Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Christian Hüning, christian.huening@haw-hamburg.de

Analysis of Performance and Scalability of the Cloud-Based Multi-Agent System MARS

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Christian Hüning, christian.huening@haw-hamburg.de

Analysis of Performance and Scalability of the Cloud-Based Multi-Agent System MARS

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thiel-Clemen
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 27. Juni 2016

Christian Hüning, christian.huening@haw-hamburg.de

Thema der Arbeit

Analysis of Performance and Scalability of the Cloud-Based Multi-Agent System MARS

Stichworte

Multiagentensystem, Verteilte Systeme, Lastverteilung, MMAS, MSaaS, Ökologische Modellierung

Kurzzusammenfassung

Agentenbasierte Simulationen werden heutzutage intensiv in einem breiten Feld verschiedener Domänen eingesetzt. Anwendungen stammen etwa aus den Bereichen der sozialen Wissenschaften, der Biologie, Ökonomie sowie der Logistik. Diese Domänen sind besonders für die individuen-basierte Modellierung großer Agentenmengen prädestiniert, um deren emergentes Verhalten einzufangen. Das MARS LIFE System als Simulationsengine des Modeling and Simulation as a Service Systems MARS wird in dieser Arbeit präsentiert und hinsichtlich seiner Fähigkeiten in Bezug auf Skalierbarkeit und Performance analysiert.

Christian Hüning, christian.huening@haw-hamburg.de

Title of the paper

Analysis of Performance and Scalability of the cloud-based multi-agent system MARS

Keywords

Multi-Agent-System, Distributed Systems, Load Balancing, MMAS, MSaaS, Ecological Modeling

Abstract

Agent-based simulations are intensively used in a wide ranging variety of domains nowadays. Applications originate from the domains of social science, ecology, biology, economy and logistics to just name a few. These fields are especially predestined for individual-based modeling of large amounts of agents in order to capture emergent behaviour. The MARS LIFE system as the actual simulation engine behind the Modeling and Simulation as a Service system MARS, is presented in this work and analyzed towards its scalability and performance capabilities.

Contents

Acknowledgement

I would like to express my gratitude to Professor Dr. Thomas Thiel-Clemen for providing me his trust, the freedom and required assets to cooperatively build the MARS System and the associated project. His critical feedback, useful comments and engagement throughout the process were priceless.

Also I wish to thank Prof. Dr. Stefan Sarstedt for his continuous inspiration and support to try new technologies and guiding me through the process. Our discussions and conjointly conducted experiments were invaluable.

I would also like to acknowledge the hard work of the MARS team without which the whole project would not have come alive and the days in the research lab would not have been so much fun.
I further like to express my thanks to the colleagues from the computer science lab, who were very supportive and always ready to lend an ear to me and my problems.

Finally I must express my very profound gratitude to my family for providing me with unfailing support and continuous encouragement throughout my years of study. Special thanks belong to my partner Wiebke Klare for her continuous support with delicious brain food, for occasionally dragging me off work whenever I needed it and for patiently listening to all the computer science stuff at the dinner table. Thank you.

1 Introduction

Agent-based simulations are intensively used in a wide ranging variety of domains nowadays. Applications originate from the domains of social science, ecology, biology, economy and logistics to just name a few.

The MARS research group situated at the University of Applied Sciences in Hamburg currently has a focus on building models from the socio-ecological and biology domains. These fields are especially predestined for individual-based modeling of large amounts (millions) of agents in order to gain additional knowledge about the multitude of interrelations these complex systems imply.

In terms of computer science the creation of a system capable of simulating millions of independent software entities, using big data as input and analyzing large amounts of output is clearly positioned in the field of complex and large-scale distributed systems. This work describes the multi-agent simulation system MARS LIFE, which has been created by the author over the past two and a half years. Also MARS LIFE will be put in the wider context of the Modeling and Simulation as a Service system MARS in which it is embedded to outline the relevance of the overall solution.

IBM, ABM and Multi-Agent Systems

It is important to clarify the usage and meaning of certain terms when talking about simulation systems and modeling. The overarching topic is titled as agent-based computing (Jennings, 1999; Wooldridge, 1998) and describes the agent-based endeavor as a whole. It includes everything agent related and spans a lot of functional domains from industrial automation all the way to ecological simulation.

In general agent-based computing is understood as a way to translate real-world problems into the computing space. Agents are seen as the potential solution to several problems on differing levels of abstraction between rather technical and practical software engineering and

conceptual modeling alike. For instance Wooldridge (1997) proposes the usage of agents as building blocks of heavily distributed systems, while Russell & Norvig (1995) understand agents as an indispensable concept to develop intelligent entities in general. The term multi-agent system thus does not describe a system which is used to gain knowledge about complex real world systems or similar questions, but a method to design agent-based systems and solve the related engineering problems (Niazi & Hussain, 2011).

The two major terminologies regarding the types of models, which are simulated with multi-agent systems are Agent-Based Modeling (ABM) and Individual-Based Modeling (IBM).
Agent-based modeling is dated back to the early days of von Neumann's self-replicating automata (Neumann, 1966) and mathematical constructs like Conway's Game of Life (or simply Life) (Gardner, 1970). The model of segregation by Thomas Schelling (Schelling, 1969) is considered as one of the first agent-based models as he used the basic concepts of individual agents acting in an environment and showing emergent behavior. Another early example showcasing the multi-domain application of ABM is Robert Axelrods book titled "The complexity of cooperation" (Axelrod, 1997), where he utilized ABM in the social sciences.

The term individual-based modeling was coined for ecology by Grimm & Railsback (2005). Their understanding is that an agent should solely represent an individual being in contrast to the ABM concept where an agent might just as well be a complete system, a piece of hardware or anything else that is allowed to be autonomous and act independently in its environment. Therefore IBM can be seen as a specialization of ABM, that is widely used in ecology and adjacent scientific disciplines. However the definition of 'individual' depends on the modeled scale as well. When modeling on the landscape scale, a city or a community of people might suffice as an individual given the larger scale the model uses, while a model focussing on local or regional agricultural developments will more likely consider single farmers as individuals and thus as agents. This is why the predominant reception of the term refers to the precise modeling of individual entities from the real world rather than to describe the level of scale to look at.

The MARS group currently focuses on ecological models in alignment with the definition of individual-based modeling. The remainder of this work will thus use IBM as the term to describe the MARS models.

Modeling & Simulation as a Service

This work describes the development and investigates upon the scalability and performance of a Modeling & Simulation as a Service (MSaaS) system. It is therefore important to provide context for the terminology of the name and its origins.

An early overview of web based modeling and simulation can be found in Narayanan (2000) but the term MSaaS is just being used by researchers in their publications since the Grand Challenge for Modeling & Simulation workshop in Dagstuhl, Germany in 2012 (Taylor *et al.*, 2012) and has since been investigated on several other important conferences like ACM SIGSIM 2013 (Taylor *et al.*, 2013) or WinterSim 2014 (Taylor *et al.*, 2014; Tolk & Mittal, 2014). The MARS group also made their contribution by describing the term in an early publication about their system (Hüning *et al.*, 2014) and with a recent publication about the current state of the MARS system (Hüning *et al.*, 2016). A specific definition of MSaaS is provided by Cayirci (2013) and follow-up papers (Johnson & Tolk, 2013; Padilla, 2014) express the interest in and necessity of cloud-based model ing and simulation. First implementations of cloud based systems like the large-scale urban systems simulation SEMSim (Zehe *et al.*, 2015) or N2Cloud, a neural-network cloud-based simulation platform by Huqqani *et al.* (2010), are also available.

MSaaS systems are expected to deliver scalable simulation execution by means of a browser accessible user interface suitable for domain experts. The interface should allow to use collected research data like csv files, time-series and GIS files, provide a code-less way of modeling at least for simple models and offer possibilities to explore the result sets created. Data confidentiality is among the most demanded features for both uploaded and created datasets throughout the system. All in all the simulation community hopes for MSaaS to be a high quality, highly available toolkit, which allows to directly start modeling and simulating without the burden of setting up and maintaining large, complex computing resources.

1.1 Why Scale Matters

Getting emergent behavior from a model is one of the main drivers when IBMs are developed. By describing each agent type and individualizing it by using collected or scientifically created datasets during initialization the sum of all agents is capable of producing results not achievable by models which work with stochastic assumptions and large scale aggregations. Or as Gilbert & Bankes (2002) put it: "(...) the ambitions of modelers are constantly rising, and there are problems for which the behavior of one million agents is significantly different from that of 100.".

The MARS group is currently developing a large-scale model form the fields of sociology, ecology and economy with a focus on South Africa. A brief description of an early and simplified version of this model is given in section 4.3. From what the agents do, this model would be classified as a "gap replacement model" by Perry & Enright (2006) as it simulates individuals and their lifecycle in a bounded space. The difference is however that the landscape scale of the MARS model would only be possible in a spatially explicit landscape model (SELM) according to Perry & Enright (2006) as these cover large areas and combine IBM with GIS data. The authors state that it might become feasible to have an IBM on a landscape level because of newly available computability capabilities (page 66).

Another model developed by MARS group member Lukas Grundmann aims to model the human immune system in an individual-based way. This biological model makes a strong case for working on a large scale as is stated by Wendeldorf *et al.* (2011) : "Large scale network models are particularly important for immune simulators to reproduce a true cellular dynamics of a in vivo system where cell concentrations can reach 10^8 / mL (Haase, 1999)".

Additional examples for large-scale models from ecology are presented by Hilbers *et al.* (2015), Childress *et al.* (2002) and Muller *et al.* (2011). Another suggestion for the usage of large-scale agent-based models to help economy policy-makers is presented by Farmer & Foley (2009) and a use case for solving transportation demand issues is provided by Balmer *et al.* (2006). To sum up, the necessity of being able to simulate massive simulation models on varying scales in time and space is clearly evident.

1.2 Hypotheses

MARS has been developed by many people and consists of a growing number of different services, libraries and concepts. For some of which a variety of implementations are available (i.e. the Environment Service Component, ESC). Performing a proper analysis of the overall system requires a breakdown into several smaller hypotheses, which then can be systematically looked into one by one.

Given the amount of virtualization being used in the MARS infrastructure setup (compare chapter 4.1.1) the impact on performance should be considered. The assumption however is that there is no significant overhead regarding CPU and memory performance due to the usage

of modern virtualization technologies and thus the flexibility that comes with virtualization may be utilized.

H1 The flexibility gained by using virtualization technologies like KVM and Docker is not diminishing CPU or memory performance.

MARS LIFE has been developed for modern multi-core machines and should be capable to leverage all available resources of a single machine. If the same model is run on a single machine with more resources, MARS should scale up.

H2 MARS LIFE scales vertically when more CPU cores and/or main memory is available.

Provided that H2 is true, MARS LIFE shall be able to scale across several hosts. Considering the amount of overhead introduced by the necessary serialization and communication, this kind of distribution is only reasonable for a model which completely consumes the resources of a single host.

H3 MARS LIFE scales horizontally when more machines are added to a LIFE cluster and if a single machine is working to capacity with the simulation model at hand.

The original design of MARS LIFE (called "RUN" back then) defined that each agent would use another layer's interface to explore and find agents residing on that layer. During the development of MARS however this concept has been neglected and an environment solution to integrate all agents from all layers has been developed by Florian Ocker. It turned out that this solution is not optimal when used with large agent amounts. Therefore H4 aims at testing the impact on performance of the Environment Service Component in its various possible setups.

H4 Using a polyglot ESC per layer outperforms a centralized ESC in distributed runs.

The AgentShadowing mechanism used in the distribution of layers and agents works with an enhanced version of the classic proxy pattern. An agent has to be resolved against the AgentShadowing service and in case of a remote agent, a proxy is returned instead of a local reference. These proxies may either be pre-initialized during the initialization phase of a simulation start or on-demand when need arises. Though having the system all wired up sounds promising, convenient and fast it can be assumed to require a lot of memory and CPU time to create all proxies in a large simulation.

H5 On-Demand creation of agent stubs is equally or more efficient than a complete pre-initialization.

MARS LIFE features a fully automated model initialization mechanism. The implementation is capable of initializing special layer types like the GIS and time-series layer as well as to create agents based on a configuration, which has been assembled in the Websuite. During the agent creation process LIFE will fetch data from various databases and use C# reflection to analyze the agent type's constructor and call it with the correct set of parameters.

H6 The fully automatic model initialization scales with linear complexity when run on multiple nodes.

1.3 Structure Outline

This work aims on analyzing the performance and scalability of the MARS LIFE simulation system, which is embedded in the overall MARS MSaaS system. To clarify the origins and developments which lead to the creation of MARS and its features Chapter 2 introduces a brief historical overview of IBM with Multi-Agent Systems (MAS) in ecological science, features recent related work in a number of categories, that are relevant to the topic of scalable architectures, provides an overview of system requirements for MSaaS and finally describes the current state of the MARS System. Chapter 3 provides an in-depth presentation of the implemented MSaaS system MARS, the related workflows and used techniques and technology. Chapter 4 describes the infrastructure setup, which has been build and is currently used to run MARS and also details the experiments used to validate or falsify the hypotheses. In chapter 5 results from the experiment execution are presented and chapter 6 finally draws a conclusion and provides an outlook towards future research and development.

2 Methodology

2.1 IBM in Ecology

2.1.1 First Steps

For more than 25 years, ecologists and social scientists among others are busy researching how the usage of multi-agent-systems may help to understand their areas of research more deeply and completely.

Although occasionally explored by others before, Huston *et al.* (1988) where among the first to make a profound statement for the benefits of Individual Based Modeling (IBM) in ecological science. They point out essential rules of biology that were violated by models used so far, with the two major rules being, that every individual is different and that actions between two entities are inherently local.

They further find that those phenomenons cannot be depicted correctly by state-variable models and thus make their point towards IBMs.

2.1.2 Ten years of ecological modelling - A review

Eleven years later Grimm (1999) reviews the evolution of IBM since Huston *et al.* (1988) stated their usefulness and develops "heuristic rules of individual-bases modeling". Grimm (1999) shifts the question from if and why someone should use IBM to how one should develop IBM based simulations. His rules are derived from a review of ecological models created and published between 1988 an 1999 and thus aim to provide a general approach for everyone who wants to create such a model of his own.

His first rule sounds as simple as it is important: "Individual-based modelling is modelling". He wants to make sure, that albeit IBM promises emergent effects from a group of more or less independent agents, this effect does never happen without proper modelling, as this activity means to really understand the problem at hand.

Another rule says that one should "Change the level of aggregation". This refers to the need of using the right level of detail when modelling reality. A too general level could ignore important facts in an underlying level, while a too detailed level could loose focus of the pattern one wants to model. Thinking about these matters leads to the question whether a scale-down or scale-up approach is better suited when designing a new model. Grimm (1999) states that starting with a detailed model and then moving up to a more general one, would produce some quite interesting insights on the different levels of aggregation, though no one would really want to abandon a working detailed model, which is why this scaling-up approach is not really used too often.

Instead the scale-down approach is far more useful as it allows to recreate a general pattern within the simulation and then add detail to it by going down while continuously assuring that the overall pattern is still valid. This also allows to leave each aspect of the model at a different but suitable level of aggregation while going down the hierarchy.

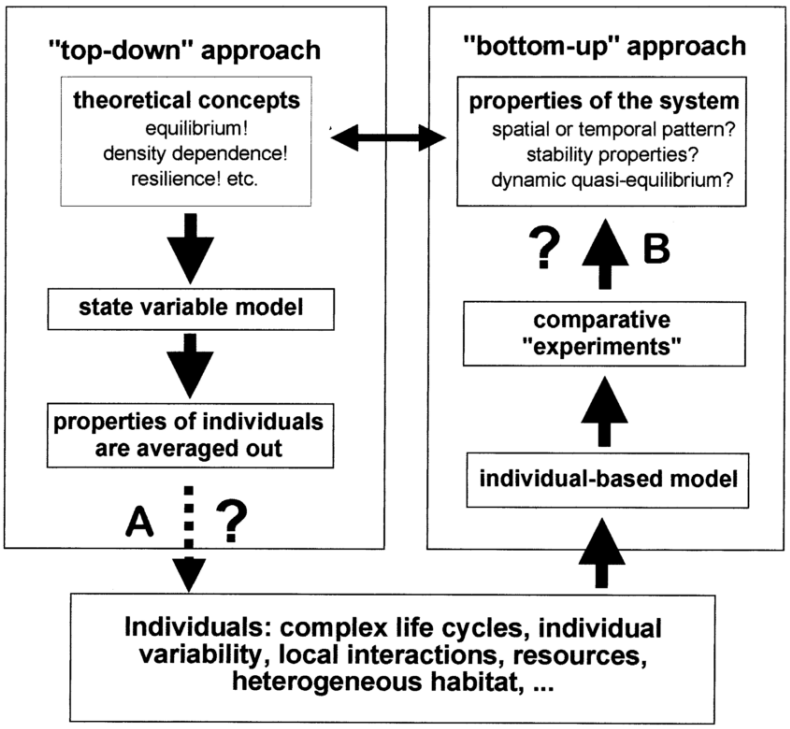


Figure 2.1: Mutual relationship of top-down and bottom-up approaches in ecological modelling. Source: (Grimm, 1999)

Comparing IBM with state-variable modelling means comparing a bottom-up approach with a top-down one. Figure 2.1 shows the mutual relationship between these two complementary approaches as Grimm (1999) sees them. They are complementary because each on its own does not lead to sufficient results. In the top-down approach you make a general assumption on the top level that then will be applied to each individual entity you find on the bottom end. This clearly might lead to wrong results in a multitude of cases. The bottom-up approach on the other hand emerges from individual models for each different entity but might run into trouble if it is not clear what you want to know at the top end. So to avoid both problems it turns out to be a good idea to combine the two approaches by validating the results from bottom-up IBM against the general patterns of top-down state-variable models.

2.1.3 Integrating Models

As more and more ecological models were created and programmed over the years, more and more paradigms and ways of implementation of these models emerged. With that another interesting aspect of IBM came along, the integration of different models. The idea is simple. Connect and integrate domain specific models from domain specific experts to create a new super model of a certain domain. If for example you would want to create a large scale model of the ecosystem of a national park in south Africa, it would be very helpful, if you could use existing models of elephants, cheetahs etc.. Actually doing that, turns out to be much more difficult, since every group of scientists working on a model uses another, individual paradigm, architecture, programming language, data format and so forth. Villa (2001) proposes his Integrating Modelling Architecture (IMA) for hat purpose. He singles out three characterizing dimensions for connecting different models:

Representation A unified semantic relating to the depiction of space, time and behavior in every respective model is needed.

Domain A clear distinction between the domain spaces of each sub-model must be made. In particular this relates to the input and output parameters which are valid for each sub-model.

Scale Data, which is exchanged between models, must be compatible or translated in space and time dimensions.

A recent contribution to the Scale dimension has been made by Thiel-Clemen (2013a), who proposes a data warehouse based information integration process on the simulation data.

These dimensions target the difficulty when technically connecting different models. A more functional view has been made by Liu *et al.* (2007) who take a look at the complexity of coupled human and natural systems. Their integration efforts aim at taking interdisciplinary research on a broader scale into account, as well as exceeding local and temporal boundaries when modelling certain ecological system. As shown by their findings, almost every ecosystem today is tightly coupled with its neighboring economic or social systems and thus these need to be taken into account when watching the evolution of that ecosystem. Filatova *et al.* (2013) move even further and demand that the corresponding aspects of ecological systems like economy, social systems and bio-physical dynamics need to be integrated into the representation of a heterogeneous landscape representation.

2.1.4 Summary

The discussion today circles around the fields of model re-usage (Holst (2013)), model integration (Filatova *et al.* (2013), Le *et al.* (2010), Liu *et al.* (2007), Villa (2001)), which makes distributed, parallel simulation execution (Cicirelli *et al.* (2010), Wang *et al.* (2009), Wang *et al.* (2012), Bellifemine *et al.* (2008), Thiel (2013), Vigueras *et al.* (2013)) necessary and the question of spatial-temporal information integration (Thiel-Clemen (2013a), Filatova *et al.* (2013)) is raised.
Since the above mentioned ideas produce a lot of computing complexity, the need for appropriate simulation tools and frameworks arises. Over the past years there have been quite a lot approaches to this field, which will be further examined in detail in the next chapter.

2.2 Simulation Frameworks

Research in the MARS Group has been focused on large scale models, scalability and MSaaS from the very beginning. Recent publications have shown that there is a growing interest on these topics by other researchers. Therefore an overview of related work in the domains of traditional, MSaaS and high-performance computing is provided.

2.2.1 General Solutions

JADE

One of the most famous frameworks is JADE (Bellifemine *et al.*, 2008) which allows to execute a simulation distributed across several JADE container processes or just locally in a single container. JADE was developed in Java to create a reference implementation of the FIPA agent

specification (http://www.fipa.org). The performance of JADE has been extensively investigated by Mengistu *et al.* (2008). Their findings show that JADE has significant performance issues in the fields of communication and agent migration due to the usage of the LDAP protocol and slow message transport services. JADE's Lookup-Directory-Service also is measured to be slow, which is caused by not using local caching on the respective nodes. Mengistu *et al.* (2008) propose improvements to both mechanisms and present promising results from experiments they conducted. However a more recent investigation of JADE's performance seems appropriate, given that the paper is almost 6 years old.

GAMA

GAMA (Amouroux *et al.*, 2007) is a modeling and simulation framework which is based on RepastJ. It features a nice model description language, called GAML, which allows non-programmers to create complex models. GAMA is written in Java and thus executable on all Java enabled systems. A very strong feature of GAMA is its visualization feature, especially when it comes to using GIS data. An easy import function allows to quickly create a scenario's environment and visualization from a GIS file and thus allows for a quick integration of that kind of data.

The downside of GAMA is, that it's not possible to distribute the system and that it does not scale well across multiple CPU cores. In fact when testing GAMA, it actually used only just up to 4 cores while running on a 24 core machine. While testing GAMA was found to have a performance threshold around 80.000 agents, with one simulation step taking more than 800ms on the aforementioned machine.

WALK

Also from 2013 comes a solution with strong focus on evacuation scenarios which has been developed here at the Hamburg University of Applied Sciences and is called WALK (Thiel, 2013). It features a dynamic (re)partitioning and distribution of agents across several compute nodes and is thus capable of running simulations with hundreds of thousands agents on commodity hardware. In fact Thiel (2013) showed in his final tests that WALK can run a 300.000 agent random walk simulation in near real time. Also remarkable about WALK is, that its agents pass the RiMEA tests and thus provide a pretty good behavior. As a recent addition Stefan Münchow added support for leadership models and social behavior to the agents implemented in WALK. These additions show very promising results and create a very high interest in re-using the leadership concept from WALK whenever human agents are explored.

Vigueras

Another interesting architecture (Vigueras *et al.*, 2013) proposes an almost completely asynchronous, distributed simulation execution to implement interactive simulations, that may be visualized in near real-time. The only time Vigueras *et al.* (2013) synchronize the execution of their agents is, when they happen to act or move beyond the boundaries of their respective environment patch.

When it comes to visualization of the simulation Vigueras *et al.* (2013) utilize visualization nodes (VS) that also act asynchronously on the distributed nodes. Each VS has a camera-style definition of its field of view and may thus only ask those nodes for information containing parts of the environment, which is in that field of view. This is very contrary to other visualization approaches (e.g. GAMA, NetLogo), since it does not attempt to visualize the whole simulation at once.
Considering the amount of agents and the sheer sice of simulated space in our upcoming scenarios, this approach might become very valuable.

2.2.2 Cloud-based Solutions

The term MSaaS as defined by Cayirci (2013) refers to Modeling and Simulation as a Service. Modeling as a Service describes the capability of creating the actual simulation model though the offered service, while Simulation as a Service usually means the service assisted execution and evaluation of simulation models. Another feature of MSaaS solutions is that they are offered though a cloud service. Three recent contributions to that category of MAS are presented below.

mJADES

mJADES (Rak *et al.*, 2012) is a SaaS framework which as a cloud application allows the user to run multiple simulations in parallel. The name of mJADES and its technology is based on the cloud middleware mOSAIC and the simulation library JADES. JADES is implemented in the Java programming language. While MARS is a multi-agent system mJADES uses Discrete Event Simulation as simulation technique.

C^{2}SuMo

C^{2}SuMo (Cloud-based, Collaborative, and Scaled-up Modeling and Simulation Framework for STEM Education) (Caglar *et al.*, 2015) is a SaaS framework for traffic simulations. It uses SUMO,

an open source road traffic simulation package, and enables scalability by employing multiple SUMO simulators in the cloud. Like its name says, C^2SuMo is developed to support education. Therefore it simplifies the SUMO interface to provide a more intuitive way for high school students creating traffic simulations.

SEMSim

There is another SaaS traffic simulation service called SEMSim Cloud Service(Zehe *et al.*, 2015). It is agent-based, web-based, uses cloud computing to execute multiple simulations at the same time, enables multi-core usage and provides a real-time visualization for running simulations. Based on these attributes SEMSim CS exhibits great similarity to MARS. But a main difference consists in the supported simulation domains. SEMSim CS is made for traffic simulations while MARS makes no assumptions regarding the model domain.

