Published at the 18th Workshop on Multi-agent-based Simulation (MABS 2017)
May, 8 - 9 , 2017

InterSCSimulator: Large-Scale Traffic Simulation in Smart Cities using Erlang

Eduardo Felipe Zambom Santana[1], Nelson Lago[1], Fabio Kon[1], Dejan S. Milojicic[2]

[1]Department of Computer Science — University of São Paulo
[2]HP Laboratories — Palo Alto
{efzambom,lago,kon}@ime.usp.br,dejan.milojicic@hpe.com

Abstract. Large cities around the world face numerous challenges to guarantee the quality of life of its citizens. A promising approach to cope with these problems is the concept of Smart Cities, of which the main idea is the use of Information and Communication Technologies to improve city services. Being able to simulate the execution of Smart Cities scenarios would be extremely beneficial for the advancement of the field. Such a simulator, like many others, would need to represent a large number of various agents (e.g. cars, hospitals, and gas pipelines). One possible approach for doing this in a computer system is to use the actor model as a programming paradigm so that each agent corresponds to an actor. The Erlang programming language is based on the actor model and is the most commonly used implementation of it. In this paper, we present the first version of InterSCSimulator, an open-source, extensible, large-scale Traffic Simulator for Smart Cities developed in Erlang, capable of simulating millions of agents using a real map of a large city. Future versions will be extended to address other Smart City domains.

Keywords: Simulation, Smart Cities, Erlang, Actor Model, Scalability

1 Introduction

The growth of cities population around the world brought numerous challenges to their management and operation, especially in big cities. These metropolises, such as São Paulo, Rio de Janeiro, New York, Mexico City, and Tokyo, have to deal with many problems in different areas such as traffic, air pollution, public transportation, health, and education. One approach to tackling these problems is the concept of Smart Cities [1] that proposes the use of Information and Communication Technologies (ICT) to find solutions to deal with the city problems.

There are already some Smart Cities experiences around the world [2–4] with initiatives in different domains. However, deploying a complete environment to test Smart City Applications and Platforms is still a great challenge due to costs and political issues. Moreover, current Smart Cities experiences have been

deployed in small to medium cities. Deploying such infrastructure in a metropolis such as São Paulo, with 11 million inhabitants, will be much more complicated.

The use of simulators can be a good alternative to support large-scale Smart Cities tests and experiments. These tools can simulate different scenarios with various solutions in many city domains such as traffic, public transportation, and resource utilization. Two main challenges arise from the use of simulators. First, the scale: to simulate an entire city, current tools demand high computational power and a long time to simulate large scenarios. Second, the usability of the tools is important because simulator users are not computer scientists. Hence, a Smart City simulator must be both scalable and user-friendly.

To tackle these two main challenges, we are developing InterSCSimulator, an agent-based Smart City simulator which offers a simple to use scenario definition with massive scalability. To achieve scalability, we used Erlang, a language developed to ease the implementation of large-scale parallel and distributed applications. To offer good usability, we studied different simulators with similar purposes such as MATSim [5] and Mezzo [6]. This paper presents the first version of InterSCSimulator which is already able to simulate large-scale traffic scenarios. Our experiments show that InterSCSimulator supports more than 4 million vehicles in a single simulation using a real map of a large city: we already tried the simulator using the maps of São Paulo, New York, and Paris.

Our simulator uses the concept of agents. In our traffic simulations, each vehicle in the simulated city is an agent that can have different behaviors such as start or stop moving, move in a defined path in the city graph, or change its path. Erlang is widely used to implement multi-agent systems [14–16]. Its programming model, called actor model, is very well-suited for this purpose. In Erlang, each application thread is an actor that executes independently of the rest of the application, and the Erlang Virtual Machine can efficiently create and manage millions of actors.

This paper is organized as follows: Section 2 presents the requirements to develop traffic simulations in Smart Cities. Section 3 compares InterSCSimulator to other traffic simulators. Section 4 describes the Actor Model and the Erlang language and relates it to the development of multi-agent applications. Section 5 presents the architecture and implementation of InterSCSimulator. Section 6 shows the simulator performance and usability evaluation. Finally, Section 7 addresses our conclusions and future work.

2 Requirements

To define the functional requirements for the initial version of our simulator, we reviewed the literature on smart cities domains [2, 3] and Smart City simulators. We then opted to begin by using traffic scenarios, which have many implementation challenges such as scalability and usability. To implement traffic scenarios we found four essential functional requirements:

City Road Network: A Smart City simulator has to represent the city road network in a model easy to manipulate algorithmically. A good approach is

to create a digraph based on the city map, which can be acquired from many different services such as Open Street Maps (OSM)[1] or Google Maps[2]. The model used must allow large scale simulations with millions of vehicles.

