TU WIEN Informatics

Self-adaptive Distributed MQTT Middleware for Edge Computing Applications

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Andreas Bruckner, BSc
Matrikelnummer 00929234

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag. Dr. Schahram Dustdar
Mitwirkung: Dipl.-Ing. Dr. Thomas Rausch, BSc

Wien, 5. Oktober 2022

Andreas Bruckner	Schahram Dustdar

Informatics

Self-adaptive Distributed MQTT Middleware for Edge Computing Applications

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Andreas Bruckner, BSc
Registration Number 00929234

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag. Dr. Schahram Dustdar
Assistance: Dipl.-Ing. Dr. Thomas Rausch, BSc

Vienna, 5th October, 2022

Andreas Bruckner	Schahram Dustdar

Erklärung zur Verfassung der Arbeit

Andreas Bruckner, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 5. Oktober 2022

Andreas Bruckner

Acknowledgements

First of all, I want to thank my advisors, especially Thomas, who supported me throughout my work on this thesis, and gave me valuable input and feedback and the encouraging nudges that kept me going. I am also thankful for being given the opportunity to support the distributed systems group as a tutor, which was not only a great and fun experience but also taught me a lot.

Furthermore, I would like to thank my family, partner and friends, for always being supportive and helpful. They have endured my complaining about basically everything, some of them for over 10 years! Special thanks to Felix for giving me additional feedback and input for my thesis.

Finally, I would like to thank Rainer for providing me a flexible work environment. This gave me all the freedom that I needed to finish my studies and gain practical experience at the same time.

Kurzfassung

In den letzten Jahren hat sich Edge Computing zu einem neuen Computing-Paradigma entwickelt, das Computing-Ressourcen in der Nähe von Clients bereitstellt und Anwendungen ermöglicht, die niedrige Latenz erfordern, beispielsweise Industrial IoT oder Urban Augmented Reality. Viele dieser Anwendungen verwenden ein Publish-Subscribe-Kommunikationsmuster, das Nachrichtenbroker für die Datenverbreitung verwenden. Derzeit unterstützen die meisten Nachrichtenbroker hauptsächlich Cloud Computing. Die potenziell langen Distanzen zwischen Clients und Cloud-Brokern führen jedoch zu Latenzen, die einige Anwendungen möglicherweise nicht tolerieren. Edge-Computing-Ressourcen sind eine Möglichkeit, um Broker in der Nähe der Clients bereitzustellen und so die Latenz zu reduzieren. Wir haben ein verteiltes Brokersystem entwickelt, das Edge-Ressourcen nutzt, um den Abstand zu Clients und somit die Latenz zu reduzieren. Unser Ansatz besteht aus der Positionsbestimmung mithilfe eines Netzwerkkoordinatensystems in Kombination mit einer automatisierten Broker-Orchestrierung. Die Verwendung von Netzwerkkoordinaten ermöglicht es uns, Entfernungen zu schätzen, anstatt sie messen zu müssen, wodurch der Überwachungsaufwand und die Netzwerkbelastung reduziert werden. Die Broker-Orchestrierung startet und stoppt Broker-Instanzen auf Edge-Ressourcen nach Bedarf, wobei Druck auf Edge-Knoten durch nahe Clients ausgeübt wird, um zu entscheiden, ob ein Broker gestartet werden soll oder nicht. Wir evaluieren unser System in einer simulierten Umgebung mit Szenarien und Topologien, die für IoT-Anwendungen repräsentativ sind. Die Ergebnisse zeigen, dass wir ein Netzwerkkoordinatensystem verwenden können, um den nächstgelegenen Broker für einen Client zu finden und gleichzeitig die Anzahl der Distanzmessungen um mindestens eine Größenordnung zu reduzieren. Wir zeigen auch, dass der Mechanismus zur automatischen Skalierung die Broker dynamisch auf den Edge-Ressourcen in der Nähe der Clients nach Bedarf bereitstellen kann und dabei den Kompromiss zwischen Latenz und Ressourcenverbrauch berücksichtigt.

Abstract

In recent years, Edge Computing has emerged as a new computing paradigm that leverages computing resources close to clients, paving the way for latency-sensitive applications like industrial IoT or urban augmented reality. Many of these applications use publish-subscribe patterns for communications that use message brokers for data dissemination. Currently, most message brokers focus on deployments in the cloud. However, the connections between clients and cloud brokers introduce latencies that some applications might not tolerate. Edge computing resources offer a possible location to deploy brokers closer to clients, reducing latency. We designed a distributed broker system that makes use of edge resources to improve Quality of Service. Our approach consists of proximity detection by using a network coordinate system, along with self-adaptive broker orchestration. Using network coordinates allows us to estimate distances instead of having to measure them, reducing monitoring overhead and network strain. The broker orchestration starts and stops broker instances on edge resources on demand, using local pressure on edge nodes by close clients to decide whether to start a broker or not. We evaluate our system in a simulated environment with scenarios and topologies that are representative for IoT applications. Results show that we can use a network coordinate system to accurately find the closest broker for a client, and at the same time reduce the number distance measurements by at least one order of magnitude. We also show that the broker scaling mechanism dynamically deploys brokers on the edge resources close to the clients, while taking the tradeoff between end-to-end latency and resource consumption into account.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In recent years, Internet of Things (IoT) applications have become an important research topic in the field of distributed systems. Examples for such applications are:

- A smart city application that connects smart traffic lights, cars and pedestrians and warns traffic participants about certain dangers. For instance, a car that is about to turn might not see pedestrians or cyclists around the corner because a bus at a bus stop blocks the view. Such an application could detect pedestrians approaching a crosswalk and warn the car driver [LMP+21].
- Healthcare applications to support home care that use IoT devices to check patient's vital parameters. The data would help healthcare providers to monitor their patients remotely [DBBR19, SLSB19].
- Factories that use IoT to connect the factory's devices and machines [QCZ+20]. Data could be processed both on-premise and in the cloud. Factory workers using augmented reality glasses to display information about machines have low-latency requirements for the application to work properly.

We identify a couple of challenges in such IoT applications that use publish/subscribe messaging middleware:

- Applications like the first example above, where traffic participants get warned about possible dangers, come with stringent latency requirements [BMZA12]. Furthermore, industrial IoT applications might have soft real-time constraints. As a consequence, proximity plays an important role, i.e., clients should use message brokers that are as close as possible.

- Clients might be mobile and change their location within the network frequently, for instance, a smart city application which involves moving vehicles that interact with smart traffic lights [CSS+16]. Moving clients should be reconnected as soon as another message broker is closer.

- In IoT applications, devices are often resource-constrained. Also, especially embedded devices often only support standardized protocols like MQTT [Nai17]. This limits possible adaptations of client applications that might be required to implement mechanisms to solve the before-mentioned challenges.

To address these challenges, IoT applications often use publish/subscribe communication patterns for device-to-device communications, using message brokers to route messages [EFGK03, VS17]. Using this messaging pattern, applications can achieve asynchronous communications, which is ideal for loosely-coupled IoT environments. Client nodes publish data and computing nodes subscribe to the relevant topics, while message brokers route messages between publishers and subscribers. IoT applications usually rely on standardized messaging protocols such as MQTT, a lightweight protocol which is well suited for resource constrained devices [YSAAH17].

1.2 Problem Statement

In IoT applications that rely on cloud brokers, the link between client and cloud can introduce issues regarding latency, scalability or availability [ZMK+15]. With Dynamoth [GSGKK15] and MultiPub [GSKK17], there are existing approaches to scale broker networks that focus on cloud resources. Even if there is always a broker in the closet cloud region available, the latency introduced by the link between client and cloud might be too high.

As a consequence, a growing number of IoT applications adopt the edge computing paradigm [BMZA12, SCZ+16, Sat17, QCZ+20]. Moving parts of the application from the cloud to the edge decentralizes the application and brings the service endpoints closer to the clients.

In a messaging middleware, we can use edge computing to reduce the mean distance between client gateways and brokers. EMMA is an existing broker network [RND18] of cloud and edge brokers. Clients connect to brokers via gateways that proxy MQTT messages. A coordinator controls gateways and requests them to connect to a certain broker. These decisions are based on knowledge about distances between clients and brokers. Its monitoring approach involves regular distance measurements between all nodes. However, this is not feasible for a large number of clients and brokers.

Other approaches include FogMq [AH16], which introduces a new protocol and does not use MQTT, or SDN-based approaches [ZJ13, BWVK18], which require specific network infrastructure.

Furthermore, it is not trivial to find the optimal location for new broker deployments. Clients appear and disappear around certain edge resources. Therefore, we need a mechanism that dynamically scales the broker network up and down on demand. None of the existing approaches allows us to deploy to specific edge resources depending on the current demand by clients nearby.

1.3 Research Questions

To drive our system design and evaluation, we posit the following research questions:

RQ1. With a network coordinate system, what is the tradeoff between monitoring overhead and accuracy when finding the closest broker for a client? In order to find the closest broker using a trivial approach, the clients would have to measure distances to all available brokers. This creates a management overhead that becomes impractical with a growing number of brokers [RND18]. To be able to scale the broker network, we have to find a mechanism that reduces the number of measurements. Network coordinate systems seem to be a promising candidate to replace trivial measurements. Using coordinates introduce inaccuracy, which might be negligible for finding the closest broker.

RQ2. What is the tradeoff between end-to-end latency and resource consumption when scaling message brokers to edge resources using self-adaptive broker scaling? If we only optimized for end-to-end latency, we could deploy a broker on every edge resource, which would lead to an enormous resource consumption. Therefore, we have to consider the tradeoff between end-to-end latency and resource consumption when scaling broker networks to the edge. When scaling, we have to be able to define a parameter that determines, how eagerly the mechanism deploys new brokers.

RQ3. Which metrics and mechanisms are appropriate for reactive autoscaling decisions on distributed broker networks? In a distributed broker network, the autoscaling decision is not trivial due to many participants. We have to find a mechanism that takes the impact of scaling decisions on all relevant nodes into account. Furthermore, this mechanism has to use appropriate metrics to reach the desired goal.

RQ4. How well does the self-adaptive mechanism cope with different publish/-subscribe topologies of IoT applications? The scaling mechanism might perform differently depending on the publish/subscribe topology of the application. For instance, publishers and subscribers that concentrate close to an edge resource will benefit from an additional broker on this resource. However, if subscribers distribute across multiple regions, the performance improvement of a local edge broker is smaller.

1.4 Solution Approach

Our solution builds on EMMA [RND18], a distributed, latency-aware broker network. We try to solve the problem of broker discovery by using a network coordinate system. By calculating network coordinates, we can estimate distances between any two nodes we know the coordinates of. This way, we need fewer direct network measurements, leading to less management overhead and network strain. Furthermore, EMMA lacks the ability to dynamically deploy edge brokers. We propose a mechanism that adapts to client presence at edges, making the broker network dynamic. While cloud brokers are always running, edge brokers can be started or stopped on demand.

In order to reduce end-to-end latency, we use edge computing principles, and deploy message brokers closer to the clients. This reduces the latency in case of on-premises device-to-device message relaying. By using both edge and cloud brokers, we can design a dynamic message broker network that allows us to deliver messages from local publishers to local subscribers with the least possible delay. At the same time, cloud brokers can relay messages to computing resources located in the cloud.

To evaluate our concept, we use a discrete event simulation that runs the same scenario as in the evaluation of the EMMA paper. This enables us to compare our monitoring system with EMMA's approach. It will show whether distance estimations suffice in order to make the same coordinator decisions as with measured distances. A second scenario simulates an Industrial IoT application and will help us evaluate the dynamic broker provisioning.

1.5 Structure

The rest of this work is organized as follows. Chapter 2 will give a brief introduction in fundamental research our work bases on, chapter 3 lists related work. Chapter 4 provides a high-level overview of our system's components and their interactions. Chapter 5 explains how we monitor distances between nodes and optimize client–broker connections to improve Quality of Service. In chapter 6, we propose a mechanism to deploy edge brokers dynamically based on local demand. In chapter 7, we will explain how we evaluated our approach and present the results, chapter 8 will discuss the results. We conclude our work in chapter 9, including remarks about possible future work.

CHAPTER 2

Background

In this chapter, we will introduce fundamentals about the technologies and paradigms we use to build our system. We will also elaborate about why we chose these technologies in order to solve the problems.

2.1 Internet of Things

In the last decade, Internet of Things (IoT) emerged as an important research topic [Sta14, XHL14, Rag15]. While there is still no exact definition of IoT, IoT refers to *things* integrated into Internet applications. Things can be mobile or sensor devices which are connected to the Internet or objects that can be identified by radio-frequency identification (RFID). Devices can connect to the Internet by different means: Either directly via network connections, or via gateway devices using near field communication technologies like Bluetooth.

Li et al. [LXZ15] describe IoT as a service-oriented architecture that consists of four layers, one of them being the network layer. This layer connects all network-enabled devices and this is what our work focuses on. From our point of view, it is not important which technology the things actually use and if a gateway device is involved or not. In the end, some device has an Internet connection and can use a messaging protocol for communication.

In designing our architecture, we have to consider a couple of properties of Internet of Things applications:

Mobility of Clients

Clients might not stay at the same location and instead move around. As a consequence, their network location changes over time. For instance, a car that communicates with other smart city devices will change its location frequently.

Constrained Resources
Things can be anything from a small microcontroller like sensor devices to a vehicle with a full-blown computer. Depending on the resources available, there might be restrictions on possible applications that can run on these devices. We have to consider resource consumption when designing our system.

Poor Connectivity
Mobile clients that connect using a wireless network might not have a stable connection to the Internet. This can lead to connectivity outages, leading to messages getting lost if we do not build our system such that it is resilient against client connection issues.

2.2 Publish/Subscribe Messaging

Following the properties outlined above, one important aspect of IoT applications is that we can not rely on stable connections between nodes. Publish/subscribe (or pub/sub) is a messaging pattern that allows to decouple communication between nodes of a distributed system. There are different kinds of client components: publishers, subscribers, and brokers. Publishers produce data and *publish* it by sending it to the broker, who will forward it to relevant subscribers. Subscribers are interested in certain categories of messages and *subscribe* to them. The third kind of component, which is called broker or dispatcher, is responsible for event dissemination. The broker knows the subscriptions and can forward published data on a topic to the subscribed clients, i.e., *notify* them about new data. Figure 2.1 shows these relations.

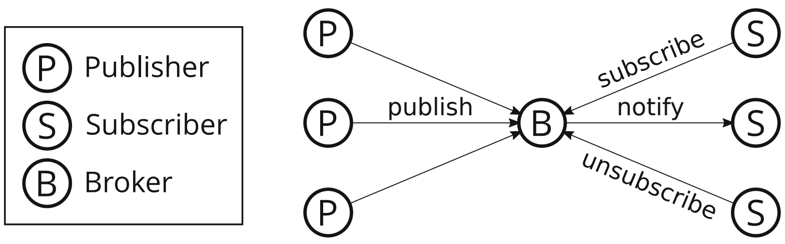


Figure 2.1: Components of a pub/sub messaging system and their relations.

2.2.1 Subscription Schemes

Subscribers need the ability to express in which particular events they are interested. This is possible in different ways:

Topic-based scheme allows subscribers to subscribe to certain topics. Publishers send each event to a single topic at a time.

Content-based scheme matches the event content or specific properties. For instance, these properties can be passed via event headers.

Type-based scheme can be used in typed languages to tightly integrate into their type system. This way, subscribers can subscribe to types of events.

The systems we build on use topic-based publish/subscribe because its simplicity makes it ideal for the use in IoT applications.

2.2.2 Decoupling

Due to their communication pattern, pub/sub messaging systems can achieve decoupling in three dimensions [EFGK03].

Time decoupling
Publishing data and notifying subscribers does not have to happen at the same time (time decoupling). For instance, the subscriber might not be connected to the broker at the time of a message being published. After some time, when the subscriber comes back online, the broker can finally send the notification about the new message.

Space decoupling
Publishers and subscribers not need to know each other directly; instead, they only have to communicate with the broker. This is a key feature of pub/sub messaging and enables communicating components to be completely unaware of each other. The only thing that components have in common is a broker (or a broker network) and the topic they publish or subscribe to.

Synchronization decoupling
Publishers do not necessarily have to wait until all subscribers have received the message. For example, a sensor that publishes measurement values does not have to wait for all subscribers.

In distributed applications, nodes might need to communicate asynchronously and decoupled. For instance, an application might include additional components that do monitoring or data aggregation. By using a publish/subscribe messaging middleware, nodes that publish information do not have to know the exact recipients. The decoupling nature makes publish/subscribe messaging an important enabling technology for the Internet of Things.

2.2.3 Quality of Service

When speaking about Quality of Service (QoS) in publish/subscribe messaging middleware, we can distinguish several classes of service guarantees [MMKB07]:

Latency the time from a publisher to send a message to a subscriber to receive it. We can calculate the latency by summarizing the forwarding delay of all links and nodes on the path from publisher to subscriber.

Bandwidth the number of messages that can pass the path between publisher and subscriber per time unit. The bandwidth largely depends on the bandwidth of network links, but also on available CPU time that might limit the bandwidth of a broker.

Reliability the ratio of messages that a subscriber to a topic receives to the number of messages published to that topic

Delivery semantics define the guarantee about message delivery to subscribers per message. The messaging middleware can give no guarantee at all (best effort), at least once (but possibly duplicated), at most once or exactly once.

Message Ordering the middleware can guarantee the order of message delivery, e.g., random, fifo (first in, first out), priority, etc.

In the edge computing applications we are studying, the main objective is to reduce latency between clients and service endpoints. Therefore, we focus on latency as the Quality of Service class in our work.

2.3 MQTT

MQTT[1] is a lightweight publish/subscribe messaging protocol. It was designed for applications that run on devices with limited resources. In our work, we focus on MQTT because it is widely adopted in IoT applications and machine-to-machine communications [Nai17].

2.3.1 Protocol

MQTT uses TCP/IP as underlying transport protocol. To keep the protocol as simple as possible, there is only a small set of packet types: Basic connection management, a simple authentication mechanism, and publish and subscribe packets. The header of MQTT packets is designed to be as short as possible. Furthermore, depending on the packet type, the header contains a variable length header. This protocol design saves overhead and enables efficient message processing on resource-constrained devices. For example, a publish packet consists of a fixed-length header of two bytes, followed by variable length fields for the topic name, and the payload.

2.3.2 Distributed Brokers

In MQTT, each client connects to exactly one broker, making it a centralized protocol. There is no concept of distributed brokers in the protocol specification. This limits the applications that build purely on MQTT because there is no mechanism that allows clients to choose from a set of brokers. However, there are existing approaches that try to implement distributed MQTT brokers. There are different methods to achieve this:

[1] `https://mqtt.org`

Clustering

In a broker cluster, a central component (a load balancer) serves as an entry point for client connections. Behind the load balancer, multiple broker instances serve the requests. Brokers and load balancer use an additional control protocol to handle messages, exchange details about connected clients, cluster members and more, depending on the cluster features. For each message, the load balancer decides which broker (inbound) or client (outbound) it forwards the message to. Examples for a clustered broker network are Nucleus [SB18] or close systems such as Amazon's AWS IoT Core[2] or HiveMQ[3].

