Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

# Extension of an Evaluation Testbed for Fog Computing Infrastructures and Applications

Robin Finkbeiner

| | |
|---|---|
| **Course of Study:** | Softwaretechnik |
| **Examiner:** | Prof. Dr. Dr. Kurt Rothermel |
| **Supervisor:** | Dipl.-Inf. Ruben Mayer |
| **Commenced:** | 2017-11-13 |
| **Completed:** | 2018-05-13 |

# Abstract

Fog computing is an emerging system architecture in the cloud and Internet of Everything realm. It aims to distribute computing, storage, and control closer to the user. In the last few years work in this field has gained traction. Publications and research projects are increasing and just recently work on an open standard has been started. The research community and industry are proposing new approaches, algorithms, and system–architectures on a very fast pace. However, it remains difficult to evaluate and test these proposals. Particularly, because real–world testbeds are expensive and hard to set up.

This work is continuing the development of the open source project EmuFog. EmuFog is an extensible and scalable emulation framework for fog computing infrastructures. It supports researchers, system–architects and developers by providing a framework to test application behavior in fog architectures. Furthermore, evaluation of algorithms for edge identification, fog node, and application placement can be carried out on large systems.

This work extends the previous system architecture to enable multi–tiered fog nodes. That is, the ability to run multiple applications on a single fog node instance. Furthermore, usage of custom placement algorithms is simplified. Additionally, groundwork for resource management was introduced to the system architecture.

# Kurzfassung

Fog Computing ist eine aufkommende System–Architektur im Cloud und Internet of Everything Umfeld. Fog Computing zielt darauf ab, Rechenleistung, Datenspeicher und Kontrollinstanzen näher zu den Nutzern zu bringen. In den letzten Jahren hat Forschung in diesem Kontext stark zugenommen. Insbesondere die Anzahl an Forschungsprojekten und Veröffentlichungen. Zusätzlich wird an einem gemeinsamen Standard gearbeitet. Die Forschergemeinde und die Industrie schlagen neue Lösungsansätze, Fog Algorithmen und System–Architekuren mit hoher Frequenz vor. Allerdings ist es schwierig diese Vorschläge zu evaluieren. Insbesondere sind Testumgebungen auf echter Hardware für Fog Computing teuer, umständlich und selten.

Diese Arbeit setzt die Entwicklung des open-source Projekts EmuFog fort. EmuFog ist ein erweiterbares, flexibles und skalierbares Emulations–Framework um Fog Computing Infrastrukturen zu testen. Es unterstützt Forscher, System Architekten und Entwickler, indem es Nutzern ermöglicht, Software Anwendungen in Fog Architekturen zu testen. Zusätzlich ist es möglich Algorithmen für Netzwerkkantenerkennung, Fog Knoten Platzierung und Anwendungsplatzierungen auf großen Netzwerken zu evaluieren.

In dieser Arbeit wurde die vorherige System–Architektur erweitert, um multi–tiered Fog Knoten zu ermöglichen. Multi–tiered Fog Knoten ermöglichen es, mehr als eine Applikation pro Instanz auszuführen. Außerdem wurde die Ausführung von Placement Algorithmen vereinfacht und Grundlagen für Ressourcen Verwaltung wurde zu der Sytem–Architektur hinzugefügt.

# Contents

# List of Figures

# List of Listings

# List of Algorithms

# List of Abbreviations

**BRITE** Boston Representative Internet Topology Generator. 26

**DDS** Distributed Data Stores. 25

**IoE** Internet of Everything. 15

**JSON** JavaScript Object Notation. 44

**MQTT** Message Queue Telemetry Transport. 21

**QoS** Quality of Service. 22

**YAML** Yet Another Markup Language. 26

**YCSB** Yahoo Cloud Serving Benchmark. 25

# 1 Introduction

Today the cloud is everywhere, many companies and users move their data and applications to the cloud, utilizing the benefits of a centralized architecture. This system architecture marginalizes the operational cost per user and allows individuals and companies access to computational power previously exclusive to very few. Many large service providers, competing in a fast–growing and competitive market, offer fast, scalable, secure and easy to manage cloud solutions for almost every use case. The cloud has proven to be an economical solution for many scenarios. [Sat17].

At the moment many new digital innovations are emerging. Smart Cities, Smart Cars, and many more Internet of Everything (IoE) [DLYE] technologies are under development. In the years to come, we will see how augmented and virtual reality, artificial intelligence, and autonomous driving change the way we live. All the aforementioned technologies will produce a great amount of data and network traffic while simultaneously demanding fast response times. Cisco is predicting that "a city of one million will generate 200 million gigabytes of data per day by 2020." [Ind16]

This explosion of traffic leads to several problems. First, centralized cloud solutions will suffer from congested network links. Second, many of those emerging technologies like autonomous driving, smart factories, and intelligent buildings impose high requirements on network latency. Such applications are dependent on secure, fault-tolerant connections with low round–trip latencies. In their work, Ang Li et al. collected average round–trip times from 260 different locations to multiple public cloud providers. They measured average round–trip times of 73 ms [LYKZ10]. This is too high for time–sensitive applications.

*Fog computing* [BMZA12] is an extension to the existing traditional cloud infrastructure. The goal is to enable time–sensitive, data–heavy and mission–critical applications. This is achieved by reducing the distance from data sources e.g. devices to the cloud. Fog architectures move computing power, storage, communication, and control closer to the network edge. By distributing the capabilities previously provided by the centralized cloud into multiple layers, network load and round–trip times can be reduced.

In a combined effort Cisco and multiple other companies and research institutes introduced a fog reference architecture to guide future research and to clarify terminology. The collective effort is organized through the *OpenFog Consortium*[1]. Their work is centered around creating a common understanding and vision for fog computing.

[1] OpenFog Consortium https://www.openfogconsortium.org/

Nonetheless, it remains difficult to test application behavior and fog algorithms. Real–world testbeds are expensive, difficult to setup and only available to few. Simulation frameworks like *iFogSim* [GVGB17] are limited in terms of executable software. And are often tailored to specific use cases.

This work is continuing the development of the open source project *EmuFog* [Gra17] [MGG+17]. EmuFog is an extensible and scalable emulation framework for Fog computing infrastructures. It supports researchers, system–architects and developers by providing a framework to test application behavior in fog architectures. Furthermore, evaluation of algorithms for edge identification, fog node placement, and application assignment can be carried out on large systems. One major drawback of the current EmuFog version is the inability to run more than one application per fog node. Additionally, more control over the executed placement algorithms is desirable.

Presented in this work are extensions to the previous system, to enable multi–tiered fog nodes. That is, the ability to run multiple applications on a single fog node instance. Furthermore, usage of custom placement algorithms is simplified and a novel step for application assignment is presented. Finally, the groundwork for resource management is introduced to the system architecture.

## Outline

This thesis is sectioned into the following chapters:

**Chapter 2 – Background** This Chapter, introduces important foundations and theoretical concepts relevant to the presented work. Furthermore, related work is introduced and discussed.

**Chapter 3 – Objectives** This Chapter, defines and motivates the objectives of proposed work.

**Chapter 4 – Concept** Here, concepts and abstractions behind EmuFog are presented and explained. In particular, concepts behind the topology representation and the placement algorithms will be elaborated.

**Chapter 5 – Implementation** This Chapter, explains the system architecture and the implementation of contributed concepts.

**Chapter 6 – Evaluation** In this Chapter, the usability and functionality of EmuFog will be evaluated by testing a real–world application in different fog architectures.

**Chapter 7 – Conclusion and future work** Finally, in this Chapter the presented work is concluded and proposals for further work on EmuFog are given.

# 2 Background

This chapter introduces foundational concepts that are relevant to this work, beginning by providing a formal definition of fog computing in Section 2.1. Highlighting its concepts, benefits and use cases. Section 2.2 discusses the concepts and benefits of network emulation and compares emulation to simulation and real–world testing. Then tools and technologies used in this work are introduced. Finally, related work is discussed in Section 2.4.

## 2.1 Fog Computing

Bonomi et al. proposed fog computing as one possible approach to enable time–sensitive cloud applications [BMZA12]. It has the potential to solve many prospective limitations of cloud computing in terms of latency and generated network traffic.

The OpenFog consortium defines fog computing as follows: "A horizontal, system level architecture that distributes computing, storage, control and networking functions closer to the user along a cloud–to–thing–continuum." [Con+] The goal of fog computing is to enable time sensitive, data heavy and mission–critical applications. It brings control and computing closer to the user.

**Differences between Fog and Edge Computing** As fog and edge computing are fairly new concepts the terminology is not yet unambiguous. Vaquero et al. see edge computing as a subset of fog computing [VR14]. The OpenFog Consortium on the other hand clearly differentiates between fog and edge computing. They view edge computing strictly limited to computation on edge devices e.g smart phones. Fog computing on the other hand also provides storage, control and networking capabilities and hierarchically distributes intelligence across several nodes. It is important to point out that fog computing is not disconnected from the cloud. To the contrary, it is a very heterogeneous environment in which nodes communicate and potentially cooperate with each other.

**Resource Characteristics** There are several different characteristics that distinguish cloud from fog computing. Varshney et. al. highlight differences between fog, cloud and edge computing. They highlight network connectivity as one of the key characteristics [VS17]. Fog nodes lie between the cloud and the network edge, therefore latency is low in both directions, to the cloud, and to the edge. The computing and storage capabilities of fog nodes are significantly lower than in the cloud. Fog computing has the potential

to enable low latencies through distribution and cooperation. Cooperation is one key to reduce generated network traffic. Data traveling upwards from device nodes to the cloud can be filtered, augmented and analyzed by intermediate fog nodes.

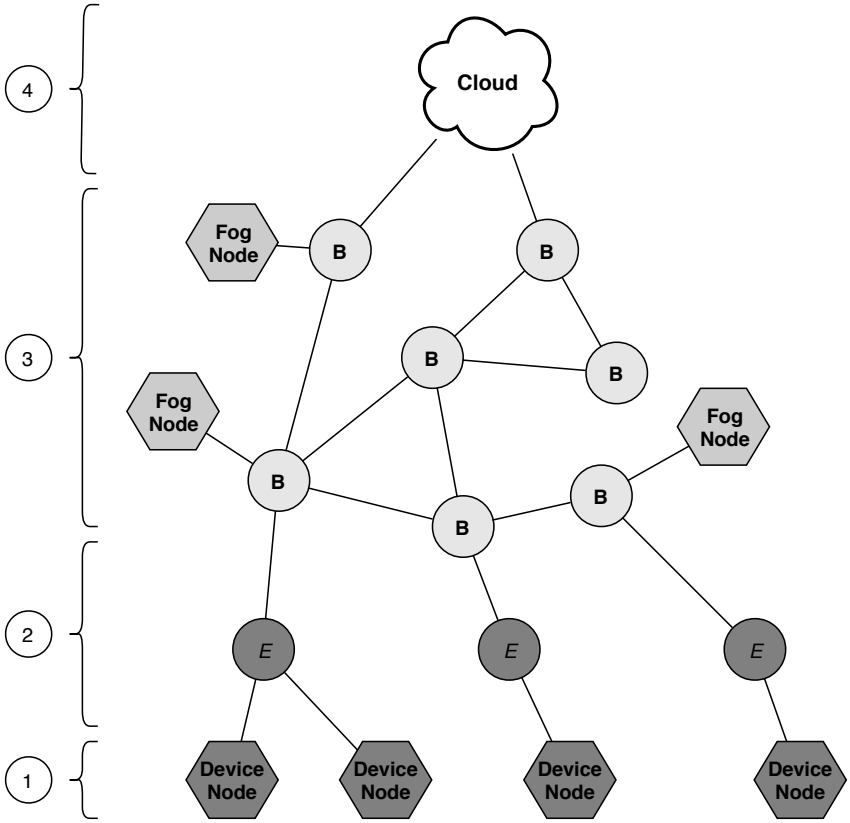



**Figure 2.1:** Fog computing system architecture.

**Fog Computing Architecture** Kapsalis et al. propose a simple fog platform architecture in their work [KKV+17]. The proposed architecture consists of four layers: *device layer* (1), *hub layer* (2), *fog layer* (3) and *cloud layer* (4). The lowest layer contains many different heterogeneous devices or sensors. The second layer is the hub layer through which devices and sensors are connected to network entities in order to communicate with the cloud, fog nodes or other devices. The third layer consists of the network backbone that lies between the edge and the cloud. This is the fog layer, here fog nodes are placed. The final layer is the cloud layer, which represents the endpoint or destination of the system architecture.

