

BEP TI3806

Creating an online auctioning clock

Authors

W.J. Baartman

C. Lemaire

M.J. van Tartwijk

P.W.P. van der Stel

J.C. de Vries

BEP T13806

Creating an online auctioning clock

To obtain the degree of Bachelor of Science at the Delft University of Technology, to be presented and defended publicly on Tuesday July 2, 2019 at 11:00 AM.

Authors: W.J. Baartman
C. Lemaire
M.J. van Tartwijk
P.W.P. van der Stel
J.C. de Vries

Project duration: April 23, 2019 – July 5, 2019

Guiding committee: Robin van den Broek Developer @ JEM-id, Client
Jill van der Knaap Marketing/Communication @ JEM-id, Client
Dr. Annibale Pannichella TU Delft, Coach
Dr. Huijuan Wang Bachelor Project Coordinator

Preface

This report announces the end of studying at the Delft University of Technology for the Bachelor's degree in Computer Science and Engineering. Although the end of this stage in growing up to be a computer scientist or engineer is near, most of us will continue our time at TU Delft with great pleasure during a follow-up master of Computer Science. In this report, we will discuss the product we created during the last ten weeks and we will touch upon the methods and knowledge we used to produce it.

Before we lay out the technical details of our final product, we would like to thank those who made our path to it possible. First of all, we would like to thank JEM-id for their hospitality and enthusiasm. After arranging to do a project we like for them, JEM-id offered us a beautiful workplace in their office. We would like to thank Robin van den Broek and Jill van der Knaap in particular for helping us with external and internal communications, and for providing us with the data required. We would like to thank Annibale Panichella from the faculty of EEMCS at TU Delft for his guidance and for sharing his insights.

Finally, we dedicate this report to Johanna Sijje Goemaat who came to pass so unexpectedly in the morning of the 12th of June 2019. We give Wesley our best wishes with recovering from this terrible loss.

Delft, July 2019

Summary

For this project, JEM-id tasked us with creating a proof of concept of a new online auction in the form of a web application for Royal FloraHolland. JEM-id is a software company operating in the agricultural domain. The pre-existing digital auction is not available as a web application and has generated technical debt over the past twenty years of its existence.

The main challenge of the project was to make sure the application is capable of sufficiently handling the current load of the auction while maintaining similar performance. This translates to a stable connection with a ping of fewer than 30 milliseconds for clients within the Netherlands. On top of that, the system had to be scalable to support higher numbers of buyers in the future. We used a microservice architecture able to balance the load over several servers to resolve this. We spread the load of communicating with clients to services separate from the main application service. This allowed the main application service to solely and adequately keep track of the state of the clock and determine the winner of a session.

To validate that we indeed achieved the main goals of the project, we created a simulation that would simulate any number of clients connecting to the clock auction and placing bids. In this process, we generated buyer and auctioneer behaviour by analysing transaction data. We extracted several distributions from the data and sampled from it to make it more realistic. In the end, we ran this simulation ten times for chunks of an auction with 610 connected clients. A few peaks showed up where pings from client to server were significantly higher than usual. However, in the long run, the system showed low standard deviations in ping, meaning the general consistency was high. Overall, the results we gathered showed that our application was able to deal with 610 connected clients.

In the end, we consider our project to be a success. First of all, we showed that a clock application in the browser can be implemented with seven weeks of development time. Secondly, we showed that such an application could handle a realistic amount of traffic without much trouble, given sufficient computing resources. These two accomplishments show that replacing the current clock application with a web-based application is feasible.

Contents

1	Introduction	1
2	Problem definition	2
3	Problem analysis	3
3.1	Buyers	3
3.2	Auctioneer	4
3.3	JEM-id.	4
3.3.1	The UI.	5
3.3.2	Back-end features	6
3.3.3	Future expansions	6
3.4	Floriday	6
3.5	Conflicting requirements	6
4	Design	8
4.1	Requirements.	8
4.1.1	Functional requirements	8
4.1.2	Nonfunctional requirements	9
4.2	Design goals	10
4.3	Architecture.	11
4.3.1	Components	11
4.3.2	Control	12
5	Implementation	13
5.1	Flow	13
5.1.1	Initialisation	13
5.1.2	Auction process.	14
5.2	Winner selection	15
5.3	SignalR communication	16
5.3.1	Incoming messages	16
5.3.2	Outgoing messages	18
5.4	Security	19
5.4.1	Authentication and authorization.	20
5.4.2	Privilege enforcement	20
6	Simulation	21
6.1	Setup	21
6.1.1	Weaknesses and limitations	22
6.1.2	Hardware and configurations	23
6.2	Distributions.	23
6.2.1	Distribution of the bidding price	24
6.2.2	Distribution of the bought quantity	24
6.2.3	Distribution of the increase price.	28
6.2.4	Distribution of simultaneous buyers	29
6.3	Scheduler	29
6.4	Results	33
7	Evaluation	37
7.1	Process	37
7.1.1	Workflow	37
7.1.2	Client interaction	37

7.2	Product	37
7.2.1	Features and requirements	38
7.2.2	Client satisfaction	38
7.2.3	Code quality	39
7.3	Performance	39
8	Ethics	41
8.1	Jobs	41
8.2	Financial aspects	41
8.3	Data	42
9	Discussion and recommendations	43
9.1	The clock system	43
9.2	Simulation	43
10	Conclusion	45
A	Original problem statement	47
B	Product Plan	49
B.1	Introduction	49
B.2	Purpose	50
B.2.1	Research phase goals	50
B.2.2	Product goals	50
B.2.3	Functional requirements	50
B.2.4	Nonfunctional requirements	51
B.2.5	Deliverables	51
B.3	Process	52
B.3.1	Communication	52
B.3.2	Workflow	52
B.4	Risks	52
B.4.1	Team problems	53
B.4.2	Expected design is infeasible	53
B.4.3	Inadequate research	53
B.5	Planning	53
C	Research Report	55
C.1	Introduction	55
C.2	Problem analysis	55
C.2.1	Current auction	55
C.2.2	JEM-id	57
C.2.3	Floriday	59
C.2.4	Conflicting requirements	59
C.3	Technologies	60
C.3.1	Tooling and deployment	60
C.3.2	Clock API	60
C.3.3	Realtime communication service	62
C.3.4	Graphical user interface	62
C.3.5	Simulations	64
D	Simulation distributions data	65
D.1	Dataset 1	65
D.1.1	Filtered data	65
D.1.2	Columns	65
D.2	Dataset 2	65
D.2.1	Filtered data	65
D.2.2	Columns	66
E	Simulation result data	67
F	Ping Heatmaps	68

CONTENTS

v

G Feedback SIG

74

H Info sheet

76

1

Introduction

The Netherlands is the largest exporter of decorative flowers in the world [1]. This billion euro industry is powered by relations between transporters and growers. Traditionally, transactions between growers and transporters or exporters are set up by a third party through an auction. We will be working to improve the *Dutch auction* set up by Royal FloraHolland, the largest flower auction worldwide [2]. In a *Dutch auction*, the auctioneer sets a starting price for the product. It is then gradually lowered until a buyer accepts the price or the price falls below some threshold. A clock-like interface, a vestige of the analogue and mechanical clocks used in the past, visualises the current price.

Our client is JEM-id, a company that develops software for the agri- and floriculture sector [3]. JEM-id has partnered with Royal FloraHolland to develop the software platform Floriday, which offers all kinds of services to growers and transporters [4].

Currently, it is possible to place remote bids on auctions using KOA (Kopen Op Afstand — “buying remotely”) [5]. However, this system requires the installation of special software. Furthermore, the code-base is old, and it is difficult to add new features. This project aims to create an online web-based version of the clock that can be integrated into Floriday. The system we created is scalable and supports multiple configurable ways of dealing with fairness in the system.

First, we will define the problem at hand in [chapter 2](#). In [chapter 3](#), the problem is analysed further. [Chapter 4](#) describes the design of the proposed solution. Next, we describe the realized implementation in [chapter 5](#). [Chapter 6](#) goes into detail on the simulation we have performed to collect insights on the performance of the proposed solution. In [chapter 7](#), we will provide an evaluation of the different aspects of the project. After that, we will discuss the ethical implications of the system in [chapter 8](#). We will finish this report by discussing the results and making recommendations in [chapter 9](#) and making some concluding remarks in [chapter 10](#).

2

Problem definition

In the agricultural sector, goods need to be sold and transported quickly. If not, the produce will expire before it reaches its final destination. The auction is one solution to this problem: A grower registers their produce at an auction house, has it transported to the auction overnight and sells it through the auction the next morning. The auction can be seen as a meeting place for growers and buyers.

The process of a Dutch auction is the same for each product that is auctioned off. Each auction takes place with a big clock as its centrepiece. The clock counts down for each product and stops when a buyer presses a button to buy the current product. The first buyer to press the button wins the auction round and purchases the current product for the price the clock indicates at that time. If there is stock left, the auctioneer increases the price on the clock back to an acceptable starting price and the process repeats. If none of the buyers presses the button before reaching a minimum price set by the grower, the auction of the product closes too.

At the moment, the auction still uses many dated methods. Until twenty years ago, the auction was entirely analogue. Because of this, clients had to travel to the auction house itself, to be able to participate. Now, Royal FloraHolland's precursor offers KOA (Kopen op Afstand — "buying remotely") to make their clock auctions accessible to anyone [6]. Even buyers who do not live nearby the auction can now access it. However, this system requires the installation of special software and the set-up of special VPN connections. Furthermore, the code-base is old, and according to Royal FloraHolland, it is difficult to add new features or alter existing ones. New features tend to break existing functionalities, which cannot be caught easily since there are no tests.