2.2.3 High Performance Computing

PDES-MAS

PDES-MAS by Suryanarayanan *et al.* (2013) is short for Parallel Discrete Event Simulation. The whole multi-agent system is modelled from logical processes, which may be distributed across several compute nodes.

PDES differenciates two types of logical processes (LP). Agent Logical Processes (ALP) model the agents' behaviour, whereas Communication Logical Processes (CLP) represent communication and interaction betweend agents. The overall paradigm used to model the latter are Shared State Variables (SSV), which hold all information important to the simulation and are changed concurrently by the agent processes. SSVs reside in the CLPs.

The scalability problem is now solved by arranging the CLPs in a tree of predefined fixed size and with the ALPs as leafs. So each ALP is directly attached to a CLP which allows for a possible colocation of logic and data. At initialization all SSVs are placed in the root node of the CLP tree. As the ALPs start working, the SSVs are being repartitioned to CLPs residing closer to the accessing ALPs. This process is called State Migration by the authors.

All ALPs are executed by a round robin scheduler and manage their own local virtual time (LVT). This local virtual timestamp is used by the SSVs in the CLPs when there is need for a rollback. Also each SSV stores a history of recents changes, mapped by the LVT. Rollbacks are

needed when the tree gets repartitioned and / or messages between nodes get lost or are delayed.

Throughout the tests the authors conducted, it became apparent that there is an optimal number of CLPs for a given simulation and ALP count. If the ALP / CLP ratio is too low, e.g. there are too many CLPs, the overhead of reorganizing and initializing the whole tree becomes too large. If there are too less CLPs on the other hand, the benefit of distribution is lost.

It has to be noted that the concept of PDES-MAS looks rather similiar to that of TupleSpaces / Linda from Gelernter & Carriero (1992) as it implements the concept of a distributed shared memory. This system, though scaling very well, raises questions when it comes to usability. Suryanarayanan *et al.* (2013) don't present a solution for importing data, visalization or an easy enough way to implement a model. The aspect of model reusability could also become very complex as a sub-model is fully integrated with all other sub-models due to the usage of shared state variables. It can be expected to be very difficult to cut out a single sub-model out of the whole set of SSVs.

Repast HPC

Collier & North (2012) present Repast HPC as a distributable fork of RepastJ or Repast Symphony as the latest version of the famous simulation framework is called. Collier & North (2012) motivation to build a large scale MAS is very similar to that of the MARS groups'. That is to allow large-scale model simulation instead of optimizing a smaller-scale model by running many parallel simulations of the same model.

Repast HPC translates models into working simulations through a concept of agents, contexts and projections. A context is a set of agents, whereas the term set corresponds to its mathematical definition. Projections at last use contexts to model the environment. This structure allows for multiple agents to take part in multiple environments, as well as to reuse certain projections.

To distribute a simulation Repast HPC uses a concept called Shared Projections. The environment created by a projection basically is a 2D grid due to the usage of the Logo language. This grid is sliced and then distributed across several processes. The slices are created by means of an influence sphere, which represents the space an agent is or may be active in. To optimize communication a shared grid buffer is attached to each slice. The buffer holds non-local agent stub objects from the neighboring slices and thus allows for changes / interactions to be made locally at first. The system then distributes the changes to the corresponding home objects in

the other processes and takes care of synchronization matters.

Just like Suryanarayanan *et al.* (2013) the work of Collier & North (2012) provides a very scalable solution, which also allows for model reusability through its projections and contexts. However the communication and distribution algorithms rely on a strong localized behavior of agents. It would be interesting to observe performance of the system with a simulation model lacking this feature.

From a usability point of view the communication and synchronisation mechanisms seem to lack usability, since the user has to provide specific pieces of code for each class he wants to take part in it. A more transparent solution would be highly desireable.

Also it must be noted that Collier & North (2012) used high-end super computing hardware (IBM BlueGene cluster with up to 65.536 cores and Infiniband network) which makes it questionable how the system will run and scale on commodity hardware. The latter would also allow smaller research teams to make use of the system.

The intense usage of the Logo language paired with RepastHPC only supporting 2D environments, mark clear restrictions towards the models, which can be implemented. It is rather complex and uncomfortable to map a 3D environment onto a 2D representation. This could be observed during the development of the WALK system (Thiel, 2013). In terms of development the authors provide the information that a skilled developer with good knowledge about both RepastJ and RepastHPC was able to translate an epidemiology model approximately within a week.

Collier & North (2012) do also not address the problem of data import, but it can be assumed that we will see that feature in the near future, since Repast Symphony is pretty strong in that field. The same may be true for the challenge of visualization. While GAMA (a fork of RepastJ by Amouroux *et al.* (2007)) has very good visualization features, RepastHPC currently only supports the creation of a global logfile with results from the simulation.

GSAM

The Global-Scale Agent Model (GSAM) (Parker, 2007; Parker & Epstein, 2011) is a distributed multi-agent system implemented in Java. It was prominently used to simulate an infectious disease model of the H1N1 virus on a global scale with 6.5 billion agents. However this approach

uses a simplified assumption that includes an active-set of agents. Only these agents need to be computed during one execution iteration, since the infection model only features the states of being infected, contagious or neither of both, and thus disables the need to execute the bunch of agents, which are neither infected nor contagious.

The architecture featured in GSAM defines so called ModelBlocks which contain a certain number of agents and cover a certain region. These ModelBlocks may be distributed across multiple compute nodes in advance, and communication between agents in different blocks is implemented by means of Java RMI in conjunction with a bulk communication approach. The bulk communication pattern boosts performance, but also implies to delay messaging up to a certain degree. This is only possible due to the nature of the featured model, since the duration of each agent's state are well known during development and thus can be utilized as a maximum delay time.

The presented system is very impressive in simulating 6.5 billion agents on a global scale within a reasonable amount of time. However the assumptions and optimizations undertaken by Parker & Epstein (2011) are unlikely to be transferable to a lot of other models from differing domains. For example the active-set approach is not eligible in more individual driven models, where it cannot be predicted when, how and by whom the state of a model might change or be changed.

2.2.4 Case Specific Implementations

LUDAS

LUDAS (Land-Use Dynamic Simulator) (Le *et al.*, 2010) implements a social-ecological, land-use/cover change (LUCC) model featuring four components, which implement human population including behavior, the environment, various policy factors with focus on land-use choices and lastly a decision making procedure which integrates the first three features. The model simulates "a watershed in Vietnam for integrated assessments of policy impacts on landscape and community dynamics". The implementation has been done in NetLogo and thus does not provide a very high performance, but showcases the scenario pretty nice.

It is not performance nor distribution which makes LUDAS interesting, but the great integration of LUCC components into a working simulation scenario. If that model can be

translated into a larger, more capable software architecture, it could provide some very decent results in future, larger scale LUCC simulations.

MASE

MASE (Ralha *et al.*, 2013) is another LUCC simulation which targets the development of robust land-use strategies. The showcase features a region called Cerrado in Brazil. Whats remarkable about MASE is, that it utilizes a methodical, empirical parameterization process for human behavior, which has been developed by Smajgl *et al.* (2011). The implementation has been done with JADE (Bellifemine *et al.*, 2008) and Matlab.

2.3 Requirements

This section provides a summary of the most important requirements found for a modern simulation system. Findings from various corresponding work as well as the experience from the MARS project group (http://www.mars-group.org) are also featured.

2.3.1 Modularity and Reusability

As shown (Liu *et al.*, 2007), almost every ecosystem today is tightly coupled with its neighboring economic or social systems and thus these need to be taken into account when watching the evolution of that ecosystem. Filatova *et al.* (2013) go even further by demanding that the corresponding aspects of ecological systems like economy, social systems and bio-physical dynamics need to be integrated into the representation of a heterogeneous landscape representation.

The integration of existing models is one of the most important requirements resulting from this circumstances. This can only be done if models or their parts, are designed in a modular and reusable manner. The idea is to connect and integrate domain specific models from domain specific experts to create a new super model of a certain domain or to reuse sub-models in completely different domains. If for example one would want to create a large scale model of a given ecosystem in south Africa, it would be very helpful, if already existing models of certain components, such as animal behaviors, weather, land erosion and so on could be reused.

Comparison is another aspect that could profit from modular and reusable models. If it was easy to integrate most of the models available, models could be run directly next to each other, consuming the same data, allowing for example to perform real-time digression analyses.

Actually integrating models turns out to be extremely difficult, since each group of scientists working on a model, tends to use another, individual, paradigm, architecture, programming language or data format. A good solution should address this problem.

2.3.2 Information Integration

Data is of huge importance in simulation. It is needed for nearly all tasks from generation of hypotheses, over simulation initialization and calibration to validation. Unfortunately the data that is being collected, has a tremendous heterogeneity in terms of temporal and spatial resolution, reference formats, completeness and error margins. To be viable in a simulation, this data has to be integrated. It must be carefully corrected, the resolutions have to be aligned, the error must be treated.

Furthermore the relevant data of all the available must be singled out and connected. Since the MARS group focuses on spacially explicit simulations, a special point is also to link data without any further reference together to establish a common context. For example we might be designing a model for an animal species in a wildlife reserve somewhere in Africa. For one concrete simulation it could be necessary to include weather data for the whole region, topology data of the general landshape, as well as a rough overview of vegetation types and population metrics for certain species in that area.

A simulation framework should assist domain experts with all the steps involved: GIS imports, data collection, data analysis and possibly transformation. These tasks target the difficulty when technically connecting different models. A more functional view to the importance of information integration has been made by (Liu *et al.*, 2007) who take a look at the complexity of coupled human and natural systems. Their integration efforts aim at taking interdisciplinary research on a broader scale into account, as well as exceeding local and temporal boundaries when modelling certain ecological system.

2.3.3 Scalability

Although it should always be the goal of a modeler, to design everything as simple as possible, some things are inherently computationally intensive. There are several scenarios that, often in combination, prohibit simulation execution on a single computer within reasonable time frames. First of all the agents themselves are becoming more complex, in order to replicate natural behavior. This is especially true for animate objects, such as for example animals or

humans. To come close to the real world, the modeler might need to use computationally expensive techniques, such as learning or planning algorithms, path-finding, collision avoidance and others, often even simultaneously. And the more models are integrated, the more of those techniques are likely to occur.

As the field of multi-agent systems research matures, the applications get also bigger, resulting in a larger number of agents. Imagine for example a continuous field with an average agent density of one agent per square meter accordingly; the system has to handle about 100 agents. Now, if the length of the field's sides is only doubled, the computational effort increases fourfold, in the three-dimensional case even eightfold .

The real world areas of interest are steadily growing larger, further intensifying this problem. This is especially true, when a model is used to forecast future developments of its real world counterpart. Initially mostly used for the understanding of dynamic systems, IBM is likely to be used increasingly for large scale prognosis as well.

Of course it may sometimes be possible to avoid the problem by extrapolating from a sample set of agents to the bigger scenarios. But that would in return diminish the factor that sets apart IBM from other simulation techniques: the ability to track individual agent's actions and states. Also, depending on the system, some desirable emergent properties of the real system are only achievable with a realistic density of agents. For example Yamamoto *et al.*, 2008 found that massively increasing the amount of agents in an auction simulation, significantly changed the outcome of the simulation.

The most promising solution to really solve this problem, is to make the simulation system scalable across multiple computers. Research budgets are not limitless, so it is important to target commodity hardware or affordable compute clouds. Scalability by definition means the computation speed of a single simulation run increases by a constant factor per added compute node.

2.3.4 Ease of Use

To be useful for and accepted by experts of other domains than computer science, a simulation system should also be as accessible as possible. There are two aspects are important emphasize in this context. One is the usability of the general toolset. All user interfaces, processes and use cases should be addressed with the final user in mind and what s/he most likely expects the

system to be like. The other aspect is the way provided by the simulation system to model the actual questions. Specifically a good solution should address and overcome the gap between a domain specific model and its corresponding technical representation in the simulation system. This means users should be able to create a model without having to deal with technical details of the underlying simulation framework in the first place. Once a model grows more complex it might however not be possible to hide all facets of implementation anymore.

2.3.5 Visualization

Since large scale simulations with millions of agents are required, a visualization solution that copes with these numbers must be found as well. Current graphics engines and hardware that allows to render these numbers in real-time at once to the screen are simply not available at this time.

Therefore the solution should be able to visualize only a specified section of the whole simulation space. It further is required for that section to be dynamically movable and resizable.

It should also be possible to read out more detailed agent states on-demand. However a visualization will most likely be used to sanity-check the model in early stages of development and for presentational issues. Scientific evaluation requires different tools, which are described in the next section.

2.3.6 Scientific Analysis

Evaluating the output of simulation models usually involves quantifying key indicators and using statistical methods on the output to draw conclusions regarding the initial research questions. Traditionally simulation output gets written to files and is then analyzed by the researchers with their tool of choice (i.e. R, Excel, Python etc.).

Given the potential scale of the output from a simulation model with millions of agents and the cloud-based nature of the proposed system, renders this approach quite questionable, since the output may easily become several gigabytes or even terabytes in size. Therefore a good solution should feature online accessible analysis tools which can work with that amount of data. An integration of widely used frameworks like for instance R would be desirable. Since GIS data is widely used as input for simulation models, it should also be possible to generate new GIS data sets from the resulting output.

2.4 MARS System

2.4.1 Overview

The MARS system is conceptualized as a Modeling & Simulation as a Service system. This is an important difference to other simulation frameworks. Every phase of the modeling lifecycle can be realized without installing additional software packages on the computer of the domain expert. Instead she or he accesses all functionality of MARS through a user friendly web interface. MARS is hosted and maintained by the MARS Group at the University of Applied Sciences in Hamburg.

These system features have been extracted from the MARS team's experiences in developing public transport disease spreading (Noetzel *et al.*, 2013) and crowd evacuation models with the predecessor version of MARS called WALK (Münchow *et al.*, 2014). It quickly became apparent that consolidating required datasets, model design decisions and the discussion of results were too inefficient when working in geographically distributed teams, without a system supporting that workflow.

2.4.2 Concepts

Roles in the MARS Framework

Users of systems like MARS are mainly domain experts. They want to utilize the capabilities of multi-agent simulations to gain a better understanding of the complex systems they consider in their research. Since creating, using and analyzing a simulation model is rarely done by a single individual, the accommodation of each domain expert or group of experts with at least one tandem partner to deal with the more technical aspects of model implementation and simulation execution is proposed.

Therefore, within the MARS system a number of user roles are defined and supported:

Modeler A domain expert who creates the model to be used in the simulation.

Model Implementer A computer scientist developing the code for the model utilizing MARS APIs and libraries.

GIS & Data Scientist An expert in the field of data integration and GIS operations, who prepares datasets to be used by the simulation model and manages these datasets within MARS.

Of course one person may be assigned to more than a single role.

MARS Modeling Paradigm: Layers & Agents

The basic concept in MARS are agents and layers. That allows a unified way of developing MARS simulation models. This concept is essential and it must be used in every model. This section outlines the basic idea and showcases the application of layers and agents in the example model from section 4.1.3.

The layer concept is inspired by the way GIS data is composed. These files are structured in layers, where each layer represents a specific aspect. This aspect may be an agent type as well as a part of the environment.

This idea is translated to a general approach for modeling the implementation of our simulation system. A domain-specific model is transformed into working code by writing a layer for every aspect of the conceptual model. An aspect should be a considerable sized, self-contained but yet manageable piece of the original model. The layers represent the environment into which the agents are placed. Figure 2.2 shows a layer model of the KNP simulation model used in this

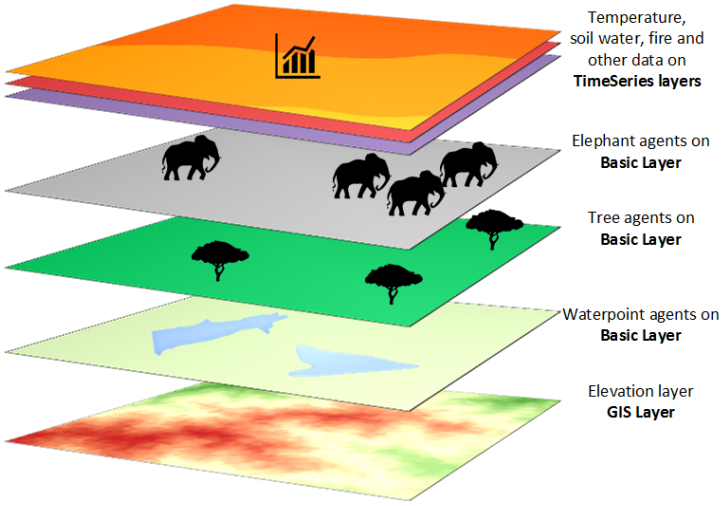


Figure 2.2: MARS Layer concept example from KNP model.

paper. The bottom level keeps the digital elevation map (DEM). Internally it is represented by a GIS shapefile in a resolution of 90 meters. Waterpoints, trees and elephants are represented as agents, which are placed on their corresponding layers. Finally a number of time-series layers are defined, which are a special type of layer used to handle multi-scale time-series data

from a database located in the MARS cloud.

This approach applies best-practice techniques from software engineering, e.g. separation-of-concerns. Hence layers could be seen as components with interfaces to each other. Each layer may expose well-defined operations to other layers through its interface. Agents may use the exported interfaces to access offered properties and services. MARS libraries provide capabilities to define sensors for agents, e.g. discover their surrounding environment. Thus, like in a service oriented architecture each layer is self-describing to external users and enables an agile way to compose and reuse agent and layer components.

Types of Layers

MARS currently provides three different base types of layers:

Basic Layer A blank layer, which has to be implemented by the user. Agents and environment are defined here.

GIS Layer A pre-implemented GIS layer. MARS is capable of filling this layer with a provided GIS file either in SHP or ASC format. Refer to section 3.2.1 for further information on how to map data to such a layer. The layer allows to query for data based on a position or a geometry, which includes polygons, multi-points and lines.

TimeSeries Layer A pre-implemented layer to access time series previously stored in the MARS ecosystem. The layer allows to query for data based on time and position.

3 Implementation

3.1 MARS Workflow

MARS follows a modeling and simulation workflow as shown in figure 3.1. This workflow is designed to be executed in a number of iterations, which include continuous refinement, simulation and validation of the model. In the final stage the results of the model are ready to be used in publications or further research. Usually the modeler starts by creating the

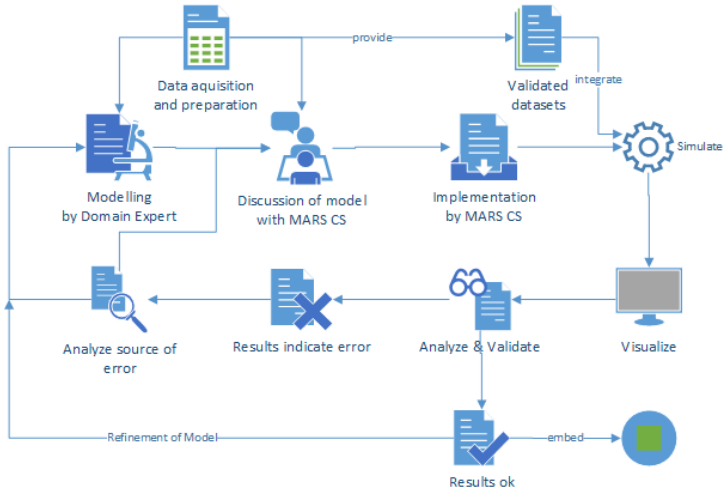


Figure 3.1: MARS Layer concept example from KNP model.

conceptual model according to the research question (Thiel-Clemen, 2013b). It might be useful to consult a computer scientist, when translating a conceptual model to a technical MARS model for the first time.

Once the modeler decides her or his model is complete enough to try a first simulation run, the model should be discussed with a computer scientist to discover possible pitfalls, which might occur throughout implementation or simulation. Sometimes the model code needs additional information, which was not obvious during the more abstract modeling stage. Also this discussion should be used to clarify certain aspects, since there might be ambiguities to the

model developer when simply reading the model description and not having deep knowledge about the domain. It should be mentioned that a modeler might write the model code himself, if she or he is trained in programming with the C# language.

With the first implementation done, the model can be uploaded to the MARS Websuite. GIS data and time-series data may be mapped to the simulation's layers and agent attributes. The data used in this step should have either been prepared (e.g. normalized) by a data scientist or the modeler himself. This task should be done in parallel with the modeling process and the data scientist should also partake in the discussion with the MARS developer.

The modeler can now trigger one or more simulation runs from the Websuite's interface and examine the results as a 3D visualization. A visual analytics page offers fundamental diagram types. Additionally, MARS offers the capability to export the results as a CSV file for further analytics with R or other solutions. These results may be accessed as soon as the first chunks of data are available from the simulation, thus a modeler does not have to wait for the simulation to finish. Validation of the results is the next step as designed by the MARS workflow. If fundamental errors are found in the results, the source of these errors will have to be searched either in the source code or the conceptual model itself. Usually modelers will work hand in hand with their tandem partner to fix these. In case the results are technically acceptable, it must be decided whether the model needs further refinement. If so, the next iteration starts. As soon as the modeler is satisfied, the results can be used in further work and the MARS cycle ends. The result files, model code, uploaded data and configurations will persist inside the MARS system for later usage.

3.2 MARS Basic MSaaS Workflow

This chapter describes the MSaaS style workflow offered by MARS to put the actual LIFE simulation system into context. Most of the described subsystems are being created by other members of the MARS team, who are referenced where applicable.

3.2.1 Simulation Model Preparation

After the conceptual model has been created and transformed into a MARS model (see figure 2.2), it needs to be implemented by a computer scientist. This step has to be accomplished in external development tools (e.g. Visual Studio or Xamarin Studio). The LIFE API is a direct match of the layer-based MARS model and together with additional supporting libraries (e.g.

for agent creation) streamlines this process. However an in-depth discussion of how models are implemented would exceed the scope of this paper.

Once the model is implemented and uploaded, the domain experts may start working with the Websuite. We start out by creating a project and a scenario inside of that project. Projects are the largest organizational unit in MARS, while scenarios are more specific setups in a project. A scenario defines wall-clock simulation timespan, temporal and spatial step size and an optional spatial boundary. Figure 3.2 shows the corresponding form.

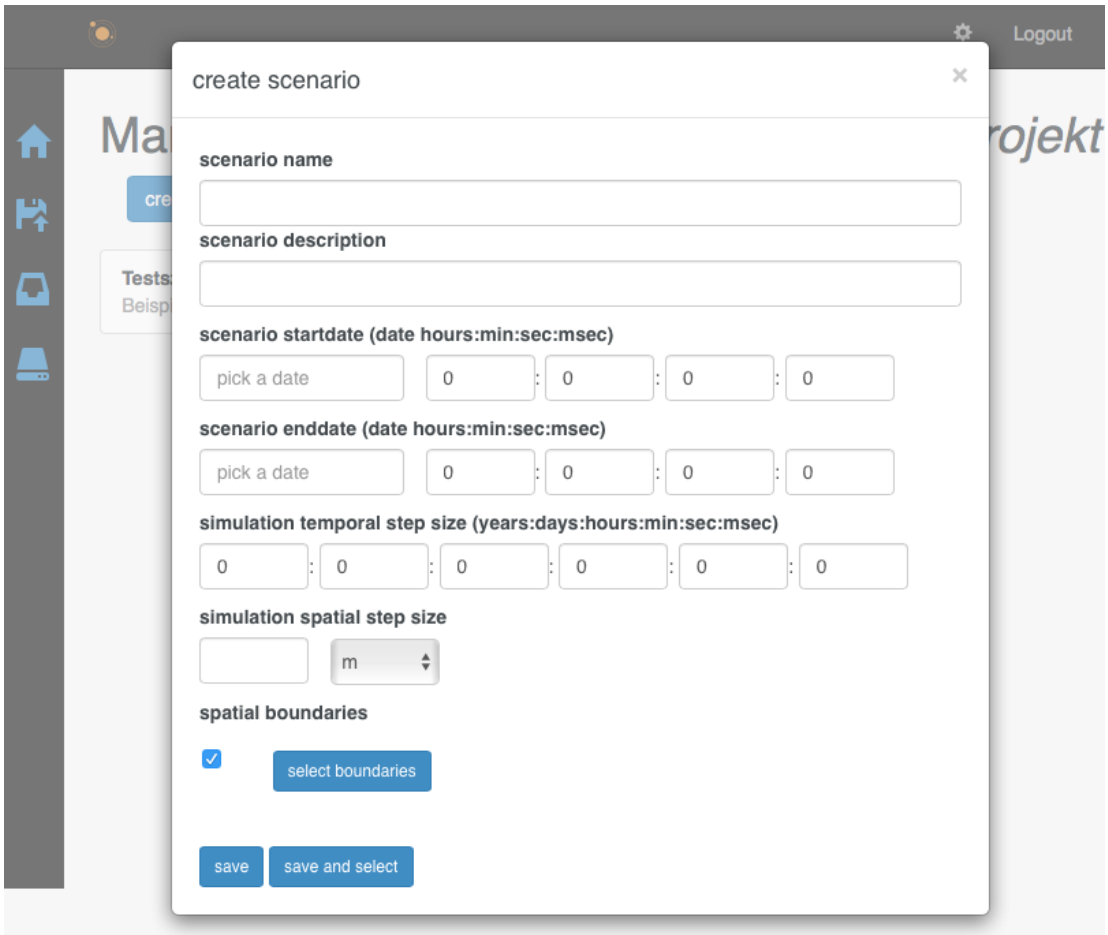


Figure 3.2: Scenario creation in Websuite.

