing they are interested in is to be notified whenever something's gone wrong.

The development of cooperative vehicles can be seen as a specific kind of software projects, too, and its peculiarity is the need for simulation. Whenever a function or a parameter of the vehicle software is changed, in addition to the tasks mentioned above, the CI should automatically re-simulate a set of pre-defined scenarios, store the results so that they can be retrieved and evaluated by the developer on-demand or notify the developer if simulation constraints were violated (for instance, if a crash was recorded). In the meantime, the developer must be able to continue working on the project. The introduction of simulator into a CI workflow poses multiple challenges, concerning its architecture, interfaces, and automated resource management. The main contribution of this paper is a *continuous simulation* platform providing scalability at increasing simulation complexity and hiding the sub-simulator architecture and the resource management overhead from the simulation engineer.

The remainder of this paper is structured as follows. In Section II we provide the foundation on which we build our work. In particular, we introduce the basic concepts of the

This work was supported by the Grant SPP1835 from DFG, the German Research Foundation.

[KKRZ19] J. C. Kirchhof, E. Kusmenko, B. Rumpe, H. Zhang:
Simulation as a Service for Cooperative Vehicles.
In: Proceedings of MODELS 2019. Workshop MASE, pp. 28--37, IEEE, Munich, Sep. 2019.
www.se-rwth.de/publications/

MontiSim simulator, the core simulation engine developed for small scale cooperative vehicle scenarios. The service architecture leveraging MontiSim to a cloud-ready simulation platform by a hook-up of arbitrarily many MontiSim instances is presented in Section III. To better understand the underlying communication patterns, we discuss the main interfaces of the microservice-based architecture in Section V. In Section VI we present experimental results, showing how large-scale simulations can benefit from a distributed cloud-based solution. In Section VII we discuss related works before we conclude our paper in Section VIII.

II. Background

MontiSim is an Intelligent Transportation System (ITS) simulator developed to facilitate the evaluation of model-driven development techniques for automated vehicles [4]. Particularly, its main focus is Model in the Loop (MiL) and Software in the Loop (SiL) testing of Component & Connector (C&C) based autonomous vehicle controllers [5], [6] and cooperative driving applications [7], [8] in automotive model-driven systems engineering processes such as SMArDT [9], [10]. Therefore, MontiSim was designed as an agent-based simulator similar to MATSim [3] and Matisse [11]. Each vehicle has its own exchangeable behavior controller making its own independent decisions based on its own sensing. Controllers can be integrated into the simulator and exchanged using the middleware modeling approach [12].

Furthermore, MontiSim provides high flexibility concerning the simulated scenarios as well as modularity and extensibility of the vehicle models, technical infrastructure, and the environment. For instance, the desired simulation scenario specifying the vehicles involved as well as their goals can be easily set up using a dedicated Domain Specific Language (DSL) [4] whereas OpenStreetMap (OSM) support enables the import of arbitrary real-world maps [13].

However, advancing to the simulation of *cooperatively interacting* vehicles leads to a set of new requirements as mentioned in Section I and elaborated in previous work [14]. To leverage MontiSim to a cooperative driving simulator, using its co-simulator pattern it was extended by a discrete event network co-simulator providing means for V2X communication. To tackle the exploding computational complexity in large simulations, the so-called sectoring approach was proposed, which will play a central role in this work. The idea is to subdivide a large simulation area into small partial regions. Each region is then simulated by a dedicated simulator instance. The core engine of each sub-simulator is still driven by the original non-distributable simulator. Note that we distinguish between co-simulators and sub-simulators in this work with co-simulators being parts of an overall simulator providing additional *functionality* while sub-simulators all have the same capabilities but work on different chunks of the simulation space.

The simulator core is embedded into a sub-simulator shell, which provides communication services for data exchange and synchronization between the instances of the simulator network. Additionally, a master entity is required to govern the sub-simulator network. Neither automatically scaling architectures, nor the performance gains achieved by the sectoring approach were covered in previous publications.

In this work, we shift the focus from the sectorization concept to its architectural and technical implications. In the course of our research, we have learned that providing a distributable architecture does not solve the problems of the simulator user instantly. Instead, new questions arise: how can an engineer deploy the distributed solution in his environment? How can he manage the amount of simulation resources or obtain the required resources on demand? How can a distributed simulator be integrated into an automated CI process providing simulation as a service (SimaaS), ensuring scalability, and allocating the required resources on demand in a multi-user environment, e.g., a company or research institute with many engineers?

To answer these questions, the distributable simulator architecture needs to be coupled with an appropriate technical infrastructure. In this work, we elaborate such a symbiosis relying on sectoring combined with *containerization* of simulator modules. The result is easily hostable on any container-based cloud service such as Amazon Web Services (AWS) or Microsoft Azure. We demonstrate scalability in a series of experiments. The concepts are applicable to any vehicle simulator with a decent Application Programming Interface (API) for simulation control, cf. Section V, e.g., SUMO-based simulators [15] including TraNS [16], iTETRIS [17], and Veins [18] using the TraCI API [19].

III. Architecture

Traditional ITS simulators are designed for the execution on a single host machine. Although well-written parallel simulation code can benefit from multiple processors, e.g., MATSim DEQSim [3], the simulation cannot scale beyond the capabilities of its physical host. When it comes to heavy tasks involving massive data such as vast maps covering tens, or hundreds of square kilometers of urban area frequented by large numbers of traffic participants, it becomes necessary to be able to distribute the simulation to *arbitrarily many* worker machines.