This project aims to develop a minimum viable product of an online web-based version of the clock that can be integrated into Floriday (see also [section 3.4](#)). The system should be maintainable, highly scalable and as fair as possible: it should (be able to) alleviate the disadvantages that stem from remote localisation of buyers compared to on-site buyers (see also [section 3.1](#)). It should also provide an auctioneer with functionalities so that they can efficiently do their work. The system is meant to be a feasibility study to show whether it is possible to build a replacement for the existing system in a web browser.

3

Problem analysis

In this chapter, we will analyse different aspects of the problem as proposed in [chapter 2](#). To do so, we will describe the problem in further detail, and we will provide more in-depth insights into how the auction works. Furthermore, we will discuss the roles of buyers and auctioneers first. After that, we will describe the role of JEM-id in the project. Next, we will talk about Floriday, an existing grower platform. Finally, we will discuss the conflicting requirements between JEM-id and Royal FloraHolland.

3.1. Buyers

For both KOA buyers and on-site buyers, the same rules apply: Each buyer can only perform a few actions. Firstly, a buyer can buy (a part of) the offer for the current clock price. Secondly, when a buyer made an incorrect purchase, they can communicate this to the auctioneer by stopping the clock. Stopping the clock is only allowed when the bought products are still on the clock; moving to the next item will commit the transaction. After stopping the clock, the auctioneer will contact the buyer and make a correction if needed. The auctioneer can also contact a buyer when they find the price unreasonably high as a precaution.

The buttons for these actions may differ per location, but every location always includes a button that both stops the auction and starts the transaction. Some locations provide specialised hardware and complete the transaction within one button press. Other locations use a voice-chat in which a buyer indicates the quantity they want to buy after winning.

Buyers physically present in the auction room have an advantage over KOA buyers in a few ways. First of all, each table in the auction room is directly wired to the clock system, meaning button presses from on-site buyers may potentially reach the auction system faster. Secondly, buyers physically present may have the opportunity to see the state of each product in real life instead of from a picture. This opportunity gives a great advantage when the taken picture is not of very high quality. Finally, depending on the auction room, the room might provide specialised hardware with a buy-button for several predefined quantities. If provided, buyers can efficiently indicate the number of containers to buy in a single button press.

KOA buyers have one obvious advantage over on-site buyers: KOA buyers do not need to be physically present at any specific location to buy from that location. This advantage implies that KOA buyers may buy from many different clock auctions at once, watch all prices and buy from the cheapest clock that day. Being on-site and using KOA can be combined to get the best of both worlds: an on-site buyer may bring their laptop to keep an eye on the other auctions through KOA.

The digital clock should aim to minimise these differences and, where possible, should try to unify all the advantages of using KOA and being on site while removing the disadvantages of both. This means that the connection should be stable for all clients. Furthermore, they all should have access to the same information about the products. Finally, all clients should be able to configure hotkeys so that they can buy easily.

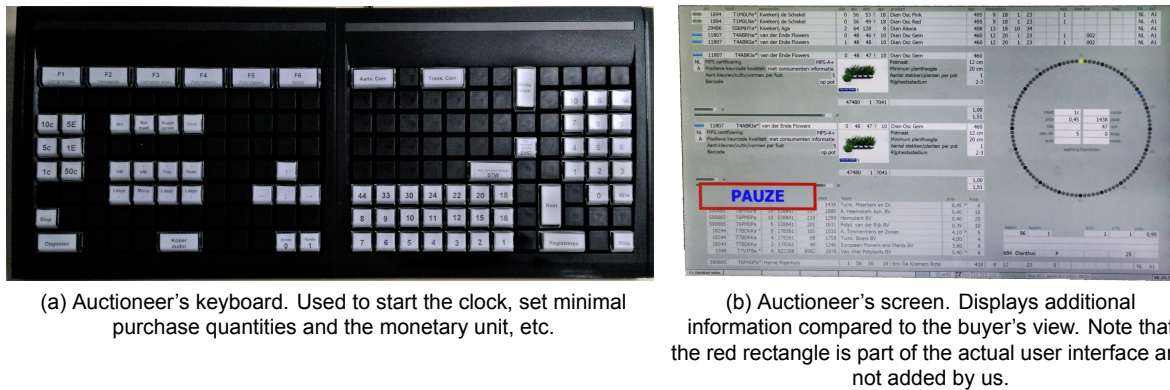


Figure 3.1: The tools used by an auctioneer to lead the auction.

3.2. Auctioneer

Each clock auction is overseen by an auctioneer. Auctioneers fulfil this role by observing all bids and detecting price changes during the auction while simultaneously keeping the auction running. It is their job to act on changes and irregularities during the auction. An auctioneer can influence many things, like the speed of the auction, the starting position of the clock, whether a transaction is approved and the minimum quantity to buy. Auctioneers have the power to ensure every grower gets a similar price for their goods, independent of what time their product is up for auction. Otherwise, growers whose products come earlier during the auction might get better prices for their produce.

An auctioneer has to perform these actions swiftly. For instance, changing the minimum amount per transaction may happen between auction rounds, which usually last only a couple of seconds. For this, the auctioneer has a special keyboard (see [fig. 3.1a](#)). This keyboard allows them to change the minimum purchase quantity of an item in a single button press, amongst other things. The advantage of this is that an auctioneer can do many actions with little button presses, making the auction run smoothly.

Auctioneers have a specialised view of the auction (see: [fig. 3.1b](#)). This screen displays many essential pieces of information to the auctioneer. In the top-left, the next six items up for auction, and a detailed overview of both the next and the current item are displayed. These detailed descriptions include information about the product and its grower. A common point of interest is the quality of both the product and its grower, and the features of the product such as colour, length of the stalk and size of a container. In addition to the information specifically about the product, the auctioneer also gets to see information about containers sold during presale in the bottom-right of the screen. The auctioneer has the same clock element as the buyers. Finally, the auctioneer has access to all previous transactions made during the auctioning of the current product in the bottom-left. When a buyer reports a mistake, the auctioneer can use this information to judge and overrule the faulty transaction. Once the next product in the queue is auctioned, these transactions can no longer be amended.

3.3. JEM-id

JEM-id is a software company that develops software for the agricultural sector. JEM-id proposed the request to build an online auction clock. This online auction clock would be a great addition to their current software, especially Floriday (see [section 3.4](#)): integrating the clock with their existing tools will provide easier access for growers and buyers using the platform. JEM-id has a clear vision of the product and how its architecture should be designed to make for an efficient and integrated auction.

JEM-id has proposed the model displayed in [fig. 3.2](#). Here, Floriday is external software, to which we can send API requests. Floriday manages the prices and pictures of plants and flowers, and information on the stock of growers, which the clock can use. To the left of Floriday, we can find the CMA, which stands for Clock Management API. This service should manage clocks and be allowed to create and

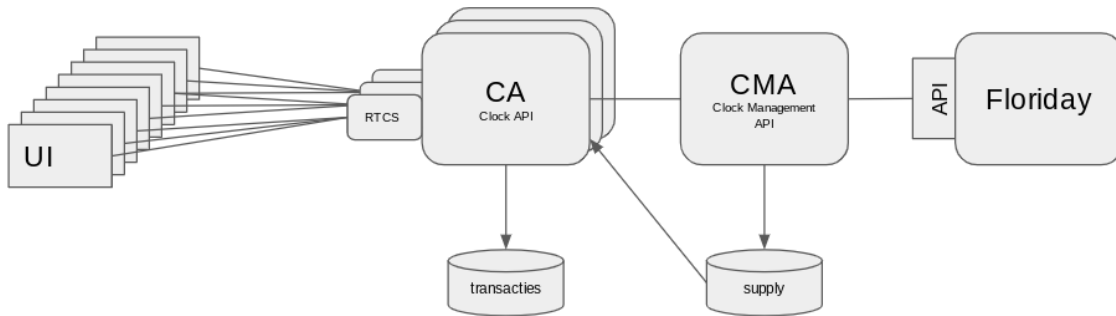


Figure 3.2: The model as proposed by JEM-id.