For the KNP model several datasets are needed. A 90 meter resolution elevation map in the SHP format is used for the Kruger National Park, a csv file containing tree positions,

another csv file with elephant herd positions and size as well as several time series in CSV format for temperature and precipitation data. All these datasets can be uploaded through the Import Data dialogue. Depending on the type of data different information has to be provided, i.e. where and when the dataset has been collected, who is the owner of the data etc. Datasets usually are available to all users of MARS, but can be flagged as being private to address data confidentiality concerns. This process has been implemented by Florian Forsthuber (UI) and Mitja Adebahr (Import services).

After uploading the datasets, the user has to define which compilation of data shall be used in the selected simulation scenario. With MARS DEIMOS a tool is offered to review uploaded data and perform a validation against the scenario definition in terms of temporal and spatial scale and data availability. Once satisfied with the selection the tool will create a specially prepared compilation to be used in the next step. Note that MARS never alters the original data. Everything is either stored as meta-data or as copies of the original files, e.g. when a transformation in another format is needed

The next and penultimate step is to map the selected datasets to the simulation model implementation uploaded by the computer scientist. This is achieved with a tool we call SHUTTLE and has been developed by Florian Forsthuber. SHUTTLE will only show parameters of agent constructors, attributed with "[PublishInShuttle]" in the model code, and further only those parameters which are mappable from the outside. This excludes other agents or layers for instance, since they will be injected automatically by MARS LIFE (see chapter 3.3.2]). SHUTTLE provides a split-pane view featuring the extracted layers and agents from the model on the left hand side and the provided datasets on the right. Users can now use the data mapping buttons to dynamically create the domain specific language expressions in the middle pane, and thus map each needed agent parameter to a column from the datasets.
Furthermore SHUTTLE exposes all GIS and TimeSeries layers and asks the user to map GIS and table datasets respectively to them. Figure 3.3 shows this process. The result of SHUTTLE's mapping process is a SimConfig file, which will be used in the simulation run to automatically initialize the simulation model with the data uploaded into the websuite.

3.2.2 Simulation Model Execution

The final step is about creating a SimulationPlan which will then be used to start one or more SimulationRuns. The user creates a SimulationPlan by selecting a SimConfig, a NodeConfig and providing a name. The NodeConfig controls on how many nodes and resources the Simula-

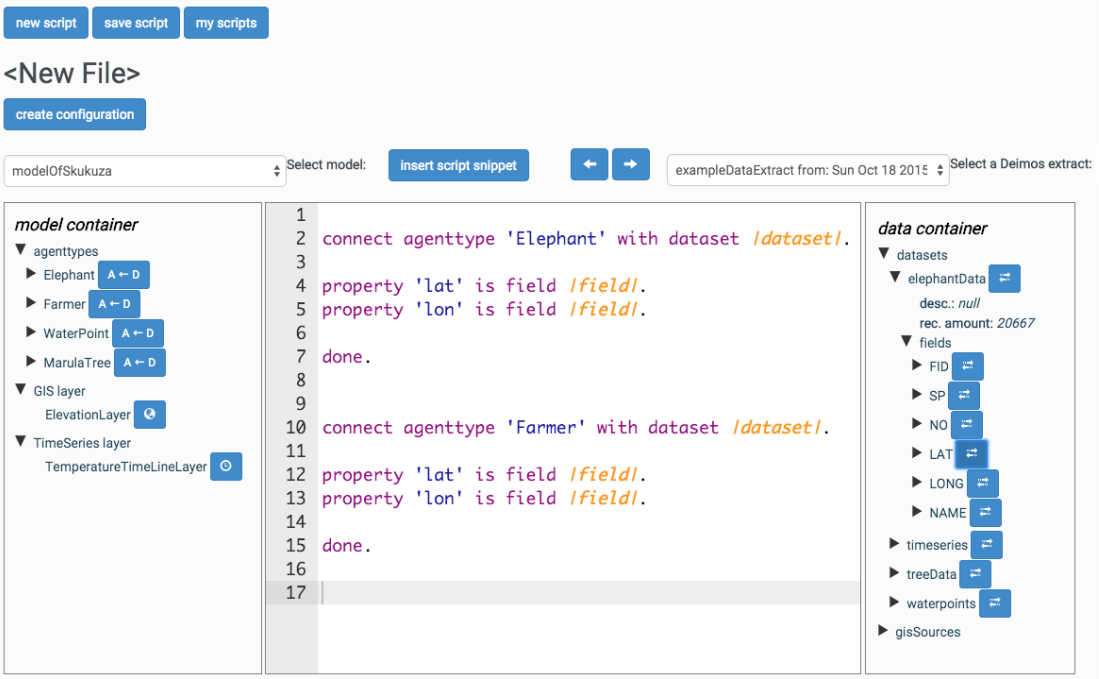


Figure 3.3: Data Mapping with SHUTTLE.

tionPlan will be executed. The SimulationPlan may then be started via the web user interface, which results in a SimulationRun being created. Figure 3.4 shows the corresponding page used in the process. Basic real-time usage statistics for CPU, memory and network load for the current run are shown, when the SimulationRun tab is expanded.

When starting a model for the first time a Docker container image containing all relevant files is created. This image includes the needed C# runtime, model code, GIS files and the SimConfig description. Once the image has been created, it is stored and can be reused. This results in subsequent runs starting almost immediately. After the model container has been started, MARS LIFE will automatically begin the model initialization by creating all layers in the order of their dependencies and by using the mapped GIS and time-series files. When the layers have been put together, LIFE instantiates all agents according to the mapping created in the SHUTTLE tool.

When run in a distributed manner with more than a single LayerContainer, LIFE automatically takes care of remotely initializing all layers and agents. Dependencies for layers are resolved by means of a LayerRegistry service.

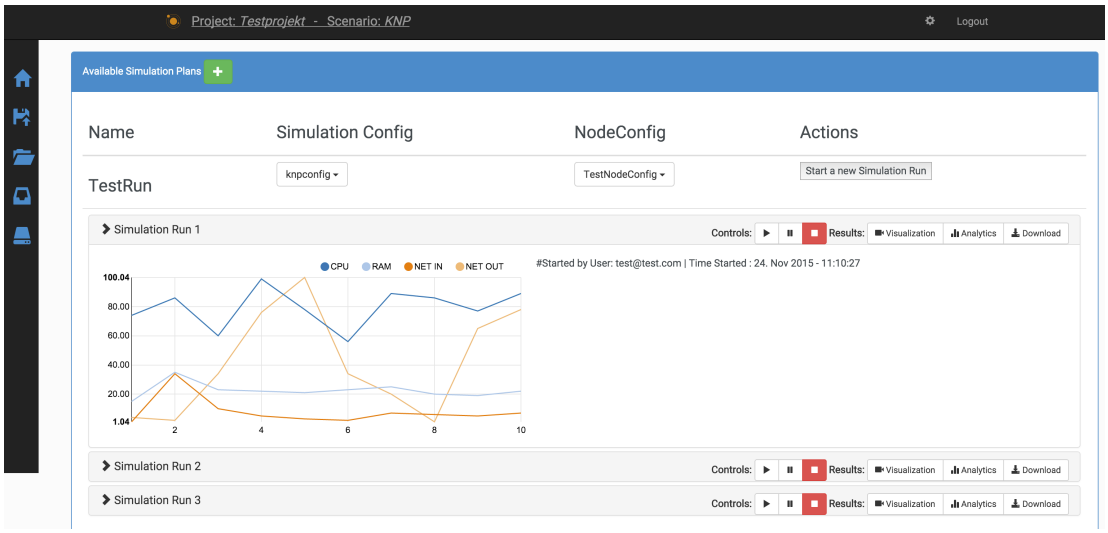


Figure 3.4: Starting a SimulationPlan from the Websuite.

3.2.3 Simulation Model Analyses

Once the model is running, first results are being sent to the Websuite and may be analyzed in any of three ways.

3D Visualization

First a 3D visualization can be displayed. This allows users to quickly check whether their simulation is performing in the way they envisioned. It is a particular good way to check for movement patterns, areal distribution of agents and if overall areal boundaries are done right. However a significant performance impact has to be expected when using this feature, since all information needs to be send to the visualization observer for every tick even though MARS optimizes this by only sending the information currently inside the viewing cone of the virtual camera. Figure 3.5 shows a sample visualization of the KNP model. The 3D Visualization feature is being developed by Jan Dalski.

Visual Analytics

A visual analytics page featuring basic graph types and maps like heat maps, may be used to create a dashboard for a SimulationPlan. The visualized data is updated in real-time as new data arrives and is very useful to check a model's indicator values as soon as they become available. This allows users to stop and readjust long-running simulations in case something

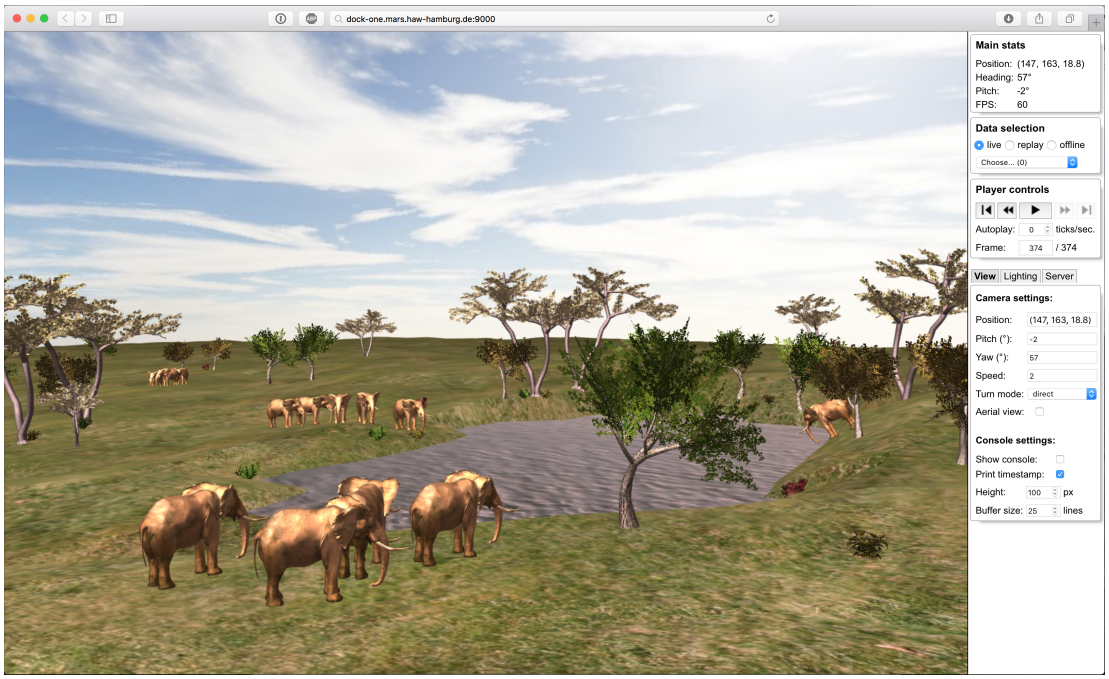


Figure 3.5: 3D visualization displaying data from the KNP model.

is off right from the start. Also modelers can leverage the dashboard while optimizing their models, without ever leaving the Websuite or having to download large data blocks for offline analyses. Figure 3.6 displays the dashboard used for the KNP model. It has been created by Janus Dybulla.

CSV Export

The third option is to download result datasets as CSV files. This is necessary when the data will be used in further research or when the capabilities of the visual analytics page are exceeded and more sophisticated statistical or visual procedures need to be performed (e.g. in R). This again is created by Janus Dybulla.

3.3 Architecture & Technology

3.3.1 Overview

MARS is deployed in two major parts. The first and most visible part is the MARS Websuite. It hosts the website which modelers and model developers use to manage their data, configure

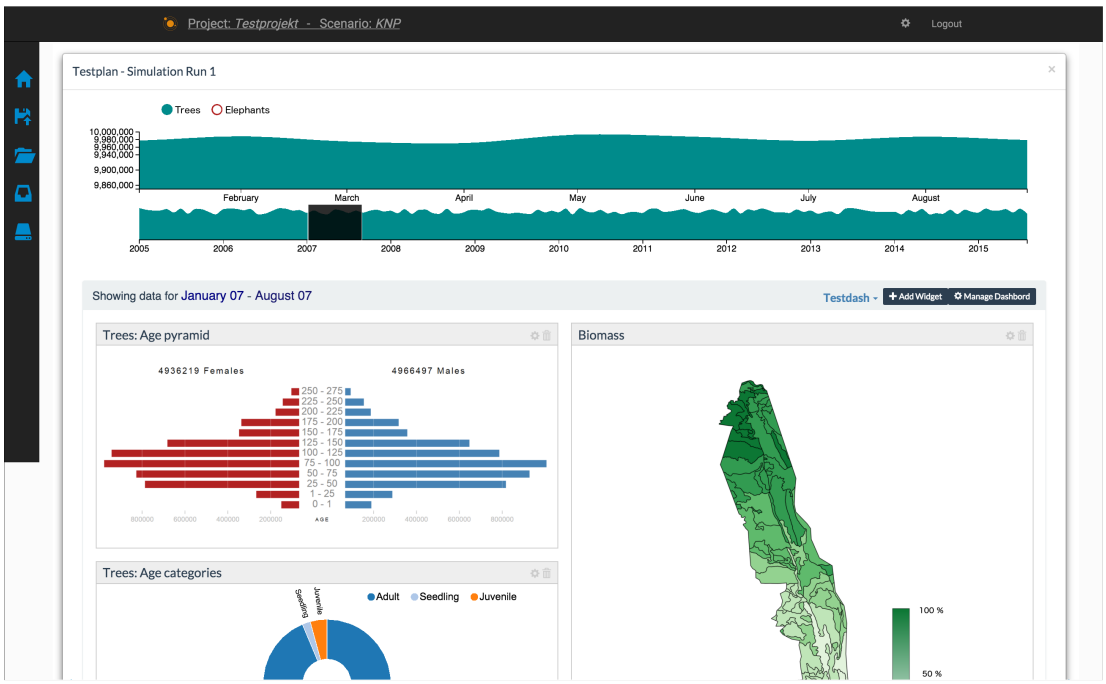


Figure 3.6: A visual analytics dashboard for the KNP model featuring age distribution charts and a biomass choropleth map. (J. Dybulla, pes. con.)

their simulations, start simulation runs and analyze the results. The second part of MARS is the simulation system. It is instantiated and configured specifically for each simulation run which is started through the Websuite. All output and results from the simulation system are transferred back to the Websuite to be evaluated by the users.

The actual simulation component in the overall MARS architecture (Hüning *et al.*, 2014) is called LIFE. It consists of two main processes which make up the distributed simulation system. The SimulationManager is the centralized controlling application for the simulation. It manages the model, calculates the distribution and scheduling pattern, takes care of the distributed initialization and finally controls the simulation run. The LayerContainer houses layers and agents, which are the two primary logical components MARS simulations are made of. A LIFE system may be composed of any number of LayerContainers among which layers and agents are distributed. Layers are treated like plugins by LIFE and thus are loaded on demand when initializing a new simulation. This approach allows for automatic dependency injection, when one layer depends on another.

LIFE is completely implemented in the C# language. It can be run via the .NET runtime system on Windows and via the Mono project on Linux and OS X. By that it resembles the same platform independence as Java. However during development and deployment into production the MARS team chose to solely use Linux Docker containers (www.docker.io) for running the MARS Cloud infrastructure services and the actual simulation runs. Hence MARS LIFE relies on the Mono runtime for C# (www.mono-project.com).

Well known Infrastructure and Platform as a Service (IaaS & PaaS) paradigms are used to host MARS . To provide IaaS Linux KVM is used as virtualization technology and OpenNebula as the management tool to operate virtual machines on top of available hardware. Linux KVM (www.linux-kvm.org) and OpenNebula (www.opennebula.org) both are open source projects and run on a wide range of hardware, which helps a great deal in achieving a cost-effective cloud environment. While hardware virtualization is provided by KVM, Docker is used to virtualize all MARS Cloud applications. Docker allows to use the same environment during development and production, which enables a very fluent deployment process.

3.3.2 LIFE Simulation System

This section provides an in-depth technical description of the LIFE system.

Architecture Style and Specialties

Just as the MARS Websuite part of the MARS Cloud, LIFE is conceptualized according to the Microservice architecture style (Fowler, 2014) . The SimulationManager and LayerContainer however do not use REST interfaces for their communication, but rely on a more performant binary protocol, since their communication is very time critical. Also these services are not bound together via the usual gateway services (Netflix Zuul & Eureka in the MARS case), but are created on-demand through the Mission Control service and discover each other by means of a NodeDiscovery service, which uses multicast messaging to advertise and find other nodes. Another distinction to the pure microservice style is that for a certain operation one may not simply call any instance of the corresponding LIFE service. So each LayerContainer and SimulationManager in a deployment must not be exchanged with another instance and calls need to be precisely addressed to a specific instance. This is due to the individual agents and their state, which resides in-memory in each process. However this could be changed in a future version if the need for fail-over or high-availability deployments arises.

The applications are deployed as Docker containers from previously created images. This approach allows to restart a simulation very quickly once the image has been created. Also an image may be moved to other infrastructure setups or even to laptops, which also enables a modeler to take a simulation with him (i.e. for a conference etc.).

Code Structure & Layout

The LIFE code base consists of two major projects, the SimulationManager and the LayerContainer. They are accompanied by additional LIFE Services, which are separate components, but usually are solely used inside the LayerContainer. Both major processes share certain smaller components grouped together in a namespace named 'Common' for the more functional parts and 'T-Components' for very technical functionality. Figure 3.7 shows the major processes and the most important smaller components. The 'Common' namespace mostly contains

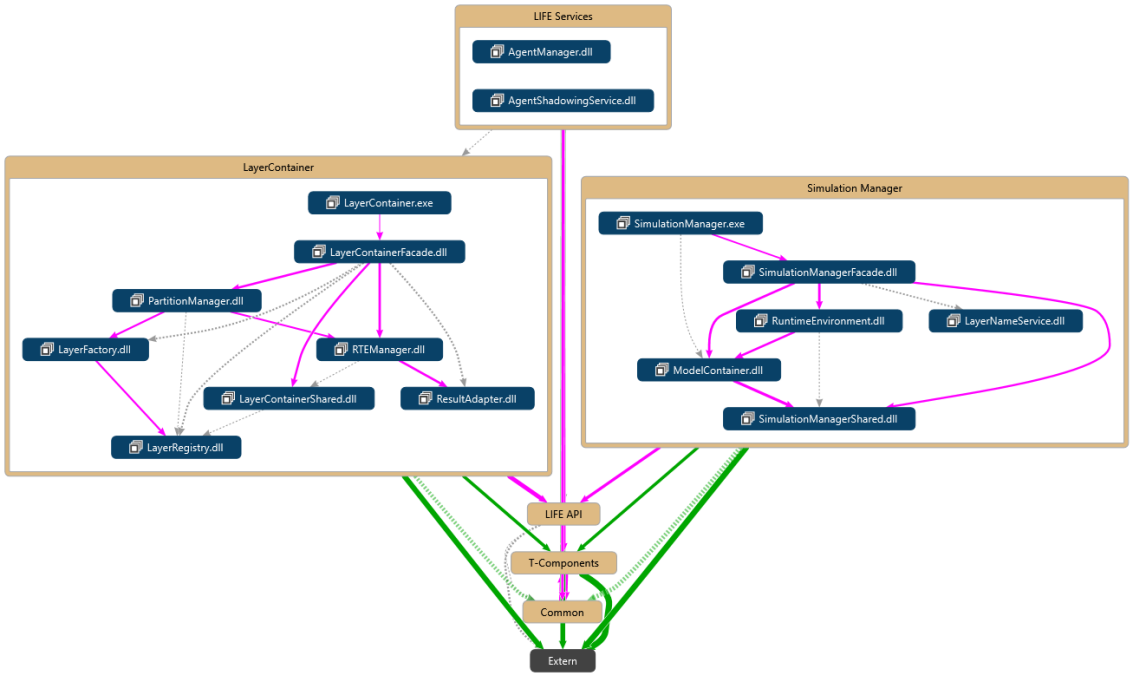


Figure 3.7: LIFE Code Map.

helper classes or commonly shared interfaces. Hence the name. An exception to that is the NodeRegistry, which is used by every LIFE process during startup to find other LIFE nodes belonging to the cluster.

Code which is directly used by model developers resides in the 'LIFE Service' and 'LIFE API' namespaces. 'LIFE Service' holds the AgentManager and AgentShadowingService classes, which may be used for automatic initialization and distributed agent resolution. The 'LIFE API' namespace is where the various Layer and agent interface definitions are located.

Finally the 'T-Components' namespace contains technical implementations of the lower level messaging components (i.e. AgentShadowing) and sub-system adapters like RabbitMQClient or MulticastAdapter.

SimulationManager

The SimulationManager as described in section 3.3.1 is the control server in a LIFE deployment. When deployed the Docker container will contain the model code and a 'SimConfig' file in JSON format. The entrypoint for the container is set to call 'mono SimulationManager.exe -m <ModelName>' which upon container start will execute the SimulationManager and order it to execute the model identified by <ModelName> through the central SimulationManagerFacade. The SimulationManager will now tell the ModelContainer component to look into its designated sub-directory for a model by the assigned name. Also a RuntimeEnvironment is created, which waits for the ModelContainer to have the model prepared.

Once the ModelContainer found a suitable model, the corresponding DLLs will get loaded into the current AppDomain and are searched for implementations of the basic ILayer interface. During this process all layer constructors are analyzed and once the model is completely scanned, the instantiation order for the model is being calculated.

Now the RuntimeEnvironment gets triggered to setup the simulation run from the provided SimConfig and by using all previously registered LayerContainer instances. At first an attempt is made to connect with all LayerContainers. This approach will only fail the setup process, when no LayerContainer can be connected at all. However if two LayerContainers were initially found, but only one can be connected, setup will inform the user but still continue with just one container.

When the cluster has been set up, the instantiation order is fetched from the ModelContainer and layers are created according to it. The LayerContainer provides a service which handles the dependency injection for references to other layers (see section 3.3.3). During

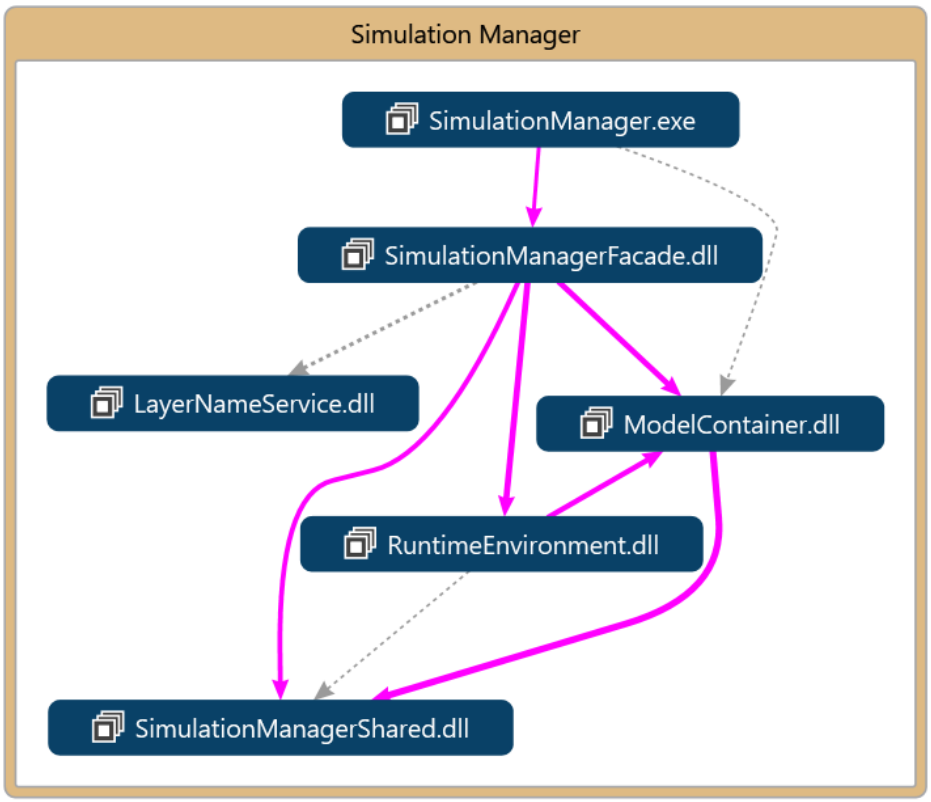


Figure 3.8: SimulationManager Code Map.

layer creation it is checked whether distribution is possible. If not, the layer is just instantiated on the only LayerContainer present. After all layers are completely instantiated, they will be initialized by issuing a call to their InitLayer() method. Depending on the type of layer and the information from the SimConfig file, the InitLayer() method's parameters will be filled differently. For instance a GIS layer needs the location of the GIS file to load, while basic layers need information about the agents which are to be created. For further details on the InitLayer process, refer to the section about the AgentManager component 3.3.3.