**Fog Deployment Models** One of the major benefits of fog computing is the flexibility of fog architectures. Fog computing supports a fluid transition between fog nodes and the centralized cloud infrastructure. The actual number of required nodes and hierarchy levels is dependent on the use cases of the application. For example, to satisfy latency demands of applications with small event–to–action windows, nodes could be distributed at the network edge close to the devices. In data heavy IoE sensor streaming applications, a hierarchical node layout could be beneficial. In such a topology, specific nodes could process only a subset of the produced data flow. To determine a suitable layout is one of the major difficulties for fog application developers.

**Scenarios for Fog Computing** Fog computing may be useful to various different scenarios like data heavy stream–processing or augmented and virtual reality where low latencies are essential. Especially the ability to filter and aggregate information traveling from the network edge to the cloud makes fog computing promising for stream– and complex–event–processing. Stream processing capabilities of fog computing in the context of Social Sensing [MGSR17b] and Message Queue Telemetry Transport (MQTT) [XMR16] have been further evaluated.

The following example use case, which is adapted out of the reference architecture of the OpenFog Consortium [Con+], further highlights the benefits of fog computing. One possible application context of fog computing is the supervision of safety–critical applications, for example, monitoring of oil pipelines. The operating company has equipped their pipelines with controllable valves as well as pressure and flow sensors. To be able to remotely control the pipeline. Without fog computing, each sensor would have to be connected directly to the centralized management software. This high proximity leads to long round trip times. In case of a malfunction, reaction time would be slow. For such a critical application this is not feasible.

With fog computing, one could imagine distributing multiple nodes along the pipeline. Such nodes could collect and process sensor data for specific subsections of the pipeline. If granted autonomy fog nodes would even be able to react to anomalies directly. Furthermore, only relevant aggregated sensor information will be transmitted from fog nodes to the cloud. This architecture will reduce the required bandwidth and decrease the reaction time drastically. Also, due to the distributed decision–making process, the pipeline will still be secure even when sections of the pipeline are disconnected from the centralized monitoring application. Responsible fog nodes can act autonomously.

This use case highlights two important properties of fog computing: reduced round–trip time due to low proximities and increased security due to the distribution of control.

## 2.2 Network Emulation

The design and development of fog applications, architectures or algorithms requires adequate evaluation and validation in order to test the feasibility and efficiency of proposed solutions. This is only possible if there is efficient instrumental tooling available. Especially for testing and evaluation tools that offer a repeatable and controllable environment are desirable.

Nowadays, three approaches are commonly used: simulation, emulation and real–world test beds. Simulation "is a classical way to achieve economical and fast protocol experimentation." [LPD12]. Often the studied problem is simplified and the simulation serves as a poof of concept. Simulation frameworks like *OMNet++*[1] are event-driven and are based on a virtual clock. With simulation, a real–time evaluation is not possible.

Real–world testing is expensive and difficult to install, as real hardware has to be used. In the fog computing context this is especially problematic as required hardware often has yet to be built. Furthermore, real–world tests are very inflexible because the required infrastructure has to be built first. Real–world testing is the final step in the development process of applications or algorithms and is therefore often only feasible at the very end.

Emulation lies between simulation and execution on actual hardware. With emulation, one has the possibility to define a specific environment suitable to the objective. For example, the underlying characteristics and behavior in terms of Quality of Service (QoS) can be defined. Emulation is used "to assess the performance of an end-to-end system." [LPD12]

This fine–grained control over the test environment combined with the cheap execution cost, compared to real–world tests, makes emulation attractive for the evaluation of fog scenarios.

In the following, emulation frameworks used in this work are introduced.

### MiniNet

*MiniNet* [LHM10][2] proposed by Lantz et al. is a widely used network emulator. With Mininet users are able to run a complete network topology on their local computer. All required elements, hosts, switches, routers, and links are emulated and behave like a complete network. Users are able to execute actual software on Mininet's virtual hosts. One further key characteristic that makes Mininet interesting for fog architecture emulation, is the ability to specify the capabilities of the network e.g. link speed, delay and further QoS attributes. Mininet allows for fast, easy and cheap fog topology evaluation. In contrast to simulation, users can run real programs which is essential for application testing.

[1] OMNet++ is available at https://www.omnetpp.org/
[2] Mininet is available http://mininet.org/

### Containernet

Peuster et al.[PKR16] proposed *Containernet*[3] as an extension to Mininet, that enables users to run *Docker* Containers on Mininet's virtual hosts. This additional feature simplifies the configuration and deployment of custom applications under test. The ability to run Containers reduces the overhead to test and evaluate software as Containerization is nowadays widely used. Furthermore, users can run the same Containers in production and Containernet.

The presented emulation frameworks fit well in the context of this work. They enable precise network modeling and measurements according to scenario specific needs. Also, they provide ways to run fog applications as close to the actual deployment as possible. By using Mininet and Containernet fast and cheap, fog topology testing is possible.

[3] Containernet is available at https://containernet.github.io/

## 2.3 EmuFog a Testbed for Fog Computing

EmuFog [Gra17] [MGG+17] is an extensible and scalable emulation framework for fog computing infrastructures. It supports researchers, system–architects and developers by providing a framework to test application behavior in fog architectures. Furthermore, evaluation of algorithms for edge identification, fog node placement, and application assignment can be carried out on large systems. Graser et al. published a first version of the framework open source on GitHub under the MIT Licence[4]. The tool is written in the Java programming language and aims to be platform agnostic. By using network emulation, EmuFog fills the gap between live testing and simulation for fog computing. The workflow when performing experiments consist of four main steps [MGG+17]:

- **Topology Generation**: A suitable network topology is generated and provided to the application.
- **Topology Enhancement**: Provided topology is parsed to the internal network graph representation.
- **Topology Transformation**: The network is augmented using a several sub–steps. First, the network edge is determined. Then devices and fog nodes are placed in the network following provided placement policies. Configuration and specification of fog and device nodes as well as applications and other scenario specific definitions is done via a configuration file.
- **Deployment and Execution**: Finally, an executable experiment script is generated out of the enhanced network topology.

The framework is useful for multiple developer and research tasks. Researchers can use EmuFog to evaluate and test developed fog layouts or placement strategies in a fast, controlled and repeatable way. System–architects can examine the characteristics of existing topologies in a fog computing context. Developers can evaluate the behavior of their application in a fog architecture while using their actual applications bundled into containers.

Currently, there is no mechanism to control the mapping of applications to fog or device nodes. Furthermore, only one application can be deployed per node. Moreover, the executed topology transformation steps are fixed an can only be adapted by recompiling the tool. These limitations among others are to be addressed in this work.

[4] MIT License is available at https://opensource.org/licenses/MIT

## 2.4 Related Work

### FogStore

Mayer et al. [MGSR17a] propose the *FogStore* concept as a improvement to existing Distributed Data Stores (DDS). They aim to improve the performance of DDS by utilizing the benefits of fog computing, especially close proximity to data consumers and producers. An examination of DDS that are placed in close proximity to the network edge is carried out. Mayer et al. developed a concept to integrate fog computing into existing DDS. Therefore, they introduce fog–aware placement strategies to place fog nodes and context–sensitive consistency levels. Furthermore, they performed evaluation measurements using network emulation and the Yahoo Cloud Serving Benchmark (YCSB).

### iFogSim

Gupta et al. [GVGB17] propose *iFogSim*, as a toolkit to evaluate fog environments. Conceptually the goals of iFogSim and EmuFug are similar. In contrast to EmuFog, iFogSim uses simulation to run the experiments. In iFogSim users are able to evaluate different fog placements and measure power consumption, latency, congestion and placement costs.

The previously motivated differences of simulation and emulation in Section 2.2 apply to iFogSim. In particular, the inability to execute actual software is a major drawback compared to EmuFog. Furthermore, the system architecture of iFogSim doesn't enable device–to–device communication, thus limiting the number of scenarios that can be tested. Moreover, the topologies under test have to be manually created. Nonetheless, simulator–based solutions like iFogSim are valuable tools for rapid prototyping and initial validation of fog algorithms.

## 2.5 State of the Art Technologies

### 2.5.1 Topology Generation

In research, several different topology generators are used for protocol testing, performance measurements and evaluation of application behavior. Proposed work uses the Boston Representative Internet Topology Generator (BRITE) [MLMB01][5] to generate topologies used as input to evaluate fog applications or algorithms.

### 2.5.2 Container–based Virtualization

*Container–based virtualization* is a system concept utilizing kernel features like *cgroups*[6] in order to isolate processes from one another. It is considered to be a lightweight alternative to *hypervisor–based virtualization* [SPF+07]. Here, each virtual machine can contain a complete operating–system and emulated hardware. Contrary, in container–based virtualization the host system is shared across multiple containers. One of the biggest advantages of container–based virtualization is the fast startup–time of containers compared to heavy virtual machines. *Containers* can be seen as an abstraction layer that encapsulates applications. These encapsulated applications bundled with all its dependencies can be executed as isolated processes.

#### Docker and Docker Compose

*Docker*[7] is one of the most used container–based virtualization solutions. The application containers in Docker are defined via *Dockerfiles*. Docker containers are lightweight and platform agnostic.

*Docker Compose*[8] is a container orchestration tool that provides features to configure multi–container applications via a Yet Another Markup Language (YAML)[9] file. Execution of the complete application stack can be controlled over a command line interface.

Container–based virtualization is beneficial to this work as it allows to define applications in such a way that they can be executed in an emulated system environment. The platform–agnostic character of containers is also very important. It can be expected that applications executed in the test environment behave similarly in the actual production environment. Which is essential to be able to realistically evaluate application characteristics in fog systems.

[5] Brite is available at https://www.cs.bu.edu/brite/
[6] Documentation is available at https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt
[7] Docker is available at https://www.docker.com
[8] Docker Compose is available at https://docs.docker.com/compose/
[9] YAML specification is available at http://yaml.org/spec/1.2/spec.html

# 3 Objectives

As previously motivated in Section 2.3 by using emulation, EmuFog presents the unique property to execute scenarios using actual software. This work introduces an extended and improved version of the EmuFog fog computing testbed. The main objectives of EmuFog remain unchanged. That is, the possibility to emulate large networks to evaluate and validate placement algorithms and fog applications. All previously defined criteria *scalability*, *extensibility*, *flexibility*, *platform independence* remain valuable [Gra17].

The emphasis of this work is to further improve *extensibility*, *flexibility* and *platform independence*. Since every scenario has different requirements, the testbed should provide ways to specify the system models in order to meet defined criteria.

In particular, it should be possible to place more than one application per device or fog node to be able to emulate more complex fog systems, e.g., a multi–tenant or multi–application system. Additionally, it should be possible to adapt existing or implement custom placement algorithms. The set of supported placement algorithms should include the following:

- *Edge Identification*: It should be possible to implement own logic to locate the network edge and backbone for a given topology.
- *Device Placement*: It should be possible to define where, which type and how many devices should be placed in the topology.
- *Fog Layout*: It should be possible to define where, which type and how many fog nodes should be placed in the topology.
- *Application Assignment*: It should be possible to define where, which and how many application instances should be placed in the topology.

Furthermore, each scenario has different demands regarding resource availability and resource limitations. It should be possible to define hardware limitations for fog and device nodes as well as limitations for applications to be placed on them.