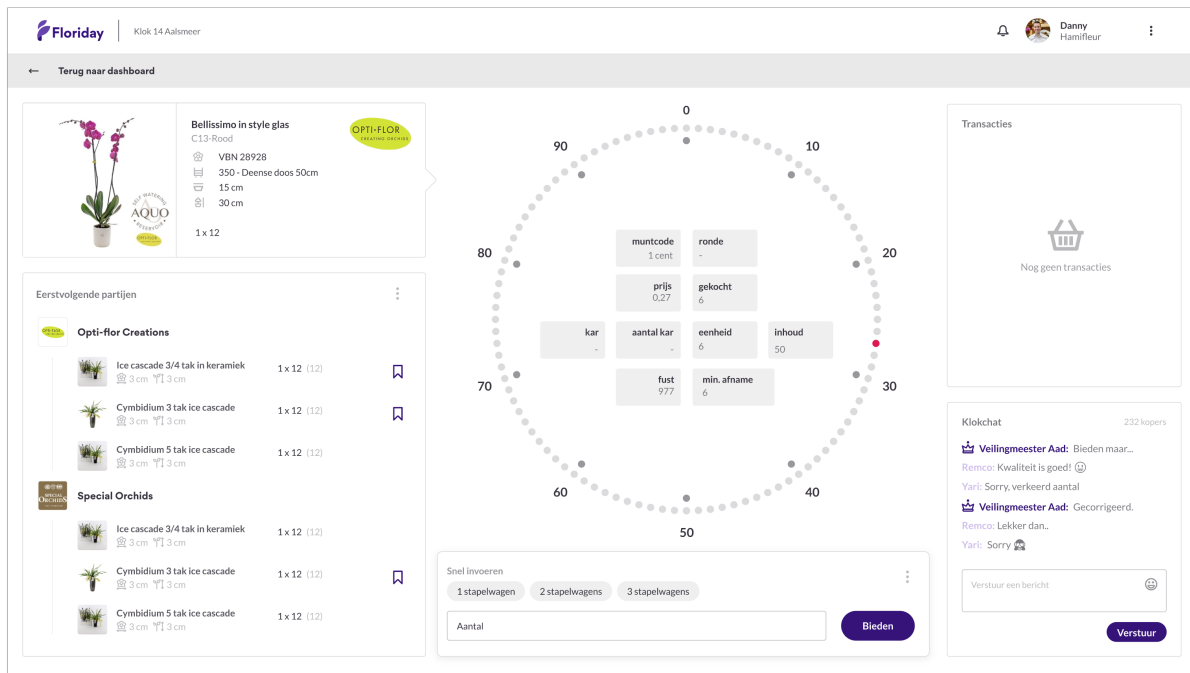


Figure 3.3: The new clock layout proposed by JEM-id. Note that this is a mock-up, not an actual UI.

destroy clock instances. CA, which stands for Clock API, represents an actual clock auction instance. There can be many of these, all managed by the same CMA. Every CA has multiple RTCSs, which stands for Real-Time Communication Service. This RTCS manages a connection with the actual client (one or more), which is represented by the UI. When an RTCS fills up, the CA requests a new one. See also [appendix C.3.3](#) for more details.

JEM-id has tasked us with implementing the UI, RTCS and CA, which is the clock and its communication with clients. Although the CA does communicate with the CMA, the CMA is beyond the scope of our project.

3.3.1. The UI

JEM-id proposed the UI for a buyer to be [fig. 3.3](#). From the image, we can already see several requirements proposed by JEM-id.

In the left part of their design, we can see the details of the current auction. Below, we see the upcoming products. In the middle, we have the clock, displayed as a circle consisting of smaller circles. The current price is coloured red. Additional sale information, such as the minimum number of products to buy, is displayed inside the circle. Underneath is the buy menu, where a buyer first has to enter

how much they want to buy before they can place a bid. On the right, we see the user's previous transactions. Beneath it, we can find a 'clock chat', in which a buyer can talk to the auctioneer. This chat comes into play when a buyer places a faulty bid.

3.3.2. Back-end features

Besides the features that are apparent from the UI, there are other features beneath the surface. One such feature is the decision process on who is the winner. Royal FloraHolland would like this to be as fair as possible, which means that, ideally, the first bidder should be the winner.

Another feature is for the CA to send all transactions to the CMA, so that Floriday and internal databases may be used to further process transactions.

When a buyer places a bid, a so-called 'grace period' is started. In this period, which lasts several hundred milliseconds, bids can still be placed. Because of this grace period, people who have a disadvantage due to delay can be compensated by evaluating all incoming requests. When the period has passed, rules defined by the auction will be used to pick a winner. For example, we could use the one that was ultimately placed first or the one with the highest price per item.

A final, somewhat hidden feature is client alerts. Clients should receive an alert on several occasions like winning (or losing) an auction or reaching the minimum price (which means closing the auction).

3.3.3. Future expansions

In the future, JEM-id would like to expand the digital clock. They would like us to take such future expansions into account already. Designing the system in an extensible way will allow later development to be smoother. Some of these features that we will keep in mind, but not pursue, include:

- Managing several clocks using the CMA. This CMA exposes information about the supply to the different CAs. We should thus anticipate the CMA being a standalone API.
- Internationalisation. It would be useful if people all over the world can use the system in their own language. Therefore we should build the system so that it has support for multiple languages.
- New clock layouts. The current layout is based on a rather old layout. Revising this layout in the future might be useful. Therefore, the UI should be extensible so that different UIs are interchangeable.
- Floriday authentication. Users should eventually be able to sign into the system using their Floriday account. See also [section 3.4](#).

3.4. Floriday

Quoting the website, 'Floriday is the worldwide digital platform for the floriculture sector that helps the grower to organise the commercial processes efficiently' [7]. It is an application developed by JEM-id and Royal FloraHolland, launched in 2018. Its purpose is to help growers organise their stock, orders and logistics. They can also direct their stock to several channels, like the (analogue) clock.

This system could be connected to our digital clock API later on (as mentioned in [section 3.3.3](#)). Floriday already allows growers to set different prices for each buyer, so it would be easy to add a minimum price for the clock auction to this data model. Floriday itself is easy to connect to since it is built using a microservice architecture.

3.5. Conflicting requirements

Where JEM-id is our client, Royal FloraHolland is their client. We should thus try to satisfy both as much as possible. We have encountered several differences in their wishes and will address them here.

The first difference we noticed is the minimum price for a given product. Where JEM-id wanted to hide this information, the current auction actively shows this with a blue dot.

Another disagreement is the chat proposed by JEM-id. Currently, a buyer can stop the auction, and by doing so, inform the auctioneer of the buyer's mistake. After this, the auctioneer makes contact with the buyer. This system could be seen as an active phone call with hundreds of people in it. JEM-id proposes a chat to replace this system. Royal FloraHolland, however, saw several problems with this, the first of which is a potential message overload. Buyers might feel less of a boundary sending a message because a chat is less formal. This can cause the chat to go too fast for the auctioneer to read and act upon. Another is the slower speed of typing when compared to talking, which means that taking corrective measures against a wrongly placed order might take longer. Instead of a chat, Royal FloraHolland proposed a standardised set of messages instead, from which the buyers can choose. The advantage of this system is that it keeps communication formal. One can also restrict the messages a particular buyer can send. Finally, if needed, a private chat can be opened to get more information about the situation. It was clear, however, that the standardised set of messages was still not considered optimal by Royal FloraHolland.

A final difference was the power of the auctioneer (and administrator). Where JEM-id gave the auctioneer authority to alter the minimum price for a product, Royal FloraHolland does not. According to the latter, this is even legally prohibited.

4

Design

In this chapter, we will explain how our software is designed. An agreement on the architecture of a software project is essential for ensuring that development goes well. As such, we will expand on the relevant topics regarding software design in this chapter.

We will first list the requirements upon which our designs are based in [section 4.1](#). We will then describe the design goals that we used while coming up with designs for the software product. Then we will present the final architecture of the software system, and how it differs from the initial designs.

4.1. Requirements

In order to create a satisfactory product that will be accepted by our client, we needed to have a list of requirements against which we could build and validate our software. The requirements are stated below, and are prioritised using the MoSCoW method, which divides requirements into *must haves*, *should haves*, *could haves* and *won't haves*. Furthermore, we make a distinction between functional and non-functional requirements.

4.1.1. Functional requirements

This section will describe the functional requirements given by both JEM-id and Royal FloraHolland. These requirements describe what functionalities the final system should have.

Must have

- The system must be simulatable.
- The system must be able to scale dynamically to support a large number (1000+) of simultaneous client connections.
- There must be a graphical user interface which sufficiently resembles the existing clock system, to ensure user familiarity.
- A buyer must be able to place a bid on the clock.

Should have

- The system should have a graphical user interface for the auctioneer which sufficiently resembles the existing user interface.
- The clock bounce should be configurable by the clock administrator.
- The clock speed should be constant and be configurable by the clock administrator.
- The graphical user interface should be fully responsive to enable interactivity on any device size.
- The clock data should come from a central system that is able to manage all clock system instances.

- The results of the auction should be stored in this same central system.
- The user interface should have an option to switch between clocks.

Could have

- The system could have a standardised set of messages that can be broadcast to the auctioneer. These messages can be sent by the buyer if they make a mistake.
- Product information that is relevant to buyers, including photos, could be displayed in the live auction feed.
- The user could be able to switch between a clock view and a numerical view.
- The graphical user interface could have support for multiple languages.
- The system could have an alarm functionality that allows buyers to receive a notification once a product is about to be auctioned.
- It should be possible for the users to pause participation in the auction process (switch to spectator mode).

Won't have

- The system won't have a chat function.
- The system won't have voice-over-IP (VoIP) functionality.
- The system won't have an artificial intelligence system to automatically control auctions.
- The realtime communication systems won't be started in multiple geographical locations.
- The system won't have wish-list functionality that automatically updates when a buyer purchases something.
- The system won't have grower reviews.

4.1.2. Nonfunctional requirements

This section will describe the nonfunctional requirements given by both JEM-id and Royal FloraHolland. Such functionality could, for example, be about the performance and environment of the software system. This section, therefore, concerns any requirements that are not functional.

Must have

- The back-end code must target a .NET Standard-compliant environment.
- The system must be cross-platform deployable.
- The system must be deployed on a cloud computing service.
- The system must be maintainable for developers after the project is finished.
- The system must be as fair as possible to all parties. By this we mean that all auction buyers are treated equally, regardless of non-excessive network delay or physical location.

Should have

- The back-end should be written in C#.
- The source code of the system should follow the C# coding conventions [8].
- The front-end of the system should be written in TypeScript.
- The front-end of the system should be written using the React front-end library.
- Front-end styling should be embedded in the TypeScript source code.
- The system should have no compilation warnings.