If distribution is possible the behavior depends on two things: The layer's type and whether or not there is a distribution configuration loaded from a distribution config.xml file. The type is important, because GIS and TimeSeries layers may only be replicated, while basic layers with agents may be distributed as several smaller pieces. The XML file is currently used as a placeholder until an user interface in the Websuite is implemented to configure the distribution. If a layer is not described in the XML file, it will simply be created in the first LayerContainer.

So in case more than a single LayerContainer is present, but no XML file is provided, the simulation will completely run on a single LayerContainer in non-distributed mode.

If a layer is to be replicated, it will simply be created on all LayerContainers consecutively. If the layer is to be distributed, the only currently supported mode is an even distribution across all available LayerContainers. When this mode is selected, the layer will as well be consecutively created but the parameters provided to the InitLayer() method will be altered to reflect the reduced amounts of agents.

When setup has finished, the RuntimeEnvironment creates an instance of the SteppedSimulationExecutionUseCase class. This class will now start the actual simulation run by triggering each LayerContainer to advance by one tick (simulation step). It will wait until all LayerContainers reported back they are done with the step and then repeat that form of stepped simulation execution for as many ticks as are specified in the SimConfig file.

Alternative Forms of Execution

The actual simulation execution has been outsourced to its own class, to allow for an easier addition of alternative simulation execution modes in the future. Not everyone may like a strict stepped simulation execution, which synchronously has to wait for all LayerContainers to finish before starting the next step. Event based simulations for instance will require a different way of execution. Adding a new mode of execution thus involves writing a new SimulationExecutionUseCase, but also might include changes in the LayerContainer, since that is where the actual agents are executed.

Layer Name Service

Centrally hosted by the SimulationManager, this service offers a simple interface to LayerContainers to register, resolve and remove layer instances by their type. For each layer type a list of TLayerNameServiceEntries is stored. These contain connectivity information needed by each LayerContainer's LayerRegistry to resolve remote references for dependency injecting them into newly created layer instances. Though currently not implemented, the Layer Name Service will also be the place for load balancing remote layer instances in order to further optimize performance in the future.

LIFE Webservice

As the control server for each LIFE cluster, a SimulationManager also hosts a webservice. It offers an endpoint to start, stop, resume or abort the current simulation and is consumed by the Mission Control Service in the Websuite.

3.3.3 LayerContainer

Just as the SimulationManager the LayerContainer is deployed as a Docker container with a default entrypoint. In this case it defaults to 'mono LayerContainer.exe', so the LayerContainer ist simply started. After the internal startup has succeeded the LayerContainer starts sending advertisement messages by means of the NodeRegistry which are received by a listening SimulationManager. As described in section3.3.2 these are then used to setup the LIFE cluster and start the initialization phase. A facade class acts as the central access point for all remote calls coming in from the SimulationManager.

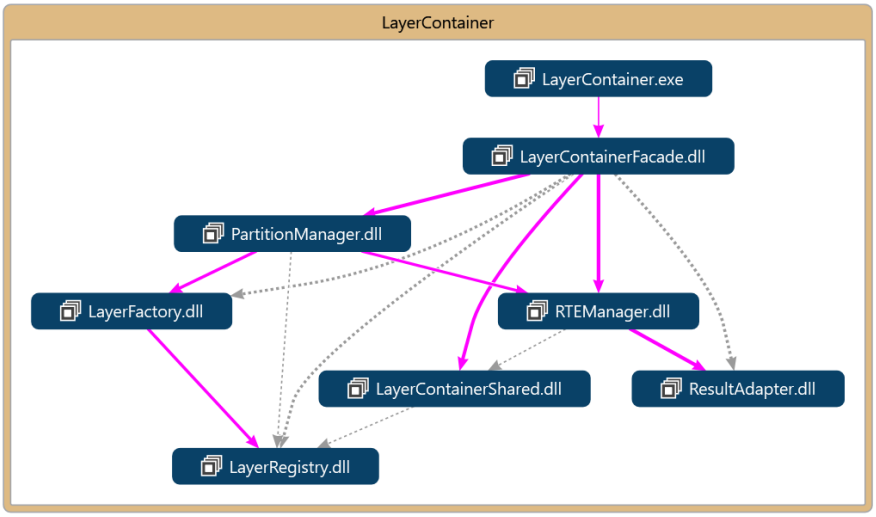


Figure 3.9: LayerContainer Code Map.

When the LayerContainer is connected to a SimulationManager and the latter starts the simulation setup, a message containing the model code is received. This code is extracted to disk and loaded by the PartitionManager, who passes this to the LayerFactory, which finally

loads the received model code via the AddinManager. The PartitionManager does not do anything else than forwarding the call at the moment, but is meant to become more important in the future as more sophisticated partitioning strategies are set into place (i.e. neighbor-based partitioning etc.). The AddinManager is a helper class, which controls the actual runtime class loading for the model code. Once that is accomplished, the LayerContainer is ready to instantiate and initialize its first layer. This happens by command of the SimulationManager and is executed by the LayerFactory component.

When a Layer is instantiated by the LayerFactory it needs to resolve the references to other Layers which are defined by the constructor parameters. To achieve that each LayerContainer features a LayerRegistry which manages a dictionary of instantiated layers. Whenever a Layer is instantiated, its reference is added to that dictionary. So when references are needed, the LayerFactory just has to ask for them by name.

This works as well in the distributed case, since the LayerRegistry itself reports registered Layer instances to a LayerNameService being hosted by the SimulationManager (compare section 3.3.2). So when the LayerFactory requests a Layer which is not present on the local node, the request will be forwarded to the LayerNameService and a proxy object for the remote reference is being generated, which is compatible with the required interface type from the constructor.

AgentManager

The AgentManager is a component that may be used by developers to automatically create agents of a specific type in the InitLayer() method. It needs to be generified with the interface and class type of the desired agent type and requires a valid AgentInitConfig as parameter, which can be retrieved from the TInitData parameter provided to the InitLayer() method.

Also any needed layer and environment references need to be passed in as well. The layer references may be obtained through dependency injection from the layer's constructor, while the environment might either be self instantiated in the constructor (polyglot ESC) or also be injected in case the environment implementation is provided as a layer. Once that is done a single call to GetAgentsForAgentInitConfig() returns a list of agents, which can be stored in the layer for further usage and management.

In order to instantiate the agents, data from databases hosted in the MARS cloud might be needed and will be fetched by the AgentManager. This is done in a parallel manner and should be as fast as possible. Depending on how many agents are to be created and how large

the used data sets are, this process might turn into a lengthy task. Compare the results from experiment 3 in chapter 5.3.

3.3.4 Agent Shadowing

Distribution and thus communication are two key aspects of scalability. In a very early version of the MARS system, layers were only distributable as a whole, so each LayerContainer needed to take care of one or more complete layers. The authors' findings however have shown, that one layer may be too complex for a single computer or there may be some rather slow compute nodes involved. So the new approach will also allow to distribute each layer across several LayerContainers, which resembles true horizontal scalability. Since distributing layers has direct influence on the agents living on them, the approach for layer distribution is tightly coupled with the distribution of agents and is meant to make the overall system scalable. That approach is called Agent Shadowing.

Agent Shadowing is the depiction of an agent living on layer instance A having its shadow cast onto layer instance B, where it is not actually instantiated, but instead is represented by a stub-like object as in remote communication concepts like RPC/RMI (Figure 3.10). In RPC/RMI each

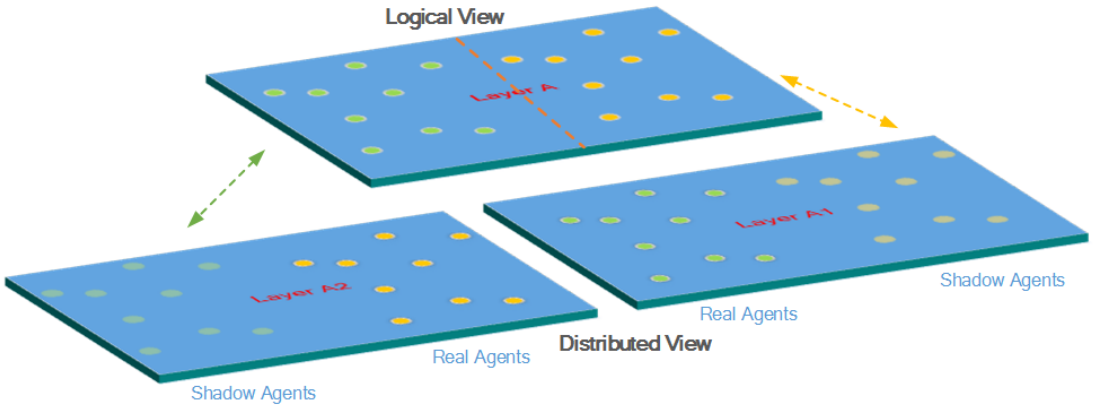


Figure 3.10: Agent Shadowing Concept - Logical View.

agent's methods and properties are callable by third parties through its stub object. Usually a stub just provides the capabilities to establish an interface-bound communication with the remote object. If the remote reference changes, in classic RPC/RMI the stub simply becomes useless, since its reference is not updated. The protocol then has to notice the broken link and re-establish a new one.

MARS LIFE creates shadow agent stubs (SAS) by making use of C#'s RealProxy class. This class consumes an interface and translates calls to that interface's methods into network messages. A SAS is then extended by the ability to hold cached attributes like its environmental position or any other attribute. The real agent object updates both, the attributes and the remote reference, whenever a change occurs. These updates are delivered via multicast when in LAN to reduce the amount of traffic. The initial remote references can be provided when the overall system is initiated since some kind of distribution information has to be provided at that state. This results in each container node containing the full environment as well as all agents, but with the difference, that only numberOfAgents / numberOfNodes (given an even partitioning) agents are really instantiated and thus have to be computed. The remaining agents are only instantiated as SASs and do not contain any agent behavior logic. An increase in container nodes would reduce the amount of agents per node that have to be actively computed, while the memory footprint per node would also potentially decrease, assuming that a SAS consumes less RAM than a full-fledged agent. There is, however, a limit to how many agent's can be instantiated as SAS during initialization. Chapter 3.3.4 provides a solution attempt for that case.

Calling or referencing another layer, works by the same pattern of either having a local instance of that layer to address directly or a stub to communicate with a remote reference. Environmentally it does not matter which remote reference to a layer is provided, since each layer instance locally has a full view of its state, even if distributed.

Within MARS LIFE all implementations regarding AgentShadowing are encapsulated within the AgentShadowingService, which itself is exposed to model developers by means of the AgentManager. That architecture allows to provide a meaningful interface to developers, while hiding a lot of the technical implementation inside.

Addressing and Grouping of Agents

Each SAS requires a unique connection to the real agent. This results in a unique port and socket being used on the client machine. Depending on which OS you are on, the maximum number of simultaneous TCP connections is limited. For example on Windows 7 you could theoretically create up to 16 million concurrent connections, but are limited by port restrictions to something closer to several thousand connections. It thus is obvious that this is not a suitable approach for a massive scale multi-agent simulation with possibly a million agents. Other approaches towards the network implementation need to be taken.

An alternative is to change the communication technology from TCP Unicast to UDP Multicast. A real agent would send its state update only one time per multicast to a group of potentially interested listeners, who then all receive the message and decide individually whether or not to consume it. This solution requires the creation of multicast groups, which could be created either on a per agent, per agent type or per layer base.

A group per agent wouldn't do much to the port problem mentioned above, since each multicast socket listener would still require its own port. Grouping by layer could result in too many messages received by too many SAS which don't need it (if a layer holds more than one agent type), or could create a bottleneck if the amount of agents is very high on each layer instance. Possibly the best result can be achieved with a grouping, based on agent type, since the recipient group is narrowed down to those agents and hosts, that can use the message. With that kind of interest management the bottleneck problem should directly scale with further distribution of the layer itself.

As an additional side effect of the per agent type multicast messaging, updating the remote references of stubs in case of cross-boundary movement of agents, will not be necessary anymore. Since each individual agent can be reached by a multicast group uniquely generated from the agent's type and multicast messaging inherently does not care about endpoints, real agents may be moved across nodes without regard to references and stubs may be removed and created dynamically as needed without further lookup.

Cross-Platform Generation of Multicast Groups

The multicast groups needed for each agent type, need to be generated on each host independently. Since we cannot know whether MARS LIFE will run with .NET or Mono, the solution for the generation algorithm needs to be platform independent as well. To create a unique multicast group based on the agent type, methods GetHashCode() and Random.next() are needed by the implementation. Simply using the build-in solutions from .NET and Mono is not working, since Mono was build without knowledge of .NETs implementation. This results in different behavior in both the Random.next() and the GetHashCode() methods and thus in different multicast groups on each platform.

MARS LIFE solves these issues by implementing specific Random and HashCode methods based on low-level mathematical and bit-shifting operations. More specifically Random is

exchanged by the XOR128 algorithm while FNVHashcode is used as a replacement for .NET and Mono's built-in GetHashCode().

Messaging & Serialization

Method calls to a Shadow Agent Stub are automatically translated by LIFE and result in a message being send via multicast to the corresponding multicast group. These messages need to be serialized. By default Agent Shadowing uses the build-in BinarySerializer from .NET / Mono. This library is capable of automatically serializing all classes, which are programmed in valid C# and by that allows for a very convenient way of sending messages and thus transparently calling remote methods.

A special case of messaging is the retrieval of attributes form other agents. Usually this results in calling a getter method or in performing a get-call to a C# property respectively. So whenever a local agent needs information about another, non-local agent, a remote call needs to be issued to retrieve the most recent value. This is not optimal, since a lot of redundant messaging happens, when multiple agents from the same node require the same attribute form the same agent. MARS LIFE handles this by means of a push mechanism. Whenever a SAS is created for an agent, the SAS is registered with the real agent. As soon as the real agent changes an attribute, this updated value is send proactively to all registered SAS instances. The SAS instance may now serve get-calls from an internal cache instead of issuing a remote message call, which should result in a great performance boost. The approach guarantees cache values to be the most recent available.

The load put on the CPU by serialization and the resulting message size are not optimal and can be greatly enhanced by using specialized serialization libraries like Protobuf. The downside of solutions other than the C#-native BinarySerializer however are the restriction towards what can be serialized and how. For instance Protobuf requires a certain mapping alongside the classes which are to be serialized and needs a really profound knowledge of its capabilities by the developer in case inheritance and other, more complex class structures or programming paradigms are in place. Also this solution removes transparency from remote method calls.

Since using a third party serialization library has its very specific demands and eradicates communication transparency anyway, MARS LIFE offers a special interface by the name of ISerializableByLIFE. Implementing this interface on a custom type, requires the developer to

write a special serialization for that type. This implementation should translate the custom class into a representation which solely uses primitive types and arrays, because these don't have to be serialized at all prior to sending. A best practice is to encode the state of an object into a bit representation, which then can be send as a byte array.
Whenever Serialization needs to take place in AgentShadowing, this means a method call needs to be handled. In doing so, all non-primitive parameters, return values and exceptions have to be serialized and de-serialized. This implies that ISerializableByLIFE needs to be implemented for all these types when opting for optimal serialization performance. MARS LIFE checks whether a type implements the interface and automatically falls back to the BinarySerializer if that is not the case.

Adding and Removing Agents at Runtime

With Agent Shadowing in place, the creation and deletion of agents at runtime becomes a matter of informing remote nodes about their birth or decease. Therefore each newly created or removed agent must be registered or un-registered with its generified AgentShadowingService in the containing layer's code. The service will then take care of all further steps. Specifically it will make sure that messages for a deleted agent, that still arrive in the same tick, are being handled by the agent instance regardless of its decease. The agent will thoroughly be removed before the next tick. This behavior ensures model developers do not have to handle this condition in their model code.

Paging for Agent Shadowing

It is expected that creating a lot of proxy instances is impracticable due to the time it takes to instantiate and manage all those objects. Therefore a paging approach for Shadow Agents has been implemented. Shadow instances are created on-demand through the AgentManager and feature a TTL (Time To Live) value which is decreased with each simulation tick in which they are not accessed. When TTL reaches zero, the Shadow Agent is erased. This approach most likely features slightly slower performance on first access attempts to remote agents, but resolves the matter of very long simulation initialization times when using very large numbers of agents.

Agent References

When simulation entities in a model are discovered by exploration or any other feasible method, finally a technical reference to the real agent needs to be established. In a non-distributed

simulation run this can easily be achieved by passing actual object references to the searching entity. In a potentially distributed simulation however a discovered entity may be a local or a distributed agent due to the transparent notion, which is provided by MARS LIFE. Therefore when an agent has been found, the retrieved agent id needs to be resolved by means of a generically typed AgentShadowingService (ASS). During initialization all agents are registered with an ASS generified to their type, whose internal data structure is static. By that each ASS knows all locally registered agents and may thus decide whether or not to create a local reference or a proxy object.

4 Experiments

To completely test the defined hypotheses and analyze MARS LIFE's performance and scalability capacities a number of experiments need to be conducted. Some of them may be done as purely synthetic tests (i.e. impact analysis of virtualization layers), while other require a reasonable (but not scientifically correct) model, that resembles some real-world complexity and challenges the usual MARS workflow. The overall goal is to verify or falsify the hypotheses and to find out which components in MARS need further attention from a performance and scalability perspective.

4.1 Setup

4.1.1 Infrastructure Setup for Experiments

Hardware

The MARS system is being run on a cloud-like infrastructure, which is hosted at the HAW Hamburg. The test setup consists of 5 physical host machines. Four of these machines are Mac Pro workstations from 2013 in a configuration with a Xeon E5 6 core 3.5 Ghz CPU, 64 GB main memory and 256 SSD each. The fifth machine features a dual Xeon X5660 with 24 cores each and 96 GB of main memory. All machines are connected to a 40 TB storage system in a Raid 5 configuration. The network uplinks are 2 Gbit from each compute node to the switch and 20 Gbit fibre channel from the switch to the storage system.

Software

The physical hardware nodes all are installed with Ubuntu Linux Server 14.04 and run on Kernel version 3.13.0-61. On top of that Linux KVM is executed to provide a virtualization layer across all physical machines. One Mac Pro and the 48 core machine are used to host the MARS ecosystem services, like webservers, databases and the microservice based architecture, which make up the MARS backend services for simulation initialization and result analytics. The remaining 3 Mac Pro machines are each equipped with a virtual machine running Ubuntu

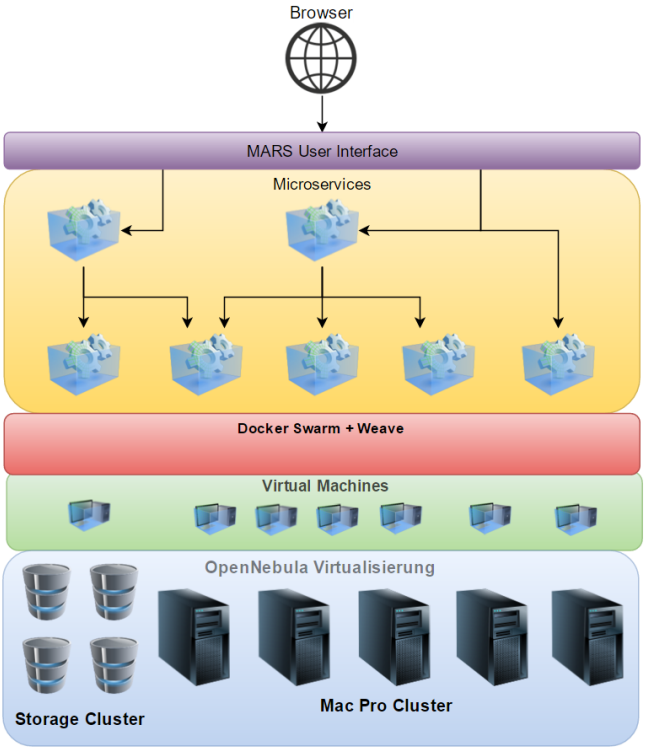


Figure 4.1: MARS Virtual Infrastructure.

15.10 with Kernel 4.2.0-34 and each host a Docker Runtime in version 1.10.3. The three Docker hosts are bound together into a Docker Swarm to allow for a single access point for container deployment and the possibility of extended scheduling and load balancing mechanisms. Figure 4.1 shows the setup.

4.1.2 Special Settings and Details

Mono Garbage Collector

The simulation containers themselves contain Mono 64bit in version 4.2.2.30 and are configured to use a nursery size of 2GB for their sgen garbage collector. This circumvents the problem of too often garbage collecting the nursery (small object space) albeit a lot of main memory is available (64GB). This dramatically increases execution speed, because the whole process does not have to be halted too often for garbage collection purposes. The nursery is the first generation in Mono's generational garbage collector. Small (< 4kb) objects start their life cycle here and only get promoted to the second level in case they get larger and / or live longer than

a certain threshold. However this means that whenever the nursery is full a garbage collection needs to be performed. This, at least in the Mono implementation, is only possible by halting all threads of the application, perform gc and resume everything. This results in the CPU usage going down to a single core, which may irritate performance measurements. Without this setting the mono runtime would not use more than 1-2 cores of the provided hardware since it had to stop every couple of seconds to do the garbage collection.

Multicast with Docker Swarm

Docker Swarm nicely integrates Docker Network including an overlay driver to allow containers on different hosts in the swarm to seamlessly communicate with each other. Multicast communication is not supported by that driver, but is required by MARS LIFE, since it is a crucial part of AgentShadowing. The Docker overlay network driver is implemented by using vxlan over unicast, which means multicast could be implemented by utilizing packet replication on a network level. This has not been done in the current version of docker for a variety of reasons.

Weave Net (https://www.weave.works/), a third party solution, however features multicast and integrates just as easily into a Docker Swarm setup. So Weave is deployed on all Swarm nodes and the 'weave' driver is used for all containers being deployed in the context of MARS.

4.1.3 An Experimental Model

Since MARS has been developed alongside the requirements which are raised by ecological models, it is just natural to use a simple yet complex enough model as test case. The scenario was chosen, since it is well aligned with the current major use case of MARS, the current partnerships of the MARS project and for its good comprehensibility.

A significant test model would need at least one layer of each type (basic, gis, time-series), moving and non-moving agents, exploration of the environment (ESC) and utilization of external data like GIS files and time-series. Thus the test model contains two agent types with fixed spatial locations (marula trees and waterpoints) as well as mobile ones (elephants). In addition a GIS shapefile containing a 90m resolution height map is used in an elevation layer as well as a timeseries layer fed from a csv file containing temperature readings. All agent types interact with each other and with their environment. So together they provide agent-to-environment and agent-to-agent interactions. Figure 4.2 depicts the conceptual model, while figure 4.3

outlines the technical model structure. For a more visual model representation according to the MARS layer concept, refer to 2.2.

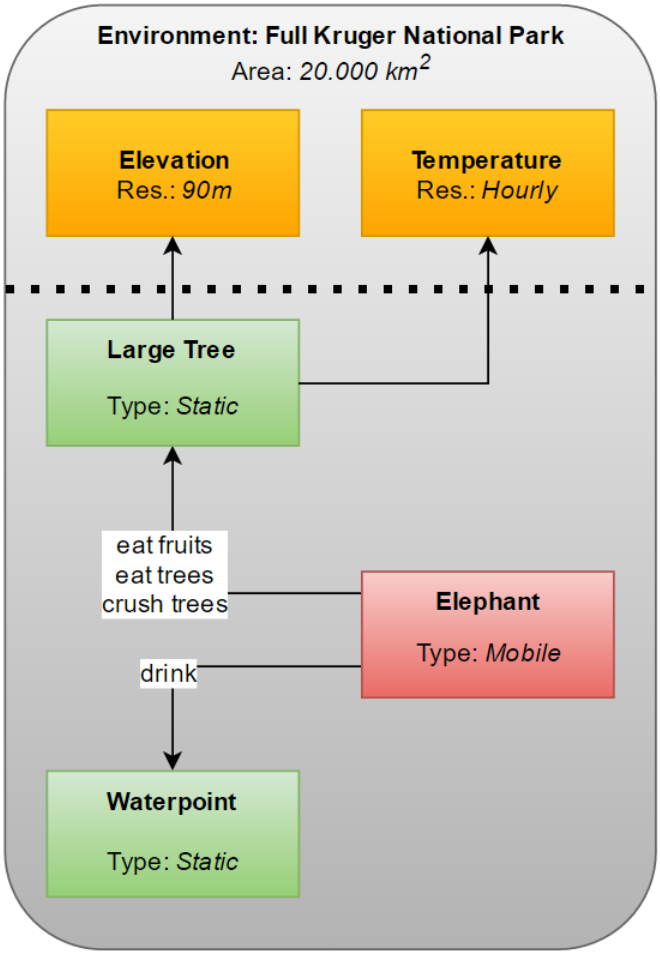


Figure 4.2: Conceptual view of KNP model including agents, environment and used datasets.