Trips Definition: It is necessary to define all the trips that will be performed during the simulation. To implement this we have two alternatives: creating a tool to generate random trips or converting an origin-destination matrix (when available) for the city we intend to simulate.

Vehicles Simulation: All traffic simulators must use a car model to calculate the speed of the cars. There are many models available in the literature, from the simple free model to complex models.

Output Generation: The simulator must generate different outputs to allow the analysis of the results of the simulation. The most common approach is to generate a file with all the events occurred in the simulation, allowing the development of visualization and statistical tools.

Besides the functional requirements, a Smart City simulator also must meet the following non-functional requirements:

Scalability: To simulate Smart City scenarios, it is necessary to manage millions of actors such as cars, people, buildings, and sensors. Therefore, the simulator scenarios have to scale from hundreds to millions of actors. To achieve this, distributed and parallel simulations are almost mandatory.

Usability: Creating descriptions of simulated scenarios for the simulator should be easy, enabling people with no knowledge of the internal implementation of the simulator to develop scenarios with little effort. Thus, the programming model has to be intuitive and independent of the internal implementation of the simulator.

Extensibility: It is unlikely that a simulator will provide all required features for Smart City simulations. The simulator has to be easily extensible, offering simple mechanisms for implementing new actors and changing their behavior, for implementing new metrics, and for the modification of the behavior of the simulator itself. So, it is important not only that the simulator be open source, but also well documented and implemented with high quality, extensible code.

3 Related Work

In our literature and Web searches for Smart City simulators, we did not find any simulator that is capable of simulating large-scale and complex scenarios with multiple actors such as cars, buildings, people, and sensors. We included in this section software that simulates individual agents, such as cars or people, and cites the development of large-scale simulations as one of its objectives.

[1] Open Street Maps — `http://www.openstreetmap.org`
[2] Google Maps — `http://maps.google.com`

DEUS (Discrete-Event Universal Simulator) is a discrete-event general purpose simulator, which was used to simulate a Vehicular Ad-Hoc Network (VANET) [7]. In this Java-based, open-source simulator, it is possible to extend the base Node and Event model to implement particular actors to simulate entities such as cars, buildings, people, and sensors. Due to its architecture and non-parallel Java implementation, however, its scalability is weak, which we verified by experiments with almost 10 thousand nodes that we carried out.

Veins is a VANET simulator that integrates [8] OMNET++[3], a well-known discrete-event network simulator, and SUMO (Simulation of Urban Mobility)[4], a microscopic traffic simulator. In Veins, it is possible to simulate traffic scenarios such as traffic jams and accidents. In our experience with it, it was difficult for us to understand the code and architecture and running it in parallel mode was not trivial.

Siafu is a Java agent-based, open-source simulator [9] used to simulate mobile events in a city. The simulator has a user interface to visualize simulation data and can export data sets. In Siafu, the agent creation is manual, so it is more appropriate for small, simple scenarios that can be visualized via a simple graphical representation of the city.

MATSim is also a Java agent-based, open-source simulator [5] that provides a large variety of tools to aid in the development of traffic simulations such as an Open Street Maps converter, a coordinate system converter, and a map editor. Balmer et al. [10] show that MATSim can scale to almost 200 thousand agents. However, due to its architecture and Java implementation as well as the lack of a distributed implementation, it does not have the necessary scalability to simulate an entire city.

Mezzo is a mesoscopic[5] traffic simulation model suited for the development of integrated meso-micro models [6]. Mezzo's most important feature is the output format, which allows easy construction of microscopic simulations after the execution of the mesoscopic simulation. We have not found any information about its implementation. However, tests presented in the paper describing it show just small simulations.

Song et al. [19] implemented a mesoscopic traffic simulator using GPUs (Graphical Processing Unit). Their objective is to use the great computational power of GPUs to process large-scale traffic scenarios in high speed. The results showed a speedup of two times comparing a GPU and a C implementation. However, they found two problems: the communication between the GPU and the CPU is a bottleneck and, normally, the memory of GPUs is small. Both are problems in the simulation of big scenarios.

None of the aforementioned simulators can scale to an entire metropolitan area with a map with thousands of streets and millions of vehicles moving in the

[3] OMNET++ — `https://omnetpp.org`

[4] SUMO — `http://sumo.dlr.de`

[5] Mesoscopic Traffic Models simulate each vehicle in transit, but with fewer details than a microscopic model. They often use a density function to determine the vehicle's speed in a street.

city. All the simulators are implemented in Java or C++, languages in which the development of parallel and distributed applications is not transparent. Hence, the use of a language better suited for the simple development of parallel and distributed applications can enable the development of very-large-scale traffic simulators.