- The realtime communication service should be scalable on multiple cloud computing instances. If load is heavy, it should then be possible to add more computing power so that the load is distributed.
- The functionings of the system should be documented.
- The front-end should use the Floriday UI package to reuse UI functionality.
- The system should have a high performance, so that its response times are less than one second.
- The system should be secure, so that no data belonging to one user is sent to another.
- The system should have a high availability, so that a user can always use the system as expected without issues.

Could have

- The system could run in a Docker container to enhance portability.
- Automatic tooling could be used to enforce proper documentation.

4.2. Design goals

In this section, we will describe the design goals that were derived from the requirements given in [section 4.1](#). These goals were set up to ensure that the final product is of high quality.

Scalability

The system should be designed to be fully scalable. This means that it should continue to perform well when a large number of users are connected. Royal FloraHolland is planning to unify several location-bound auction clocks into one country-wide auction clock, simplifying both the buyer and logistics processes. Estimates are that a single auction clock will serve around ten thousand (10,000) users simultaneously if the auction is made available worldwide.

Such scalability includes performance guarantees that are necessary to facilitate a real-time auction process. For example, Royal FloraHolland considers a network latency between 0-30 milliseconds to be fast. Buyers with a slower network connection can be disadvantaged. This meant that our system would have to be able to maintain connections with all clients and send data to all of them, even when many clients are connected.

Portability

The final product should be highly portable, both in the backend server and the front-end application. To begin, the backend must be written in a language that can target many platforms, so that it can be deployed on any operating system or cloud provider. This allows the client to deploy it using any server provider, or even use multiple providers if necessary.

The front-end should also be portable. We have chosen to build this front-end as a web application: any user will reasonable have a modern web browser installed on their device of choice, given that they regularly use a web browser to consume other online services. Building the front-end as a web app also has the benefit that a user will not have to install additional software to use it: the web browser should be enough. We support all modern browsers with a market share greater than 1% of the total browser market by using the tool Babel [9].

Usability and familiarity

While the system is intended to be a completely new system, without any technological constraints of the old system, it must still have some resemblance of the old system. This way, users will still know how to use it and can recognise it as something familiar. For example, we aim to implement the clock graphic that all users of the current system know. We think that such familiarity can significantly benefit the adoption rate of the current system. After all, users are far more likely to pick up a new user interface with fewer changes than when there are many changes.

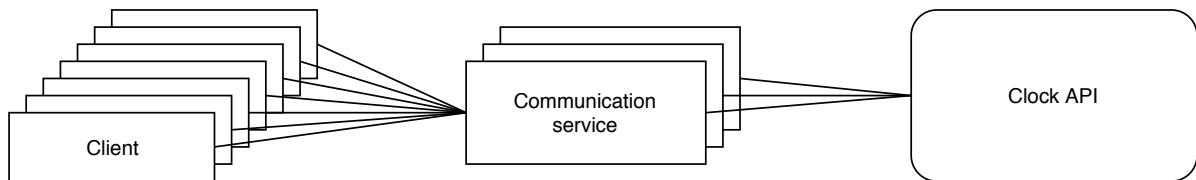


Figure 4.1: The system architecture.

Maintainability

It should be easy to maintain the current system. As such, it should be simple to add, modify or delete features. We can do this by making the code modular and readable. Furthermore, it should be possible to replace one running service with another easily. This allows us to achieve high availability so that a user can connect at any time.

4.3. Architecture

In this section, we will go into detail about the architecture of our software project. We will first give a high-level overview of the components in the system. We will continue by explaining the flow of signals that are exchanged between the different components. This section is not meant to provide any implementation details. These will be given in [chapter 5](#).

4.3.1. Components

We will now describe the high-level architecture of our system. The main priority of the system is high scalability. We define a system as highly scalable if it has two properties. The first is the ability to connect with a large number of clients simultaneously. The second one is that it does not suffer from performance losses that may potentially destabilise the system. If the system were to destabilise, it could prevent potential buyers from purchasing the goods they want, or give users a negative experience with the system.

Besides being scalable, the system must also be highly performant. If it were not performant, scalability would not matter, as the system would still be slow. As such, our architecture should provide us with the freedom to create a system that is modular, performant, and potentially distributed to some degree. As previously described, only latencies of 30 milliseconds or less were considered to be fast, which is a rather small margin of error.

Upon starting the project, we were given a design by JEM-id. Since JEM-id is a software company, it has some opinions on how the architecture and graphical user interface might be realised. We have evaluated this design and decided to keep it, as we found no blocking issues with this set-up. The design consists of three main components: the clock API, several real-time communication services, and the clients connecting to the clock auction. This architecture design is visualised in [fig. 4.1](#) and is meant to be a microservices-oriented system, where microservices are defined by Lewis *et al.* [10].

The central component of our architecture is the *clock API*. We take the concept of a single auction clock and translate it to an API service. This clock API maintains the clock state, allows buyers to bid, and generally attempts to keep all connected clients in sync. It should only ever maintain one single auction clock. If we want to have multiple clocks, several clock API services are started. This ensures that each clock has dedicated computing power.

The next part of our design is the real-time connection services, shortened to RTCSs. The RTCSs are distributed. Furthermore, Multiple RTCSs can exist for one single clock API instance. These services are intended to serve as the edge of our network infrastructure: they solely exist to handle large volumes of connected users. This way, the responsibility of maintaining connections is distributed across the several RTCS instances, and the clock API can use its resources for ensuring that the performance of the entire system remains sufficient.

When starting an RTCS, it will connect to its clock service and register itself. The clock API will keep track of the different RTCS connections, but it does not keep track of the users connected to each RTCS. An RTCS will always attempt to keep a connection with its clock API, reconnecting if the live

connection fails. The RTCS services should be able to run on any given platform since the services are relatively simple.

Finally, we have the clients. These clients will run in the web browser of the user and are inherently massively distributed. The client application can fulfil two roles, namely buyers and auctioneers, depending on the type of user that logged in to the system. As with the current system, the buyer is only able to bid on items. Conversely, an auctioneer user can control the clock, but cannot purchase any stock.

Upon loading the web page, the application presents both types of users with a dashboard that allows them to select a clock they would like to use. A buyer user will only be able to see the clocks they are subscribed to, whereas an auctioneer can see all clocks. In the clock interface itself, the buyer will have a control panel that allows them to buy items and configure keyboard shortcuts. The auctioneer can see statistics and has an option to configure keyboard shortcuts.

4.3.2. Control

In our architecture, all control lies with the clock API service. It controls all state related to the auction. For example, it keeps track of the clock state and whether it is moving or not. It also stores the item currently being auctioned. This information is sent to all connected clients through different channels.

All buyer clients connect to the clock API service through an RTCS service. The clients first make a request to the clock API, which then returns a URL to one of the RTCS servers. This allows our system to do a bit of load balancing so that no one RTCS service is particularly crowded. In the future, this system could be improved such that an RTCS with a correct geographical location and low load is chosen. This may then decrease network latency even more.

Contrary to buyer clients, an auctioneer client connects to the clock API directly. This design was chosen because there will be many buyers, but at most a handful of auctioneers per clock in practice. As such, there should not be much of a performance loss on the clock API side. Even more so, removing the RTCS from the auctioneer chain could potentially result in lower network latency, thus improving the experience for all connected users.

The clock API transmits status change signals. Most signals are sent to all users, but some are restricted to a particular type of user. For example, the signal that a clock has started is sent to both buyer and auctioneer clients, since both users should have access to that event. Other signals, such as a *bid won* event is only sent to the user who won the current bidding session. This is done by broadcasting the signal to all RTCSs and then letting each RTCS decide if it should pass the signal on to one of its clients.

The auctioneer user can send control commands to the server. An example is the signal to start increasing the clock. The server responds with a status change and broadcasts this change to all connected clients, which then will display this information to the user. An auctioneer may also choose to skip a certain item. Again, the server registers the change, moves to the next item and lets all clients know using an update signal.

5

Implementation

5.1. Flow

In this section, we will describe the control flow that is used by the digital clock system. We will explain how the different microservices are implemented, and how they interact with one another. To do so, we will describe the actions taken by the different entities as they would typically occur. In this section, actions taken by an auctioneer or buyer are considered to be taken by the user interface. The implementation we describe here complements the model proposed in [section 4.3](#). External dependencies will be mentioned, but will not be described in detail.

5.1.1. Initialisation

All microservices in the system are loosely coupled: one microservice may depend on one or more other services, but these dependencies are not hardcoded and are resolved at run-time. Furthermore, no cyclic dependencies exist between microservices. The result of these choices is that services must start in a predefined order, to ensure that a service is available for its dependents. In practice, this constraint should not cause any issues as services may be restarted, and most may wait until the other services required are online.

When the actual system is fully deployed, this start-up sequence does not matter as much. In the end, only the Clock Manager API will be able to start CA instances, which will, in turn, be able to start their own RTCS instances. Therefore, once the CA is deployed, the system should be able to manage its initialisation automatically for the most part. The consequence is that the CMA will then be a single point of failure, which does need to be mitigated somehow. However, this is all beyond the scope of our project.

The lifecycle of our system starts with the initialisation of a clock API service instance. Since a clock API service instance is responsible for only one clock, several can be started if multiple clocks are needed. We envision that a management tool will exist in the future, that can automatically create a clock API instance, deploy it on a public cloud provider, and shut it down when the service is no longer needed.

Once the clock API is started, it is time to start one or more real-time connection services (RTCSs). An RTCS has a dynamically configurable URL to the public-facing interface of a clock API service stored in its configuration. The RTCS will initialise a SignalR connection with the clock API, after which both can communicate in real-time. This connection will likely use WebSocket connections, but SignalR will fall back to other methods if WebSockets cannot be used for any reason. After setting up this connection, the RTCS transmits some information about itself, such as the URL at which it can be accessed.