The quantitative structure of the model covers the whole Kruger National Park. This means an area of almost 20.000 square kilometers with 15.000 elephant, 415 waterpoint and 5.2 million tree agents is simulated. The spatial resolution is set to meters, while the minimum temporal step size is 1 hour. Thus all agents are executed in an hourly fashion, except for the trees, which are implemented in such a way, as to wait for 24 hours to pass and then execute a step on a daily base.

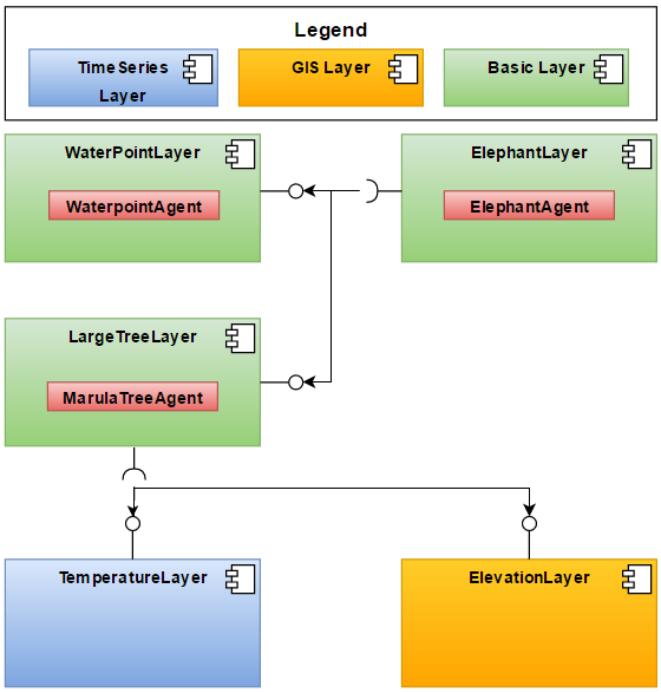


Figure 4.3: Component view of KNP model and its layers.

As described by the conceptual model (figure 4.2), the waterpoint agents are the least active agents. They pretty much just react by refilling themselves to the elephants drinking from them. The Large Tree agents are more active in the sense that they use elevation and areal temperature data for their growth actions in each tick. Also based on the current date they may produce fruits and distribute the daily fruit amount over a course of three months based on a gaussian distribution function.

While these two agent types are static and mostly self-centered, the elephant agent type is the very opposite of it. Its lifecycle is designed around its metabolism. Elephants basically have to eat and drink constantly throughout the day and do so while wandering the savannah ecosystem. So the elephant agent first updates its food and water consumption depending on the passed time step and checks whether it's still alive or not. Next up the surrounding environment is explored for waterpoints and trees. Waterpoints allow for drinking from them and thus provide a comparably easy interface. When an elephant finds a waterpoint it fills up until hydration is no longer an issue (200 liters in this model). When trees are found the decision making is more complex. It incorporates a selection of trees depending on distance and

whether or not a tree bears fruits or is a seedling or not. Eating fruits is the most favored case, consuming the whole tree in case its a seedling is the second. If none of these cases applies, the elephant pushed the tree to model general damage on vegetation while walking through the environment. So no matter which condition becomes true, each elephant interacts with a tree in each if any are found. Figure 4.4 provides the activity diagram for internal elephant logic.

These three agent types represent a vertical slice through the possible incarnations of agents. While a waterpoint agent is not moving and just refills itself by means of a simple formula, the elephant agent features a multitude of complex internal and external actions and interactions with other agents. Not only has the agent to explore its environment, but it also needs to move according to the exploration results and the rules of its inner logic, which is largely dictated by herd behavior in this case.

4.2 Experiment Description

Based on the example model and the hypotheses (chapter 1.2) the following experiments are defined. The outcome of an experiment might be relevant for several hypotheses, while others are strictly defined for a single hypothesis.

4.2.1 EXP1: Performance comparison of bare-metal, KVM VM & Docker on KVM

In order to test the first hypothesis (H1) the well-known Linux benchmark tool 'sysbench' will be used in version 0.4.12. The benchmark will calculate prime numbers. The test will first be executed on a bare metal MacPro with the configuration described in chapter 4.1.1. It will then be repeated on a virtual machine running Ubuntu 14.04 with 12 virtual cores, since the Mac Pro features 6 real cores plus hyper-threading, which results in 12 logical cores visible to the hypervisor. Finally a third run will be executed using inside a Docker container running on Docker 0.9 in the aforementioned VM. The comparison of the three results will show whether the virtualization layers have any impact on cpu performance or not.

4.2.2 EXP2: AgentShadowing standalone test

The AgentShadowing concept as described in chapter 3.3.4 is at the core of MARS LIFE's distribution concept. It is therefore crucial to analyze the performance impact of its messaging and agent resolution capabilities. A synthetic test is defined to compare the resolution speed

of remote agents against that of local agents. Also the other stages involved during setup like agent creation and registration with the AgentShadowing service are benchmarked. The idea is to simply show the impact of using AgentShadowing in a model but also to check the feasibility of using distributable model code in a non-distributed setup. If the overhead is small enough the general usability of model code is greatly enhanced, since no additional changes are required to create a distributable version.

A second test aims at the communication component inside of the AgentShadowing solution. The three stages of message creation, message serialization and the actual message sending will be tested for several agent amounts. As a baseline the test will be run on a Windows machine with native .NET 4.5.

This experiment relates to hypothesis H3 as it highlights the impact of the actual distribution component in MARS LIFE free of side effects from any model implementation.

4.2.3 EXP3: Test of KNP model initialization on 1, 2 and 3 nodes

To test hypotheses H5 and H6 two initialization benchmarks will be conducted. All will use the full KNP model and will be executed on 1, 2 and 3 nodes to measure scalability. The difference is in the way ShadowAgents are being created. On the first run they will be created during initialization, on the second run this will be omitted and ShadowAgents are created on-demand during simulation. The better option will be used for the remaining experiments.

4.2.4 EXP4: Test of KNP Model with single, central ESC on single node

As a baseline test it is important to run the full scale KNP model on a single node. The moving agents (elephants) will use the Environment Service Component (ESC) very frequently, which makes the ESC's performance crucial to the overall execution speed. So this experiment will act as a baseline not only for overall simulation performance, but also for the different possible setups of the ESC. The ESC instance is a non-distributed one which uses a bounding volume hierarchy as its internal data structure and runs without collision detection for movement. The collision detection feature is turned off, because its impact on performance is so heavy that it wouldn't make any sense to execute a simulation with more than 5.000.000 agents on it. Also the example model does not need fine grained collision detection for the movement of its entities, since the temporal resolution is set to one hour.

This test will be run on a full scale VM representing a single Mac Pro in the MARS Cloud infrastructure and on a reduced VM featuring half the resources to relate to hypothesis 2 and the question of vertical scalability.

4.2.5 EXP5: Test of KNP Model with polyglot ESCs in each layer on single node

This test is about the comparison of performance between a centralized ESC for all agents, against a polyglot ESC usage in every layer. What that means is that opposed to storing all agents inside a single ESC instance, each layer has its own ESC and only stores agents residing on that layer in it. This comes with a number of possible benefits. The number of agents per ESC is reduced and thus the complexity of the internal data structure. Also the ESC component does not have to perform costly type checks, when an exploration is performed. Finally each layer may have another internal implementation of the ESC. For instance the waterpoint and marula tree layers only host non-moving agents and thus may avoid ESC implementations that support movement, which usually comes at a cost, since the internal data structures (usually some sort of tree) has to be rearranged.

4.2.6 EXP6: Test of KNP Model with distributed, single ESC on 2 and 3 nodes

The first distributed test will be run with a logically central ESC that manages all agents of every type. The ESC is configured to be distributed via replication and is instantiated once on every node.

When distributing a simulation with MARS LIFE a decision has to be made regarding which layer goes where and whether is is distributed, replicated or kept as a whole. For the sake of this and the following distributed experiments a setup as shown in figure 4.5 is used. The Marula trees are equally distributed across all nodes, since there are 5,2 millions of them. Waterpoint and elephant agents reside on the same node in a non-distributed manner, since they are in close interaction and are rather few agents (15.415 combined). The elevation and temperature layers are data layers and thus provide access to locally available datasets (elevation) or act as facades and caches to database instances running in the MARS cloud (timeseries layer). Therefore these layers are replicated on all notes, so as to provide fast local access.

4.2.7 EXP7: Test of KNP Model with polyglot ESCs in each layer on 2 and 3 nodes

EXP7 is the same test as EXP5 but in a distributed deployment on 2 and 3 nodes. This means the used ESC instances are of another type. A distributable ESC is used, which implements distribution in a rather naive way: Every instance of the ESC is a master replication, which is achieved by sending every action performed on any instance to all other instances by means of a multicast message. For the KNP scenario and the used distribution setup this results in a distributed ESC for Marula trees. Every added or removed marula tree from any instance will result in a message being send to the two remaining instances in order to update their internal state. Explorations performed on any node are also propagated to the remaining nodes and the result is aggregated.

4.2.8 EXP8: Test of KNP model with Result WriteOut including the best options from the above tests

The impact of writing out results from a simulation needs to be analyzed. Having millions of agents will results in a lot of data to be written out for each simulation step. Currently MARS features a solution, which simply outputs everything for every agent after each tick has been executed. Jan Dalski, the master student, working on this feature recently came up with significant performance improvement ideas, but these haven't been implemented in MARS as of the writing of this paper. So the current state will be monitored while acknowledging that a potentially better solution is coming.

4.2.9 Summary

Table 4.1 shows how the the different experiments match to the hypotheses from chapter 1.2. Given the complexity of MARS and the multitude of parameters responsible for the outcome of an experiment, most hypotheses depend on multiple exepriments.

Hypothesis to Experiment Mapping		
Hypothesis	Related Experiment	Comment
HYP 1	EXP 1	Hypothesis 1 can be tested by simply benchmarking the key performance indicators with sysbench.
HYP 2	EXP 2 EXP 4	A part of EXP 2 tests the AgentShadowing's agent resolving methods. These are crucial for scalability. EXP 4 is directly designed for this hypothesis.
HYP 3	EXP 2 EXP 3 EXP 5 EXP 6 EXP 7 EXP 8	Testing AgentShadowing is crucial so EXP 2 is relevant. Initialization of a model should also scale hence EXP 3 is crucial. EXP 5 and EXP 6 are the experiment which directly compare simulation runs on 1, 2 and 3 nodes.
HYP 4	EXP 4 EXP 5 EXP 6 EXP 7	HYP 4 uses almost the same experiments as HYP 3, with the difference that results are compared against EXP 4. This allows to draw conclusions regarding the ESC's usage.
HYP 5	EXP 3 EXP 2	Agent creation is especially crucial during initialization and in between simulation ticks. Therefore results from EXP 2 and EXP 3 relate to the question whether on-demand creation of new agents during runtime is feasible.
HYP 6	EXP 3	EXP 3 is designed for HYP 6

Table 4.1: Mapping of experiments to hypotheses.

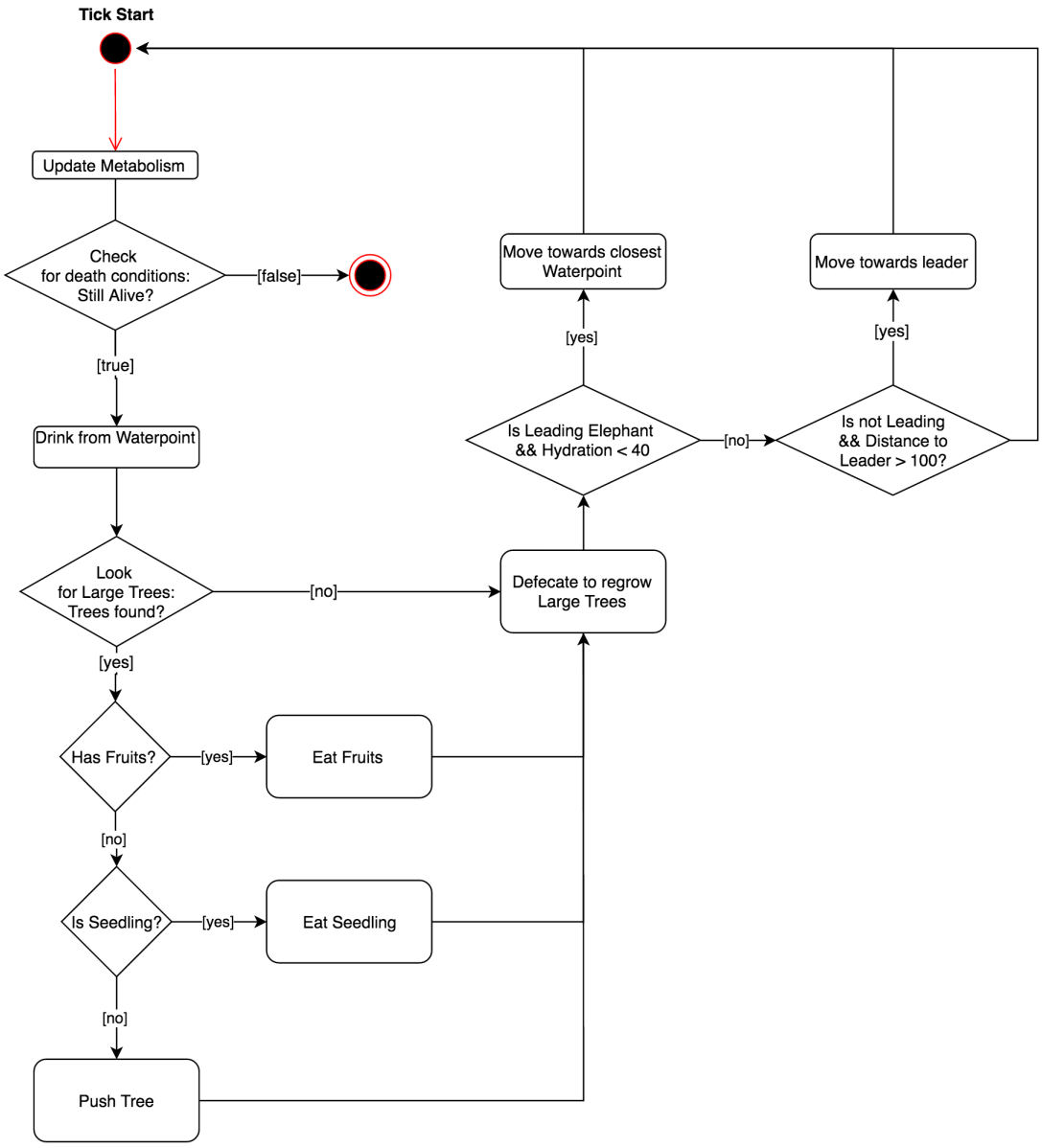


Figure 4.4: Activity diagram for internal elephant logic.

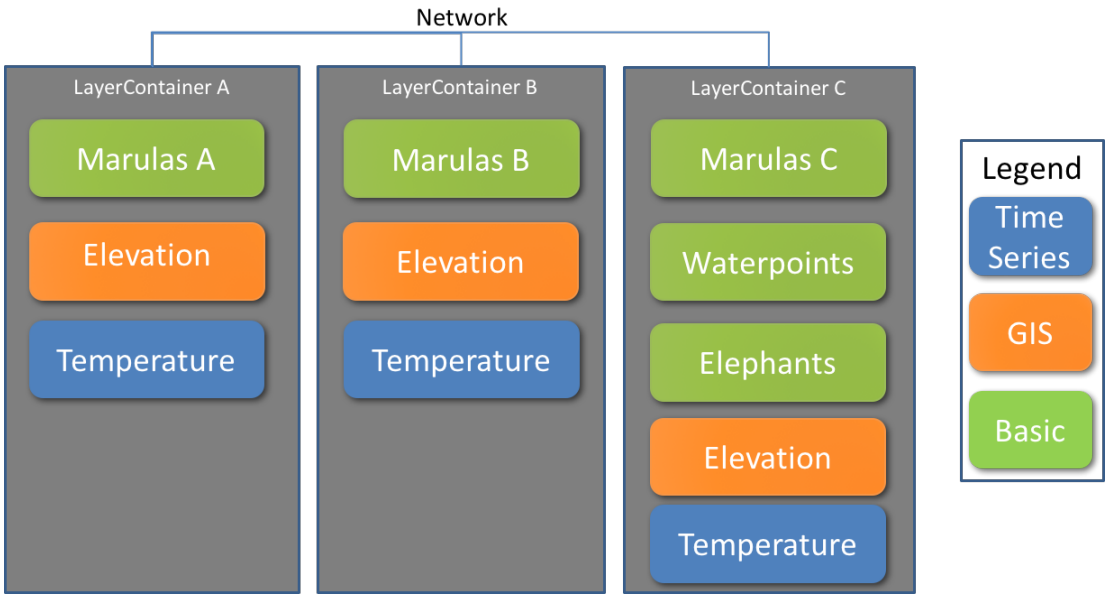


Figure 4.5: Layer Configuration used for benchmarking of distributed simulation run

5 Results

5.1 EXP1 - Performance Impact of Virtualization layers.

Once the whole infrastructure was set up, experiment 1 was rather easy to execute. The three stages bare-metal, VM, Docker on VM were performed one after another but without the not needed technology running. That is no virtualization was present when the bare-metal node was benchmarked and so on. The results as seen in figures 5.1 and 5.2 clearly show how small

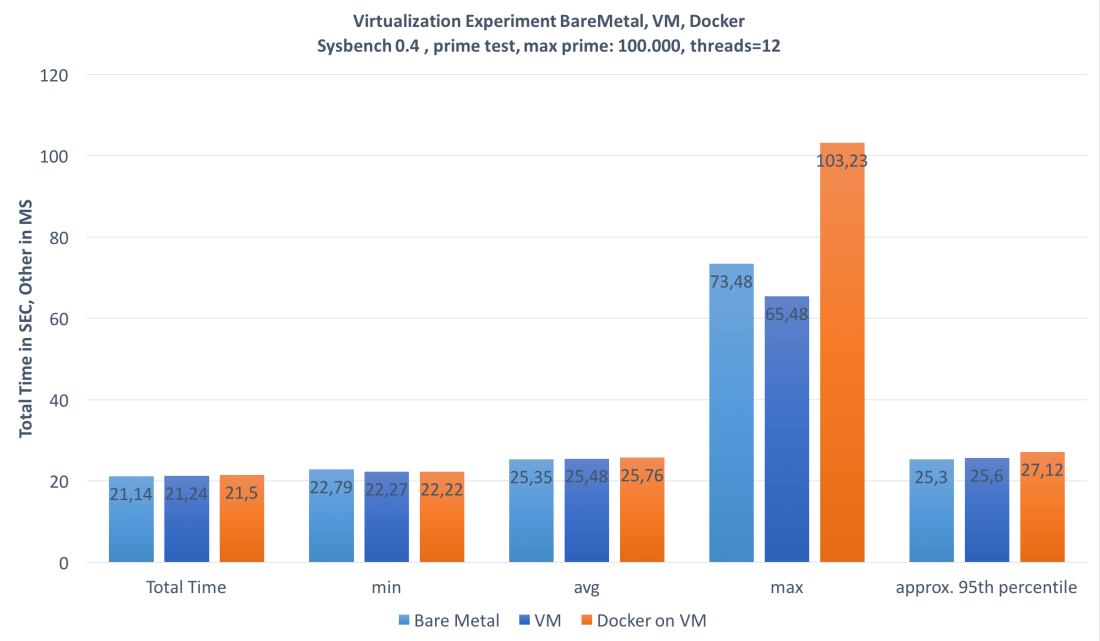


Figure 5.1: Benchmark BareMetal vs. VM vs. Docker.

the impact of virtualization is on overall CPU performance. The 95th percentile difference between Docker and bare-metal is only about 1,82 ms per request while the overall execution time of the benchmark differs by 0,36 seconds. The peaks in the 'max' category are repeatable and originate from the initial process / thread allocation by the underlying kernel. While the bare-metal execution needs to create a new process, the VM schedules the run on a vCPU

which already is present as a process on the host system. Docker finally has to make its way through the LXC kernel extension in the VM and then onto the host system, which causes the greater delay. On average the difference between Docker and bare-metal is only about 0,41 ms per request. Looking at the absolute values executing on bare-metal clearly is the fastest option (compare color coded table 5.2), but given the amount of flexibility and the benefits during development time that are created through virtualization, this is a very acceptable impact on overall performance.

Sysbench, 12 threads, max_prime: 100.000 HW: 6 cores + HT, 64 GB Ram				
	Bare Metal	VM	Docker/VM	
Total Time	21,14	21,24	21,5	secs
Per-Request stats				
min	22,79	22,27	22,22	ms
avg	25,35	25,48	25,76	ms
max	73,48	65,48	103,23	ms
approx. 95th percentile	25,3	25,6	27,12	ms

Figure 5.2: Benchmark results for BareMetal vs. VM vs. Docker. Results are row-wise color coded with slowest time being red.

5.2 EXP2 - Agent Shadowing Standalone

5.2.1 Benchmark 1 - Local Behavior

The first benchmark in this experiment measures the time needed to resolve a local and a remote agent via the AgentShadowing service. Also the other stages involved in the usage of AgentShadowing are benchmarked. These include creation and registration of agents as well as calling a method on each agent. Measurements are taken with the high-resolution Stopwatch class built into C#.

Results

Figure 5.3 shows the linear scaling behavior of all four aspects. The two setup aspects are a little above 1 second for 200.000 agents, while the runtime aspects of resolving agents and calling a method on them are significantly faster at around 100ms and 1ms respectively. The method calling values for 20 to 20.000 are not visible in the diagram because their values are reported as 0 ms by the Stopwatch and the y-axis is logarithmic. Figure 5.4 shows all test results in a table view.

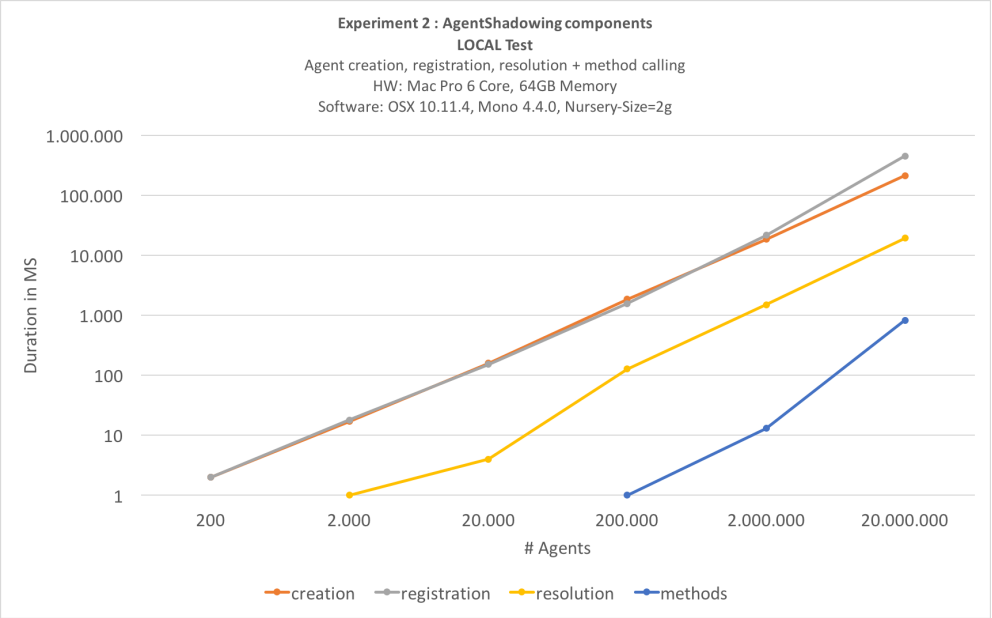


Figure 5.3: AgentShadowing component test in non-distributed scenario.

5.2.2 Benchmark 2 - Network Behavior

The second benchmark is designed as a white box test of the AgentShadowing's components. Each required step is executed manually and measured independently of the others. The steps involved are message creation, serialization and actual sending. The overall total time of performing the whole process is also measured.

Message creation is about turning method calls into messages, which involves basic object creation and value transfers.

As a serializer the BinaryFormatter from .NET/Mono is used. This serializer is by far not the fastest available (i.e. compared to Protobuf.NET), but it is the most convenient as it is capable of serializing every .NET type without any mapping. The idea is to start model development with a non-optimal serializer, and tune it later, when the needed types are defined in the agent code.
The message sending finally is implemented with the AsyncSocketEventArgs pattern from Microsoft, which is their approach towards high-performance socket applications: "The SocketAsyncEventArgs class is part of a set of enhancements to the System.Net.Sockets.Socket class that provide an alternative asynchronous pattern that can be used by specialized high-performance socket applications. This class was specifically designed for network server applications that require high performance.(...)" (from: https://msdn.microsoft.com/de-de/library/system.net.sockets.socketasynceventargs(v=vs.110).aspx).