The considerations of this section are not only intended to provide a specific solution for distributed continuous simulation, but can also be seen as a reference architecture for new distributable simulators. Additionally, it reveals essential prerequisites for the retrofitting of existing stand-alone solutions. A distributable simulation solution S_d can be obtained by enriching a stand-alone simulator S_s with a set of adapters A as well as distribution functionality D. Furthermore, a set of interfaces I is crucial for seamless integration of S_s into a distributable simulation platform. The interfaces I essentially allow configuring, controlling, and observing the simulation and are further discussed in Section V. The adapters A are used to connect the distribution functionality D to the interfaces I, e.g., by adding networking capabilities not included in S_s.

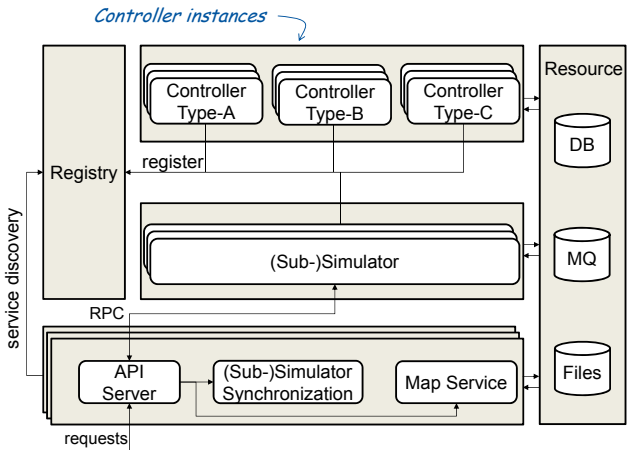

Fig. 1. Overview of the architecture of the distributed simulator.

Our reference architecture consists of a set of heterogeneous components as depicted in Fig. 1. The architecture mainly consists of four elements: a set of simulators that (unknowingly) collaborate to execute a common simulation, controllers that determine the behavior of the simulated vehicles, a registry used for discovering simulators and controllers, and an API server that carries out coordination tasks and allows specifying the simulation.

Each **sub-simulator** is an instance of the stand-alone simulator S_s (in our evaluation MontiSim) responsible for a partial area of the overall map, or a so-called *sector*. MontiSim is unaware of the fact that it is embedded in a distributed context. The stand-alone simulator S_s is designed to do the actual simulation work. Its functionalities include the simulation of vehicles, pedestrians, traffic lights, etc. To embed S_s in a simulator ensemble, we need to be able to control the simulation process from outside. Therefore, S_s is required to support a set of actions, namely, vehicle initialization, vehicle removal, map loading/re-loading, simulation data retrieval, and, most importantly, simulation start, step, pause, and stop. To make these actions accessible to other components as a service, they are exposed via the simulator service interface I. Common solutions for the technical realization of I are RPC, RESTful and RESTful-RPC hybrid [20, p. 16]. They have their own advantages and disadvantages: while RPC-style APIs are great for actions, RESTful services are more resource oriented. Therefore, an RPC-style interface is the most appropriate for our distributed architecture.

Controller models represent the software controlling the complete vehicle behavior, usually designed using the EmbeddedMontiArc modeling language [5], [6]. In the simulation process, sensor inputs of a vehicle are forwarded to its respective controller, the controller makes the driving decision based on these data and returns the resulting actuator commands back to the simulator [14]. In our architecture, we introduce controllers as stand-alone services available to simulators.

The communication between the controller and the simulator is based on an Remote Method Invocation (RMI) interface provided by the controller service. Decoupling the controller from the simulator is a central architectural decision. If the simulator is regarded as a means to validate driving behavior, e.g., in agile autonomous driving projects, the controller is a piece of software which is exchanged and updated on a frequent basis. Exchanging the vehicle behavior must, therefore, be easy and should not affect the simulator. Furthermore, as is inherent for agent-based systems, we require the possibility to equip each traffic participant with an individual behavior configuration. Therefore, whenever a simulator requests a controller for one of its vehicles, it can specify the concrete controller type. The controller service then instantiates the correct controller implementation as well as its parameters. This enables simulating realistic heterogeneous mixed-traffic simulations, as well as comparing the performance of competing solutions against each other. Running control code as an external process is a common pattern, e.g., used in Gazebo through a Robot Operating System (ROS) interface [21]. Without an appropriate controller management infrastructure, this is cumbersome to cope with: the user, among other tasks, needs to instantiate a controller per vehicle, take care of the associations to their respective vehicles, and clean up after usage. These management tasks are provided by the controller service in our architecture.

Maintaining reliability in such a dynamically extensible distributed infrastructure requires elaborate service management and supervision. In our architecture, this functionality is provided by the Apache Zookeeper framework [22]. To make simulation and controller services visible to other components, we introduce a service **registry**. In this registry, Zookeeper organizes the registered services as nodes in a tree structure. Services publish and keep updating their respective configurations including host addresses, ports, versions, and available APIs to this registry. Furthermore, a health check mechanism ensures that only reachable services are published. Other components of the system discover available services through this service registry and monitor their liveness by watching a set of specific nodes. For instance, if we have a sub-simulator `sim1`, it is registered with its hostname, port, and further information as a node called `/service/simulator/sim1`. Other components can discover all available simulator instances by querying the children of `/service/simulator`. The same holds for controller instances. If the simulator spawns a new vehicle and needs an appropriate controller instance, it can look it up using the registry service.