Naturally, the system architecture should be done in a way that it remains flexible for future extensions. Moreover, the existing code should be refactored so that previously listed criteria are fulfilled. Finally, the set of executed algorithms should be configurable before the application is executed.

# 4 Concept

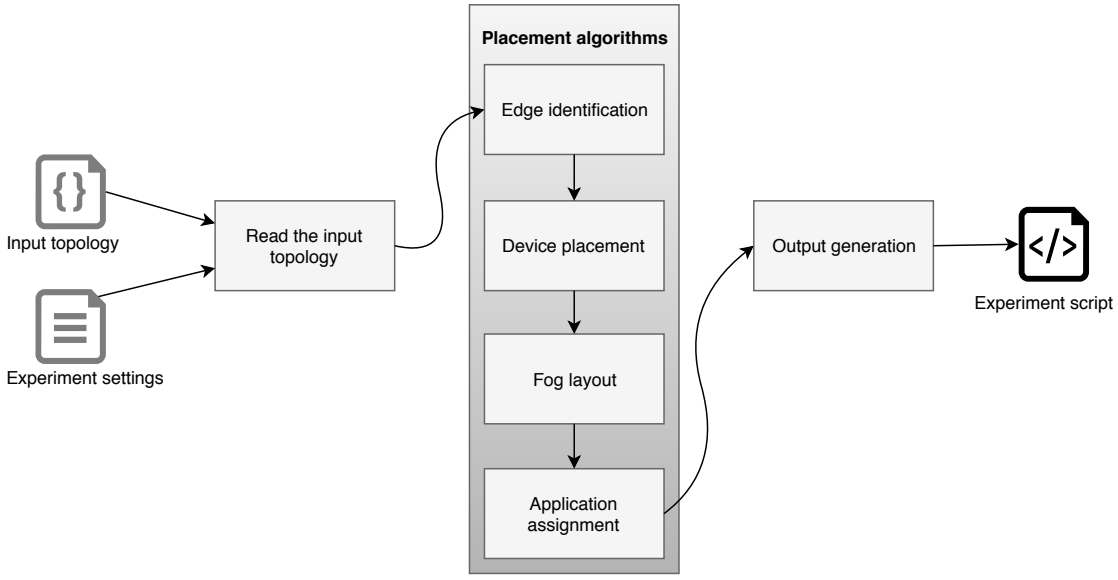



**Figure 4.1:** Overview of the workflow.

This Chapter describes the concepts, extensions, and improvements implemented in this work. The general workflow of EmuFog depicted in Figure 4.1 remains unchanged compared to the previous version. However, much of the internal workings, as well as the input configuration, has been refactored.

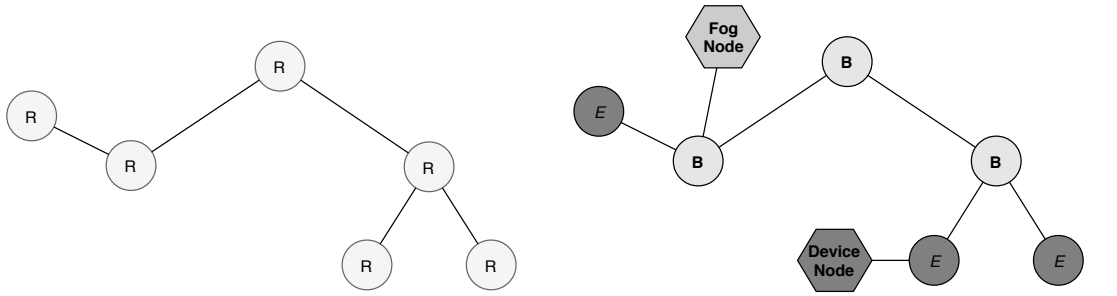
Each execution of EmuFog starts at the leftmost point of the workflow by providing some form of network topology and a configuration file. This input is then parsed into the internal topology representation described in Section 4.2, after that the desired placement algorithms described in Section 4.3 are executed. Finally, the desired experiment script described in Section 4.4 is generated.

Below, following the steps of execution, each step will be described in more detail. Concepts and abstractions for all relevant system entities will be introduced and explained.

## 4.1 Input Topologies

The first step of each execution is to read and parse the provided network topology. Internally, the topology representation is flexible enough, that each undirected graph can be used as input. The presented version of EmuFog currently only supports the BRITE input format. Although, the implementation is done in a way so that support for additional or custom formats is possible. The provided topology will be mapped onto the internal graph representation. This step is mapping links and router nodes to the respective nodes in the topology representation.

## 4.2 Topology Representation




**(a)** Initial, unclassified network topology

**(b)** Classified network topology with assigned node entities

**Figure 4.2:** Internal topology representation

After successfully reading the input, the topology is internally represented as an undirected, connected weighted graph Figure 4.2a. This topology graph provides the basis for all subsequent steps. All steps are performed on the same graph and are either adding nodes and links or are augmenting existing entities with additional information. This additional information can reach from applications that are to be associated with specific nodes to configurations that describe node characteristics. A topology contains two entity types, edges which represent network links, and a generalized node wrapper object. Specifics are described in the following subsections. Depicted in Figure 4.2b is an example of a topology with classified router nodes and connected fog and device nodes.

$$G = (V, E) \tag{4.1}$$

$$v \in V(G) = \{Router,\ Device\ node,\ Fog\ node\} \tag{4.2}$$

$$\{u, v\} \in E(G) = Link \tag{4.3}$$

**Figure 4.3:** Formal graph definition

### 4.2.1 Edge Abstractions

The set of edges $E(G)$ represents the network links. Each edge has the following associated attributes:

- **unique identifier** in order to address specific link instances during the execution of placement algorithms.
- Two edge weight parameters **delay** and **bandwidth** to model system characteristics.

### 4.2.2 Node Abstractions

The EmuFog system architecture includes three different node types: *router nodes*, *fog nodes* and *device nodes*. Router nodes are rather intuitive, they represent network routers and can be classified either as a backbone, edge or simple router instances. Fog and device nodes, on the other hand, have two additional concepts associated with them, depicted in Figure 4.4. First, the *node–type* defining node characteristics e.g hardware limitations. And second, the *node–configuration* containing applications associated to the respective node. Conceptually, fog and device nodes are very similar. However, they have different placement restrictions in terms of which router types they can connect to. Additionally, each subclass has unique parameter fields. The *node–configuration* can contain $0..n$ associated applications.

#### Router Nodes

Represent network routers in the topology. They can be classified either as a

- **Router** modeling the initial unclassified state.
- **Backbone router** as part of the backbone router set.
- **Edge router** sitting at the network edge and serving as a access point for device nodes.

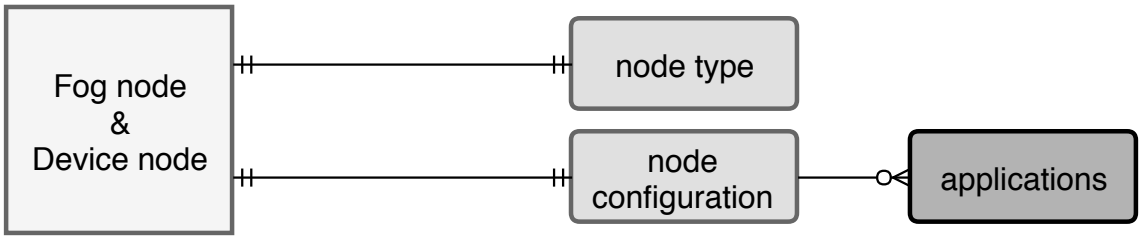



**Figure 4.4:** Entity relationship diagram for fog and device nodes

### Fog Nodes

The perception of what qualifies as a fog node can be different from scenario to scenario. Fog nodes can reach from intermediate server instances with multiple cores to very small single-core nodes. Via the *FogNodeType* configuration the following characteristics, among others, can be defined:

- **hardware limitations** e.g number of cpu–cores and available memory
- **connectivity limitations** like bandwidth and latency
- **maximum connections** the respective instance is able to serve
- **cost** to model different placement costs

That way users have the ability to model fog nodes suitable for their experiment. Fog nodes can be placed anywhere in the network as there are scenarios conceivable where fog nodes are placed at the network edge along with device nodes. In order to provide users maximum flexibility, the underlying topology representation aims to make as few assumptions as possible.

### Device Nodes

Represent all different device types required for an experiment. Similar to fog nodes the *DeviceNodeType* configuration allows the configuration of node characteristics. Hardware and connectivity limitations can be defined in the same way. In contrast to fog nodes, devices can only be connected to routers that are classified as a **edge router**.

This new topology representation simplifies the creation of custom topology nodes and introduces two novel levels of abstraction. In particular, the *node–configuration* now allows for fine–grained control of application assignment to fog or device nodes.

## 4.3 Placement Algorithms

The placement algorithms form the heart of the framework and model the complete experiment logic. Covered logic ranges from topology classification over fog layout creation to application assignment. The algorithms are executed sequentially, each operating and augmenting the same graph instance. The presented work offers a four–step approach. But both the number of executed algorithms and the contained logic can be customized to suit scenario–specific needs.

The algorithms for edge identification, device distribution and fog layout creation are based on solutions presented by Graser [Gra17]. Especially the algorithms for edge identification are conceptually identical. However, due to the additional levels of abstraction, e.g. multiple applications per node and the new router types, the algorithms had to be adapted accordingly. The fourth step is a novelty presented in this work.

The following subsections describe the purpose, concept, and complexities of each executed step. There are no universal solutions, especially for the fog layout creation and edge identification. Therefore, presented algorithms can only be seen as proposals.

### 4.3.1 Edge Identification

At first, the network edge for the provided topology is determined. Again, there is no single correct approach to classify the network edge. The perception of what classifies as an edge router depends on the executed experiment. The proposed algorithms from Graser use router characteristics like edge degree, among others to classify the edge. Initially, the set of backbone routers is empty and routers are simply classified as **router**. Algorithm 4.1 describes the edge identification process. The algorithm takes the topology as input and modifies existing router nodes by following the proposed four steps.

- Mark AS Edge Node described in Algorithm 4.2
- Convert High Degree Nodes described in Algorithm 4.3
- Build Single Backbone described in Algorithm 4.4
- Convert Remaining Routers described in Algorithm 4.5

After the heuristic was successfully applied to the topology the set of router nodes only contains either **backbone** or **edge routers**

**Algorithm 4.1** Edge identification algorithm

```
1: routers ← {}
2: procedure IDENTIFYEDGE(G(V, E))
3:     routers ← G.GETROUTERS
4:     IDENTIFYBACKBONE(G)
5:     return G
6: end procedure

7: procedure IDENTIFYBACKBONE(G)
8:     MARKASEDGENODES(G)
9:     CONVERTHIGHDEGREES(G)
10:    BUILDSINGLEBACKBONE(G)
11:    CONVERTREMAININGROUTERS(routers)
12: end procedure
```

### Mark AS Edge Nodes

The underlying idea of this heuristic is that if routers lie at the border of an autonomous system the probability of being a **backbone router** is very high. The algorithm works as follows. It checks for all nodes whether they have a link connecting routers from two different autonomous systems. If so, the routers connected by the respective edge are added to the set of backbone routers.

**Algorithm 4.2** Mark AS Edge Nodes

```
1: neighbors ← {}
2: procedure MARKASEDGENODES(G)
3:     for all node ∈ G.NODES do
4:         neighbors ← G.ADJACENTNODES(node)
5:         for all neighbor ∈ neighbors do
6:             if ISCROSSASEDGE(node, neighbor) then
7:                 if !ISBACKBONEROUTER(node) then
8:                     node.SETTYPE(BACKBONE_ROUTER)
9:                 end if
10:                if ISROUTER(neighbor) then
11:                    neighbor.SETTYPE(BACKBONE_ROUTER)
12:                end if
13:            end if
14:        end for
15:    end for
16: end procedure
```

### Convert High Degree Nodes

Here, the proposed algorithm assumes that router nodes with a higher than average edge degree are members of the backbone. Therefore, the algorithm sketched in Algorithm 4.3 calculates the average node degree for the given topology instance and then compares each router node to the calculated $averageDegree$ value. If a router has a degree above the calculated average he will be classified as a backbone router.