In the future, the provisioning and initialisation of RTCS instances could be managed dynamically by the clock API. If it detects, for example, that many users are connected, it could start a new RTCS to ensure that no RTCS will be overloaded. Currently, the starting of RTCSs must be performed manually, which is sufficient for debugging, testing and simulation purposes.

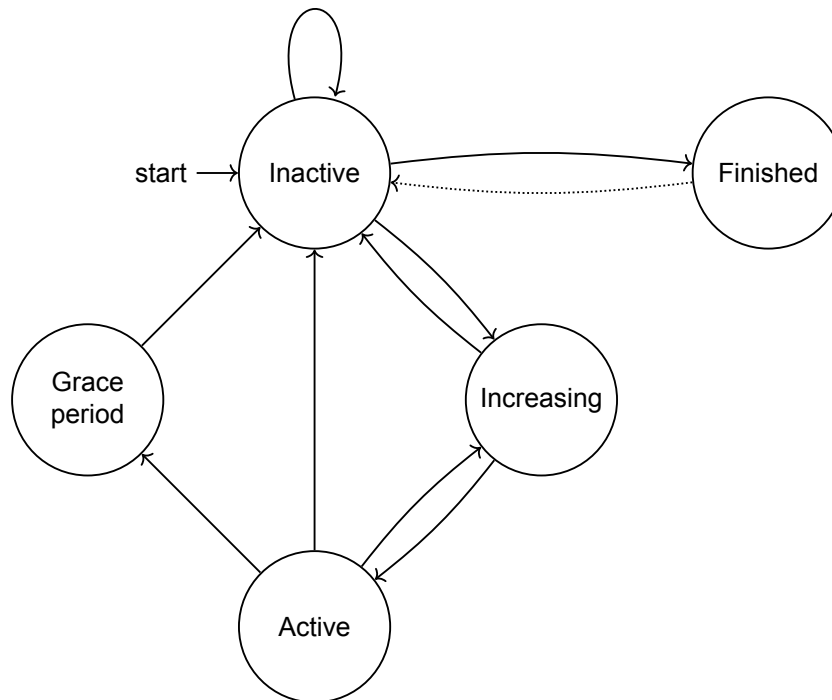


Figure 5.1: A state diagram representing the states of the clock.

Once both services are available, the user interface (UI) client can start initiating connections to one or more services. We again consider two types of users here: auctioneers and buyers. A buyer creates simple HTTP request to the clock API, asking for a URL to one of the RTCSs connected to the clock API. If none exist, the clock API service responds with HTTP status code 503, which means that no service is available [11]. After the clock API returns such a URL, the client connects to the given RTCS instance using SignalR. Once the connection is established, the client is ready to send and receive messages from the clock API, via the RTCS.

If the signed-in user is an auctioneer, a real-time connection is established as well, but not via an RTCS instance. We decided to directly connect the auctioneer to an instance of a clock API since there will be only one auctioneer in practice. This decision means that an auctioneer is not dependent on an RTCS instance being online, and the auction can still be controlled, should a critical error occur on one of the RTCSs. The separation of buyers and auctioneers also has some security benefits, which we will describe in [section 5.4](#).

5.1.2. Auction process

Once the auction is in progress, the auctioneer fully controls the clock. At its core, the clock rotates through five predefined states, which are *Inactive*, *Increasing*, *Active*, *Grace period* and *Finished*. In this section, we will describe how and when these state transitions occur. A visual representation of the transitions is displayed in [fig. 5.1](#).

When the clock is initialised, it is in the inactive state. This state means that the clock is idle and waiting for auctioneer input. In this state, the clock will still maintain values such as the current auction item, the last known price, and other values that are useful when a client connects mid-auction. To start the auction process, the auctioneer client can invoke methods on the clock API using SignalR. Most clock control calls also transmit the price that the auctioneer clock displays, so that all clients can automatically sync if needed.

Once the auctioneer presses the *Increase* key on their keyboard, the UI invokes the `IncreaseClock` call on the clock API, which transitions to the *Increasing* state. This action is then broadcasted to all clients (both auctioneer and buyers), who in turn start an animation in which the clock is increasing. This way,

all animation duties lie with the user interface; the server does not actively animate the clock.

The auctioneer will at some point release the *Increase* key, which means that the clock should stop increasing, and start decreasing. For this, the auctioneer UI sends the `StartClock` signal. When the clock receives the message, it transitions to the *Active* state, registers the start time and starts a timeout delay task that will fire when the price reaches the minimum auction item price. Despite not actively animating the clock, the server does know what the current price should be by computing the time difference between the current time and the start time and multiplying that value by the number of steps per second.

Now that the auction has started for a specific auction item, there are three possible outcomes: a buyer places a bid, the auctioneer interrupts the auction, or the timeout delay task will fire. In the first case, the clock service will validate and store the bid if it is valid. An example of an invalid bid is when the price attached to the bid (which is set by the buyer's UI client) is lower than the price that is known to the server. After all, the price sent by the client can never be lower than the one the clock API computes, especially when taking into account network delays. In the second case, the auctioneer may interrupt the auction by increasing the clock again, or placing it in a paused state. In the latter case, the clock API detects that no bids have been placed once the clock price reaches the minimum item price. The clock API will then move on to the next item as the current item is not wanted for a price above the minimum price. In both cases, the clock API will first move to the *Grace period* state, during which it will still accept bids that may be delayed by network delays. Depending on the winner selection algorithm used (see also [section 5.2](#)), it may or may not consider such bids.

Once the grace period has ended, the clock API moves back to the *Inactive* state and will no longer accept incoming bids. If any stock of the current item remains, it will decrease the remaining quantity by the quantity purchased in the last bid. All clients are notified and can update their indicators accordingly. If no quantity of the current product remains or the item was skipped, the next item is selected, and all clients are again notified. If no next item is available, the clock API moves to the *Finished* state and broadcasts the new state.

When the clock is in the *Increasing* or *Active* state, the auctioneer can pause the auction. The clock is then immediately moved back to the *Inactive* state. All clients are notified of this action, and their clock animations stop. Furthermore, a visual indicator indicates that the auction is paused.

A final noteworthy state transition in the diagram is the dotted one from the *Finished* state to the *Inactive* state. During normal operations, this transition should not occur and is therefore not allowed. However, for debugging purposes we have included an option to reset the clock, which sets the current auction item to the first one in the list, and moves the clock back to the *Inactive* state.

5.2. Winner selection

In this section, we will describe how the winner of a bidding session is selected. We will first describe what exactly a bidding session is, and then we will describe how a winner is selected.

When the clock starts after being increased, a new bidding session is created. In practice, this is a unique string that is broadcast to all clients upon starting the clock. Any client placing a bid must include this unique string in its request to place a bid, which allows the clock API to verify if the bid is placed on the correct item. The actual implementation of this mechanism uses the GUID structure in .NET to automatically generate a new string. Such a string has a very low probability of occurring more than once [12]. We consider this enough of a safeguard to ensure that our clock system accepts no incorrect bids.

When the grace period has ended after being triggered by a bid, the system has a list of bids. If only one buyer placed a bid, there will be only one bid. If several buyers placed a bid (all except the first one arriving during the grace period), there will be multiple bids in this list. We have implemented several ways of deciding a winning bid, called *arbitrations*:

- **First:** Select the first bid that came in. We have aimed to make this algorithm more accurate by registering the timestamp of the moment the bid arrived at the RTCS instance. This negates a large portion of network delay if the RTCS is geographically close to the client.

- **Highest price:** Select the bid with the highest price per unit. This was one of our original ideas for creating an arbitration system that would be entirely fair, as it is not dependent on timing and/or network delays.
- **Highest quantity:** Select the bid with the highest quantity.
- **Multi transaction:** Originally an experiment by Royal FloraHolland, this method aims to give all buyers whose bids were accepted during the grace period the quantity they attempted to order. Thus, multiple transactions can be generated during one bidding session.

The implementation of these arbitrations is relatively simple. They are functions that accept a list of bids, the current price of the clock, and the remaining quantity of the current auction item. The return type is again a list of bids since they should be able to return multiple bids that have passed the checks. However, there are functions which return a list containing a single item.

These functions are invoked by an instance of the `ArbitrationService` class, which is a helper service. It uses a factory object to obtain the correct arbitration function, based on the configuration with which the system was started. We decided to make this a configuration option so that this is easily configurable later on. As bids come in, they are stored in the arbitration service. Once the grace period ends, the `GetWinners` method is invoked, which runs the arbitration function on the list of bids that have arrived and returns the list of winning bids.

5.3. SignalR communication

In this section, we will describe the different SignalR messages send between the frontend and the back-end. We will make a distinction between incoming and outgoing messages, from the perspective of the frontend. Thus, incoming messages are sent from the server to the frontend while outgoing messages are sent from the frontend to the server. [Section 5.3.1](#) describes all SignalR messages that can be received by the frontend while [section 5.3.2](#) describes all messages that the frontend can send. Furthermore, it is irrelevant for the frontend to know whether it is connected to a communication service (buyers) or directly to the clock API (auctioneers).

5.3.1. Incoming messages

In this section, we will describe incoming messages. The server sends these messages to one or more frontend clients connected to them over a real-time connection.

ClockIncreaseStarted(startPrice)

Sent to All clients
Parameters startPrice 32-bit integer The price at which the clock has started increasing.
Description This message is sent when connected clients should start the 'clock increase'-animation.