So far, we have only introduced interfaces between system components. However, to make the architecture useful, we also need to provide an interface to the outside world. In particular, this interface must enable the simulator user to set up a simulation task and output the computed results. We provide such an interface through the **API server**. This component can be seen as a *facade pattern* [23] dispatching requests to the respective components and hiding the actual complexity of the architecture from the user. To set up a simulation, map and vehicle data including the vehicles' source and destination

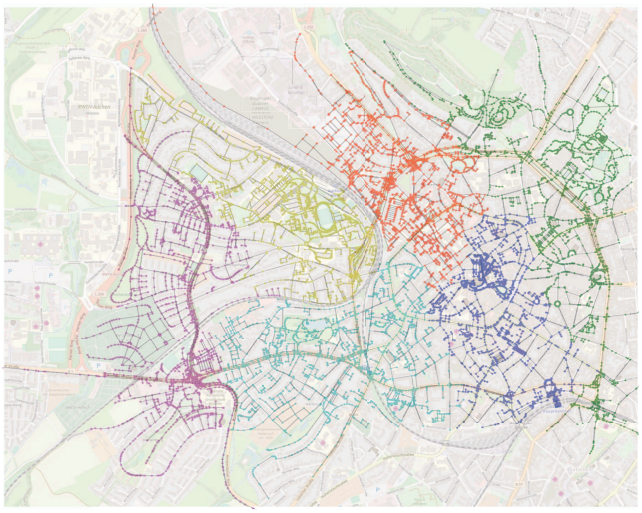

Fig. 2. Sectorization result of Aachen. Map nodes that belong to the same sector are marked with the same color.

coordinates must be provided to the simulation through the API server. This can be done interactively through a web GUI connected to the simulator [14] or in a headless way, e.g., by a script in a CI pipeline.

Distributing the simulation among several independent sub-simulators requires us to synchronize the sub-simulators after each simulation step. The synchronization phase is carried out by a module of the API server and deals with tasks related to handing over vehicles between map sectors and V2X communication between agents located in different sectors. Assume we have a vehicle V currently driving through map sector Sec_a simulated by sub-simulator S_a. At some point, V reaches and crosses the boundary of Sec_a and enters Sec_b, simulated by simulator S_b. To implement the handover, we need to *remove* V from S_a and *initialize* a new vehicle V' in S_b where $V = V'$, that is, we create a copy of V in S_b and remove it from S_a.

This process requires some data exchange between the API server and the simulators and adds overhead to the simulation. In contrast to [14] where the map is subdivided into equally-sized rectangles, we reduce vehicle handovers by applying the parallel graph partitioning algorithm METIS [24] to our road network. By minimizing the number of edge cuts, it also minimizes the potential number of trajectories crossing sector borders. What is more, trajectory planning on a large map can also benefit a lot from using overlay graphs [25]. We encapsulate the *map service* carrying out the sectorization and routing tasks in a dedicated module. Decoupling it from the API server facilitates further experimentation on sectorization and route planning. As an example, we show the sectorization result in Fig. 2, where we use METIS to sectorize the map of Aachen into 6 different sectors.

Being a central entity in our architecture, the API server also plays an important role in resource management. Before starting a simulation, it decides how to choose concrete simulator and controller service instances in order to utilize resources effectively. For example, if the whole simulation involves n map sectors, it is obviously unwise to reserve more than n simulator instance at the beginning.

Finally, we employ a set of data services as depicted in the rightmost box in Fig. 1. Their purpose is the storage of resources such as map and configuration data as well as data exchange between functional components. We use a MySQL database (DB in Fig. 1) to store the path of map files and the map partition results. Message queues (MQ in Fig. 1) are used to exchange messages between simulators to support V2X communication. Storage and management of all map data can be done locally on the API server or alternatively in a cloud-based service such as Amazon EFS (Files in Fig. 1)).

IV. Continuous Simulation

Our goal is to provide a solution that integrates seamlessly into agile self-driving vehicle development processes. Continuous integration is an important and widely used means for quality assurance, however seldom adopted in disciplines beyond classical software engineering. Whenever engineers commit code changes, a CI server is triggered and performs a series of tests and other quality assessment tasks. In MiL, SiL, and Hardware in the Loop (HiL) tests this cannot be done without a simulator service in the CI toolchain.

Due to our service-based architecture, a simulation can be triggered in a CI pipeline by interacting with the API server managing the simulation cluster. Thereby, the API server expects a simulation description model telling the simulator what it is expected to simulate [4] as well as a corresponding OSM map. Once the simulation is finished, simulation results are returned to the CI server and can be used for automated evaluation or to generate statistics. For instance, we want to ensure that our autonomous driving controllers do not produce vehicle crashes and also keep the vehicle on the road. Simulation results are provided as timed data frames containing all trajectory and crash information, cf. Fig. 3. This data can be analyzed automatically, e.g., it can be searched for crashes and off-road trajectories. For instance, the shown frame contains the state of the vehicle with the id defined in line 11. Its position is held in a three-dimensional vector representing the vehicle's longitude, latitude, and height information in lines 4-6. The variable `totalTime` denotes the absolute point in simulation time at which the frame was captured, while `deltaTime` is the simulation time elapsed since the last frame for this vehicle was recorded. `steering` contains the actual steering angle for this vehicle. The Boolean variable `collision` is set to true, iff the collision detection observed a collision for this vehicle during the current time frame.

If the analysis terminates with no findings, the CI pipeline succeeds. Otherwise, a problem report is generated, e.g., mentioning the problem, the corresponding simulation time, as well as the vehicles involved, causing the CI pipeline to fail. In the case of a failed pipeline, the engineer is notified and can analyze the problem. Additionally, the frame data can

```
"frames": [                                                    JSON
  {
  "deltaTime": 33, //simulation time since last frame
  "position": [    //current vehicle position
    5.8806556, 50.83668824517681, 0.5429999999999999
  ],
  "steering": 0.0165, //steering angle in radians
  "collision": false, //no collision at this sim. step
  ... //further vehicle information
  "totalTime": 4059,
  "vehicleID": "45124cec-120e-11e9-85d6-d2002090b701" //
      ↪ unique vehicle id
  },
  ... ]
```

Fig. 3. Excerpt of a vehicle frame serialized using the JSON format.

be used to visualize the simulation helping to understand the circumstances of the problem [14].