**Algorithm 4.3** Convert High Degree Nodes

```
1: procedure CONVERTHIGHDEGREES(G)
2:     averageDegree ← CALCULATEAVERAGEDEGREE(G) × BACKBONE_DEGREE_PCT
3:     for all router ∈ routers do
4:         aboveAverage ← G.DEGREE(router) ≥ averageDegree
5:         if aboveAverage then
6:             router.SETTYPE(BACKBONE_ROUTER)
7:         end if
8:     end for
9: end procedure
```

### Build Single Backbone

Up to this point, the presented heuristics may have classified multiple routers as backbone routers. This step is concerned with connecting backbone routers into a subgraph. The Algorithm 4.4 starts by selecting one backbone router and then iterates over the graph structure following the Breadth Fist Algorithm. That way the complete topology is explored and intermediate routers between backbone routers are classified as backbone routers as well.

**Algorithm 4.4** Build Single Backbone

```
 1: procedure BUILDSINGLEBACKBONE(G)
 2:     backboneRouters ← G.GETBACKBONEROUTERS
 3:     visited ← {}
 4:     seen ← {}
 5:     predecessors ← {}
 6:     b ∈ backboneRouters
 7:     Q ← {b}                                   // Get first router ∈ backboneRouters
 8:     while Q ≠ {} do
 9:         c ← Q.DEQUEUE
10:         visited ← c
11:         for all neighbor : G.ADJACENTNODES(c) do
12:             if neighbor ∉ visited ∧ neighbor ∈ seen then
13:                 if neighbor ∈ Q∧!ISCROSSASEDGE(c, neighbor) then
14:                     if ISBACKBONEROUTER(c) ∧ neighbor ∈ Routers then
15:                         predecessors.PUT(neighbor, c)
16:                     end if
17:                 end if
18:             else
19:                 predecessors.PUT(neighbor, c)
20:                 Q ← neighbor
21:                 seen ← neighbor
22:             end if
23:         end for
24:         if ISBACKBONEROUTER(c) then
25:             predecessor ← predecessors.GET(c)
26:             while predecessor ∈ Router do
27:                 predecessor.SETTYPE(BACKBONE_ROUTER)
28:                 predecessor ← predecessors.GET(predecessor)
29:             end while
30:         end if
31:     end while
32: end procedure
```

#### Convert Remaining Routers

This last step assumes that all routers that are still unclassified are part of the edge router set. Therefore, Algorithm 4.3 iterates over the set of routers and classifies each remaining router as an edge router.

**Algorithm 4.5** Convert Remaining Routers

```
1: procedure CONVERTREMAININGROUTERS(routers)
2:     for all router ∈ routers do
3:         if router.EQUALS(ROUTER) then
4:             router.SETTYPE(EDGE_ROUTER)
5:         end if
6:     end for
7: end procedure
```

### 4.3.2 Device Distribution

The device distribution placement algorithm allows users to specify the number and locality of provided device node types, in order to emulate different network workloads. Device nodes can be placed anywhere at the previously classified network edge. Here, the number, type, and location of required device nodes can be defined. The proposed work contains a simple device distribution algorithm which places a random number of devices at the network edge.

### 4.3.3 Fog Layout Creation

Similar to the edge identification there is no one solution to fog layout models. As already motivated in Chapter 2 there are many approaches for fog architectures.

The proposed fog layout uses a simple heuristic by specifying delay and hop count boundaries. The goal is to cover all edge devices with as few fog nodes as possible while still complying to the specified limitations.

The presented Algorithm 4.6 works as follows. As long as there are edge routers, that have device nodes connected to them, which are not covered by a fog node, and there are still fog node capacities available, a set of candidate routers is determined. Out of this set, the router that covers most devices under the given limitations is selected as a connection point. The next fog node instance will be connected to this router. Next, given that there are more than one *fog–node–types* provided. The best fitting node type has to be selected. This is done by using the following cost function to determine the most valuable *node–type* for the corresponding connection point.

$$ratio = \frac{connectedDevices - maximumConnections}{costs} \tag{4.4}$$

This $ratio$ is calculated for each *node–type* and the instance with the highest ratio is selected.

After the fog node was successfully placed, the set of edge routers has to be updated. All covered edge routers are removed from the edge routers list. A router is covered if all connected devices can be served by already placed fog nodes.

This process repeats until all devices are covered or there are no more fog nodes available. In the first case, execution was successful in the second case no suitable fog layout could be determined under the given limitations.

**Algorithm 4.6** Identify Fog Nodes

```
1:  procedure IDENTIFYFOGNODES(G)
2:      edgeRouters ← G.GETEDGEROUTERS
3:      remainingNodes ← MAXFOGNODES
4:      threshold ← COSTTHRESHOLD
5:      delayBoundary ← DELAYBOUNDARY
6:      while edgeRouters ≠ {} do
7:          if remainingNodes ≥ 0 then
8:
9:              canditateRouters ← DETERMINECANDIDATEROUTERS
10:             connectionPoint = GETBACKBONENODEWITHHIGHESTEDGECOVERAGE
11:             fogNodeToPlace = new FogNode(FINDCOSTOPTIMALNODETYPE(connectionPoint))
12:             PLACEFOGNODE(connectionPoint, fogNodeToPlace)
13:
14:             for all coveredRouter ∈ edgeRouters do
15:                 if coveredRouter.COVERED then
16:                     edgeRouters.REMOVE(coveredRouter)
17:                 end if
18:             end for
19:
20:             remainingNodes.DECREMENT
21:         end if
22:     end while
23: end procedure
```

### Determine Candidate Routers

As part of the fog node placement algorithm Algorithm 4.7 uses a greedy Dijkstra [Dij59] algorithm to calculate all shortest paths for the current $edgeRouter$. Then, in order to find all routers that fulfill the specified delay and hop count requirements the set of $backboneRouters$ is filtered and only valid candidates are added to the result set.

**Algorithm 4.7** Determine Candidate Routers

```
1:  procedure DETERMINECANDIDATEROUTERS
2:      for all edgeRouter ∈ edgeRouters do
3:          CALCULATESHORTESTPATHS(edgeRouter)        // start Dijkstra for current edgeRouter
4:
5:          for all backboneRouter ∈ G.GETBACKBONEROUTERS do
6:              if (backboneRouter.SHORTESTPATH ≤ threshold)
7:                  ∧ (backboneRouter.DISTANCE ≤ delayBoundary) then
8:
9:                  candidateRouters ← backboneRouter
10:             end if
11:         end for
12:     end for
13: end procedure
```

### 4.3.4 Application Assignment

This placement algorithm step is a novelty presented in this work. After the devices and fog nodes are placed in the topology, one has to specify on which instances applications should be placed. The assignment is partitioned into two separate steps. First, the device application mapping is executed. Here device applications are assigned to devices and required configurations can be made. Such configurations could include defining exposed Ports or definition of required Environment variables. Second, the fog application mapping in which fog applications are assigned and configured. The goal of this placement algorithm is to provide fine–grained control over applications required to execute the respective experiment.

## 4.4 Output Generation

At the end of the workflow, the experiment script is generated. In this step routers, links, devices and fog nodes are translated from the internal representation to the desired output format. Also, assigned applications are placed. In order to satisfy the *flexibility* objective this step can be adapted to comply to experiment–specific requirements. Presented version of EmuFog provides implementations for Containernet experiments. However, custom implementations can be added.

# 5 Implementation

In this Chapter, the realization and implementation of previously introduced system concepts is presented. This work extends the open–source tool EmuFog[1]. First, an overview of the proposed system architecture is given. Then the implementation of core concepts including changes compared to the previous EmuFog version will be explained in more depth.

## 5.1 System Architecture

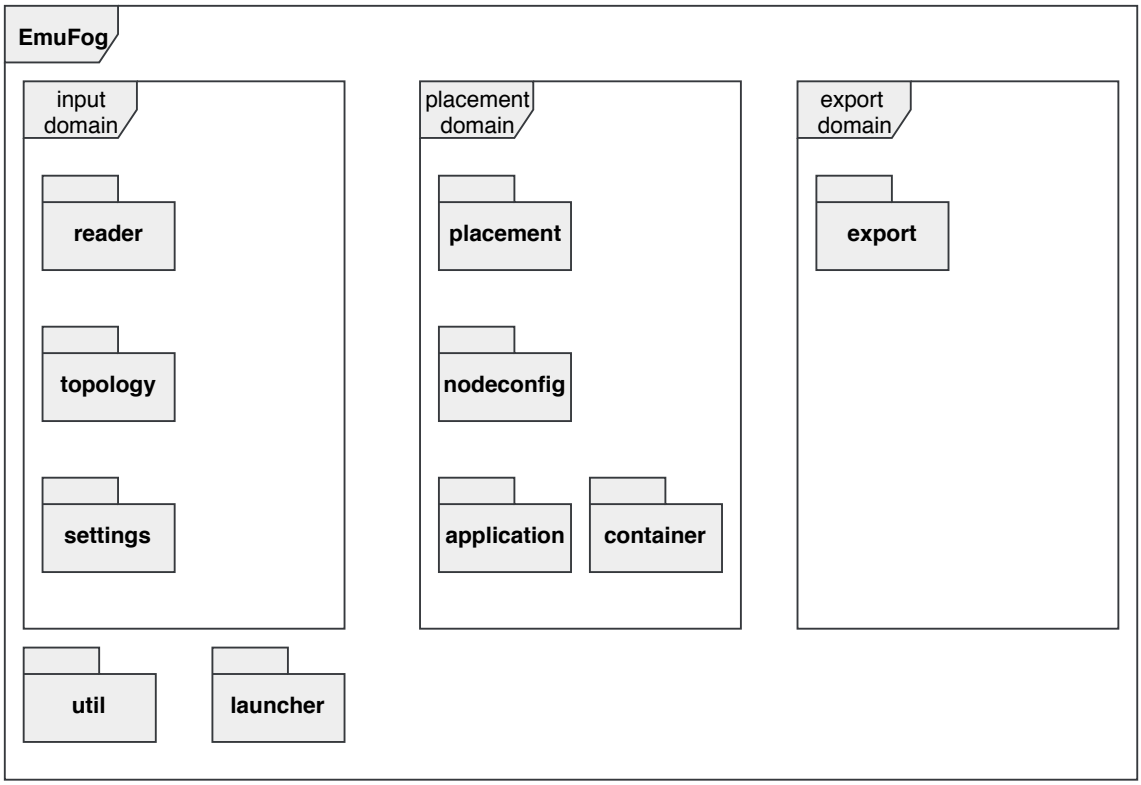



**Figure 5.1:** EmuFog system architecture.

[1]Emufog is available at https://github.com/emufog/emufog

The EmuFog System Architecture depicted in Figure 5.1 can be divided into three domains. First, the input domain which contains logic to define experiment settings, read input topologies and to initialize the internal topology representation. The second domain is concerned with the execution of the placement algorithms described in Section 4.3. And finally, the export domain generates all required output files in order to run the experiment.

### 5.1.1 Input Domain

This domain contains all packages concerned with input specification, input topology parsing, and topology representation.

#### Reader Package

This package contains the input reader implementations. In contrast to the previous EmuFog implementation, the proposed work only provides a $BriteFormatReader$. Nonetheless, the general structure is identical. For future extensions, only the abstract $GraphReader$ class has to be implemented. The reader package will be called by the $Topology$ class in order to instantiate the topology representation.