4 Actor Model

The Actor Model is a powerful model for the development of highly concurrent, distributed software. In this model, each actor is a processing unit, and they can communicate only using asynchronous messages. Each actor has a mailbox which stores the messages until the actor processes them. After processing a message, an actor can change its state, send other messages, or create new actors.

This model diminishes two great problems of concurrent systems: race conditions, as the actors do not share state or resources, and blocking waits, as all the messages between actors are asynchronous. Although the actor model is not a new idea [20], this model is gaining popularity in the last years because of multi-core architectures.

As with the implementation of concurrent applications, the development of distributed software is also very straightforward because there is no difference if two actors are executing in the same or different machines. The unique requirement is that the language based on the Actor Model has to implement a communication model that allows the message exchange of actors running on different machines. Currently, many languages are based on the actor model such as Erlang and Scala [18], and many others have an actor implementation such as Ruby[6], and Java[7].

4.1 Erlang

Erlang is a functional programming language based on the Actor Model developed mainly for the implementation of large-scale, distributed, parallel applications. It was created by Ericsson[8] for use in the development of telecommunication applications. Currently, the language is used in various domains such as Internet communication[9], database systems [13], and simulators [12, 11].

Most of Erlang characteristics, inherited from the Actor Model, are suitable for the development of large-scale simulators:

Parallelism: The Erlang Virtual Machine allows the creation of a massive number of system threads. In the Erlang programming model, each thread is an actor that can execute functions independently and spontaneously or when it receives a message from another actor.

[6] Celluloid - https://celluloid.io/
[7] Reactors.io - http://reactors.io/
[8] Ericsson — `https://www.ericsson.com/`
[9] WhatsApp - `https://goo.gl/If6k3d`

Distribution: In the Erlang actor model, it makes no difference whether two actors that need to exchange messages are running on the same or different machines. Therefore, the distribution of Erlang applications is very simple and almost transparent to programmers. The unique requirement is the creation of a text file with all the machines where Erlang actors can be deployed.

Fault Tolerance: Each actor in Erlang is independent of the others; therefore an error in an actor does not propagate to the rest of the application.

Communication Protocol: Erlang processes communicate only through messages, which is very useful in the development of parallel applications because that minimizes the necessity of mutual exclusion algorithms.

The Erlang language is frequently used to implement multi-agent systems. The actor model has many similarities with the idea of agents, such as communication mechanisms, multi-thread features, and fault-tolerance [14]. Moreover, each actor can have many different actions triggered by an event that can be the receiving of a message or a timeout. McCabe et al. [17] present a comparison of nine languages used to develop multi-agent systems; Erlang had the third best results, just after OpenMP and C++, but both are low-level languages, making it harder to implement parallel and distributed simulators.

The main Erlang disadvantage for the implementation of a simulator is thread synchronization: because each thread is independent of each other, it is impossible to know the order of the thread execution. Therefore, it is necessary to implement a mechanism to synchronize the execution of the actors. Another problem is the scarcity of proper tools for the development of Erlang applications, such as Integrated Development Environments and testing tools.

In the development of InterSCSimulator, we used Sim-Diasca (Simulation of Discrete Systems of All Scales) [12], a general purpose, discrete-event simulator developed in France by the EDF energy company[10] that has the goal of enabling very large-scale simulations. This simulator is implemented in Erlang, allowing the implementation of massively parallel and distributed simulations. Moreover, Sim-Diasca has a simple programming model enabling fast development of simulation scenarios. Our experiments with Sim-Diasca demonstrated that it scales much better and is much easier to use and extend that the other simulators mentioned in Section 3.

5 InterSCSimulator

InterSCSimulator is an Open-Source, scalable, Smart City simulator that has the objective of simulating various, complex, and large-scale Smart City scenarios. This section presents the implementation of the first version of the simulator that already simulates traffic scenarios with cars and buses. The simulator is implemented on top of Sim-Diasca and has all the advantages mentioned above related to the use of the Erlang language.

[10] EDF — `https://www.edf.fr/content/sim-diasca`

Figure 1 presents the simulator architecture. The bottom layer is the Sim-Diasca simulator, responsible for the discrete-event simulation activities such as Time Management, Random Number Generation, Deployment Management, and the Base Actor Models. The middle layer is the Smart City Model, which we developed as part of our research and implements the required actors for traffic simulations such as cars, buses, and the streets that represent the city graph. The top layer comprises the scenarios that can be implemented using the Smart City model.