Bridging

This mechanism uses MQTT itself to forward messages between brokers. On a published message, the broker who receives the message not only forwards the message to the clients but also to other brokers. To that end, brokers have to know which other brokers have clients that subscribed to a certain topic, e.g., by maintaining a bridging table. Additionally, brokers have to form an overlay network to communicate with each other. Existing broker networks that support bridging are HiveMQ or EMMA [RND18], which also supports dynamic reconfiguring of the network.

Evaluations have shown that clustering is a more resource-consuming mechanism, making bridging more suitable for edge computing applications with resource-constrained devices [LRCM22]. The required control protocol to manage the broker network could be implemented in-band, i.e., by using special topics, or out-band by implementing a custom protocol. For this reason and because our system builds on the architecture of EMMA, we will use bridging between brokers in our implementation.

Both methods have in common that the client has to connect to a single entry point of the broker network. Scaling broker networks from the cloud to the edge is not trivial because clients have to know which broker they should connect to. In our work, we will solve this problem by introducing a mechanism that allows clients to find their optimal broker.

2.3.3 Quality of Service

In context of MQTT, Quality of Service (QoS) denotes message delivery guarantees. MQTT offers different levels of QoS.

Level 0

On the lowest level, there is no guarantee that a published message arrives at the subscribers at all. This means that the message will be delivered at most once.

[2]`https://aws.amazon.com/iot-core/`

[3]`https://www.hivemq.com`

Level 1
By introducing an acknowledgement message for *publish* messages, level 1 guarantees at least one delivery of the message. The message could also be delivered more than once in cases where the sender does not receive the acknowledgement and re-sends the message.

Level 2
For exactly-once message delivery in QoS level 2, MQTT specifies a two-phase commit protocol. When the broker receives a *publish* message, it will acknowledge to the client that it has received the packet. However, the broker will not yet process the package (i.e., forward to subscribers). After the client receives the acknowledgement from the broker, it will send a *publish* release packet. This indicates that the broker can proceed and forward the message to the subscribers. After all recipients received the package, the broker sends a *publish complete* packet back to the sender. This way, QoS level 2 guarantees exactly-once delivery.

With each level, the amount of packages required to publish a message doubles. Furthermore, from the sender's point of view, level 2 publish transactions can take a long time. This is because the broker has to wait for all recipients to complete the two-phase commit protocol. For this reason, QoS level 2 implicitly enforces synchronization coupling. Our system design supports all QoS levels. In our evaluation, we focus on QoS level 0 because we want to evaluate latency and resource consumption and not the reliability of message delivery.

2.4 Edge Computing

Cloud computing has become an important computing paradigm for all kinds of applications, including IoT. It abstracts away from the infrastructure details and allows developers to focus on the implementation of the application itself. Furthermore, it eases horizontal scaling of the application both within the same geographical cloud region and across regions.

With the emergence of IoT, applications produce ever more data at the edge of the network, i.e., where clients reside, which introduces a lot of traffic on Internet links. For instance, Internet of Vehicles applications require low-latency content delivery and computation services [ZL20]. These requirements can not be guaranteed when using cloud computing resources. Edge computing [SCZ+16, Sat17], sometimes called fog computing [BMZA12], tries to move computing resources from the cloud to the edge and hence closer to the client.

Cloudlets [VSDTD12] are one possible infrastructure model of edge computing. By moving certain application components from the cloud to a computing environment on the edge, cloudlets offer flexible computing resources close to the clients. For example, smart city applications could rely on cloudlets deployed in existing urban infrastructure [GSK+18].

2.5 EMMA

Most existing brokers are centralized and designed for cloud deployment, for instance HiveMQ[4]. As mentioned above, IoT applications might require low latency between publishers and subscribers. In this case, deploying brokers on the edge can make one order of magnitude of difference compared to a broker in the cloud (e.g., 2 ms versus 20 ms). Most existing broker networks do not take the location of clients into account. As a consequence, it is hard to dynamically scale the broker network to edge resources.

EMMA [RND18] is a messaging middleware that uses a network of cloud and edge brokers and connects the clients to the best brokers. By maintaining an overview of the client's locations, EMMA can optimize the broker network dynamically. New brokers can join or existing brokers can be removed from the network, and existing client connections will be reconnected to the optimal broker. We build our system on EMMA because it already addresses the issue of finding the optimal broker for clients in a broker network that spans across cloud and edge resources.

2.5.1 Broker network

The brokers use bridging to forward messages published on a certain topic to other brokers with subscribers that are interested in this topic. A central coordinator monitors the network state and reconfigures the network if it is not optimal w.r.t. some metric, e.g., end-to-end latency.

To advise clients to connect to a specific broker, EMMA introduces gateways that act as a proxy between clients and brokers. The gateway implements a control protocol that enables the coordinator to direct the gateway to the optimal broker. This way, clients do not have to implement the control protocol while the gateway hides the broker network from the client. From a client perspective, it seems as if it is always connected to the same broker.

On the broker side, EMMA uses an MQTT broker which is extended by the control protocol. Furthermore, the brokers have to maintain a common bridge table to forward messages to all brokers with subscribers on a certain topic.

2.5.2 Monitoring

To get an overview of the distances in the network, the controller can request gateways to perform measurements to certain brokers. On such a request, gateways send ping requests to brokers, which send a ping response to the gateway. The controller receives the measurement results from the gateways and updates its topology model.

A complete overview of all distances requires a fully connected graph between all clients and brokers. This results in a high monitoring overhead. To lower this overhead, we

[4]`https://hivemq.com`

propose an extension to EMMA that uses a network coordinate system to estimate distances, see chapter 5.

2.5.3 Scaling

EMMA can scale the broker network by automatically integrating new brokers into the network, but it can not reconfigure the network automatically. The concept of osmotic computing [VFD+16] dynamically moves workloads from cloud to edge computing resources depending on the location of clients. In our work, we introduce a self-adaptive network configuration mechanism that uses an osmotic computing model to scale the broker network to edge resources on demand, see chapter 6.

2.6 Network Coordinate Systems

The need for distance estimation has emerged with new Internet technologies that created overlay networks. An overlay network might be a peer-to-peer network, e.g., file sharing networks. But also, other applications forming overlay networks like content distribution networks need to know the distance between two arbitrary nodes.

An early approach to estimate distances was IDMaps, a platform to perform measurements and provide data as a distance map [FJPa99]. A network of tracers monitors distances between them and distribute the gathered data over multicast. Each of the tracers is responsible for certain IP address prefixes. If a client wants to estimate the distance to another node, it requests the distance from its associated tracer to the tracer close to the destination.

The next evolutionary step was Global Network Positioning [NZ02], a mechanism that is able to compute coordinates in a vector space that approximates the actual distances between Internet nodes. The architecture of GNP uses peer-to-peer communication patterns. This makes GNP more scalable than IDMaps or similar approaches. However, GNP still relies on a set of known hosts called landmarks that provide reference coordinates for other hosts' orientation.

2.6.1 Vivaldi

To remove the need for special infrastructure nodes like tracers or landmarks, Vivaldi [DCKM04] uses a different approach. The algorithm uses only peer-to-peer communication and does not need any coordination. This makes the algorithm suitable for dynamic overlay networks with ephemeral or mobile members.

We use Vivaldi to calculate network coordinates of client and broker nodes. This reduces the monitoring overhead that EMMA adds to the system, because by using network coordinates, we do not have to measure distances between all nodes but can estimate distances by using coordinates.

Vivaldi uses a coordinate model that maps the network nodes to a multidimensional vector space. This is an approximation that contains an error because latencies between Internet nodes violate the triangle inequality. The error can be calculated as:

$$E = \sum_i \sum_j (L_{ij} - \|x_i - x_j\|)^2$$

where $\|x_i - x_j\|$ denotes the distance between the coordinates of nodes i and j. Vivaldi tries to minimize this error by adapting the spring forces according to observed distances from the network. Between each two nodes there is a spring with a rest length of the known round-trip time (RTT) L_{ij}. First, we will explain how the algorithm works in a centralized manner, i.e., running on a single node. Then, we will describe the distributed algorithm that runs on each node individually.

Centralized Algorithm

We explain the basics of the algorithm by assuming that a single node has knowledge of all measured distances between all nodes (i.e., a latency matrix). Each connection between nodes is modeled by a spring with a rest length of the known RTT. The current spring length resembles the distance in coordinate space, hence the error function E is equivalent to the spring force. By moving the coordinates such that the spring force is minimal, the system will eventually come to a rest. In the equilibrium state, all errors are minimized and so is the error of the approximation.

Formally, the force of a spring is

$$\vec{F}_{ij} = (L_{ij} - \|\vec{x}_i - \vec{x}_j\|) \cdot \mathrm{u}(\vec{x}_i - \vec{x}_j)$$

where the scalar part $(L_{ij} - \|x_i - x_j\|)$ is the error between the observed RTT L_{ij} and the RTT calculated from the current coordinates $\vec{x}_i$ and $\vec{x}_j$. The right part $\mathrm{u}(\vec{x}_i - \vec{x}_j)$ (a unit vector) is the direction between nodes i and j and defines the direction of force exerted on node i. The sum of all forces from other nodes defines the new coordinates. To smoothen the changes, the algorithm simulates the spring movements over time. For this, a time step scales the amount of force exerted at one point in time.

Taking the time step δ into account, the new coordinates after each interval can be calculated by

$$\vec{x}_i = \vec{x}_i + \delta \cdot \vec{F}_i$$

Distributed Algorithm

In the distributed version, each node runs the algorithm on its own and is responsible for maintaining its coordinates. Vivaldi continuously calculates the node coordinates based on its current coordinates and measured RTTs to other nodes. Applications using Vivaldi can use RTT data from metadata of application messages, or perform additional measurements to gather RTT data, or both. With each message, nodes also send their own coordinates, i.e., the recipient learns the tuple $(rtt, \vec{x}_j)$ where rtt is the measured round-trip time and $\vec{x}_j$ are the coordinates of the sender. This tuple is the input for the Vivaldi algorithm, which calculates the new coordinates as follows:

$$\vec{x}_i = \vec{x}_i + \delta \cdot \underbrace{(rtt - \|\vec{x}_i - \vec{x}_j\|)}_{error} \cdot \underbrace{\mathrm{u}(\vec{x}_i - \vec{x}_j)}_{direction}$$

Adaptive Timestep

In the trivial implementation above, the timestep value δ is constant. However, this does not ensure convergence to an equilibrium state: On the one hand, with small timestep values, the algorithm would need a long time to reach equilibrium. On the other hand, if the timestep is too large, coordinates oscillate and never converge.

For this reason, Vivaldi uses local and remote errors to change the timestep value dynamically.

$$\delta = c_c \times \frac{e_{local}}{e_{local} + e_{remote}}$$

where c_c is a constant in order to be able to tweak the influence of the error, and e_{local} and e_{remote} are the local and remote errors, respectively. The local error is a weighted average that is updated at each Vivaldi run.

This results in the following: In the beginning, when errors are high, the timestep value is big, leading to quick movements to the correct coordinates. Later, when errors are getting smaller, so does the timestep value and the coordinates converge to an equilibrium. Also, new nodes that join the network do not have a high impact on the older nodes' coordinates.

Algorithm 2.1 shows the complete Vivaldi algorithm including error calculation. The additional constant c_e enables tweaking of the influence the error has on the coordinates.

Model Selection

Since Vivaldi uses an n-dimensional vector space for coordinates, we have to choose an n such that the average error rate is minimal. The authors of Vivaldi evaluated different values for n and came to the conclusion that more than three dimensions do not significantly lower the error. Generally speaking, network coordinate systems that use an embedding do not improve accuracy with more than six dimensions [LGP$^+$05]. The remaining mapping error results mostly from the fact that access links (the last mile) introduce additional delay which it disproportional to the geographical distance.

Algorithm 2.1: Vivaldi algorithm with adaptive timestep

```
    Data: current ($(\vec{x}_i, e_i)$), received ($(rtt, \vec{x}_j, e_j)$)
    Result: updated ($(\vec{x}_i, e_i)$)
1   Procedure vivaldi($rtt$, $\vec{x}_j$, $e_j$)
2       $w = e_i/(e_i + e_j)$                                         // error weight
3       $\delta = c_c \cdot w$                                         // timestep value
4       $\vec{x}_i = \vec{x}_i + \delta \cdot (rtt - \|\vec{x}_i - \vec{x}_j\|) \cdot \mathrm{u}(\vec{x}_i - \vec{x}_j)$   // update coordinates
5       $e_s = |\|\vec{x}_i - \vec{x}_j\| - rtt|$                      // relative error of sample
6       $e_i = e_s \cdot c_e \cdot w + e_i \cdot (1 - c_e \times w)$    // updated local error
```

There are several possible reasons for that, e.g., queuing delay or low bandwidth. To solve this problem, the authors introduced the concept of a height vector. In addition to the vector, a scalar height value gets added to the coordinates and models the distance of the access link. Using a height value, the coordinates in a 2D vector space are more accurate than in a 3D vector space without height, as shown by the authors [DCKM04]. Ledlie *et al.* [LGS07] showed that using four dimensions plus height, the error can be improved by another 43% compared to two dimensions plus height.

CHAPTER 3

Related Work

In this chapter, we present work that is related to the context of our work. First, we need a broker network that can span across cloud and edge sites to optimize for certain Quality of Service aspects (in our case, responsiveness in terms of latency). Second, to dynamically react to location changes of mobile clients, we need an auto-scaling mechanism that can deploy brokers close to the clients.

3.1 Publish/Subscribe Messaging Systems

In order to exchange data, nodes of distributed IoT applications mainly use a publish/-subscribe communication pattern (see section 2.2). There are two main approaches to perform message dissemination.

3.1.1 Broker Networks

In a broker network, there are central nodes that relay messages between publishers and subscribers. To that end, the brokers have to exchange interests, i.e., which nodes are interested in which events, to be able to route events to the subscriber nodes. Examples for early broker networks for topic-based pub/sub are SIENA [CRW00] and Hermes [PB02]. However, these systems build on legacy technologies and do not meet the requirements of a modern IoT application that relies on cloud and edge computing.

PADRES [JCL$^+$10] is a content-based pub/sub system that uses brokers. Publishers and subscribers describe their advertisements and interests using an SQL-like language called PSQL. Brokers save those PSQL queries in routing tables in order to forward events to brokers with interested subscribers. We build a system for IoT applications that commonly use a topic-based subscription scheme (see subsection 2.2.1) and do not need complex query languages like PSQL.

Dynamoth [GSGKK15] is a topic-based pub/sub system for the cloud. It supports a mechanism to scale the broker network and dynamically distributes channels such that the broker load distributes evenly. Load is defined as the ratio between measured outgoing bandwidth and the theoretical maximum bandwidth of the server. While this metric might be a heuristic for latency, it does not take the actual network distance into account. This differs from our approach where we use round trip time as a metric for overlay topology and scaling decisions.

3.1.2 Overlay Networks

The rising number of Internet nodes led to a different approach that does not rely on brokers, which might become a bottleneck. Instead, nodes directly exchange events by using a peer-to-peer overlay network. There are no central components that route messages between nodes, except for finding other peers. Examples are SpiderCast [CMTV07], PolderCast [SvSVV12] or Vitis [RGPH11]. Usually, the overlay topology structures in a way that it resembles the relations between publishers and subscribers. Algorithms like the one proposed by Onus *et al.* [OR11] optimize the topology such that there is the minimum number of links between nodes. This enables passing events efficiently from publisher to subscribers. While this approach works well for regular Internet nodes, it might not be feasible to build a peer-to-peer network between IoT devices. Also, it is harder to guarantee delivery latency in dynamic peer-to-peer networks. Therefore, this approach is not suitable for our scope of IoT applications.

3.1.3 Network Coordinate Systems

Extensive research about the accuracy of network coordinate systems [LGP+05, NZ02, SPvS04] show its applicability in broker networks. In dynamically scalable broker networks, there has to be a mechanism that places brokers on the optimal nodes. Some existing approaches already use network coordinate systems (e.g., from Garbacki *et al.* [GEvS08]), however, they use peer-to-peer overlay networks. In our system, we want to scale a centralized broker network where clients connect to one single broker instead of multiple peers.

3.1.4 IoT and Edge Computing

The growing interest in IoT pushed the work on messaging systems designed to meet the specific requirements of IoT. Among these requirements is a lightweight protocol to save energy and resources on IoT devices, and low message delivery latency. The previously described work focuses on neither of those requirements. Moving data to the edge requires new broker networks that aso span to the edge [VS17].

Banno *et al.* [BSF+17] proposed a broker network that uses an Interworking Layer of Distributed MQTT brokers (ILDM). The ILDM is a component that resides between clients and edge brokers. Clients do not connect to brokers directly. Instead, they connect to the ILDM, which forwards messages to local and remote brokers. There are different modes of cooperation between ILDM. For example, the ILDM could flood all publications to all other ILDM nodes, which forward them to their attached brokers. The ILDM is similar to the gateway node of EMMA (see section 2.5), with two main differences. First, it includes the logic for flooding messages to other ILDM nodes. Second, it requires both local broker and client, while EMMA only requires a local client and the gateway can dynamically connect to any broker. This requires a higher number of brokers, while our work enables brokers to be deployed dynamically.

MultiPub [GSKK17] provides middleware for cloud brokers. It chooses a single, optimal region that serves a topic, taking latency and cloud costs into account. Clients have to connect to a broker in a region that is designated to handle this topic. MultiPub only considers inter-site latencies, but we also consider latencies between clients and brokers.

FogMq [AH16] uses device clones as gateways that manage connections to other clones. They run on a cloud or edge resource and are the single endpoint for the actual clients. FogMq forms a peer-to-peer overlay between clones. This approach introduces a new protocol and does not rely on pre-existing protocols like MQTT.

Another approach is to use software-defined networking (SDN) to route messages from clients to brokers (Zhang *et al.* [ZJ13], Benson *et al.* [BWVK18]). The mechanisms that build and maintain the network are similar to previously mentioned approaches that form an overlay network between the nodes. In contrast to our work, deploying and using an SDN requires a specialized network infrastructure (e.g., SDN-enabled switches).

Khare *et al.* [KSZ$^+$18a, KSZ$^+$18b] described a data-driven approach to model IoT broker loads. In their work, they formalize an optimization problem to balance the load between cloud and edge brokers based on the topic. Their analysis shows that co-located topics, i.e., multiple topics served by the same broker, impacts the end-to-end latency. To limit this impact, their model introduces a maximum number of co-located topics, resulting in a k-topic co-location problem. This problem is not within the scope of our work. Instead, we focus on scaling the broker network and broker selection for clients.

Hasenburg *et al.* [HSTB20] proposed a solution that forms broadcast groups of brokers. The broadcast groups resemble latency groups, i.e., brokers that are close to each other join the same group. Each broker group has a leader, and each new broker that joins the system is leader of its own group. Leaders monitor distances between each other and join a group if the latency between them is below a configured threshold. Within a group, brokers flood messages to all other brokers (hence the term broadcast group). Leaders also forward messages to a cloud broker that is responsible for intergroup message dissemination. While this approach offers low local latency due to message flooding to close brokers, it does not include a dynamic scaling mechanism. Also, monitoring overhead to measure latency between group leaders might become an issue with a growing number of broadcast groups.