#### Settings Package

The settings package is responsible to read in experiment settings. The experiment settings are defined via a YAML markup file. EmuFog uses *Jackson* by FasterXML[2] to transform the input into a Java object. The $Settings$ class holds all relevant experiment information and is accessible as a singleton object throughout the execution of the application.

#### Topology Package

Here the complete topology is specified. Nodes types and links are defined and the topology object is created in the $Topology$ class. This class is responsible to provide the topology graph throughout the execution of the application. The topology object is most important for the execution and all subsequent steps work on the same instance and are either adding information or entities to the same instance.

[2] https://github.com/FasterXML

### 5.1.2 Placement Domain

This domain contains the complete placement algorithm logic as well as the required node–configurations and all node–application related implementation. Applications are modeled in the *Application* package. Each Application instance contains a Container object. This abstract Container class functions as a simple wrapper to allow support for different containerization solutions. Currently, only Docker is implemented but if support for other technologies is required one simply has to implement the abstract Container class.

The *Node Configurations* package contains the node–type and node–configuration implementation for fog and device nodes.

#### Placement Algorithms

Here, the proposed placement algorithms are implemented. To realize the proposed approach. Four interfaces are provided, one for each step. The presented work includes default implementations for each step as presented in Section 4.3. Again, custom implementations can be added over the provided interfaces. At execution, each algorithm calls the *Topology* package to get the topology object to work on.

### 5.1.3 Export Domain

After the topology was successfully created, the Export Domain is concerned with the creation of executable experiment scripts. To allow easy adaption for custom export types an interface $ITopologyExporter$ is provided. The proposed work supports exports to Containernet with the $ContainernetExporter$ class. To extend EmuFog with support for custom output types one has to implement the provided interface.

## 5.2 Contribution

In this section, major changes to the previous version are explained and motivated in more depth. The most important changes took place in the following areas. First, the way how users define and specify experiments was simplified. Second, the internal topology representation was completely reworked. Third, as one of the main objectives of this work, the concept of multi–tiered nodes was implemented. Finally, the ability to control the assignment of applications was introduced and a Containernet exporter was added.

### 5.2.1 Input Configuration

Previously, the required experiment specifications were defined via a settings file using the JavaScript Object Notation (JSON)[3]. Although, JSON is a very common and universal data interchange format it is tedious to write and prone to typing errors. Especially, representation of nested objects is inconvenient due to the heavy usage of parentheses in JSON. In order to make the settings file more human–friendly the data format was changed to YAML. The advantages of YAML are its readability and simple syntax without losing any expressivity.

Also, execution of the previous EmuFog version heavily relied on command line arguments to parameterize the application. For example, the provided topology type was defined via a command line argument. The main shortcoming of this approach is that selected parameters are not replicable after execution. To increase transparency and repeatability of experiments most of the configuration parameters were moved into the settings file. The set of available parameters is introduced in Appendix A.

Furthermore, mechanisms to dynamically select specific versions of placement algorithms are introduced. Now, users are able to specify the desired implementation of let's say the *EdgeIdentification* algorithm by providing the classpath in the respective section in the settings file. Then at execution, the desired implementation is selected via dynamic class loading. Otherwise, the application will fall back to defined default algorithms. That way users are able to run experiments with different algorithms without recompiling the application.

In addition to that, applications can be defined in the settings. Conceptually, the implementation follows the schema of *docker-compose*[4]. Applications for fog and device nodes can be defined and are then accessible for the application placement algorithm during execution of EmuFog. The implemented features are documented in Appendix A.

[3] JSON specification is available at https://tools.ietf.org/html/rfc7159
[4] Docker compose is available at https://docs.docker.com/compose/

### 5.2.2 Topology Representation

The internal graph representation of EmuFog experienced a complete remodeling. The previous version used a custom–made graph data structure. While trying to extend the architecture to support multi–tiered nodes and a more flexible placement algorithm execution, this custom data structure limited the possibilities of extension severely. For example, it would have been very difficult to integrate the introduced *node–configurations*, containing the application definitions, to the previous data structure. Therefore, with future extensions in mind, the graph structure was replaced by a commonly used graph implementation provided by the *Guava*[5] library. The flexibility of the Guava graph implementation (almost every Java object can be used as a node), as well as the comprehensive feature set, are beneficial to EmuFog users and developers.

Furthermore, the topology representation lives in the application as a singleton object (there is never more than one topology object). And conceptually follows a sequential builder pattern depicted in Figure 5.2. Upon instantiation, the *build* method is responsible to call all configured placement steps. First, the selected *read* method to instantiate the graph is called. Then the edge is identified, the devices are assigned and all other subsequent steps are called. After all build steps are completed successfully the topology can be exported using the desired exporter implementation.

The benefit of this approach is its flexibility regarding the number of executed steps as well as the actual implementation of each step. As mentioned in Section 5.2.1 the executed algorithms can be defined in the settings file and are then dynamically loaded in the *build* method.

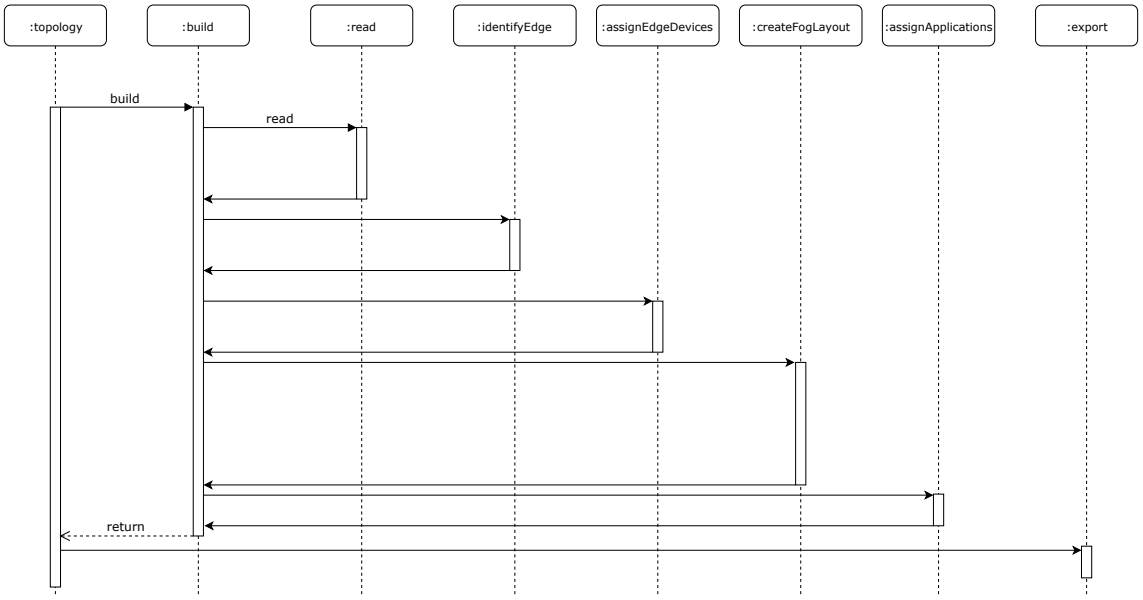


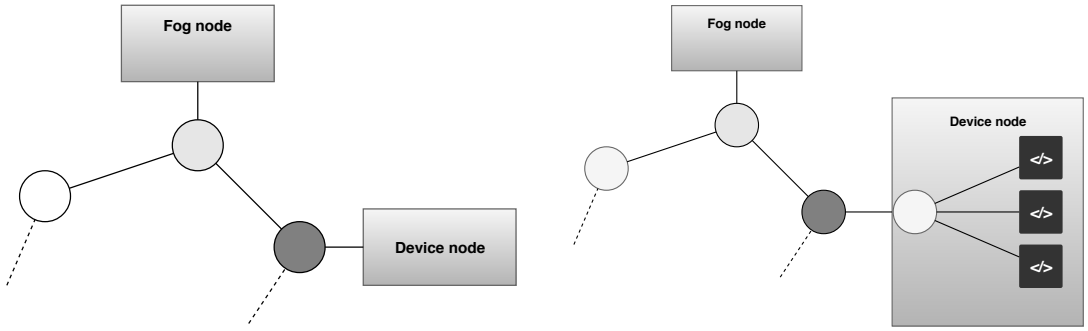

**Figure 5.2:** Sequence diagram for topology creation

[5] Guava: Google Core Libraries for Java https://github.com/google/guava

### 5.2.3 Multi–Tier Node Abstraction

One of the main objectives of this work was the introduction of nodes with multi–application capabilities. The previous version of EmuFog only allowed one application per fog or device node. These applications are bundled in Docker containers. This one–to–one limitation is due to the fact that Containernet, as the underlying emulation software, only supports nodes with exactly one container assigned. Two possible approaches to a solution were evaluated.

A possible solution would have been to adapt the Containernet implementation to suit our requirements. This would have been a very laborious and time–consuming process. Because the eventually proposed solution would still have to be accepted by the Containernet developers. The second and implemented approach depicted in Figure 5.3 makes use of a unique property of emulated network links which is the ability to emulate links with zero delay. That way multiple applications can be connected without losing the characteristics of the topology under test.




**(a)** Previous, single–application architecture **(b)** Proposed multi–application architecture

**Figure 5.3:** Multi-tier node abstraction

In Figure 5.3a the previous node abstraction is depicted. A fog node running a single container instance is connected to a backbone router and a device node is connected to an edge router. The proposed multi–application architecture in Figure 5.3b extends the node abstraction. For each node, an intermediate router is added to the topology. This intermediate router is connected to the respective backbone or edge router. The link characteristics are specified in the *node–configurations*. Now applications associated to the node can be connected via zero delay and maximum bandwidth links to the intermediate router. Because emulation allows links with zero delay, connected applications behave to the rest of the system as if they were connected directly to the actual topology router.

Listing 5.1 shows the actual implementation in Containernet. In line 3 the intermediate router is added and connected to the corresponding topology router in line 5. Then the application is connected to the intermediate router in line 14.

**Listing 5.1** Multi Tier Node in Containernet

```
# createMultitierSwitch
info('*** Create multi tier switch for d11\n')
mtsd11 = net.addSwitch('mtsd11')
info('*** Adding link from mtsd11 to r5\n')
net.addLink(mtsd11, r5, cls=TCLink, delay='10.0ms', bw=5.0)

# serviceA
info('*** Adding docker container d1117 with ubuntu:trusty\n')
d1117 = net.addDocker('d1117', ip='10.0.0.2', dimage="ubuntu:trusty", mem_limit=256, ...)

# connect application to mtsd11
info('*** connect application to  mtsd11\n')
info('*** Adding link from d1117 to mtsd11\n')
net.addLink(d1117, mtsd11, cls=TCLink, delay='0.0ms', bw=1000.0)
```

Although this approach increases the number of emulated routers by the number of connected nodes, we still believe this solution is feasible. Proposed solution only makes use of general features of emulated networks. And should, therefore, be extendable to other emulation tools as well. A custom Containernet extension, on the other hand, would only work for Containernet and thus limit the flexibility of EmuFog. For Example, an easy integration of MaxiNet wouldn't be possible anymore. We believe *flexibility* is in this case more important than *efficiency* and accept the associated increased emulation complexity.

### 5.2.4 Containernet Output

This version of EmuFog replaced the previous MaxiNet experiment output with a Containernet implementation. Mostly due to the fact that running Containernet experiments has a lower overhead. Containernet itself is available as a Docker Container and can, therefore, be used on most platforms. MaxiNet, on the other hand, is only optimized for the Ubuntu distribution. During implementation, a bug in Containernet that prevented the configuration of port–mappings in the experiment script was encountered. Fortunately, the Containernet contributors were very helpful and responsive and fixed the bug.