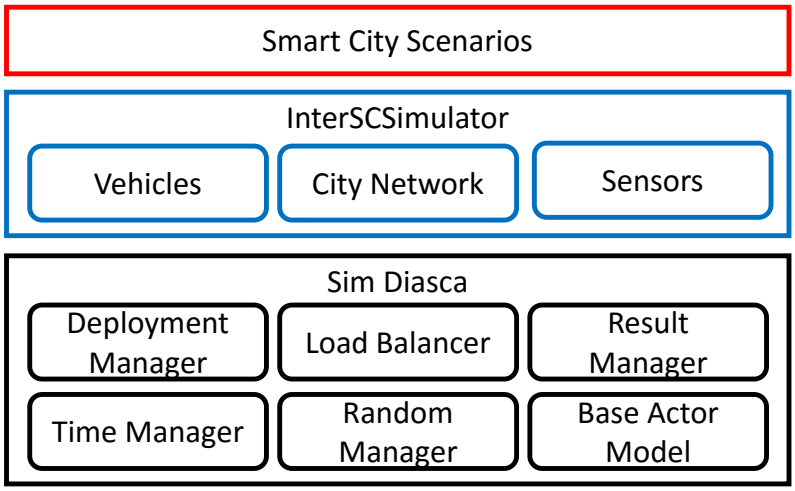

Fig. 1. InterSCSimulator architecture

5.1 InterSCSimulator Components

The InterSCSimulator has four main components: the **Scenario Definition** that receives the input files and creates the city graph and first vehicles; the **Simulation Engine** that executes the simulation algorithms and models and generates the simulation output; the **Map Visualization** that receives the simulation output and creates a visual visualization of the city map and the movement of the vehicles; finally, the **Chart Visualization** that also receives the simulation output and generates a series of charts with information about the simulated scenario. Figure 2 presents the components and their interactions with their inputs and outputs.

5.2 Inputs

InterSCSimulator uses three XML files as inputs. The first, map.xml, is the description of the network of a city. This file can be generated from a region

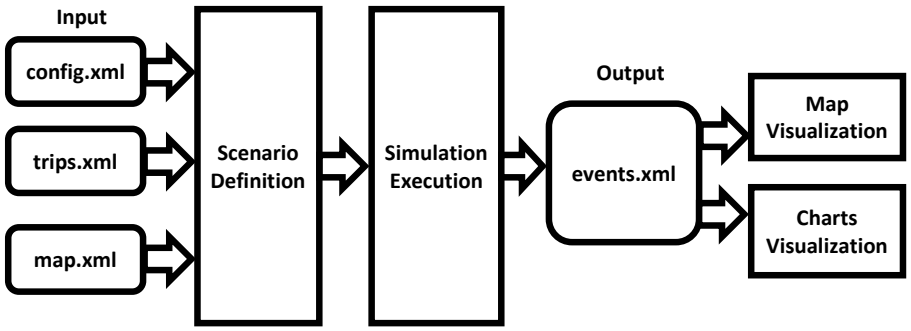

Fig. 2. InterSCSimulator Components

in Open Street Maps (OSM) using a tool that converts the OSM format to an oriented graph using the Erlang's Digraph API. We tested this tool with very large maps such as the entire São Paulo metropolitan area that has more than 80 thousand nodes and 120 thousand links. Listing 1 presents an example of a map file with 3 nodes and 3 links.

Listing 1. XML file with the city network

```
<network>
  <nodes>
    <node id ="1"
    x="-46.65805" y="-23.58162" />
    <node id ="2"
    x="-46.65828" y="-23.58342" />
    <node id ="3"
    x="-46.65228" y="-23.59341" />
  </ nodes>
  <links>
    <linkid="35985" from="1" to="2"
    length="100" freespeed="40" />
    <linkid="35985" from="2" to="3"
    length="200" freespeed="40" />
    <linkid="35985" from="3" to="1"
    length="80" freespeed="50" />
  </ links>
</ network>
```

The file is divided into two sections. The first section describes the nodes of the graph which are street crossings in the city map; the second section contains all the links which represent stretches of the city streets. Note that many links can represent a single street.

The second XML file has all the trips that must be simulated. Each trip has the origin and destination nodes in the graph and the simulation time when the trip will start. Optionally, the trip can have a fixed path, mainly to simulate buses, or it may be up to the simulator to calculate the best path from the origin to the destination (using algorithms of the Erlang Digraph API[11]). Listing 2 presents a stretch of the trip file.

Listing 2. XML file with the trips to simulate

```
<scsimulator_matrix>
    <trip origin="247951669" destination="60641382"
        type="car" start_time="28801" />
    <trip origin="60641382" destination="247951669"
        type="car" start_time="63001" />
    <trip origin="4511105625" destination="2109902387"
        type="car" start_time="16201" />
    <trip origin="247951669" destination="60641382"
        type="car" start_time="54001" />
    <trip origin="246650787" destination="247951670"
        type="car" start_time="54001" />
    <trip origin="247951670" destination="246650787"
        type="car" start_time="66601" />
    <trip origin="246650787" destination="60641382"
        type="car" start_time="54001" />
</scsimulator_matrix>
```

Finally, the third file contains some important parameters to the simulation such as the total time of the simulation, the path to the map and trip files, the output file path, and the charts that have to be generated at the end of the simulation.