Local Test Agent creation, registration, resolution + method calling HW: Mac Pro 6 Core, 64GB Memory Software: OSX 10.11.4, Mono 4.4.0, Nursery-Size=2g				
#agents	creation	registration	resolution	methods
200	2	2	0	0
2.000	17	18	1	0
20.000	159	152	4	0
200.000	1860	1568	128	1
2.000.000	18624	21735	1503	13
20.000.000	213490	452766	19482	826

Figure 5.4: AgentShadowing component test in non-distributed scenario. All values in ms.

The test implementation involves two sets of components for two distinct sets of agents in order to simulate two nodes talking to each other. During the test each agent set attempts to call a method on an agent on the other node, but each step is executed explicitly by the code. The test is run for 20, 200, 2.000, 20.000, 200.000 and 2.000.000 agents. Given the fact that Mono is a reimplementation of .NET for Windows, the tests were run on a windows machine as well to get a baseline to compare against.

Results

Figures 5.5 and 5.7 show the results for Mono on OS X and .NET on Windows respectively. The Mono benchmark shows a linear increase in duration from 2000 agents onward but fails with a low level stack overflow exception from within the Mono framework itself when trying to execute the run for 2.000.000 agents. The cause of this exception remained unclear, but seems to result from a bug in the mono runtime.

From 200 to 2.000 agents there is a sudden increase in execution duration at the order of two magnitudes, which was reproducible across multiple runs. The Windows benchmark shows a steady linear increase from 2.000 agents onwards and also executes flawlessly with 2.000.000 agents. The absolute overall execution duration though is almost always faster by two orders of magnitude on Windows as shown in figure 5.10 even though the used hardware is a lot weaker. This becomes very evident when directly comparing the values from tables 5.6 and 5.8 for 200.000 agents. 682.452ms on Mono compete against 10.636ms on .NET. So Mono is more than 64 times slower than .NET in this test.

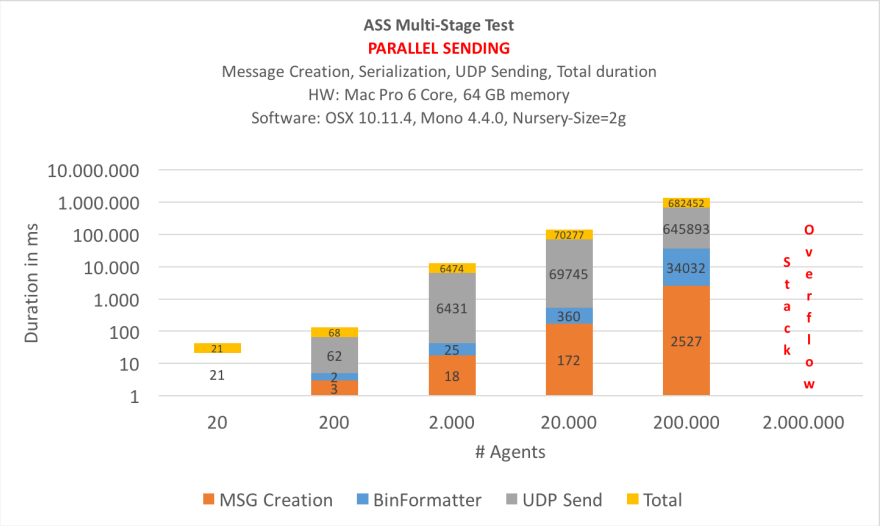


Figure 5.5: AgentShadowing Communication Benchmark separated by components. Ran on Mono

When searching for a possible cause of this behavior it is worth noting that the messaging ("UDP Send" column in tables 5.6 and 5.8) share of overall execution duration on Mono is 94,64 % while on .NET it is 91,17 %. So messaging itself takes a similar percentage of the time. Looking at the message creation and serialization values reveals that Mono takes 40 times the time of .NET here as well. Given that these tasks are implemented in a way that relies heavily on multi-threading and thus should make use of all available cores, the problem could be situated in Mono's Parallel Task API implementation.

Measuring the CPU and ethernet bandwidth usage on Windows during the benchmarks' execution reveals that the Windows machine is probably locked by its CPU at 90+ % utilization (figure 5.11) resulting in 78 Mbit/s throughput (figure 5.12) on the 100Mbit ethernet device, while the OS X machine with Mono rarely reaches a 50 % CPU load (figure 5.13) and peaks at around 1,5 Mbit/s in throughput. It is worth noting that the drops in CPU usage are mostly visible during the UDP sending part of the test. The other tasks execute with a steady and more reasonable CPU load (compare figure 5.14).

It can therefore be concluded that the Mono implementation of System.Net.Sockets and most probably the Parallel Task API as well have serious performance and scaling problems, which are not present in the .NET Framework on Windows. Furthermore as indicated in figures 5.5

ASS Multi-Stage Test
Message Creation, Serialization (BinForm), UDP Sending
HW: Mac Pro 6 Core, 64GB Memory
Software: OSX 10.11.4, Mono 4.4.0, Nursery-Size=2g

#agents	MSG Creation	BinFormatter	UDP Send	Total
20	0	0	21	21
200	3	2	62	68
2.000	18	25	6431	6474
20.000	172	360	69745	70277
200.000	2527	34032	645893	682452
2.000.000	Stack Overflow			
20.000.000				

Figure 5.6: AgentShadowing Communication Benchmark separated by components. Ran on Mono. All values in ms.

and 5.6 the Mono implementation crashed with a stack overflow exception on attempting to run 2.000.000 agents or more. The corresponding error messages and stack traces hint towards a low level bug issue in the System.Net.Threading API.

5.3 EXP3 - Test of KNP model initialization on 1, 2 and 3 nodes

The first run of the initialization test was the same for both setups, since on a single node no ShadowAgents need to be created. Tests for 2 and 3 nodes showed some pretty interesting results though (figure 5.15). While the on-demand version scales nicely with more nodes added (mean factor: 1,79) the pre-init variant starts tremendously slower (almost 30 minutes) and is rather reluctant to scale with more hardware (factor from 2 to 3 nodes: 1,22). Further investigation of the pre-init process showed at around 1.000.000 agents that the initialization of the ShadowAgents turns into a very long running task even on the native .NET platform on Windows. Performance tracing with dotTrace (https://www.jetbrains.com/profiler/) revealed that the .NET event system is responsible, since the time it takes to add a listener to events in the ShadowAgents's underlying implementation increases with each agent. This behavior has been expected to some extent (see 3.3.4) and on account of the results, it is recommended to completely switch over to on-demand ShadowAgent creation. The TTL of each ShadowAgent prevents the listener problem while still maintaining convenient communication. For all remaining tests on-demand creation will be used.

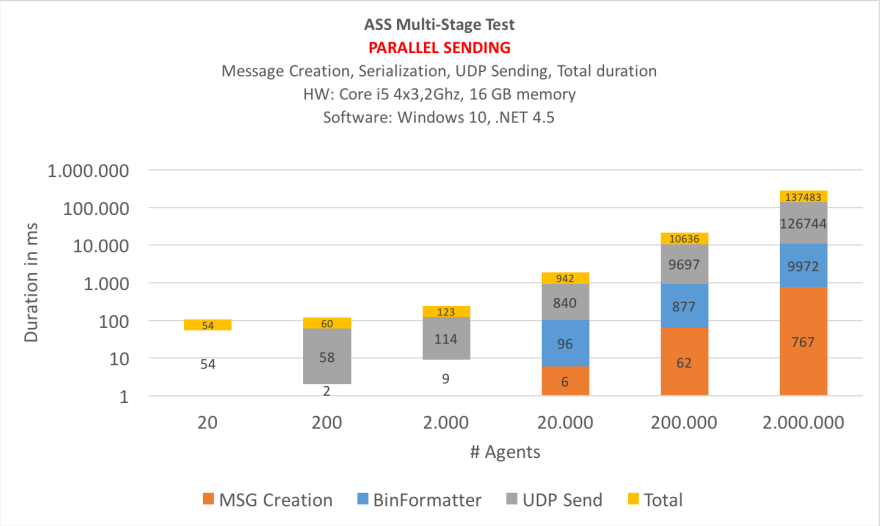


Figure 5.7: AgentShadowing Communication Benchmark separated by components - Windows

5.4 EXP4 - KNP Model with central ESC on single node

This experiment has been executed two times. The first run was allowed to utilize all available resources of the hosting VM, which translates to 6 cores and 60 GB of memory. The second run had those resources cut in half to measure the vertical scalability of MARS LIFE for the given model. The used hardware was restricted by the Docker tools, which allow to reduce a containers resources to a specific level or ratio. As figure 5.16 shows there is a measurable improvement in execution speed when resources are increased. The average improvement in execution duration when doubling the hardware is about 0.76 times that of the baseline, which in return is 1.32 times slower than running on full resources. So the speed bump is a bit less of half of what could be expected in an ideal case. Considering the problems introduced by the Mono runtime in experiment 2, this is likely to indicate an introduced overhead in Mono for scheduling more threads on more cpu cores and handling additional locks throughout the simulation.

5.5 EXP5 - KNP Model with polyglot ESCs in each layer on single node without result output

The execution of the KNP model on MARS LIFE is divided into two parts. During the initialization required data is fetched from databases located in the MARS Cloud. When everything is set

ASS Multi-Stage Test Message Creation, Serialization (BinForm), UDP Sending HW: Core i5 4x3,2 Ghz, 16GB Memory Software: Windows 10, .NET 4.5				
#agents	MSG Creation	BinFormatter	UDP Send	Total
20	0	0	54	54
200	0	2	58	60
2.000	0	9	114	123
20.000	6	96	840	942
200.000	62	877	9697	10636
2.000.000	767	9972	126744	137483

Figure 5.8: AgentShadowing Communication Benchmark separated by components. Ran on Windows. All value in ms.

up the simulation may run I/O free, in case it does not produce any output as in this benchmark.

Results for this benchmark are shown in figure 5.17. During initialization for every agent type the constructor parameters are analyzed and matched to the data retrieved from the MARS Cloud. Also a number of type checks need to be performed for every parameter value to reflect special cases like required agent IDs, instances of IEnvironment or other layer types, which are needed as reference. Finally all static values retrieved from the provided SimConfig need to be parsed into their corresponding C# type. The initialization phase took almost 14 minutes, which is considered an acceptable value given that data for 5.2 million agents is fetched from the MARS Cloud and the above mentioned process needs to be executed for that amount of agents.

The CPU utilization is very feasible during this procedure (around 75% mean.). Once initialized each simulation step takes around 145 seconds with a CPU load between 85 - 98 %. Since there is no distribution and communication involved in this benchmark the socket problems discovered in EXP2 do not have their negative impact on performance.

It is worth noting that performance, CPU usage and overall execution were far worse, when the Mono garbage collector was used with its default settings. The process had to stop every couple of seconds to perform a garbage collection. The root source of this problem resides in the Mono garbage collector implementation, which is a generational one. It contains three generations for objects in memory. The first and smallest is the "Nursery", which holds very small and young objects. Its default size is 4 MB. Once the nursery is filled with small objects, a garbage collection has to be performed. During the process all no longer referenced objects

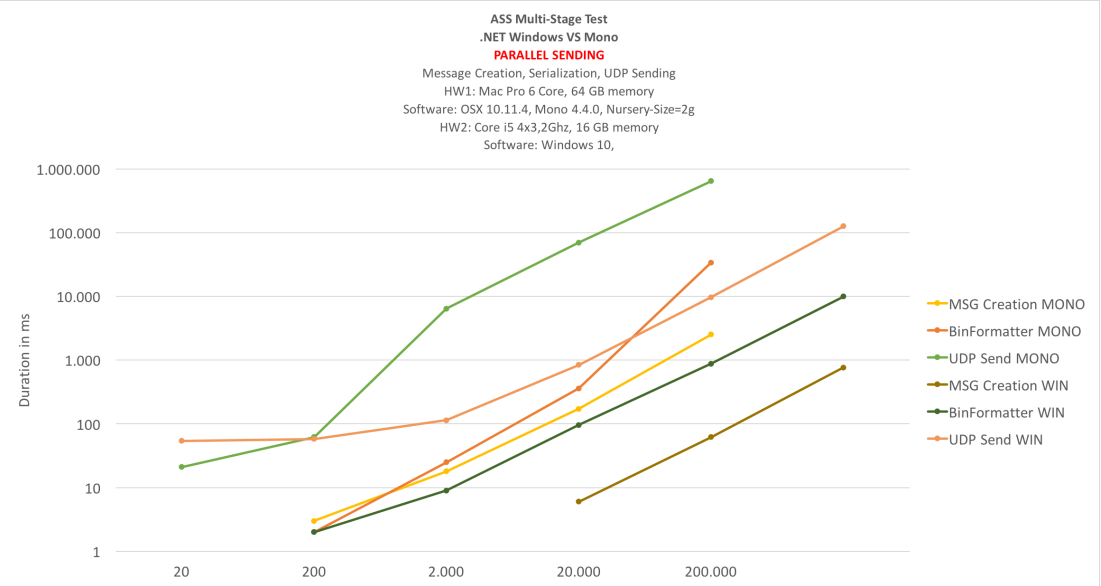


Figure 5.9: Staged performance evaluation of AgentShadowing - Mono vs. Windows.

are being wiped out from the Nursery, while all still used objects are moved into the second generation, which is where long living objects are stored. Lastly generation three is the large object store. Objects with a size above a certain threshold are moved here directly without having to go through the Nursery.

Tweaking the nursery-size from 4MB to 2GB resulted in the improved performance as described above and should be taken as the standard setting when executing any considerably sized model. Though this setting results in a slightly larger memory footprint of the mono process, this effect can be ignored given that the simulation run consumed around 50 GB of memory anyways. However even with the optimized garbage collection mechanism the simulation had to halt every so often. More sophisticated garbage collector implementations like the one used by .NET on Windows are capable of performing far better. For instance .NET has a server mode for garbage collection, which creates a dedicated garbage collection thread and heap for every core the machine has. By that garbage collection can be handled independently on different cores.

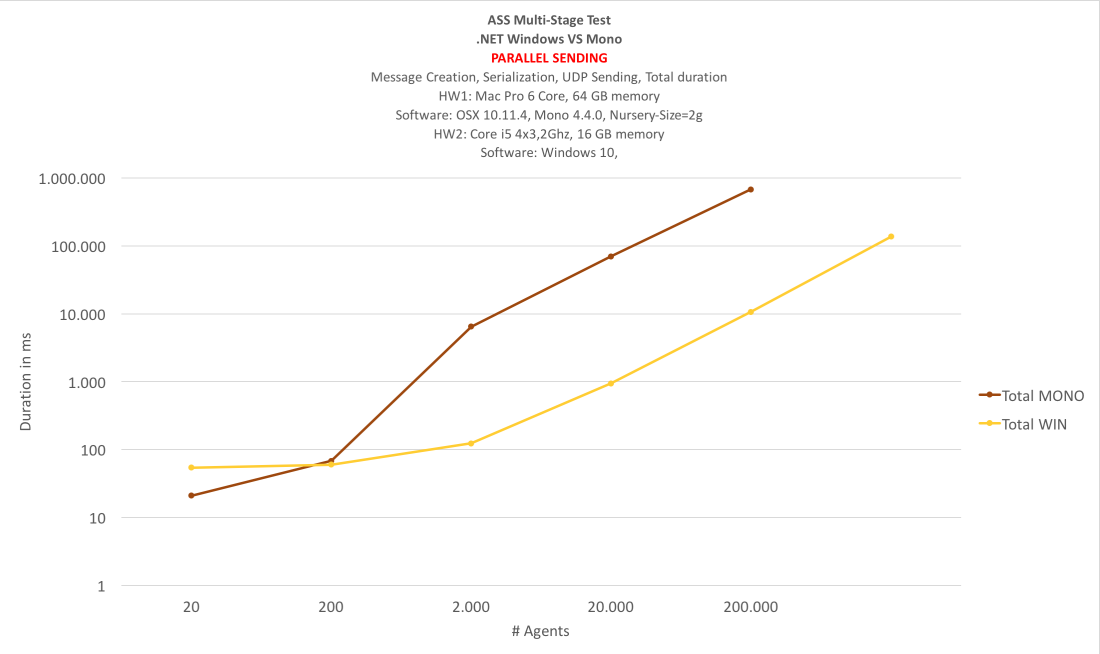


Figure 5.10: Total runtime performance evaluation of AgentShadowing - Mono vs. Windows.

5.6 EXP6 - KNP Model with distributed, single ESC on 1, 2 and 3 nodes without result output

It was not possible to run this experiment. As it turned out, the attempt to distribute a single, logical instance of the current ESC implementation ran into type loading errors. The problem occurs when an exploration is performed by an agent and replicated to all ESC instances. The exploration will eventually return a result, which contains agents of a type, whose Assembly and Namespace have not previously been loaded in the current C# runtime's app domain. This is due to agents of that type are not originally running on the current node and thus relates directly to the distribution configuration used for this simulation run.

As an example consider a setup as shown in figure 4.5 and the exploration of an elephant agent from LayerContainer 'C', which contains a waterpoint agent as a part of the result. This call will fail on LayerContainer 'A', since the type 'WaterPointAgent' is not present on this node and thus the ESC instance does not know how to treat the type.

Effectively this means that currently it is not possible to have a single, logical, but distributed

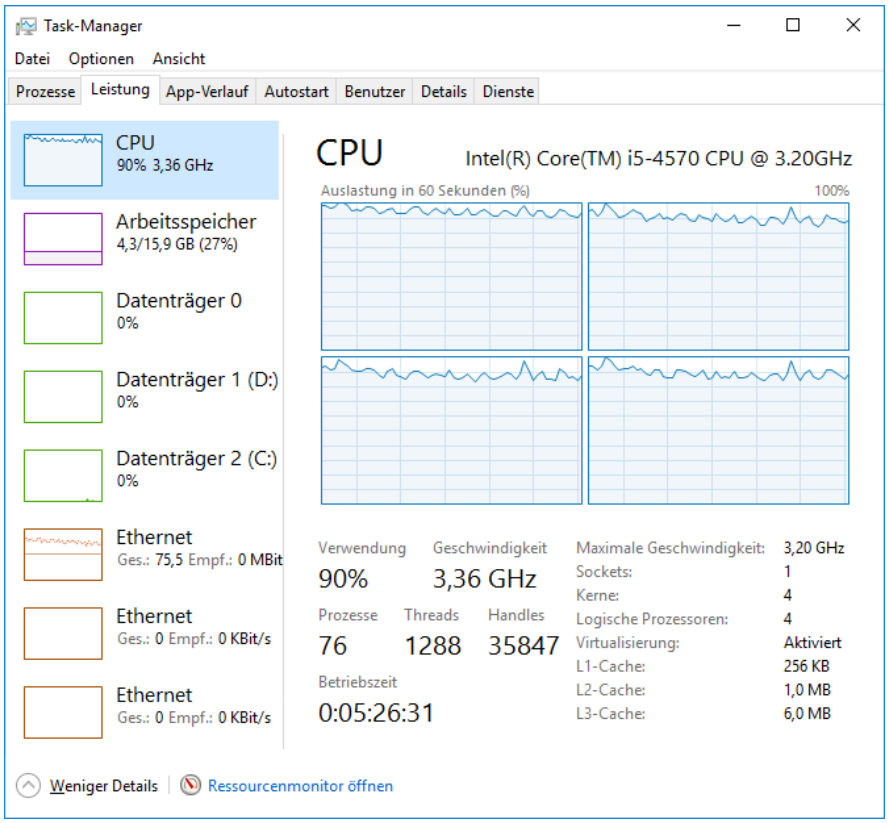


Figure 5.11: CPU load during ASS Multi-Stage test on Windows

ESC instance, which is collectively being used by all agent types. The only way to achieve this in a distributed scenario would be by using a layer which hosts a single, non-distributed ESC and exposes the IEnvironment interface as its layer interface. That approach however would be the least performant of all, since every environmental request from every agent not being hosted on the same node as the environment layer would need to be serialized and send over the network. Also as the model grows the amount of memory required by the central ESC would be quite considerable.

5.7 EXP7 - KNP Model with polyglot ESCs in each layer on 1, 2 and 3 nodes without result output

The results of the distributed run across 1, 2 and 3 nodes are presented in figures 5.18 and 5.19. Initialization on 3 nodes is 1,7 times faster than on a single node, and takes around 8,08 minutes

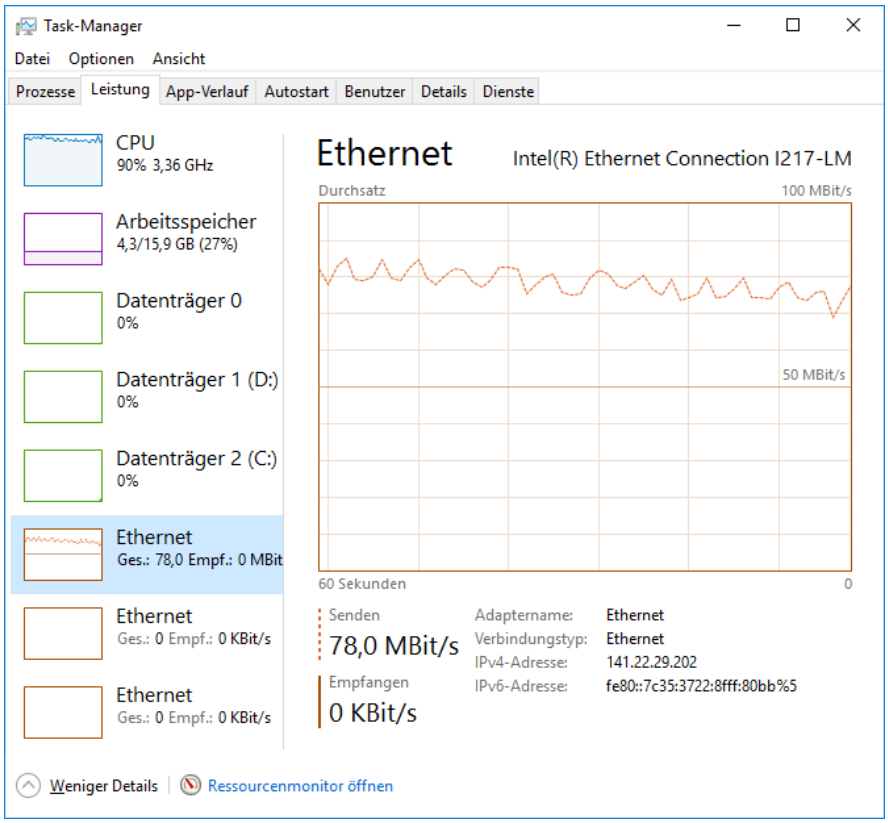


Figure 5.12: Ethernet load during ASS Multi-Stage test on Windows

for 5.2 million agents. The execution speed during simulation is up to 2,32 times larger when run on 3 nodes compared to running on a single node. The performance gain could possible be significantly improved by running the distributed simulation on 3 windows nodes, considering the results from experiment 2 regarding Mono's performance problems.

When observing the values for 2 nodes in figure 5.19 the discordant values for ticks 4, 6, 7, 8 and 9 are noticeable. The same phenomenon can be seen for ticks 0, 4 and 7 for 3 nodes. This sudden increase in execution duration during the simulation shows the impact of agent interaction, which might peak during some ticks. Since the simulation is non-deterministic this is quite expected.

Table 5.19 also shows the projected duration for each setup in hours on the assumption that scenario time covers a full year, which translates to 8760 ticks at an hourly time step. As it turns out the simulation would run for almost 15 days on a single node, 11 on two nodes

and still would take 6 days on three nodes. In a production environment this is not really practicable given that real scenarios are even more complex and might well get larger regarding scale and the amount of agents. Considering the findings from experiment 2 (compare section 5.2.2) this simulation can be expected to execute up to 64 times faster when run on the .NET platform instead of on Mono. This theoretically would bring down projected durations to 6, 4 and 2,3 hours respectively, which are very acceptable values.

One goal of this work is to measure the resource consumption of specific components. Thus the ESC share of overall execution duration especially in a distributed scenario is of particular interest. To measure that impact a run distributed across 2 nodes but with a non-distributed ESC has been performed in order to compare against the original setup with a polyglot ESC. The results are shown in figure 5.20 and reveal that the average negative performance impact of the distributed ESC during execution is 31,8 % and 37,72 % during initialization respectively. So optimizing the ESC towards a non replicated implementation will be well worth it.

5.8 EXP8 - KNP Model with polyglot ESCs in each layer on 1, 2 and 3 nodes with result output

Figure 5.21 shows the results from experiment 8 against those from experiment 7. The first three bars of every tick show the execution duration including output, while the last three bars don't include the output. The actual measured values are shown in table 5.22, where the first three rows do not include output while the last three do. The row labeled "#times faster" is of special interest, since it shows the factor by which the simulation experiment ran faster than the previous one. Comparing these scaling factors of 1.35 and 1.72 for two and three nodes when no simulation output is created and 1,38 and 1,50 with output respectively, reveals that both modes of execution scale within the same range of complexity.