3.2 Auto-scaling Mechanisms

In our system, we want to scale the broker network dynamically based on a Quality of Service metric that reflects the location of clients. This will enable clients to connect to close brokers and reduce message latency. Existing auto-scaling techniques for elastic applications mainly focus on resource consumption on the server itself [LBMAL14]. This does not cover our requirement of considering the Quality of Service in terms of end-to-end latency for message delivery.

Nucleus [SB18] is a solution that provides autoscaling of MQTT brokers for IoT applications. It uses Kubernetes to manage the containers that host the broker instances. However, Nucleus focuses on single-site deployments and focuses on CPU utilization as the metric for scaling decisions. In our work, we use end-to-end latency instead to enable scaling across different edge and cloud sites (see chapter 6).

Another approach by Khare *et al.* [KSZ$^+$18b] uses integer linear programming to find the optimal broker placements. This allows guaranteeing certain latencies per topic, but as the authors proved, solving this problem is NP-hard. Since we do not consider topics when scaling the broker network but only the overall average latency, our solution can use a much simpler, threshold-based mechanism.

Happ and Bayhan [HB20] presented two heuristics for deploying applications and pub/sub brokers on cloud, fog and edge nodes. They take into account the clustering of the users at certain locations. As in our work, their aim is to find a tradeoff between application provider cost and end-to-end delays. The placement decision is based on resource capacities, bandwidth and a cost factor. The authors do not elaborate on how to detect the node locations, which is a central part of our work (see chapter 5).

Based on the principle of elastic computing [DGST11], Villari *et al.* [VFD$^+$16] proposed osmotic computing, a new approach to scaling cloud resources to the edge. The idea is that the presence of clients close to edge resources and a growing demand in terms of certain QoS criteria create osmotic pressure that eventually leads to the deployment of a cloud service on the closer edge resource.

Rausch *et al.* [RDR18] applied this approach to messaging middleware in context of IoT applications. As already established, these applications might depend on certain QoS criteria (e.g., latency). Therefore, the authors propose to use osomotic pressure to deploy additional brokers on edge resources, which will improve QoS for the clients. We base on this concept in our approach to define a scaling mechanism that calculates the osmotic pressure based on distances between network nodes (see chapter 6).

EMMA also contains a monitoring plane that is responsible to keep an overview of network proximity between clients and edge resources. However, they do not elaborate about how this monitoring mechanism can be implemented efficiently. Our approach addresses this problem by using a network coordinate system, reducing the monitoring overhead.

CHAPTER 4

System Overview

In this chapter, we will give an overview of the components and mechanisms involved in our system. Our goal is to create a broker network that solves two problems of distributed MQTT broker networks.

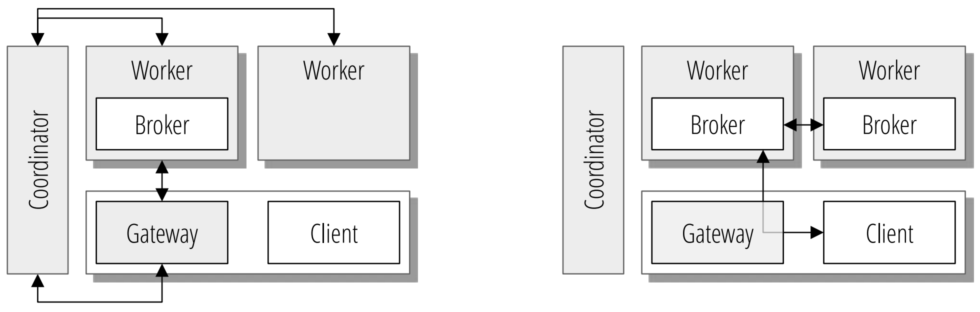
- Clients should be able to find an optimal broker in terms of network distance efficiently (see chapter 5). To that end, we use Vivaldi, a network coordinate system, that allows us to estimate distances between nodes (see subsection 2.6.1). Since clients do not have an overview of the network distances, we use a coordinator component that directs the clients to their optimal broker.
- In IoT applications, clients might roam between different edges of the network. To keep distances between clients and brokers as short as possible, we scale the broker network to edge computing resources. An auto-scaling mechanism monitors the distances between clients and brokers and reconfigures the broker network to optimize the mean latency (see chapter 6).

4.1 Components

Our system consists of four component types that communicate with each other using (a) MQTT and (b) a control protocol. Coordinator and gateway base on the components of EMMA. Additionally, we introduce the worker as a new component.

Broker

A regular publish/subscribe message broker that supports bridging in order to forward messages to other brokers (see subsection 2.3.2).



(a) Control Protocol (b) Regular MQTT messages

Figure 4.1: Components and their interactions

Worker
: A cloud or edge computing node that can host a broker. The worker also runs a daemon that supports the control protocol. This daemon is always active, whether a broker is running or not.

Coordinator
: The central component that monitors the system state and is responsible for auto-scaling brokers. It communicates with workers and gateways using the control protocol.

Client Gateway
: A service that runs close to the client (e.g., on the same machine) that serves as a proxy between client and brokers. The gateway is responsible for connecting to the broker as requested by the coordinator.

Figure 4.1 shows an overview of the components and their interactions. Our middleware provides the grey components, white components are regular MQTT brokers and clients. Following, we will briefly describe the tasks and responsibilities of each component.

4.1.1 Broker

The broker serves as the message broker component for publish/subscribe message dissemination. In order to work in a broker network, the broker has to be able to forward messages to other brokers if another broker has a subscriber to the respective topic. Each broker informs the coordinator about which active client subscriptions it has. The coordinator distributes this information to all other brokers in order to enable proper message routing. As a consequence, the broker not only has to implement a publish/subscribe protocol, but also a control protocol that enables communication with the coordinator.

4.1.2 Worker

A worker is a cloud or edge compute node that has resources to host a broker. Each worker runs a monitoring daemon that measures latencies to other workers and reports them to the coordinator. The daemon is always active to be able to deploy a broker on the worker host. Figure 4.1a shows two possible states of workers: In the left state, the worker is hosting a broker, while in the right state, only the worker daemon is running. Since worker nodes can join and leave the system dynamically at any time, we need some kind of central coordinator that can react on these topology changes.

The worker daemon can perform distance measurements to another node and feed the result into the Vivaldi network coordinate algorithm. With each message sent to the coordinator over the control protocol, the worker transmits its current coordinates. Also, the worker daemon can receive requests to start or stop the broker on the worker node.

4.1.3 Coordinator

This component orchestrates the message broker network. It uses multiple mechanisms to gather information about the current network state. This includes the coordinates of the nodes (workers and gateways) as well as the bridging table.

By using a management interface to the cloud and edge infrastructure provider, the coordinator has knowledge about which nodes are currently available. In some implementations, the coordinator might also be integrated into the cloud provider itself. For example, in Kubernetes clusters, the coordinator could gather information about cluster nodes from the Kubernetes API. The scaling mechanism could be implemented as a combination of a Horizontal Pod Autoscaler together with a custom scheduler that assigns the brokers to the desired nodes.

Each node implements a control protocol to accept commands from the coordinator. On the worker node, a separate daemon is necessary because workers that currently do not host a broker process still have to be able to receive commands from the coordinator. On client nodes, the control protocol is integrated in the gateway component. The nodes regularly perform distance measurements to other nodes. The nodes use these results to calculate their network coordinates and send them to the coordinator.

Having knowledge about the current topology, the coordinator is able to reason about the consequences of certain scaling actions. For instance, if there are many subscribers close to a worker that currently has no running broker, the coordinator can calculate the impact on the Quality of Service if it would start a broker on this node. On the other hand, if a broker has few subscribers connected, the coordinator might decide to shut down this broker in order to save resources.

4.1.4 Client Gateway

On the client side, our extensions require support of a custom control protocol. If we required clients to implement the control protocol themselves, regular MQTT clients would not be compatible with our system. To tackle this problem, we use the client gateway component that was already introduced in EMMA and extend it in our work.

The gateway is a proxy service that behaves like a broker on the client side and like a client on the broker side. It forwards all regular MQTT packets to the currently selected broker and implements our control protocol. Two features require a custom control protocol that's implemented by the gateway:

- Gateways should connect to the best (with regard to distance) broker. The controller can direct the gateway to connect to a specific broker.
- We want to monitor latencies to certain brokers to be able to estimate the client's location within the topology. The controller sends a request to the gateway containing a list of brokers to perform measurements to.

4.2 Mechanisms

Our system adds multiple mechanisms to the broker network in order to dynamically assign client gateways to brokers and start or stop brokers as needed.

4.2.1 Node Location Awareness

In a system with more than one broker, client gateways have to decide which broker to use. In the EMMA paper [RND18], this decision was made based on measurements between all nodes of the network. Since this imposes a high management overhead, we introduce network coordinates calculated by Vivaldi. Since neighbor selection for measurements has a high influence on the error of the coordinates (see Vivaldi paper [DCKM04]), we select both local and more distant nodes. The node selection changes over time to reflect new knowledge about distances between the nodes. With the coordinates, it is possible to estimate distances between all known nodes and base decisions on these estimations. In chapter 5, we will explain our implementation in more detail.

4.2.2 Broker Scaling

Trivially, if every worker hosted a broker, the performance (defined by measures like message delivery latency) would be optimal. However, this would lead to high resource consumption on both cloud and edge resources. The decision how many brokers should be running is a tradeoff between resource usage and performance. We introduce a scaling mechanism that deploys brokers on worker nodes. The mechanism scales the broker network to find a state that balances resource consumption and performance. Chapter 6 will go into further detail on this topic.

4.2.3 Control Protocol

We use a control protocol to perform the tasks described above. Coordinator, worker daemons and gateways need to support it. Due to the fact that neither clients nor brokers have to implement it, we can use existing clients and brokers for our network. The protocol uses different types of messages to support our mechanisms, which we will describe in the next chapters as well.

Since we are building MQTT messaging middleware, we can use in-band MQTT messages to specific topics to deliver control protocol packets. Each node subscribes to its own topic that others can publish messages to. The responses will be delivered back to the topic of the node that originally sent the request. This mechanism enables us to perform request-response style communications required for our control protocol. This way, we do not have to set up separate connections between nodes but use the existing MQTT connections. In each control packet, the sending node also includes the following metadata:

- The timestamp of the node's current time (to measure distances between nodes)
- The currently known coordinates of the sending node, including estimated error (which are required for Vivaldi coordinate calculation)

The receiving node uses this metadata to update its coordinates the Vivaldi algorithm. For details about how Vivaldi calculates coordinates, see Section 2.6.1.

CHAPTER 5

QoS-aware Broker Discovery

In a distributed broker network with more than one broker, clients have to select which broker to connect to. This decision is critical and non-trivial since it impacts the Quality of Service as experienced by the client. In our case, we are aiming to reduce end-to-end latency, and therefore our goal is to select the closest broker. To that end, we need detailed information about the topology, which we gather by performing distance measurements between nodes of the network. The resulting distance graph serves as base for deciding which client should connect to which broker and where we have to spawn new brokers when autoscaling. In this chapter, we will first explain how we perform latency measurements and estimate distances, and describe the broker selection mechanism.

5.1 Latency Measurements

As we study IoT applications which require low latency, we focus on latency as Quality of Service class (see subsection 2.2.3). Other types of applications can use metrics that are more tailored to the application and optimization goals. For instance, for an on-demand video streaming application, bandwidth might be a more appropriate QoS metric than latency.

Latency can be measured on different layers. To measure end-to-end latency as experienced by the users, we would have to use application-layer metrics. This kind of metric also takes processing time of the application into account, which is a non-negligible factor for user experience. For example, consider a smart city application that uses computer vision to detect pedestrians and warn car drivers. In this case, end-to-end latency on application level would be the time between detection of a pedestrian to warning the driver. For general compatibility with any application, we can not use application-layer monitoring, since we would have to integrate our monitoring system into the application.

Instead, as an approximation, we perform distance measurements below the application layer. This can take place on the messaging middleware layer. There, we can define end-to-end latency as the time between publishing a message and receiving it at the subscriber. On the transport layer, we could use ICMP to perform latency measurements. However, the lower the layer, the less accurate the approximation gets because each layer adds potentially non-constant overhead. Also, note that we consider IoT applications, where clients might not be reachable via ICMP from the Internet depending on their network link (e.g., behind network address translation).

On which layer we perform measurements is a tradeoff between simplicity and accuracy. Since we build messaging middleware, we can use the messaging layer (i.e., the middleware layer) for monitoring. The fan-out of messages at message brokers is an I/O bound task that adds delay. As a consequence, latency on the middleware layer is a good proxy metric if and only if the application itself is also I/O bound. Instead, if the application does CPU bound data processing, end-to-end latency as experienced by the users could differ from the end-to-end latency on the messaging layer. Varying processing time within the application (e.g., due to computationally intensive tasks) can introduce additional overhead compared to measurements on the transport layer. For our work, we assume that the application does not add processing time, and we can use measurements on the messaging layer.

5.2 Distance Estimation

In order to find the closest broker for each client gateway, we have to know the distances between all gateway and broker nodes. The naive approach to do that would be to ping all brokers from all gateways, which is what EMMA does [RND18]. This produces a management overhead and unnecessary network load, especially when scaling to a higher number of brokers. For instance, the topology of an application that spans lots of cities in multiple continents would consist of dozens of cloud brokers and hundreds of edge brokers. In this case, the naive approach is not feasible due to the large number of measurements that gateways have to perform to find the closest broker. To reduce this overhead, we use a network coordinate system to estimate distances (see section 2.6). We chose Vivaldi because the topology of an edge computing application using our middleware might include ephemeral clients and dynamically available edge resources.

5.2.1 Distance Measurements

By using network coordinates, we want to reduce the monitoring overhead imposed by the measurements that would be required using the trivial approach, i.e., performing measurements for each link of the fully connected graph between nodes. Vivaldi still needs distance measurements between nodes in order to iteratively calculate the coordinates. However, the number of measurements is much lower compared to the trivial approach, as we show in our evaluation (see chapter 7). We perform these measurements in the following way.

The source of measurements is the metadata of ping/pong control protocol packets, which include the timestamp and the sender's coordinates. On a new *ping* message, the receiving node sends a *pong* message to the senders' control protocol topic. The pong message includes the timestamp of the ping message. This enables us to calculate the latency without having to store the ping message's timestamp or another correlation identifier on the sender's side. The calculated latency and the coordinates serve as input for the calculation of the coordinates. Gateways and workers use different techniques to select the peer nodes for distance measurements.

Gateways are the most ephemeral and mobile components in the system: As laid out in section 2.1, clients might be moving around and change their network location. Gateways run on the same device as the client application or at least close to them. To keep their coordinates up-to-date, gateways regularly perform measurements to a number of workers. Doing so, they alternate between random workers and close workers, requesting the sets of workers from the coordinator. In our evaluation, we found that this mechanism delivers the most accurate results (see Chapter 7). The results are passed to the Vivaldi algorithm that updates the node's network coordinates.

Workers ping all other workers periodically and send their coordinates to the coordinator. This leads to stable, accurate network coordinates in the network core and decreases the error rate for client coordinates. In exchanging measurement ping/pong messages, receiving nodes use the sender's coordinates to update their own coordinates (see subsection 2.6.1). Coordinates with high error rates (i.e., coordinates of nodes that have not converged to their stable coordinate) have little impact on nodes with stable coordinates. Therefore, new workers can not disrupt the coordinates of the existing ones.

5.2.2 Vivaldi Execution

When a node receives a pong packet, it calculates the delay between the timestamp of sending the ping packet and the current local timestamp. Together with the remote coordinate and the remote error, Vivaldi can update the node's coordinate. Subsequent messages to other nodes will include the updated coordinates, propagating them to others.

Vivaldi coordinates contain an estimated error, which can be used by the client gateways to vary the measurement interval. This allows us to reduce the number of explicit measurements: As soon as the error gets lower (i.e., coordinates converge to their optimal position), we can reduce the measurement frequency.

5.2.3 Gateway Measurement Algorithm

We developed this algorithm using iterative prototyping (see chapter 7). It turned out to be optimal to measure distances to both randomly chosen (and potentially distant) nodes and nodes that are in the node's neighborhood. We present the results of the experiments to find this algorithm in Figures 7.5 and 7.6. The gateway performs regular measurements as follows (see Figure 5.1).

1. The gateway requests a set of 4 random workers $w_1 \dots w_4$ from the coordinator. This will include both close and distant brokers, and the measurements will yield a rough network location.
2. The gateway will now perform measurements to the random brokers and update its coordinates using Vivaldi.
3. In the next request to the coordinator the gateway retrieves the 6 closest workers $w_5 \dots w_{10}$. This request includes the previously calculated coordinates, allowing the coordinator to return a selection of potentially close workers.
4. After performing the measurements to close workers, the gateway passes the results to the Vivaldi algorithm. The second round of measurements will further improve the accuracy of the coordinates as the coordinates move towards the gateways' neighbors.
5. Having updated the coordinates the gateway sends them to the coordinator.
6. The next time the coordinator checks for possible optimizations the coordinator might direct the gateway to a better (i.e., closer) broker.

5.2.4 Management Overhead

The client gateways select a constant number of workers for distance measurements each minute. Hence, the management overhead is linear relative to the number of client gateways. Given the set of client gateways $\mathcal{C}$, the number of measurement packets P per minute is

$$P/\min = 10 \cdot |\mathcal{C}|$$

Especially in large broker networks, this saves a lot of measurements compared to the original EMMA approach, where all gateways ping all brokers every 15 seconds. Given the set of brokers $\mathcal{B}$, the P_{EMMA} per minute is

$$P_{EMMA}/\min = 4 \cdot |\mathcal{B}| \cdot |\mathcal{C}|$$

Trivially, it follows that for more than three brokers, $P/\text{min} < P_{EMMA}/\min$. Therefore, we can drastically reduce the monitoring overhead (see Figure 7.9).