# 6 Evaluation

In this chapter, the usability and functionality of EmuFog will be evaluated. Therefore a DDS application will be deployed on different fog layouts and latency tests will be executed. First, the used applications *Cassandra* and the *YCSB* will be introduced. Then, the necessary preparations in order to run the applications in an emulated network are described. Next, the evaluation setup and tested fog layouts are presented. Finally, the results will be discussed. This evaluation follows two goals. First, to test the usability and functionality of the proposed framework. Second, to examine the behavior of a DDS in a fog context, similar to FogStore [MGSR17a].

### Cassandra

Cassandra [LM10][1] is a widely used highly scalable and fault–tolerance distributed data store. Cassandra works as a key value store and was developed by Lakshman et al. at Facebook as a open–source implementation under the Apache license. Several large software companies use Cassandra in production. One of the largest cluster has over 75.000 Nodes storing over 10 PB of data.

### Yahoo Cloud Serving Benchmark

YCSB [CST+10][2] proposed by Cooper et al. is a framework to benchmark and classify performance of distributed database systems. The framework provides several different workloads to test characteristics like read versus write performance, latency and consistency. Furthermore, custom tests can be implemented.

[1] Cassandra is available at http://cassandra.apache.org/
[2] YCSB is available at https://github.com/brianfrankcooper/YCSB/wiki

The framework comes with six core workloads:

- **Workload A**: Update heavy workload with a mix of 50/50 reads and writes.
- **Workload B**: Read mostly workload with a 95/5 read/write mix.
- **Workload C**: Read only with 100% read.
- **Workload D**: Read latest workload. Here, new records are inserted and the most recently inserted record read most often.
- **Workload E**: Here, short ranges of records are queried.
- **Workload F**: Read–modify–write, in this workload, records are written, modified and then read again.

## 6.1 Evaluation Preparations

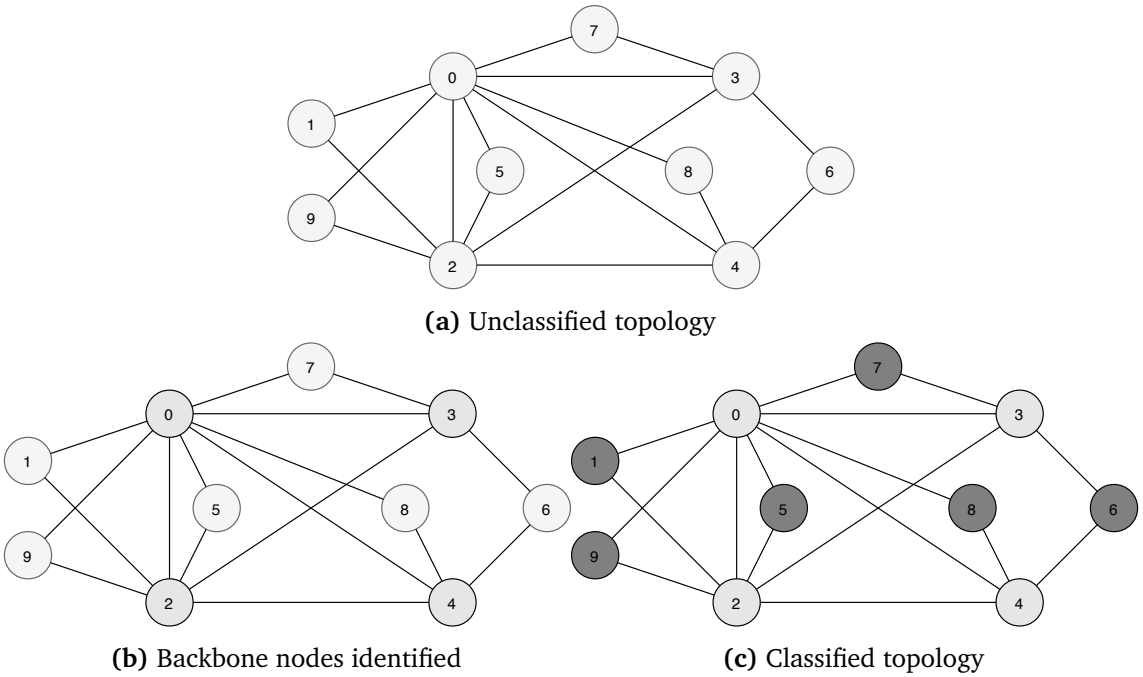
In this section, important preparations in order to run the experiments are described. In particular, implementation details and lessons learned are documented. Furthermore, one example for topology classification will be discussed.

**Cassandra** To be able to run Cassandra with EmuFog, Containernet and YCSB several prearrangements have to be made. First, one has to configure the correct network for Cassandra to start on. Per default, the docker network interfaces are used. In order to connect to the emulated network running in Containernet the startup sequence of Cassandra has been adapted using a custom startup script. This startup script also creates necessary tables to run YCSB workloads at boot time. To be able to support multi–node Cassandra clusters a Seed Node is required and wait times for nodes to discover each other have to be taken into account. Therefore, a custom Application Assignment Placement algorithm was introduced to EmuFog. This placement algorithm takes care of setting the respective environment variables to define the Seed Node and specify wait times.

During the evaluation, two different input topologies were used. One small topology consisting of 10 nodes and one larger topology with 100 nodes. For the small topology, three different fog layouts have been generated and for the larger topology two.

### Topology Classification

Depicted in Figure 6.1 is the classification process of the small input topology. Fist, in Figure 6.1a the raw unclassified is shown. Then during the execution of EmuFog the edge identification algorithm identified the nodes 0, 2, 3, and 4 as backbone nodes in Figure 6.1b. Finally, the remaining nodes are classified as edge nodes in Figure 6.1c.




**(a)** Unclassified topology

**(b)** Backbone nodes identified

**(c)** Classified topology

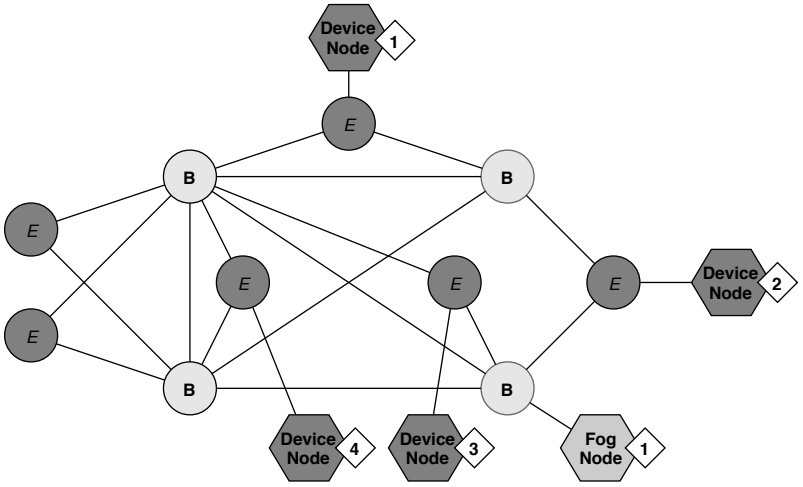
**Figure 6.1:** Input classification

## 6.2 Evaluation Setup

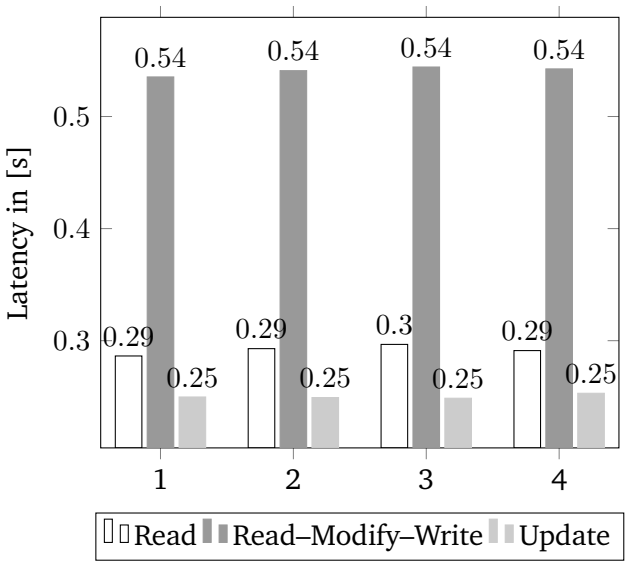
All measurements were carried out on the same system. The underlying hardware is a 4 x 16 Core Intel(R) Xeon(R) CPU E7-4850 v4 @ 2.10GHz and 1TB RAM. In the following sections, the three generated fog layouts are depicted and measurement results are presented. First, a centralized layout with a single fog node has been created to model a cloud system architecture. Then a simple multi–node fog layout was measured and finally, an edge fog layout has been created. Additionally, two different layouts on the larger 100 node topology were evaluated. Several different YCSB Workloads were executed. For each fog layout, the results of Workload F are shown. The Workload was executed with two different consistency levels. First, with level *one*, read from one arbitrary node and write to one arbitrary node. Second, with level *all*, read and write from and to all nodes.

### 6.2.1 Centralized Layout Measurements

Here, the placement limitations and node configurations were set in such a way that one fog node is capable to serve all device nodes. The specified fog node type can serve up to ten devices. The following placement restrictions were made: no further than two hops away and lower than 20 ms link delay. The generated topology is depicted in Figure 6.2a. The *device placement* algorithm distributed four devices at the network edge and the *fog layout* algorithm determined backbone router four as the most suitable connection point. In this topology, there is only one Cassandra instance running. In Figure 6.2b the results of Workload F are depicted. From left to right latencies for device node 1 to 4 are shown.




**(a)** Centralized fog node layout



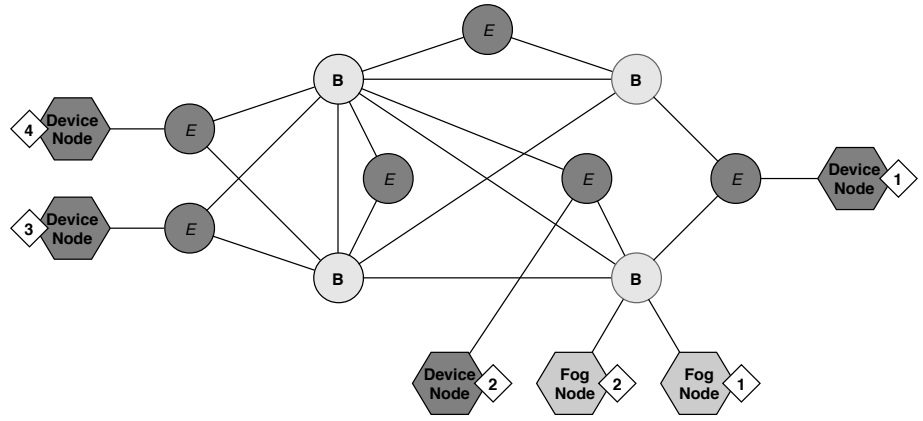

**(b)** Workload F: **Read–Modify–Write** with consistency level *one*

**Figure 6.2:** Centralized fog node layout measurements

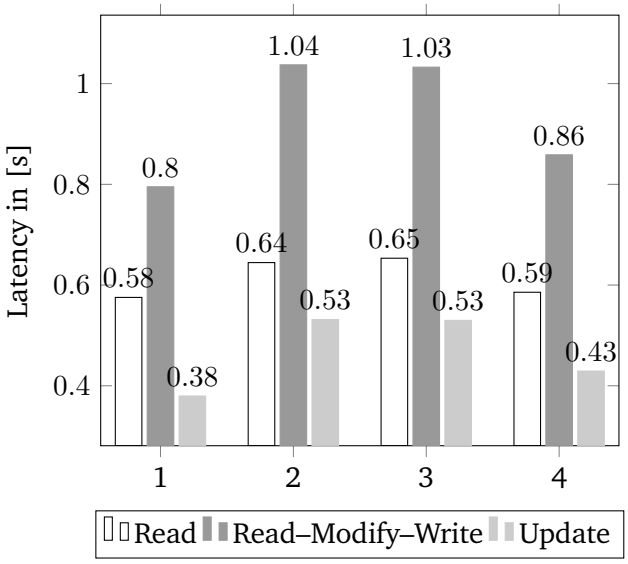
### 6.2.2 Multi Fog Node Layout Measurement

Here, the fog nodes had fewer capabilities than in the first layout. The placement restrictions were similar: no further than two hops away and lower than 20 ms link delay. The generated topology is depicted in Figure 6.3a. The *device placement* algorithm again distributed four devices at the network edge and the *fog layout* algorithm determined backbone router four as the most suitable connection point. Due to the lower capabilities of the defined fog node types, two instances were required to serve all devices. In this topology, two Cassandra instances are deployed.