ClockStarted(startPrice)

Sent to All clients
Parameters startPrice 32-bit integer The price at which the clock has started decreasing.
Description This message is sent when connected clients should start the 'clock decrease'-animation.

NewItemStarted(index)

Sent to All clients
Parameters index 32-bit integer The index of the new item.
Description This message is sent when the clients should be notified that a different item is being auctioned.

NewBiddingSession(biddingSession)

Sent to All clients
Parameters biddingSession string The GUID of the bidding session that was started.
Description This message is sent when the auctioneer has released the increase button, and the bidding session had started

QuantityChanged(quantity)

Sent to All clients
Parameters quantity 32-bit integer The new remaining quantity.
Description This message is sent when the remaining quantity of the current item has changed due to it partially being sold on the auction.

RoundChanged(round)

Sent to All clients
Parameters round 32-bit integer The new round number of the clock position.
Description This message is sent when the auctioneer has manually changed the round number.

StepSizeChanged(stepSize)

Sent to All clients
Parameters stepSize 32-bit integer The new step size in tenths of cents.
Description This message is sent when the auctioneer has changed the step size.

AuctionFinished()

Sent to All clients
Description This message is sent when all items are sold.

AuctionPaused(price)

Sent to All clients
Parameters price 32-bit integer The price at which the auction was paused.
Description This message is sent when the auctioneer has paused the auction or when the auctioneer was disconnected.

GracePeriodStarted(price)

Sent to All clients
Parameters price 32-bit integer The price at which the grace period was started.
Description This message is sent when the grace period was started. The auctioneer always receives this message. However, the buyer only receives this message if this is enabled in the configuration.

BiddingSessionWon(biddingSession, index, quantity, price)

Sent to Buyers
Parameters biddingSession string The GUID of the bidding session that was won.
index 32-bit integer The index of the item that was purchased.
quantity 32-bit integer The quantity of the item that was purchased.
price 32-bit integer The price per item at which the items were bought.
Description This message is sent when a buyer wins a bidding session.

BiddingSessionLost(biddingSession, index, quantity, price)

Sent to Buyers
Parameters biddingSession string The GUID of the bidding session that was lost.
index 32-bit integer The index of the item that was purchased.
quantity 32-bit integer The quantity of the item that was purchased.
price 32-bit integer The price per item at which the items were bought.
Description This message is sent when a buyer loses a bidding session. Multiple of these messages may be received during a multi-transaction auction.

BiddingSessionExpired(biddingSession, index)

Sent to Buyers

Parameters biddingSession string The GUID of the bidding session that was lost.
index 32-bit integer The index of the item that was not purchased.

Description This message is sent when the minimum price of a product is reached.

WinningBid(biddingSession, index, userId, quantity, price)

Sent to Auctioneers

Parameters biddingSession string The GUID of the bidding session that finished.
index 32-bit integer The index of the item that was purchased.
userId string The identifier of the winning user.
quantity 32-bit integer The quantity of the item that was purchased.
price 32-bit integer The price per item at which the items were bought.

Description This message is sent when a buyer wins a bidding session.

TotalNumberOfBids(biddingSession, index, bidCount)

Sent to Auctioneers

Parameters biddingSession string The GUID of the bidding session that finished.
index 32-bit integer The index of the item that was purchased.
bidCount 32-bit integer The number of valid bids placed during the grace period.

Description This message is sent when a buyer wins a bidding session.

NumberOfClientsChanged(amount)

Sent to Auctioneers

Parameters amount 32-bit integer The new total number of buyers connected to the auction.

Description This message is sent when a buyer connects or disconnects.

5.3.2. Outgoing messages

In this section, we will describe outgoing messages. These messages are sent by frontend clients to the service they are connected to over a real-time connection.

PlaceBid(biddingSession, price, quantity)

Sent by Buyers

Parameters biddingSession string A GUID identifying the targeted bidding session.
price 32-bit integer The price at which the bid was placed.
quantity 32-bit integer The quantity the user wants to buy.

Description The message is sent when a buyer wants to place a bid. All parameters are validated by the clock API.

Register(userId)

Sent by Buyers

Parameters userId string The user's ID.

Description This message is used for development purposes to differentiate different buyers, while using client credentials for authentication [13], before proper authentication is implemented as described in [section 5.4.1](#).

RoundTripTime()

Sent by Buyers

Result string ISO 8601 [14] formatted time stamp of the local server time.

Description The message is sent when a buyer wants to get the round trip time (RTT) to the communication service it is connected to. The RTT can be retrieved by measuring the time between sending this message and getting the resulting value. This result can be discarded.

IncreaseClock(startPrice)

Sent by Auctioneers

Parameters startPrice 32-bit integer The price at which the clock has started to increase.

Description This message is sent when an auctioneer starts to increase the clock.

StartClock(startPrice)

Sent by Auctioneers

Parameters startPrice 32-bit integer The price at which the clock has started to decrease.

Description This message is sent when an auctioneer releases the increase button, and the bidding session starts.

ResetClock()

Sent by Auctioneers

Description This message is sent when an auctioneer resets the auction currently active on the clock.

SetRound(round)

Sent by Auctioneers

Parameters round 32-bit integer The number of the new round.

Description This message is sent when an auctioneer changes the round number during a clock increase.

SetStepSize(stepSize)

Sent by Auctioneers

Parameters stepSize 32-bit integer The new step size in tenths of cents.

Description This message is sent when an auctioneer changes the step size currently used by the clock. This message can only be sent if the clock is inactive or increasing.

Skipltem(biddingSession)

Sent by Auctioneers

Parameters biddingSession string The bidding session GUID used of the session that needs to be skipped.

Description This message is sent when an auctioneer skips the item currently being auctioned if anomalies in the auctioned item are detected by the auctioneer.

Pause(price)

Sent by Auctioneers

Parameters price 32-bit integer The price at which the clock was paused.

Description This message is sent when an auctioneer manually pauses the auction.

GetClientCount()

Sent by Auctioneers

Result 32-bit integer The current number of connected buyers.

Description This message is sent when an auctioneer establishes a connection and requests the current number of connected buyers.

5.4. Security

In this section, we will describe the security implementation of the system. We will start by describing how authentication occurs, and then we will discuss how privileges are enforced by the system in [section 5.4.2](#).

5.4.1. Authentication and authorization

All interactions between the different services are authenticated. For this we use the OpenID Connect 1.0 framework, which is built on top of OAuth 2.0 [15]. OpenID Connect supports authentication using the OAuth 2.0 protocol and allows for authorization using its own mechanisms. This ensures that a third party without valid credentials will not be able to obtain data that should not be accessible publicly, or with incorrect credentials.

All RTCS instances authenticate themselves against the clock API server using the OAuth 2.0 *client credentials* flow [16]. All clients store a secret key that is securely transmitted to the central OpenID connect server using an ordinary HTTP request [16]. This identity server then returns an encrypted JSON Web Token (JWT) [17]–[19], which can be validated by a service provider. The roles that a resource owner may assume are embedded in the JWT. Since the JWT is encrypted using the server's private key, this information cannot be modified by an attacker.

Clients also authenticate themselves using the OpenID Connect framework. This currently happens using the *client credentials* flow, which is considered insecure when used in a publicly available client. After all, the web client is distributed as a JavaScript bundle, so any malicious actor would be able to extract the client secret from the JavaScript code. We have chosen to use this flow because it makes debugging more straightforward, and is more convenient when simulating the system. In the future, this flow will be replaced by the *implicit flow* or the *authorization code with PKCE* flow, which do not require the client to store a secret securely. The application will likely be modified so that it uses Okta as its authentication provider, which supports these flows [20]. This does not require a significant change in the application: the only necessary change is modifying how the JWT is retrieved.

5.4.2. Privilege enforcement

Users have different roles within the system. A buyer user should never be able to control the state of the clock, except when placing a bid. To enforce this invariant, we use OpenID Connect scopes to determine the scopes a user may access [15]. This enables us to determine whether a user can connect with any given API endpoint or real-time connection.

The separation allows us to enforce that a buyer cannot control the clock. The clock control functions are only exposed to the real-time connection for auctioneers, not using any other interface. Buyers are unable to connect to this real-time connection and are thus not able to control the clock. Similarly, an auctioneer cannot connect to the buyer connection. Both connections can perform different actions, and we utilise this to separate what both user types can do.

In all scenarios, the auctioneer user is considered a trusted entity by the system. Whereas buyer cannot change the server state (except for placing a bid), the auctioneer can perform all kinds of state changes on the server, even actions that would typically not occur during a regular auction.

6

Simulation

Testing a system is essential to ensure that it performs well and is stable for the duration of the test. In our case, it is especially crucial that we can guarantee that our system will remain stable, even when it experiences heavy load from users. We found that a good way of simulating a heavy load could be achieved by simulating the users, using real-world data about actual users of the current system.

In this chapter, we will describe how we have tested and simulated our system. We will first describe the system we used for testing. Secondly, we will explain our testing methodology, and the numerical values used. Finally, the results of our simulation will be listed.

6.1. Setup

In this section, we will describe the software setup we have used to test our system. Although this simulation software is not exactly part of our final deliverables for the project, it is relevant to describe how it works and some shortcomings the system may have.

In our simulations, we would like to simulate the behaviour of many users. The original specification we based our research on specified that the system should be capable of handling 6000 users simultaneously. As we received more information from Royal FloraHolland, we found that the actual number of buyers using the system simultaneously is far lower than this number, usually in the hundreds. However, we have still built the system with thousands of users in mind. We have done this because the system may need to handle thousands of users in the future, should Royal FloraHolland decide to expand the clock system to other regions.