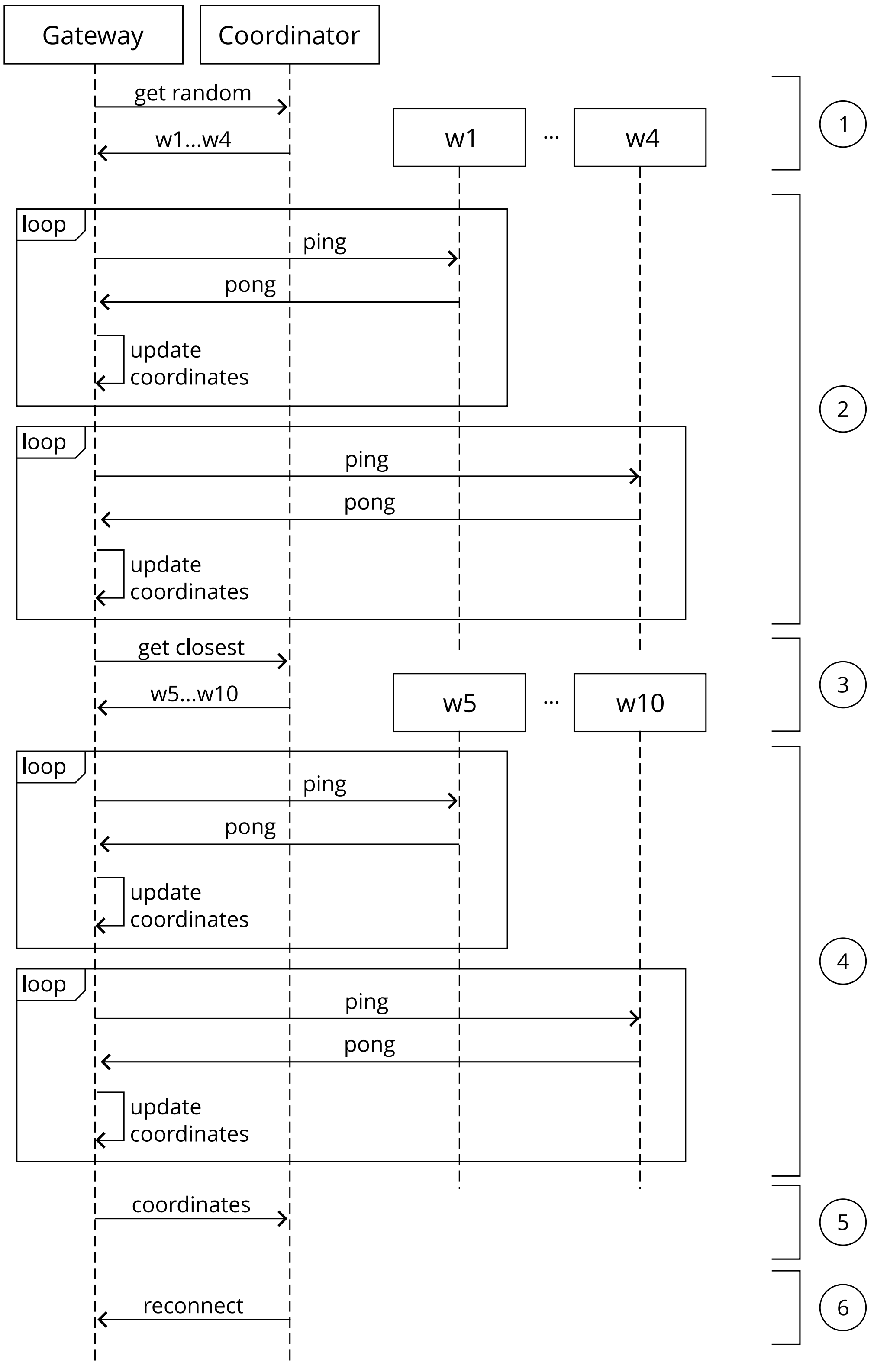


Figure 5.1: Sequence diagram that shows the regular client gateway measurements.

5.3 Broker Selection

As the central coordinator receives the nodes' coordinates, it maintains an overview of the overlay topology. Based on the coordinates calculated by the nodes, the coordinator can assign the optimal broker for the gateways.

The coordinator periodically checks for every client gateway if there is a closer broker available. The algorithm works as follows. The coordinator loads the current broker the gateway is connected to and the gateway's coordinates. Then, the running brokers are sorted by distance to the client gateway coordinate. To find the closest broker, the algorithm groups the brokers by latency where the group boundaries increase exponentially. We do this because if the distance between two similarly close brokers varies, the algorithm might perform unnecessary migrations.

In order to decide which broker the client gateway should connect to, we can distinguish three different cases.

1. The client gateway is connected to the optimal broker. Then, nothing has to be done.
2. The optimal broker is in a lower latency group than the current broker. Then, the coordinator requests the gateway to switch to the optimal broker.
3. Both current and optimal broker are in the lowest latency group. Then, the coordinator uses the mechanism described below to prevent unnecessary migrations.

5.3.1 Threshold-based Migration

In third case described above, the coordinator uses a threshold-based migration mechanism based on the number of subscriptions per broker. The objective of the mechanism is to prevent flapping between brokers that are similarly close to the client gateway. If we moved clients as soon as there is another broker with fewer subscribers, they would be moved back and forth with every iteration.

Therefore, we use a threshold-based migration mechanism when deciding whether to move a client or not. A weight factor θ allows us to define the threshold. Intuitively, θ influences the eagerness of the coordinator to move a client between brokers of the same latency group. Assume that that B_i is the set of brokers in latency group i and θ a weight between 0 and 1. Then, the coordinator checks if the number of subscribers at the new broker b_n plus the weighted sum of total subscribers in the latency group is lower than the number of subscribers at the current broker b_c:

$$\text{subscribers}(b_n) + \theta \cdot \sum_{b \in B_i} \text{subscribers}(b) < \text{subscribers}(b_c)$$

Trivially, if $\theta = 0$ it follows that $\delta = 0$ and the algorithm migrates the client as soon as the number of subscribers at the new broker is lower than at the current broker. Such an extreme parameter setting would lead to the flapping behavior we would see without a threshold. For the other extreme ($\theta = 1$) the algorithm will never migrate the client. Choosing a value *between* one and zero, however, leads to a less fluctuant migration behaviour.

5.3.2 Bootstrapping

In a newly deployed system, all nodes have the same coordinates, the origin of the vector space. At this stage, coordinates can not be used to estimate distances. As workers join and perform a certain number of measurements, the node's coordinates converge to an equilibrium state. The evaluation shows how many measurements we have to perform for the coordinates to converge (see section 7.3).

When new client gateways appear for the first time, their coordinates are not accurate enough to find the closest broker. As long as the error rate of the client coordinate is high, the coordinator can return random nodes instead. Furthermore, the client gateway can remember the last coordinates and the next time it comes online, it can reuse these coordinates as starting point.

CHAPTER 6

Self-adaptive Network Configuration

In this chapter, we present a mechanism that allows the coordinator to scale the broker network. The broker network leverages cloud and edge workers to host brokers. While there is always at least one cloud broker running, the coordinator starts and stops edge brokers on demand. We use the concept of osmotic pressure to define a metric that reflects the proximity of clients to edge resources that can host brokers. The coordinator uses the pressure value in order to optimize the broker network (i.e., start and stop brokers on available cloud and edge workers) towards low distances between clients and brokers, resulting in a lower end-to-end latency.

6.1 Broker Scaling Problem

Broker networks can run and scale across cloud and edge computing resources (see chapter 2). Especially in topologies where publishers and subscribers concentrate in local neighborhoods, spawning a broker on an edge worker can improve Quality of Service in terms of end-to-end latency. For example, in an industrial IoT scenario, a factory could have components that subscribe to topics that other components publish messages to. It would be much more efficient to route messages via a locally deployed edge broker than via a cloud broker.

The scaling decision, i.e., where to deploy brokers is not trivial, as the following example shows. A naive approach could deploy an edge broker as soon as there are clients close to the worker. However, this can lead to a waste of resources: Assume there are 10 clients close to edge worker A and one client close to edge worker B, with a distance of 10 ms to each edge worker and 30 ms to the closest cloud worker. Then, there are four scenarios where to deploy brokers:

Scenario	Edge A	Edge B	Mean Distance
1	✓	✓	10 ms
2	✓	–	12 ms
3	–	–	30 ms
4	–	✓	32 ms

The difference between scenario 1 with one edge broker and scenario 2 with two edge brokers is only 2 ms. The naive approach would deploy both edge brokers (scenario 1), although the impact on the overall Quality of Service is relatively small. This shows that scaling the broker network is a tradeoff between Quality of Service improvement and resource consumption.

To reflect this tradeoff in an auto-scaling mechanism, we define a metric that allows us to set thresholds. Doing so, we have to consider more than just the mere presence of clients to an edge worker. We have to design a model that quantifies the proximity of clients to a worker node in order to decide about proper broker locations. In other words, the metric should enable us to design an algorithm where we can define a threshold to influence which scenario of the example above the algorithm will choose.

6.2 Osmotic Pressure

Rausch *et al.* [RDR18] proposed a model based on osmotic computing. In this model, clients apply pressure to edge computing nodes that are close to these clients. As soon as there is a worker with a pressure value that is higher than a specified up-scaling threshold, a new broker will be deployed on the edge with the highest pressure. If the broker pressure decreases below a down-scaling threshold, the workload will be removed in order to save resources.

In our system, we want to optimize the mean distance between clients and workers, i.e., possible broker locations. If we want to apply osmotic pressure in the context of our middleware, we require the pressure to have the following properties:

1. A high pressure value means that a large number of clients are close to the worker node. In this case, the coordinator will spawn a broker on the worker node to decrease the mean client-broker distance.

2. A low pressure value means that the running broker has few or no clients in its neighborhood. The coordinator can stop the broker without increasing the mean client-broker distance.

3. Pressure values should be globally comparable. Otherwise, a certain threshold might be optimal for one region, but not for another region, where the latency on the link between clients and edge resources differs from the first region (see example in subsection 6.2.1). Therefore, we should be able to normalize pressure in order to define threshold values that we can apply at any location in the topology.

Next, we will present two approaches to define an algorithm to calculate the pressure value.

6.2.1 Distance Pressure

One possible way to calculate the pressure on a worker w is to calculate the sum of the inverse distances of all clients to the worker.

$$\text{pressure}(w) = \sum_{c \,\in\, clients} \frac{1}{\text{distance}(c, w)}$$

This fulfills the first two properties that we require. However, it is hard to normalize the pressure for the whole topology. Edge resources that are more distant from the closest cloud might require a different threshold than edges that are closer to the cloud. For example, consider the following scenario. Two cloud regions C_1 and C_2 have one edge site each, E_1 and E_2 (see Figure 6.1). In the figure, the labels of the connections denote the distances between the regions. It is hard to find distance pressure thresholds that are optimal for both regions. If the thresholds are optimal for one edge region, they would not fit for the other region, leading to over- or under-provisioning. Therefore, this approach violates property 3 above.

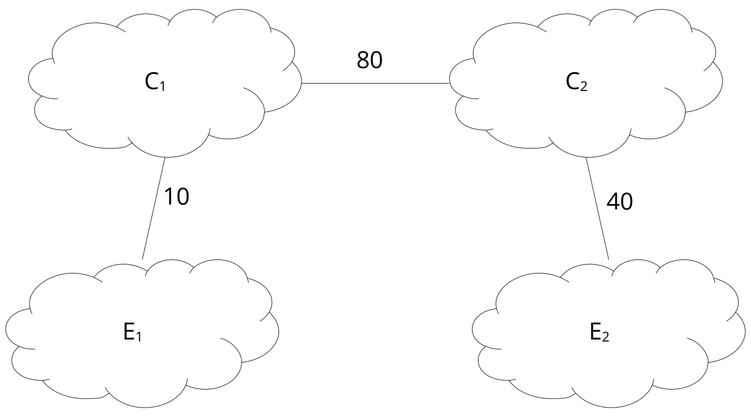


Figure 6.1: Distance pressure example scenario

6.2.2 Distance-Diff Pressure

To solve the issue with distance pressure, we do not take the distances into account. Instead, we compare the possible distance improvement or penalty for clients, respectively, if we add or remove a broker. We use two different pressure values.

Worker Pressure

This is the pressure that acts on worker nodes with no running broker. We calculate the worker pressure for a certain worker w in the following way. First, we select the subset $\mathcal{C}'$ of all clients $\mathcal{C}$ that are closer to w than to their currently selected broker:

$$\mathcal{C}'(w) = \{c \in \mathcal{C} \mid \text{distance}(c, w) < \text{distance}(c, \text{broker}(c))\}$$

We skip clients if there is already a closer broker available. This prevents provisioning brokers for clients that have not been migrated to the optimal broker yet.

The pressure $P(w)$ is the sum of the distance deltas of all closer workers:

$$P(w) = \sum_{c \in \mathcal{C}'(w)} \log(\text{distance}(c, \text{broker}(c)) - \text{distance}(c, w))$$

where we take the log of the distance deltas in order to define the thresholds as orders of magnitude. For example, a threshold of 2 means that one client can improve the latency by two orders of magnitude or two clients can improve the latency by one order of magnitude.

Then, each iteration, and if the worker pressure exceeds the predefined threshold, the scaling algorithm will start a broker on the worker with the highest worker pressure.

Broker Pressure

This pressure acts on broker nodes, i.e., workers that host a broker. The broker pressure quantifies the impact of removing a broker with regard to client distances. We assume that the client would be connected to the next-closest broker. Under this assumption, we can remove the broker if the broker pressure is low, i.e., the client distances do not increase too much.

For the calculation of the worker pressure, we select the subset $\mathcal{C}'$ of all clients $\mathcal{C}$ that are connected to the broker in question.

$$\mathcal{C}'(b) = \{c \in \mathcal{C} \mid \text{broker}(c) = b\}$$

Similar to the worker pressure calculation, we skip clients that can be re-connected to a better broker. The pressure $P(b)$ is the sum of the distance deltas to the next-best broker of all those clients (the latency penalty if the current broker was removed):

$$P(b) = \sum_{c \in \mathcal{C}'(b)} \log(\text{distance}(c, \text{next}(\text{c})) - \text{distance}(c, b))$$

where next(c) returns the next-best broker for client c.

Listing 6.1: Scaling Algorithm

```
# check if we have to scale up
worker_pressures = get_worker_pressures()
if len(worker_pressures) > 0:
    candidate_worker, pressure = max(worker_pressures)
    if pressure > threshold_up:
        candidate_worker.start_broker()

# check if we can scale down
broker_pressures = get_broker_pressures()
if len(get_brokers()) > 1:
    candidate_broker, pressure = min(broker_pressures)
    if pressure < threshold_down:
        candidate_broker.shutdown()
```

6.3 Dynamic Scaling Algorithm

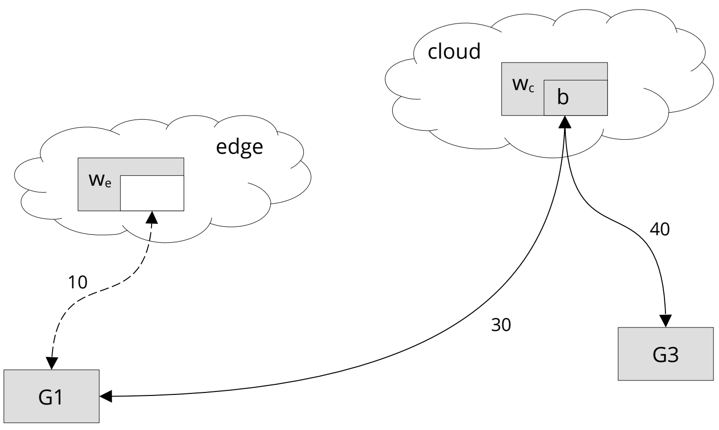
Having defined osmotic pressure, we can solve the scaling problem using a simple, threshold-based algorithm. The scaling decision is a tradeoff between latency and edge resource consumption. Depending on the application's requirements, the administrator has to find the optimal thresholds. For example, take an industrial IoT application uses the middleware. Assume that a factory has 10 IoT devices and the factory is close to an edge site to which the latency is one order of magnitude lower than to the next cloud site. Then, for the algorithm to deploy a broker on the edge, the maximum threshold the administrator can choose is 10.

The scaling algorithm calculates the broker and worker pressures periodically. In each run, the algorithm selects the worker with the highest pressure as the candidate worker. If this pressure is above the configured up-scaling threshold, the coordinator starts the broker. As a consequence, the pressure value will drop below the threshold. If there is no more broker with a pressure value above the threshold, the broker network reached an equilibrium state. Otherwise, the coordinator will deploy another broker in the next iteration to further optimize the broker network.

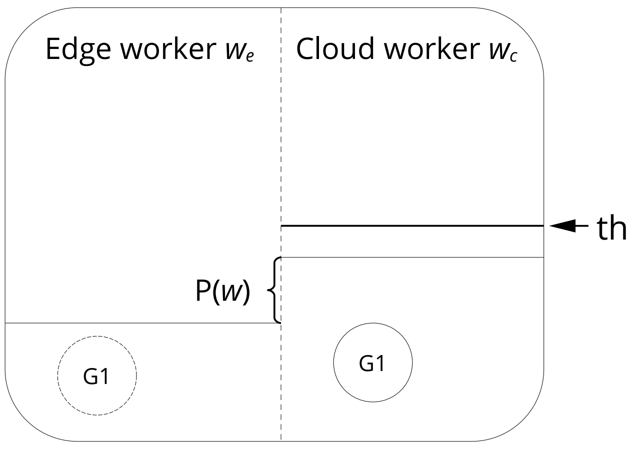
Then, if there is more than one broker left, the broker with the lowest pressure gets selected as the candidate broker to be stopped. If the lowest broker pressure is below the configured down-scaling threshold, the coordinator stops the broker. Similarly to the upscaling, this will affect the pressure value of the broker.

Listing 6.1 shows this mechanism in pseudocode. The functions get_worker_pressures() and get_broker_pressures() calculate the pressure values as described above.

To illustrate this, we use an example topology that consists of one cloud and one edge region. We assume that all gateways are currently connected to the broker on the cloud

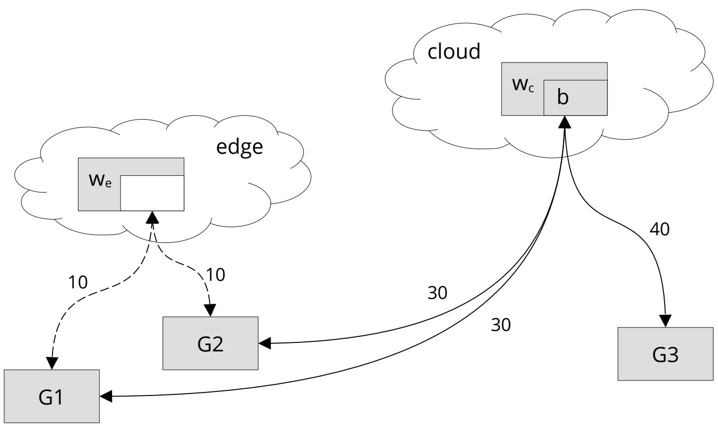


(a) Only one gateway is close to the edge worker.

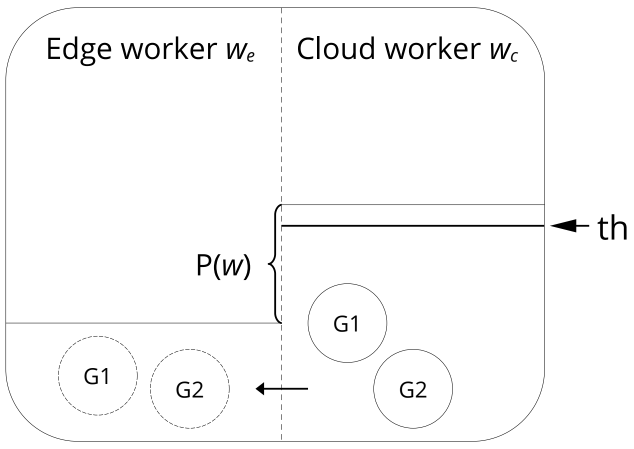


(b) Worker pressure $P(w)$ is below the threshold th. No broker will be deployed on w_e.

Figure 6.2: Scenario 1 of the osmotic scaling example.



(a) Two gateways are close to the edge worker.



(b) Worker pressure $P(w)$ is above the threshold th. A new broker will be deployed on worker w_e.

Figure 6.3: Scenario 2 of the osmotic scaling example.

worker w_c. Following, we will discuss two different scenarios and the resulting worker pressure.

In the first case (see Figure 6.2), only client gateway `G1` is active and close to edge worker w_e. This client could improve its latency if there was a broker running on w_e. Here, the worker pressure $P(w_e)$ is below the configured threshold th (see Figure 6.2b) and the algorithm does not change the system state.