With the three files loaded (map, trips, and configuration), a Simulation Scenario is created. This component is responsible for the creation of all Erlang actors necessary for the simulation. Each vehicle (car or bus) is an actor, and each vertex of the city graph is also an actor that knows all its immediate neighbors. The vehicles are active actors that periodically send messages to some city vertex; these, in turn, are passive actors.

5.3 Simulation execution

In this first version of InterSCSimulator, the Vehicle actor is the main agent of the simulation. This actor can be a car or a bus moving in the city from an origin to a destination vertex in the city graph. Currently, we do not try to check if a single car performs more than one trip throughout the simulation nor do we try to handle individual passengers, which might use more than one Vehicle in a single trip. This actor has four main behaviors: it may **Start Travel**, when the simulation reaches the start time for the vehicle; **Move**, when the simulation

[11] Erlang Digraph API — `http://erlang.org/doc/man/digraph.html`

reaches the time of the next movement for the vehicle; **Wait**, when the vehicle has to wait until its next move action; and **Finish Travel**, when the car arrives at its destination. One agent is created to simulate each trip in the trips input file.

Another important actor is Street, which represents each vertex of the city graph. This actor knows its neighbor nodes and the links that connect them. At each movement, a car asks the vertex what is the link that it has to use to follow in its path. The street actor answers with the link and the time the car will take to cross the link. Then the car waits until its next movement. This distributed model of the city graph and the fact that all message exchanges are local allow the simulator to scale very well, as there is no central actor that manages the city graph, which would be a bottleneck.

In this first version we use a very simple free-flow model to calculate the time that a car will spend in a link: $time = link\_length/vehicle\_speed$. Each link stores the number of cars that are in the street at each moment. If the number of cars in the link is equal to its capacity, then no vehicle can enter the link until at least one car leaves the street. If this happens, then there is a traffic jam in the simulation. We already save the number of cars in the links in each moment to allow the future development of more complex models.

InterSCSimulator can use any map collected from OSM. Figure 3 presents an execution of the simulation using the map of São Paulo. This map has approximately 50 thousand vertices and 120 thousand links; in this simulation, 500 thousand trips with 250 thousand actors were used in a one-day simulation. Each actor goes to work and goes back home at random times. In this graphic visualization, we used OTFVis[12], a visualization tool developed as part of the MATSim project.

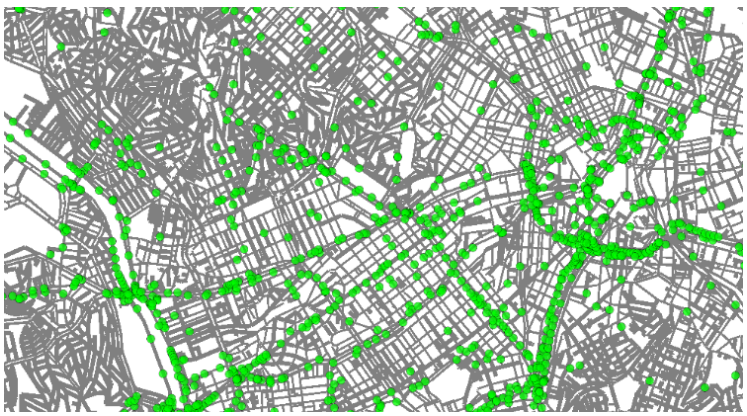

Fig. 3. Simulation execution over the São Paulo street map

[12] OTFVis — `http://matsim.org/docs/extensions/otfvis`

5.4 Outputs

The InterSCSimulator generates an XML output file with all the events that occurred during the simulation. We used the same format as MATSim, which allows us to use OTFVis and other MATSim tools. Listing 3 presents a segment of the output file with the events of two cars saved in an example simulation. The events stored in the file are the same described in listing 2.

Listing 3. Simulation events file

```
<events version="1.0">
    <event time="4" type="start_trip"
    person="2121" link="5243" legMode="car"/>
    <event time="4" type="start_trip"
    person="2223" link="1002" legMode="car"/>
    <event time="11" type="move"
    person="2223" link="4005" />
    <event time="31" type="move"
    person="2121" link="4005" />
    <event time="38" type="move"
    person="2223" link="2007" />
    <event time="52" type="finish_trip"
    person="2121" link="4005" />
    <event time="52" type="finish_trip"
    person="2223" link="5243" />
</events>
```

Besides the file, we also created a service that runs R scripts to make statistical analyses with the data generated at the end of the simulation. These scripts produce a series of charts such as the most used links during the simulation and the biggest trips of the simulation. Figure 4 shows a graph produced by these scripts showing the ten most used links during a simulation.