So MARS even scales while outputting simulation results, albeit absolute simulation performance is a lot slower. How much slower exactly is shown in figure 5.23, which highlights that execution speed without output can be up to 4 times faster. To put it the other way around: When output is included it consumes 68,94 % of an average ticks' duration. A complete simulation of the model for its designated wall-clock simulation time of 1 year or 8760 ticks would take around 33 days to finish, when executed with the used configuration including Mono. Just as mentioned in section 5.7 this can theoretically be reduced to around 7.5 hours by using Microsoft's native .NET implementation.

To find the root cause behind the heavy performance impact simulation output has on MARS LIFE further investigations where conducted. To do that the target database (MongoDB) was monitored alongside the CPU utilization of LIFE. As it turned out, the target database is not continuously writing or receiving data when LIFE executes the ResultAdapter responsible for writing the data after each tick. Comparing output from MongoDB activity monitoring against the LIFE Mono process cpu load statistics reveals that when MongoDB is not writing any data the Mono process' cpu load is going down to a single CPU. Figure 5.24 shows the monitored behavior. This indicates a garbage collection (see section 4.1.2), which is a logical behavior since the ResultAdapter is creating a lot of tiny objects that will likely fill up the nursery / 1st generation very fast even though it has 2 GB of space. So the problem of slow result output is as well closely related to the Mono runtime garbage collection behavior.

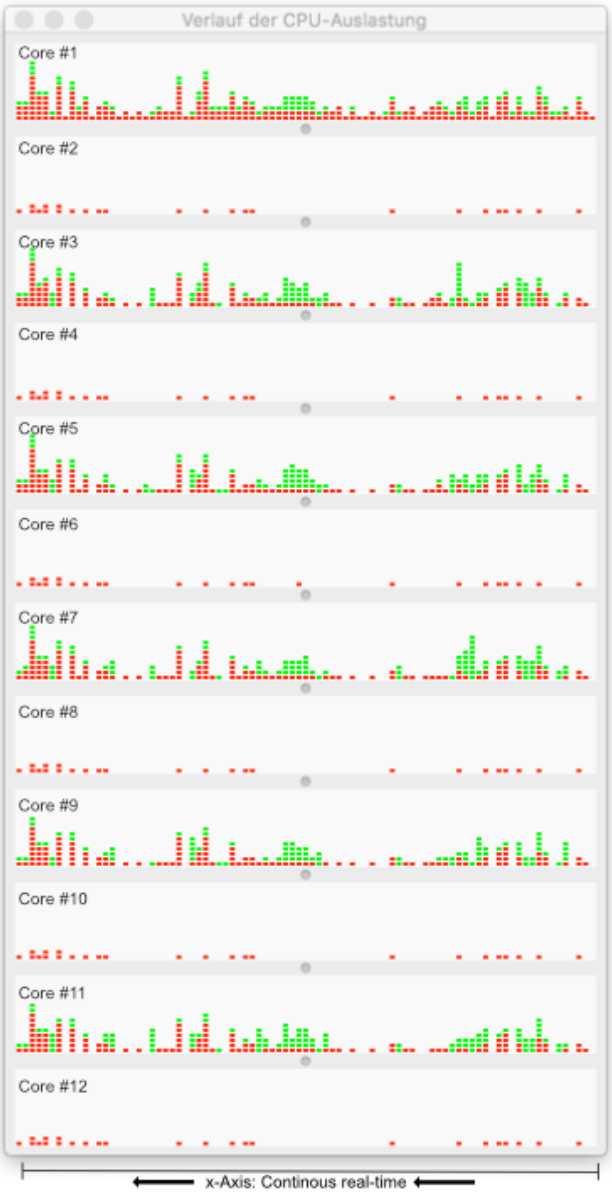


Figure 5.13: CPU load during consecutive runs of AgentShadowing benchmark on Mono. X-axis shows continous time, while y-axis displays percentage of cpu usage per core from 0 to 100.

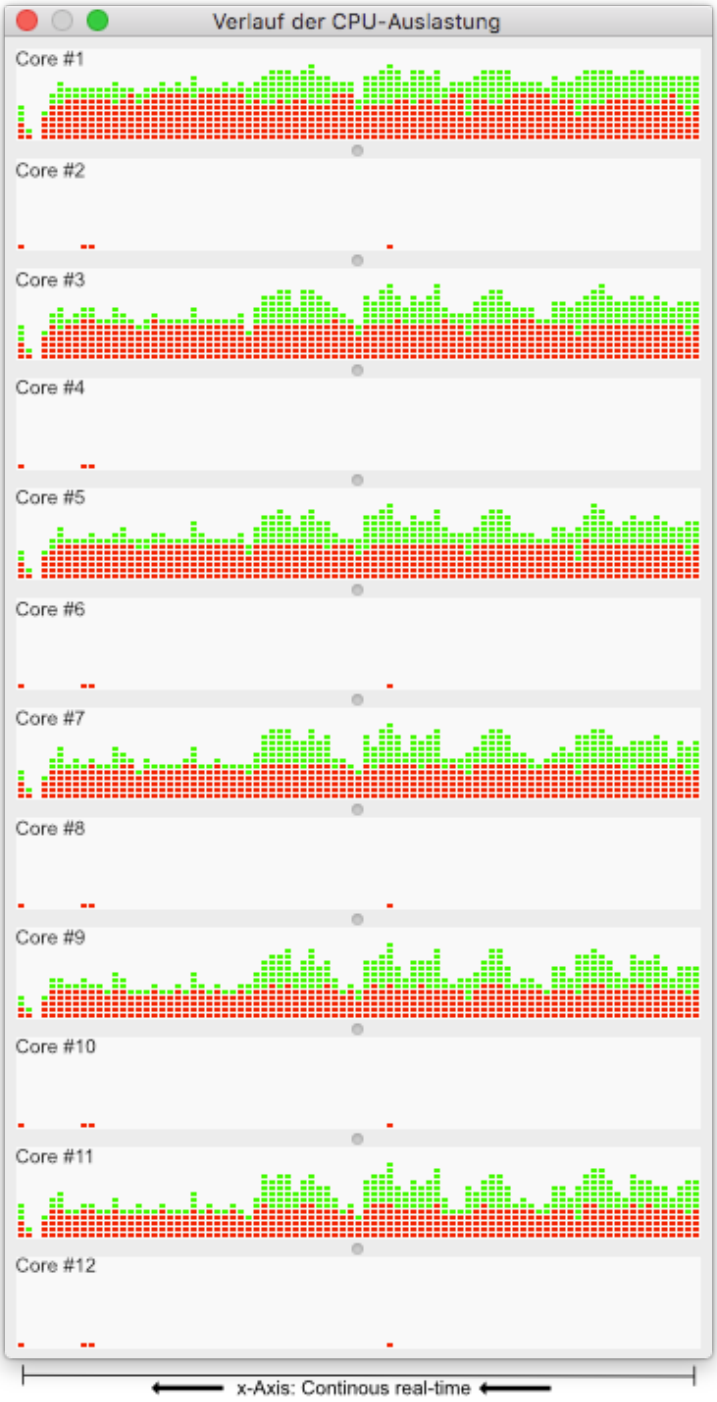


Figure 5.14: CPU load during serialization of 200.000.000 messages with BinaryFormatter on Mono. X-axis show continous time, while y-axis display percentage of cpu usage per core from 0 to 100.

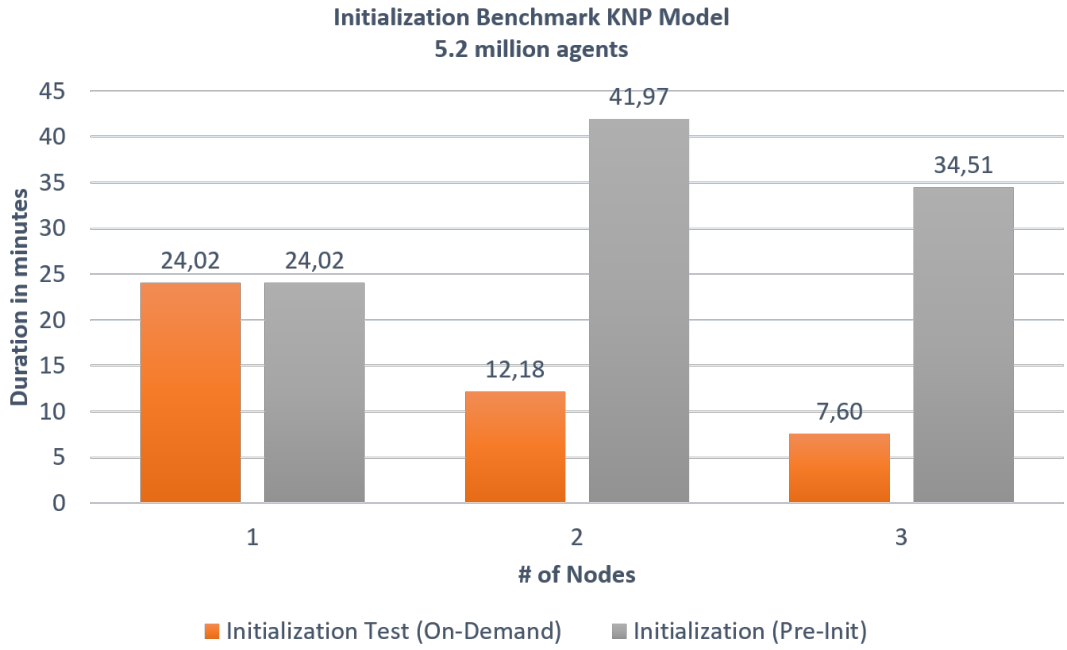
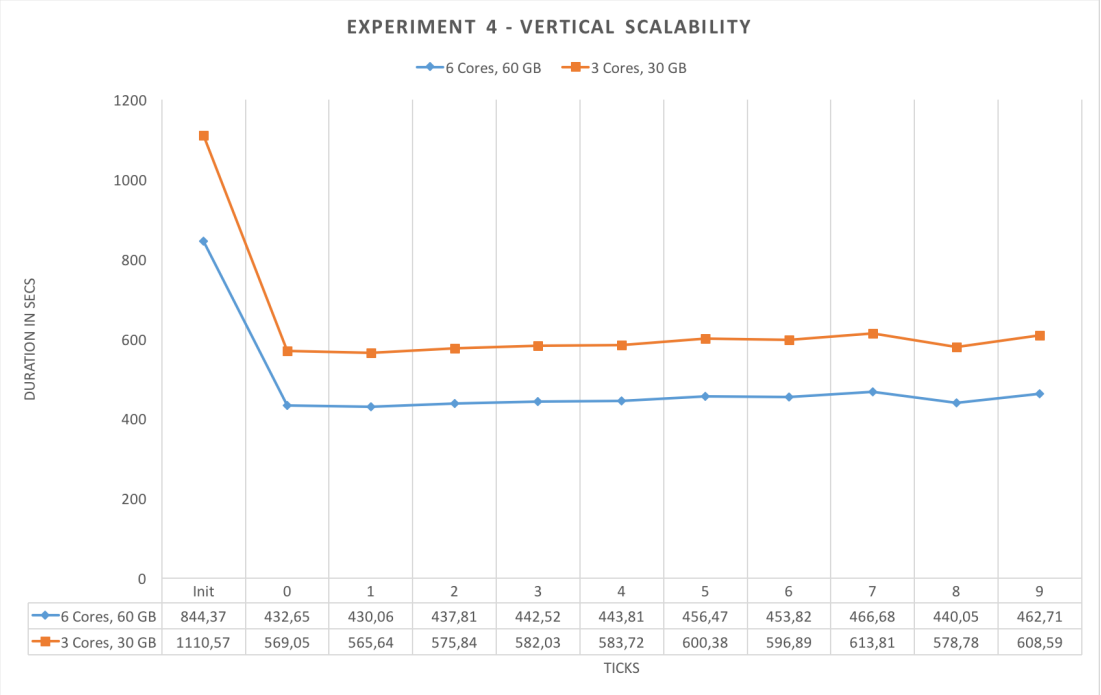


Figure 5.15: Model Initialization with and without on-demand ShadowAgent creation.



	Init	0	1	2	3	4	5	6	7	8	9
6 Cores, 60 GB	844,37	432,65	430,06	437,81	442,52	443,81	456,47	453,82	466,68	440,05	462,71
3 Cores, 30 GB	1110,57	569,05	565,64	575,84	582,03	583,72	600,38	596,89	613,81	578,78	608,59

Figure 5.16: Execution duration comparison for initialization and first 10 ticks for full KNP model on a 6 core, 60GB and a 3 core, 30GB machine. Times are in seconds.

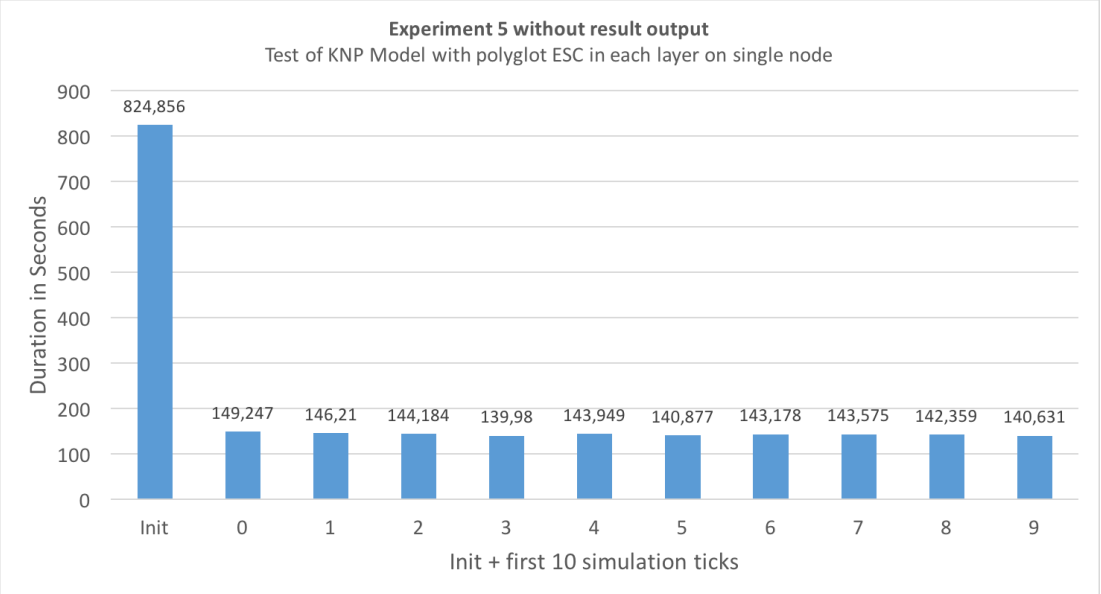


Figure 5.17: Duration of initialization and first 10 ticks of full scale simulation.

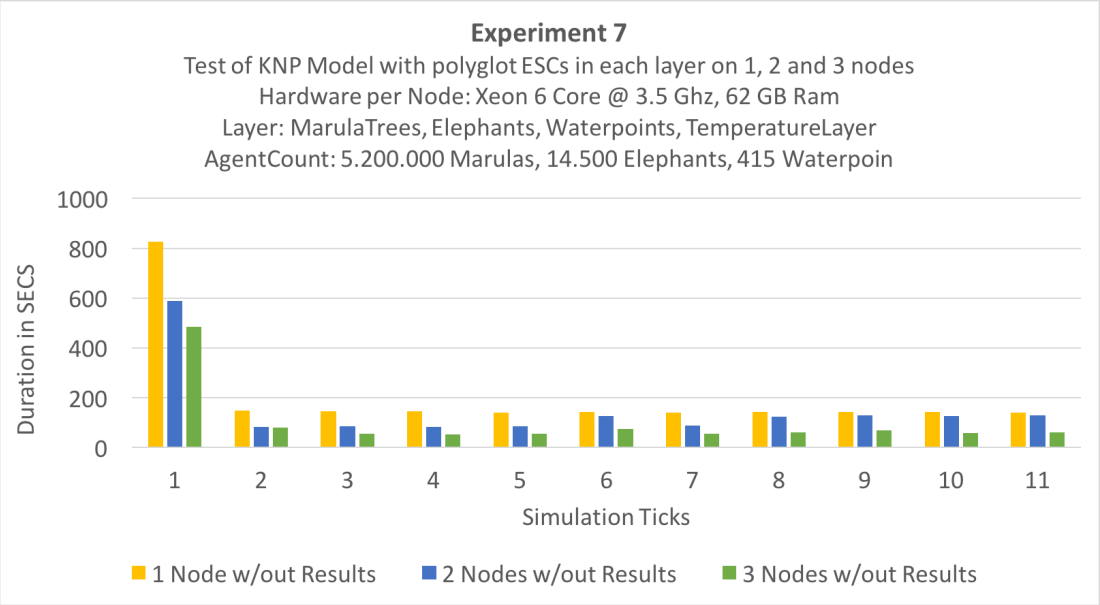


Figure 5.18: Duration of initialization and first 10 ticks of full scale simulation on 1, 2 and 3 nodes.

	Tick	Init	0	1	2	3	4	5	6	7	8	9	
1 Node	Duration	824,86	149,25	146,21	144,18	139,98	143,95	140,88	143,18	143,58	142,36	140,63	secs
	Avg. Duration	143,42											
	#times faster	Baseline											
	Duration	349,22											hours
2 Nodes	Tick	Init	0	1	2	3	4	5	6	7	8	9	
	Duration	587,96	82,90	85,74	83,74	85,05	125,43	88,89	122,43	128,09	127,52	129,94	secs
	Avg. Duration	105,97											
	#times faster	1,35											
	Duration	258,03											hours
3 Nodes	Tick	Init	0	1	2	3	4	5	6	7	8	9	
	Duration	485,63	78,90	54,55	53,05	55,10	74,61	53,94	59,44	69,21	57,34	61,21	secs
	Avg. Duration	61,73											
	#times faster	1,72											
	Duration	150,35											hours

Figure 5.19: Duration of initialization and first 10 ticks, average duration and projected duration in hours of full scale simulation on 1, 2 and 3 nodes.

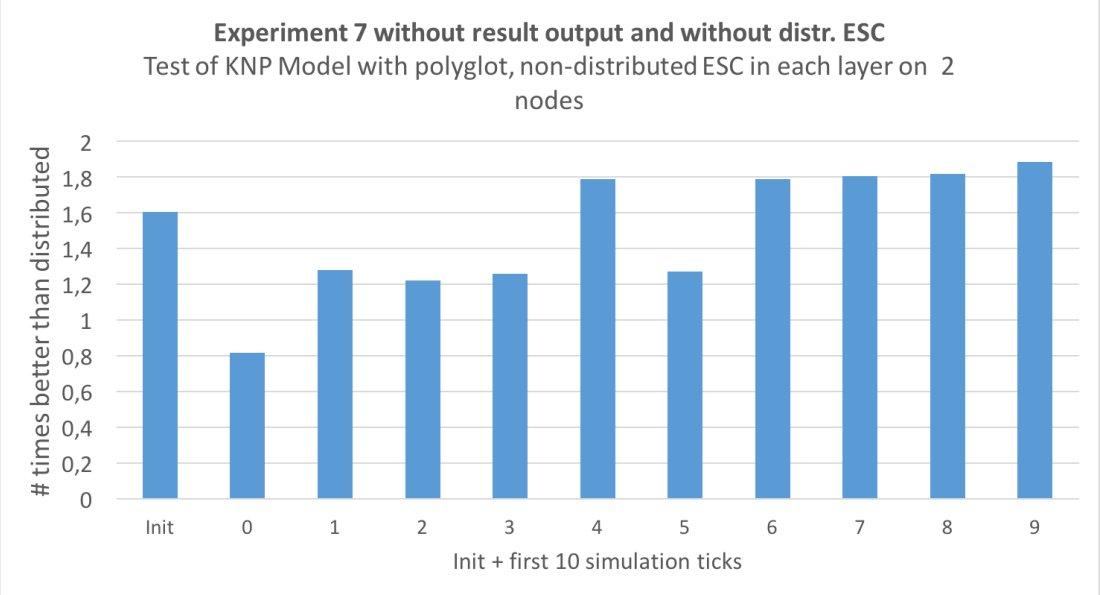


Figure 5.20: Performance impact of distributed ESC during initialization and first 10 ticks of full scale simulation.

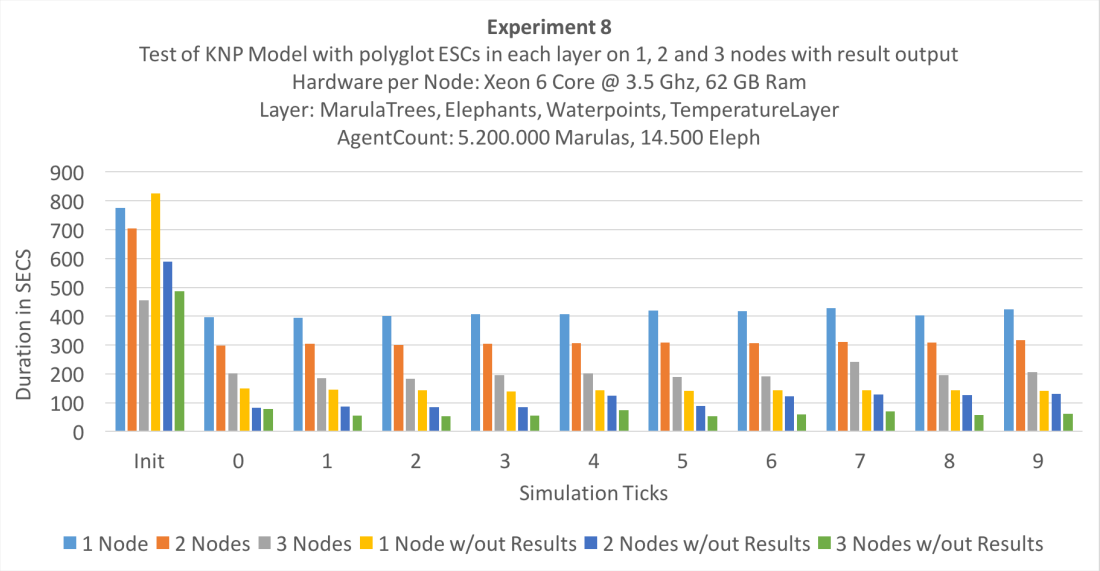


Figure 5.21: Duration of initialization and first 10 ticks of full scale simulation including result output.

1 Node	Tick	Init	0	1	2	3	4	5	6	7	8	9	
	Duration	824,86	149,25	146,21	144,18	139,98	143,95	140,88	143,18	143,58	142,36	140,63	secs
	Avg. Duration	143,42											
	#times faster	Baseline											
	Duration	349,22											hours
2 Nodes	Tick	Init	0	1	2	3	4	5	6	7	8	9	
	Duration	587,96	82,90	85,74	83,74	85,05	125,43	88,89	122,43	128,09	127,52	129,94	secs
	Avg. Duration	105,97											
	#times faster	1,35											
	Duration	258,03											hours
3 Nodes	Tick	Init	0	1	2	3	4	5	6	7	8	9	
	Duration	485,63	78,90	54,55	53,05	55,10	74,61	53,94	59,44	69,21	57,34	61,21	secs
	Avg. Duration	61,73											
	#times faster	1,72											
	Duration	150,35											hours
1 Node w. Output	Tick	Init	0	1	2	3	4	5	6	7	8	9	
	Duration	774,66	396,93	394,55	401,66	405,98	407,16	418,78	416,35	428,15	403,72	424,51	secs
	Avg. Duration	409,78											
	#times faster	Baseline											
	Duration	997,35											hours
2 Nodes w. Output	Tick	Init	0	1	2	3	4	5	6	7	8	9	
	Duration	704,95	297,35	305,17	300,55	304,98	305,82	309,04	306,42	309,84	309,46	317,93	secs
	Avg. Duration	297,3											
	#times faster	1,38											
	Duration	723,63											hours
3 Nodes w. Output	Tick	Init	0	1	2	3	4	5	6	7	8	9	
	Duration	456,01	201,89	184,46	182,71	196,32	201,55	188,61	190,46	240,75	195,66	205,03	secs
	Avg. Duration	198,74											
	#times faster	1,50											
	Duration	483,74											hours

Figure 5.22: Duration of initialization, first 10 ticks, average duration of a tick and projected duration of full scale simulation including result output. Color coding is done separately for the first three rows and the last three.