The presented simulation as a service (SimaaS) is a central tool to the continuous simulation lifecycle enabling the developer or the researcher to focus on functionality and only requiring a human to look at simulation results when something undesirable happens. Particularly, if the simulation infrastructure is deployed in a cloud, as CI services mostly are today, the developer does not have to supervise the simulation, nor to deal with technical details of its execution, configuration, or resource management.

The purpose of cloud computing is not just to outsource a static amount of computational resources to an infrastructure provider, but rather to provide the required resources on-demand. This is usually achieved by packaging executables into so-called containers: a container can be understood as a lightweight Virtual Machine (VM) simulating the operating system environment including all of its dependencies and other resources required to execute the actual job. Working out-of-the-box as long as the host system is supported, software encapsulated in a container is easy to ship, deploy, and update. When the computational load increases or more data needs to be processed, containers are replicated and executed on newly allocated resources. When a container is not needed anymore, it is destroyed and the computational resources are freed. Docker is the most common containerization tool to date. Cloud services like AWS are built around Docker technology thereby providing a systematic way to enable fast and automated deployment of Linux applications inside portable containers [26].

Our service-based architecture is tailored to fit the pattern described above. In Fig. 1, components that are required to scale frequently are depicted as component stacks: sub-simulators and controllers are created on-demand, i.e., they are only needed if a simulation is requested. Moreover, the number of required sub-simulators depends on the size of the currently simulated area. Similarly, the number of controller instances is determined by the number of vehicles we want to control in the simulation. Therefore, we enable cloud-capability for our simulator architecture by encapsulating each service instance in Fig. 1 in a Docker container.

How can this technology be integrated into agile research and development processes for cooperatively interacting vehicles? Consider a team working on a new module for cooperative trajectory planning. To validate behavior correctness, the module needs to stand several simulations, e.g., without producing a crash. Whenever new commits are pushed into the code repository of the module, a new Docker image for the controller service is automatically built and pushed into the Docker registry in the cloud by the team's CI server. A simulation is triggered using the new version of the cooperative trajectory planning module for some previously defined scenarios and the results are sent back to the CI server. The latter can now mark the pipeline as succeeded or failed and inform the developer in charge. This guarantees that a software failing to avoid accidents or to fulfill other requirements in the simulation is automatically rejected by the Q/A as long as the CI pipeline keeps failing.

Another positive aspect is that the cooperative driving development team automatically uses the latest available simulation software. Whenever the simulator vendor releases a new version, new simulator containers are built and deployed. Thereby, all old service instances are replaced by new ones featuring the latest simulator version. The target machines which the components are running on just need to pull the latest Docker image and start the new container. We do not roll out an update to all machines at the same time, which is also referred to as zero-downtime updating. If a problem is found and rollback is needed, the worker machines only need to pull the previous Docker image and replace the problematic one.

Scaling dockerized software also imposes a series of challenges: server clusters consist of many machines having different hardware, different network configurations, such as IP addresses, available ports, etc. How can we schedule and coordinate distributed simulations in such heterogeneous environments? Container orchestrating solutions are emerging in the cloud era, with the most popular solutions being Kubernetes, Mesos, and Docker Swarm. We employed Kubernetes in our architecture since it is best supported by many cloud providers.

In a Kubernetes cluster, a set of master components provide the cluster's control plane. There are also nodes, which are worker machines and may be either a virtual or a physical machine. They contain services necessary to run pods and are managed by the master. The pod is the basic building block and management unit of Kubernetes. It runs a single container or a small number of tightly coupled containers. In our prototype, we deploy each component as a single container pod. Another essential element of Kubernetes is *deployment*, the purpose of which is to keep identical pods running and upgrade them in a managed way. Kubernetes performs rolling updates for pods by default. The zero downtime updating feature is backed by this mechanism. Kubernetes Deployments can also be used to make the application scalable. It enables us to scale the components of our simulation architecture. Our components can be configured to auto-scale based on CPU utilization and customized metrics, more specifically, on the number of available controller and simulator instances. For instance, if all available containers are busy, Kubernetes can

initialize new ones pre-emptively so that incoming simulation requests can be handled with lower latencies.

V. Interfaces

As we have seen in the previous sections, the simulator needs to be interfaced by CI jobs, GUIs, and other simulator components. In the following we list some of the most important requirements regarding the interfaces:

- (R1) Map configuration (upload/partition): the simulator must offer an interface for map upload/selection by an external actor (a user or a CI pipeline). Additionally, the number of sectors the map is separated into also needs to be configurable, since it affects simulation performance and resource consumption. More specifically, this number affects how many simulator instances will be consumed by the simulation.
- (R2) Vehicles creation/deletion: the simulator needs to offer an interface to define and configure vehicles before the simulation is started. This action must happens inside the distributed system. Before simulation starts, the vehicles are created in their respective sub-simulator instances by the API server. Furthermore, vehicle creation and deletion is needed for vehicle handovers between sub-simulator instances as discussed in Section III.
- (R3) Simulation control (start/step/pause/stop): The interface must offer a way to start the simulation asynchronously after map and vehicles have been configured. That is, clients such as browsers, or our CI servers must be able to start the simulation using this interface and receive a simulation id right away. They can use this returned id to control (pause/stop) the simulator or retrieve the results once available. Once the client has requested a simulation, the API server needs to dispatch the simulation tasks to selected sub-simulators and to continue executing single simulation steps followed by a synchronization phase until the simulation is finished.
- (R4) Data exchange: the simulator needs an interface for the exchange of simulation data such as configuration, results, status, etc. Clients can use this interface to retrieve simulation results for analysis. The API server also needs this interface to retrieve the vehicles' status to synchronize data between all sub-simulators. For instance, if a vehicle approaches a sector boundary, the API server needs to initiate a handover.