In the second case (see Figure 6.3), an additional client gateway `G2` close to w_e becomes active. Now, already two brokers could benefit from a closer broker on the edge. This increases the sum of the possible distance improvements in the neighborhood of w_e. As a result, the worker pressure $P(w_e)$ rises above the threshold (see Figure 6.3b) and the algorithm deploys a broker on w_e. Then, the coordinator can move the gateways `G1` and `G2` to the edge broker.

CHAPTER 7 Evaluation

With our evaluation, we want to answer the research questions that we stated in Chapter 1. The overall goal is to reduce the end-to-end latency for clients while reducing monitoring overhead and resource consumption. To achieve this, we use a two-fold approach. First, to reduce management overhead, we use a network coordinate system that allows us to estimate distances instead of having to measure them (see chapter 5). We evaluate if these estimations are accurate enough to select the closest broker for the client. Second, we evaluate the auto-scaling mechanism based on osmotic pressure that enables us to deploy brokers close to clients on demand (see chapter 6).

In this chapter, we will first describe our methodology, followed by the experiment setup. Finally, we will present the results of the experiments.

7.1 Methodology

We use multiple simulation experiments that can simulate interactions between the components of our system (see chapter 4):

EX-V1. In a simple experiment, we evaluate if Vivaldi can be used to estimate distances in a network of edge and cloud brokers. In this step, we will also try to find the minimum number of measurements that Vivaldi needs for sufficiently accurate distance estimates.

EX-V2. Still within the simple experiment setup, we will furthermore evaluate which peers a node has to select to perform measurements to. These measurements provide the input for Vivaldi to train the coordinate model.

EX-S1. The scenario described in the EMMA paper [RND18] is the first simulation scenario. This allows us to evaluate and compare our setup with the EMMA results.

EX-S2. Our dynamic scaling algorithm turns the static EMMA scenario into a dynamic variant. This means that we do not have to decide where to spawn brokers statically but let our algorithm decide it based on client traffic.

EX-S3. An Industrial IoT scenario will help us to evaluate the behavior of the dynamic scaling mechanism in a topology with dynamic nodes. While nodes only join the network successively in the EMMA scenario, nodes will also leave the network in the IIoT scenario.

7.1.1 Topology

In our experiments, we simulate interactions between the components of our messaging system which run on nodes that connected via WAN (wide area network, i.e., the Internet) and LAN (local area network) links. The experiments use a topology model that is based on a graph data structure where vertices represent nodes, connected by edges that have the distance attached. The regions of the Cloudping dataset[1] form the Internet core of the graph. In each region, there is a cloud datacenter that can host cloud brokers as well as the coordinator.

We add additional edge sites to the topology, connecting them to a region. Then, we add the components of our system (see chapter 4) to the graph: cloud and edge brokers, clients and workers (which can host brokers on demand). We assume that all nodes of the broker network can reach each other. This means that each broker can reach any other broker in the network and measure the distance to it.

To synthesize the topology model, we use the ether library[2] [RLF+20]. The library uses the graph data structure provided by networkx[3]. Ether provides a pre-defined set of components (e.g., cloudlets, hosts, etc.) which can be added to the topology.

The distances between the cloud and edge sites and between edge sites and connected clients are not constant. Every time we calculate the latency of a link a packet passes on the path from one node to another, we sample the latency from a log-normal distribution. This simulates the varying latencies that we observe in reality. The links between cloud and edge brokers simulate WAN links and therefore have a higher mean latency than the LAN links between edge brokers and clients. Also, the standard deviation varies depending on the type of link, such that LAN links have a smaller standard deviation than DSL or Wi–Fi links.

7.1.2 Vivaldi Evaluation

For experiments EX-V1 and EX-V2, we do not need to simulate the actual packets transmitted between nodes because we focus on the coordinate accuracy and neighbor detection. This way, we can run experiments without having to start a discrete event

[1]`https://api.cloudping.co/averages/day`, downloaded on 2020-06-20
[2]`https://github.com/edgerun/ether`
[3]`https://networkx.org/`

simulation. We run two different experiments in this environment. Following, we will describe the common experiment setup, followed by the detailed description of the two experiments.

Distance Measurements

We simulate a distance measurement in the topology in the following way: First, we calculate the shortest path (using ether, which uses the Dijkstra algorithm internally). For each link on the path, we take a sample of the latency distribution of the link. Now, we can use the sum of the link latencies as a distance measurement sample for the simulated ping.

Error Calculation

We calculate the error rate of the distance estimation by comparing the distances between the nodes' coordinates with the actual distances in the topology graph. However, errors don't necessarily lead to wrong decisions. In order to assess the impact of distance estimation errors, we compare the estimated neighbors with the actual neighbors. Only if the neighbors guessed from Vivaldi coordinates differ from the actual neighbors, the coordinator will select the wrong broker for the client gateway. Otherwise, the exact order of the neighbors does not matter since we use latency groups anyway (see section 5.3).

7.1.3 Simulation Scenarios

We implemented a discrete event simulation that can simulate interactions between coordinator, workers, brokers and clients. Components can send messages to each other by delivering packets into message queues. This way, we can simulate both MQTT messages and control protocol messages sent between nodes. Note that this kind of simulation differs from low-level network simulators as we do not consider the network layers below our messaging middleware. We can abstract from the lower layers since we perform distance measurements on the messaging layer (see section 5.1). Furthermore, this simplifies the implementation of the simulation.

Each message delivery is delayed for a duration according to the topology distances which leads to realistic network latencies. This enables us to evaluate the accuracy of distance estimations as well as different approaches of broker provisioning. Following, we will describe which processes interact with each other. Each process is associated with one node of the topology graph (except the scenario process itself, which controls the experiment).

Nodes can run one or more processes at a time. Worker nodes (i.e., compute nodes in the cloud or on the edge) run at least a worker process and – if deployed – a broker process. Client nodes run a process for QoS monitoring, one or more publisher processes and one process that handles all other tasks.

ClientProcess receives incoming messages on topic it has subscribed, handles control protocol messages that direct the client to connect to a specific broker, perform distance measurements to other nodes when requested by the coordinator and (if enabled) publishes messages in the configured interval.

WorkerProcess monitors distances to all other worker nodes periodically, regardless of whether a broker is running or not. This way, workers maintain accurate network coordinates.

BrokerProcess runs on worker nodes. Depending on the evaluation scenario, brokers either run from the beginning or start as soon as the coordinator decides to do so. BrokerProcesses maintain a table of client subscriptions and forward publish messages to all subscribers. For our simulation, we assume that all brokers have knowledge about subscriptions on other brokers. The concrete implementation is out of the scope of this thesis.

CoordinatorProcess runs on a single node of the topology. It monitors distances between the clients by requesting them to perform measurements. Based on the results, the coordinator decides which broker the clients should connect to.

DistanceDiffBrokerScalerProcess optionally runs on the coordinator node and dynamically starts or stops brokers on demand. It uses the scaling algorithm described in chapter 6, periodically checking the system for possible optimizations.

ScenarioProcess handles the scenario playbook, i.e., it creates and starts clients and brokers and manages subscribers and publishers. This enables us to simulate different scenarios, ranging from static configurations to dynamic provisioning of brokers using scaling algorithms.

7.2 Experiment Setups

In this section, we describe in detail the experiment setups within the respective environment. The first two subsections deal with the Vivaldi experiments (EX-V1 and EX-V2), while the last three subsections explain the discrete event simulation scenarios (EX-S1 to EX-S3).

7.2.1 EX-V1. Accuracy Evaluation

With this experiment, we evaluate if Vivaldi can accurately estimate distances between network nodes in a. This could reduce the need for distance measurements by instead calculating them based on coordinates (see section 5.2). To evaluate accuracy, it is sufficient to run a simple experiment that executes Vivaldi on behalf of each node.

In the experiment, we first select random pairs of nodes and execute Vivaldi with the distance as defined in the topology. Each execution of Vivaldi uses one of the tuples (n_1, n_2) where n_1 and n_2 are brokers. Because Vivaldi improves the coordinate accuracy iteratively, the accuracy of distance predictions depends on the number of Vivaldi executions on each node. Therefore, we define a break condition that stops the execution as soon as all nodes have executed vivaldi at least x times.

7.2.2 EX-V2. Peer Selection Evaluation

We will use this experiment to evaluate which strategy results in the most accurate coordinates when selecting peers for distance measurements on the client gateways. As in experiment EX-V1, we use the simple experiment environment instead of a discrete event simulation.

We evaluate different strategies that will show how well Vivaldi predictions match the actual distances depending on the peer selection. Each strategy provides a selection of nodes from a static set of nodes or a selection function that returns such a set. A selection function enables us to dynamically change the peer selection in each iteration.

Before we execute the measurements on the client gateways, we run 300 measurements on all workers. This ensures precise worker coordinates since these nodes usually have been active for a longer time when the client gateway joins the network.

During the experiment, the main loop cycles the sets or selection functions until it reaches a configured iteration count. An experiment configuration defines one or more sets of peers or selection functions. Possible sets or selection functions are:

- Static sets of brokers (edge or cloud) that does not change over the runtime of the experiment
- Functions that return a set of random brokers (edge, cloud, or both)
- Functions that return close brokers according to the current coordinates
- Combinations of the above

For instance, each node could randomly choose a static set of five brokers from the set of all cloud brokers at startup and cycle this set during the whole lifetime. A dynamic configuration could choose five random peers each cycle. Another configuration could alternate between dynamically selecting 5 cloud brokers and 5 edge brokers. For example, a configuration could alternate between one function that selects five random brokers and another function that selects the five closest brokers.

7.2.3 EX-S1. Static EMMA scenario

In this scenario, we evaluate if using the estimated distances calculated from network coordinates leads to the same result as the evaluation in the EMMA paper [RND18]. To be able to directly compare the results, we use the same scenario.

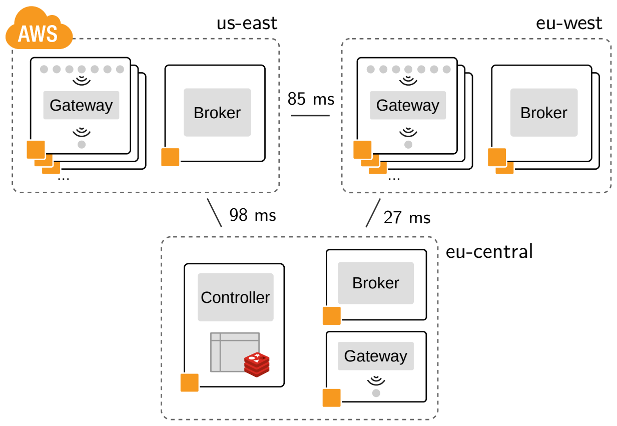


Figure 7.1: EMMA scenario topology

The authors of EMMA used the following setup for their evaluation. The testbed spans three data centers, each hosting brokers and gateways (see Figure 7.1 from [RND18] p 194).

The scenario comprises multiple steps that are run sequentially, each adding or removing components from the system. The steps are as follows (from [RND18] pp 194–195):

1. Clients appear and subscribe to the topic *global*
   - one publisher and subscriber in *us-east* and *us-west*
   - one subscriber in *eu-central*
2. Client group[4] appears in *us-east*
3. First broker spawns in *eu-west*
4. Client group appears in *us-west*
5. Broker spawns in *us-east*
6. Second broker spawns in *eu-west*
7. Subscriber to topic *global* in *eu-central* disappears
8. Broker in *us-east* shuts down

In the EMMA paper, the scenario steps were triggered manually. In our evaluation, we trigger the next step after two minutes have passed in the simulation.

[4]A client group consists of 10 hosts running a client gateway, 1 subscriber and 7 publishers

Minute	Factory A	Factory B	Factory C
0	✓	✓	✓
2	-	✓	✓
4	-	-	✓
6	-	-	-
8	✓	-	-
10	✓	✓	-
12	✓	✓	✓

⋮

Table 7.1: Factory state in Industrial IoT scenario

We will use the results of the EMMA paper as baseline. Our objective is to show if our system selects the same brokers based on distance estimations as EMMA does based on measured distances. Since our system also relies on pings to some nodes to gather distance data, we will compare the number of packets. This will show if we can reduce the management overhead for topology awareness and still maintain accuracy.

7.2.4 EX-S2. Dynamic EMMA scenario

With this experiment, we evaluate the broker scaling algorithm (see chapter 6). We use the same scenario as in EX-S1, which allows us to compare the static configuration with the dynamically scaled broker network.

The scaling process uses the distance-diff pressure calculation as described in subsection 6.2.2. In this experiment, we will use the static EMMA scenario, adding new workers over time and running the same steps as described before, except for the brokers: The scaling process will automatically start or stop brokers on demand on the existing workers.

7.2.5 EX-S3. Industrial IoT scenario

As in EX-S2, we aim to evaluate the broker scaling algorithm. However, in the Industrial IoT scenario, we tried to simulate a more dynamic system than in the EMMA scenario.

The core topology is similar to the previous scenario. Again, we use three regions to form the Internet backbone and spawn one cloud broker per region. In each region there is one factory which can be started or stopped. A factory consists of three production lines, each of which can host a broker. Figure 7.2 gives an overview about the scenario topology.

During the scenario, we start and stop factories as follows. At the beginning, all three factories are started. Then, every minute, we stop one factory after the other and wait between the steps. After that, we start the factories (again in an interval of two minutes) and leave them running until the end. This simulates three different time zones and the resulting operation hours. Table 7.1 gives an overview about the factory states over time.

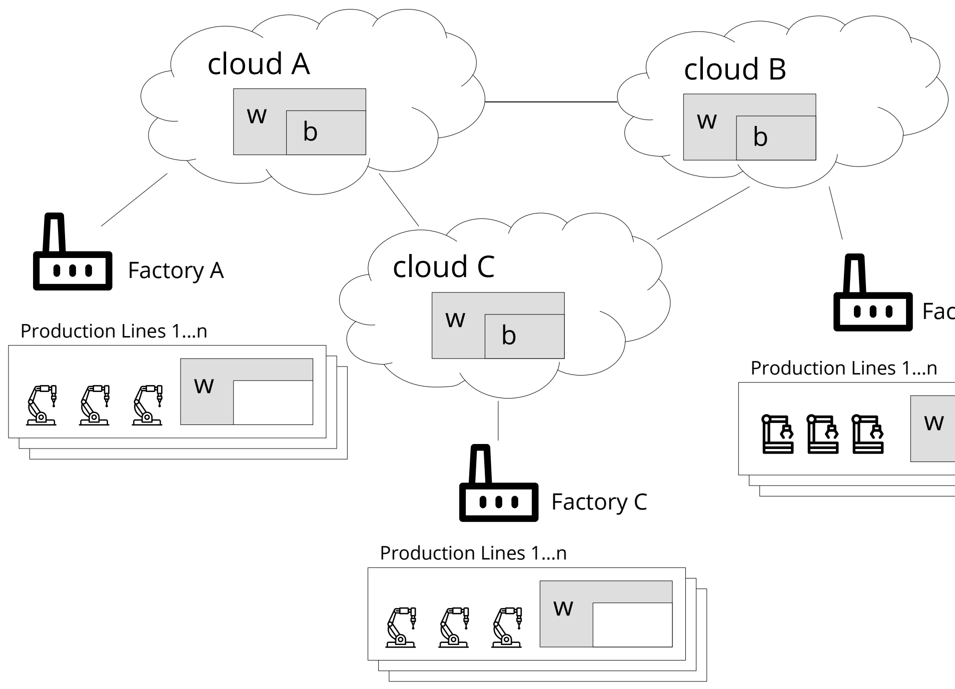


Figure 7.2: Industrial IoT scenario topology

The scenario can run in a dynamic or a static configuration. In the dynamic configuration, a scaling process monitors the network and regularly checks if new brokers should be spawned or existing should be shutdown. Furthermore, a coordinator monitors distances between clients and brokers and advises the clients to reconnect if a closer broker is available. In the static configuration, there is no scaling processes and brokers run at all production lines and in one cloud region. The two different configurations allow us to evaluate if we can save broker resources and still maintain the same performance by using the static configuration as baseline.

7.3 Results

7.3.1 EX-V1. Accuracy Evaluation

In each of the 20 regions, we create one cloud brokers and five edge brokers with one client each. This results in a total of 20 cloud brokers, 100 edge brokers and 100 clients.