6 InterSCSimulator Evaluation

To evaluate the simulator we tested mainly the scalability, which is our most important feature in comparison to other simulators. We also present some remarks about the usability of the simulator, which is important because people that will use this kind of simulator may not be computer specialists, such as city and traffic managers and traffic engineers.

6.1 Scalability

To test the simulator scalability, we created a scenario based on an Origin-Destination (OD) matrix produced by the subway company of São Paulo[13]. The

[13] Origin-Destination Survey — `https://goo.gl/DNM8in`

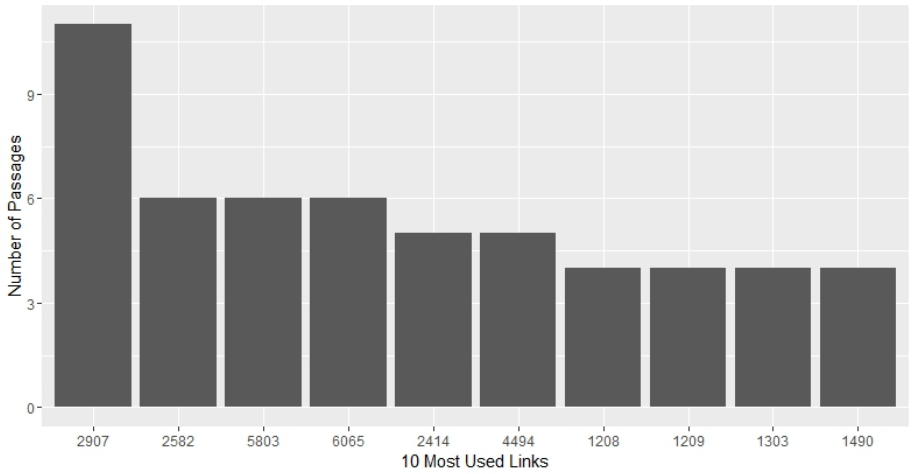

Fig. 4. Top 10 Most Used Links During the Simulation

OD matrix has 170 thousand trips of people in the city, mainly going to or coming back from work. We extrapolated the trip data in this matrix (by simple replication) to create four different synthetic scenarios: 1 million, 2 million, 3 million, and 4 million trips. All the scenarios simulate an entire day in the city. Most of the trips take place during the peak hours in the morning (07:00 to 09:00 am) and in the afternoon (05:00 to 07:00 pm).

The tests showed that the simulator scales almost linearly with the number of agents. Figure 5 shows two charts, the first with the execution time of the four scenarios (in minutes) and the second with the total amount of memory used (in gigabytes) in the four scenarios. All scenarios were executed on a machine with 24 cores and 200 GB of memory. It should be possible to run the same simulations in a distributed system using a group of machines with more modest resources each.

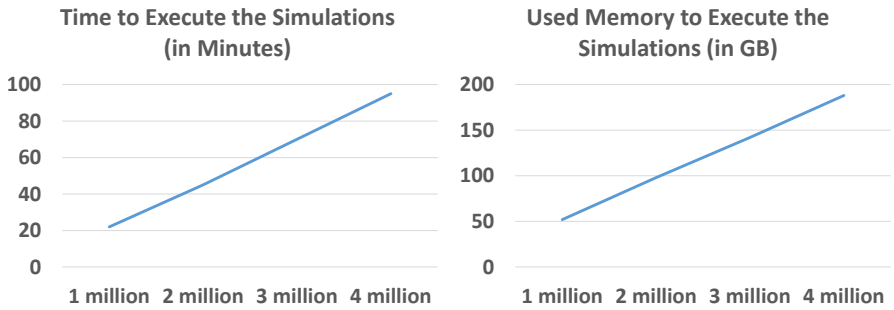

Fig. 5. Execution time and memory used in the four scenarios

Very large simulations usually take many hours or days to complete very large scenarios. With InterSCSimulator, we were able to complete all the four tested scenarios in a comparatively short time: the first scenario simulation took just 22 minutes, the second 45 minutes, the third 70 minutes, and the fourth 95 minutes. All scenarios simulated 24 hours of city traffic and the simulation time grows linearly with the number of simulated agents. The second chart shows that the memory usage growth was also almost linear in the four scenarios. Comparing with the simulators presented in section 3, only the work of [19] has comparable execution times.

Based on the data collected in the four scenarios, we used a linear regression algorithm to estimate the necessary resources to simulate all of the city of São Paulo, currently with 11 million inhabitants, which is our final goal. Table 1 compares the scenarios and shows estimations to simulate the entire city.