We have presented the distributed simulation solution S_d as a microservice-based architecture. In this architecture, components are deployed into a distributed environment. To implement the required interfaces, we face a new challenge: communication. In a monolithic software, components invoke one another by calling language level functions. In a distributed environment, however, components mostly do not run on the same machine, let alone in the same process or memory space. This means the communication between the components has to be routed through the cluster network. Thereby, we must avoid the fallacies of distributed computing [27] in this approach. For instance, we always assume that the network is not reliable, hence we deploy multiple instances for each component on different machines, to ensure that if a machine is not reachable, there is always a backup instance available. This is critical for components that are necessary for other services, such as the registry. We support two strategies for the instantiation of backup containers: proactive instantiation spawns more containers than actually needed. This enables fast failover if a container needs to be replaced. On the other hand, maintaining idle components is expensive. This strategy should be employed only if high performance is crucial. Reactive instantiation, on the other hand spawns backup containers when they are needed. This is the recommended strategy for most applications.

Moreover, we take advantage of the deployment component offered by Kubernetes to ensure that unreachable services are always restarted or recreated on healthy nodes. Since new machines could be added to the distributed system, or services could be redeployed on other machines, the network topology is volatile. This is solved by the internal DNS service offered in Kubernetes. From the services' point of view, an IP and port are no longer involved to communicate with others, as the hostname is a sufficient resource locator. If a service changes its access point (IP, port), the Kubernetes DNS service will update this information accordingly.

Note that the requirements listed above contain interfaces used for communication (1) between users and the distributed simulator S_d and (2) inside of S_d. For (1), they are a set of public APIs that are exposed to the client. They are simple and stateless. Hence, we realized them as a set of RESTful APIs. For this RESTful API, we model resources such as vehicles and maps according to the simulation description format as defined in [4]. Since RESTful APIs are resource oriented, make full use of HTTP 1.1 verbs, and the payload is represented using the JSON format, it is human readable, developer friendly, and can be easily integrated with other software. For example, we define the remote access methods **POST /vehicles** to create vehicles, **GET /vehicles/:id** to retrieve the vehicle status for a given id, **POST /simulations** to set up a simulation, **POST /simulations/:id/start** to start it, and **GET /simulations/:id** to pull the simulation results.

In some use cases, users need to see the visualization of the simulation on-line, i.e., without having to wait for the simulation to finish. To enable fast visualization, we integrated a WebSocket API sending simulation results frame by frame to the clients as soon as they are available. This is practical for engineers if they want to use the simulator interactively receiving immediate feedback on their software.

Interfaces belonging to type (2), on the other hand, are different. Being a set of internal APIs they are invoked much more often than type (1) interfaces and generate the majority of network traffic inside S_d. The main load is thereby generated by the simulator synchronization as discussed above. We implemented type (2) interfaces using RPC-style APIs, since most RPC frameworks provide much better performance than HTTP 1.1 and RESTful. These frameworks use data serialization to reduce traffic and provide alternatives to HTTP

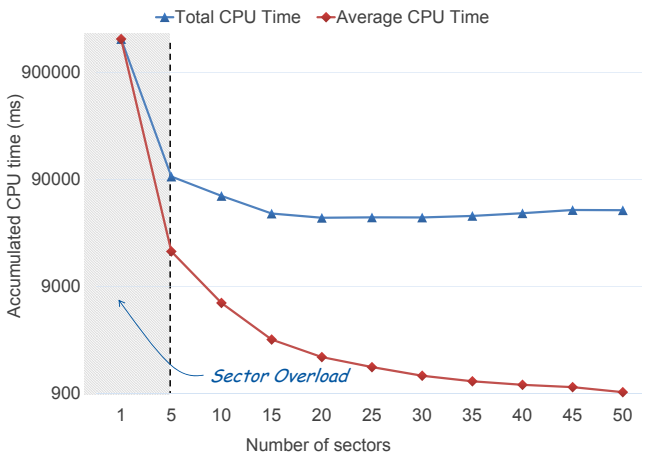

Fig. 4. Computing resource consumptions as the total (blue) and average (red) CPU time consumption of the sub-simulators.

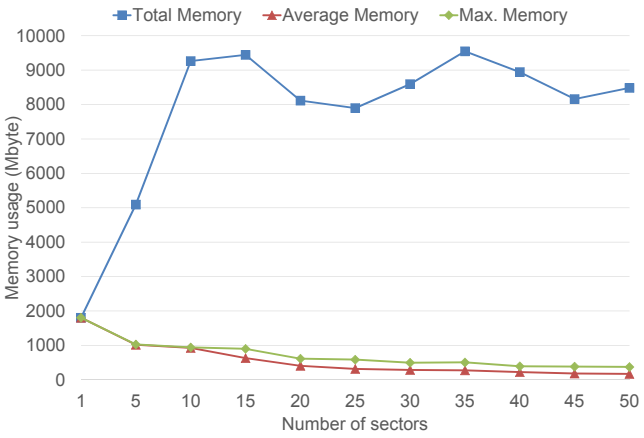

Fig. 5. Memory resource consumptions as the total (blue) and average (red) maximum memory usage of the sub-simulators.

1.1 such as TCP sockets, HTTP 2, etc., to optimize the network performance. To be precise, we chose Google's gRPC framework due to the following reasons: First, gRPC uses HTTP/2, which provides long stable connections. This reduces the (re-)connection overhead during sub-simulator synchronization. Second, gRPC uses protobuf, which is more efficient than other RPC serialization formats such as XML. This reduces the amount of transmitted data during synchronization.

VI. Experiments

To validate that our distributed simulator scales for an increasing number of sub-simulators and is hence suitable for large scale cloud-based simulations, we set up a series of experiments measuring the resource consumption depending on the number of sectors. Note that this is *not* an evaluation of the core simulator performance, but rather of the scalability of our distribution concepts.