We use a setup similar to the one described by Noor *et al.* [21], with a few minor changes. It is infeasible to run a single client on thousands of physical machines, or even virtual machines. We would not be able to obtain and setup sufficient resources to perform such a massive run throughout our project. Instead, we have chosen to run multiple clients on one machine, dramatically reducing the resources needed. We will be describing the setup in the following paragraphs. A visualisation is given in [fig. 6.1](#).

We deploy a single clock API instance at a known URL. This clock API service will be the one used during the run. In addition, we set up one or more RTCS instances that will be used to connect clients to the clock API server. These RTCSs are given a configuration that points to the URL of the clock API server so that they can automatically connect. Once set up, the simulated components are ready to be tested.

To test these components, we set up more services. We first set up several *headless buyer* programs. These can initiate many real-time connections with one of the previously set up RTCSs. It does so in the same fashion any usual client would: it requests an RTCS url from the clock API and connects to that URL. The program can keep setting up clients until it reaches the number we have defined in its configuration file.

In addition, we start a single *headless auctioneer* client. This client connects directly to the clock API as

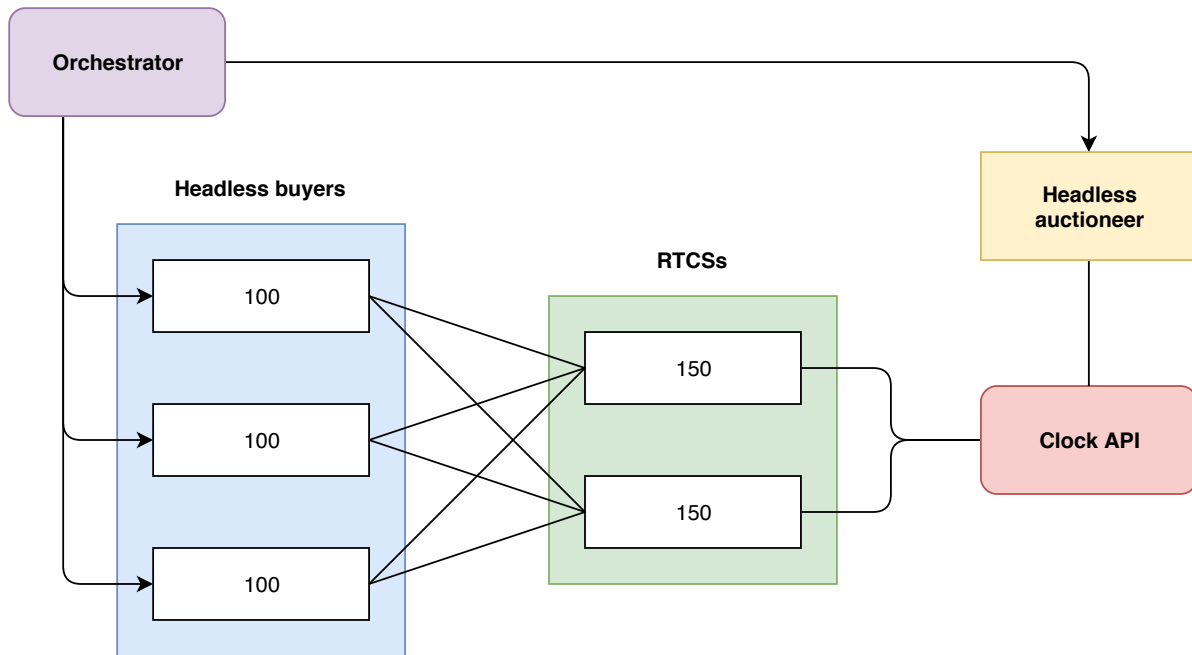


Figure 6.1: A visualisation of the setup we have used for our simulation system.

an actual auctioneer would. It increases the clock and starts it at pre-scheduled moments allowing the automated clients to place bids at pre-determined prices. A full auction can be executed automatically using both programs.

All clients, of both types, are controlled by an *orchestrator* service. The orchestrator has a component that can schedule actions. For example, it decides to what point the headless auctioneer client should increase the clock for each item. It also computes values at which a headless buyer should place a bid. These values, and the distributions they sampled from, are detailed in [section 6.2](#).

All headless buyer clients fetch this bidding schedule from the orchestrator and place a bid when it is their turn. The headless auctioneer client does a similar thing by increasing the clock to the price determined by the scheduler after every session. All clients respond to signals sent by the clock API service over the real-time connection, proxied by an RTCS in the case of buyer clients. In this way, we simulate the network traffic generated by an actual auction. This means that our simulation can potentially be used to stress test the network capacities of our system.

Finally, buyer clients and auctioneer clients keep track of some events, such as receiving the result of a ping or receiving a message that the current bidding session was won. When such an event happens, it is stored on the buyer before it gets sent to the orchestrator server.

6.1.1. Weaknesses and limitations

While the simulation described in [section 6.1](#) can provide insights on how the system will behave in practice, it may not be an accurate representation of reality. After all, we are simulating multiple clients on the same machine. This means that network delays would be the same for all clients. Although it is possible to introduce signal delays for individual clients, this is merely an approximation of reality and does not have to be accurate.

As such, we would like to point out that our simulation does not guarantee that the system will have a good performance. We have developed the simulator for two main reasons: load testing and verification of the system. We wanted to ensure that the system was able to handle large volumes of clients, and still perform correctly under such load. It was not built to guarantee that no errors will occur (which is impossible to do), and should therefore not be used to show that no errors will occur. A successful simulation program is good to have and can produce useful results, but should not be used for any other purpose than the one it was explicitly designed and built for.

6.1.2. Hardware and configurations

We have configured the simulation to simulate 610 clients, since we thought the application should be able to handle this with the current available resources. We decided the simulation should be repeated at least ten times. For this reason, we configured the auction schedule to include only the first 50 items, making the full simulation run for approximately 5 minutes. The first-come-first-served setting was used as the fairness constraint. Finally, clients were configured to record the ping between it and its RTCS server every 3 to 7 seconds. Additionally, they were configured to send their results to the orchestrator every 10 seconds.

To run the simulation, we required several services in the cloud and a number of computers to simulate the clients. For the final runs, we used the same hardware for each run. The orchestrator, clock and two RTCS services were all hosted by Azure within the same S3-tier resource plan (400 ACU, 7GB memory). The clients were distributed over four computers on two different networks. Finally, the auctioneer client was run on a separate computer on a separate network. For reproducibility, we will list the specifications of our machines below. All machines used a 64-bit operating system.

Wesley - Desktop

Processor	AMD Ryzen 7 2700X Eight-Core Processor, 3700 Mhz, 8 Core(s), 16 Logical Processor(s)
Physical memory	16 GB
Network speed	100 Mbps up / 100 Mbps down
GPU	AMD Radeon R9 290
Executed	183 buyer clients

Wesley - Laptop

Processor	Intel(R) Core(TM) i7-4720HQ CPU @ 2.60GHz, 4 core(s), 8 Logical Processor(s)
Physical memory	16 GB
Network speed	100 Mbps up / 100 Mbps down
GPU	NVIDIA Geforce GTX 960M
Executed	122 buyer clients

Paul - Desktop

Processor	Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz, 8 core(s), 16 Logical Processor(s)
Physical memory	32 GB
Network speed	700 Mbps up / 700 Mbps down
GPU	NVIDIA Geforce RTX 2070
Executed	183 buyer clients

Paul - Laptop

Processor	Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz, 4 core(s), 8 Logical Processor(s)
Physical memory	8 GB
Network speed	700 Mbps up / 700 Mbps down (after run 2)
GPU	Nvidia Quadro M1000M 4GB GDDR5
Executed	122 buyer clients

Chris

Processor	Intel(R) Core(TM) i7-2600 3.40GHz
Physical memory	10GiB 2x 4GiB, 1x 2GiB 1333MHz DIMM DDR3
Network speed	100 Mbps up / 100 Mbps down
GPU	(none, terminal session)
Executed	1 auctioneer client

6.2. Distributions

For our simulation, we have randomized several parameters. The distributions according to which they were drawn, are described in this section.

6.2.1. Distribution of the bidding price

Found distribution

The found distribution is dependent on both $x \sim U(0, 1)$ and the minimum price of the item that is being sold $minPrice$:

$$biddingPrice(x, minPrice) = minPrice \cdot (factor(x) + 1) \quad (6.1)$$

With the factor:

$$factor(x) = \text{gamma}^{-1}(1.1941, 5.9515) \quad (6.2)$$

where $\text{gamma}^{-1}(\alpha, \beta)$ is the inverse gamma function.

Used data

For this distribution, the columns *bidding price* and *minimum price* of dataset 2 with $n = 157466$ samples (see section D.2) were used. Since the simulation is based on dataset 1 (see section D.1), we cannot give actual prices based on a minimal price: the simulation will not use these minimal prices. Instead, we will look at the proportions, and give back a number from the distribution based on both a random number between zero and one, and the minimal price.

Data processing approach

From the minimum price and the bidding price, we calculated the difference as a factor with formula 6.3. Here, we calculate the additional percentage of the minimal price a buyer paid on top of the minimal price.

$$factor = \frac{bidding\ price}{minimum\ price} - 1 \quad (6.3)$$

In table 6.1, we have grouped the factor in bins with size two and counted the occurrences. Since it is highly unlikely someone places a bid at the minimal price, and it is impossible to pay less than the minimal price, the chance of these occurrences is set to zero and thus discarded.