Figure 7.3 shows the difference between distances estimated from Vivaldi coordinates (y axis) compared to the distances between the corresponding nodes according to the distance distributions of the links between the nodes in the topology (x axis). About

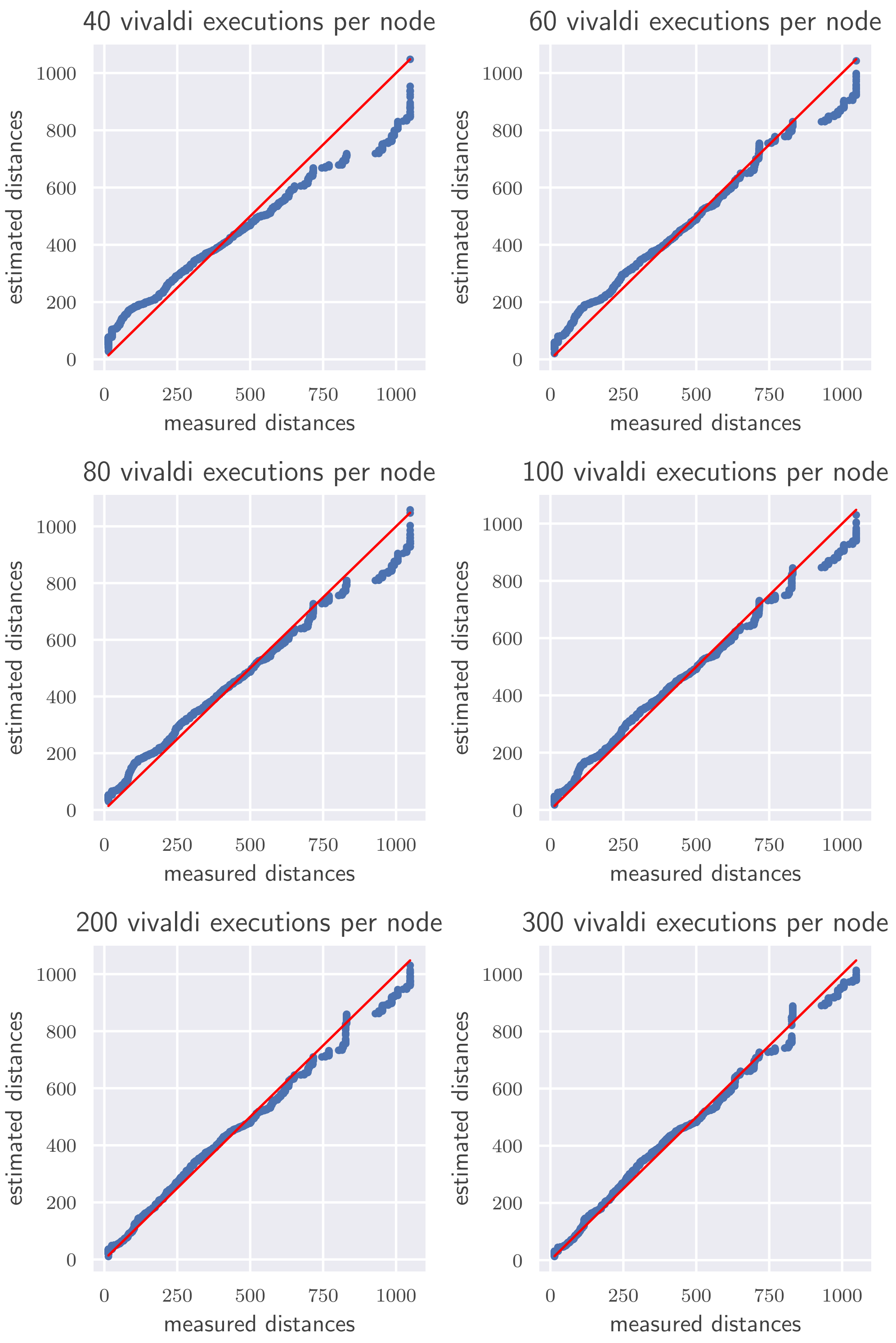


Figure 7.3: Q–Q plots comparing the distances as predicted by Vivaldi with the ground truth (the mode of the distance distributions) after different Vivaldi execution counts

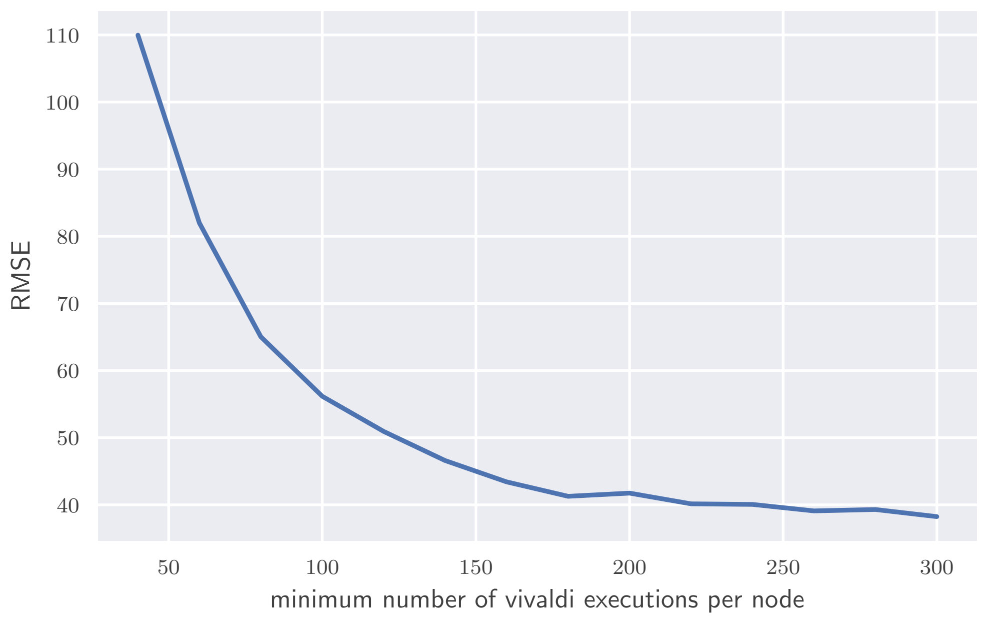


Figure 7.4: Plot of the root mean squared error (RMSE) of Vivaldi distances vs. measured distances

100–200 executions are sufficient to obtain coordinates that enable distance estimations that can be used to find neighbors.

Figure 7.4 shows the root mean squared error (RMSE) of distances estimated from Vivaldi coordinates versus actual distances:

$$\text{RMSE} = \sqrt{\frac{\sum_{x \in N, y \in N} (d_v(x,y) - d_m(x,y))^2}{|N|^2}}$$

where N is the set of nodes, d_v a function that returns the distance between the coordiantes of two nodes and d_m the measured distance between two nodes.

During the first 150—200 executions, the error declines rapidly. After that, we do not observe significant improvements any more.

7.3.2 EX-V2. Peer Selection Evaluation

We ran experiments for the following configurations. Each configuration contains 10 workers, to each of which the client performs 5 measurements per iteration.

- 5 static cloud workers, 5 static edge workers

- 5 dynamic cloud workers, 5 dynamic edge workers
- 5 dynamic cloud workers, 5 closest workers
- 5 dynamic edge workers, 5 closest workers
- 5 dynamic workers, 5 closest workers
- 4 dynamic workers, 6 closest workers

For each configuration, we show two graphs (see Figures 7.5 and 7.6): On the left, we show the root mean squared error (RMSE) of the clients' distance to the closest broker according to Vivaldi coordinate distance compared to the distance to the actually closest broker after each loop run. On the right, we plot the percentage of clients of which the region and neighbor derived from the Vivaldi coordinates matches the actual neighbors or region, respectively.

The configurations that only uses a static set of brokers did not yield accurate neighbor prediction. This happens due to the low diversity of peers that Vivaldi can use to calculate coordinates. Therefore, we show only one static-only configuration (Figure 7.5a). All other configurations are dynamic, i.e., they use a new set of workers in each iteration.

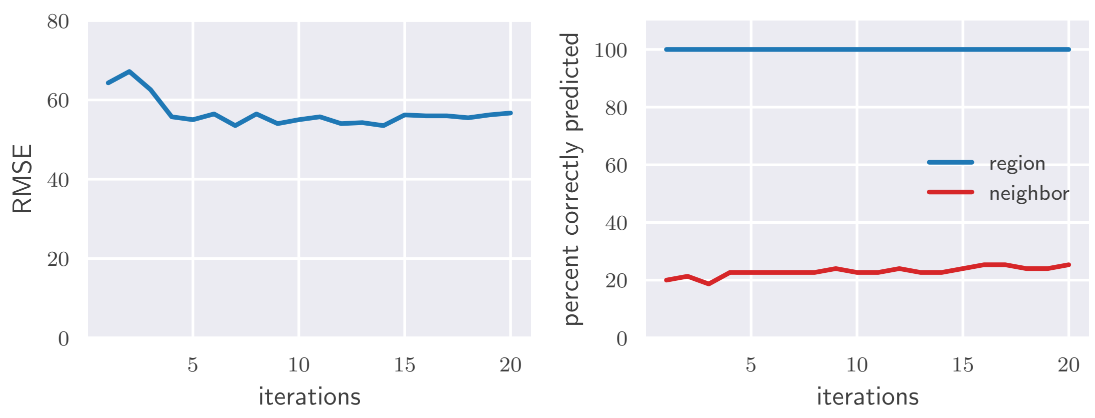
Investigation of possible dynamic configurations involved a number of configurations that use cloud and edge brokers. Figure 7.5b exemplary shows one such configuration. These configurations did lead to better prediction of neighbors, but never found the correct neighbor.

To improve the detection of the neighbors, we added the closest brokers to the peer selection. Each iteration, the client gateway selects the five closest workers as estimated from the Vivaldi coordinates at that point in time. As soon as we included the closest workers, our algorithm predicted the neighbor correctly after varying numbers of iterations in all configurations. The configuration that uses 4 dynamic workers and 6 closest workers yielded the best results (Figure 7.6c). After only 3 iterations, the neighbor as found using Vivaldi coordinates was the actual neighbor.

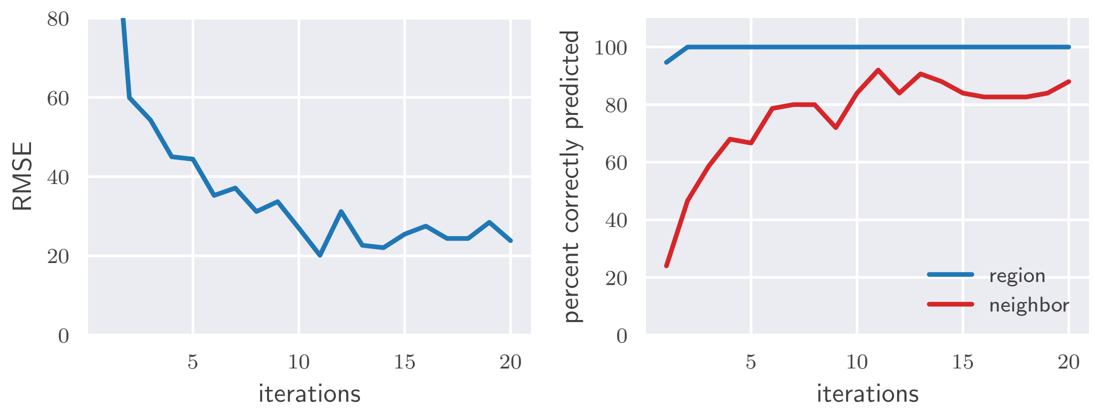
7.3.3 EX-S1. Static EMMA Scenario

Since we use the EMMA scenario and its result as baseline for our evaluation, we ran the exact same scenario without Vivaldi coordinates. We use regular measurements to all other nodes as implemented in the EMMA paper evaluation. Figure 7.7 shows the mean end-to-end latency of messages grouped by topic.

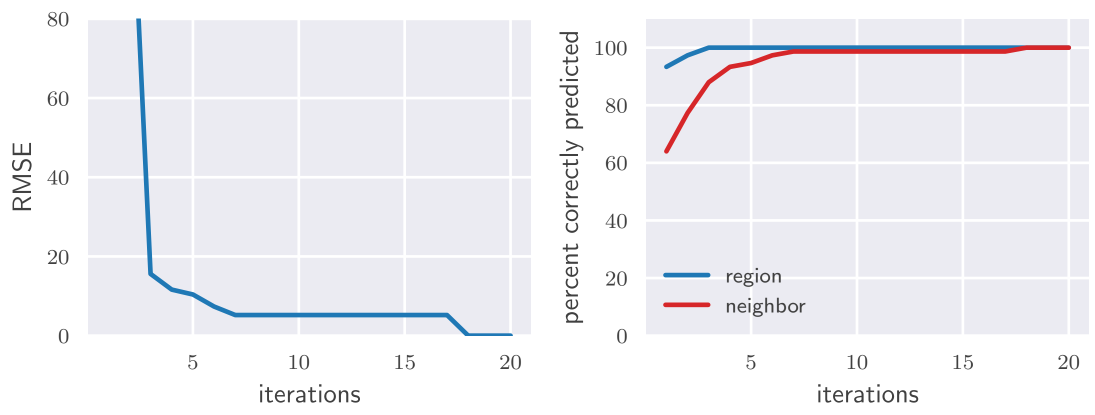
The upper two Figures 7.7a and 7.7b compare the results from the EMMA paper with our simulation without Vivaldi. In this experiment setup, the coordinator periodically requested all client gateways to measure distances to all brokers, as described in the EMMA paper. This allows us to determine if our baseline scenario from EMMA reproduces the same results in our simulation environment.



(a) 5 static cloud workers, 5 static edge workers

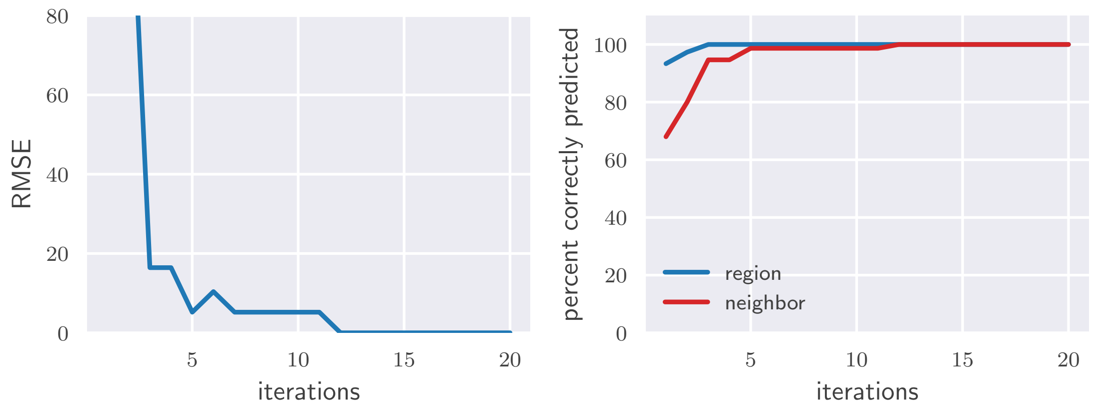


(b) 5 dynamic cloud workers, 5 dynamic edge workers

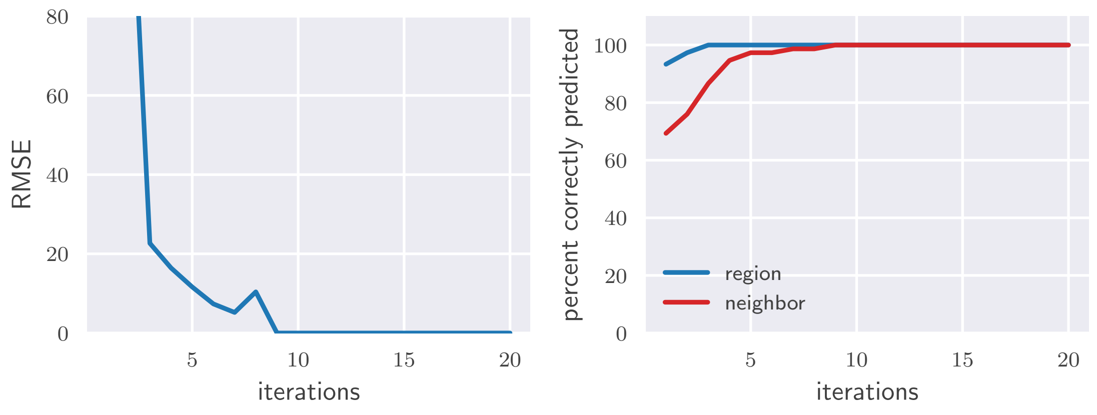


(c) 5 dynamic cloud workers, 5 closest workers

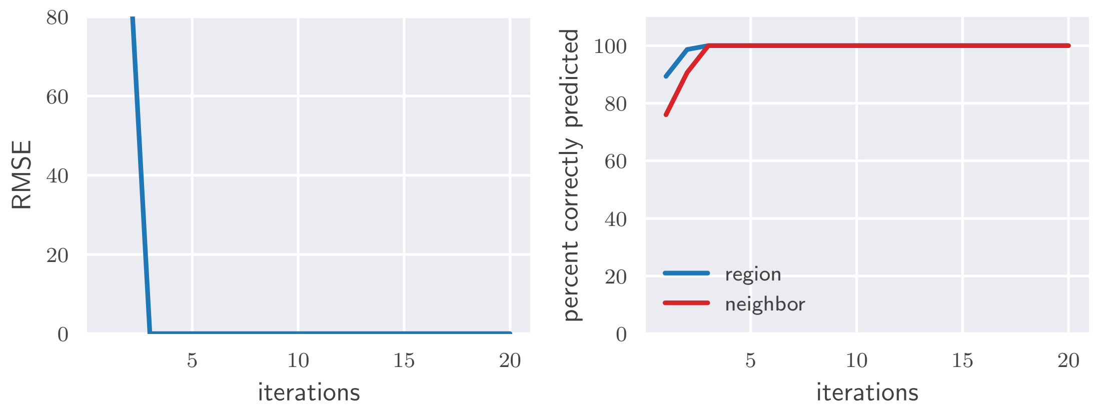
Figure 7.5: Evaluation results of different peer selection scenarios



(a) 5 dynamic edge workers, 5 closest workers

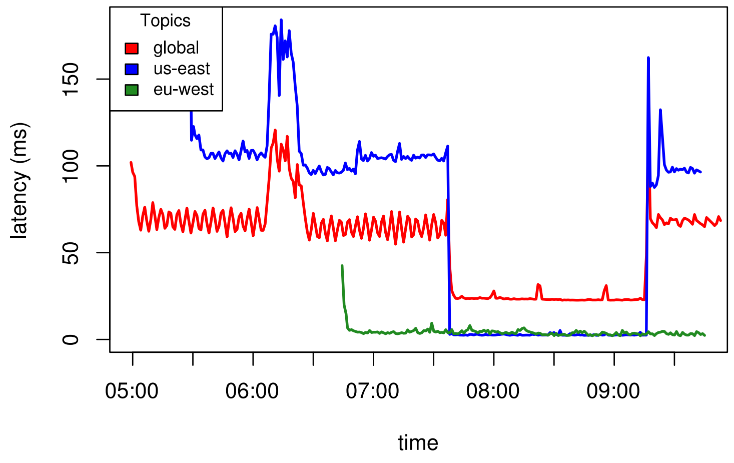


(b) 5 dynamic workers, 5 closest workers

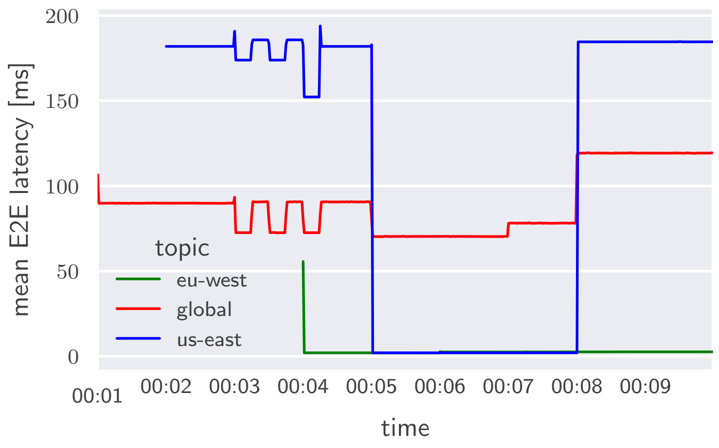


(c) 4 dynamic workers, 6 closest workers

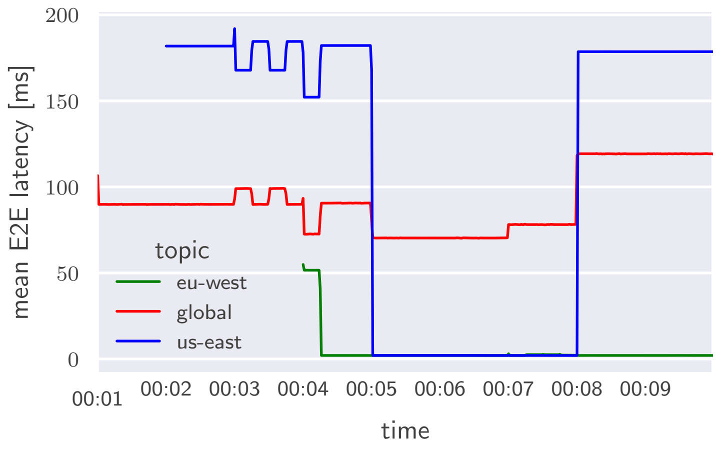
Figure 7.6: Evaluation results of different peer selection scenarios (ctd.)