In the first experiment, we demonstrate that not only the average resource consumption per sub-simulator but also the total resource consumption can be tremendously reduced by distributing the simulation workload. We simulate an OSM map containing 267784 nodes and covering 32km^2 of the city of Munich spanned by the coordinates (N48.1164000° E11.5338000°) and (N48.1593000°, E11.6242000°). The map is separated subsequently into different numbers of sectors, from 1 (corresponding to no distribution) to 50, and populated with 500 randomly, spatially uniformly distributed vehicles.

For each sub-simulator, the individual CPU and memory usage is obtained from its container's Pseudo-file [28] used by Docker to log its resource usage statistics. The experiment is performed on a VM consisting of 100 GB RAM, 50 CPU cores clocked at 2 GHz, and Docker 18.09.7. The containers are based on the OpenJDK 8 image which includes JVM version 8u212; its memory is limited to 8GB; other Docker settings are set to default. In each simulation, we create completely fresh containers. Fig. 4 shows the total CPU time consumption of the simulation (blue) and the sector average consumption (red) depending on the number of sectors (which is also the number of sub-simulators). Note that the vertical axis has a logarithmic scale due to the wide range of values.

The average CPU consumption decreases by two orders of magnitude as we increase the number of sectors from one to five. It decreases drastically mainly because each sub-simulator computes fewer trajectories as the number of sectors increases. Intuitively, distributed algorithms should introduce some overhead, i.e., the *total* resource consumption should increase. Interestingly, though, distributing the simulation to multiple sub-simulators has a positive impact on the total CPU usage, as well. The most likely reason for this is that if the host machine is overloaded by the simulation, e.g., because the map does not fit into the memory, the simulation process needs to spend more time dealing with resource management, e.g., memory swapping.

Fig. 4 shows that using five or fewer sub-simulators overloads the host hardware, which is denoted by the hatched area. As we continue to increase the number of sectors, the reduction of average CPU consumption gets slower. This is mainly due to the fact that beyond a certain point no tasks can be distributed anymore. A minimum amount of resources is needed to keep an idle sub-simulator alive. The overload region varies depending on the concrete simulation scenario, particularly on the number of vehicles involved. In summary, we manage to reduce the average CPU consumption by 99.95 %, from 1846821 ms with 1 sub-simulator to 931.66 ms with 50 sub-simulators.

Looking at the memory consumption in Fig. 5, we realize that the average maximum memory consumption per sub-simulator (red) decreases steadily, almost linearly. The average memory usage is reduced by 90.59 % when the number of sectors is increased from one to fifty. In contrast to the CPU plot, we can clearly see the overhead we have to pay for the distribution as an increase in the total memory usage. Nevertheless, we think that this increase is justified by the reduction in CPU time and average memory consumption.

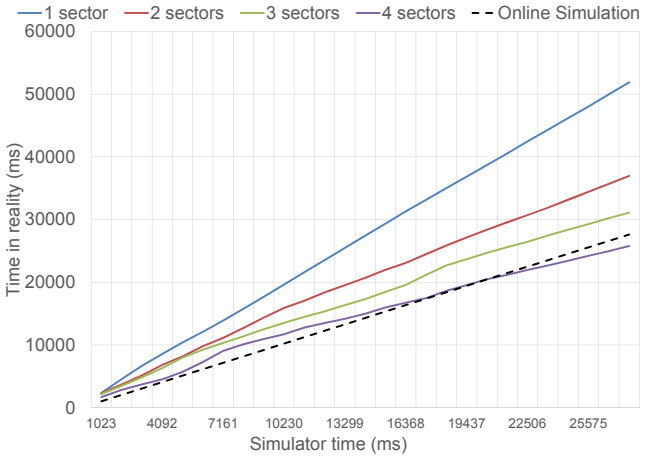

Fig. 6. Simulation result (dataframe) generating speed when running the same simulation scenario with 1, 2, 3 and 4 sectors.

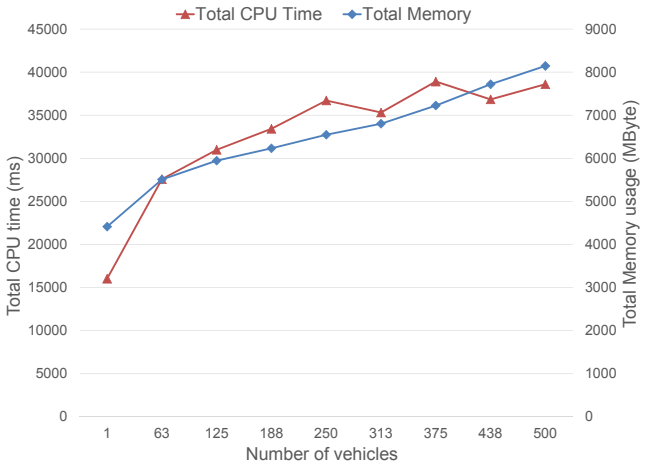

Fig. 7. Resource consumption when simulating with different number of vehicles.

To ensure that we have an approximately even memory distribution among the sub-simulators, the green curve in Fig. 5 denotes the maximum memory consumption allocated by a single sub-simulator in the course of the simulation (the maximum is over all sub-simulators and all points in time). It is an upper bound on the memory needed per sub-simulator to perform the simulation. As this curve barely differs from the average memory curve, we can conclude that the distributed architecture is really scalable—no sub-simulator has to carry the majority of the workload.

The goal of our second experiment is to show that our simulator performs faster when simulating fewer vehicles per sector. In this experiment, we use a machine with a quad-core Intel E3-1230 CPU@3.3 GHz and 16 GB RAM, JVM memory is limited to 4 GB. Other settings remain the same. The simulation is based on a map of Aachen spanned by the coordinates (N50.7669000°, E6.0762000°) and (N50.7802000°, E6.0932000°) with 14517 nodes covering an area of 1 km^2. Four vehicles are simulated for approximately 26 seconds of simulation time using one to four sectors. For each number of sectors, the vehicles are positioned so that each sector contains at least one vehicle. After each simulation step of 1023 ms of simulation time, we record the time consumed to compute it.