	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Estimation
Agents	1 million	2 million	3 million	4 million	11 million
Map Size (Nodes)	50.000	50.000	50.000	50.000	140.000
Memory	51 GB	98 GB	142 GB	196 GB TB	515 GB
Time (Minutes)	22 m	45 m	70 m	95m	480 m
Events	70 million	140 million	210 million	280 million	910 million
Output File Size	2 GB	4.1 GB	6 GB	8.3 GB	23 GB

Table 1. Simulated Scenarios and Estimation

The table compares the following characteristics:

Agents: The number of simulated agents in each scenario and the total number of agents to simulate the entire city.

Map Size: The number of nodes in the city graph. In this version, we used just the map of São Paulo, but to simulate all of the city it is also necessary to include parts of the extended metropolitan area.

Memory: The maximum amount of memory used in the simulation and the necessary memory estimated to simulate the entire city.

Events: The number of events that occurred during the simulations and the estimated number of events to simulate the entire city. These events are saved in the output file.

Output File Size: The final size of the file that stores all the simulation events and the estimated size of the file to simulate the entire city.

This data suggests that, if the simulator indeed continues to scale linearly to bigger scenarios, it will be possible to simulate the entire city. We also made some preliminary distributed tests with the simulator on a basic machine. We created three containers using Docker 9 in a machine with 6 cores and 16 GB of memory. As mentioned in subsection 4.1, distribution is one of the main Erlang characteristics, and it is very straightforward to execute distributed Erlang applications. Figure 6 compares the same simulation running using one, two, and

three containers. The chart shows that, as the number of containers grow, simulation time decreases. We have to investigate further why this happens and also what is the impact of communications among the containers in the simulation.

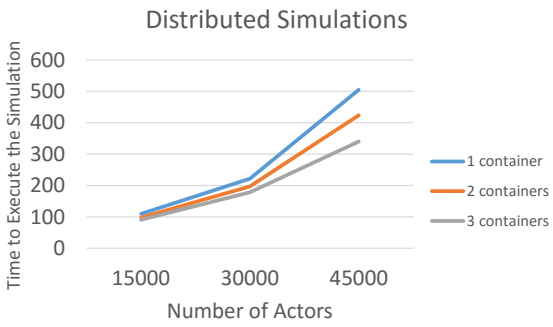

Fig. 6. Execution time of the distributed simulations

6.2 Usability and Extensibility

To verify usability, it is important to analyze how to create the Smart City scenarios. We based our model on MATSim, which requires the creation of a map, a trip, and a configuration file to create the simulation. The map file is based on Open Street Maps, the trip files can be created manually or parsed from an OD matrix, and the configuration file is very simple, with just some options as the input and output folder and the total time of the simulation. Both InterSCSimulator and MATSim provide tools for the creation of these files.

Veins and DEUS have a similar way of defining scenarios using XML files that describe the initial actors and their behavior, but they do not provide any additional tool to facilitate the development of the scenarios. Siafu has a visual interface to define the scenarios, which is good and easy when dealing with small simulations but makes the creation of large simulations with many actors impractical.

Also, extensibility is necessary to allow other researchers to change and add simulation models. InterSCSimulator and DEUS have a very similar programming model. Both provide a base class (Actor in Sim-Diasca and Node in DEUS) that developers can extend to implement the simulation actors. MATSim has many interfaces that new models can extend changing the behavior of the simulation. The Siafu programming model is a little different, and the programmer has to understand all the code of the simulator to use it. In Veins, adding new components to the simulator depends on changing OMNET++ and SUMO and its communication. Therefore, InterSCSimulator, MATSim, and DEUS seem to be more easily extensible than Siafu and Veins.

7 Conclusions

This paper described the development of InterSCSimulator, a simulator that aims to advance the state of the art in the integrated simulation of Smart Cities, offering scalability and a straightforward programming model. In this first version of the simulator, we implemented actors for the simulation of traffic scenarios. The experiments showed that the simulator is scalable, a fundamental requirement to simulate large traffic scenarios; it is reasonable to expect similar performance in other domains. Compared to other simulators, InterSCSimulator is also easy to use and makes it possible to generate charts and an animated simulation with a GUI using the results of the simulations.

We also developed tools to aid in the creation of real scenarios such as an Open Street Maps parser and a parser to read the São Paulo OD Matrix. In our ongoing work, we are experimenting with larger scenarios, going up to the entire vehicle fleet of an enormous city with 11 million inhabitants. To do that, we need to execute the simulator both in larger machines with more cores and in a distributed environment, exploring the parallelism supported by the Actor model of Erlang. However, we anticipate that we will need to address several challenges and bottlenecks before we can achieve that, such as the size of the output file, the maximum number of supported actors in an Erlang virtual machine, and the communication costs in distributed environments.