(a) Results from EMMA paper



(b) Results from our system without Vivaldi



(c) Results from our system with Vivaldi

Figure 7.7: Mean latency of messages grouped by topic (static EMMA scenario)

Table 7.2: Management Overhead

Scenario	% of message count	% of traffic
baseline	6.24	2.18
vivaldi	0.50	0.23

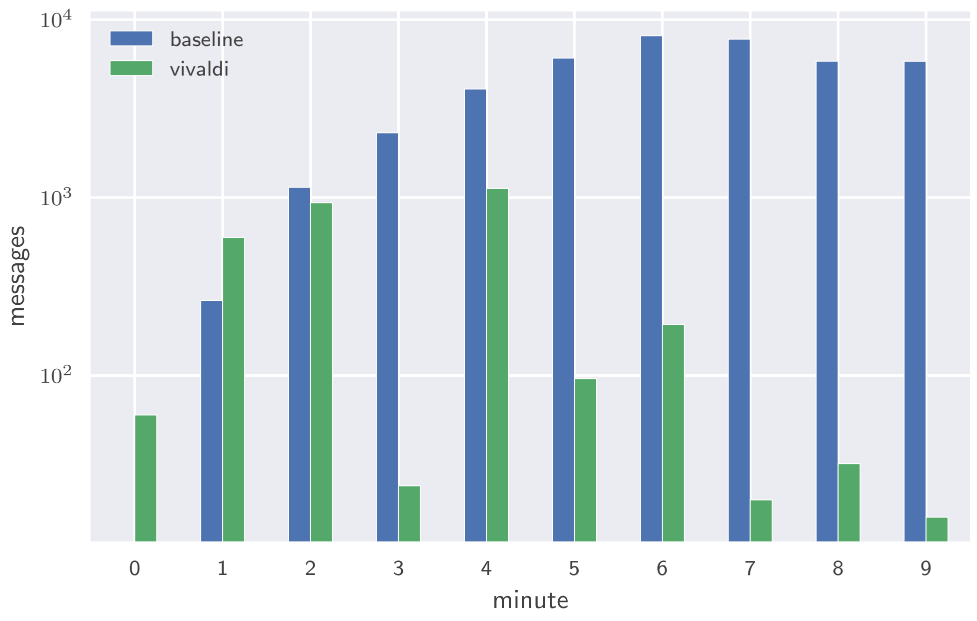


Figure 7.8: Management traffic in messages per minute in EMMA without and with Vivaldi

In the lowest Figure 7.7c, we show the results from our simulation with Vivaldi enabled. Compared to Figure 7.7b, we can see that the latency is similar, which means that the same broker selection decision is made as with measurements.

In Table 7.2, we show the overall share of monitoring messages in percent of the message count and traffic in bytes. Looking at the management overhead over time in Figure 7.8, the overhead rises with additional brokers and clients (minutes 1–4). The configuration without Vivaldi performs measurements to all brokers at a constant interval. Our implementation with Vivaldi coordinates uses an adaptive interval. The interval calculates from the relative error as returned by the Vivaldi algorithm. This results in a high interval for new coordinates which have a high relative error. As soon as the coordinates become more precise and the error decreases, the measurement interval increases and the management overhead is reduced by 1–2 orders of magnitude (minutes 5–9). Since client gateways still perform periodic measurements, but with a longer interval, we see a higher number of management overhead in the minutes 7–9.

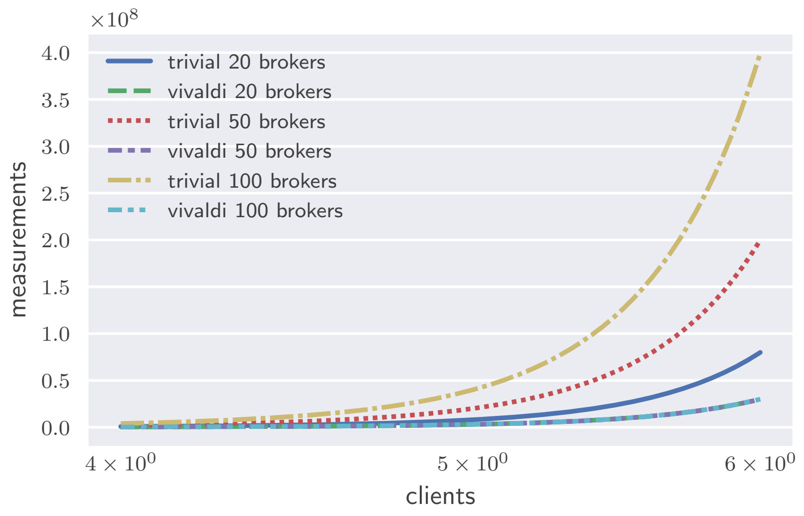


Figure 7.9: Distance measurements in the first minute using the trivial approach vs. Vivaldi coordinates with different broker counts

In general, the more brokers there are in the network, the higher the benefit of using coordinates to estimate distances compared to the trivial approach with distance measurements to all brokers. Figure 7.9 shows this graphically. The graph shows the number of clients (logarithmically scaled x-axis) compared against the number of measurements. Using different broker counts, we can see that the number of measurements a client has to perform rises with each additional broker with the trivial approach. On the contrary, using Vivaldi to estimate distances based on coordinates, the number of measurements is limited to 10 brokers per client.

7.3.4 EX-S2. Dynamic EMMA Scenario

In the dynamic scenario, the scaling mechanism works as expected. Figure 7.10 shows the active brokers each minute of the experiment run. The first two configurations have dynamic scaling disabled (*static*) or enabled (*dynamic*). In an additional third configuration (*dynamic (all workers)*), we create all workers at the beginning of the scenario run and do not stop any workers during the experiment. This results in a different behavior compared to the static scenario:

- The broker in *us-east* is started earlier (minute 2, when a client group appears in its region).
- The second broker in *eu-west* is never started.
- The broker in *us-east* is not stopped.

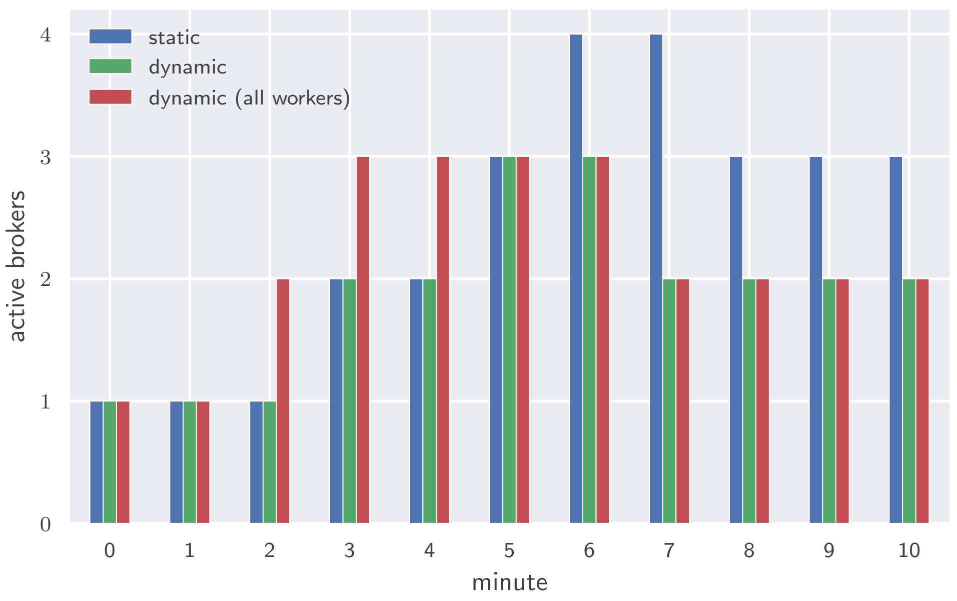


Figure 7.10: Number of active brokers in static vs. dynamic EMMA scenario.

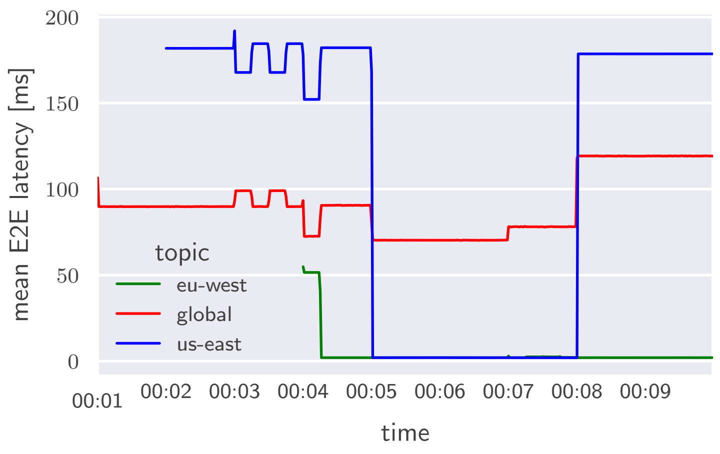
In Figure 7.11, we show the mean end-to-end latency by topic, compared to the static EMMA scenario with Vivaldi. Comparing Figure 7.11a and Figure 7.11b, we see that the end-to-end latencies of all topics are the same with the dynamically deployed brokers. We can observe a consequence of the different behavior: The latency of the *global* topic does not increase after minute 8 since the broker in *us-east* is not stopped.

With all brokers available from the beginning (Figure 7.11c), the mean latency for the topic *us-east* drops already at minute 4, when the scaling mechanism spawns the broker in *us-east*.

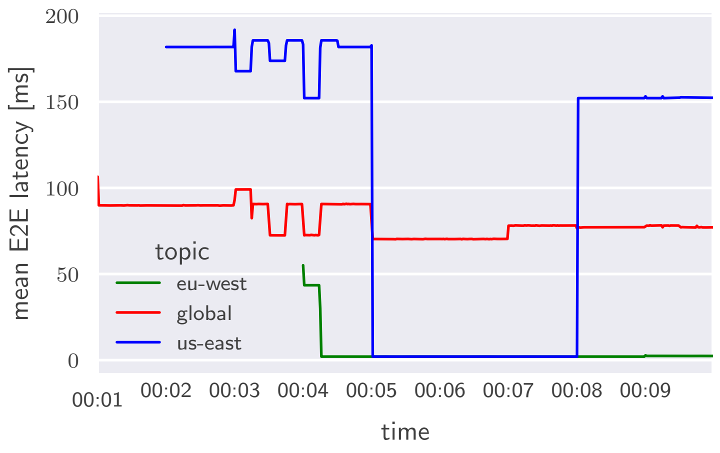
7.3.5 EX-S3. Industrial IoT Scenario

The results of this scenario (Figure 7.12) compare the *static* and *dynamic* configurations. In the upper plot, we show the mean end-to-end latency for all messages. Since all edge brokers are always running in the *static* configuration, the mean latency is minimal (close to zero). In the *dynamic* configuration, we see short latency spikes every time a factory gets started. Shortly after, the scaling mechanism deploys a new broker and the latency drops to the near-zero latency that we see with the static configuration.

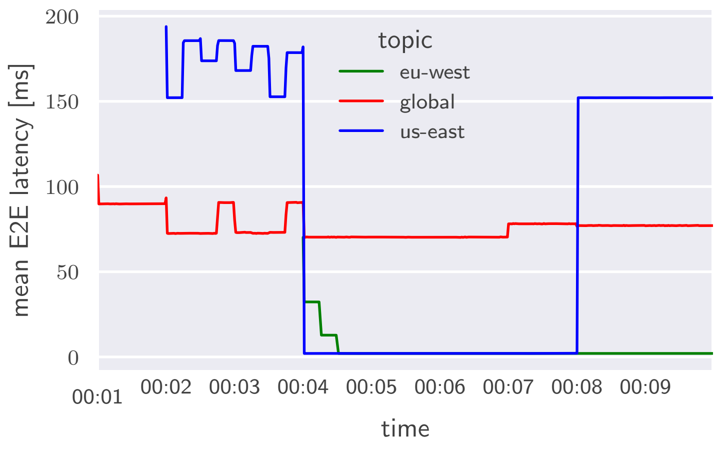
The lower plot shows the number of active brokers, together with the number of active factories (dashed line). We start the scenario with one cloud broker, but with three active factories in the beginning, the scaling mechanism immediately starts three edge brokers. Then, as we stop and start factories, we see that the scaling mechanism scales down and up correctly. Note that there is always one more broker running because we do not scale down the cloud broker in this scenario.



(a) No dynamic scaling



(b) Dynamic scaling, workers added over time



(c) Dynamic scaling, all workers available from the beginning

Figure 7.11: Mean latency of messages grouped by topic (static EMMA scenario)

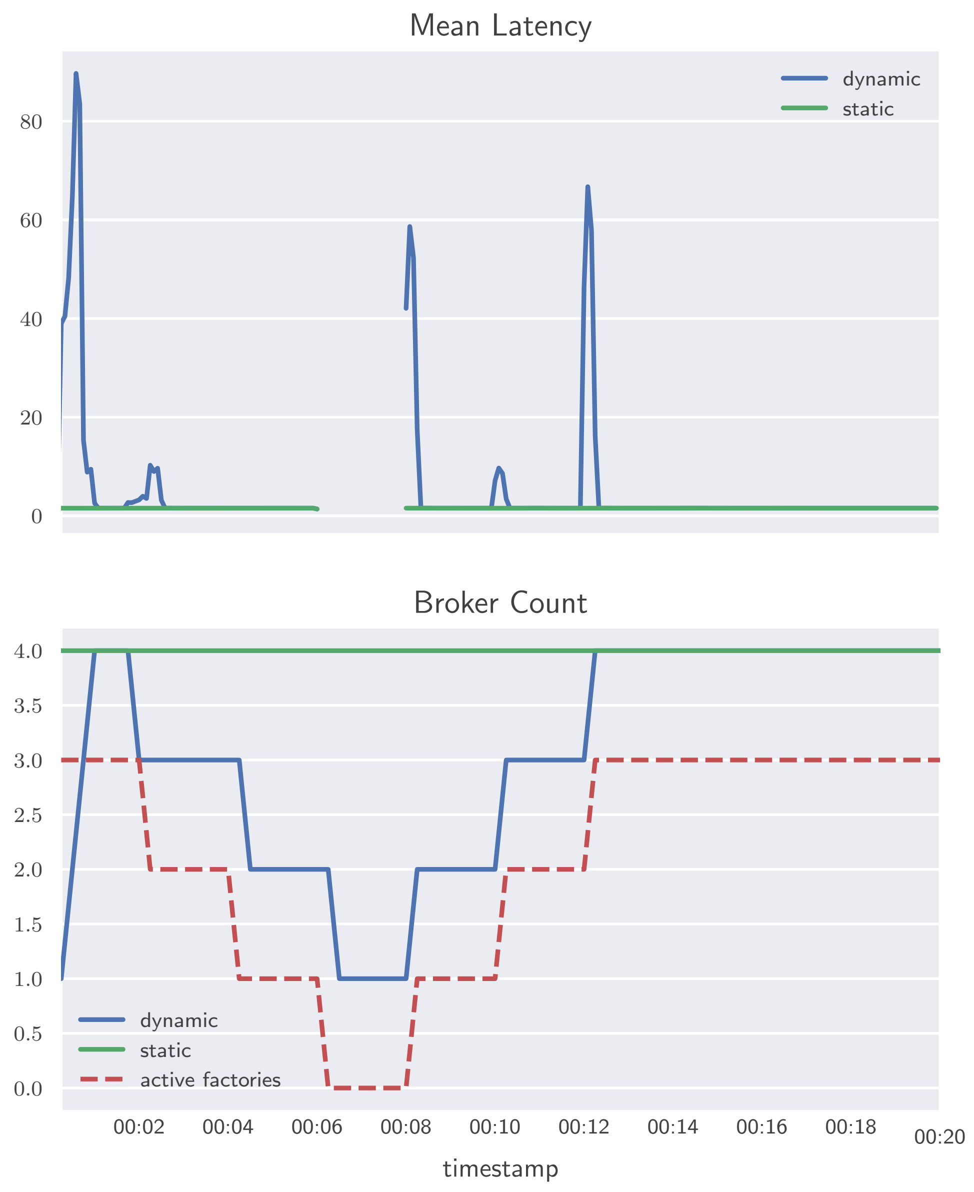


Figure 7.12: Results from the Industrial IoT scenario that show the impact of dynamic broker scaling.

CHAPTER 8

Discussion

In the following, we discuss the results presented in Chapter 7. First, we discuss the results of the experiments in the context of broker selection. After that, we deal with the evaluation findings of the self-adaptive scaling mechanism. Finally, we will consider limitations of our approach.

8.1 Broker Discovery

The first experiments (results in Figures 7.3 and 7.4) show that Vivaldi can be used as network coordinate system in a distributed broker network. In the results of the experiments to find an appropriate peer selection for distance measurements (Figures 7.5 and 7.6), we see that it is possible to reliably find the closest broker as long as the clients use an appropriate peer selection strategy.

Next, we compare our implementation of broker discovery using network coordinates to the evaluation results of the EMMA paper (Figure 7.7). As soon as the closer brokers become available for clients, they find the brokers as fast as with the trivial broker selection approach from EMMA. At the same time, we see that the management overhead can be drastically reduced when the clients do not have to measure distances to all available brokers (Figure 7.8). Only at the beginning of the scenario, the management overhead was slightly higher due to distance measurements between brokers that are not necessary without a coordinate system.

Compared to the trivial broker selection approach where clients have to perform measurements to all brokers, the number of measurements per client has an upper bound in our implementation. Figure 7.9 shows the number of distance measurements in the first minute for different numbers of brokers. The peer selection method we chose (4 random workers, 6 closest workers) results in at most 10 workers per minute. Additionally, we reduce the frequency of measurements depending on the error of coordinates. This leads to a very small number of measurements as soon as the coordinate system converges to an equilibrium state.

8.2 Self-adaptive Scaling

We use the EMMA scenario to compare static provisioning with self-adaptive broker scaling to edge resources. The results show that the scaling mechanism detects growing client numbers close to edges correctly. As soon as the potential latency improvement rises above the defined threshold, the scaling process starts a new broker. We see this in the comparison between the static and dynamic EMMA scenarios (Figure 7.10) as well as in the industrial IoT scenario (Figure 7.12). Also, we clearly see the benefit of dynamic scaling in terms of resource consumption. Trivially, the latency would be optimal if we always ran all brokers (as we see in the static industrial IoT scenario). Except for the short latency peak before the scaling mechanism starts new brokers, we can achieve the same latency in the dynamic scenario, but save edge computing resources when factories are not running.

8.3 Limitations

In our system, we made some assumptions about the deployment environment that might impose limitations:

- It is always possible to run a gateway component close to the client.

  The gateway hides the dynamic connection handling from the client and is a central part of the middleware. This could become an issue with resource-constrained IoT devices that might not be able to run the gateway to connect to a broker. In that case, the gateway would have to run on another device close to the client that has the resources. Alternatively, we could deploy the gateway on an edge computing resource. This would be cheaper than running a broker but would still involve the deployment of an additional service. As long as the latency to the gateway is minimal (i.e., it resides on the same network edge and is closer or equally close to edge brokers), this will not impact the network optimization mechanisms.

- To keep an overview of the topology, the coordinator relies on edge workers to host a monitoring process that measures distances to other workers.

  A daemon keeps the coordinates of the workers up to date by performing regular measurements and submits them to the coordinator. If running these processes is not feasible or too expensive (e.g., because of high cost for pay-by-use resources or to reduce power consumption), it would suffice to run scheduled jobs instead. The job would regularly perform measurements in order to keep the coordinates stable and correct.

- Publishers and subscribers of a topic reside close to the same edge or at least the same cloud region.