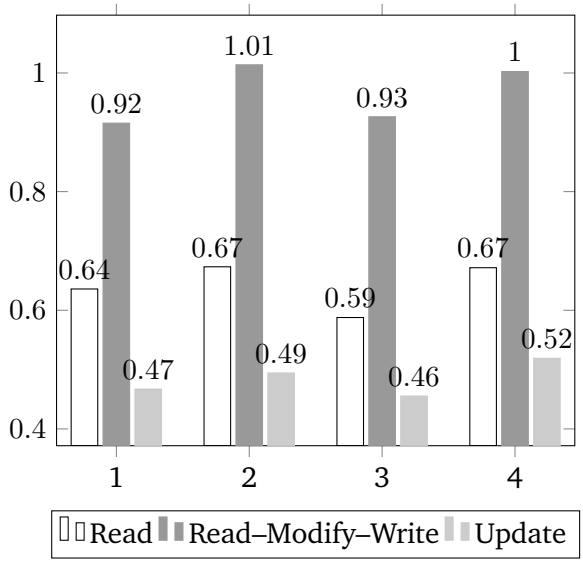
In Figure 6.3b and Figure 6.3c the results of Workload F are depicted. From left to right latencies for device node 1 to 4 are shown.




**(a)** Multi node fog layout




**(b)** Workload F: **Read–Modify–Write** with consistency level *one*



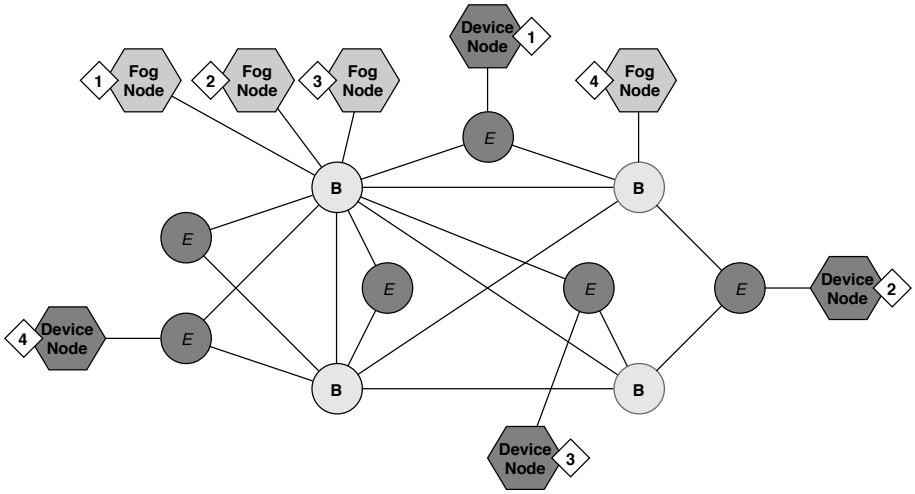

**(c)** Workload F: **Read–Modify–Write** with consistency level *all*

**Figure 6.3:** Multi Fog Node Layout

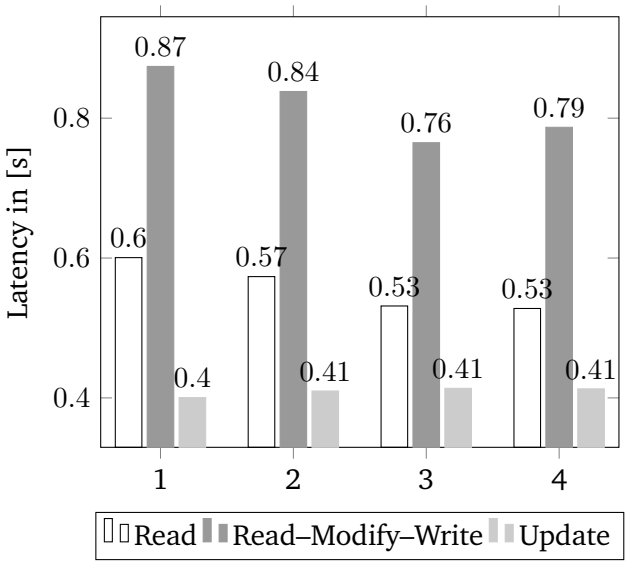
### 6.2.3 Edge Fog Layout Measurements

Here, the placement restrictions were the strictest: not further than one hop away and lower than 20 ms link delay. In addition to that defined fog node types were only capable to serve one connection. The generated topology is depicted in Figure 6.4a. In order to cover all devices four fog node instances were placed in this topology. In this topology, a Cassandra instance is running on each fog node.

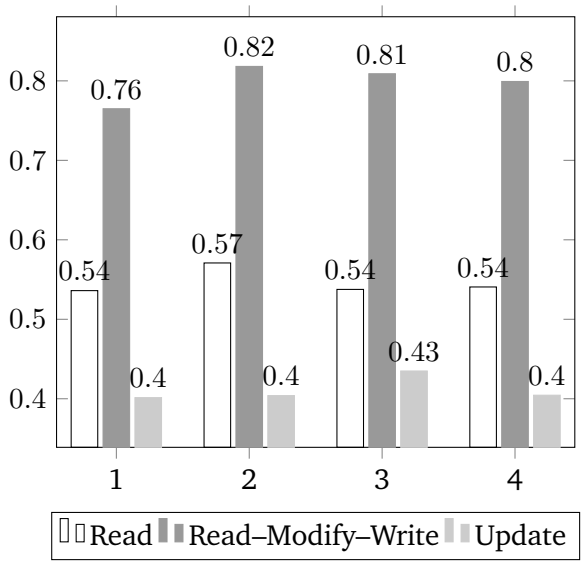
In Figure 6.4b and Figure 6.4c the results of Workload F are depicted. From left to right latencies for device node 1 to 4 are shown.




**(a)** Edge fog node layout




**(b)** Workload F: **Read–Modify–Write** with consistency level *one*




**(c)** Workload F: **Read–Modify–Write** with consistency level *all*

**Figure 6.4:** Edge fog node layout

### 6.2.4 Larger Topologies

Two larger topologies were evaluated as well. The first containing 7 Cassandra instances and 32 device nodes. And the second, containing 17 Cassandra instances and 22 device nodes. Both, the 7 and 17 node Cassandra cluster could be started. The YCSB tests were executed from one of the devices. The placement restrictions for the large topologies were:

- **1**: not further than five hops away and lower than 20ms link delay. Maximum connections of fog node type 5.
- **2**: not further than one hop away and lower than 20ms link delay. Maximum connections of fog node type 20.

In Figure 6.5 the results of Workload F are depicted. From left to right latencies for topology one and two are shown.

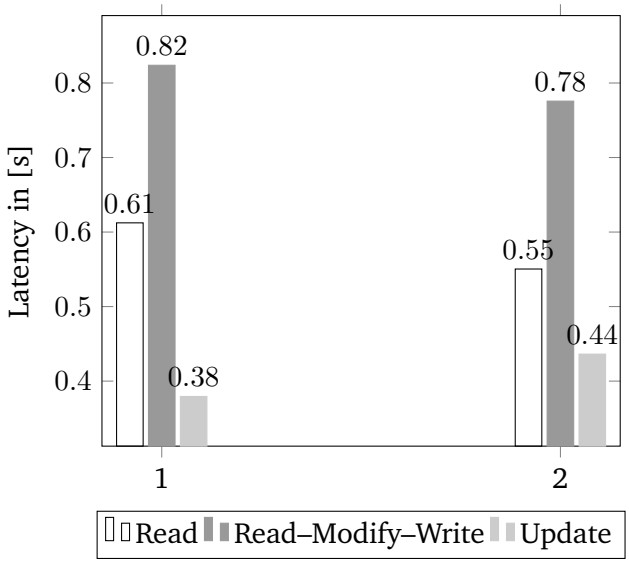



**Figure 6.5:** Workload F: Read–Modify–Write with consistency level *one*

## 6.3 Results and Discussion

The first goal of this evaluation could be reached. It was shown that real–world applications can be configured, deployed and executed using the EmuFog framework and network emulation. Multiple fog layouts were created according to different scenario requirements.

Furthermore, in the larger topologies, one can observe, that latencies are lower in fog layouts that are closer to the edge. Also, update latencies were significantly higher in topologies with more replicas. It is interesting to see that for small topologies, according to the performed measurements, a centralized layout is able to deliver the best performance. Furthermore, it has been shown that performance characteristics for the tested consistency levels are very similar in small topologies. It can be expected that in larger topologies the impact of higher consistency requirements has more impact.

In order to draw conclusions about the behavior of DDS in fog layouts further measurements have to be carried out. Overall, the evaluation was able to show one possible field of application for EmuFog.

# 7 Conclusion

In this work, an improved version of EmuFog was presented. Several shortcomings of the previous version have been resolved. It is now possible to run multiple applications on device and fog nodes. The concept of multi–tiered nodes have been implemented for the Containernet emulation framework. However, the presented solution can also be applied to other emulation frameworks like *MaxiNet*[1] with low overhead.

Furthermore, the workflow of EmuFog has been streamlined and extended with further topology augmentation steps. Beginning at the start of each EmuFog execution, the configuration and definition of scenario specifics have been centralized into a single settings file. This improves repeatability and versioning of executed scenarios.

Then, the internal graph representation and execution of the topology enhancement steps, now called placement algorithms, have been adapted. The desired placement algorithms can be defined via the input settings and are dynamically loaded at application start. Internally, the executed placement algorithms are modeled in an exchangeable fashion.

In order to control the assignment of applications to nodes, a novel placement step was introduced. Now it is possible to specify which applications should run on which node entity. In the settings file application specific configurations e.g. environment variables, scripts can be added. EmuFog comes with default implementations for each of the five placement steps. The existing algorithms for edge identification and fog layout creation have been updated due to the new graph representation and novel application assignment algorithms were added.

To further increase the platform agnostic design goal of EmuFog, a containerized version of the framework is available.

The functionality and usability of EmuFog have been successfully evaluated by benchmarking a DDS application in different fog layouts. It has been shown that the proposed framework is able to run real–world applications and enables users to generate use case specific network topologies in a fast and repeatable way.

[1] Maxinet is available at https://maxinet.github.io/

## Future Work

In future work, several areas of EmuFog could be improved:

- **Placement algorithms**: The set of provided placement algorithms could be extended with more sophisticated solutions. Especially for the edge identification and fog layout creation. It would be conceivable to implement fog layout algorithms building hierarchical or meshed layouts.
- **Investigate alternative fields of application**: The flexible nature of EmuFog and the ability to programmatically control the placement of applications in a predefined topology opens an opportunity to use the framework in additional fields of application. For instance, EmuFog could prove useful for automated integration testing of e.g. microservice architectures. As the services under test can be executed in a completely controlled environment in a fast an repeatable and automatable fashion. That way, the behavior of multiple services can be evaluated during implementation as part of the continuous integration pipeline of the software. And possible limitations could be detected early on.
- **Resource management**: A global resource management representation to configure the overall system capabilities may be useful to system–architects and other users. One could imagine adding a separate placement algorithm to ensure that the constructed topology does not exceed the available resources. The accessible system resources could be modeled via the settings file.
- **Mobile fog and device nodes**: The static character of the current implementation makes an evaluation of mobile fog scenarios difficult. But in the context of fog computing mobility and flexibility are core concepts. It would be desirable to have the ability to model more flexible network topologies. One possible step toward that goal could be the introduction of nodes that can be moved during the execution of the experiment. This could possibly be achieved by providing ways to programmatically bend links from one endpoint to another. That way the mobility of nodes could be modeled.
- **Maxinet exporter**: Presented work only includes a Containernet exporter implementation. To be able to evaluate larger systems an updated Maxinet exporter implementation, incorporating the novel multi–tiered capabilities, could be useful.
- **Input topologies**: Presented work only includes support for BRITE input topologies, to provide users more flexibility support for additional input types is desirable

The presented list of possible further work is not necessarily ordered by importance. Although, the proposals for mobile fog and device nodes, as well as the introduction of additional Placement Algorithms, are straightforward. Additionally, the alternative fields of application cloud be investigated as an additional benefit.