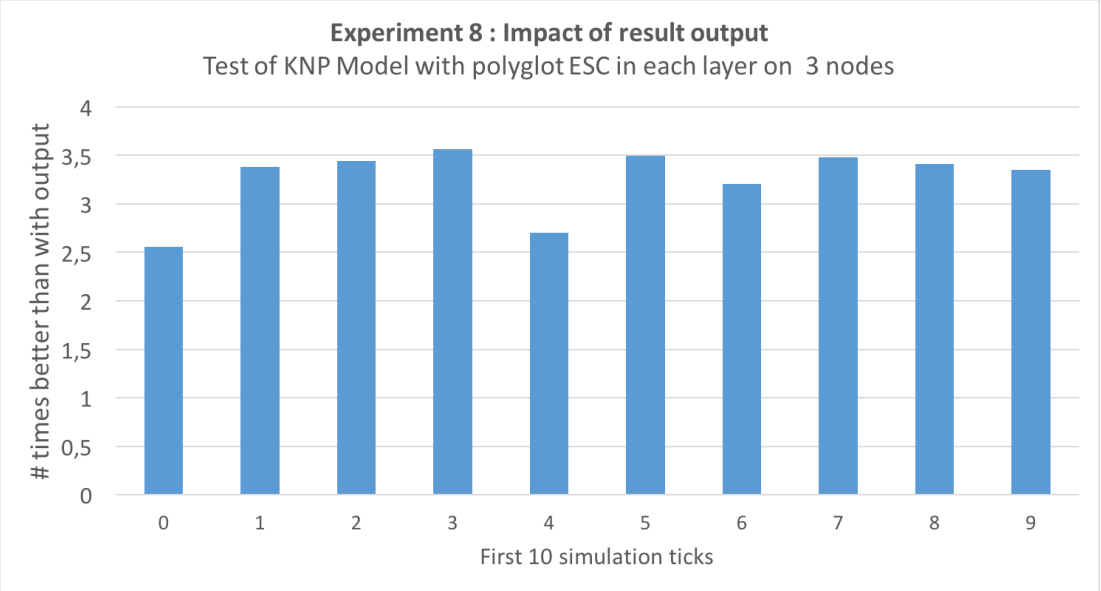


Figure 5.23: Performance impact of result output during first 10 ticks of full scale simulation.

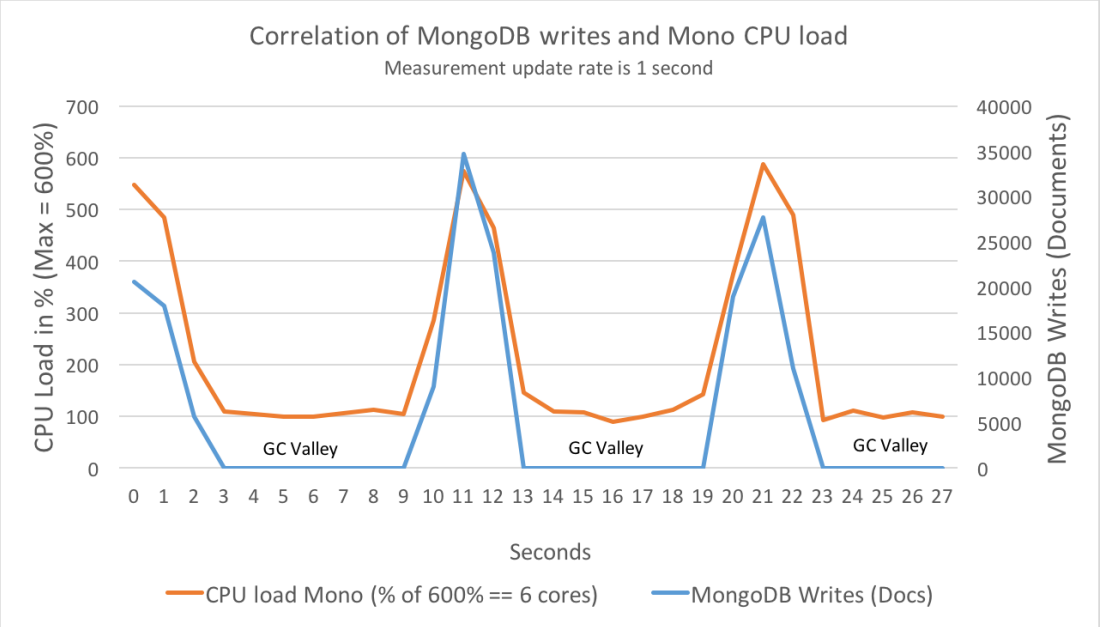


Figure 5.24: Correlation of MongoDB Writes and Mono CPU load.

6 Conclusion & Outlook

6.1 Conclusion

6.1.1 Hypotheses Validation

Hypothesis 1

Experiment 1 clearly shows the performance impact of virtualization with KVM and the overhead introduced by using Docker as container technology. However at 1.02% the relative loss in execution speed is neglectably small compared to the gains in flexibility and increase in productivity which virtualization introduces. So hypothesis 1 is validated.

Hypothesis 2

There are two aspect ins MARS LIFE, which are crucial for vertical scalability: The scalability of distributable code, when run in non-distributed setups and the overall ability to make use of an increased amount of hardware resources. Both aspects are briefly evaluated here.

A MARS LIFE model developer has to decide whether his code should be distributable or not. Being able to write the same code for both scenarios is highly desirable and hence that solution needs to be tested regarding its scalability and performance. This is exactly what benchmark one from experiment two aims at. The results show nicely how the basic agent creation and resolving methods scale linearly with an increased size of agents. The absolute performance is quite excellent as well. So the first aspect of vertical scalability for MARS LIFE is validated.

Regarding resource utilization experiment four reveals that when resources are doubled the execution duration is reduced by 24%. This result is a bit discomforting as it shows problem in hardware utilization and the introduction of possible overheads when the Mono runtime needs to schedule its threads across more cpu cores. While not a truly bad result and still validating hypothesis two, there is a lot of room for improvement in vertical scalability for MARS LIFE.

Hypothesis 3

Hypothesis 3 has six relating experiments and thus is probably the most complex to validate. Horizontal scalability or scaling out inherently implies networking and communication, which is why experiment two is the first to look at. Two benchmarks were performed during that experiment. The first tested the basic and local features of AgentShadowing and showed excellent linear scaling behavior. The second test examined every component involved when actually sending messages over the network. The results reveal a performance problem with Mono and a low-level bug in the used Mono version (4.4.0) but otherwise the measured values as well increase with linear complexity. Additionally a comparison test on Windows with .NET was executed, whose results validate the implementation of AgentShadowing as scaling with linear complexity and performing very well (compare figures 5.6 and 5.8).

With the basic implementation validated the initialization and actual simulation had to be checked in several distributed scenarios. Experiment three validates the scalability of the initialization process by showing excellent linear scaling behavior, since the baseline duration of 24 minutes is divided by the amount of nodes (i.e. 7.6 minutes on three nodes).

Experiment five acts as the baseline simulation benchmark and shows good scale up capabilities by almost completely utilizing the available hardware. The first distributed run from experiment six was prevented by implementation issues inside the ESC, which is why a different ESC configuration (polyglot) was used in experiment seven. These distributed simulation experiments on two and three nodes revealed a linear scaling behavior with a mean speed improvement factor of 1.53 for each added node. Experiment eight adds result output to the setup of experiment seven and while naturally having longer execution durations, the scalability factor resides in the same range with a mean value of 1.43 per added node. So hypothesis 3 finally is validated.

Hypothesis 4

Since experiment six could not be executed the comparison of polyglot versus central ESC in a distributed case must be omitted. Using a polyglot ESC with distribution via replication currently is the only way to execute a distributed simulation experiment and hence the fastest one available. In that sense hypothesis four is valid.

Hypothesis 5

When initializing a distributed simulation there are two possibilities for the creation of Shadow Agents. They may either be created during initialization and stored in-memory to potentially reduce ramp-up times during the simulation or they may be created on-demand when needed. The on-demand solution would be best, when no change in implementation would be required to switch from local to distributed execution and benchmark 1 from experiment two validates the practicability of this feature by presenting very good performance results and linear scalability. A variant from experiment 3, which created all Shadow Agents during initialization, clearly showed that this solution is not feasible. The initialization takes up to four times longer to save a tiny amount of time during the simulation. Also this solution requires a lot more memory (30GB) and since new agents are created during the simulation anyways, the well performing and scaling on-demand creation of Shadow Agents is clearly superior. Thus hypothesis five is validated.

Hypothesis 6

As stated above the fully automated model initialization scales with linear complexity. Experiment 3 measured the initialization of the full KNP model on 1, 2 and 3 nodes and the results show how the duration is divided by the number of nodes (24, 12 and 7.6 minutes for 1, 2 and 3 nodes respectively). However it has to be noted that the data required during initialization is fetched from one or more databases. So there might be an upper limit for scaling out, since each database instance may only serve so many clients. The used databases can also be clustered and distributed and by that annihilate this possibly limiting effect, but it needs to be taken into account when addin new database technologies to the stack. Nevertheless hypothesis 6 is validated.

6.1.2 Result Summary

The scalability abilities of MARS become evident when looking at the hypothesis validation. In almost all tests a linear scaling behavior has been observed and documented. So the initial goal of creating a scalable Simulation as a Service system for large-scale scenarios has been achieved.

Though MARS scales, its absolute performance using Mono as runtime is by far not good enough to support large-scale model execution in a productive way. The KNP model used in this paper would take around a whole month to run to completion on the current setup of the platform, which is simply not feasible. Throughout hypothesis validation some major

performance problems have been found, that need to be fixed.

The top three performance impediments for MARS LIFE are:

#1 The Mono runtime has a significant impact on overall performance especially in distributed runs. Simulations may take up to 100x longer compared to being executed with the .NET Framework on Windows 10 (see section 5.2.2).

#2 The ResultAdapter and the corresponding output strategy for MARS as currently implemented introduce a significant performance overhead. When activated 68,94% of a tick's duration are spend with writing out simulation results on average (see section 5.8).

#3 The currently available implementation of the distributed Environment Service Component consumes a 37% portion of an average tick's duration.

6.2 Outlook

When looking at the results, performance problems and current state of MARS, it needs to be mentioned that all benchmarks and measurements have been conducted during January of 2016 and thus all results reflect the state of development from January. Since then the MARS team is busy working on improvements in the discovered major areas of problems.

Since the number one impediment is Mono a lot of effort has been put into improving it and tweaking its garbage collector in the past months. Some considerable progress has been made like discovering the best settings for the sgen garbage collector for instance. However since Microsoft announced its purchase of Xamarin (the company behind Mono) and thus Mono in addition to the relicensing of their .NET platform to be Open Source, the focus in the future should lie on refactoring MARS LIFE to use the newly available .NET Core and .NET FX cross-platform implementations. These are designed to be used in microservice like, high-performance back-end applications and include a largely enhanced garbage collector. Though these new frameworks share most of the codebase with the current implementation of MARS LIFE, still the refactoring of MARS LIFE will require a reimplementation of the central distribution solution, because most of the namespaces used by the .NET framework in AgentShadowing are discontinued or replaced in .NET Core / FX. So replacing Mono will require a considerable amount of work in the future, but must not be omitted if MARS LIFE is to be used in large-scale real world scenarios.

The result output strategy as implemented in January was very simple and is currently being heavily overhauled by Jan Dalski. Among the new solutions that he is adding right now are compression methods, delta updates combined with a key-frame mechanism and an improved method for selecting what is being written out as a result from the simulation. Also the solution will greatly benefit from the improved garbage collector in .NET Core, given that the majority of problems in simulation output resulted from the garbage collector halting the system.

Lastly the environment service component will undergo a complete redesign. Since the very beginning of the MARS project in 2013 the requirements for the ESC have changed. Though the current solution is working, it will face upper boundaries (mostly memory-wise) on large-scale models and in heavily distributed environments, where replication is no longer a suitable way of distribution. Taking the polyglot ESC approach into account, the new solution should not try to fit all use cases, but be either more adaptive or consist of many different implementations tailored towards specific scenarios. This would allow to select the best fit for every part of a model during development and should increase both, simulation performance and development speed alike.

Bibliography

Amouroux, Edouard, Thanh-quang, C H U, Boucher, Alain, & Drogoul, Alexis. 2007. GAMA : an environment for implementing and running spatially explicit multi-agent simulations.

Axelrod, Robert. 1997. *The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration.* Vol. 1. Princeton, NJ: Princeton University Press.

Balmer, Michael, Axhausen, Kay, & Nagel, Kai. 2006. Agent-based demand-modeling framework for large-scale microsimulations. *Transportation Research Record: Journal of the Transportation Research Board*, 125–134.

Bellifemine, Fabio, Caire, Giovanni, Poggi, Agostino, & Rimassa, Giovanni. 2008. JADE: A software framework for developing multi-agent applications. Lessons learned. *Information and Software Technology*, **50**(1-2), 10–21.

Caglar, Faruk, Shekhar, Shashank, Gokhale, Aniruddha, Basu, Satabdi, Rafi, Tazrian, Kinnebrew, John, & Biswas, Gautam. 2015. Simulation Modelling Practice and Theory Cloud-hosted simulation-as-a-service for high school STEM education. *Simulation Modelling Practice and Theory*, **58**, 255–273.

Cayirci, Erdal. 2013. Modeling and simulation as a cloud service: A survey. *Proceedings of the 2013 Winter Simulation Conference - Simulation: Making Decisions in a Complex World, WSC 2013*, 389–400.

Childress, W. Michael, Coldren, Cade L., & McLendon, Terry. 2002. Applying a complex, general ecosystem model (EDYS) in large-scale land management. *Ecological Modelling*, **153**(1-2), 97–108.

Cicirelli, Franco, Furfaro, Angelo, Giordano, Andrea, & Nigro, Libero. 2010. Parallel Simulation of Multi-agent Systems Using Terracotta. *2010 IEEE/ACM 14th International Symposium on Distributed Simulation and Real Time Applications*, oct, 219–222.

Collier, N., & North, M. 2012. Parallel agent-based simulation with Repast for High Performance Computing. *Simulation*, **89**(10), 1215–1235.

Farmer, J Doyne, & Foley, Duncan. 2009. The economy needs agent-based modelling. *Nature*, **460**(7256), 685–686.

Filatova, Tatiana, Verburg, Peter H., Parker, Dawn Cassandra, & Stannard, Carol Ann. 2013. Spatial agent-based models for socio-ecological systems: Challenges and prospects. *Environmental Modelling & Software*, apr, 1–7.

Fowler, Martin. 2014. *Microservices, a definition of this new architectural term.* URL: http://martinfowler.com/articles/microservices.html. Last accessed at: 30.04.2016.

Gardner, Martin. 1970. Mathematical Games – The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 120–123.

Gelernter, David, & Carriero, Nicholas. 1992. Coordination languages and their significance. *Communications of the ACM*, **35**(2), 96.

Gilbert, Nigel, & Bankes, Steven. 2002. Platforms and methods for agent-based modeling. *Proceedings of the National Academy of Sciences*, **99**(Supplement 3), 7197–7198.

Grimm, Volker. 1999. Ten years of individual-based modelling in ecology: what have we learned and what could we learn in the future? *Ecological Modelling*, **115**(2-3), 129–148.

Grimm, Volker, & Railsback, Steven. 2005. *Individual-based Modeling and Ecology (Princeton Series in Theoretical and Computational Biology).*

Haase, A T. 1999. Population biology of HIV-1 infection: viral and CD4+ T cell demographics and dynamics in lymphatic tissues. *Annual review of immunology*, **17**, 625–56.

Hilbers, Jelle P, van Langevelde, Frank, Prins, Herbert H T, Grant, C C, Peel, Mike J S, Coughenour, Michael B, de Knegt, Henrik J, Slotow, Rob, Smit, Izak P J, Kiker, Greg A, & de Boer, Willem F. 2015. Modeling elephant-mediated cascading effects of water point closure. *Ecological Applications*, **25**(2), 402–415.

Holst, Niels. 2013. A universal simulator for ecological models. *Ecological Informatics*, **13**(jan), 70–76.

Hüning, Christian, Wilmans, Jason, Feyerabend, Nils, & Thiel-Clemen, Thomas. 2014. MARS - A next-gen multi-agent simulation framework. *Simulation in Umwelt- und Geowissenschaften, Workshop Osnabrück 2014*, 1–14.

Hüning, Christian, Adebahr, Mitja, Thiel-Clemen, Thomas, Dalski, Jan, Lenfers, Ulfia, Grundmann, Lukas, Dybulla, Janus, & Kiker, Gregory A. 2016. Modeling & Simulation as a Service with the Massive Multi-Agent System MARS. *In: Proceedings of the 2016 Spring Simulation Multiconference*.

Huqqani, Altaf Ahmad, Li, Xin, Beran, Peter Paul, & Schikuta, Erich. 2010. N2Cloud: Cloud based neural network simulation application. *The 2010 International Joint Conference on Neural Networks (IJCNN)*, 1–5.

Huston, M, DeAngelis, D, & Post, W. 1988. New computer models unify ecological theory. *BioScience*, **38**, 682–691.

Jennings, Nicholas R. 1999. Agent-Based Computing : Promise and Perils. *Pages 1429–1436 of: 16th Int. Joint Conf. on Artificial Intelligence (IJCAI-99)*.

Johnson, Harry E, & Tolk, Andreas. 2013. Evaluating the Applicability of Cloud Computing Enterprises in Support of the Next Generation of Modeling and Simulation Architectures. *Proceedings of the Military Modeling & Simulation Symposium*.

Le, Quang Bao, Park, Soo Jin, & Vlek, Paul L G. 2010. Land Use Dynamic Simulator (LUDAS): A multi-agent system model for simulating spatio-temporal dynamics of coupled human-landscape system. 2. Scenario-based application for impact assessment of land-use policies. *Ecological Informatics*, **5**(3), 203–221.

Liu, Jianguo, Dietz, Thomas, Carpenter, Stephen R, Alberti, Marina, Folke, Carl, Moran, Emilio, Pell, Alice N, Deadman, Peter, Kratz, Timothy, Lubchenco, Jane, Ostrom, Elinor, Ouyang, Zhiyun, Provencher, William, Redman, Charles L, Schneider, Stephen H, & Taylor, William W. 2007. Complexity of coupled human and natural systems. *Science (New York, N.Y.)*, **317**(5844), 1513–6.

Mengistu, Dawit, Tröger, Peter, Lundberg, Lars, & Davidsson, Paul. 2008. Scalability in distributed multi-agent based simulations: The JADE case. *Proceedings of the 2008 2nd International Conference on Future Generation Communication and Networking, FGCN 2008*, **5**, 93–99.

Muller, S, Muñoz-Carpena, R, & Kiker, G. 2011. Model Relevance. *Pages 39–65 of:* Linkov, Igor, & Bridges, S Todd (eds), *Climate: Global Change and Local Adaptation*. Dordrecht: Springer Netherlands.

Münchow, Stefan, Enukidze, I, Sarstedt, Stefan, & Thiel-Clemen, Thomas. 2014. WALK: A Modular Testbed for Crowd Evacuation Simulation. *In:* Weidman, U (ed), *Proceedings of the 6th International Conference on Pedestrian Evacuation Dynamics*. Springer.

Narayanan, S. 2000. Web-based Modeling and Simulation. *Proceedings o f the 2000 Winter Simulation Conference*, 60–62.

Neumann, John Von. 1966. *Theory of Self-Reproducing Automata*. Champaign, IL, USA: University of Illinois Press.

Niazi, Muaz, & Hussain, Amir. 2011. Agent-based computing from multi-agent systems to agent-based models: A visual survey. *Scientometrics*, **89**(2), 479–499.

Noetzel, Carsten, Reintjes, Ralf, & Thiel-Clemen, Thomas. 2013 (sep). Die Rolle öffentlicher Verkehrsmittel bei der Übertragung und Verbreitung von Krankheitserregern. *In:* Handels, H, & Ingenerf, J (eds), *58. Jahrestagung der Deutschen Gesellschaft für Medizinische Informatik, Biometrie und Epidemiologie e.V. (GMDS)*.

Padilla, Jose. 2014. Cloud-Based Simulators: Making Simulations Accessible To Non-Experts and Experts Alike. *Proceedings - Winter Simulation Conference 2014*, 3630–3639.

Parker, Jon. 2007. A flexible, large-scale, distributed agent based epidemic model. *Proceedings - Winter Simulation Conference*, 1543–1547.

Parker, Jon, & Epstein, Joshua M. 2011. A Distributed Platform for Global-Scale Agent-Based Models of Disease Transmission. *ACM Transactions on Modeling and Computer Simulation*, **22**(1), 1–25.

Perry, George L.W., & Enright, Neal J. 2006. Spatial modelling of vegetation change in dynamic landscapes: a review of methods and applications. *Progress in Physical Geography*, **30**(1), 47–72.

Rak, Massimiliano, Cuomo, Antonio, & Villano, Umberto. 2012. mJADES: Concurrent Simulation in the Cloud. *2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems*, 853–860.

Ralha, Célia G., Abreu, Carolina G., Coelho, Cássio G.C., Zaghetto, Alexandre, Macchiavello, Bruno, & Machado, Ricardo B. 2013. A multi-agent model system for land-use change simulation. *Environmental Modelling & Software*, **42**(apr), 30–46.

Russell, SJ, & Norvig, P. 1995. A modern, agent-oriented approach to introductory artificial intelligence. *SIGART Bulletin*, **6**(2), 24–26.

Schelling, Thomas C. 1969. Models of Segregation. *Pages 488–493 of: Papers and Proceedings of the Eightyfirst Annual Meeting of the American Economic Association*, vol. 59.

Smajgl, Alex, Brown, Daniel G, Valbuena, Diego, & Huigen, Marco G A. 2011. Empirical characterisation of agent behaviours in socio-ecological systems. *Environmental Modelling Software*, **26**, 837–844.

Suryanarayanan, Vinoth, Theodoropoulos, Georgios, & Lees, Michael. 2013. PDES-MAS: Distributed Simulation of Multi-agent Systems. *Procedia Computer Science*, **18**(jan), 671–681.

Taylor, S J E, Balci, O, Cai, W, Loper, Margaret L., Nicol, David M., & Riley, George. 2013. Grand challenges in modeling and simulation: expanding our horizons. *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation.*, 409–414.

Taylor, Simon J E, Fujimoto, Richard, Page, Ernest H., Fishwick, Paul a., Uhrmacher, Adelinde M., & Wainer, Gabriel. 2012. Panel on grand challenges for modeling and simulation. *Proceedings - Winter Simulation Conference*.

Taylor, Simon J. E., Kiss, Tamas, Terstyanszky, Gabor, Kacsuk, Peter, & Fantini, Nicola. 2014. Cloud computing for simulation in manufacturing and engineering: introducing the CloudSME simulation platform. *Proceedings of the 2014 Annual Simulation Symposium*, 12.

Thiel, Christian. 2013. *Analyse von Partitionierungen und partieller Synchronisation in stark verteilten multiagentenbasierten Fußgängersimulationen.* Master Thesis, Hamburg University of Applied Sciences.

Thiel-Clemen, Th. 2013a. Information Integration in Ecological Informatics and Modelling. *Pages 89 – 96 of:* Wittmann, J., & Müller, M. (eds), *Simulation in Umwelt- und Geowissenschaften, Workshop Leipzig*.

Thiel-Clemen, Thomas. 2013b. Designing Good Individual-based Models in Ecology. *Pages 97–106 of:* Wittmann, J, & Müller, M (eds), *Simulation in Umwelt- und Geowissenschaften, Workshop Leipzig*.

Tolk, Andreas, & Mittal, Saurabh. 2014. A necessary paradigm change to enable composable cloud-based M&S services. *Proceedings of the 2014 Winter Simulation Conference*, 356–366.

Vigueras, Guillermo, Orduña, Juan M., Lozano, Miguel, & Jégou, Yvon. 2013. A scalable multi-agent system architecture for interactive applications. *Science of Computer Programming*, **78**(6), 715–724.

Villa, Ferdinando. 2001. Integrating modelling architecture: a declarative framework for multi-paradigm, multi-scale ecological modelling. *Ecological Modelling*, **137**(1), 23–42.

Wang, B, Yao, Y, & Himmelspach, Jan. 2009. Experimental analysis of logical process simulation algorithms in JAMES II. *Pages 1167–1179 of: Proceedings of the Winter 2009 Simulation Conference*.

Wang, Y, Lees, M, & Cai, W. 2012. Grid-based partitioning for large-scale distributed agent-based crowd simulation. *Proceedings of the Winter 2012 Simulation Conference*.

Wendeldorf, Katherine V., Bassaganya-Riera, Josep, Bisset, Keith, Eubank, Stephen, Hontecillas, Raquel, & Marathe, Madhav. 2011. ENteric Immunity SImulator: A Tool for in silico Study of Gut Immunopathologies. *2011 IEEE International Conference on Bioinformatics and Biomedicine*, **11**(3), 462–469.

Wooldridge, Michael. 1997. Agent-based software engineering. *IEE Proceedings Software Engineering.*, **144**(1), 26–37.

Wooldridge, Michael. 1998. Agent-based computing. *Interoperable Communication Networks*, **5**(5), 71–98.

Yamamoto, G, Tai, H, & Mizuta, H. 2008. A platform for massive agent-based simulation and its evaluation. *Massively Multi-Agent Technology*.

Zehe, Daniel, Knoll, Alois, Cai, Wentong, & Aydt, Heiko. 2015. {SEMSim} Cloud Service: Large-scale urban systems simulation in the cloud. *Simulation Modelling Practice and Theory*, **58**.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 27. Juni 2016 Christian Hüning, christian.huening@haw-hamburg.de