  We aim to optimize the latency between publishers and subscribers by using additional brokers on edge resources. However, this will not reduce the latency between nodes that are not close to the newly deployed brokers. In this case, we have to improve Quality of Service by other means. For instance, we could ensure that the brokers between publishers and subscribers have sufficient bandwidth available.

- The scaling mechanism only considers one edge worker that can host one broker per edge site.

  When a broker forwards a message, it needs to iterate all subscriber's connections, performing multicast over unicast. With a growing number of subscribers, message brokers need more CPU time and bandwidth in order to handle the fan-out. If this becomes a limiting factor, we need to scale brokers horizontally, i.e., we need more than one broker per edge. Currently, our pressure calculation does not include CPU load or bandwidth. Therefore, the scaling mechanism will never deploy more than one broker. We also see this in the evaluation in Figure 7.10, where we never reach the broker number of the static scenario because the static scenario starts two brokers in *eu-west*.

- The coordinator can keep bridging tables on brokers in a consistent state.

  To forward messages to all interested subscribers, brokers hold bridging tables. The coordinator holds the global bridging table and distributes it to the brokers accordingly. This is required because if the coordinator reconfigures the broker network, it has to inform the brokers about the subscriptions. However, managing bridging tables across the broker network requires an additional mechanism that ensures consistency, e.g., by using distributed transactions.

CHAPTER 9

Conclusion

In this chapter, we will wrap up our work and evaluation results. We tried to solve two problems that arise in distributed broker networks that can be scaled to edge resources.

The first problem regards clients that want to connect to a broker. Those clients have to select an appropriate broker based on specific criteria. In latency-sensitive applications, the most important criterion is the network distance to the broker. Using distance measurements, clients can determine the closest broker. To reduce the management overhead and network stress incurred by frequent network measurements between nodes, we use a network coordinate system that enables the client to estimate distances based on a small number of measurements.

The second problem is the question of where we can automatically deploy additional brokers to further reduce the latency of clients. To that end, we use the distance information that we have already available from our network coordinates to monitor the impact if we scale out the broker network to a certain edge resource. We designed a mechanism that quantifies the possible latency improvement by using a model of osmotic pressure.

In the following, we will answer the research questions that we stated in Chapter 1.

RQ1. With a network coordinate system, what is the tradeoff between monitoring overhead and accuracy when finding the closest broker for a client? We ran multiple experiments in order to be able to answer this question. First, the accuracy of Vivaldi network coordinates is accurate enough to find the closest neighbor in terms of latency. We see the low RMSE between estimated and actual distances in the results of the accuracy evaluation (see Figures 7.3 and 7.4) after a certain number of measurements. The warm-up phase is much shorter in an already converged network of coordinates (for instance, when there are already cloud and edge workers present with stable, accurate coordinates). This reduces the number of measurements required to reach a low error level. In the evaluation of bootstrap mechanisms, we simulate the warm-up phase by pre-calculating coordinates for all workers. Results show that joining clients can reliably find the closest broker with a small number of measurements (see Figures 7.5 and 7.6). At the same time, we could decrease the monitoring overhead by at least one order of magnitude compared to the trivial approach, as we can see in Table 7.2 and Figure 7.8.

RQ2. What is the tradeoff between end-to-end latency and resource consumption when scaling message brokers to edge resources using self-adaptive broker scaling? We already know from the EMMA paper [RND18] that the deployment of additional brokers can improve the end-to-end latency. In our evaluation, we were successfully able to run an experiment that reproduces the results from the EMMA paper. Additionally, our work also shows that it is possible to define a threshold at which level of demand a new broker will be deployed. By doing so, we can decrease resource consumption in case there is no significant advantage in terms of latency improvement when scaling up the broker network. The results of the simulations show that scaling with the increase or decrease of demand has no impact on the overall latency. Only shortly after the factory starts its operation, the clients experience a brief latency peak. As soon as the scaling mechanism deploys an additional broker, the latency drops to a low level (see Figure 7.12).

RQ3. Which metrics and mechanisms are appropriate for reactive autoscaling decisions on distributed broker networks? The context of our work is to improve the performance of latency-sensitive edge computing applications. For this reason, we based our scaling mechanism on the distance between clients and edge or cloud computing resources. For up-scaling, our mechanism takes into account the possible distance reductions for clients if there was a closer broker available. Analogously, to determine whether we can remove a running broker, our mechanism checks the penalty for clients if the respective broker stopped. With the experiments, we were able to show that this mechanism scales brokers up and down in time (see Figure 7.12). The delay depends on the time when the scaling algorithm checks for possible optimizations. In our setup, we timed the point in time when the scaling job runs shortly after network changes, i.e., after a new factory starts or stops operation.

RQ4. How well does the self-adaptive mechanism cope with different publish/subscribe topologies of IoT applications? In addition to the scenario from the EMMA paper, we synthesized an Industrial IoT scenario using Ether [RLF$^+$20]. We were able to show that the scaling mechanism works in both scenarios. In Figure 7.10, we see that the scaling mechanism takes the same scaling decisions as the manual scaling steps in the scenario: As soon as there additional edge workers become available and clients benefit from an additional broker there, the scaling mechanism starts the broker. Also, in the Industrial IoT scenario, the scaling mechanism scales up and down depending on the current demand (see Figure 7.12). In our scenarios, the scaling works well because of the high proximity of publishers and subscribers. In different topologies where publishers and subscribers of a single topic span across multiple regions, the network nodes might not benefit from scaling brokers to the edge. In this case, the end-to-end latency would be the same because of the long network distance between publishers and subscribers. Since we can not improve latency in these situations, we assumed that publishers and subscribers reside in the same region, where we can improve QoS by deploying local edge brokers.

9.1 Future Work

- The coordinator is a central component in our system which might become a bottleneck in larger topologies. To scale the coordinator, it is possible to partition the coordinator's authority by region. Clients and workers would connect to the coordinator of their region (e.g., using anycast [KW00, BF05]). The coordinator would only monitor and optimize the network within its own region. This requires that each region hosts at least one broker because the scaling process of the coordinator only manages cloud and edge workers of their region.

- When calculating the osmotic pressure, we do not consider the number of clients connected to a particular publish/subscribe topic in order to balance the load between brokers. Approaches like MultiPub [GSKK17] do topic-based load balancing. In combination with our current osmotic pressure mechanism, we could calculate a pressure value for each topic. In addition, we could also include the broker load (in terms of bandwidth or CPU load) in the pressure calculation.

- Currently, all workers perform measurements to all other workers in order to have stable coordinates that allow precise distance estimations from clients. With a growing number of workers, this might not be feasible anymore. Workers could select only a subset of the other workers similar to the measurement peer selection of the clients.

- In our evaluation, we designed a dynamic Industrial IoT scenario, where clients join and leave the network, changing osmotic pressure over time. However, the scenario does not include mobile clients. An additional scenario could consider clients that disconnect from the network and reconnect at another edge. This would allow us to evaluate the scaling algorithm in a topology where not only the presence of clients changes, but also their coordinates.

List of Figures

Bibliography

[AH16] Sherif Abdelwahab and Bechir Hamdaoui. FogMQ: A Message Broker System for Enabling Distributed, Internet-Scale IoT Applications over Heterogeneous Cloud Platforms. *arXiv:1610.00620 [cs]*, October 2016. arXiv: 1610.00620.

[BF05] Hitesh Ballani and Paul Francis. Towards a global IP anycast service. *ACM SIGCOMM Computer Communication Review*, 35(4):301–312, August 2005.

[BMZA12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh 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*, MCC '12, pages 13–16, New York, NY, USA, August 2012. Association for Computing Machinery.

[BSF$^+$17] R. Banno, J. Sun, M. Fujita, S. Takeuchi, and K. Shudo. Dissemination of edge-heavy data on heterogeneous MQTT brokers. In *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*, pages 1–7, September 2017.

[BWVK18] Kyle E. Benson, Guoxi Wang, Nalini Venkatasubramanian, and Young-Jin Kim. Ride: A Resilient IoT Data Exchange Middleware Leveraging SDN and Edge Cloud Resources. In *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 72–83, April 2018.

[CMTV07] Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. SpiderCast: A Scalable Interest-aware Overlay for Topic-based Pub/Sub Communication. In *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*, DEBS '07, pages 14–25, New York, NY, USA, 2007. ACM. event-place: Toronto, Ontario, Canada.

[CRW00] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 219–227, New York, NY, USA, July 2000. Association for Computing Machinery.

[CSS$^+$16] S. Chun, S. Shin, S. Seo, S. Eom, J. Jung, and K. Lee. A Pub/Sub-Based Fog Computing Architecture for Internet-of-Vehicles. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 90–93, December 2016.

[DBBR19] Sujata Dash, Sitanath Biswas, Debajit Banerjee, and Atta UR Rahman. Edge and Fog Computing in Healthcare – A Review. *Scalable Computing: Practice and Experience*, 20(2):191–206, May 2019.

[DCKM04] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A Decentralized Network Coordinate System. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 15–26, New York, NY, USA, 2004. ACM. event-place: Portland, Oregon, USA.

[DGST11] Schahram Dustdar, Yike Guo, Benjamin Satzger, and Hong-Linh Truong. Principles of Elastic Processes. *IEEE Internet Computing*, 15(5):66–71, September 2011.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[FJPa99] P. Francis, S. Jamin, V. Paxson, and and D. F. Gryniewicz and. An architecture for a global Internet host distance estimation service. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*, volume 1, pages 210–217 vol.1, March 1999.

[GEvS08] Paweł Garbacki, Dick H. J. Epema, and Maarten van Steen. Broker-placement in latency-aware peer-to-peer networks. *Computer Networks*, 52(8):1617–1633, June 2008.

[GSGKK15] J. Gascon-Samson, F. Garcia, B. Kemme, and J. Kienzle. Dynamoth: A Scalable Pub/Sub Middleware for Latency-Constrained Applications in the Cloud. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 486–496, June 2015. ISSN: 1063-6927.

[GSK$^+$18] J. Gedeon, M. Stein, J. Krisztinkovics, P. Felka, K. Keller, C. Meurisch, L. Wang, and M. Mühlhäuser. From Cell Towers to Smart Street Lamps: Placing Cloudlets on Existing Urban Infrastructures. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 187–202, October 2018.

[GSKK17] J. Gascon-Samson, J. Kienzle, and B. Kemme. MultiPub: Latency and Cost-Aware Global-Scale Cloud Publish/Subscribe. In *2017 IEEE 37th*

International Conference on Distributed Computing Systems (ICDCS), pages 2075–2082, June 2017. ISSN: 1063-6927.

[HB20] Daniel Happ and Suzan Bayhan. On the impact of clustering for IoT analytics and message broker placement across cloud and edge. In *Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '20, pages 43–48, New York, NY, USA, April 2020. Association for Computing Machinery.

[HSTB20] Jonathan Hasenburg, Florian Stanek, Florian Tschorsch, and David Bermbach. Managing Latency and Excess Data Dissemination in Fog-Based Publish/Subscribe Systems. In *2020 IEEE International Conference on Fog Computing (ICFC)*, pages 9–16, April 2020.

[JCL+10] Hans-Arno Jacobsen, Alex Cheung, Guoli Li, Balasubramaneyam Maniymaran, Vinod Muthusamy, and Reza Sherafat Kazemzadeh. The PADRES Publish/Subscribe System. *Principles and Applications of Distributed Event-Based Systems*, pages 164–205, 2010.

[KSZ+18a] Shweta Khare, Hongyang Sun, Kaiwen Zhang, Julien Gascon-Samson, Aniruddha Gokhale, and Xenofon Koutsoukos. Ensuring Low-Latency and Scalable Data Dissemination for Smart-City Applications. In *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 283–284, April 2018.

[KSZ+18b] Shweta Khare, Hongyang Sun, Kaiwen Zhang, Julien Gascon-Samson, Aniruddha Gokhale, Xenofon Koutsoukos, and Hamzah Abdelaziz. Scalable Edge Computing for Low Latency Data Dissemination in Topic-Based Publish/Subscribe. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 214–227, October 2018.

[KW00] Dina Katabi and John Wroclawski. A framework for scalable global IP-anycast (GIA). *ACM SIGCOMM Computer Communication Review*, 30(4):3–15, August 2000.

[LBMAL14] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing*, 12(4):559–592, December 2014.

[LGP+05] Eng Keong Lua, Timothy Griffin, Marcelo Pias, Han Zheng, and Jon Crowcroft. On the accuracy of embeddings for internet coordinate systems. In *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*, pages 11–11, 2005.

[LGS07] Onathan Ledlie, Paul Gardner, and Margo Seltzer. Network Coordinates in the Wild. 2007.

[LMP$^+$21] Ivan Lujic, Vincenzo De Maio, Klaus Pollhammer, Ivan Bodrozic, Josip Lasic, and Ivona Brandic. Increasing Traffic Safety with Real-Time Edge Analytics and 5G. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '21, pages 19–24, New York, NY, USA, April 2021. Association for Computing Machinery.

[LRCM22] Edoardo Longo, Alessandro E.C. Redondi, Matteo Cesana, and Pietro Manzoni. BORDER: a Benchmarking Framework for Distributed MQTT Brokers. *IEEE Internet of Things Journal*, pages 1–1, 2022.

[LXZ15] Shancang Li, Li Da Xu, and Shanshan Zhao. The internet of things: a survey. *Information Systems Frontiers*, 17(2):243–259, April 2015.

[MMKB07] Shruti P. Mahambre, Sd Madhu Kumar, and Umesh Bellur. A taxonomy of QoS-aware, adaptive event-dissemination middleware. 2007.

[Nai17] Nitin Naik. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In *2017 IEEE International Systems Engineering Symposium (ISSE)*, pages 1–7, October 2017.

[NZ02] T. S. Eugene Ng and Hui Zhang. Predicting Internet network distance with coordinates-based approaches. In *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 170–179 vol.1, June 2002.

[OR11] Melih Onus and Andréa W. Richa. Minimum Maximum-degree Publish-subscribe Overlay Network Design. *IEEE/ACM Trans. Netw.*, 19(5):1331–1343, October 2011.

[PB02] P.R. Pietzuch and J.M. Bacon. Hermes: a distributed event-based middleware architecture. In *Proceedings 22nd International Conference on Distributed Computing Systems Workshops*, pages 611–618, July 2002.

[QCZ$^+$20] Tie Qiu, Jiancheng Chi, Xiaobo Zhou, Zhaolong Ning, Mohammed Atiquzzaman, and Dapeng Oliver Wu. Edge Computing in Industrial Internet of Things: Architecture, Advances and Challenges. *IEEE Communications Surveys & Tutorials*, 22(4):2462–2488, 2020. Conference Name: IEEE Communications Surveys & Tutorials.

[Rag15] Dave Raggett. The Web of Things: Challenges and Opportunities. *Computer*, 48(5):26–32, May 2015.

[RDR18] T. Rausch, S. Dustdar, and R. Ranjan. Osmotic Message-Oriented Middleware for the Internet of Things. *IEEE Cloud Computing*, 5(2):17–25, March 2018.

[RGPH11] Fatemeh Rahimian, Sarunas Girdzijauskas, Amir H. Payberah, and Seif Haridi. Vitis: A Gossip-based Hybrid Overlay for Internet-scale Publish/-Subscribe Enabling Rendezvous Routing in Unstructured Overlay Networks. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 746–757, May 2011. ISSN: 1530-2075.

[RLF$^+$20] Thomas Rausch, Clemens Lachner, Pantelis A. Frangoudis, Philipp Raith, and Schahram Dustdar. Synthesizing Plausible Infrastructure Configurations for Evaluating Edge Computing Systems. 2020.

[RND18] T. Rausch, S. Nastic, and S. Dustdar. EMMA: Distributed QoS-Aware MQTT Middleware for Edge Computing Applications. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 191–197, April 2018.

[Sat17] Mahadev Satyanarayanan. The Emergence of Edge Computing. *Computer*, 50(1):30–39, January 2017. Conference Name: Computer.

[SB18] S. Sen and A. Balasubramanian. A highly resilient and scalable broker architecture for IoT applications. In *2018 10th International Conference on Communication Systems Networks (COMSNETS)*, pages 336–341, January 2018. ISSN: 2155-2509.

[SCZ$^+$16] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5):637–646, October 2016.

[SLSB19] Ali Hassan Sodhro, Zongwei Luo, Arun Kumar Sangaiah, and Sung Wook Baik. Mobile edge computing based QoS optimization in medical healthcare applications. *International Journal of Information Management*, 45:308–318, April 2019.

[SPvS04] M. Szymaniak, G. Pierre, and M. van Steen. Scalable cooperative latency estimation. In *Proceedings. Tenth International Conference on Parallel and Distributed Systems, 2004. ICPADS 2004.*, pages 367–376, July 2004. ISSN: 1521-9097.

[Sta14] John A. Stankovic. Research Directions for the Internet of Things. *IEEE Internet of Things Journal*, 1(1):3–9, February 2014.

[SvSVV12] Vinay Setty, Maarten van Steen, Roman Vitenberg, and Spyros Voulgaris. PolderCast: Fast, Robust, and Scalable Architecture for P2P Topic-based Pub/Sub. In *Proceedings of the 13th International Middleware Conference*, Middleware '12, pages 271–291, New York, NY, USA, 2012. Springer-Verlag New York, Inc. event-place: ontreal, Quebec, Canada.

[VFD$^+$16] Massimo Villari, Maria Fazio, Schahram Dustdar, Omer Rana, and Rajiv Ranjan. Osmotic Computing: A New Paradigm for Edge/Cloud Integration. *IEEE Cloud Computing*, 3(6):76–83, November 2016.

[VS17] M Veeramanikandan and Suresh Sankaranarayanan. Publish/subscribe broker based architecture for fog computing. In *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, pages 1024–1026, August 2017.

[VSDTD12] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. Cloudlets: bringing the cloud to the mobile user. In *Proceedings of the third ACM workshop on Mobile cloud computing and services*, MCS '12, pages 29–36, New York, NY, USA, June 2012. Association for Computing Machinery.

[XHL14] Li Da Xu, Wu He, and Shancang Li. Internet of Things in Industries: A Survey. *IEEE Transactions on Industrial Informatics*, 10(4):2233–2243, November 2014.

[YSAAH17] Muneer Bani Yassein, Mohammed Q. Shatnawi, Shadi Aljwarneh, and Razan Al-Hatmi. Internet of Things: Survey and open issues of MQTT protocol. In *2017 International Conference on Engineering & MIS (ICEMIS)*, pages 1–6, May 2017. ISSN: 2575-1328.

[ZJ13] Kaiwen Zhang and Hans-Arno Jacobsen. SDN-like: The Next Generation of Pub/Sub. *arXiv:1308.0056 [cs]*, July 2013. arXiv: 1308.0056.

[ZL20] Jun Zhang and Khaled B. Letaief. Mobile Edge Intelligence and Computing for the Internet of Vehicles. *Proceedings of the IEEE*, 108(2):246–261, February 2020. Conference Name: Proceedings of the IEEE.

[ZMK+15] Ben Zhang, Nitesh Mor, John Kolb, Douglas S. Chan, Ken Lutz, Eric Allman, John Wawrzynek, Edward Lee, and John Kubiatowicz. The Cloud is Not Enough: Saving {IoT} from the Cloud. 2015.