Fig. 6 depicts the results of this experiment. The gradient of a curve denotes the real-time/simulated-time ratio α. The less load a sub-simulator has, the smaller the slope and the faster the simulation. While we observed $\alpha \approx 2$ when no distribution was applied, we obtained a simulation speed close to real-time by increasing the number of sub-simulators to four. This is a significant bound, as real-time simulations can be particularly important for HiL testing [1] or if instant visualization is required.

The goal of our third experiment is to analyze the relation between resource consumption and the number of simulated vehicles created in the sub-simulators. We simulate the same map as in the first experiment with 20 sectors. In each simulation, we populate the sub-simulators with random, evenly distributed vehicles.

Fig. 7 shows the results of this evaluation. The blue curve shows that as we increase the number of simulated vehicles, the overall memory consumption increases almost linearly.

The process of vehicle creation mainly includes trajectory planning, loading physical models into memory, and initializing sensors. Every sub-simulator only needs to load its map once. Thereafter, it can compute trajectories for all vehicles in its sector. For this reason, the trajectory planning does not have a large impact on the overall memory usage. Loading physical models and initializing sensors, on the other hand, induces similar overheads for each vehicle. Therefore, the overall memory consumption increases roughly linearly with the total number of simulated vehicles.

The curve of total CPU time shown in Fig. 7 differs from the one representing the total memory usage. Overall, the total CPU time increases as the number of vehicle increases. At some segments, e.g., going from 250 to 313, and from 375 to 438 vehicles, however, the total CPU time decreases. The reason for that is that the time the JVM spends on resource management is not predictable in our experiment. More specifically, the CPU cost for garbage collection in the experiment where we created 250 vehicles is higher than the cost with 313 simulated vehicles. Besides CPU cost of the JVM itself, we expect the total CPU time to increase because the sub-simulators need more CPU resources for trajectory planning.

Overall, we found out that splitting the simulation task into several sectors can drastically reduce resource consumption. However, an optimal setting depends on traffic density, total map size, and the host hardware. The absolute values shown in this evaluation strongly depend on the implementation of the core stand-alone simulator, namely MontiSim. Yet we assume that integrating another simulator, e.g., Sumo or Veins, would lead to similar positive effects.

VII. Related Work

Shekhar et al. offer the SimaaS concept, as well [29]. Similar to our approach, the authors propose to subdivide a larger simulation into multiple sub-simulators running in independent Docker containers. However, in contrast to our approach, their work focuses on executing multiple simulations with different parameters and aggregating the results afterwards. Our system instead splits up large simulations into multiple jobs that are synchronized while the simulation is executed. In a case study, the proposed system carries out a traffic simulation based on SUMO. In contrast to MontiSim, SUMO is not capable of testing different vehicle controllers. Furthermore, by decoupling the system-under-test, i.e., the vehicle controller, from the simulation environment, the system-under-test can be replaced more efficiently. This is crucial in a CI use case where many different revisions of the system-under-test need to be tested.

SEMSim [30] also offers an architecture for executing traffic simulations in the cloud. Similar to our architecture, an API takes a simulation specification as input. A dispatch server then assigns the requested simulation to a set of VMs called *simulation instances*. Compared to our approach, SEMSim provides more sophisticated data protection mechanisms. However, SEMSim only parallelizes multiple runs of a simulation, while our architecture allows distributing single runs of a simulation across multiple containers. Hence, our architecture is more scalable as it allows running large simulations that cannot be executed by a single VM. At the same time, our approach can parallelize multiple runs of a simulation by sending multiple requests to the API Server.

The main driver behind our parallelization of a traffic simulation is the division of the map into several sectors, where all sectors are simulated in parallel. This strategy has previously been proposed in other parallel traffic simulators such as the parallel implementation of the TRANSIMS microsimulation [31]. Similar to our approach, this work represents the map as a graph and then uses the METIS graph partitioning algorithm to distribute the workload across multiple CPUs. The authors show that this strategy enables an efficient simulation for maps consisting of large numbers of edges. The core simulator TRANSIMS focuses on high-level strategic routing decisions of vehicles based on, e.g., congestion. In contrast, our simulator focuses on low-level decisions based on, e.g., sensor data that are necessary to develop vehicle controllers.

Kiesling and Luthi discuss how to parallelize traffic simulations by subdividing the simulated time rather than the map [32]. To do so, a synchronization between time slices is required. While this parallelization strategy might apply to comparably simple traffic models such as the Nagel/Schreckenberg model [33], the authors remark that their strategy is ineffective if the simulation relies on complex states. Hence, their parallelization approach cannot be applied in use cases targeted by MontiSim, where the vehicle controllers of each vehicle rely on a large number of parameters such as sensor inputs.

VIII. Conclusion

In this work, we presented an architecture as well as a reference implementation for distributable agent-based ITS simulators. The concepts can be applied to any stand-alone simulator to enable large scale simulations as long as it fulfills a set of discussed prerequisites. The achieved scalability is supported by a series of experiments showing that both workload and memory consumption of each worker can be reduced substantially by the sectoring approach combined with containerization. Although the distribution is not for free, as we pay with increased total memory consumption and network resources, surprisingly we observed a decrease in total CPU time when multiple sectors are used.

However, the employed sectoring approach has both advantages and disadvantages. A major drawback of static sectoring is that some sectors might be deserted while other, highly frequented sectors are still overloaded. The optimal number of sectors and their shapes are parameters which need to be chosen thoroughly and are subject to future research. Temporarily shutting down simulators assigned to abandoned sectors could improve the resource consumption.

References

[1] Christina Obermaier, Raphael Riebl, and Christian Facchi. Fully Reactive Hardware-in-the-Loop Simulation for VANET Devices. In *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 3755–3760. IEEE, 2018.