# A How To Use EmuFog

In this section the usage of EmuFog is described. First, the required input data and possible configurations are introduced. Then, the execution of the application is explained. And finally, instructions on how to use the generated output in containernet is are given.

## Input Configuration

**Input Topology** The proposed version of EmuFog is able to work with BRITE topologies. The path to the desired file has to be specified in the settings file as shown in Listing A.1.

**Input Data** Configuration of EmuFog is done via a YAML file. The settings are sectioned into five different segments described in the following.

**Input and Output Settings** First, the basic input and output settings have to be defined. Here, as shown in Listing A.1 paths to the input data and the desired output path are configured. From Line 9 to 14 the placement algorithms can be chosen.

**Listing A.1** Input and Output Settings

```
1  ######################################################################
2                      Input/Output Settings
3  ######################################################################
4  inputGraphFilePath: "100_1.brite"
5  exportFilePath: "containernet_out.py"
6
7  overWriteOutputFile: true
8
9  applicationAssignmentPolicy: "emufog.placement.DefaultApplicationAssignment"
10 devicePlacement: "emufog.placement.DefaultDevicePlacement"
11 edgeIdentifier: "emufog.placement.DefaultEdgeIdentifier"
12 fogPlacement: "emufog.placement.DefaultFogLayout"
13 exporter: "emufog.export.ContainernetExporter"
14 reader: "emufog.reader.BriteReader"
```

**Basic Settings** This part of the settings file defines the basic experiment settings. The number of Fog Nodes and the cost heuristics are defined.

**Listing A.2** Basic Settings

```
15 ######################################################################
16                           Basic Settings
17 ######################################################################
18 baseAddress: "10.0.0.0"
19 maxFogNodes: 20
20 costThreshold: 2
21 delayBoundary: 20
22
23 # Settings for Parallel building:
24 threadCount: 1
25 parallelFogBuilding: false
```

**Device Node Types** Here, one or more Device Node Types can be specified. The specified types are then available to the *devicePlacement* algorithm.

**Listing A.3** Specification of Device Node Types

```
26 ######################################################################
27                           DeviceNodeTypes
28 ######################################################################
29 deviceNodeTypes:
30     - name: deviceNode1
31       scalingFactor: 1
32       averageDeviceCount: 1
33       memoryLimit: "'256mb'"
34       cpuShare: 1
35       nodeLatency: 10
36       nodeBandwidth: 5
```

**Fog Node Types** Here, one or more Fog Node Types can be specified. The specified types are then available to the *fogPlacement* algorithm.

**Listing A.4** Specification of Fog Node Types

```
##################################################################
                          FogNodeTypes
##################################################################
fogNodeTypes:
  - name: fogNodeType1
    id: 1
    maximumConnections: 1
    costs: 1
    memoryLimit: "'4gb'"
    cpuShare: 1
    nodeLatency: 0
    nodeBandwidth: 100
```

**Applications** This section defines all available applications. Currently, there are two different application types. First, *fogApplications* that can be placed on Fog Nodes. And second, one ore more *deviceApplications* can be defined. Specified applications are available to the *applicationAssignmentPolicy* algorithm.

**Listing A.5** Specification of fog and device applications

```
##################################################################
                          Applications
##################################################################
fogApplications:
  - name: serviceA
    container:
      image: "ubuntu:trusty"
      ports:
         -"8080"
      portBindings:
         -"80:8080"
      labels:
         -"'com.emufog'"
      environment:
         -"'FOO=Bar'"
      commands:
         -"ls"
      volumes:
         -"'settings.yaml:/home/settings.yaml'"
deviceApplications:
  - name: serviceA
    container:
      image: alpine
```

## Run EmuFog

There are two ways to run EmuFog. One can either directly execute the compiled application as shown in Listing A.6 or use the provided Container and mount all neccessary files as shown in Listing A.7.

**Listing A.6** An exemplary launch of EmuFog

```
$ java -jar emufog.jar -s settings.yaml
```

**Listing A.7** Run EmuFog in a Container

```
version: "2.1"
services:
  emufog:
    image: emufog:latest
    container_name: emufog
    volumes:
      - /emufog/emufogDocker/settings.yaml:/usr/app/emufog/settings.yaml:rw
      - /emufog/emufogDocker/5er.brite:/usr/app/emufog/5er.brite:rw
      - /emufog/emufogDocker/out:/usr/app/emufog/out:rw
    command: java -jar emufog.jar -s settings.yaml
```

## Output Files

**Containernet Experiment** The generated output file is executable in Containernet. The implementation is done in a way that one has to start the desired SDN Controller by hand. In order to support Networks with loops usage of the Ryu SDN Framework[1] is recommended. The framework already provides a controller implementation with Spanning–Tree[98] capabilities.

To Start the Ryu controller execute command depicted in Listing A.8

**Listing A.8** Start the Ryu controller

```
$ ryu-manager ryu.app.simple_switch_stp_13
```

## Build Automation

With future extensions and collaborative development in mind groundwork for build automation was introduced in this work. In order to ensure that there cannot be broken code committed to the repository *Travis CI*[2] is used as a *Continuous Integration*[DMG07] solution. Before, adding new code to the repository each commit will be compiled by Travis to prevent broken code in the repository.

[1] Ryu Framework https://osrg.github.io/ryu/
[2] Travis CI https://travis-ci.org/

# Bibliography


[98] "IEEE Standard for Information Technology- Telecommunications and Information Exchange Between Systems- Local and Metropolitan Area Networks-Common Specifications Part 3: Media Access Control (MAC) Bridges." In: *ANSI/IEEE Std 802.1D, 1998 Edition* (1998), pp. i–355. DOI: 10.1109/IEEESTD.1998.95619 (cit. on p. 63).

[BMZA12] F. Bonomi, R. Milito, J. Zhu, S. Addepalli. "Fog computing and its role in the internet of things." In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM. 2012, pp. 13–16 (cit. on pp. 15, 19).

[Con+] O. Consortium et al. *Architecture Working Group,"Open-Fog Architecture Overview," Feb 2016* (cit. on pp. 19, 21).

[CST+10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears. "Benchmarking cloud serving systems with YCSB." In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 143–154 (cit. on p. 49).

[Dij59] E. W. Dijkstra. "A note on two problems in connexion with graphs." In: *Numerische mathematik* 1.1 (1959), pp. 269–271 (cit. on p. 39).

[DLYE] B. Di Martino, K.-C. Li, L. T. Yang, A. Esposito. "Internet of Everything." In: () (cit. on p. 15).

[DMG07] P. M. Duvall, S. Matyas, A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007 (cit. on p. 63).

[Gra17] L. Graser. "Design and implementation of an evaluation testbed for fog computing infrastructure and applications." MA thesis. 2017 (cit. on pp. 16, 24, 27, 33).

[GVGB17] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, R. Buyya. "iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments." In: *Software: Practice and Experience* 47.9 (2017), pp. 1275–1296 (cit. on pp. 16, 25).

[Ind16] C. G. C. Index. "Forecast and methodology, 2015-2020 white paper." In: *Retrieved 1st June* (2016) (cit. on p. 15).

[KKV+17] A. Kapsalis, P. Kasnesis, I. S. Venieris, D. I. Kaklamani, C. Z. Patrikakis. "A cooperative fog approach for effective workload balancing." In: *IEEE Cloud Computing* 4.2 (2017), pp. 36–45 (cit. on p. 20).

[LHM10] B. Lantz, B. Heller, N. McKeown. “A Network in a Laptop: Rapid Prototyping for Software-defined Networks.” In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. Monterey, California: ACM, 2010, 19:1–19:6. ISBN: 978-1-4503-0409-2. DOI: 10.1145/1868447.1868466. URL: http://doi.acm.org/10.1145/1868447.1868466 (cit. on p. 22).

[LM10] A. Lakshman, P. Malik. “Cassandra: a decentralized structured storage system.” In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40 (cit. on p. 49).

[LPD12] E. Lochin, T. Pérennou, L. Dairaine. “When should I use network emulation?” In: *annals of telecommunications - annales des télécommunications* 67.5 (June 2012), pp. 247–255. ISSN: 1958-9395. DOI: 10.1007/s12243-011-0268-5. URL: https://doi.org/10.1007/s12243-011-0268-5 (cit. on p. 22).

[LYKZ10] A. Li, X. Yang, S. Kandula, M. Zhang. “CloudCmp: comparing public cloud providers.” In: *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM. 2010, pp. 1–14 (cit. on p. 15).

[MGG+17] R. Mayer, L. Graser, H. Gupta, E. Saurez, U. Ramachandran. “EmuFog: Extensible and Scalable Emulation of Large-Scale Fog Computing Infrastructures.” In: *arXiv preprint arXiv:1709.07563* (2017) (cit. on pp. 16, 24).

[MGSR17a] R. Mayer, H. Gupta, E. Saurez, U. Ramachandran. “FogStore: Toward a Distributed Data Store for Fog Computing.” In: *arXiv preprint arXiv:1709.07558* (2017) (cit. on pp. 25, 49).

[MGSR17b] R. Mayer, H. Gupta, E. Saurez, U. Ramachandran. “The fog makes sense: Enabling social sensing services with limited internet connectivity.” In: *Proceedings of the 2nd International Workshop on Social Sensing*. ACM. 2017, pp. 61–66 (cit. on p. 21).

[MLMB01] A. Medina, A. Lakhina, I. Matta, J. Byers. “BRITE: an approach to universal topology generation.” In: *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 2001, pp. 346–353. DOI: 10.1109/MASCOT.2001.948886 (cit. on p. 26).

[PKR16] M. Peuster, H. Karl, S. van Rossem. “MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments.” In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. Nov. 2016, pp. 148–153. DOI: 10.1109/NFV-SDN.2016.7919490 (cit. on p. 23).

[Sat17] M. Satyanarayanan. “The Emergence of Edge Computing.” In: *Computer* 50.1 (Jan. 2017), pp. 30–39. ISSN: 0018-9162. DOI: 10.1109/MC.2017.9 (cit. on p. 15).

[SPF+07] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, L. Peterson. “Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors.” In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 3. ACM. 2007, pp. 275–287 (cit. on p. 26).

[VR14] L. M. Vaquero, L. Rodero-Merino. “Finding your way in the fog: Towards a comprehensive definition of fog computing.” In: *ACM SIGCOMM Computer Communication Review* 44.5 (2014), pp. 27–32 (cit. on p. 19).

[VS17] P. Varshney, Y. Simmhan. “Demystifying fog computing: Characterizing architectures, applications and abstractions.” In: *Fog and Edge Computing (ICFEC), 2017 IEEE 1st International Conference on*. IEEE. 2017, pp. 115–124 (cit. on p. 19).

[XMR16] Y. Xu, V. Mahendran, S. Radhakrishnan. “Towards SDN-based fog computing: MQTT broker virtualization for effective and reliable delivery.” In: *Communication Systems and Networks (COMSNETS), 2016 8th International Conference on*. IEEE. 2016, pp. 1–6 (cit. on p. 21).

All links were last followed on May 11, 2018.

**Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature