Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Henrik Sang Midtun

Auction-Based Task Assignment in Fog Computing Architectures with Publish-Subscribe Messaging Patterns

Master's thesis in ICT & Machine Technology
Supervisor: Amund Skavhaug
Co-supervisor: Vesa Hirvisalo
January 2022

NTNU
Norwegian University of
Science and Technology

Nordic Hub IIOT

Henrik Sang Midtun

Auction-Based Task Assignment in Fog Computing Architectures with Publish-Subscribe Messaging Patterns

Master's thesis in ICT & Machine Technology
Supervisor: Amund Skavhaug
Co-supervisor: Vesa Hirvisalo
January 2022

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Abstract

This thesis presents a decentralised task assignment method that uses auctioning mechanisms to propagate requests. It is designed to be implemented on top of the MQTT protocol and to be used in a fog computing network. The use-case is to route a request from an edge client to a fog node that is suitable for processing the requested service. The auction based task assignment method behaves such that a node that is incapable of handling a requested service invites its neighbouring fog nodes to participate in auctions. The neighbours place a bid in the auction that is proportional to their capabilities of providing the service. By propagating the request to the highest bidders, this approach finds the most optimal next step in the propagation based on the information that is contained in the bid. The feasibility of the method was proven by implementing a working demonstration program. The performance of the program was tested against two other methods, a modified version of the system outlined by Battistoni et al. in [1] and a random approach to request propagation. A limitation of the demonstration program is that the sizing of the bids was based on an expression that solely considers the immediate processing capability of the node in question. The method was shown to have better performance for some examples where the capabilities of nodes that would be part of a re-auction were included as a component in the bid. The main contribution of this thesis is *this selective method for propagating messages in an MQTT network that contains multiple brokers based on autonomous collaboration.*

Sammendrag

Denne masteroppgaven beskriver et desentralisert system for å delegere forespørsler ved hjelp av auksjoner. Metoden er designet for MQTT protokollen og har fog computing nettverk som tenkt bruksområde. Formålet med metoden er å videreføre forespørsler fra en edge klient til en fog node som er en god kandidat for å utføre den aktuelle tjenesten. Auksjonsmetoden fungerer slik at når en node får en forespørsel som den ikke burde utføre selv så inviterer den sine nabonoder til en auksjon. Naboene byr på forespørselen i henhold til deres evne til å håndtere forespørselen. Ved å videresende ansvaret for forespørselen til de høystbydende naboene så sørger auksjonsmetoden for at de best mulige valgene blir tatt ut ifra den tilgjengelige informasjonen som er innbakt i budene. Den praktiske gjennomførbarheten ved auksjonsmetoden ble vist ved å lage et fungerende demonstrasjonsprogram. Ytelsen til auksjonsmetoden ble testet opp imot en modifisert utgave av Battistoni et al. sitt verk i [1] og en tilfeldig tilnærming til delegering. En mangel ved demonstrasjonsprogrammet er at størrelsen på budene er utregnet med et uttrykk som utelukkende tar i betraktning nodenes evne til å utføre den forespurte tjenesten. Auksjonsmetoden er igjennom eksempler vist til å yte bedre når den også tar hensyn til de forventede budene fra nodene som vil være en del av neste budrunde. Bidraget er gitt ved å fremlegge *denne selektive metoden for å delegere meldinger i et MQTT nettverk som inneholder flere brokere ved hjelp av autonomt samarbeid.*

Contents

Figures

Tables

Chapter 1

Introduction

The need for remotely available Infrastructure as a Service (IaaS) products has been made clear by the emergence and quick adoption of cloud technology. The dawn of the COVID-19 pandemic in March 2020 brought cloud computing into the households of the general public as more and more had to leave their offices and work from home. The transition to the home-office meant getting used to daily interaction with remote computing services for video conferencing, VPNs, project management, and other work-at-home solutions. The cloud has become a part of everyday life for many people. The digital industry has been employing the cloud for a long time and are using it to replace local IT infrastructure either partially or fully. Cloud computing has overshadowed other remote computing paradigms for some time now, but recently there is much talk about moving towards the edge. That means moving closer to the application logic and has been a natural step because the cost of computing power is lower than ever. Despite hardware historically being at the lowest price-performance ratio there are still benefits to be had from remote computing. However, the decreased cost of computing resources provide greater opportunities for highly distributed systems that place powerful hardware closer to the locations of the consumers.

The services that are provided by the cloud are naturally centralised and that is a downside in certain applications. Any system with limited network capabilities are especially at a deficit because they can not reliably connect directly to a distant data centre. Many such systems that are traditionally constrained in the network department belong in the industrial and Internet of Things (IoT) landscape. These are comprised of low-cost edge devices that require remote computing to perform the tasks that they are unsuited for because of their limited capabilities. The *fog computing technology* aims to solve this challenge by introducing an intermediary layer between the edge and the cloud. This intermediary layer, which is commonly referred to as just the fog, is a distributed network of fog nodes that will be closer to the edge devices than any data centre that is offered by the cloud. The purpose of fog computing is to offload edge devices by placing an interface to the fog that is in their proximity. This allows the edge devices to be built with less powerful hardware and to use networks that would otherwise be too unreliable for edge to cloud communication.

This thesis presents a decentralised task assignment method for fog networks. The presented approach employs the MQTT protocol and uses auctioning mechanisms to propagate requests from the edge clients to fog nodes. The MQTT protocol provides reliable messaging with low bandwidth requirements and decouples the fog nodes such that they can operate in a dynamic fog network. Assigning tasks through auctioning ensures that a request is delegated to the most capable fog node in each step of the propagation. The auction approach is a selective message propagation method for cooperative and distributed multi-broker architectures. A demonstration program of the auction approach has been implemented and has been tested against two other decentralised task assignment schemes. A random approach to task assignment and a modified implementation of the method outlined by Battistoni et al. in [1].

1.1 Aims and Objectives

The main objectives for this thesis is to present and determine the feasibility of a decentralised task assignment system for fog computing architectures. The basis of the method is to hold autonomous auctions between fog nodes to propagate requests from edge clients. The feasibility of implementing the method will be determined by developing a representative demonstration program where the fog nodes exclusively communicate in accordance with the MQTT protocol. The thesis also aims to evaluate the viability of the auction approach by empirically comparing it to a random approach and an approach inspired by the method outlined by Battistoni et al. in [1]. To achieve this, the performance of the approaches will be measured in experiments that feature simulated fog computing networks that are populated with fully functional MQTT clients.

1.2 Structure of Document

This thesis is written with the intention to introduce its readers to a fog computing architecture that employs the MQTT messaging protocol. It is assumed that the readers are familiar with some networking terminology and general IoT concepts.

Chapter 2 gives an overview of fog computing, MQTT, and a look at the most important elements that found a basis of publish-subscribe messaging in fog architectures. The chapter also covers previous work on task assignment in fog computing networks. Chapter 3 outlines the components and behaviour of the auction approach, its intended functionalities, and an overview of the component interactions. Chapter 4 presents a working implementation of the auction approach. In addition to the behaviours of the random and modified Battistoni approach that it was compared to. The chapter also covers the method and the results of the experiments. Chapter 5 reflects on the results from the experiments, the advantages of using the MQTT protocol in fog networks, and provides an evaluation of the auction approach. Chapter 6 concludes the report and points to areas of further research.

Chapter 2

Theory

This chapter provides the necessary context and technical knowledge that lay the fundamentals of the work presented in this thesis. It does so by introducing the technologies that are used and background literature on the topic. This thesis presents an MQTT-based auctioning mechanism which is used for assigning service requests in a fog network. This chapter starts with a description of the fog computing paradigm and the MQTT messaging protocol. Following this, there are sections that will explain how MQTT can be used in fog computing architectures and previous research on this topic. By the end of this chapter, the reader should have a good understanding of the scope of fog computing and how MQTT can be used to route requests in a fog network. The following chapter 3 covers how requests can be propagated with auctions.

2.1 Fog Computing

The scope of fog computing is a widely discussed topic, for this thesis we are using the definition and elaborations as outlined by the survey of Yousefpour et al. in [2]. They mention the definition of fog computing as provided by the Open Fog Consortium (OFC), which is now a part of the Industry IoT Consortium (IIC). The OFC define fog computing as "a horizontal system-level architecture that distributes computing, storage, control and networking functions closer to the users along a cloud-to-thing continuum" [3]. Yousefpour et al. further elaborates on this definition by comparing fog computing to other similar paradigms. The names of these paradigms are often used interchangeably because of their similarities. However, this has lead to confusion when differentiating between the various architectures. The comparisons of Yousefpour et al. include well known paradigms such as edge computing, cloud computing, mobile ad hoc cloud computing (MACC), etc. It is important in the context of this thesis to have a clear picture of the scope and properties of fog computing. Therefore, a discussion outlining the differences between fog computing and some of the similar paradigms is included for clarity.

2.1.1 Comparing Fog Computing to Similar Paradigms

Fog computing bridges the gap between the edge and the cloud by introducing an intermediary component, the fog network, which is a collection of fog nodes that are in a closer vicinity to the edge in comparison to the cloud. The fog nodes enable computing, storage, networking and data management to be performed close to the edge. The term fog computing is often used interchangeably with edge computing, however these two paradigms are not identical. Edge computing relates to computing done at the edge and provides no hierarchical connection to the cloud. The exact location of the edge is dependent on the type of system. But in the context of the general IoT landscape, the edge is thought of as being contained in the local network and thus the edge nodes are normally a single hop away from end-devices. Consequently, edge computing is concerned with the organisation and offloading of tasks for a localized system. The edge servers can provide necessary real-time computations and services to constrained end-devices. In contrast to edge computing, fog computing is concerned with providing its services to multiple local systems.

One of the main characteristics of the fog computing architecture is that it acts as a junction for the local edge and remote cloud. Fog computing is often confused with edge-cloud computing because the fog network also connects the edge with the cloud. In the edge-cloud paradigm, some tasks are performed locally at the edge and some tasks are requested to be done remotely by a cloud provider. These remotely executed tasks are often referred to as services. In edge-cloud there are no services that are done along the way between the edge and the cloud. This is the major distinction between fog computing and edge-cloud. The fog network can provide services to the edge in addition to handling the transport of messages between the edge and cloud. The fog computing architecture is flexible and the messages it routes from the edge to the cloud may be any message which is typically used in edge-cloud, e.g., serverless function invocations, database entries and queries, raw data, remote procedure calls (RPC).

Fog computing is similar to cloud computing in the way that it also provides additional hardware resources and software as a service to the edge. Despite this resemblance, fog computing is rarely mistaken for cloud computing because they have clearly different strategies for organizing their hardware. A fog network is a highly distributed system, which means that its resources consist of many entities which are spread out across a wide area. Cloud computing architectures are also distributed, however to a lesser extent in comparison to fog computing. The cloud is comprised of large data centres that are spread out over a few select locations. A fog network consist of heterogeneous fog nodes which means that each individual fog node's computing resources may be different. A variety of moderately powerful devices can take the role of a fog node, e.g., small data centres, personal computers, sophisticated routers, IoT gateways. A fog network consists of interconnections between the fog nodes and also extended connections to the edge devices and cloud clusters. The fog network can be managed in a decentralized fashion, i.e., the fog nodes operate without a central organizational entity. However, centralized methods can also be used to manage a fog network, e.g., to introduce new fog nodes, delegate tasks, and maintain the network topology.

A fog network is dynamic, which means that the amount of participating nodes can vary as well as the available resources. As a result, the capabilities of a fog network can be scaled by adding new fog nodes. The dynamic and distributed nature of fog computing resemble the MACC paradigm. In addition, the MACC architecture also leverages cloud data centres. The main difference between fog and MACC is in how the architectures view the reliability and longevity of their participants. A MACC network is more dynamic than a fog network. The nodes in MACC are generally seen as mobile participants that create temporary ad hoc networks. In comparison to MACC, fog nodes are much more static and are expected to participate in the network for an extended period. Fog nodes are additionally considered to have moderately robust Internet connections such that they have greater availability than MACC nodes.

To summarize, the fog computing architecture describes three layers to do computations. The local edge, the intermediate fog network, and the remote cloud. The fog network consists of fog nodes which are distributed such that they cover a large geographical area. The fog network is moderately dynamic so the nodes may join, leave, and re-join the network but they will mostly remain as activate participants. The fog nodes are on the scale of small servers and can do moderately intensive tasks for multiple clients simultaneously. However, cloud data centres are much more powerful than a single fog node. Lastly, a fog network is an intermediate component that can aid the edge by providing remote services at the fog nodes or bridge requests to the cloud.

2.1.2 Naming the Fog Computing Components

This thesis uses a consistent set of nicknames for the components of the fog computing paradigm to alleviate the excessive need for the fog prefix throughout the text. The fog network is referred to as the *Fog*, the fog nodes that provide the services in the Fog are *Nodes*, and finally, the edge devices or fog clients that request services from the Fog are called *Clients*. Large scale cloud data centres are part of the fog computing paradigm and are referred to as a single entity, the Cloud.

Note that, the components which are nicknamed here are used as entities that are part of the fog computing architecture described by this thesis, i.e., the components are later assigned with qualities and requirements which are specific only to the proposed design. The section 3.2 provides further descriptions of the components in relation to the presented auction approach which uses MQTT and auctioning mechanisms for task assignment.

2.1.3 How Resources are Provisioned by the Fog

The fog computing paradigm is a three-layer model that includes the Clients, the Fog and the Cloud. Processing happens at each layer, however the Clients located at the far edge will at times issue requests for remote services to the Fog. Each request from the Clients must be allocated suitably to either a Node or the Cloud. A task assignment method is necessary in order to decide on which entity the request shall be delegated to. The aim of the task assignment method is to ensure a good trade-off between efficient use of resources and good performance. Finding a suitable Node or forwarding the request to the Cloud is referred to as finding a solution. A solution is associated with positive attributes such as low response times, load balancing, and energy efficient.

Each request from a Client is associated with some task that must be remotely processed either at a Node or by the Cloud. The task is referred to as a service that the processing entity provides to the Client. The requests should be delegated in an inherently load balancing method such that there is a smaller chance that a Node is overloaded. When a Node is overloaded it will either fail to provide its services or have low performances because it has been delegated too many concurrent requests. A request should be sent to the Cloud if it will lead to a better solution than what is available in the Fog. Additionally, requests may have properties that make them more suitable to be handled by the Cloud, such as long term storage, hosting globally accessible servers, and heavy computations.

Besides accounting for the isolated capabilities of each available Node the assignment method could also consider network congestion in the Fog. This thesis assumes that the connection between a Node and a Client is primarily established by propagation. Such that a request may pass through multiple Nodes before it reaches the Node that finally processes the service. This assumption is in place because the Clients are already assumed to have restrictive network properties that may not allow them to directly connect to a sufficiently large group of Nodes or the Cloud. The route that a request takes from a Client to the processing Node is referred to as the transmission path, propagation path, or just the path depending on the context. Some transmission paths are more constrained and could be compromised if a lot of request traffic was routed through them. The path considerations can be handled by routing requests with Software Defined Networking (SDN) tools. This has been studied for fog computing architectures by Gupta et al. in [4].

The assignment of requests can either be done in a decentralised fashion or through central delegation mechanisms. Centralized methods for organising and delegating requests in the Fog rely on transparent networks. With a complete view on the current load on the Nodes and network congestion it is possible to optimally assign tasks to the most appropriate Nodes in terms of a trade-off between the total cost and performance. An example of a practical implementation of a centralised task assignment system intended for fog computing was explored by Jung-Fa et al. in [5]. Their method can make specified trade-offs between time consumption and the cost of processing by solving an objective function reliant on detailed knowledge about the participating nodes. They were able to determine the optimal global solution but notes that the computational complexity of the centralised approach increases rapidly in proportion to the size of the network.

Therefore, a presumed disadvantage of centralized task assignment systems is that they will scale poorly because they have to process large amounts of data at a single point.

A decentralised task assignment method assumes that the Nodes are autonomous agents that are able to interact with other Nodes such that they can collaborate by sharing information and propagating requests. The Nodes will collectively strive to find the most optimal solution that can be found based on the local information that is available to them. The data that each Node will base its decision on will be limited to the knowledge it has about itself and any additional data that has been shared by its neighbours. As a result of not having a global state of the system, a Node will at each step of the propagation path determine if it should process the service and/or further delegate the responsibility to other Nodes. There are numerous decentralised task assignment systems that can be applied to fog computing and this thesis presents one in chapter 3 that utilises auctions and the MQTT publish-subscribe protocol. The advantages of a decentralised task assignment system is that the task of delegating requests is separated into sub-tasks that can be solved at each Node and that a global state of the system does not have to be maintained. The disadvantage is the challenge of finding good solutions based on the limited data that is available to each Node.

Auction algorithms to task assignment is not a novel topic but it has not been fully explored in neither a fog computing context or how it can be applied to publish-subscribe protocols in practice. Brunet et al. presents a consensus-based auction algorithm for decentralised task assignment in the context of autonomous UAVs in [6]. They showed through numerical experiments that the their auction algorithm was able to find identical solutions to a centralised greedy algorithm with good average performance. Le et al. introduce a device-to-device middleware architecture that enables multi-hop assignment of RPCs using Wi-Fi Direct in [7]. They proved the feasibility of their design on a test-bed comprised of Android smartphones.

2.1.4 How the Fog Provides Services to Clients

The Fog makes a pool of shared computing resources available to the Clients. The Clients can utilise the computing power of the Fog by making a request to one of the Nodes. The request is delegated to a subset of the available Nodes according to the applicable assignment method. The Fog delegates the requests without intervention from the Clients. Therefore, the Client applications should not assume the location and identity of the Node that will process the requests. The previous section 2.1.4 covered how the Fog allocates service requests based on the desired trade-off between performance and resource efficiency. The result is that a requested service can be processed by any suitable Node. The maintainers of the Clients do not necessarily have an opportunity to affect the configuration of the Nodes and therefore the Client applications that interact with the Fog must do so through a standardised interface. The implementation of the interface will vary based on the types of services that a Fog provides to its Clients. This section covers some of the proven types of services that can be offered by a fog computing architecture.

The Fog consists of many heterogeneous Nodes which means that the Nodes can run a variety of different hardware and operating systems. Therefore, the Fog needs to offer hardware and operating system abstraction to provide the Clients with the same experience when using whichever Node. The Fog is well suited for providing serverless services to the Clients. Because serverless services are state-less and agnostic to the hardware such that each request can be assigned to any Node without affecting the functionality. In their article Cheng et al. present Fog Function, a serverless programming model for fog computing applications [8]. Golem has deployed a peer-to-peer computation network that is used in a serverless manner where users request services by publishing task templates [9]. However, there is a demand for running self-written stateful services in the Fog and this has lead to designs that leverage virtual machines and container technology. In their article Hoque et al. present an architecture for orchestrating containers in fog networks via a centralized manager node [10]. SONM is distributed marketplace that allows its customers to upload Docker containers to suppliers utilizing the shared resource pool of the participants [11].

2.2 Overview of MQTT

This section is a summary of the MQTT messaging protocol and covers the most important mechanisms in the context of this thesis. Some readers may already be familiar with the material and can continue reading from 2.3, however 2.2.3 includes material on the request-response pattern which is new in version 5.0 of the MQTT specification. This summary is included to familiarize the reader with MQTT. Which is the underlying message protocol that is used by the presented auction-based task assignment method.

MQTT is a popular messaging protocol that is commonly used for M2M communication and IoT applications. The protocol was first developed by Arlen Nipper and Andy Standford-Clark in 1999 to be used with SCADA systems in the oil and gas industry [12]. MQTT is an open OASIS standard for a publish-subscribe messaging protocol [13]. A widely used alternative to MQTT is HTTP. The major difference between these two application layer protocols is the messaging pattern. HTTP is a request-response type of protocol, where a client sends its request to the server and receives a response. In contrast, MQTT is a publish-subscribe protocol where clients publish or subscribe to a topic hosted on a broker.

2.2.1 Publish and Subscribe

The MQTT protocol leverages the publish-subscribe pattern for delivering messages. MQTT defines three main entities: the broker, the subscriber, and the publisher. The broker is a server application and organises the routing of messages in an MQTT network. The publisher and subscribers are clients that are connected to the broker. A publisher is a sender of messages and a subscriber is a recipient. The connection of a subscribing client is being kept open by the broker such that it remains connected until it actively disconnects or the connection is broken. A publisher will send (publish) a message to the broker onto a specified topic. The broker will in response forward the message to any subscriber that has subscribed to the topic. A topic is a structured string where each topic level is separated by a forward slash.

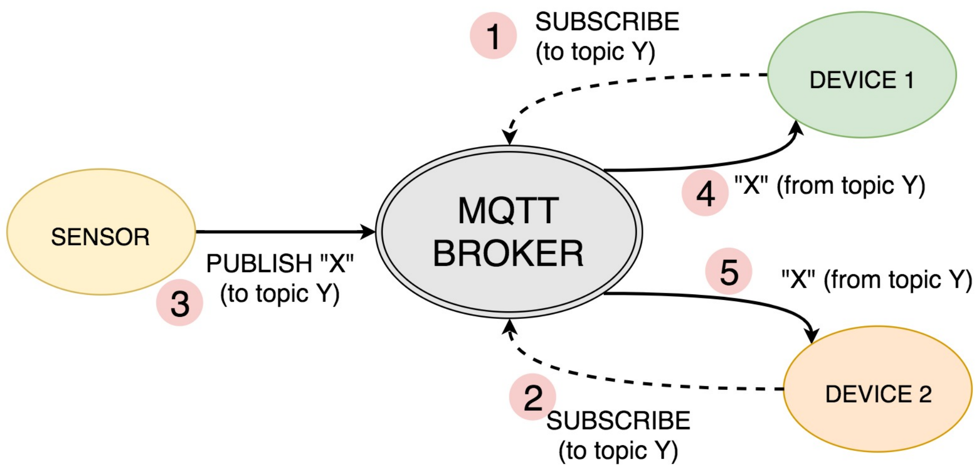

Figure 2.1: An overview of the MQTT topic-based publish-subscribe messaging. [14]

The figure 2.1 shows how messages are being routed in an MQTT network. The two devices (subscribers) on the right are subscribed to the topic "Y". They are expecting to receive the messages which are published to topic "Y". The sensor (publisher) publishes a message containing "X" onto the aforementioned topic "Y". The published message first reaches the broker, which in turn forwards the message to the subscribers of topic "Y".

The MQTT protocol is able to decouple the senders of messages from the recipients by handling the messaging in this fashion. In other words, the publisher is not aware of the existence of the subscribers who are receiving its messages. Neither do the subscribers need explicit information regarding the publishing clients. The subscribers are interested in any message that is posted onto a topic which they are subscribed to, without concerns about who the original publisher is. It is important to note that an MQTT client is able to act as a publisher and as a subscriber at the same time. Such that a single client can send outgoing messages and also receive incoming messages.

2.2.2 Quality of Service

The MQTT protocol has Quality of Service (QoS) defined as part of its specifications. QoS refers to the guarantee that is given for the successful transmission of a message. The MQTT protocol supports three levels of QoS. A higher level of QoS will offer more consistency in message delivery, however it comes with the cost of increased latency and bandwidth consumption. The following bullet points summarize the different levels of QoS that are available in MQTT.

0. The broker/client will deliver the message *at most once*, with no confirmation.
1. The broker/client will deliver the message *at least once*, with confirmation required.
2. The broker/client will deliver the message *exactly once* by using a four step handshake.

The QoS level is specified for either a subscription or to a published message in MQTT. Therefore, QoS can be applied by both subscribing and publishing clients. A publisher can make sure that its message is received by the broker if it specifies a QoS higher than 0. A subscriber can make sure that it receives all messages which it is subscribed to by specifying a QoS higher than 0. This assumes that the messages were successfully published onto the broker with a QoS higher than 1.

MQTT allows clients to set their preferred QoS. A higher level of QoS allows clients to be confident that their data is being received or sent in a fault tolerant matter. Being able to set the QoS is beneficial for systems that either have to assure message delivery or that has unreliable client connections.

2.2.3 Using the Request-Response with MQTT

The request-response pattern is new in MQTT v5.0 and is later referenced in section 3.3.5 where it is used to propagate responses back to the client. The request-response pattern is one of the most common ways of handling the transaction of messages between a client and a server. In request-response, the client will send a request to the server, and the server will send back a response to the client.

The request-response pattern requires that a response must be tied to a specific request. In other words, the client can match an incoming response to a request that it had previously sent. For practical reasons, implementations of request-response commonly use messaging protocols that offer 1-to-1 and synchronous client-server connections, such as HTTP. The benefit of having a 1-to-1 relationship between a client and the server is that their communication channel is exclusive to them. Which means that there wont be any interference from other clients. With synchronous communication, the request is directly followed up by a response and there is no concurrent exchange of messages. The combination of these properties let clients issue requests and be certain that the following response is only intended for them and that the response is an answer to their last request.

The request-response pattern is not inherently supported by the publish-subscribe messaging used in MQTT. With publish-subscribe the requesting clients are decoupled from the responding clients. Therefore, the publish-subscribe pattern can not assure a 1-to-1 relationship. The messaging in publish-subscribe is also done asynchronously such that a requesting client may receive any number of messages before the awaited response arrives. This makes it challenging to match a response to a specific request from the requesting client's perspective.

As of MQTT v5.0 the specification defines a standard method for implementing the request-response pattern in a publish-subscribe network. The main challenge is to match a response to a specific request. To achieve this, the request and response messages may contain additional data. The attributes of a message may contain a response topic and some correlation data. The response topic indicates which topic the responding clients should publish their response on. The correlation data is used to match a response to a request. When using request-response with MQTT there can be multiple responses to a single request. This situation occurs when there are multiple responding clients subscribed to the same request. However, the requesting client will still be able to recognize that these responses are tied to a single request by checking the correlation data. Additionally, the messaging can be done asynchronously because the responses are linked to requests by the correlation data which is embedded as a part of the properties of the messages.

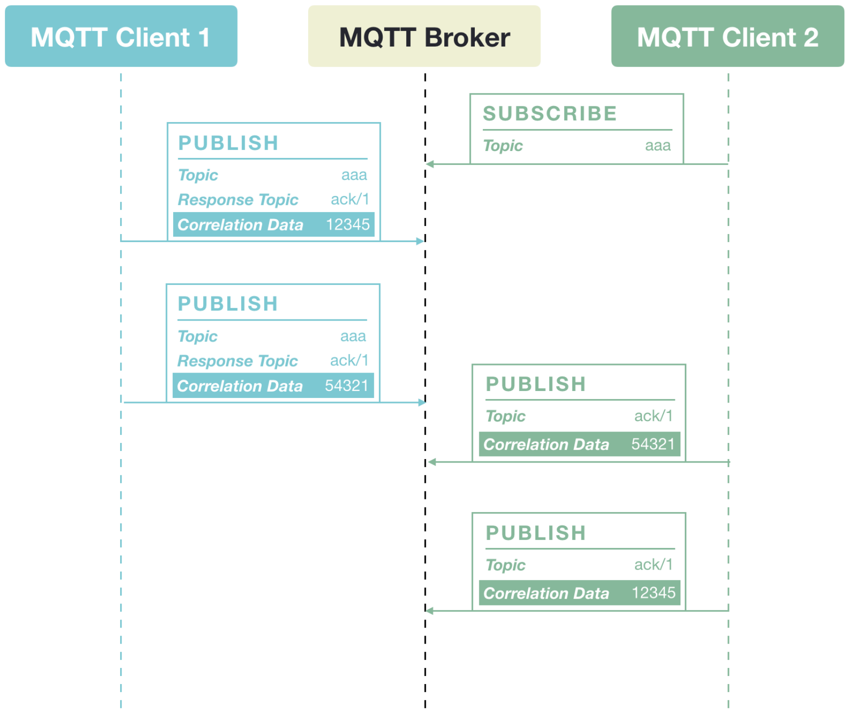

Figure 2.2: An overview of the Request and Response procedure in MQTT.[15]

The figure 2.2 is a sequence diagram that shows how the request-response pattern is implemented in MQTT. The figure includes the broker and two clients, a requesting client and a responding client. The client on the left is a requesting client and will publish messages that contain some request. On the right, the response client will be in charge of handling the request and publishing a response. The responding client is subscribed to the topic "aaa", which is where it will receive its requests. The requesting client publishes a message to the topic "aaa". This is the request and it contains two important additional pieces of information compared to a regular MQTT message. The request has attached a response topic "ack/1" and some correlation data "12345". The requesting client sends another request before it has received a response to its first request. The second request is also published to the topic "aaa" and has the same response topic. However, the correlation data has changed to "54321". On the right hand side of the diagram, the responding client first publishes a response with correlation data "54321" and then a second response with correlation data "12345". The requesting client will be able to tell which response is intended as an answer to the first and second request by matching the correlation data.

The request-response pattern can be implemented in MQTT by utilizing the response topic and correlation data attributes. This allows MQTT clients to verify the responses to their requests. The request-response pattern is practical when a client wants to invoke some remote functionality offered by another client in the network that results in a response.

2.3 Remote Service Requests in Fog Architectures with MQTT

One of the primary subjects of this thesis is to use the MQTT protocol for communicating remote service requests in a fog computing architecture. The architecture that is considered consists of multiple connected Nodes that each host a broker and a set of MQTT clients. Each Node provides remote services that are available to the Clients and upon receiving a request they can delegate the responsibility of handling the request to their neighbouring Nodes. Depending on the situation, a request can be propagated to none, one, a few, or all of the neighbours. Therefore, a task assignment method is needed to decide if a Node will initiate the processing of a service and how it further propagates a request to its neighbours. This thesis presents a task assignment method that uses auctions mechanisms to do so. Before covering the auction approach in chapter 3, this section provides an overview of how Clients publish their requests to the Fog, the service types that can be offered, and previous work on the topic.

2.3.1 Providing Remote Services with MQTT

An MQTT client can order a request for a remote service by specifying the name of the function in either the message payload or by publishing to a designated topic.

The message that contains the request must reach at least one subscribing client and contain some information that maps to a specific service on that client. The receiving client will interpret the contents of the request and handle it accordingly.

The figure 2.3 shows an example where a Client that is located at the edge requests a service from a Node in the Fog. The Client makes a request for Service A by publishing an MQTT message onto the topic "Request/ServiceA" with input arguments "input" in the payload. The Node hosts the broker which the request is published to in addition to an internal MQTT client that is subscribed to the topic. The internal client on the Node receives the request, recognizes the requested service-name from the topic, and carries out the processing of Service A with arguments "input".

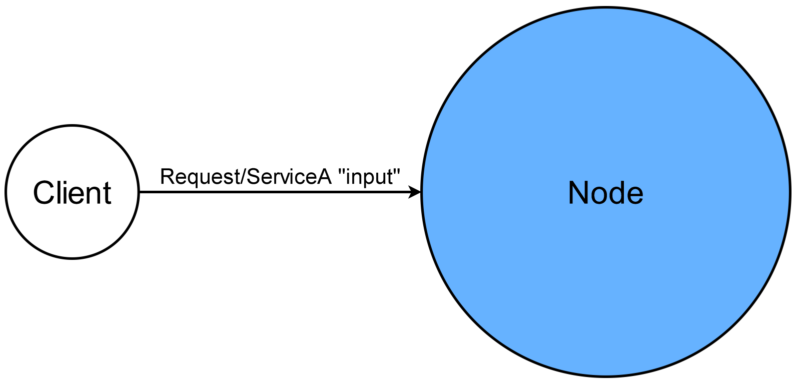

Figure 2.3: A representation of a Client requesting a service from a Node in the Fog. The topic is being used to specify the request of Service A.

Service Types

The MQTT protocol can be used to request services but using a 1-to-1 request-response protocol is far more common in practice. That is because MQTT has additional considerations in regards to maximum message size, asynchronous behaviour, and publish-subscribe messaging. The MQTT protocol allows for a maximum payload size of 255 MB [16]. This is a substantial message size in practice and it is common to have broker instances with lower limits set because of bandwidth or application constraints. Due to the generous payload size, small files can be sent as a single message but messages that are too large must be published as multiple messages in sequence. However, ordered messaging would require supporting application logic because MQTT is asynchronous and does not guarantee the order of messages. As a result, MQTT is inherently better suited for transporting messages with small footprints and is not suitable for sending large data structures and ordered sequences such as high resolution images and video streams.

MQTT can effectively be used for transmitting stateless service requests and possibly be used to initialise stateful services. A service that is stateless has a short life cycle that terminates after some task has been performed and does not store state information between requests. An example of a stateless service is downloading a file from a web server. Stateless services can be provided by a different server each time and are characterised by a single request and a response. On the other hand, a stateful service maintains a record of past requests or transactions and are long-term commitments for the provider of the service. An example of a stateful service is file storage. The stateful services are characterised by maintaining a history of past events and are therefore usually provided by only one server. A general challenge for requesting services with a publish-subscribe protocol, such as MQTT, is that the relationship between the publisher and the amount of subscribers is not guaranteed. As a result, any amount of clients may receive and want to respond to a single published request. Furthermore, the identity of the subscribers may change between consecutive requests as any client may subscribe or unsubscribe from the topic. When requesting a stateless service, the requesting party is generally unconcerned with the identity and amount of responding parties because the expected outcome is unchanged. As a result, MQTT can naturally be used for stateless service requests as long as the published message reaches at least one provider. In contrast, stateful service requests are required to reach the same provider every time because the provider is maintaining the state of the service. The MQTT protocol is unsuitable for guaranteeing this type of strong coupling between clients. For stateful service requests, it would be practical to establish a direct connection between the requester and provider with a protocol such as HTTP after the initial request has been made. The initial request can be issued with the intent of identifying a provider of a stateful service and could feasibly be communicated as a request-response message in MQTT.

This is a conceptual example of how the stateful service of hosting a Docker container can be initiated with MQTT. The size of a minimal Docker image with Ubuntu is 188 MB and could be sent as a single MQTT message. An Ubuntu container image might in practice be larger than the maximum message size if it is more sophisticated or if the maximum payload size has been configured to a lower size than the specification allows. However, in such cases there exists smaller containerized environments that may be used instead, e.g. an Alpine Docker Image is only 5 MB. To host the container a Client publishes the container image to a Node. That Node and its neighbouring Nodes are notified of the request. A task assignment method is used to determine the Node that will host the container. The application that is contained on the image is a program for a web server. The Node that is assigned with the task runs the image and publishes a response to the Client that requested the service. The Client can recognise that it is the intended recipient of the response if the request-response pattern that was covered in section 2.2.3 is used. The contents of the response is the public IP-address of the web server and the address of the REST API that the Node hosts as an interface for managing the container. The Client interacts with the Node over HTTP through the REST API to manage the container for the remaining lifespan of the service.

The work of Benedetti et al. in [17] analyse the suitability of serverless functions in IoT applications by comparing the performance of cold start and hot start function pods. They present an architecture that monitor published events on an MQTT broker where the contents of the event can trigger the invocation of a serverless function with the OpenFaaS framework. An experiment performed by the Ericcson Research group shows that MQTT messaging (at QoS 1) can provide a higher throughput and lower latencies compared to HTTP for a smart vehicle that requests services from an edge or cloud server [18]. They showed that MQTT performed significantly better than HTTP in cases where there were many connected clients.

2.3.2 Decentralised Task Assignment with MQTT

The system architecture in this thesis is a distributed system that contains multiple MQTT brokers. The brokers are hosted by Nodes and the Nodes are connected to each other. The connections between Nodes are further covered in section 3.2.2. The Clients send their service requests to the Node that they are connected to which is presumably their closest Node. All Nodes that are part of the Fog are able to provide a set of services and can also further delegate the responsibility of a request to their neighbouring Nodes which they are connected to. Upon receiving a request, a Node will decide if it shall process the service and/or propagate the service to neighbouring Nodes. The specific actions that the Node will have in response to a request is dependent on the task assignment method that is implemented. The task assignment method determines which Nodes that will process the service that is associated with a request and will therefore affect the performance of the system.

There are two perceived paradigms of decentralised task assignment that can be used to delegate the responsibility of handling a request. The Node that currently has the responsibility of handling a request can order, as in command, its neighbours to take responsibility or the neighbours can freely decide if they would like to take the responsibility. Instructive assignment from the currently responsible Node is the most primitive way of propagating a message because it requires the least interaction between the delegating Node and its neighbours. On the other hand, the neighbours can also participate in the decision-making process but this requires additional communication. The minimal requirement is that the currently responsible Node must inform all or a subset of its neighbours of the request. This will allow the informed neighbours to make a decision for themselves if they wish to handle the service. For some task assignment methods this is sufficient to make a decision but the drawback is that the number of neighbours that will handle the request can not be guaranteed. Such that it is probable that all of the neighbouring Nodes decide to handle the request or that none of them will. Other task assignment methods, such as the auction approach, allows the informed Nodes to collaborate in the decision making process by publishing a response to the given information on the request. The response can either be publicly available to the all the neighbouring Nodes or sent privately to the delegating Node. The response will contain information that is relevant for the decision making process. The information that was previously only available to each individual Node is shared such that the Nodes can collaborate in the decision making.

This thesis present an auction approach that a Node uses to delegate the responsibility of handling a request to neighbouring Nodes. If a Node decides that it is unable to process the service that is associated with a request that it is responsible for then it invites its neighbours to an auction. The auction is initiated by informing all the neighbours of the request. The neighbours respond by placing their bids in the auction that is held by the currently responsible Node. The bid contains information that reflects the performance that can be expected if the request was delegated to the Node that placed the bid. As such, the highest bidder in the auction is, based on the available information, expected to offer the best performance. The bids that are placed in an auction are publicly available to all the neighbours that participate in the auction. By comparing their bids to the highest bid, the neighbouring Nodes are able to decide for themselves if they should handle the request.

The auction approach is inspired by the works of Battistoni et al. in [1] and the system architecture used in this thesis is comparable to the one presented in their article. The article presents a decentralised approach to propagating requests with MQTT. The intended use-case was to route requests for remote services in a fog-mist environment. The objectives of their work was to offload mist nodes by leveraging the available computing resources of other mist nodes. The mist nodes are connected to a local fog node which is responsible for hosting an MQTT broker that relays the request from a mist node to the other mist nodes. The mist nodes are defined as edge devices that have limited hardware and storage resources, additionally they may provide services to multiple end-devices. Therefore, they sometimes need assistance to perform their tasks. The article includes a tactic for producing a response as well as a topic structure that can be used to route the requests and responses. The framework proposes a routing mechanism where a request is routed to every available mist node in the network. The tactic used for producing a response is a direct competition between the mist nodes to complete the task the fastest. The winner is the fastest mist node to publish its result. The losers will stop their ongoing computations when they are notice that another mist node has completed before them and return to their normal operation. The drawbacks of their design is that the competitive tactic does not offer conditional assignment of tasks which could lead to inefficient use of resources. A modified version of the system outlined by Battistoni et al. was implemented and tested against an implementation of the auction approach. The results are available in section 4.6.

Chapter 3

Method

This chapter covers the main characteristics and components of an auction-based task assignment system. In addition, considerations that are related to certain design aspects are included. The auction approach is a decentralised system that can be used for propagating service requests in the Fog. The benefit of the auction approach is that each Node's capability of handling a request is represented as a one-dimensional bid. A request is propagated through a series of auctions in which the participating Nodes place their bids. The winners of the auctions are assigned with the responsibility of further handling the request. A Node that is in charge of handling a request will either process the service or auction it to its neighbouring Nodes. A working demonstration program that uses the auction approach is covered in chapter 4.

3.1 Introduction to Assigning Requests with Auctioning in the Fog

The fundamental strategy is that the Fog uses an auction-based way of assigning the processing of remotes service to the Nodes or the Cloud. A Client that is placed on the edge will be connected to the MQTT broker that is hosted by a Node in close proximity. This allows the Client to request services from the Fog. The requested services may be processed by any capable Node in the Fog or the Cloud. The auction approach delegates the task of processing a service by propagating a request through a series of auctions. The propagation ends when a suitable Node has been declared the winner of an auction. A Node is decided to be suitable for processing the requested service if its bid is above the asking price. The asking price is a threshold that puts a requirement on the capabilities of processing Nodes. If a Node wins in an auction with a bid that is lower than the asking price. Then it will still be responsible for the request, but it is unsuitable for processing the associated service. The Node will therefore re-auction the request to its neighbouring Nodes. The Cloud may also participate in auctions in the same manner as a Node that is part of the Fog. This thesis uses the term 'Node' to refer to fog nodes, however it should be noted that a Node may in some circumstances also be run from within the Cloud.

When a Node is assigned with the responsibility of handling a request then it will decide if it is going to process the service or auction it to its neighbouring Nodes. The decision is determined by the Node's bid size for the given request and the asking price of the request. If the Node has a bid that is above the asking price then it will process the service and if it does not then it will auction the request. When a request is auctioned then it is referred to as an item. The participants of an auction will bid on the item according to their capabilities of handling the request. A Node that auctions an item will hold the auction on a topic that is available through the MQTT broker that the Node is hosting. The neighbouring Nodes receive notice of auctions because they have MQTT clients that are subscribed to the auction topic on multiple Nodes in their vicinity. An MQTT client may only be connected to a single broker and therefore a Node has one client for each Node that it is connected to. The MQTT clients on a Node are responsible for participating in auctions by publishing bids and observing the bids of other participants, detecting new auctions, and also managing the Node's auctions.

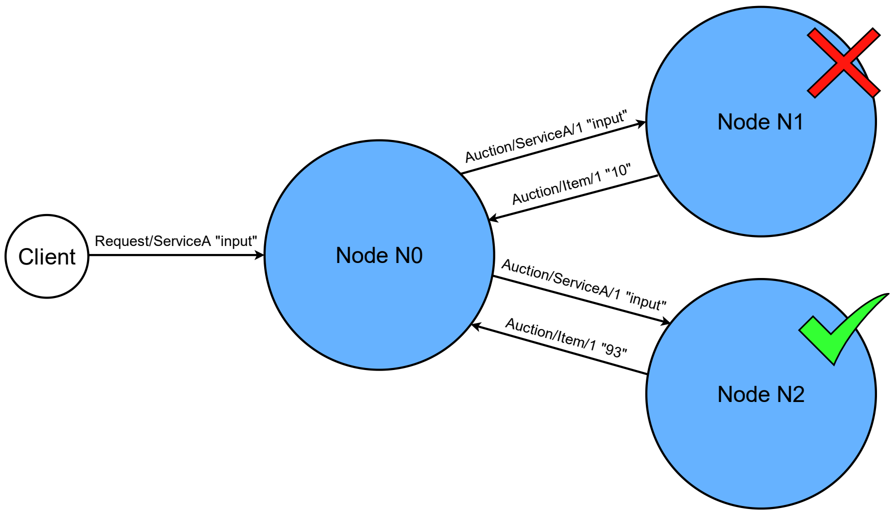

Figure 3.1: A representation of a Client requesting a service from a Node in the Fog. The Node starts an auction and the request is assigned to a neighbouring Node.

The figure 3.1 shows an example of a Client that requests Service A to be done remotely by the Fog. The Fog contains three Nodes in this example: N0, N1, and N2. The request is first published to N0 on the topic "Request/ServiceA" with a payload containing the input arguments. The Node N0 decides that it will start an auction and publishes the request as an item on the topic "Auction/ServiceA/1". In the figure, the auctioned item has been given an ID of 1 and the input arguments of the original request is in the payload of the item. The Nodes N1 and N2 receive the published item as they are subscribed to auctions for the Service A and they participate in the auction by placing their bids on the item. The bids are published onto the topic "Auction/Item/1". For this example, assume that the valid bidding range is from 0 to 100. N1 places a relatively low bid of 10 because it believes that it wont be able to adequately process the request. This could be due to latency between N0 and N1, heavy load on N1 from other requests, the nature of the requested service, and so on. N2 places a high bid of 93 because it believes it is a great candidate to process the request. The Node N2 has made a higher bid than N1 and therefore N2 is the winner of this auction. As the winner, N2 should be given the responsibility of handling the request of Service A. In this example, N2 presumably processes the service because of its large bid size, however if it does not then it is responsible for re-auctioning the item to its own neighbours.

3.1.1 Auction Propagation

The auction approach provides a method for selectively propagating service requests in a system that has multiple brokers. A request is propagated from the domain of one broker to another broker by publish and subscribe mechanisms that provided by a set of MQTT clients that are managed by a centralised control program in the form of a Node. The Node manages clients that are connected to multiple brokers. This method has been chosen over something such as broker bridging because it is more suitable for making decisions based on the contents of messages. Broker bridging is a strategy for forwarding the messages that are related to a specific set of topics from one broker to another broker. It is provided by most MQTT broker implementations.

The Nodes in the auction approach manages a set of MQTT clients that they use to forward requests from one broker to another. The propagation of requests from a Node to its neighbouring Node is achieved by holding an auction. The auctioning Node manages an MQTT client that announces a new auction by publishing a message onto its own broker. The MQTT client managed by its neighbouring Node is subscribed to the auction topic on the auctioning Node's broker. The subscribing client is notified of the auction and can publish a bid on the behalf of the neighbour Node. The neighbour Node can further propagate the request by initiating an auction on its own broker. The auction approach is selective because the neighbour Node will only initiate a new auction when it is unable to process the service because it has a bid that is below the asking price. Additionally, the neighbour Node will not consider a re-auction if it did not win the auction that was held on the original auctioning Node.

The figure 3.2 visualises the bids of Nodes as a scale and the asking price as a threshold. Any bid can be in the range of 0 to 100 and the asking price is set to 50. The asking price of a request determines the minimum bid size required to carry out the processing of that specific request. In this example, any Node that won the auction with a bid below 50 will be forced to re-auction the item its neighbours. In addition to providing an example of an asking price, the figure also separates the bid range into three colored sections. The green section represents the winners

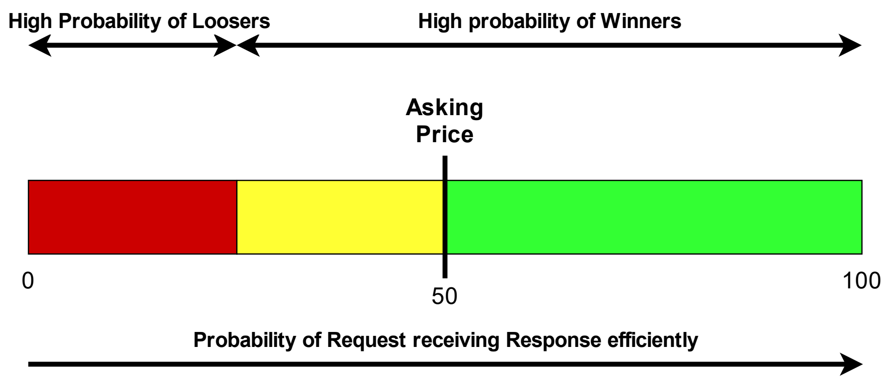
High Probability of Loosers
High probability of Winners
Asking Price
0
50
100
Probability of Request receiving Response efficiently

Figure 3.2: The asking price is set to 50, any Node with a lower bid than this will be forced to re-auction the item in the case where they win the auction.

who are above the asking price and which could proceed with directly processing the request. The yellow section indicates that the winners here will have to re-auction the request, and the red section shows a part of the bid range where there should mostly exist losers. The probability of winning an auction will scale with the size of the bid and the bid size gives an indication to the expected capabilities of the winning Nodes.

The figure 3.3 is an example of how the asking price of an auctioned item can be used to propagate requests in the Fog. The Client is represented as a white circle and the Nodes by slightly larger and colored circles. The top part of the figure shows that a Node may be in four different states. Any Node which is not occupied with the specific request in this example is deemed as idle, i.e., it may process other unrelated requests. A Node can go from the idle state to either a losing or winning state. The winners are categorised into winners who are above and below the asking price. In this example, every auction with multiple participants will have two winners. Step 1 in the figure shows that all Nodes in the network are in the idle state and additionally that the Client sends the initial request for a service to the top-most Node. Step 2 shows that the first Node was unable to bid above the asking price and therefore auctions the item to its neighbouring nodes that are one layer below it in the graph. Step 3 shows the result of the re-auction that was started in step 2. Among the three active nodes in step 3 there are two winners and one loser. The losing Node does not continue handling the request and can return to its idle state. The winning Node that bid above the asking price will process the service and not re-auction the request. However, the winning Node with a bid below the asking price re-auctions the the request to its neighbours in the layer below. Step 4 concludes the figure and shows that there was another winner above the asking price for the re-auction that occurred in step 3.

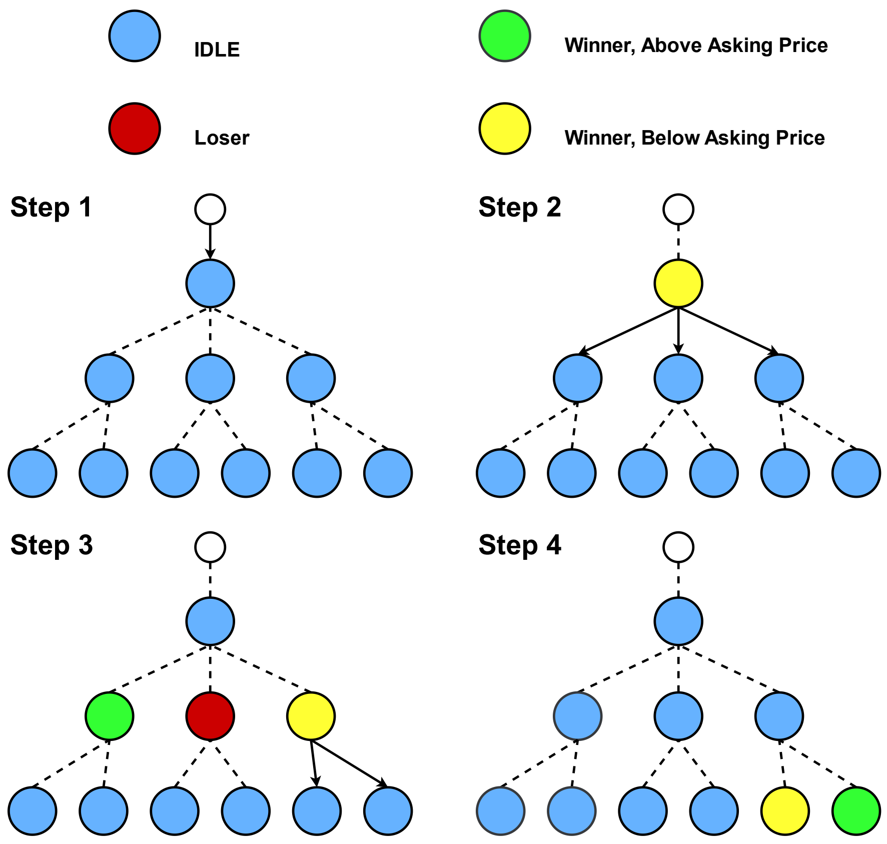

Figure 3.3: The asking price decides whether a Node should re-auction the item. The auctions in the figure have two winners when possible.

3.2 Participants of the Auction

This section revisits the key architectural components of the fog computing architecture but in the context of the design of the auction approach. As presented in 2.1.1, the main fog components are the Nodes, the Clients, and the Cloud. Where the Fog is comprised of a network of Nodes. The descriptions of the fog components in this section are specific to the auctioning approach that is presented in this thesis and therefore the attributed qualities are not true for all fog computing architectures.

3.2.1 The Clients

The Clients represent end-devices or local systems which are placed at the edge. The Clients are requesting services to be performed remotely by the Fog. To be able to request services from the Fog, the Clients are required to be connected to a Node in their proximity. They request the services by publishing MQTT messages to a Node. Therefore, the Clients must be able to run an MQTT client. To request a specific service, the Clients must be aware of the catalogue of services that are provided by the Fog. This thesis assumes that the requests which are made by the Clients will always map to a legitimate service that is provided by Nodes in the Fog or the Cloud. However, missing services may appear in practical applications and complete service availability is not a requirement of the auction approach in general.

The Clients must have a set of qualities and abilities to interact with the Fog:

- They have network capabilities that allow them to connect to a Node in their vicinity.
- They are able to run an MQTT client which can publish and subscribe to a Node.
- They know the set of services that are offered by the Fog.

3.2.2 The Nodes

The Fog is comprised of multiple connected Nodes which provide services to the Clients. The Nodes are connected to each other by a group of managed MQTT clients. The Nodes are responsible for holding and participating in auctions. Each Node carries a set of services which they are able to provide to the Clients. The services can be requested by publishing to any Node. The Node handles the request, which means that it either delivers the requested service or that it puts the request up for auction. They should also be able to route responses back to the Clients. Note that this is not part of the demonstration program of the auction approach that is covered in section 4.2 but guidelines to how responses may be propagated is presented in section 3.3.5.

A Node is the interface between the Fog and the Clients. The Nodes have the responsibility of hosting an MQTT broker such that the Clients in their proximity can connect to them and request services from the Fog. The neighbouring Nodes are also connected to the broker such that they can subscribe to auctions and publish their bids. Each Node has a set of MQTT clients in addition to the broker. The MQTT clients are in charge of functionalities which are not supported by the broker. That includes tasks such as handling requests and services, placing bids, and managing auctions.

The Nodes will partake in auctions as either bidders or auctioneers. When a service request is put up for auction it becomes an item. The Nodes bid on an auctioned item when they receive notice of a new auction that is hosted by one of their neighbouring Nodes. The Nodes become responsible for a request whenever they receive a request directly from a Client and when they win an auction. The Nodes put a request up for auction if they are not able to bid above the asking price. An auction can either be in the form of active or automatic participation. The types of auctions are described in section 3.3.3. For active auctions, the neighbours of an auctioning Node are notified of the start and end of an auction such that they can place their bids in the active auction. Automatic auctions only notify about the end of an auction. In addition to placing bids and holding auctions, the Nodes are also responsible for declaring the winners and losers of the auctions. The tactics for determining the winners of an auction are discussed in section 3.3.3.

The Nodes are in this thesis assumed to carry the complete catalogue of services that is offered by the Fog but are not required to in practice. The Fog consist of Nodes that are heterogeneous in terms of hardware and capabilities. That means that some Nodes may not be able to carry specific services. Additionally, the demand for a service will affect the necessity that a certain Node carries it. Therefore, a Client may request a service that is not implemented on its nearby Nodes. If that is the case, then the request must be forwarded to other Nodes or the Cloud. If it is a recurring request then a new and capable Node can be introduced or a nearby Node upgraded such that similar requests can be handled in the future. How to handle service availability is outside the scope of this thesis and is therefore not further discussed.

The Nodes may leverage the Cloud to offer or improve some of its services. The Cloud can be contacted through an interface, such as cloud agent, to offer parts of the service. For example, this would be useful if the Node could offer services such as inserting data into and querying from a remote database. The Cloud may also be used to off-load the Fog if tasks such as heavy computations are forwarded to a powerful cloud-hosted server. The Cloud can also be used as an interface to manage the Nodes and the network topology in the Fog.

The Nodes have a set of qualities and capabilities required to partake in the Fog:

- They must host an MQTT broker and clients.
- They offer a set of services that may be requested by Clients.

- They can participate in and manage auctions autonomously.
- They can collaboratively route responses back to the Clients.
- They are able to connect to the Cloud.
- They are able to subscribe and publish to neighbouring Nodes.

3.2.3 The Cloud

The Cloud is located at centralized data centers and is seen as a single entity in this thesis, however in reality the Cloud is a distributed system. The Cloud carries an implementation of all of the services that may be offered by the Fog. Additionally, the Cloud also carry services that are beneficial for the Nodes and Clients. The Cloud can aid the Nodes with services such as neighbour discovery, downloading missing services, and supplementary computing resources. The Clients may want to use the Cloud to receive a set of available services, finding their closest Node, tracking their activity and expenditures, etc. The Cloud is a good candidate for delivering these services as it is publicly accessible to all Nodes and Clients and can supervise the Fog as a centralized component.

The Cloud may join the network in a similar fashion as a Node in the Fog. Which means that the Cloud can participate in auctions, manage auctions and deliver services to Clients. The Cloud can join the Fog at multiple locations because it is able to process more requests than a single Node and therefore serve effectively as a neighbour to more Nodes compared to the typical Node.

The Cloud must offer certain functionalities which benefit the Nodes and Clients:

- It carries the full catalogue of services that may be requested.
- It can join the Fog at various positions as a Node.
- It offers services to both Nodes and Clients.

3.3 Design Aspects of the Auction

This section covers the key design aspects that have to be considered when designing and implementing an auctioning tactic for task assignment in publish-subscribe style systems. The introduction to the auctioning tactic was given at the start of this chapter in section 3.1. Furthermore, an overview of the responsibilities and requirements of the participating components were outlined section 3.2. This section aims to clarify some of the details that were left out in the previous sections by providing a roughly chronological look at how a service request is delegated to a set of Nodes. It starts of with how Clients issue their requests to the Fog. Then the choices with regards to auction design are examined which affects the flow of messages in the Fog. Lastly, the challenges related to routing a response back to the Client and late joiners in auctions are considered.

3.3.1 Requesting Services From the Fog

The service requests that are sent by the Clients must contain information regarding the specific service that is requested, which is referred to as the service class, and the request must be structured in a standardised format. Any service request is sent as an MQTT message and contains information related to the request in the topic that it was published on, the payload, and in the properties of the message.

When the requested service induces a response then the request from the Client should contain a response topic and optionally some correlation data. In this case, the request-response information is contained in the properties of the MQTT message. How to handle services that incite a response is covered in section 3.3.5. The class of the requested service can either be present in the payload or in the topic that was published to. The input parameters of the requested service is a part of the payload because these can not be used to categorize the service. The input parameters may be unique for every request for a certain service and therefore they do not follow a pattern such that they can not be subscribed to at the topic level. The request may also contain a suggested asking price in cases where it is not equal to the standard asking price that is associated with the requested service. The asking price is further examined in the following section 3.3.2. The table 3.1 summarizes the information that a request might contain and displays how the message is constructed.

Message Format of Service Request		
Information	Placement	Required
Service Class	Payload or Topic	Yes
Input Arguments	Payload	Depends
Response Topic	Properties	Depends
Correlation Data	Properties	No
Asking Price	Payload	No

Table 3.1: Service Request Message Structure

3.3.2 Setting the Asking Price

The asking price of an item puts a requirement on the capabilities of the Node that is handling a request. A Node is considered to be unsuitable to process a requested service if it bids below the asking price in an auction. If a Node wins with a bid below the asking price, it should re-auction the item to search for better candidates. As a result, the magnitude of the asking price determines the probability that the item will be re-auctioned. It should be noted that the bid size of a Node may change during the lifespan of the system such that a given Node may have a bid that is above the asking price at one point and below at another. This thesis does not give a precise answer to how the asking price of a request can be calculated but introduces the one-dimensional product as a way to propagate items in the

Fog. However, this section describes what the asking price practically represents and suggests that it may be implemented as a weighted sum.

The asking price of an item is a one-dimensional number that represents a multi-dimensional sum of the various factors that determine the processing capabilities of a Node. The asking price is expressed as a weighted linear combination in the equation 3.1, where w_i is the weight that is associated with the characteristic c_i. The characteristics represent the performance specifications of a Node, e.g., CPU clock speed, persistent storage space, bandwidth. The value of the weights depends on the service that is requested and each implementation of a service defines their own set of weights in accordance with their priorities. Therefore, the only considerations when setting the asking price for an item are the preferred characteristics of the Nodes. The asking price of an item should be set such that any Node with a combination of characteristics that allows it to bid above the asking price will be suitable for processing the service. The section 3.3.4 shows how the bid size of Nodes are calculated with a component that is equal to the expression for the asking price.

$$\text{Asking Price} = \sum_{i=0}^{n} w_i c_i \tag{3.1}$$

Items with tall asking prices are more likely to be re-auctioned and therefore they are propagated further into the Fog than items with lower asking prices. Items with lower asking prices find a suitable Node more easily because there are more Nodes that can bid above the asking price. But will expect a lower performance from the Nodes. The benefit of setting a tall asking price for an item is that only highly motivated Nodes will be able to bid above the asking price. A Node that is highly motivated is confident in its current ability to perform the requested service. However, the asking price of an item can be overly tall such that it becomes troublesome to find a Node that is able to produce a satisfactory bid.

The class of the service that is requested is the main factor that determines the magnitude of the asking price for an item. Some classes of services will have high performance requirements for the Nodes and as a result have tall suggested asking prices. Each class of service has a set of weights that determine the priority of a Node's various characteristics. The set of weights for a service will determine how Nodes calculate their bid size for items that represent that service. The asking price for an item may also be adjusted in accordance to the preferences of the requesting Client. This is beneficial in cases where the Client have higher or lower requirements to the performance than what is typical for a given service.

3.3.3 Managing an Auction

There exists a vast variety of ways to manage an auction and the style considered for this thesis was chosen to be easily implemented with MQTT and understandable. The work of Parsons et al. in [19] provides a general guide to different ways of managing an auction and has been used as a reference for technical terms and as a source of inspiration.

The auction style presented in this thesis is by formal definitions an open cry, single unit, and one-sided auction. The bids that are placed are one-dimensional. In other words, the bids are made publicly, the auctions are held for a single item that may consist of bundled sub-items, and the auctioneer does not actively match bids with items based on the asking price. The bids are given as a number, however they represent conditions that are multi-dimensional. A unique feature of the auction style that is used for this thesis is that the bidders themselves will declare the winners and losers of the auction. Therefore, the auctioneer has limited responsibilities compared to typical auction styles and is only in charge of presenting an item, starting the auction, providing a space to make bids, and marking the end of the auction.

Starting an Auction

A Node will start a new auction when it is not able to bid above the asking price of a request that it is responsible for. A Node becomes responsible for an request in two distinct cases. Either when it is the first Node to receive the request from a Client or when it wins an auction.

A new auction for an item starts by informing the neighbours of the auctioning Node. This is done by publishing an MQTT message on to an auction topic that the neighbours are subscribed to. The message contains the fields that were listed in table 3.1 and, optionally, information related to participating in the auction. Depending on the implementation, the auction topic which the item is published on may represent a marketplace for all auctioned items or a separate auction room for a specific item.

The procedure of participating in an auction will be determined by the topic structure that is used to hold the auctions and the subscription pattern of the neighbours. Depending on the topic structure, the auctioneer may only notify neighbours that are interested in handling the service in question or it may notify them whenever any new auction is started. The topic structures are referred to as either "inclusive" or "selective". The subscription pattern may be designed such that neighbours that wish to participate in an auction will have to actively enter an auction room to join in on the bidding round or automatically subscribe them to all auction rooms. The subscription patterns are referred to either as "active" or "automatic".

The figure 3.4 shows an overview of how the topic structure and subscription pattern will affect where auctions are being held and which Nodes that will participate. It features two ways for an auction space to be represented as an MQTT topic and two ways of joining an auction. These are combined to make four examples of auction systems. The categories active and automatic refer to how Nodes join an auction. The active category refers to a subscription pattern where the Nodes deliberately join only the auction rooms which they are inter-

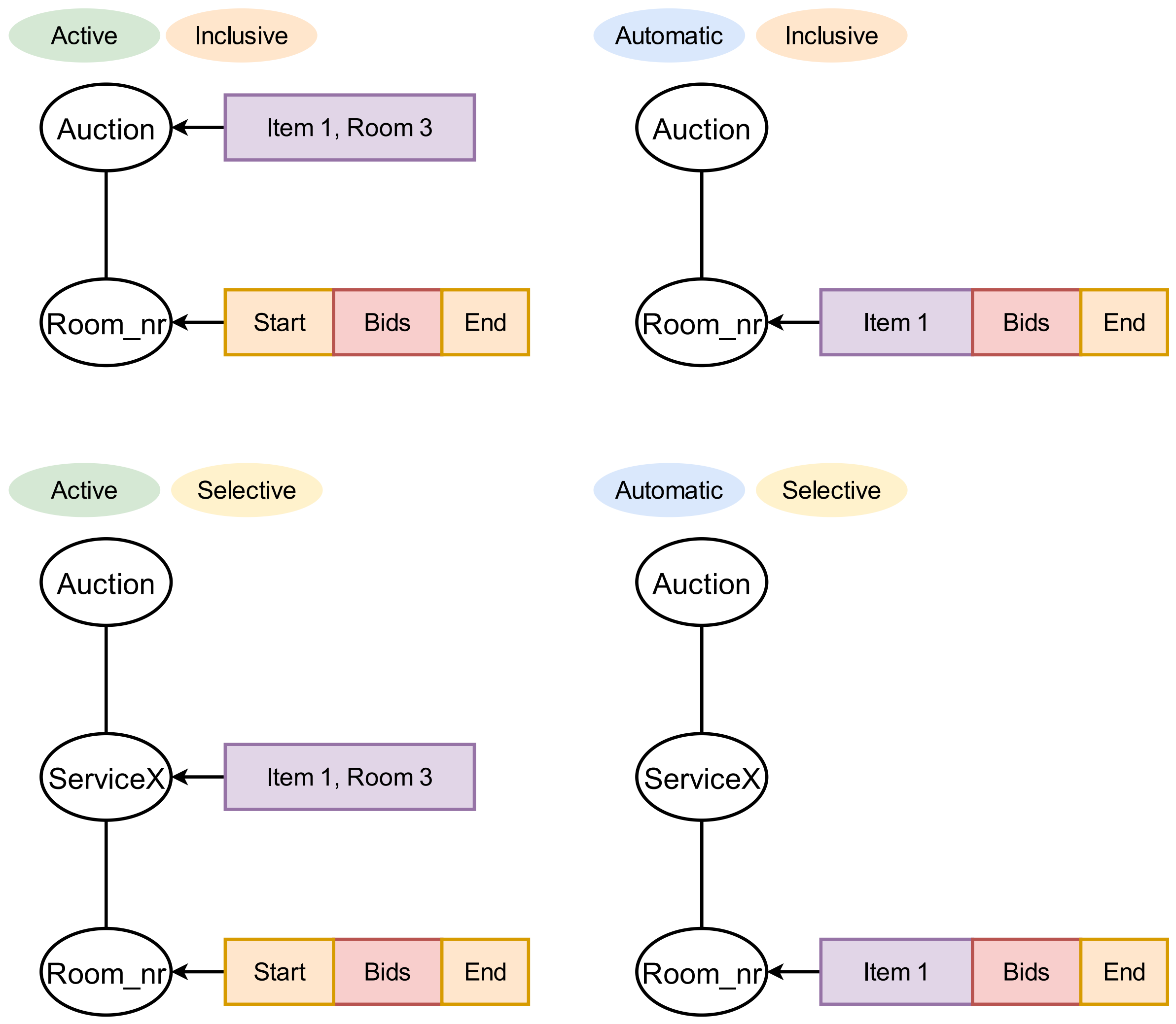

Figure 3.4: Visualisation of some possible topic structures for auction rooms and subscription patterns. Highlighting active vs. automatic subscription pattern and inclusive vs. selective topic structures.

ested in. While for the automatic category, all Nodes are invited into the auction room even if they do not plan to bid on the item. The selective and inclusive categories decides the topic structure for the auction rooms. The selective way of holding auctions allow Nodes to only subscribe to auctions for services which they are interested in bidding on. The inclusive way of holding auctions will not discern between the service that the item represents at the topic level.

The top left diagram is an example of an active and inclusive system. The Nodes are subscribed to the Auction topic and are notified of the new auction with a message that contains the item, Item 1, and an auction room topic, Room 3. All Nodes which are subscribed to the auctioneer are notified of the auction because this is an inclusive system. However, the Nodes can choose to subscribe to the auction room topic Auction/Room 3 after having looked at the nature of Item 1. This is possible because it is an active system. The Nodes that do not subscribe to Auction/Room 3 will not see the bids that are posted for Item 1. The first message to be sent in the auction room topic is a start signal. A start signal is recommended for active participation auctions. After the start signal has been sent, the auction participants may place their bids until the end signal is published. The start signal is included such that the early bids are not lost at a frequent rate and ensure that late joiners are aware of their lateness. The section 3.3.6 further discusses how to deal with late joiners. After receiving the end signal, the auction participants can determine if they won the auction as there will be no further bidding. The period from when a new auction has been declared until the start signal is published to the auction room topic is referred to as the joining period. The active period is the time between the start signal and the end signal where participating Nodes may place their bids. The sum of the joining period and the active period is named as the auction period.

The top right diagram shows an automatic and inclusive system. The neighbouring Nodes are subscribed to all the auction room topics with a wildcard subscription because this is an automatic system. That includes Auction/Room 3 and the Nodes receive notice of the auction when Item 1 is published directly in the designated auction room topic. The bids are also placed in Auction/Room 3 such that all Nodes will be able to look at the bids posted for Item 1. The automatic participation auctions do not need a start signal because the participants are already subscribed to the auction room such that no early bids will be lost. The auctioneer will publish an end signal in automatic auctions as well. The suitable time to close an auction is discussed in the following section 3.3.3. There is no joining period for automatic auctions such that the active period is equal to the auction period.

The bottom left diagram shows an active and selective system. The system is selective because the Nodes are able to only subscribe to the auctions of services which they are interested in as indicated by ServiceX. ServiceX may be any service and items published on Auction/ServiceX will only contain items that require the processing of the service X. The item, Item 1, is posted on the selective topic Auction/Service X and contains the auction room topic, Room 3. The system is active, so the Nodes can decide if they would like to subscribe to Auction/ServiceX/Room

3 where the bids will be posted.

The bottom right diagram shows an automatic and selective system. The item, Item 1, is posted directly to the auction room topic Auction/ServiceX/Room 3. Only the Nodes that are subscribed for auctions for service X will be notified, however all of these Nodes will see the bids on Item 1 because this is an automatic system. The bids are made in Auction/ServiceX/Room 3.

Closing an Auction

The auctioneer has the responsibility of closing an auction and does so by publishing a message to the auction room that indicates that the auction has been closed. The ideal time to close an auction will depend on the bids and the class of service requested. If a highly satisfactory bid is made early on in the auction then there are only marginal and potential benefits to be gained from continuing to search for a more suitable candidate. Additionally, an auction for a service request that requires a low latency response should keep the active period short such that a winner can be determined quickly and the processing of the service can start without further delay. However, the auctioneer is faced with increased overhead if it must dynamically consider the bids that have been made and the class of the service. Therefore, an auctioneer can also be implemented to close an auction based on independent conditions, such as predetermined joining and active periods. It is also possible to implement a mixed strategy that includes observing how the auction unfolds in addition to having predetermined conditions for when an auction should close.

Deciding the Winners of an Auction

The winner of an auction is the Node that produced the highest bid for the item. The bids that are made in an auction are public such that all participating Nodes can see how their bid compares to the others. This allows the Nodes to declare the winner of the auction by themselves after receiving the end signal. The responsibility of handling the item is delegated to the winners of the auction.

Depending on the implementation, an auction may have multiple winners. The amount of winners will decide how the item is blooming across the Fog. The number of winners could be configured for the whole network as a general rule or be auction or item specific. If the number of winners is high for an auction then there will be many Nodes that are handling the same item and there is a high likelihood that the service will be processed shortly by either some of the winners of the current or future auctions. In contrast, if there is a single or fewer winners then there will be less load across the Fog which will spare networking resources and processing resources at the Nodes. But the requested service will take longer to be processed because there are fewer candidates.

The bidders in an auction will have to decide if they won the auction or not at the end of the auction. Any bid that is received by the bidding Nodes later than the end signal from the auctioneer will not be able to win the auction. A bid may be received after the end signal because they were either published too late or had a long transmission period to the broker on the auctioning Node. The process of deciding the winners of an auction should only consider the messages that were received by the Nodes through their subscriptions. In other words, if a Node makes a bid and the auction closes before the Node's own bid is echoed back through its subscription then it was too late to bid on the item and can therefore not win the auction. MQTT assures the order of messages will be consistent for messages that have the same level of QoS [13]. This means that the auctioneer will publish the bids in the same order as it receives them. However, it should be noted that this does not keep an accurate account of the real ordering that considers the time when the bids were sent by the Nodes.

There are some special cases to consider when deciding a winner that might make it troublesome to correctly decide on a set of winners. There are considerations that should be made with regards to multiple Nodes producing equal winning bids, when Nodes join auctions after it has already started, and the QoS used between the auctioneer and the bidders.

When there are more Nodes that have produced equal winning bids than the number of winners for that auction then there is an issue when selecting the winners. By the assumptions of how bids are defined in section 3.3.4, any of the Nodes can be chosen as winners because the Nodes are assumed to be equally capable because their bids are the same. Therefore, this could be resolved by simply choosing the Nodes that issued their bids first.

A Node may subscribe to the auction after it has already started such that the Node has not been notified of the previous bids that have been made before it joined the auction room. The resolution to this challenge requires some further interaction with the auctioneer. The Node which has joined late needs to see the previous bids that have been made such that it knows if its own bid beats these when a winner is decided at the end. The auctioneer must send at least the largest current bids to any Nodes that join an auction late. This is challenge is discussed in more detail in section 3.3.6.

If the QoS level set for the published bids or subscriptions to the auction rooms are set to 0 then some bids may not reach all of the participants. The result of not communicating all bids to the participants is that they will have an inconsistent history of the bids. This could potentially lead to more winners than expected in an auction. There will be too many winners if a Node has not received the winning bids in an auction and therefore declares itself as the winner. Therefore, a QoS of at least 1 should be set for the bids and the subscriptions to the auctioneer. This may lead to duplicate messages, but the size of all the bids will be received by the participants.

3.3.4 Bidding in an Auction

A Node shows its motivation to handle a request with the bid that it places in the auction. The size of the bid is determined by two components, the naive portion and the intelligent portion. The naive portion represents the Node's static capability to process the service which is associated with the item. The intelligent portion is a dynamic component that factors the current state of the Node in addition to surrounding factors. The Nodes may place their bids during the active period of an auction.

The following list is a summary of the factors that may be considered when a Node calculates its bid size for an item:

- If the Node has the service implementation installed
- The previous bids on similar items from neighbouring Nodes
- The current load and available resources of the Node
- The characteristics of the Node
- The weights of the service

The method that a Node uses to calculate its bid size is two-fold. The first part is similar to how the asking price was expressed in equation 3.1, this is referred to as the naive portion of the bid. The second portion of the bid is the intelligent portion and accounts for dynamic factors that affect a Node's capability to handle an item.

To calculate the naive portion of its bid, a Node inspects the auctioned item to determine the service class which is requested. The Node will input its characteristics and multiply these by the corresponding weights to produce its naive portion of the bid. The naive portion of the bid could be used as the final bid for a Node but it does not take into account the dynamic variables that affects the performance in terms of processing a service. The naive portion is therefore static and assumes that the Node has all of its computing power available.

The intelligent portion is a dynamic component that takes into account additional information that a Node may have at a certain time that should affect its bid size. This could be excessive current load, availability issues, scheduled downtime, etc. The implementation and magnitude of the intelligent portion can vary but it will add an additional component to the bid size that will either lower or increase the size of the final bid. The intelligent portion of the bid is included to make the bids more realistic in terms of what can be expected of a Node at the current time. The intelligent portion can also be used to determine a Node's capability of further propagating a request by including an expectation of the bids that would be present if it were to re-auction the item to its neighbours. As with the naive portion, presenting a practical method for calculating the intelligent portion is outside of the scope of this thesis.

$$\text{Bid} = \sum_{i=0}^{n} w_i c_i + C \tag{3.2}$$

The equation 3.2 is an expression for the bid size that a Node can provide for an auctioned item. The sum is similar to what was used in equation 3.1 for the asking price. This is the naive portion of the bid. The intelligent portion is represented by C.

An example scenario of when the intelligent portion should affect the final bid of a Node is when a neighbour is a good candidate for processing a service. Assume that two Nodes with equal characteristics are bidding against each other in an auction for an item. One of the Nodes, N1, has no neighbours that would bid above the asking price if the item was re-auctioned. The other Node, N2, has a neighbour, N3, that can provide a high bid on items of this kind. If both N1 and N2 were to use the naive portion as their final bid, then their bids would be equal. Assume that N1 and N2 can not bid above the asking price such that if either won the auction it would trigger a re-auction of the item. In this scenario, it is inherently better that N2 wins the auction because it has the motivated neighbour N3. However, N1 and N2 are seen as equal without including an intelligent portion. The information could be gained from historic events where N2 has noted that N3 has provided a high bid for items that are similar to the currently auctioned item. If N2 and N1 had derived their bids with an intelligent portion that included this information, then N2 would out-bid N1. However, when their bids are equal or N1 has a higher bid than N2 then a less optimal decision will be taken.

The MQTT protocol's QoS level should be set such that the auctioneers can assure the delivery of all bids to the Nodes that are participating in an auction. All published bids should be made with QoS 1 or 2 such that a Node knows that its bid has been sent successfully to the auctioneer and be guaranteed that its bid will be delivered to the other participants. The participants of an auction should also subscribe with a QoS of 1 or 2 such that they can be guaranteed that they will be aware of all the bids that have been posted to the auction. If QoS is set to 0 then some bids may unknowingly be lost and this will lead to uncertainty when deciding a winner. As mentioned in section 3.3.3, the order that the bids are received by the Nodes that subscribe to the auction is guaranteed by the auctioneer if all of the bids and subscriptions share the same QoS level. The QoS level should not be mixed because this could lead to inconsistent ordering.

3.3.5 Routing the Response Back to the Client

Some services may produce a response that should be routed back to the Client that made the request. The Fog delegates the requested service to a Node and wherever the Node resides it should be able to send the response back to the Client. The MQTT request and response pattern that was covered in section 2.2.3 can be used to implement this behaviour. The system architectures that is presented in this thesis uses multiple brokers and if the same network shall be used for propagating the response back to the Client then it may pass through a series of Nodes. Providing a method that uses the network in the Fog is essential because establishing a direct connection between the processing Node and the Client may be impossible. The response can be routed back to the Client in numerous ways and this section covers some routing schemes for responses.

Initially a service that generates a response starts as a request from a Client and is sent to the Node that the Client is connected to. The Node that first receives the request is referred to as the origin Node. The request includes a response topic that indicates where the Client expects the response to be published. The Client subscribes to the response topic on the broker that the origin Node is hosting. In the simplest case, the first Node that receives the request, the origin Node, is able to process the service and the resulting response is published on the response topic of the origin Node's broker which the Client is directly subscribed to.

An item can be routed deeper into the Fog through a series of auctions. The resulting response must in every case reach the response topic on the origin Node's broker that the Client is subscribed to. The first Node to publish the response will always be the Node that has processed the service. However, there are multiple options regarding where the response should subsequently be sent. Three natural candidates are the brokers hosted on either the processing Node, the previous auctioneer, or the origin Node. Any of these options has their own rules that decide how the response is routed back to the Client.

The origin Node is the final Node which the response shall be published onto. In some cases, it is possible to include the address of the origin Node as part of the request such that the processing Node can connect and directly publish the response to the origin Node. This is a simple solution but assumes any Node can reliably connect to the origin Node. Depending on the nature of the Nodes that assumption might not be satisfied.

The response can be routed back to the origin Node by using the connections or paths that are already present in the Fog. When doing so, the response follows the same path back to the Client as the item took to the Node, however in the opposite direction. The processing Node can either publish the response onto itself or to the previous auctioneer.

When a processing Node publishes the response back onto itself then it is assumed that the previous auctioneer has subscribed to it. This is ensured by introducing a rule that every auctioneer should connect and subscribe to the winners of an auction that they have held if the item will produce a response. When the previous auctioneer receives the response that was posted on a winner's response topic, then it will publish the response back onto its own broker. This method will follow the trail back until the origin Node receives the response and publishes the response onto its own response topic which the Client is subscribed to. The method where a Node publishes the response onto its own topic is referred to as self-publishing. The drawback of the self-publishing method is that an auctioning Node is not necessarily managing MQTT clients that are connected to the winning Nodes.

The figure 3.5 shows an example where a Client requests a service that generates a response and the response has to be routed back to the Client from a Node which it is not connected to. The example uses the self-publishing response method. The figure includes a Client and two Nodes N1 and N2. The Client publishes its request to N1 and subscribes to its response topic. Then N1 decides that it will not carry out the processing and auctions the item. N2 wins the auction

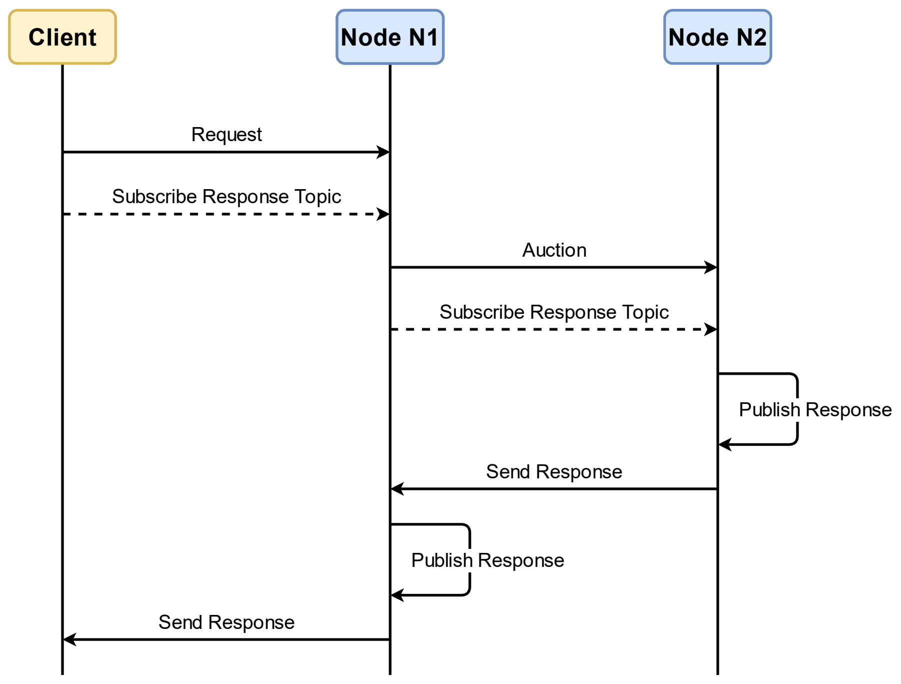

Figure 3.5: The self-publishing response method. The auctioneer subscribes to the winning Node's response topic. The processing Node publishes the response to its own broker.

and N1 subscribes to a response topic on N2. The processing of the service is performed by N2 and it publishes the response onto the response topic on its own broker. Since N1 is subscribed to N2's response topic, the response is sent to N1. Upon receiving the response N1 re-publishes the response onto its own broker. The Client is subscribed to N1 and is therefore sent the response.

A second option that keeps the existing connections, without further potential new connections, is that the processing Node will publish the response back to the previous auctioneer. To route the message back to the Client a rule must be made such that all Nodes that turn into auctioneers will have to subscribe to a response topic on their own broker if the item generates a response. The auctioneer will subscribe to the response topic and the winner of the auction will publish its response back to the auctioneer. When the auctioneer receives the response, it re-publishes the response to its previous auctioneer if applicable. The Client will be notified when the response reaches the origin Node. The origin Node does not have an auctioneer that was previous to it and will therefore not re-publish the response, unless the Client has subscribed to a response topic that is different from what was used between the Nodes. Then the origin Node will publish the response to that topic. This response propagation method is referred to as self-subscribing.

The figure 3.6 shows an example of how the response is routed back to the Client where the processing Node publishes the response back to its previous auctioneer. The figure illustrates the self-subscribing response method. The figure

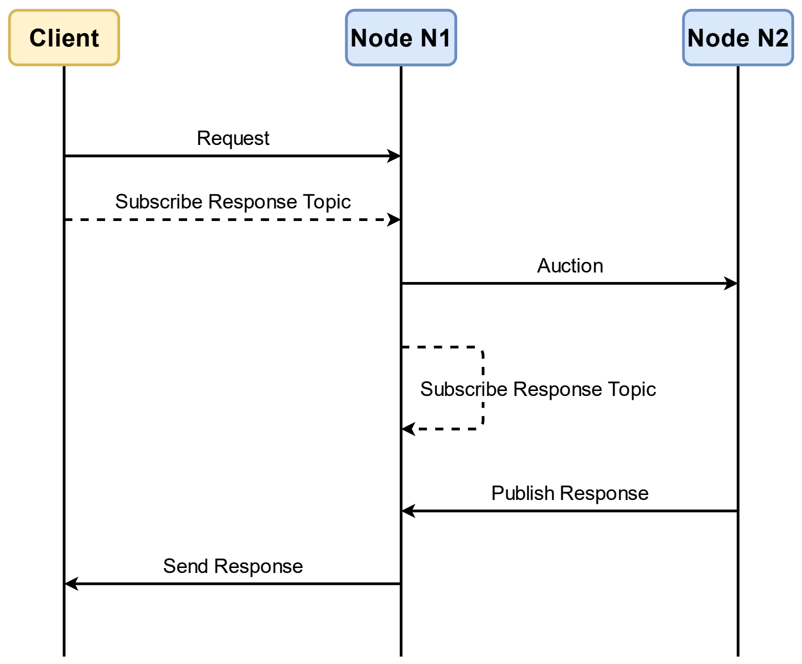

Figure 3.6: The self-subscribing response method. The auctioneer subscribes to its own response topic. The processing Node publishes the response to its previous auctioneer.

contains a Client and two Nodes N1 and N2. The Client requests a service that will generate a response message on N1 and subscribes to the response topic on N1. The item is auctioned to N2 because N1 decides to not process the service itself. After the item has been auctioned, N1 subscribes to its own response topic. The service is processed by N2 and the response is sent to the response topic on N1. The response is sent to the Client because it is subscribed to the response topic on N1. Note that, N1 is the origin Node and therefore does not have any previous auctioneers. Because N1 is the origin Node it does not re-publish the response. In addition, N2 does not subscribe to itself because it did not auction the item and is therefore not expecting any responses for this item. If N2 had to re-auction the item, it would subscribe to a response topic hosted on its own broker.

If multiple requests are expected to share the same response topic then correlation data should be included to distinguish between the responses. Each request and auctioned item can have correlation data attached to it, so if there is a collision on the response topic then the correlation data may be used to determine if the response should be re-published by the Node or accepted by the Client.

3.3.6 Joining Late to an Auction

For active participation auctions as seen in 3.4 the Nodes subscribe to the auction room when they receive notice. It could happen that a Node joins an auction late and therefore will not see the bids that were posted before it arrived. Additionally, the Nodes that participate in active auctions will be unaware that they have joined late without further mechanisms to alert them of this. Therefore, active auctions should include a start signal that indicates that the auction has begun and that the Nodes can start placing their bids on the item. The period between when the Nodes are notified of a new auction and the start of the auction is referred to as the joining period for that auction. The joining period can be set uniquely for every auction and the duration will affect how much the delay the auction causes for the processing of the service.

Nodes that subscribe after the joining period will not receive any previous bids to their subscription. To address this issue, the joining Node can publish a message to the auctioneer which indicates that it was late to the auction room and has not been sent the bids yet. The only bids that the joining Node needs to be aware of is the currently highest bid for auctions with a single winner and a set of the highest bids if there may be multiple winners with unequal bid sizes. The auctioneer will have to respond with a message that contains the currently highest bid or bids for that auction. Alternatively, an active auctioning system could be implemented such that Nodes that miss the start signal are not allowed to join the auction.

For automatic participation auctions this is not an issue since the neighbouring Nodes are already subscribed to the auction room topic and have received all the previous bids. The bidding Nodes may in the case of a temporary disconnect from the auctioneer during an active auction receive the full message history if they have set their MQTT connection to be persistent [13].

Chapter 4

Results

An implementation of an auction-based task assignment system was developed for this thesis. The auction approach propagates requests to Nodes by selecting the most capable Node that is available at every step. The approach and its components were outlined in the previous chapter. Two additional approaches, the modified Battistoni and random choice, were also implemented with the intent of comparing their performance characteristics with the auction implementation. The modified Battistoni is an assignment method that is inspired by the approach that is outlined by Battistoni et al. in their paper on off-loading edge devices [1]. It is a competition-based system that favors a rapid response time more than the conservation of resources. The random choice approach propagates the responsibility of handling a request to a neighbouring Node in a random fashion. These methods were chosen as comparisons because they test two important aspects of the auction implementation with regards to performance. The modified Battistoni was selected to see the performance trade-off in response time that is made when assigning requests to individual Nodes in contrast to any that are able. The random choice approach was selected to see if a demonstration program based on auctions with naive bids would outperform a random assignment method. A set of experiments were performed to compare the implementations. Each of the experiments changed a single independent variable that influences the performance of the implementations. The variables that were experimented with were the mean bid size of Nodes, the standard deviation of bids, and the network topology.

This chapter covers the design of the auction, random choice, and modified Battistoni implementations and a comparison of their performance in a series of experiments. It starts of with the common design aspects of the implementations. The following sections cover the specific details of each implementation. The final parts of this chapter explain how the performance of the implementations were compared and the results of the experiments.

4.1 General About the Implementations

The task assignment implementations that were developed for this thesis share numerous similarities in the way that they were developed, in attributes that define their behaviour, and factors that affected their performance in the experiments. This section intends to provide an overview of the characteristics that are common for the auction, random choice, and modified Battistoni implementations. All of the implementations were developed to represent systems that might have been deployed in a distributed fog computing context by using functional MQTT clients. However, note that the MQTT messages are passed through a single broker and not multiple. The system was made such that they can be installed onto a physical multi-brokered test-bed with only slight modifications to the topic schemes and broker addressing. However, the performance of the implementations were gathered through experiments that were performed on a single virtual machine. The implementations are using MQTT as their messaging protocol and the Nodes are represented as MQTT clients. The capabilities of the Nodes were represented as an integer and is referred to as their bid, however only the Nodes in the auction implementation communicate their bids. This section is dedicated to covering these and other similarities between the implementations in more detail.

4.1.1 Development

The Nodes, Client, and other components made for this thesis were programmed in Python. The Nodes and Client are MQTT clients and the MQTT client library Paho version 1.6.1 was used to implement the specifications that are required for communication with the MQTT protocol [20]. The open-source Mosquitto MQTT version 2.0.13 broker was used to govern the MQTT traffic [21] and all the traffic is directed through a single broker. The source code for the implementations and supporting software is publicly uploaded as a repository to Github [22]. An excerpt of the main program file and a description is also included in the appendix A.

4.1.2 Simulating Multiple Brokers

The implementations were designed to work with a single broker instead of each Node hosting its own broker. Each Node is assigned with a top-level topic which represents their broker. This was done in order to simulate the intended construction of the Nodes in a distributed application setting. The MQTT traffic that is under a top-level topic of a Node is interpreted as traffic that would be local to the Node's broker. The Client and Nodes are connected to the broker such that there are no MQTT connections between the entities. Any entity that is supposed to be connected to a broker that is hosted by another entity will instead subscribe and publish to the relevant top-level topic. For all of the implementations the Nodes subscribe and publish with QoS 2. Additionally, there was no network latency between the entities when the experiments were performed because the broker was run on the same hardware as the Nodes and the Client.

4.1.3 Simplifications of Services

The type of service that was implemented for this thesis is simplified compared to the service types that were covered in section 2.3.1. The service implementation offers no practical use cases beside being used to compare the performance of the implementations in a series of experiments. The intent of the experiments was to measure the processing and completion time that is associated with a request. Therefore, the service implementation was only required to have two key attributes, a standard processing time and an asking price.

The standard processing time attribute serves as a baseline for how long a Node will process the service from start to finish. When a Node starts to process the service, it will calculate its actual processing time in accordance with the expression found in equation 4.1 and set a timer that expires after the processing period has ended. When the timer expires, the requested service is interpreted as being completely processed and the request is satisfied. The asking price determines which Nodes that will be selected to do the processing. The asking price of the service is a one-dimensional integer that lies somewhere in the bid range of the Nodes. The service type that was implemented has no associated weights and neither do the implemented Nodes have specialised characteristics that would differentiate their capabilities of carrying out varying types of services. The asking price of the service is a static parameter that is determined by the service configuration and can not be changed by the Client or Nodes.

The service type that was implemented does not induce a response that would be routed back to the Client upon completion. In addition, the task assignment implementations were not developed to offer services that would require them to send a response back. Therefore, the request-response pattern that was covered in section 2.2.3 and 3.3.5 is not offered by the implementations.

4.1.4 Simplification of Bids

The implementations use bids as a measure of their capability in regards to processing a service. To extensively include the features of the bid as they were explained in section 3.3.4 would increase the complexity of the implementations to a large extent and would introduce more variables to consider during the comparisons. Therefore, the bids were designed such that they are one-dimensional, static, and to only include the naive portion of the bid. Considering the simplifications, the bids still represent a Node's capability to process a given service. That is because the bid directly affects the processing time. The calculation of the processing time that was used for this thesis is expressed in equation 4.1.

The bid size is one-dimensional and is represented as an integer in the range of 0 to 100. The services themselves do not have any weights associated with them and neither do the Nodes have characteristics that make them objectively better suited for processing a given service. The amount that a Node is willing to bid on a service is directly tied to the service that is requested and is pre-configured, e.g., Node N1 is configured such that it will have a bid of 42 for the service A. The bids are static and the intelligent portion of the bid has not been implemented. Which means that each Node bids without accounting for dynamic and surrounding factors such as current load and the expected capabilities of its neighbouring Nodes. The Nodes are therefore completely naive and their bid sizes do not change in response to any events during operation.

4.1.5 Calculating the Processing Time

The same expression for calculating the processing time was used for all of the task assignment implementations that are covered in this thesis, namely the auction, random choice, and modified Battistoni. The Nodes in all of the implementations have a bid attribute that represents their capabilities and is used to calculate the time it takes to process a requested service.

$$T = T_0 \cdot \left(1 - k \cdot \frac{\text{Bid} - 50}{50}\right) \tag{4.1}$$

The equation 4.1 is the expression for how the processing times are calculated by the implementations. The processing time refers to how long it takes for a certain Node to complete the processing of a service. The processing time T is related to the bid which the Node made for the item. The standard processing time of a service T_0 is defined in the catalogue of services and is the time it would take for a Node that produces a bid of 50 to complete the processing of the given service. The processing time T will be longer for Nodes with bids below 50 and shorter for bids above. The size of any bid is defined to be in the range of $[0, 100]$. The factor k determines the impact that the bid size has on the processing time and is a constant in $[0, 1)$. If k is zero then the bid size has no effect on the processing time. Note that if the bid of a Node is 50, then $T = T_0$ and that $0 < T < 2T_0$ for all bids and values of k.

4.2 Auction Implementation

This section covers the implementation of an auction-based approach to assigning requests in the Fog. The implementation was made with the intent of providing a demonstration program and to show the capabilities of the system in comparison to other approaches. The previous chapter on Method provided a general outline of the behaviour and components of any auction approach. The implementation that was made is a only a demonstration and therefore some of the covered features were either simplified or not implemented. Most importantly, the auction implementation propagates requests by holding local auctions between neighbours and the most capable Nodes are assigned with the responsibility of further handling the auctioned request.

This section starts of with a top level view of the implementation by exploring the rules regarding the auction. The rules are followed by a summary of the simplifications that were made regarding the bid sizes of the Nodes. Then the topic structure of the auction implementation is covered. Lastly, this section provides an example of the auction implementation in action which is intended to provide clarity as to how the auction implementation decides on a suitable Node to process a requested service.

4.2.1 Rules

This section provides an overview of how the auction implementation operates. The auction implementation is an active and inclusive auctioning system such that the Nodes themselves have to decide if they want to participate in an auction. The auctioneer is in charge of declaring a new auction, starting it, and finally ending it. In addition, the Nodes in the auction implementation must follow a set of rules that are specific to the particular implementation that is featured in this thesis.

These are the rules for the implemented auction-based task assignment system:

1. All Nodes are assumed to carry the requested services.
2. If a Node wins an auction with a bid below the asking price, then it should re-auction the item to its neighbouring Nodes.
3. The Nodes should wait with publishing their bids until they see the start signal in the auction room.
4. The Nodes can determine if they were a winner at the end of the auction which is marked by an end signal.
5. The auctioneer participates with a bid in its own auctions.
6. The auctioneer will process the service if it wins the auction and not re-auction the item.
7. The auctioneer will process the service if it is the only bidder in its auction.
8. There will only be multiple winners to an auction if there were multiple equal bids that were winning.

The list above provides an overview of the rules that are applicable for the auction implementation. It starts of with an assumption that all Nodes carry the requested services in rule 1. This assumption is made to avoid the increased complexity of handling requests that have been propagated to the end of the Fog without having found a suitable match. In this case the last Node must process the requested service and is therefore required to carry its implementation. This behaviour is covered by rule 7 that states that the auctioneer must process the service given that no other Nodes have bid on the item. The premise of the auctioning approach is that an unsuitable Node should re-auction an item to its neighbours, this is covered by rule 2. The rules 3 and 4 cover how a Node that participates in auction knows when it supposed to publish its bid and when to determine a winner. The implication of rule 3 is that a Node is unable to bid in an auction if it subscribed after the start signal was sent. This was done to alleviate the need for mechanisms that would inform late joiners of the currently highest bids, how to handle late joiners was discussed in section 3.3.6. The rules 5 to 7 describe how auctioneers participate in the auctions which they are holding. In the implementation made for this thesis, an auctioneer will bid in its own auctions as stated by rule 5. According to rule 6, the auctioneer will process the service if it wins its own auction regardless of its bid size. This is to avoid further propagation to neighbours that are seen as more unsuitable for processing the service. Lastly, rule 8 clearly states an exception to the general behaviour of the auction implementation. Primarily, each requested service is only handled by a single Node. However, in the case where multiple Nodes provide equal winning bids, then they are all tasked with the responsibility of treating the request. This rule is in place such that the winning Nodes do not have to make an agreement between themselves.

4.2.2 Topic structure

The auction implementation is an active and inclusive system. Such that when a Node auctions an item it will notify all of its neighbours. On receiving the notice of the auctioned item, the neighbouring Nodes can decide to participate in the auction or not. If they wish to join the auction, then they must subscribe to the dedicated auction room topic. Nodes that do not participate in the auction will not be notified of any events occurring in the auction room. The other types of auctioning systems were explained in section 3.3.3.

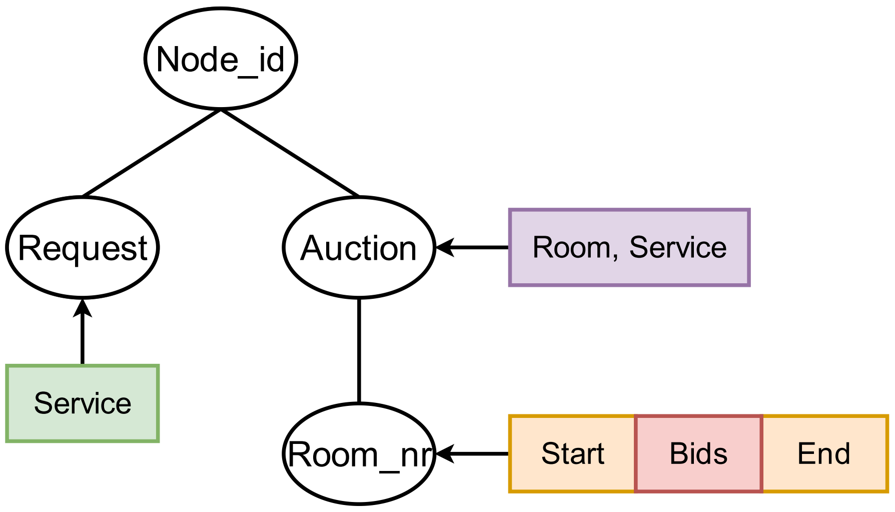

Figure 4.1: A visualisation of the topic structure used for the auction implementation and common messages that are sent on these topics. The auction implementation is an active and inclusive auctioning system.

The figure 4.1 provides a visual representation of the topic structure that was used for the demonstration. As mentioned in section 4.1.2, each Node is given a top level topic that represents their own broker. Each Node is initialised with a subscription onto their id as their top level topic, in addition to a wildcard subscription for all lower level topics. An origin Nodes receives requests from Clients on the request topic. If a Node is unable to process the requested service, then it must notify its neighbours of a new auction by publishing onto the auction topic which its neighbours are subscribed to. The message from the auctioneer contains the service class in question and a room number that indicates which topic the auction will be held in. The neighbours that wish to participate in the auction subscribe to the auction room topic that was indicated by the room number. In the auction room topic, the first message to appear is the start signal from the auctioneer. The participating Nodes may then begin to publish bids onto the auction room topic. The auction closes when the auctioneer publishes a message that marks the end of the auction. When the auction has ended, the participating Nodes will individually determine the winner.

4.2.3 Practical Example

This section covers an example case of the auction implementation handling a request from a Client. This has been included to provide clarity surrounding the order of operations and how the auction implementation acts in response to an incoming request. The example covers a Fog that consists of three auction Nodes, where one of the Nodes act as the origin Node and the rest are direct neighbours to origin Node. The origin Node has an insufficient bid and auctions the request to its neighbours. The neighbours compete in the auction and determine a winner.

The figure 4.2 provides a visual look on how the auction implementation handles an incoming request. There are three Nodes, N1, N2, and N3, in addition to a Client. Each Node is initialised by a wildcard subscription to all topics on their broker that is represented by their id in the top level topic. The topology of the Fog is formed by N2 and N3 subscribing to the auction topic of N1. After doing so, N2 and N3 are ready to receive notice of any auctions that will be held by N1. The Client publishes a request for service A onto the request topic of N1. After deciding to re-auction the item because it can not bid above the asking price, N1 publishes a message containing the requested service class, A, and a room number, 0, onto its own auction topic. N2 and N3 receive the message because they are subscribed to the auction topic of N1. They both wish to participate in an auction for a service of class A and subscribe to auction room 0. The auctioneer N1 publishes the start signal that indicates the start of the auction in room 0. Then N2 publishes a bid of 55 and N3 bids 52. The auctioneer N1 also publishes a bid of 48 in its own auction in case its the highest bidder. All of the participating Nodes are aware of the bids made by any other Node because they are subscribed to the auction room topic. The auction continues, but is in this example idle, until N1 publishes the end signal. The participating Nodes are now individually able to decide a winner of the auction. Because the auction has closed, N2 and N3 are no longer required to subscribe to the auction room and unsubscribe from it such that it may be reused at a later point. The highest bidder and winner of the auction was N2, it has a bid above the asking price and therefore processes the service A.

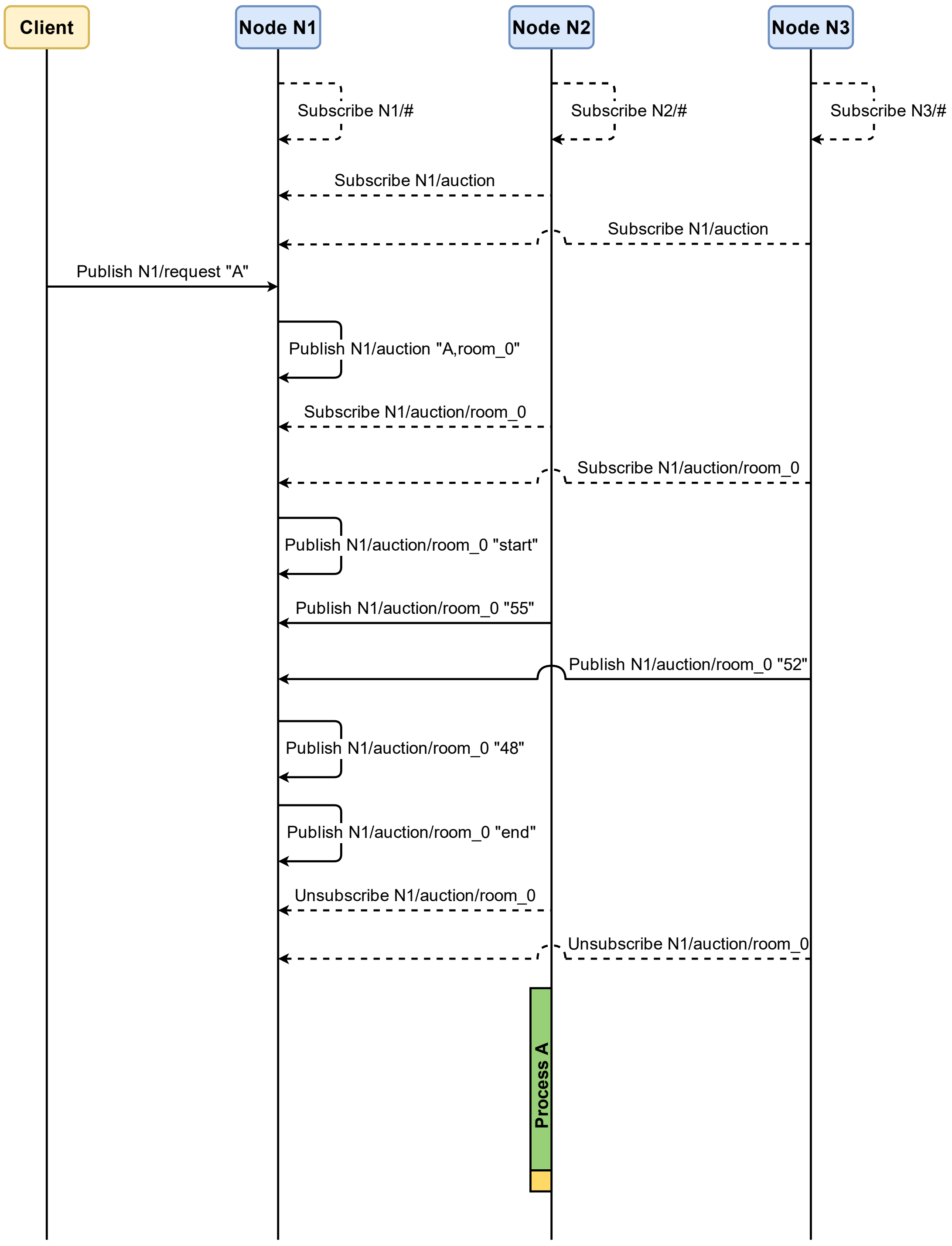
Client
Node N1
Node N2
Node N3
Subscribe N1/#
Subscribe N2/#
Subscribe N3/#
Subscribe N1/auction
Subscribe N1/auction
Publish N1/request "A"
Publish N1/auction "A,room_0"
Subscribe N1/auction/room_0
Subscribe N1/auction/room_0
Publish N1/auction/room_0 "start"
Publish N1/auction/room_0 "55"
Publish N1/auction/room_0 "52"
Publish N1/auction/room_0 "48"
Publish N1/auction/room_0 "end"
Unsubscribe N1/auction/room_0
Unsubscribe N1/auction/room_0
Process A

Figure 4.2: A sequence diagram that shows the flow of messages in the auction implementation. The messages that are caused as a result of subscriptions are not included.

4.3 Random Choice Implementation

A random approach to task assignment was one of two methods that were compared to the auction implementation that was covered in section 4.2. The random choice implementation randomly assigns a request to a neighbour if the original recipient of the request has a bid that is below the asking price. To allow for a direct comparison to the auction implementation, the asking price of a requested service was used to determine if a Node in the random choice implementation should process the task or further propagate to its neighbouring Nodes. Each random choice Node is assigned a non-communicated bid size which they compare to the asking price of an incoming request.

4.3.1 Rules

The random choice implementation is publish-based in contrast to how the implemented auctioning approach relies on subscriptions for task assignment. For the random choice implementation a Node that decides to propagate a request randomly selects one of its neighbours to publish the request to. To keep track of the Nodes neighbours, all Nodes carries an internal list of neighbouring Nodes which they may propagate requests to. When propagating requests, a Node publishes a similar message as a Client would. As a result, the propagated request from a Node is handled in the same fashion as the initial request from a Client.

To summarize the behaviour of the random choice Nodes, a list of rules has been added below. Rule 1 states that the general condition for when a Node should process or propagate a request. Rule 2 decides how random choice Nodes decides on a neighbour to propagate a request to, this is done in a random fashion. Rule 3 is an edge case where a request has been propagated to the end of the network and the last remaining choice is for the currently responsible Node to process the request.

The rules for the random choice Nodes:

1. A request should be propagated if the responsible Node has a bid on the service that is below the asking price.
2. A Node must randomly select a neighbour to publish the request to when propagating.
3. A Node must process the service if it has no neighbours to propagate a request to.

4.3.2 Topic Structure

The figure 4.3 shows the topic structure that was used for the random choice implementation. Each Node has a wildcard subscription to a top level topic which is given by their id. The requests from Clients or neighbouring Nodes are received on the request topic. The message contains the requested service class. To build a list of neighbours that each Node stores internally, a connect topic is used. A neighbouring Node may request to be appended as a neighbour by publishing their own id to another Node's connect topic.

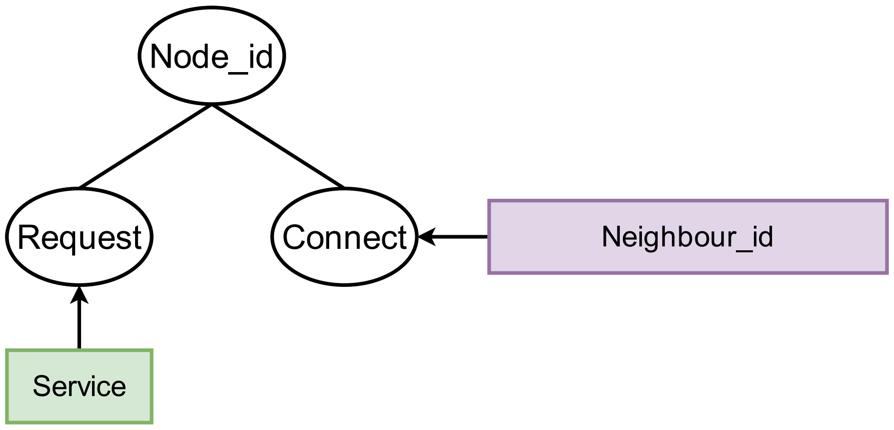

Figure 4.3: A diagram of the topic structure for the Nodes that handle requests by assigning randomly them to their neighbours.

4.3.3 Practical Example

The figure 4.4 provides an overview of the behaviour of the random choice implementation in light of an example. The network is initialised with three Nodes N1, N2, and N3. Each Node makes a wildcard subscription to its designated top level topic as indicated by its own id. N2 and N3 are then added to N1's neighbour list by publishing their ids to N1's connect topic. The Client publishes a request to N1 on the request topic. The request is for service A. N1 receives the request and decides that it should propagate it to one of its neighbours because its internal bid is lower than the asking price of the request. The decision between N2 and N3 is randomly made, but in this case N1 propagates the request to N3. This is done by publishing to the request topic of N3 with a message containing the requested service. N3 starts to process the request either because it has no neighbours or because it has a bid above the asking price associated with service A.

4.4 Modified Battistoni Implementation

The task assignment system designed by Battistoni et al. was adapted such that the modified Battistoni implementation more closely resembles the design of the auction and random choice implementations. The changes that were made affect the topic structure, subscription patterns, and the way Nodes connect to each other. However, the competitive behaviour of the Nodes have been kept such that the fundamental strategy of the original design remains in the modified version. The modified Battistoni implementation allows multiple Nodes to process a request

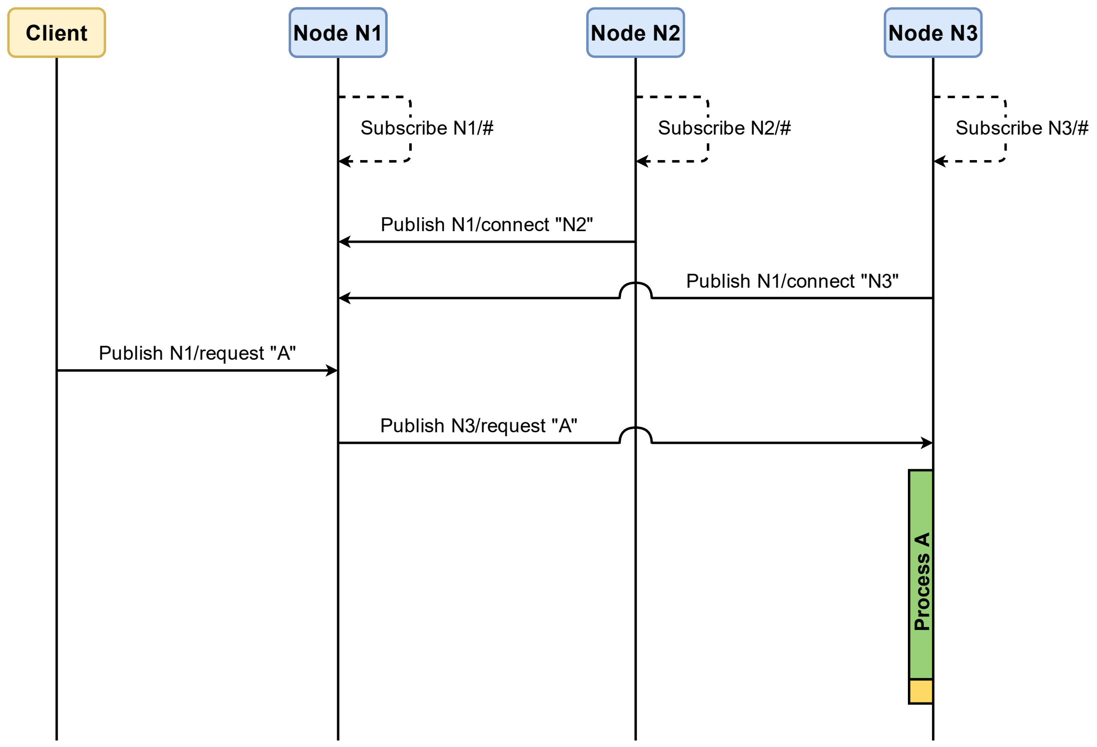

Figure 4.4: A sequence diagram that shows the behaviour of the random task assignment implementation. The messages that are caused as a result of subscriptions are not included.

simultaneously in contrast to the auction and random choice implementations where only a select few Nodes end up in the processing stage.

The original system was intended to be used for edge computing and offloading purposes. Therefore, there is a difference in the naming convention used for the Clients and Nodes when comparing the descriptions found in this thesis and the original paper. In the original descriptions the clients are the main source of processing power and a node is serving as a host for an MQTT broker. Further on, this thesis addresses the processing entities of the modified Battistoni implementation as Nodes and views the MQTT broker as its own instance. This is to keep the naming convention consistent throughout the discussions that compare the implementations and to avoid confusion when explaining the implementation of the modified Battistoni. To more closely follow the Node design for auction systems that was explained in section 3.2.2, the modified Battistoni Nodes are designed such that they also host their own MQTT broker.

4.4.1 Rules

In the original design, the Nodes may decide based on their current capabilities if they want to assist by processing the task related to a request. The capabilities of a Node will be affected by the hardware, load, etc. In the auctioning approach proposed in this thesis, the current capabilities are represented by the bid size. Since the concepts are similar, each modified Battistoni Node carries an internal and un-communicated numerical bid for each service class. The modified Battistoni Nodes will opt out of processing a task if the asking price of the request is lower than their bid. Therefore, all Nodes in the modified Battistoni implementation will process a request when notified if they have a bid above the asking price.

The modified Battistoni Nodes are not collaborating and therefore they do not notify each other when they start processing a service. It might happen that all of the modified Battistoni Nodes opt out of processing a request and the request goes unprocessed. To ensure that all requests are eventually processed an addition to the original design was made. When the origin Node in the modified Battistoni implementation has a bid below the asking price it will generally not process the request. However, if the request has not completed processing after the worst case guaranteed by the expression 4.1 then the origin Node will start processing. The processing of the task is done with the assumed capabilities of the origin Node such that processing time will vary according to the bid size of the origin Node. The origin Node will set a reminder when it receives a request from a Client to check if the request is still active after the worst case processing time has elapsed.

The rules for the modified Battistoni Nodes:

1. A Node propagates all incoming requests by publishing a room number to its service topic.
2. When propagating a request, the Node subscribes to its associated room topic.

3. A Node will subscribe to the room topic on the propagator whenever it receives a propagated request.
4. A Node must process the service if it has a bid above the asking price.
5. When a Node has finished processing a service it publishes an end signal to its own room topic.
6. When a Node receives an end signal on its own room topic, it unsubscribes from its own and propagator's room topic and publishes an end signal to its propagator's room topic.
7. When a Node receives an end signal on its propagator's room topic, it publishes the end signal in its own room topic.
8. A Node stops the processing of a service if it receives an end signal in the associated room topic.
9. The origin Node must process a requested service if it has not received an end signal after the worst case period has elapsed.

The table above has been included to provide an overview of the behaviour of the modified Battistoni Nodes. Any time a Node receives a request from either a Client or another Node it should propagate the request by publishing a room number onto its service topic. This is covered by rule 1. The room number that is published refers to a room topic that the Node will associate with the request. As per rule 2, the Node subscribes to the room topic after it has propagated the request. If it was a propagated request, then rule 3 applies and the Node should also subscribe to the propagators room topic. In the modified Battistoni implementation, all Nodes that have a bid above the asking price shall process the request. This is stated in rule 4. After a Node has finished processing a request then it notifies Nodes that are subscribed to its room topic by publishing an end signal according to rule 5. Rule 6 states that when a Node receives an end signal on its own room topic then it unsubscribes from both its own and its propagator's room topic, it then publishes an end signal to the propagator's room topic. This ensures that the propagator is notified as well, and the propagator will in turn notify its own propagator. Rule 7 states that when an end signal has been received on a propagator's room topic, then the Node will publish the end signal to its own matching room topic as well. The Node will subsequently receive an end signal on its own room topic and follow rule 6. According to rule 8, a Node shall stop its ongoing processing of a service if it receives an end signal in the associated room topic. This ensure that unnecessary resources are not spent on requests that have already been satisfied. Rule 9 is an insurance that guarantees that all requests will at some point be satisfied by the origin Node.

4.4.2 Topic Structure

The proposed topic scheme and subscription pattern in the original paper suggests that a Client will only handle one request for a specific service at the time. This implies that there will be no concurrent requests of the same service class at any time. The topic structure was modified to allow for concurrent requests instead of guaranteeing no such cases. Therefore, the topic structure for the modified Battistoni implementation uses the same strategy as the auction implementation that dedicates a room topic to active requests.

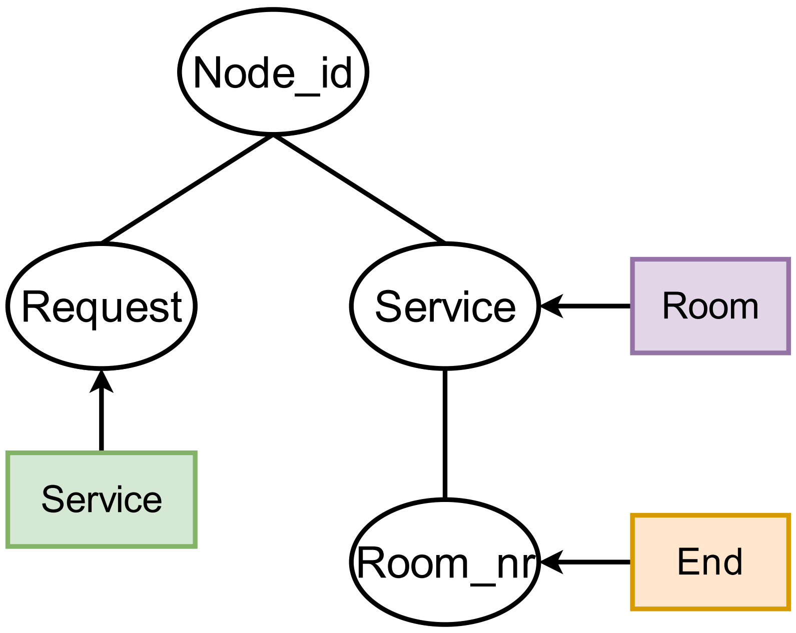

Figure 4.5: A diagram of the topic structure for the Nodes that are in the modified Battistoni approach.

The figure 4.5 provides an image of the topic structure that was used for the modified Battistoni implementation. Each Node has a corresponding top level topic that is given by their id. The top level topic represents the brokers that each Node would be hosting. Each Node is subscribed to their own request topic. The requests from Clients are published on the request topic with a message indicating the service class to process. Upon receiving a request, a Node will publish to the appropriate service topic. The messages that are published to a service topic includes a room number that the Node associates with the corresponding request. The room number points to an active room topic that the Node subscribes to. It knows that the request has been processed once a message containing an end signal is published in the active room topic.

4.4.3 Practical Example

The figure 4.6 shows an example of the modified Battistoni implementation where there are three Nodes, N1, N2, and N3 and where a Client requests a service A. The network has been initialized by N2 subscribing to any service topic on N1, and N3 doing the same on N2. Note that, each Node is also subscribed to their own request topic. The Client issues a request by publishing to N1's request topic. The message contains a request for service A. Upon receiving the request, N1 chooses

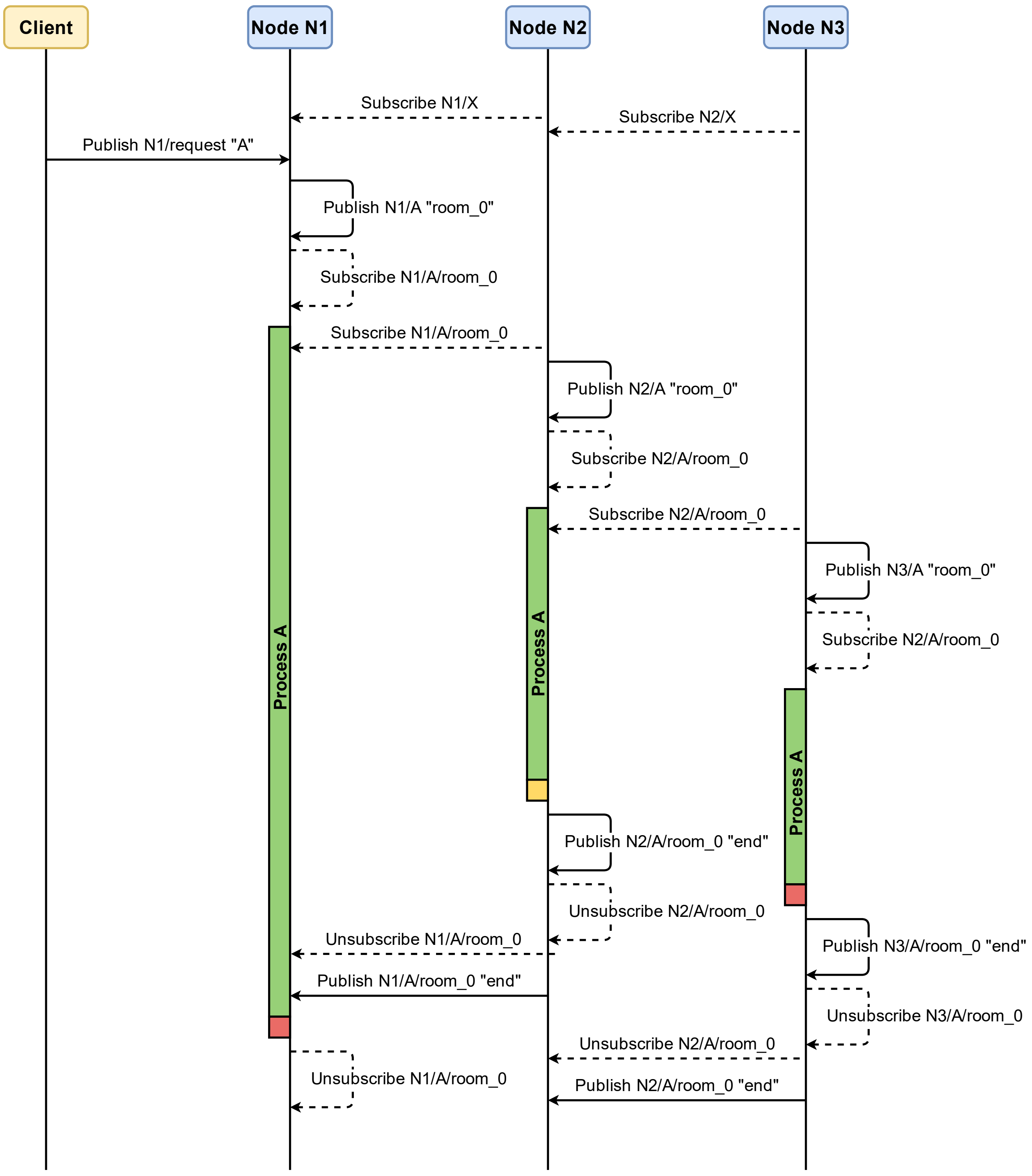

Figure 4.6: A sequence diagram that shows the flow of messages in the modified Battistoni implementation. The messages that are caused as a result of subscriptions are not included.

an available room number and publishes to the service topic that is dedicated to service A. The message contains a room number that points to the active room topic which N1 associates with the request. After publishing the information on which active room topic N1 will be using, N1 starts processing service A because it has a higher internal bid than the asking price of the request. The neighbour, N2, was notified of the request because of its subscription to N1's service topic. N2 subscribes to the the active room topic on N1 such that it is notified whenever an end signal is published. After doing so, N2 decides on a room number that it wishes to associate with the request and publishes a message onto its own service topic. N2 subscribes to its active room topic and starts processing. N3 do the same steps upon receiving notice of the request as N1 and N2 previously performed and starts processing. The first to complete the processing of service A is N2 and it indicates that it has done so by publishing an end signal to the active room topic on itself. N2 then unsubscribes from the topic. When N2 was notified of the request it subscribed to an active room topic on N1, because it has completed the task it may unsubscribe from that topic as well. After doing so it publishes an end signal onto the active room topic on N1 as well, such that N1 and any other subscribers are notified of the completion. When N1 receives the end signal, it aborts its own processing and unsubscribes from the active room topic it hosts. Because N3 is subscribed to the active room topic on N2 it was notified when N2 published its end signal. In response, N3 aborts its processing and publishes an end signal to the active room topic it hosts before unsubscribing from it. The active room topic on N2 may also be unsubscribed from. Because N3 does not regard who the end signal came from in the modified Battistoni implementation it publishes to the active room topic on N2 to ensure that N2 and any other subscribers are notified.

4.5 Design of the Experiments

This section covers how the auction, random choice, and modified Battistoni implementations were compared and the results of the comparisons. The implementations have been tested with various configurations of network topology, mean bid sizes, and standard deviation of bids in a series of experiments. The data that was gathered from the experiments are the total processing time that is spent when handling a request and the completion time which is the elapsed time from when a service is requested until it has been processed.

This section starts of with how the data was acquired with the request monitor. That is followed by how the experiments were designed and performed. This covers the default configuration and the variations that were tested. The data that was gathered from the experiments was used to compare the implementations and the results are found in section 4.6.

4.5.1 Measuring the Total Processing and Completion Times

Two types of data was gathered from the experiments to compare the performance of the implementations. The intent of the experiments was to gather data that reveals the behaviour of the task assignment implementations when responding to a request from the Client under varying configurations. The collected data was the total amount of time that was spent on processing the requested service and the time it takes from when a service has been requested until it is processed. The total processing time was chosen because it reveals how much resources that is spent on processing a task by all of the Nodes. An approach like the modified Battistoni will generally have multiple Nodes processing a request, while the random choice will dedicate the processing to a single Node. The efficiency of the approaches may be compared by looking at the total processing time. The response speed of the implementations was compared by looking at the completion times of the requested services. The completion time is unrelated to the amount of Nodes that are processing a request and only accounts for the first completion. Most importantly, the completion time gives an indication of how much time it takes to find a suitable Node and the quality of that Node. For instance, the time it takes to auction an item will be present in the time it takes to complete a request, however not in the time it takes for each Node to process it. The total processing time and the completion time of a request was collected by the request monitor. The request monitor is an independent entity that the Client and Nodes notify whenever they perform an action on a request.

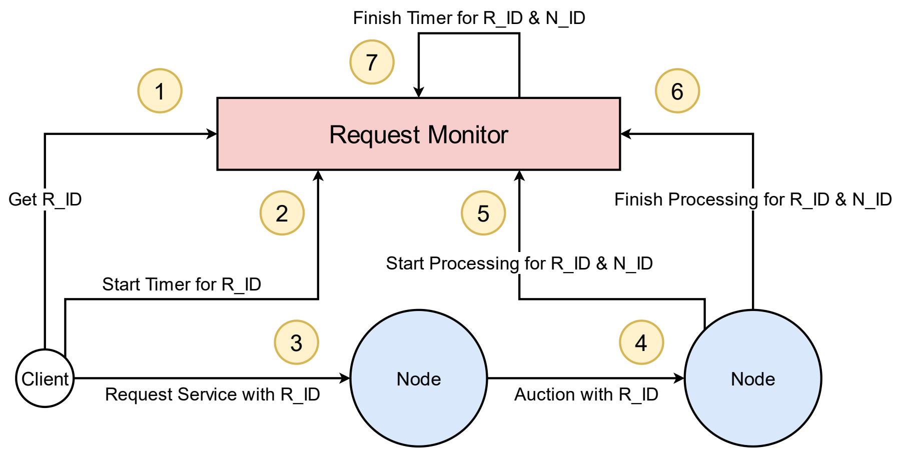
Finish Timer for R_ID & N_ID
7
1
6
Request Monitor
Get R_ID
2
5
Finish Processing for R_ID & N_ID
Start Processing for R_ID & N_ID
Start Timer for R_ID
3
4
Client
Node
Node
Request Service with R_ID
Auction with R_ID

Figure 4.7: A diagram for how the processing times for each request was kept track of for the demonstration. This diagram is drawn in a context for the auctioning approach.

The request monitor as seen in figure 4.7 measures the time it takes between a request is sent and the service is processed. It allows for multiple Nodes to process the request and stores each processing time. The request monitor is reachable for all Nodes and Clients. The Client will get a request ID from the request monitor before it publishes its request to the origin node. When the request is published, the Client will also start a timer for the request by notifying the request monitor. The request ID is part of the message such that the Nodes are able to communicate which request they are processing to the request monitor. A Node that processes a service notifies the request monitor before it starts the processing and also when it finishes. In this way, the request monitor can keep a history of how much time a Node spent on processing each request. In addition to this, the request monitor will also track the first completion of a request. In this way it is possible to measure the fastest time to serve a request and the total time that was spent processing it.

The results of the experiments are stored in csv files that has information regarding the completion time and the total processing time for each request that was made. Each file contains a history of multiple requests that were made under different configurations and each line is referred to as a run.

4.5.2 Comparison Method

Three experiments were conducted for this thesis. For each experiment, a single independent variable in the configuration was changed while keeping everything else equal. The intent of the experiments was to gather data about the performance of the implementations under various configurations. The data that was collected is total processing time and completion time which were measured with the request monitor. The experiments were done on an Microsoft Azure `D2s_v3` virtual machine running an Ubuntu 18.04 operating system. The mean bid size, the standard deviation of bids, and the network topology of the connected Nodes in the Fog were the independent variables that were explored by the experiments performed for this thesis. All of the implementations were tested on the same hardware with equal configurations. They also use the same formula calculate their processing times, the expression can be found in equation 4.1. The same random seed was used throughout the experimentation for setting the bid sizes of the Nodes such that the results can be reproduced.

Static parameters and their value	
Parameter	Value
Asking Price	50
Standard Processing Time, T_0	3 seconds
Processing Constant, k	0.5
Number of Nodes	15
Joining period	0.1 seconds
Active period	0.1 seconds

Table 4.1: An overview of the static parameters that was used for the experiments and their values.

During the experiments a set of static parameters were used throughout. The table 4.1 provides an overview of the static parameters. The asking price of a request is tied to the service that is requested. The service that was requested stayed the same throughout the experiments and had an asking price of 50. The standard processing time of the service was 3 seconds. The processing constant k was set to 0.5 which indicates that the range of the processing times lies between 0.5 T_0 and 1.5 T_0 for all possible bid sizes. The amount of Nodes in the Fog was set to 15 for all of the experiments, however the effect of changing the topology has been explored. The joining period for the auction implementation is 0.1 seconds, which is the period before an auctioneer publishes the start signal to its auctions after it has notified its neighbours. Additionally, the active period of each auction is also 0.1 seconds.

Variable parameters and default value		
Parameter	Variations	Default
Topology	Network 1, 2, 3	Network 1
Mean bid size	30, 50, 70	50
Standard deviation of bids	10, 20, 40	20

Table 4.2: An overview of the variable parameters that was used for the experiments and their default values.

The values for the independent variables that were used for the experiments are featured in table 4.2. Each experiment was performed with three different values for each of the independent variables. While an experiment for an independent variable was done, the values of the remaining variables were set to their default value. The network topology was chosen to see how the implementations would respond to a differently structured Fog. For instance, a deeper network might require the implementations to propagate the request further to find a suitable match because for each layer there might not be a candidate Node with a bid larger than the asking price. The mean bid size was chosen as an independent variable for one of the experiments to determine how the average capabilities of the Nodes in the Fog will influence the performance of each implementation. A lower mean bid size means that fewer Nodes in the Fog will have a bid which is larger than the asking price of the request. Note that the distribution of bid sizes in the Fog are only centered about the mean bid size, such that some Nodes may still have bids that are below the asking price even when the mean bid size is larger. The standard deviation of the bids in the Fog will determine variance in capability between the Nodes. A higher standard deviation indicates that there is a larger discrepancy between the capabilities of the Nodes and therefore an approach that actively selects the best Nodes, for instance though an auction, might do better than a random approach. The standard deviation of the bids was chosen as an independent variable for one of the experiments to see how the implementations performed in response to a smaller and greater difference between the capabilities of the Nodes.

The figure 4.8 provide a visual representation of the networks that were used for the experiments. All of the networks contain 15 Nodes and the origin Node is

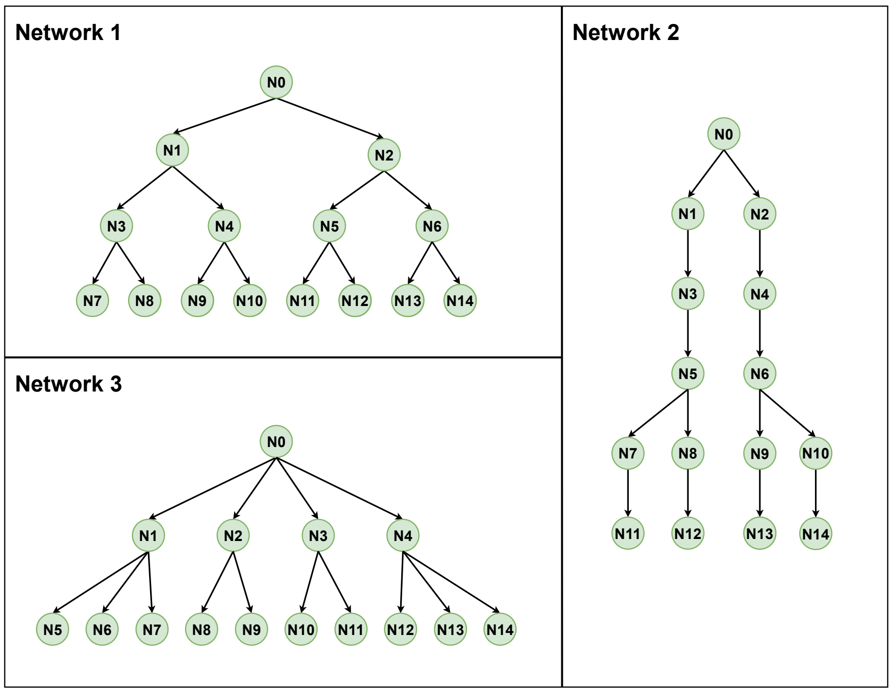

Figure 4.8: A visualisation of the three network topologies that were used in the experiments. Network 1 is the default network.

the root of the structure. Network 1 is a full binary tree with four layers and acts as the default network. Network 2 has a topology that is deeper than Network 1 and Network 3 is the widest topology that has been tested.

A total of three experiments were performed where each experiment tested a single independent variable, either the network topology, the mean bid size, or standard deviation of bid size. Each of the experiments were divided into three sessions where each of the sessions had a set value for the independent variable that was tested. For instance, one of the sessions that are associated with testing the network topology have the network topology of Network 3. The sessions are a collection of runs. A run is defined as a single request from a Client on a given configuration. Each time a new run is performed, every corresponding Node for all of the implementations are instantiated with the same bid sizes. Such that the state of each implementation is equal and the only factor that separates them for a given run are the respective task assignment methods. Note that the state of the implementations will vary for each run because the bid sizes are drawn from a distribution, however the implementations will always be tested on the same set of states. For the experiments performed in the context of this thesis, each session consists of 500 runs. Each run will carry information regarding the completion time and total processing time that was spent handling the associated request.

4.6 Results of Experiments

The comparisons of the implementations are divided into the three experiments that were performed in accordance with the previous section. They were made by testing the implementations with equal states on a varied set of configurations. This ensures that the difference in total processing time and completion time between the implementations are as a result of their method for assignment. The static parameters that were used for the experiments our found in 4.1 and the variable parameters in 4.2. Diagrams for the experiments are included in the following subsections and feature the average values of the performance metrics for each experiment. The diagrams have been created with the matplotlib library in Python.

The results of these comparisons show that the auction implementation has a significantly longer completion time in most cases when compared with the modified Battistoni implementation and that it has comparable completion time performance to the random choice implementation, however the auction was consistently slower by a small margin. The total processing time of the auction implementation is comparable to that of the random approach and the modified Battistoni has a substantially longer processing time than the other implementations.

4.6.1 Experiment: Network Topology

An experiment was dedicated to researching the effect that the network topology would have on the performance of the implementations. The different network topologies that were used for this experiment had an equal number of 15 Nodes, but had different connections. The networks that were used for this experiment were made such that they would show how the implementations would behave in response to a wider and deeper network compared to the default of Network 1.

The figure 4.9 displays the average completion times that the implementations had for three sessions in this experiment. The colour of the bars represent the auction, random choice, and modified Battistoni implementations as green, yellow, and red respectively. In all of the sessions, the auction implementation had the highest average completion time followed by random choice which performed slightly better. The modified Battistoni clearly had the lowest completion times in this experiment. Changing the network topology between Network 1 and 2 had no significant effect on the behaviour of the implementations. However, for the widest network, Network 3, the auction performed slightly better and random choice slightly worse. The result is that they had nearly identical completion times for the session that used Network 3.

The table 4.3 shows some statistical measurements that are included to give some insight into the completion times of the implementations and their response to the tested network topologies. As previously mentioned, the auction imple-

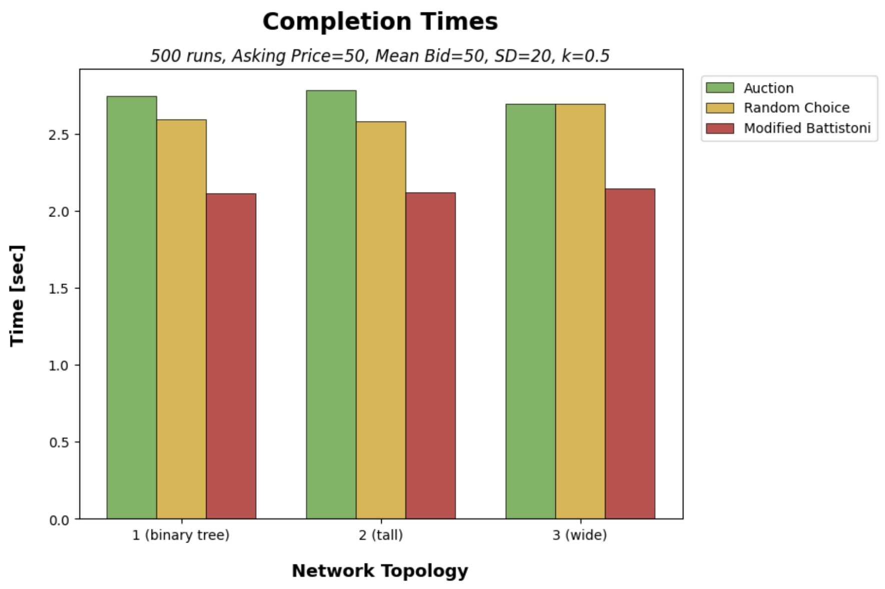

Figure 4.9: A bar chart of the completion times for the various task assignment methods for Network 1, 2, and 3.

Measurement	**Auction**	**Random Choice**	**Modified Battistoni**
Network 1			
Mean Completion Time (s)	2.74	2.59	2.11
Min Completion Time (s)	1.58	1.57	1.53
Max Completion Time (s)	3.93	4.51	2.94
SD Completion Time (s)	0.45	0.39	0.27
Network 2			
Mean Completion Time (s)	2.78	2.58	2.12
Min Completion Time (s)	1.54	1.54	1.54
Max Completion Time (s)	4.22	4.24	2.93
SD Completion Time (s)	0.46	0.37	0.27
Network 3			
Mean Completion Time (s)	2.70	2.69	2.15
Min Completion Time (s)	1.53	1.54	1.55
Max Completion Time (s)	4.06	4.26	2.92
SD Completion Time(s)	0.38	0.46	0.27

Table 4.3: A table of the obtained statistics on completion times for the implementations on networks 1, 2, and 3.

mentation had the longest completion times on average. It had a mean completion time that was up to 0.2 seconds slower than the random choice and up to 0.66 seconds slower than the modified Battistoni implementation. The auction performed the worst on Network 2, which is the deepest network. The minimum completion time for the implementations were close in all of the sessions, but the the modified Battistoni performed the best in terms of maximum completion time. The modified Battistoni implementation had the smallest standard deviation in completion times for all of the sessions, it was consistently 0.27 seconds. The auction implementation had the highest standard deviation for the completion times for Network 1 and 2, but was lower than random choice for Network 3.

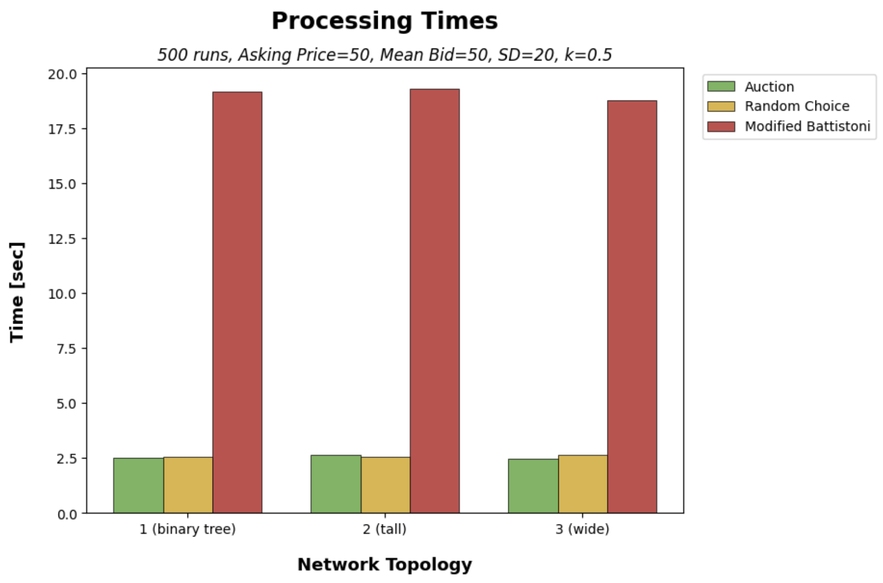

Figure 4.10: A bar chart of the processing times for the various task assignment methods for Network 1, 2, and 3.

The figure 4.10 illustrates the average processing times that were measured on the different network topologies. The processing times for the modified Battistoni is substantially higher than the other implementations. The modified Battistoni implementation had a slightly lower average processing time on the widest topology, Network 2, compared to the other topologies. The auction implementation had marginally better performance than random choice on Network 1 and 3, but random choice had a lower average processing time on Network 2. In general, the processing times for the implementations did not change to a great extent between the sessions.

The table 4.4 has statistical measurements on the processing times of the implementations from this experiment on network topologies. In all of the sessions, the modified Battistoni implementation had the longest processing time by a large margin. It had its longest mean processing time of 19.29 seconds on Network 2, which is the deepest network, and its shortest mean processing time of

18.78 was on Network 3. Additionally, it had the largest standard deviation in processing times as well. The modified Battistoni had its largest standard deviation of 4.32 seconds on Network 2 where it had a minimum processing time of 8.46 seconds and a maximum of 33.00 seconds. Which results in a gap of 24.54 seconds between its longest and shortest processing time. The random choice implementation had the lowest standard deviation in processing times in all sessions. It was comparably similar to the standard deviation measured for the auction on Network 1 and 3. However, the auction implementation had a significantly higher standard deviation on Network 2. In this session, the auction had a maximum processing time of 8.91 seconds which compared to the random choice implementation in the same session is more than double. The reason for the large discrepancy is likely because the auction implementation chose to process with multiple Nodes during a run. In general, the processing times of the auction and random choice were fairly equal, however the auction implementation had the longest maximum processing time in all sessions.

Measurement	**Auction**	**Random Choice**	**Modified Battistoni**
Network 1			
Mean Processing Time (s)	2.54	2.57	19.20
Min Processing Time (s)	1.56	1.56	9.82
Max Processing Time (s)	4.62	4.38	30.90
SD Processing Time (s)	0.39	0.38	4.18
Network 2			
Mean Processing Time (s)	2.64	2.56	19.29
Min Processing Time (s)	1.53	1.53	8.46
Max Processing Time (s)	8.91	4.11	33.00
SD Processing Time (s)	0.61	0.36	4.32
Network 3			
Mean Processing Time (s)	2.49	2.67	18.78
Min Processing Time (s)	1.53	1.53	7.93
Max Processing Time (s)	6.06	4.23	30.66
SD Processing Time (s)	0.49	0.45	4.23

Table 4.4: A table of the obtained statistics on processing times for the implementations on networks 1, 2, and 3.

4.6.2 Experiment: Mean Bid Sizes

The amount of Nodes that have a bid size above the asking price of a request is related to the mean bid size of the participating Nodes. In addition, the processing time of a service is tied to the bid of the processing Node. Therefore, the mean bid size of the Nodes was used as factor to determine the overall capabilities of the Fog to handle requests. This experiment was aimed at researching how the performance of the implementations would vary according to the mean bid size of the Nodes. In each session, the mean bid size of the Nodes was set to a value that was either below, exactly, or over the asking price of the request. For all of the experiments, the bid size for each Node was drawn from a truncated normal distribution. Therefore, some Nodes may have bid below the asking price even when the mean bid size was larger.

The results of this experiment that tested the implementations on a few selected mean bid sizes showed that the average completion times get significantly lower for all implementations when the mean bid size increases. Additionally, it showed that the total processing time of the modified Battistoni implementation increased when more Nodes had a bid above the asking price. However, the auction and random choice implementations had lower total processing times as a response to an increase in the mean bid size.

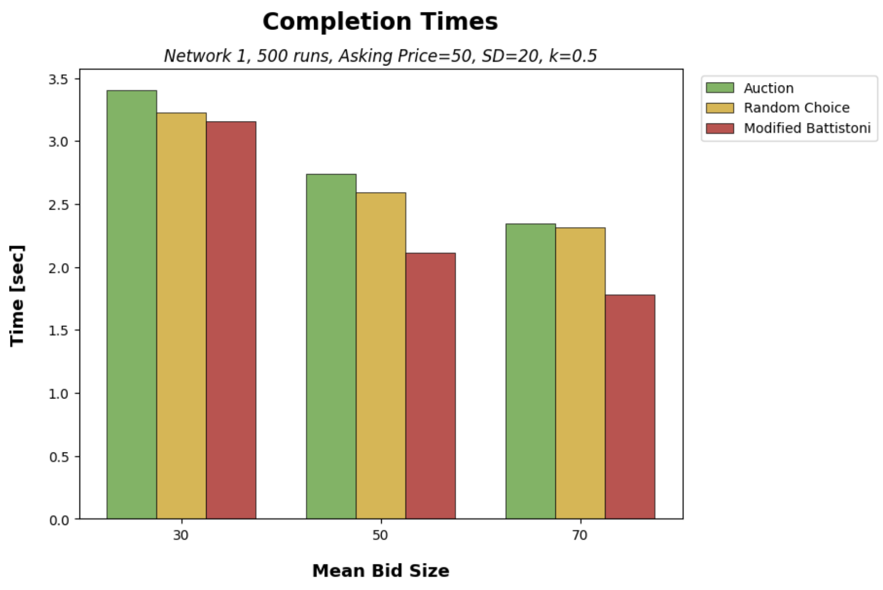

Figure 4.11: A bar chart of the completion times for the various task assignment methods for when Nodes have a mean bid size of 30, 50, and 70.

The figure 4.11 visualises the completion times for the implementations over three sessions that had mean bid sizes of 30, 50, and 70. The asking price of the requests are static and has been configured to be 50. For all of the implementations, the completion time got lower when the mean bid size was increased. The auction implementation had the longest average completion time in all of the sessions. Random choice had the second best results and the modified Battistoni had the lowest average completion time for all sessions. The modified Battistoni implementation showed substantially better performance than the others for the sessions with mean bid sizes of 50 and 70. However, random choice was close in completion time when the mean bid was 30.

The table 4.5 provides statistical measurements of the completion times from the sessions. It shows that the standard deviation in completion time tends to

Measurement	Auction	Random Choice	Modified Battistoni
Mean Bid Size = 30			
Mean Completion Time (s)	3.40	3.23	3.16
Min Completion Time (s)	1.86	1.81	1.62
Max Completion Time (s)	4.81	4.62	10.10
SD Completion Time (s)	0.59	0.63	1.90
Mean Bid Size = 50			
Mean Completion Time (s)	2.74	2.59	2.11
Min Completion Time (s)	1.58	1.57	1.53
Max Completion Time (s)	3.93	4.51	2.94
SD Completion Time (s)	0.45	0.39	0.27
Mean Bid Size = 70			
Mean Completion Time (s)	2.34	2.31	1.78
Min Completion Time (s)	1.51	1.50	1.50
Max Completion Time (s)	3.38	3.01	2.43
SD Completion Time (s)	0.40	0.39	0.16

Table 4.5: A table of the obtained statistics on completion times for the implementations when the mean bid size of the Nodes are 30, 50 and 70.

decrease as the mean bid size is increased. The modified Battistoni was heavily affected by this trend. It had a lower standard deviation than the rest when the mean bid sizes were 50 and 70. However, when the mean bid size was 30 the standard deviation of the modified Battistoni was 1.90 seconds. That is substantially larger than the standard deviations in completion time for the auction and random choice, which were 0.59 and 0.63 seconds respectively. The reason for this may be due to how the implementations handle requests that fail to find a Node that has a bid above the asking price. The auction and the random choice implementations will force the last responsible Node to process the service while the modified Battistoni implementation delegates this responsibility to the origin Node that starts processing once it is certain that no other Nodes will process it.

The figure 4.12 provides a view on the total processing times of the implementations and how they were affected by changing the mean bid size of the Nodes. The modified Battistoni had a substantially longer total processing time compared to the other implementations. The processing time for the modified Battistoni implementation increased with the mean bid size. In contrast, the auction and random choice implementations had a reduction in processing time as the mean bid size grew larger. The performance in terms of processing time for the auction and random choice implementations were close to equal in all of the sessions.

The table 4.6 provides statistical measurements on the total processing times that were gathered from the sessions. The mean of the total processing times for the modified Battistoni was much longer compared to the other methods. For a

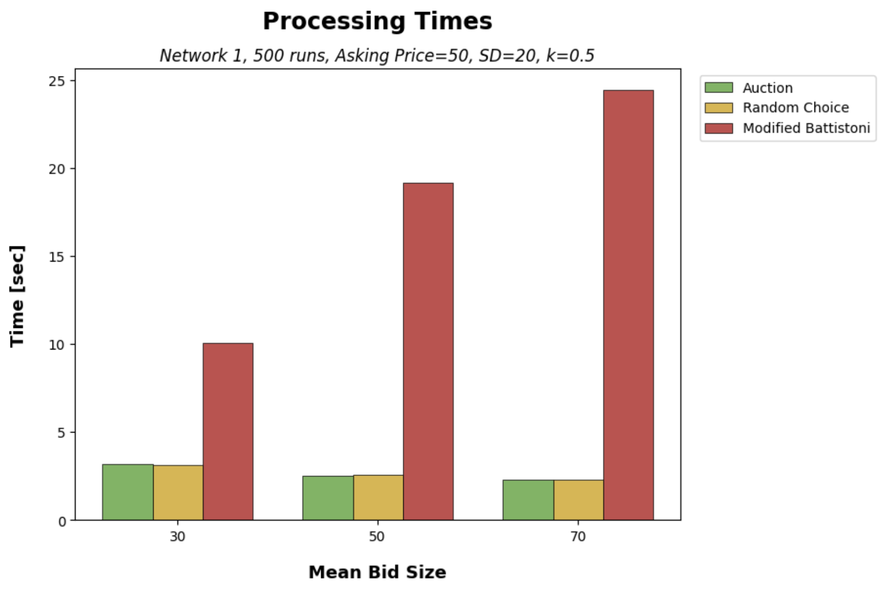

Figure 4.12: A bar chart of the processing times for the various task assignment methods for when Nodes have a mean bid size of 30, 50, and 70.

Measurement	**Auction**	**Random Choice**	**Modified Battistoni**
Mean Bid Size = 30			
Mean Processing Time (s)	3.18	3.16	10.06
Min Processing Time (s)	1.56	1.80	2.98
Max Processing Time (s)	10.98	4.50	22.47
SD Processing Time (s)	1.10	0.60	3.67
Mean Bid Size = 50			
Mean Processing Time (s)	2.54	2.57	19.20
Min Processing Time (s)	1.56	1.56	9.82
Max Processing Time (s)	4.62	4.38	30.90
SD Processing Time (s)	0.39	0.38	4.18
Mean Bid Size = 70			
Mean Processing Time (s)	2.31	2.31	24.44
Min Processing Time (s)	1.50	1.50	16.01
Max Processing Time (s)	5.28	3.01	37.12
SD Processing Time (s)	0.45	0.39	3.08

Table 4.6: A table of the obtained statistics on processing times for the implementations when the mean bid size of the Nodes are 30, 50 and 70.

mean bid size of 70, the modified Battistoni implementation spent on average 24.44 seconds processing a request. In comparison, the auction spent on average 2.31 seconds and the random choice 2.31 seconds processing a request in the same session. The auction implementation had in all sessions a maximum processing time that was longer than random choice, the difference was substantial for the sessions with mean bid sizes of 30 and 70. In the session with a mean bid size of 30, the auction implementation had a maximum processing time that was more than double compared to the random choice. However, the mean processing times of the auction and random choice implementations were close to equal for this and other sessions. The reason might be that the auction implementation was able to select a better candidate to do the processing in some of the runs and that this brought their mean processing times closer despite a worse maximum performance.

4.6.3 Experiment: Standard Deviation of Bids

The bid size for each Node, regardless of implementation, is sampled from a truncated normal distribution which is centered around the mean bid size. Each bid can not be less than 0 or more than a 100. However, the shape of the distribution and the likelihood to get a sample close to the mean bid size is determined by the standard deviation that is used for the bid size. This will impact the spread of the bid sizes but will not affect the mean. A higher standard deviation will result in Nodes that have a larger variance in their bid sizes and may represent a heterogeneous network of Nodes that have significantly different capabilities. This experiment explored the effect that altering the standard deviation of bid sizes will have on the performance of the implementations.

The results of the experiment are that the average completion time of all the implementations was shorter when a larger standard deviation was used for the bids. In accordance with this, the results showed that the completion time is longer for a smaller standard deviation. The trend for the total processing time is similar. A larger standard deviation will result in a shorter total processing time. However, only the processing time of the modified Battistoni implementation was substantially affected by this.

The figure 4.13 provides an overview of the average completion times that the implementations had when the standard deviation of bids were set to 10, 20, and 40. The completion times of the implementations were the longest for the session that had the lowest standard deviation. As the standard deviation increases, the average completion decreases. The auction implementation has the longest completion time in all of the sessions, followed by the random choice and the modified Battistoni implementation which was the fastest. The effect of changing the standard deviation are similar for all of the implementations. However, the modified Battistoni approach performs better with a higher standard deviation compared to the other implementations.

The table 4.7 includes statistical measurements of the completion times that

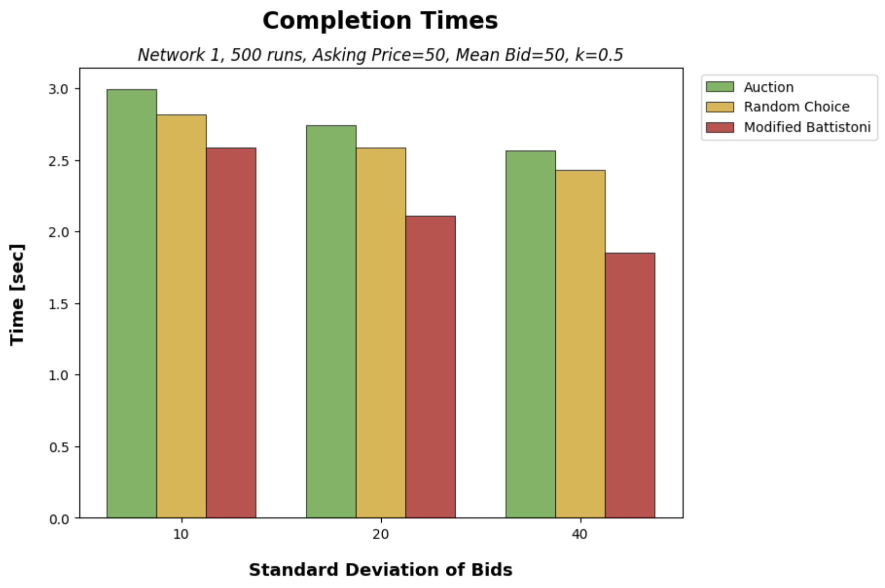

Figure 4.13: A bar chart of the completion times for the various task assignment methods for when the Nodes bid size has a standard deviation of 10, 20, and 40.

Measurement	**Auction**	**Random Choice**	**Modified Battistoni**
		SD Bid Size = 10	
Mean Completion Time (s)	2.99	2.82	2.59
Min Completion Time (s)	2.11	2.11	2.02
Max Completion Time (s)	3.97	3.90	2.97
SD Completion Time	0.31	0.21	0.17
		SD Bid Size = 20	
Mean Completion Time (s)	2.74	2.59	2.11
Min Completion Time (s)	1.58	1.57	1.53
Max Completion Time (s)	3.93	4.51	2.94
SD Completion Time	0.45	0.39	0.27
		SD Bid Size = 40	
Mean Completion Time (s)	2.57	2.43	1.86
Min Completion Time (s)	1.51	1.51	1.51
Max Completion Time (s)	4.30	4.57	2.80
SD Completion Time	0.51	0.52	0.23

Table 4.7: A table of the obtained statistics on completion times for the implementations when the bid size of the Nodes have a standard deviation of 10, 20, and 40.

were gathered in the sessions. The minimum completion time was similar for all of the implementations, however the modified Battistoni has the lowest maximum completion time in all of the sessions. For the session that had selected bid sizes with a standard deviation of 40, the maximum completion time of the auction was 4.30 seconds, the random choice was 4.57 seconds, and the modified Battistoni 2.80 seconds. In this session, the modified Battistoni was 1.77 seconds faster than the random choice implementation. The standard deviation of the completion times were in most sessions largest for the auction implementation. However, the random choice implementation showed a trend of increasing standard deviation in completion time as the standard deviation of bids increased. When the standard deviation of bids were 40, the auction and random choice implementations had effectively equal standard deviation in completion times. In general, the modified Battistoni was the most consistent as it had the lowest standard deviation in completion times in all of the sessions.

The figure 4.14 shows an overview of the average total processing times that the implementations had when they were tested with different standard deviations of bids. The modified Battistoni implementation had the longest average processing time by a large margin. The auction and random choice implementations were close to equal in processing times. A higher standard deviation of bids resulted in lower average total processing times for the implementations. The modified Battistoni implementation was the most affected by this trend.

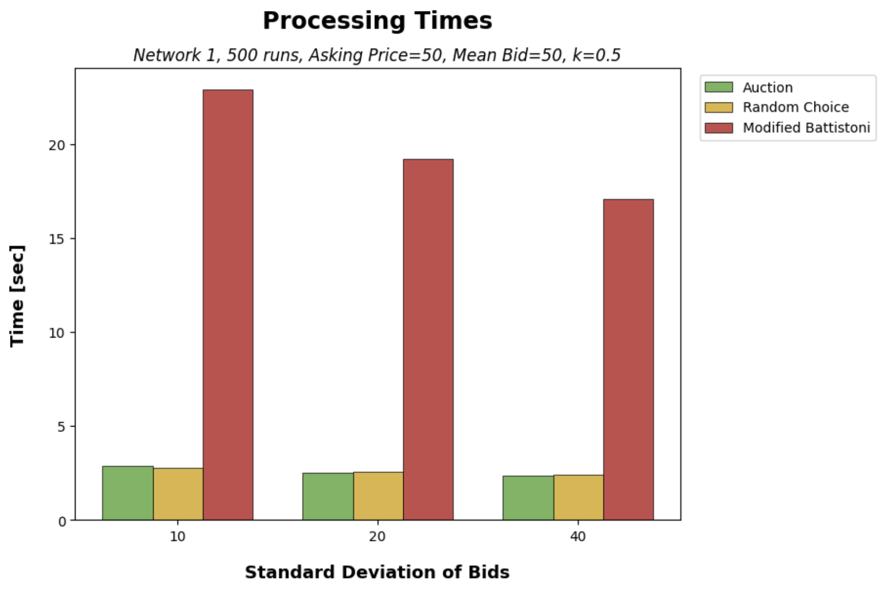

Figure 4.14: A bar chart of the processing times for the various task assignment methods for when the Nodes bid size has a standard deviation of 10, 20, and 40.

The table 4.8 provides an overview of the statistical measurements on total processing time that was collected from the sessions. The maximum processing times for the auction implementation are slightly or substantially higher compared to the random choice implementation. However, the differences between their mean processing time are insignificant. Collectively, the Nodes in the modified Battistoni implementation spent substantially more time processing compared to the other methods. In the session where the standard deviation of the bids were set to 10, the modified Battistoni had a mean processing time of 22.86 seconds while for the auction it was 2.86 seconds. This mean that the modified Battistoni implementation spent 8 times more processing time on average than the auction implementation in this session. The standard deviation in processing time became lower for the modified Battistoni implementation as a result of the standard deviation of bids getting larger. For the random choice it was the opposite, the standard deviation in processing time got larger with an increase in standard deviation of bids. The auction implementation had its lowest standard deviation of processing times, 0.39 seconds, when the standard deviation of bids were 20.

Measurement	**Auction**	**Random Choice**	**Modified Battistoni**
		SD Bid Size = 10	
Mean Processing Time (s)	2.86	2.80	22.86
Min Processing Time (s)	2.10	2.11	8.75
Max Processing Time (s)	9.00	3.87	36.57
SD Processing Time	0.57	0.20	4.95
		SD Bid Size = 20	
Mean Processing Time (s)	2.54	2.57	19.20
Min Processing Time (s)	1.56	1.56	9.82
Max Processing Time (s)	4.62	4.38	30.90
SD Processing Time	0.39	0.38	4.18
		SD Bid Size = 40	
Mean Processing Time (s)	2.38	2.41	17.07
Min Processing Time (s)	1.50	1.50	7.97
Max Processing Time (s)	5.88	4.44	29.87
SD Processing Time	0.55	0.51	3.32

Table 4.8: A table of the obtained statistics on processing times for the implementations when the bid size of the Nodes have a standard deviation of 10, 20, and 40.

Chapter 5

Discussion

This thesis has explored the auction approach, which is a decentralised task assignment method. The approach was designed with the publish-subscribe pattern of MQTT in mind and the presented use case is for propagating requests in a fog computing architecture. The auction approach was implemented as a demonstration program and it was tested against other decentralised task assignment methods in a series of experiments. This chapter covers a discussion surrounding the results of the experiments, the potential benefits of using MQTT in fog applications, and a final evaluation of the auction approach.

5.1 Discussing the Results of the Experiments

A functioning demonstration program of the auction approach was developed for the purpose of showing the feasibility of an implementation. The auction approach was then tested against other task assignment systems in a series of experiments and the results were promising. The auction implementation did not include sophisticated features that would be present if the approach was used in a real system. However, the basic implementation served as a baseline that could be compared to other decentralised approaches for propagating requests. The findings in section 4.6 showed that the auction implementation was consistently able to achieve lower total processing times than the implementation that was inspired by the design of Battistoni et al. in [1]. In addition, the experiments indicated that the auction implementation had similar processing times and slightly higher completion times when compared to a random approach. This section emphasizes the most crucial findings, argues for the final outcome of the experiments, and sheds light on the implications of the results.

5.1.1 Evaluating the Performances

The distinct properties of the implemented approaches caused some early expectations regarding how they would perform in the experiments. The results were mostly in line with the anticipations, but some were also surprising. The modified Battistoni implementation was expected to have the lowest completion time and the highest processing time due to its competitive behaviour. The auction implementation was expected to perform better than the random choice implementation because it chooses an optimal solution at every step in the propagation. This section provides an overview of the expected and unexpected results from the experiments.

Expected Results

The modified Battistoni implementation was expected to have a significant advantage in producing quick responses. That was confirmed on the basis that the Battistoni approach had the lowest completion times throughout the experiments. The intuitive argument for this outcome is that the Battistoni approach allows more Nodes to process a service than the other approaches. In fact, all Nodes with a higher internal bid than the asking price of the request will immediately process the service when notified. The optimal Node, the Node with the highest bid and lowest processing time, will therefore be part of the group that processes the service. That is under the assumption that the optimal Node has a bid that is above the asking price. In addition, the Battistoni approach is able to notify all available Nodes with inconsiderable delay. As a result, the modified Battistoni implementation had the lowest completion times of the bunch.

The auction and random choice implementations had lower total processing times than the modified Battistoni implementation. That was expected given that both of the approaches will in general only select a single Node to process a service. However, the auction implementation would sometimes have multiple winners in auctions. Because of this possibility, the auction approach was assumed to have longer total processing times than the random approach. It was apparent when this occurred because the auction implementation clearly had higher maximum processing times than the random choice implementation in a few sessions.

Unexpected Results

It was unexpected that the auction and random choice implementations had nearly the same performance in the experiments. The auction implementation was assumed to perform better due to the active decision making process that took place in the auctions. However, for most of the sessions, the random choice implementation performed slightly better in terms of completion times. The auction implementation had slightly lower average total processing times in about half of the sessions.

It was presumed that the auction implementation would not always be able to find the best global solution for each run. The reason is that the first winner of an auction with a bid that is higher than the asking price would process the service. Therefore, a more suitable Node residing at a deeper layer would not be found. As a result, the random approach may have found better solutions at a deeper layer in some of the runs. However, there is no consistent pattern in the way that the networks were populated that suggests that a solution at a deeper layer would be different from an earlier solution. Although the random approach may miss an earlier solution, its completion time is not significantly affected by finding a solution at a deeper layer. That is because its method of propagation does not have any inherent delays. Unlike the auction approach that has a delay in each step of the propagation which is equal to the auction period. The auction period is the total of the joining period and the active period. Given that the random approach finds a solution it will have used less time on making decisions and the solution it finds is expected to be equally as good. Depending on the exact structure of the network and the population, this will in some cases yield a better performance. This may explain why the random approach was a slight favourite in completion times. The likelihood of the random approach finding a solution is further explored in section 5.1.2.

It was assumed that the auction implementation would have a lower standard deviation in terms of processing times and completion times than the random approach. The reasoning was that the auction implementation, by taking an active choice in the auctions, would be able to find a suitable solution when the random approach would be forced to go to the end of the network and encounter an unsuitable Node. That could lead to a long processing and completion time. However, the results show that the auction approach has for some sessions a significantly larger standard deviation in the processing time. The most likely reason is that some runs had multiple auction winners which effectively increases the processing time by a factor equal to the amount of winners. The standard deviation in completion times were in most cases higher for the auction implementation as well. Each auction had an auction period of 0.2 seconds and having a varying amount of auctions in each run will yield significant variations in the delays that are caused by the auctions. The variance in the amount of auctions might explain the higher standard deviation in completion times. In contrast, the standard deviation in completion times for the random approach were not expected to be greatly affected by finding deeper solutions. But it was assumed to have a moderately high standard deviation because it may reach unsuitable Nodes at the end of the network quite often.

5.1.2 Further Analysis

One of the potential reasons for the auction implementation having not only comparable, but slightly worse performance than the random choice implementation in both total processing time and completion time is that the auction implementation did not include an intelligent bid portion. An intelligent bid portion that includes the expected bid sizes that would be present in a re-auction might have improved the choice of winners in auctions. This section covers two examples that will highlight why the experimental results of the auction implementation and random choice implementations were surprisingly similar and how the performance of the auction approach might be improved with the intelligent portion.

The Cumulative Probability Properties of Auction vs. Random Propagation

The auction implementation had an advantage over the random choice implementation at every step in the propagation because of its ability to select the Node with the highest bid. If the winner of an auction had a higher bid than the asking price then the Node can start processing the service. The random approach has a chance of selecting an incapable Node and then further propagate the request to a deeper layer. A solution, that is any capable Node, found at a deeper layer may be assumed to have equal processing times compared to a solution found at a previous layer. The random approach does not have any inherent delay in its propagation method and, unlike the auction approach, the completion time is not significantly affected by finding deeper solutions. Therefore, a random approach may perform better than auctioning in cases where it is likely to find a solution at a deeper layer that has similar qualities to a missed solution at a previous step. The effect of this is further considered by looking at an example session from the experiments. The session that is regarded is from the experiment on mean bid sizes and the mean bid was set to 30, the results from the experiment are found in section 4.6.2.

For the experiment on mean bid sizes, the bids were drawn from a truncated standard distribution with a standard deviation of 20 and Network 1 was used for all the sessions. The session that is considered had a mean bid size of 30. The three-sigma rule states that 68% of findings from a standard distribution will be found within 1 standard deviation from the mean. When the mean bid size was 30 and with a standard deviation of 20, then it is assumed that there was approximately a 16% chance that the bid size of any Node was above the asking price. Going by this approximation, there was an 84% chance that the first Node would not have a bid above the asking price. This would incite an auction or further propagation for both implementations. Network 1 is a binary tree such that every subsequent branch contains two Nodes. In each step of the propagation there was a 29% chance that at least one of the two Nodes had a bid above the asking price. The auction approach would find a solution if present, but the random choice implementation had only a 50% chance of finding it. The probability of the random choice implementation being able to find a solution at every step is equal to the probability of the selected Node being above the asking price, which is 16%. The probabilities of there existing a solution at every layer are independent. As a result, for two repeated trials, the random choice implementation had a cumulative probability of 29% for finding a solution at either the first or second propagation step according to the binomial theorem. In comparison, the auction implementation would have a cumulative probability of 50%. However, it would choose a solution at the first step if it was present.

In this example, the probability of the random approach finding a solution at the first or second step are similar to the probability of the auction implementation finding a solution in its first auction. This partially explains why the random choice approach fared slightly better in deep networks and the auction did a bit better in the wider networks. When the network topology is narrow, then there are few options for the random approach to miss and few options to choose from in the auctions. When the network topology is wider, then the possibility of choosing between multiple solutions is more likely to be present and the random approach given more options would be less accurate in finding a solutions. To exemplify this, given the same circumstances as the previous example but with 4 Nodes at every branch. The probability of there at least being one solution at the first step is 50%, the random approach would still have a 16% chance of finding it in the first step. The cumulative probability of finding a solution at step two in the propagation is 29% for the random approach and 75% for the auction approach. Maybe trivial, but its worth noting that the cumulative probabilities would be the same for both approaches if each layer only had a single Node.

Introducing the Intelligent Portion

An example of where the heuristic component would be advantageous is when the topology of the network is similar to the one showed in figure 5.1. If this case was part of the experiments then the auction implementation would have a worse processing time and completion time in comparison to the expected times of the random choice implementation. The auction implementation would go down the path N0, N1, N3 and N3 would do the processing. The random choice implementation would have an equal chance of selecting N0, N2, N4 as its path. If the processing constant k was set to 0.8 and the standard processing time T_0 was set to 2 seconds, then the auction implementation would have a processing time of 2.16 seconds and the random choice implementation would have an expected processing time of 1.6 seconds given that the processing times are given by expression 4.1. The completion time of the auction implementation would be 2.56 seconds given that the auction period is set to 0.2 seconds. The random choice implementation is assumed to have no delay in propagation and the processing and completion times are seen as equal. As a result, the expected difference in processing time is 0.56 seconds and 0.96 seconds for the completion time which accounts for the delay that is introduced by auctioning twice.

The best solution for the example in figure 5.1 is to choose the path N0, N2, N4. This path leads to N4 which has a processing time of 1.04 seconds. The random choice implementation is expected to choose this path half of the time, but the auction implementation with naive bids will never choose this path. The intelligent portion of the bid can steer the auction implementation to choose the correct path. The figure 5.2 shows an example where an intelligent portion is included that accounts for the bids of neighbouring Nodes. The composite bid of each Node is adjusted according to the naive bid portion of its neighbouring Node that would be part of a re-auction. The adjustment in this example is made according to expression 5.1. The intelligent portion is added to the naive portion to

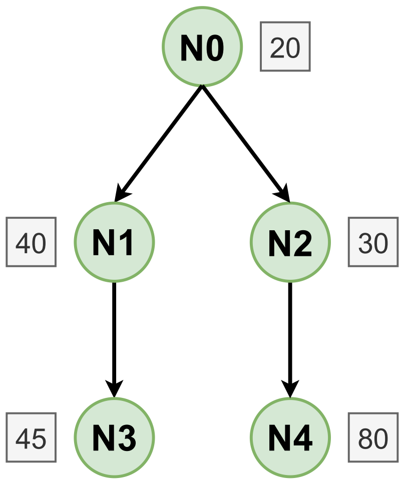

Figure 5.1: A network topology with naive bids. One of the paths are clearly better than the other, but the second layer Nodes have misleading bid sizes.

make up the composite bid marked in blue boxes for the path on the left and in red on the right. Note that the origin Node N0 has two adjusted bids, but that the method for joining them is inconsequential in this case. The result of including information on the bids of neighbouring Nodes by adjusting the bids accordingly is that the auction implementation will choose the correct path in this case.

$$\text{Intelligent Portion} = \frac{\text{Naive Bid} - 50}{2} \tag{5.1}$$

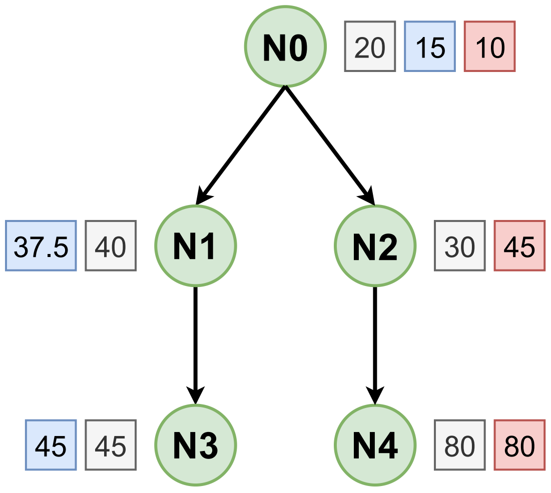

Figure 5.2: A network topology with intelligent bids with heuristics on neighbouring bids. The bids are adjusted such that the best path is chosen through auctions.

	Auction w/ Naive bids	Random Choice	Auction w/ Intelligent bids
Processing Time (s)	2.16	1.6	1.04
Completion Time (s)	2.46	1.6	1.44

Table 5.1: An overview of the performance of the auction implementation in comparison to the random choice implementation when bids are completely naive and when including an intelligent portion. The data is drawn from an example case.

This exemplifies the potential improvement that could be gained by adding an intelligent portion to the auction implementation. In this example, the expected performance of the random choice implementation is better when competing against an auction implementation that only uses naive bids. However, the auction implementation performs better with the addition of intelligent bids. The improvement in performance for this example is summarised in table 5.1. A further discussion on the implications of including an intelligent portion in implementations of the auction approach are further discussed in section 5.3.3.

5.1.3 Implications

The data that was gathered in the experiments indicates that a task assignment system that is based on an auction approach may be a viable strategy for a fog computing architecture. The auction approach has been shown to statistically perform better against the random approach on wide and shallow networks. However, the auction approach has an inherent delay for each step in the propagation. It is therefore more suited for networks where a good solution is likely to exist in the layers that are in close vicinity to the origin Node. The auction approach with naive bids does not consider the best path to an optimal solution. However, it can be adjusted to do so by including an intelligent portion that accounts for the capabilities of neighbouring Nodes. The modified Battistoni implementation has a trade-off that prefers quick completion times over the efficient use of resources. It was clearly the fastest approach but in one of the sessions it had a total processing time that was more than 10 times longer when compared to the auction and random approach. In a real application, this translates to higher electricity costs and opportunity cost in terms of what other tasks that could have been performed instead.

5.2 Advantages of Using MQTT in the Fog

The MQTT protocol is commonly used for IoT applications because of its lightweight messaging and publish-subscribe messaging pattern. The properties that make MQTT successful in distributed sensor networks are also beneficial for routing requests in a fog computing architecture. The main reasons are the decoupling features of topic-based publish-subscribe style messaging, reliable message delivery, and lightweight messaging. A fog network consists of a moderate amount of physically distributed Nodes and it likely has some dynamic properties because of Nodes entering or leaving, unreliable connections, and the eventual restructuring of connections. The MQTT messaging protocol has purpose-built features that make it suitable for distributed and dynamic networks.

5.2.1 Lightweight Messaging

The response times and bandwidth consumption for MQTT compared to HTTP are significantly lower as reviewed by C. Wang for Google in [23]. The results were found by measuring the response time and packet size when sending multiple messages to a remote server and was tested with QoS 1 over persistent and reused MQTT sessions. The MQTT protocol has small packet headers in comparison to other commonly used application layer protocols such as HTTP. This makes MQTT messages lightweight and the protocol generally uses little bandwidth. This is one of the main reasons that MQTT is popular among constrained devices because MQTT clients can publish messages with relatively small packet sizes. The payload is often small and the short header allows the payload to be the primary factor that determines the packet size. This is beneficial when the devices are connected to rudimentary networks with low bandwidth.

A fog computing architecture will also benefit from lightweight messaging. The MQTT protocol will save on bandwidth when it is used in both Node-Node and Node-Client relationships. However, lightweight messaging is likely to primarily benefit the Clients. They might be constrained devices that are located on the edge. If this is the case, then they will in most cases have limited networking capabilities to save on device and network costs. The Clients should for that reason be able to communicate with the Fog over a lightweight protocol such as MQTT. The Nodes are assumed to have moderately powerful hardware that can perform computations and utilize a large bandwidth network. However, in certain applications, e.g., where the Nodes are either mobile or placed in desolate areas, the network can be a limiting factor for the Nodes as well. A network connection in a remote area may have costs associated with the transmission of messages. For example, an LTE connection with a set amount of monthly data. In such case, the Nodes will also benefit from using the network capacity sparingly and gain the advantages of lightweight messaging. Therefore, the MQTT protocol can provide benefits with its lightweight messaging at various levels in the Fog. It may only be used for communications between Nodes and Clients but it can also be used for communication between Nodes. Lightweight messaging will aid in fulfilling any constraining criteria that are set on network capabilities and the associated costs of messaging.

5.2.2 Reliable Message Delivery

Important messages such as the request of a service might be lost when transmitting it over an unreliable network. To counteract this, the MQTT protocol features a range of built-in message delivery guarantees which are referred to as QoS. The levels of QoS and their use-cases were covered in 2.2.2. In fog computing applications, the Nodes and Clients may be subjected to unreliable network conditions due to the signal type that is used, some hardware issue, unfortunate accidents, etc. The benefit of applying a high level of QoS in the Fog is that the Nodes and Clients can be sure that their requests will not be lost as a result of any unstable connections.

5.2.3 Decoupling

MQTT clients are naturally decoupled because MQTT is a topic-based publish-subscribe messaging protocol. The message traffic passes through a broker such that even the clients that interact with each other are not directly connected. The decoupling nature of MQTT was covered in section 2.2.1. The main advantage of having decoupled clients is that the functionality of client does not rely on any other specific client. The implication is that when a client is removed from the system it may be replaced by another client that performs the same duties. The replacement client may have vastly different properties in terms of hardware components, service implementations, or network capabilities. However, as long as the interfaces and functionalities that it provides match those of the original client, then other clients that relied on interactions with the original client will be unaffected by its replacement. The term unaffected has been loosely used here, because the performance of the replacement can be different. However, it is assumed that the replacement client behaves the same as the original client and that the application logic is decoupled as well.

During the life cycle of a Fog, some Nodes have to be replaced, added, or removed for various reasons. If the Nodes were not decoupled, it would be difficult to keep the affected part of the Fog operating as intended after either of these incidents. When a Node is added to the Fog it connects to the broker hosted on an existing Node. The new Node may also connect to multiple existing Nodes. Any existing Nodes may connect to the new Node as well. After the new Node has connected to any number of existing Nodes it can support the Fog through its interactions with the Nodes which it is connected to. The Nodes that it connected to will not change their behaviour to accommodate for the newly added Node because of the decoupled nature of MQTT. The subscriptions and any topics that the existing Nodes publish to will remain unchanged. When a Node is removed from the Fog then the behaviour of the Nodes that it was connected to do not change. However, any Nodes that were connected to Node that was removed may lose their relative connections to other Nodes. This occurs when the removed Node served as an intermediate connection to any other Nodes. In the worst case, some Nodes can be separated from the Fog if an intermediate Node leaves the network.

Therefore, when a Node is permanently removed from the Fog there should exist some mechanism for restructuring the network. Assuming that such a mechanism is implemented, then the decoupled properties of MQTT allow the Nodes to not change any other parts of their behaviour except for their connections. The same case applies when a Node needs replacement. The replacement Node will likely have a different address than the original Node. To replace a Node, the connections to address of the original Node are dropped and equal connections are made with the replacement Node on the new address.

5.3 Evaluation of Auction-based Propagation

The argument for using an auction approach for assigning tasks is that the method inherently finds the best local solution at every step in the chain of propagation. This will ensure that the processing of the request will be performed by a single or a few suitable Nodes. The approach uses a decentralised design for request propagation and does not require a supervisor that has knowledge of the entire state of the Fog. However, the approach has drawbacks such as inherent latency, traceability issues, and may choose unfavourable paths when presented with bids that are based on incomplete information. This section serves as an evaluation of the auction approach. It starts of with the advantages of propagating requests with auctions. Then the disadvantageous aspects of the approach are covered.

5.3.1 Advantages

Any task assignment method that is used for propagating requests in the Fog must be able to find good solutions in an efficient manner. The auction approach is able do achieve this without relying on centralised services. That is beneficial for the many applications that have requirements such that it is either impractical or impossible to depend on centralised services. The auction approach is decoupled and provides the necessary flexibility to either assign tasks in a contained local network or be expanded by connecting additional Nodes to the Fog which enables the approach to provide broader coverage. In cases where it is applicable, tasks may be transferred from the Fog to the Cloud in a standardised manner.

Optimal Local Solutions

The auction approach is a suitable method for request propagation in a Fog where the Nodes are likely to have varying capabilities. The task assignment system used in a Fog should be able to arrive at an optimal Node to process any given request, or at least a Node that is close to optimal. Other task assignment systems may achieve this through periodic updates on the capabilities of neighbouring Nodes, rely on centralised supervision and control, or introduce some redundant delegations in an effort to include the optimal Nodes. The auction approach finds the current capabilities of neighbouring local Nodes in a decentralised fashion by holding auctions at every step in the propagation. This ensures that the auction approach makes its decision based on recent information and that requests are delegated to the optimal Node among the auction participants. As discussed in section 3.3.4, the approach may also be expanded to include heuristics on the capabilities of Nodes that reside at deeper layers by introducing intelligent bids. This would enable the auction approach to make better long term decisions and is further elaborated on in section 5.3.3.

Decentralised Decision making

Decentralised task assignment schemes such as the auction approach do not rely on a global up-to-date state of the system, which may be hard to maintain in a large and dynamic Fog. A Node that operates according to a centrally governed assignment strategy would be required to have the information about the optimal Node to propagate their request to. To acquire the current optimal solution, the Node must constantly receive over-the-air updates or poll the central system whenever it handles a request. In contrast, a decentralised propagation system leverages local information that is either immediately available to the Node or through updates from neighbouring Nodes. As a result, the Nodes in a decentralised system do not need to maintain a robust connection to a centralised service. This is an advantage because the Nodes can operate in unreliable networks in remote areas and are only required to have an immediate connection to their neighbouring Nodes.

Decoupled Components

The presented auction approach is designed such that the Client is only directly connected to the broker on its origin Node. The origin Node acts as the Client's interface to the rest of the Fog and any requests from the Client are sent to its origin Node. The advantage of this structure is that the Client does not need to make any new connections after it has established its connection to the origin Node. As a result of this, the topology of the Fog may change but the Client would not notice this or need to change its behaviour as long as it remains connected to the origin Node. This is because the Client is entirely decoupled from the rest of the Nodes in the Fog. The Nodes are only partly decoupled because they need to connect to each others brokers to construct the Fog. This is common for all the approaches presented in this thesis. However, the auction approach is also decoupled at the application logic because the Nodes do not need to update how they manage their auctions to accommodate for any changes to the number of participating Nodes. However, a Node or a complete branch of Nodes may be separated from the rest of the Fog if an intermediate Node is terminated. This was discussed in section 5.2.3 which covered the decoupling properties of MQTT.

Cloud Participation

In some cases a request would most effectively be handled by the Cloud. This will occur whenever a request can not be handled by the Nodes or the Cloud is the better alternative performance wise. The Nodes may be unable to carry out the request because it has premium hardware or network requirements or when the Nodes are already overloaded with concurrent requests. The auction approach allows the Cloud to participate in the Fog as any other Node. Incorporating the Cloud directly into the Fog is advantageous because the Cloud has superior computational capabilities compared to the Nodes and it is better suited at reliably handling long term services. The Cloud can join the Fog at several locations such that it can aid in computations wherever it is needed. The considerable network latency to the Cloud can be accounted for in the bids that the Cloud place in auctions.

5.3.2 Disadvantages

The auction approach requires the cooperation between its participating Nodes to successfully propagate requests. The requests are propagated through a series of auctions. Each auction causes a delay into each step of the propagation and therefore a challenge of the auction approach is to perform well despite the delay. Using a method that relies on propagation, instead of establishing a direct connection between the Client and the processing Node, makes it hard to track the path of a request. The Nodes must also be trusted to provide valid bid sizes that accurately represent their capabilities. The auction approach is unable to locate optimal solutions at deeper layers if the intelligent portion is not included in the bid.

Auctions Introduce Latency

The strategy of holding auctions introduces a source of latency into each step of propagation. This is because any style of auction will have to be active for an extended period. An auction consists of the declaration of the item to be auctioned and the auction round itself. The properties of an item must be communicated to the bidders and the participants of an auction need time to place their bids. The inherent delay of holding auctions may be compensated for if the strategy is able to find considerably better matches than faster task assignment methods. The challenge is to find an optimal trade-off between consistently being able to find the optimal candidates and to do so in a timely manner when compared to other task assignment approaches. The experiments showed that the auction implementation was not able to outperform a random approach under the given circumstances. The results from the experiments show that it is uncertain if the increased overhead of holding auctions is worthwhile in practice.

Difficult to Trace

The path that a request has taken from the origin Node to the processing Node may be difficult to follow for task assignment methods that rely on propagation such as the auction approach. This is because each Node along the path of a request is only aware of who the request came from and who the request was sent to. If each Node kept a full log of all the requests that they have handled then the complete path of a given request may be found by joining logs. Starting from the origin Node, the path of a request can be trailed by iterating through each intermediate Node until the processing Node is reached. This method for identifying the path of a request is impractical in a fog computing context because each Node is a separate computer and the request logs of each Node is unlikely to be readily available. However, it is highly beneficial to be able to trace the path of a request to gain performance metrics, monitor the network, and for debugging purposes.

Inaccurate Bids

An important consideration of the auction approach is to what extent the Nodes can be trusted to place accurate bids that honestly represent their capabilities. The Nodes calculate their bids individually and could provide dishonest or inaccurate information about their capabilities by altering their bid sizes either intentionally or unintentionally. A Node that bids too large will in certain situations outbid other Nodes due to its inflated bid. As a consequence of this, a Node that bids too high based on its capabilities will win more auctions and handle more requests than it should. In the opposite direction, a Node can also bid too low in auctions. A Node that bids too low will receive fewer than optimal requests to handle. The result of Nodes having inaccurate, and potentially dishonest, bids is that less qualified Nodes will win more auctions.

Requires Heuristics to Choose the Best Paths

An auction approach that uses naive bids will sometimes choose a path that leads to a suboptimal solution. The naive portion of a bid equates to a Node's static capability to process a requested service under ideal conditions. Dynamic factors such as the current load and heuristic factors like the capabilities of its neighbouring Nodes are only accounted for in the intelligent portion of a bid. Regardless of how bids are composed, the highest bidder in an auction is always declared as the winner of the auction. If the winner of an auction is unable to process the requested service then the request is re-auctioned. However, if the winning bid was purely naive then it did not take into account the capabilities of the Nodes that will be participating in the re-auction. This is also true for dynamic bids that include the intelligent portion, but that do not incorporate this heuristic factor. Therefore, there is a significant chance that choosing a different Node in the original auction would have been better. Because another participant may have better neighbours than the winner despite having a lower bid in the auction. As such, the auction approach can make suboptimal choices when the bids are naive and all the participants of an auction bid below the asking price of the item.

5.3.3 The Importance of the Intelligent Portion

The implementation of the auctioning approach did not include an intelligent portion in the bids. It was not included because the other implementations would not benefit from it, the complexity of the implementation would increase, and the auction approach is able to operate without it. However, including an intelligent portion with heuristics on the capabilities of neighbouring Nodes might have improved the path of propagation. This might have increased the performance of the auction implementation in the experiments. Possibly to the point where it would have better performance than the random choice implementation.

The naive and the intelligent portions of a bid conceptually represent two different reasons for a Node's suitability to win in an auction. The naive portion of the bid is a static representation of the Nodes capability to perform the requested service if it was forced to do so under its ideal conditions. Any dynamic factors that might affect the performance of the Node are not accounted for. Therefore, the auction approach will only find the Node that is the most suitable for instantly processing a service when the bids are completely naive. This may be acceptable in situations where the most optimal solution is found in the first auction. However, the most capable Node in the first auction is likely to win a lot of auctions because it has a high static bid. This may over time reduce its performance due to the load itself and additional dynamic reasons are also not communicated though the naive bid. Additionally, any solutions that would have been found in the second, third, or later auctions are not considered. If the Nodes in the first auction were only barely capable then the most optimal solution might have been found in the subsequent auction. The choice of Node at that point also matters because the path to the optimal Node in any potential second auction is not affected by the bids in the first auction. Therefore, choosing the highest bidder in the first auction gives no guarantees to finding the optimal or even a well qualified Node in a re-auction. The method takes the best choice given the circumstances, but may make mistakes that end up being costly at a later stage.

The intelligent portion of the bid aims to adjust the bid according to the dynamic factors that directly affect the capabilities of a Node and give a heuristic measure to what capabilities that could be expected from its neighbours if an item were to re-auctioned. The benefit that is enabled by having an expectation or heuristic on the bid size of neighbouring Nodes is vital to explore the most efficient paths to a processing Node. Without this heuristic it would not be, disregarding network latency, meaningful to be delegating an item to the Node that bids the highest in an auction if the winning bid is still below the asking price. Because it will be equally likely that a loosing Node will be connected to a better Node than the winning one.

A composite bid of the naive and the intelligent portion of a bid will give a more honest bid from the Nodes. Because it represents the Node's current capability of handling the request in general. However, the size and components of the naive bid and the intelligent bid has not been studied widely in this thesis. Although the implementation may have uncovered the importance of it. It is unlikely that throughout the experiments, having an active decision regarding the most optimal Node did not yield any results. That makes it likely that the auction approach is unable to make the correct decisions, in particular when it comes to choosing good paths.

An alternative to including an intelligent portion is to allow all Nodes to re-auction an item given that no Node was able to bid above the asking price. This would allow all participating Nodes to search for a suitable candidate among their neighbours. A strategy as this would increase the processing time since the item is sent along multiple paths, however it may produce a faster completion time as well.

Chapter 6

Conclusion

This thesis has presented a decentralised task assignment system that uses an auction mechanism and the MQTT protocol to propagate requests. The intended use for the auction approach is to be applied in fog computing architectures that require interaction between multiple MQTT brokers. The show the feasibility of implementing the auction approach a demonstration program was made alongside implementations of two other decentralised task assignment methods. The program was developed in Python and the fog nodes are represented by genuine MQTT clients, the source code is available in [22]. A series of experiments were performed to evaluate the performance of an implementation of the auction approach with naive bids.

The demonstration program was tested against a random approach and an implementation based on the works of Battistoni et al. in [1]. The total processing times and completion times when responding to requests were measured under a variety of configurations. The results of the experiments showed that an auction approach with entirely naive bids was on par with the random approach in terms of completion times and processing times. It outperformed the modified Battistoni implementation with regards to processing times, but it had longer completion times. The demonstration program of the auction approach was unable to find the most favourable propagation paths beyond the current step and as a result there were occasions where the random approach would have a lower processing times. However, the performance of the auction approach was shown to be improved by including additional heuristics on the subsequent propagation stages for some network configurations. Therefore, further studies of the presented auction approach should consider employing a heuristic component in the form of an intelligent bid portion as described in section 3.3.4.

The auction approach has been shown to be viable as a decentralised task assignment method and does not require centralised control mechanisms in order to selectively propagate messages in a multi-broker architecture. The performance of a demonstration program that uses the auction approach with naive bids has been measured and empirically compared to similar task assignment methods. The auction approach offers flexible parameters in the asking price and bids that can be configured to adapt the behaviour of the system. The auction approach also accommodates the use of heuristics, cloud participation, and request-response services. As a result, the auction approach is an expandable method that based on publish-subscribe patterns is able to find suitable fog nodes for off-loading edge clients in response to remote service requests.

Bibliography

[1] P. Battistoni, M. Sebillo and G. Vitiello, 'Computation offloading with mqtt protocol on a fog-mist computing framework,' *Internet and Distributed Computing Systems*, vol. 12, pp. 140–147, 2019.

[2] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong and J. P. Jue, 'All one needs to know about fog computing and related edge computing paradigms: A complete survey,' *Journal of Systems Architecture*, vol. 98, pp. 289–330, 2019.

[3] O. F. Consortium, *Openfog reference architecture for fog computing*, Feb. 2017. [Online]. Available: https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf.

[4] H. Gupta, S. Nath, S. Chakraborty and S. Ghosh, 'Sdfog: A software defined computing architecture for qos aware service orchestration over edge devices,' Sep. 2016.

[5] T. Jung-Fa, H. Chun-Hua and L. Ming-Hua, 'Optimal task assignment strategy in cloud-fog computing environment,' *Applied Sciences*, vol. 11(4), p. 1909, 2021.

[6] L. Brunet, H.-L. Choi and J. How, 'Consensus-based auction approaches for decentralized task assignment,' *AIAA Guidance, Navigation and Control Conference and Exhibit*, Aug. 2008. DOI: 10.2514/6.2008-6839.

[7] M. Le, S. Clyde and Y.-W. Kwon, 'Enabling multi-hop remote method invocation in device-to-device networks,' *Human-centric Computing and Information Sciences*, vol. 9, no. 20, Jun. 2019. DOI: 10.1186/s13673-019-0182-9.

[8] B. Cheng and T. Sanada, 'Fog function: Serverless fog computing for data intensive iot services,' in *2019 IEEE International Conference on Services Computing (SCC)*, 2019, pp. 28–35. DOI: 10.1109/SCC.2019.00018.

[9] Golem, *Golem.network wiki*, 2021. [Online]. Available: https://docs.golem.network/#/.

[10] S. Hoque, M. Brito and A. Willner, 'Towards container orchestration in fog computing infrastructures,' Jul. 2017. DOI: 10.1109/COMPSAC.2017.248.

[11] SONM, *About sonm*, 2021. [Online]. Available: https://docs.sonm.com/.

[12] A. S.-C. IBM, *Mqtt is the leading protocol for connecting iot devices*, Jan. 2019. [Online]. Available: https://developer.ibm.com/blogs/open-source-ibm-mqtt-the-messaging-protocol-for-iot/.

[13] A. Banks, E. Briggs, K. Borgendale and R. Gupta, *Mqtt version 5.0*, Mar. 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

[14] R. Ranøyen Homb, *Mqtt – what is it? and how can you use it?* Jul. 2017. [Online]. Available: https://www.norwegiancreations.com/2017/07/mqtt-what-is-it-and-how-can-you-use-it/.

[15] Z. Zhou, *Request response - mqtt 5.0 new features*, Sep. 2020. [Online]. Available: https://www.emqx.com/en/blog/mqtt5-request-response.

[16] A. Banks and R. Gupta, *Mqtt version 3.1.1*, Dec. 2015. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

[17] P. Benedetti, M. Femminella, G. Reali and K. Steenhaut, 'Experimental analysis of the application of serverless computing to iot platforms,' *Sensors(Basel)*, vol. 21, no. 3, Jan. 2021. DOI: 10.3390/s21030928.

[18] Z. Laaroussi, R. Morabito and T. Taleb, 'Service provisioning in vehicular networks through edge and cloud: An empirical analysis,' Oct. 2018. DOI: 10.1109/CSCN.2018.8581855.

[19] S. Parsons, J. Rodríguez-Aguilar and M. Klein, 'Auctions and bidding: A guide for computer scientists,' *ACM Comput. Surv.*, vol. 43, p. 10, Jan. 2011. DOI: 10.2139/ssrn.3794860.

[20] E. foundation, *Paho python mqtt client*, Jan. 2022. [Online]. Available: https://www.eclipse.org/paho/index.php?page=clients/python/index.php.

[21] E. foundation, *Eclipse mosquitto*, Jan. 2022. [Online]. Available: https://mosquitto.org/.

[22] H. S. Midtun, *Auction Fog*, version 1.0.0, Jan. 2022. [Online]. Available: https://github.com/HenrikMidtun/AuctionFog.

[23] C. Wang, *Http vs. mqtt: A tale of two iot protocols*, Nov. 2018. [Online]. Available: https://cloud.google.com/blog/products/iot-devices/http-vs-mqtt-a-tale-of-two-iot-protocols.

Appendix A

Additional Material

The Python code below is an excerpt from the main program file. It has been included to provide some insight into how the results of the experiments can be tested at a later point and how to modify the program to test other scenarios. The full source code for the program is available through GitHub in [22]. The main file is located at AuctionFog/src/main.py. It contains code to run a single session. Lines 30-38 creates a dictionary which represents the network topology. Line 39 initialises the request monitor that measures the total processing time and completion time of each request in the session. Line 40 initialises the node controller that has methods to create Nodes and modify them. The network controller is initialised in the following line 41 and is in charge of creating each network, populating the network, and making changes to all the Nodes. The session is started by calling the run_sim function in line 42.

The run_sim function initialises the networks and runs the session, the output is then written to a file whose name is given as a parameter. The function definition starts at line 1. Lines 5-8 populates multiple networks according to the structure given and the approach types that are requested. The origin Node of each network is then added to the appropriate key in the origin_nodes dictionary. In this case, each network is structured as the default Network 1, 5 networks for each approach is created, and all approaches are included. Lines 12-20 makes requests to the origin Nodes and updates their bids and services for each run. The Client is initialised in line 13. The Client makes a request to each of the origin Nodes that were created. The id of the request is inserted into the dictionary named requests at the appropriate key. The bids and services are updated for all the networks in line 20. After doing so, another batch of runs may begin with the updated state. After completing the session, the results are written to a .csv file. The session writer that is initialised in line 25 queries the request monitor for each request id that is present in the request dictionary. The results are returned and written as a line into the file.

The code in the main program file may be changed to simulate a different network structure. To do so, a dictionary like net1 at line 30 is made. Each key is the id of a Node and the values are its neighbours, leaf Nodes do not have a key associated with them. The ids given in the dictionary are not the actual ids that are given to the Nodes, and may be any valid string or number. The amount of simulations that are run simultaneously can be adjusted at line 6. Note that when simulating all approaches, the amount of networks will be the number in the range multiplied by 3. Each Node has its own thread to keep track of the processing, so this number should be kept low. However, the simulation can be performed faster with a larger number of simulated networks. The amount of subsequent simulations is configured in line 14. In this example there are 15 networks and 3 batched iterations, therefore there will be a total of 45 runs. Any parameters such as the mean bid size, standard deviation of bid size, asking price, auction period, etc. can be set in the configuration file found in AuctionFog/src/Configuration.py

```
def run_sim(network_controller, request_monitor, filename, structure, n_type):

    #Create Nodes and make connections according to structure

    origin_nodes = {"auction":[], "choice":[], "battistoni":[]}
    for k in range(5):
        for type, node_obj in network_controller.create_network(structure=structure, n_type=n_type).items():
            origin_nodes[type].append(node_obj)

    #Make Requests to origin Nodes and update all Node bids in batches.

    requests = {"auction": [], "choice": [], "battistoni": []}
    client = Client(request_monitor=request_monitor)
    for i in range(3): #Amount of batched iterations
        for type, o_nodes in origin_nodes.items():
            for node in o_nodes:
                req_id = client.make_request(origin_node_id=node.client_id, service="A")
                requests[type].append(req_id)
        sleep(6) #Allow Nodes to finish up processes
        network_controller.updateNetworkServices(service_probabilities={"A":100})

    #Write requests to file

    sleep(10)
    writer = SessionWriter(request_monitor=request_monitor)
    writer.set_headers(filename=filename)
    for n_type, req_ids in requests.items():
        for req in req_ids:
            writer.write_request_results(filename=filename, request_id=req, n_type=n_type)
net1 = {
    0: [1,2],
    1: [3,4],
    2: [5,6],
    3: [7,8],
    4: [9,10],
    5: [11,12],
    6: [13,14]
    }
request_monitor = RequestMonitor()
node_controller = NodeController(request_monitor=request_monitor)
network_controller = NetworkController(node_controller)
run_sim(network_controller,request_monitor, structure=net1, n_type="all", filename="default")
```

Figure A.1: Excerpt from the main program file.

Henrik Sang Midtun

Auction-Based Task Assignment in Fog Computing Architectures with Publish-Subscribe Messaging Patterns