As future work, we intend to implement other Smart City scenarios such as disaster management and smart grids. We also plan to make large-scale distributed simulations, since we only tested the distribution model of Erlang in small scenarios. Finally, we plan to perform a functional evaluation with city officials and public policy makers to validate the simulated scenarios and improve the simulator usability.

Acknowledgments. This research is part of the INCT of the Future Internet for Smart Cities funded by CNPq, proc. 465446/2014-0, CAPES proc. 88887.136422/2017-00, and FAPESP, proc. 2014/50937-1 and was partially funded by Hewlett Packard Enterprise (HPE).

References

1. Caragliu, A., Del Bo, C., Nijkamp, P.: Smart cities in Europe. Journal of urban technology. 18, 65–82 (2011)
2. Sanchez, L., Muñoz, L., Galache, J. A., Sotres, P., Santana, J. R., Gutierrez, V.,Pfisterer, D.: SmartSantander: IoT experimentation over a smart city testbed. Computer Networks. 61, 217–238. (2014)
3. Zanella, A., Bui, N., Castellani, A., Vangelista, L., Zorzi, M.: Internet of things for smart cities. IEEE Internet of Things Journal. 1, 22–32 (2014)
4. Grimaldi, D., Fernandez, V.: The alignment of University curricula with the building of a Smart City: A case study from Barcelona. Technological Forecasting and Social Change. (2016)

5. Horni, A., Nagel, K., Axhausen, K. W.: The multi-agent transport simulation MATSim. Ubiquity. 9 (2016)
6. Burghout, W., Koutsopoulos, H. N., Andreasson, I.: A discrete-event mesoscopic traffic simulation model for hybrid traffic simulation. IEEE Intelligent Transportation Systems Conference. (2006)
7. Picone, M., Amoretti, M., Zanichelli, F.: Simulating smart cities with DEUS. International ICST Conference on Simulation Tools and Techniques. (2012)
8. Darus, M. Y., Bakar, K. A.: Congestion control algorithm in VANETs. World Applied Sciences Journal. 21, 1057–1061 (2013)
9. Nazário, D.C., Tromel, I.V.B., Dantas, M.A.R. and Todesco, J.L., 2014, June. Toward assessing quality of context parameters in a ubiquitous assisted environment. IEEE Symposium on Computers and Communication (ISCC). (2014)
10. Balmer, M., Meister, K. and Nagel, K.: Agent-based simulation of travel demand: Structure and computational performance of MATSim-T. ETH Zürich, IVT Institut für Verkehrsplanung und Transportsysteme. (2008)
11. Toscano, L., D'Angelo, G., Marzolla, M.: Parallel discrete event simulation with Erlang. ACM SIGPLAN Workshop on Functional high-performance computing. (2012)
12. Song, T., Kaleshi, D., Zhou, R., Boudeville, O., Ma, J.X., Pelletier, A. and Haddadi, I.: Performance evaluation of integrated smart energy solutions through large-scale simulations. Smart Grid Communications. (2011)
13. Anderson, J.C., Lehnardt, J. Slater, N.: CouchDB: the definitive guide. O'Reilly Media, Inc.(2010)
14. Di Stefano, A., Santoro, C.: eXAT: an Experimental Tool for Programming Multi-Agent Systems in Erlang. WOA (2003)
15. Varela, C., Abalde, C., Castro, L. Gulias, J.: On modeling agent systems with Erlang. ACM SIGPLAN Workshop on Erlang (2004)
16. Krzywicki, D., Stypka, J., Anielski, P., Turek, W., Byrski, A. Kisiel-Dorohinicki, M.: Generation-free agent-based evolutionary computing. Procedia Computer Science. 29, 1068–1077. (2014)
17. McCabe, S., Brearcliffe, D., Froncek, P., Hansen, M., Kane, V., Taghawi-Nejad, D., Axtell, R.: A Comparison of Languages and Frameworks for the Parallelization of a Simple Agent Model. Multi-Agent-Based Simulation (MABS) Workshop. (2016)
18. Tasharofi, S., Dinges, P. and Johnson, R.E.: Why do Scala developers mix the actor model with other concurrency models?. In European Conference on Object-Oriented Programming. Springer Berlin Heidelberg. (2013)
19. Song, X., Xie, Z., Xu, Y., Tan, G., Tang, W., Bi, J. and Li, X.: Supporting real-world network-oriented mesoscopic traffic simulation on GPU. Simulation Modelling Practice and Theory, 74, pp.46–63. (2017)
20. Agha, G.A.: Actors: A model of concurrent computation in distributed systems. Massachusetts Institute of Technology. (1985)