[2] Stefan Neumeier, Christina Obermaier, and Christian Facchi. Speeding up OMNeT++ Simulations by Parallel Output-Vector Implementations. *5th GI/ITG KuVS Fachgespräch Inter-Vehicle Communication*, page 22, 2017.

[3] Michael Balmer, Marcel Rieser, Konrad Meister, David Charypar, Nicolas Lefebvre, and Kai Nagel. MATSim-T: Architecture and simulation times. In *Multi-agent systems for traffic and transportation engineering*, pages 57–78. IGI Global, 2009.

[4] Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles. In *EXE at MODELS*, 2017.

[5] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[6] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 447 – 457. ACM, October 2018.

[7] Maximilian Kloock, Ludwig Kragl, Janis Maczijewski, Bassam Alrifaee, and Stefan Kowalewski. Distributed model predictive pose control of multiple nonholonomic vehicles. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 1438–1443. IEEE, 2019.

[8] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling dynamic architectures of self-adaptive cooperative systems. *Journal of Object Technology*, 18(2), 2019.

[9] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. SMArDT modeling for automotive software testing. *Software: Practice and Experience*, 49(2):301–328, February 2019.

[10] Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Mike Lorang, Bernhard Rumpe, Albi Sema, Georg Strobl, and Michael von Wenckstern. Model-Based Development of Self-Adaptive Autonomous Vehicles using the SMARDT Methodology. In *Proceedings of the 6th*

International Conference on Model-Driven Engineering and Software Development (MODELSWARD'18), pages 163 – 178. SciTePress, January 2018.

[11] Behnam Torabi, Rym Z Wenkstern, and Mohammad Al-Zinati. An agent-based micro-simulator for its. In *21st Int. Conf. on Intelligent Transportation Systems (ITSC)*, pages 2556–2561. IEEE, 2018.

[12] Alexander Hellwig, Stefan Kriebel, Evgeny Kusmenko, and Bernhard Rumpe. Component-based Integration of Interconnected Vehicle Architectures. In *30th Intelligent Vehicles Symposium (IV'19). Workshop on Cooperative Interactive Vehicles*, pages 146–151. IEEE, June 2019.

[13] Mordechai Haklay and Patrick Weber. Openstreetmap: User-generated street maps. *IEEE Pervas Comput*, 7(4):12–18, 2008.

[14] Christian Frohn, Petyo Ilov, Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Alexander Ryndin. Distributed Simulation of Cooperatively Interacting Vehicles. In *International Conf. on Intelligent Transportation Systems (ITSC'18)*, pages 596–601. IEEE, 2018.

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

[16] M. Piórkowski, M. Raya, A. Lezama Lugo, P. Papadimitratos, M. Grossglauser, and J.-P. Hubaux. TraNS: Realistic Joint Traffic and Network Simulator for VANETs. *SIGMOBILE Mobile Computing and Communications Review*, 12(1):31–33, January 2008.

[17] Michele Rondinone, Julen Maneros, Daniel Krajzewicz, Ramon Bauza, Pasquale Cataldi, Fatma Hrizi, Javier Gozalvez, Vineet Kumar, Matthias Röckl, Lan Lin, et al. iTETRIS: a modular simulation platform for the large scale evaluation of cooperative ITS applications. *Simulation Modelling Practice and Theory*, 34:99–125, 2013.

[18] Christoph Sommer, Reinhard German, and Falko Dressler. Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis. *IEEE Transactions on Mobile Computing*, 10(1):3–15, January 2011.

[19] Axel Wegener, Michał Piórkowski, Maxim Raya, Horst Hellbrück, Stefan Fischer, and Jean-Pierre Hubaux. TraCI: An Interface for Coupling Road Traffic and Network Simulators. In *Proceedings of the 11th Communications and Networking Simulation Symposium*, CNS '08, pages 155–163, New York, NY, USA, 2008. ACM.

[20] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, Inc., 2008.

[21] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an Open-Source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[22] The Apache Software Foundation. ZooKeeper 3.1 Documentation. [Online]. Last checked 6. July 2019. Avail.: https://zookeeper.apache.org/doc/r3.1.2/zookeeperOver.html.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publ. Co., Inc., Boston, MA, USA, 1995.

[24] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[25] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. *Journal of Experimental Algorithmics (JEA)*, 13:5, 2009.

[26] David Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[27] A. Taivalsaari and T. Mikkonen. A Roadmap to the Programmable World: Software Challenges in the IoT Era. *IEEE Software*, 34(1):72–80, January 2017.

[28] Docker Inc. Runtime metrics. [Online]. Last checked 6. July 2019. Avail.: https://docs.docker.com/config/containers/runmetrics/.

[29] Shashank Shekhar, Hamzah Abdel-Aziz, Michael Walker, Faruk Caglar, Aniruddha Gokhale, and Xenofon Koutsoukos. A simulation as a service cloud middleware. *Annals of Telecommunications*, 71(3):93–108, Apr 2016.

[30] Daniel Zehe, Alois Knoll, Wentong Cai, and Heiko Aydt. SEMSim Cloud Service: Large-scale urban systems simulation in the cloud. *Simulation Modelling Practice and Theory*, 58:157 – 171, 2015. Special issue on Cloud Simulation.

[31] Kai Nagel and Marcus Rickert. Parallel implementation of the TRANSIMS micro-simulation. *Parallel Computing*, 27(12):1611 – 1639, 2001. Applications of parallel computing in transportation.

[32] T. Kiesling and J. Luthi. Towards time-parallel road traffic simulation. In *Workshop on Principles of Advanced and Distributed Simulation (PADS'05)*, pages 7–15, June 2005.

[33] Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. *Journal de Physique I*, 2(12):2221–2229, December 1992.