After this, we plotted the bins against their chance of occurrence in figure 6.2a, and saw it resembled a gamma distribution. To find out the right parameters for the gamma distribution, we first laid a gamma distribution with $\alpha = 2$ and $\beta = 1$ on top of it in figure 6.2b. Then, we calculated the sum of squared errors (SSE) with formula 6.4. After this, we used Microsoft's Excel add-in program 'solver' to minimize the SSE [22]. This gave us the parameters $\alpha = 1.1941$ and $\beta = 5.9515$ with a SSE of 39,65 and an absolute error of 0,93% on average, shown in figure 6.2c.

$$SSE = \sum_{i=0}^n (D(i) - G(i))^2 \quad (6.4)$$

where:

SSE = Sum of standard errors

n = Number of samples

$D(i)$ = Cumulative chance of factor according to eq. (6.3) at bin i

$G(i)$ = Cumulative chance according to the gamma distribution at bin i

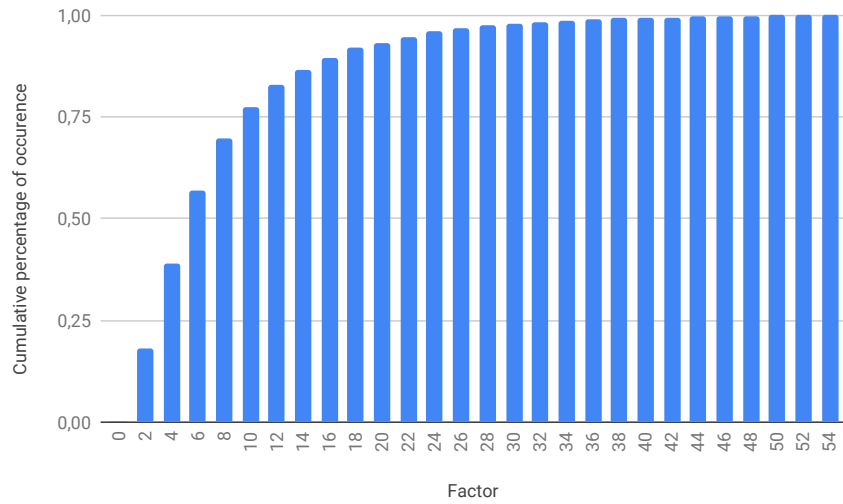
6.2.2. Distribution of the bought quantity

Found distribution

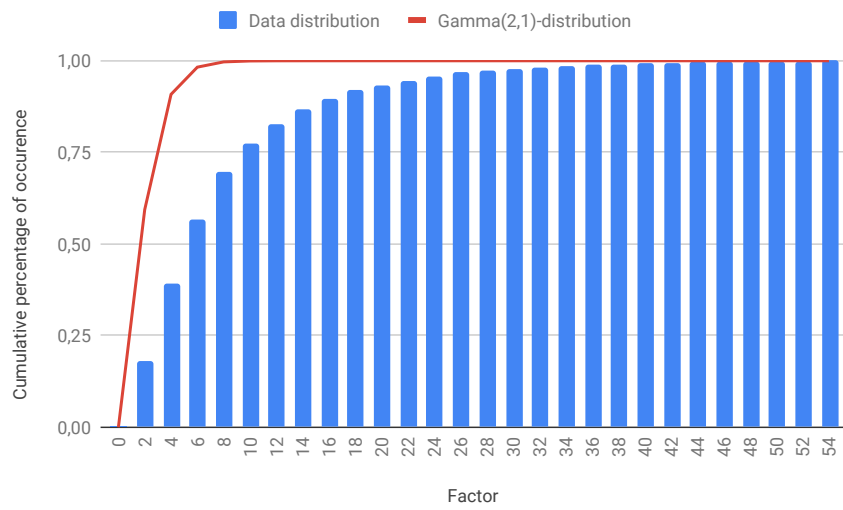
$$quantity(x) = \begin{cases} minimal\ amount - 1 & x \in [0, 0.1726) \\ minimal\ amount & x \in [0.1726, 0.8357) \\ maximal\ amount & x \in [0.8357, 0.9156) \\ maximal\ amount + 1 & x \in [0.9156, 1] \\ 0 & else \end{cases} \quad (6.5)$$

Table 6.1: Bins of the factor of bidding price and it's properties

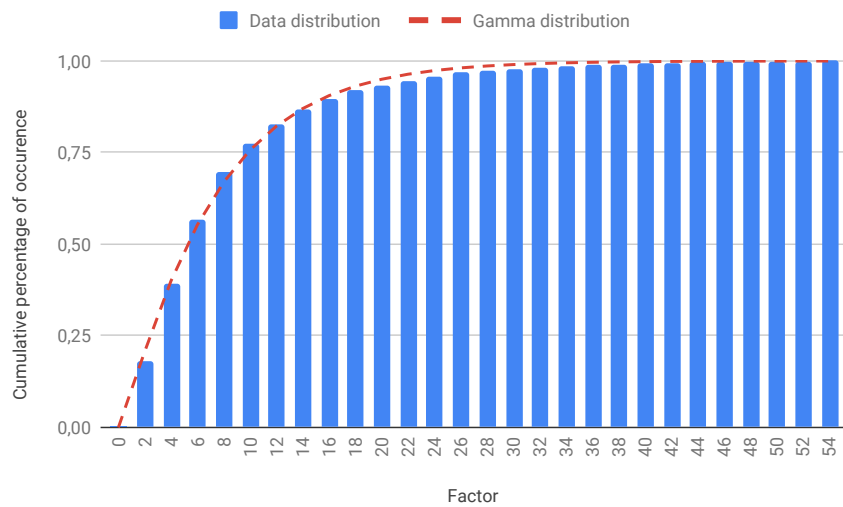
Factor	Occurences	Chance according to data	Cummulative chance according to data	Cummulative change accorcng to gamma
0	33934	0.00	0.00	0.00
2	22159	0.18	0.18	0.21
4	26117	0.21	0.39	0.40
6	21749	0.18	0.57	0.55
8	15980	0.13	0.70	0.67
10	9568	0.08	0.77	0.76
12	6638	0.05	0.83	0.82
14	4777	0.04	0.87	0.87
16	3689	0.03	0.90	0.91
18	2881	0.02	0.92	0.93
20	1624	0.01	0.93	0.95
22	1740	0.01	0.95	0.96
24	1455	0.01	0.96	0.97
26	1124	0.01	0.97	0.98
28	840	0.01	0.97	0.99
30	480	0.00	0.98	0.99
32	616	0.00	0.98	0.99
34	420	0.00	0.99	0.99
36	361	0.00	0.99	1.00
38	208	0.00	0.99	1.00
40	190	0.00	0.99	1.00
42	238	0.00	0.99	1.00
44	208	0.00	1.00	1.00
46	119	0.00	1.00	1.00
48	114	0.00	1.00	1.00
50	94	0.00	1.00	1.00
52	61	0.00	1.00	1.00
54	82	0.00	1.00	1.00



(a) The cumulative data distribution



(b) First try to find theoretical distribution



(c) The final distribution on top of the data

Figure 6.2: Distributions of the bidding price

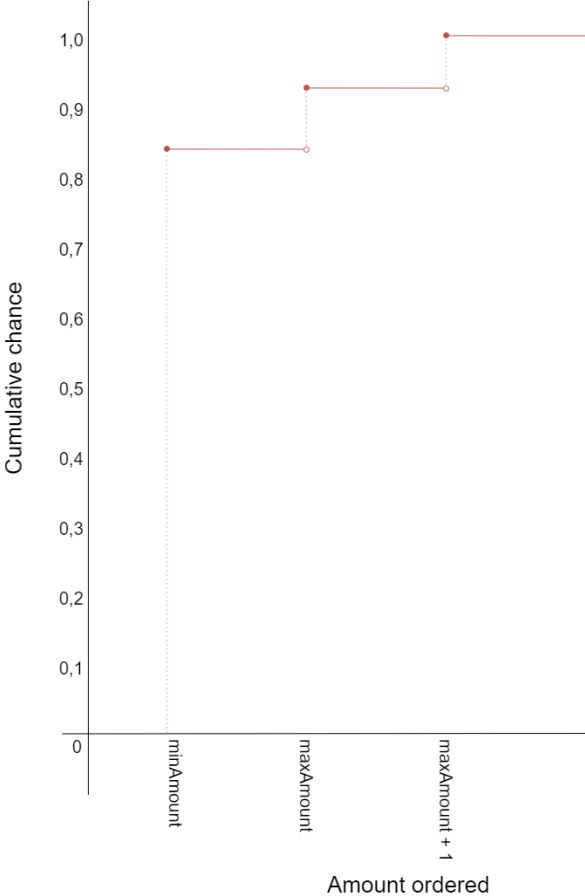


Figure 6.3: Distribution of the ordered quantity

Used data

For this distribution, the columns *auction number*, *minimum quantity*, *available quantity* and *ordered quantity* of dataset 1 with $n = 2420$ samples were used. The ordered amount may be less than the minimum quantity or more than the available quantity. However, the quantity the buyer actually receives is capped according to formula 6.6. Furthermore, we know that the available quantity is at least the minimum quantity.

$$bought\ quantity = \max(\max(minimum, ordered), \min(available, ordered)) \quad (6.6)$$

Data processing approach

For each sample, we determined whether the buyer ordered less than the minimum quantity, precisely the minimum quantity, exactly the available quantity or more than the available quantity. We then concluded that a product was sold 6.16 times on average by dividing the number of transactions that had an ordered quantity equal or greater than the available amount by the total number of transactions (see equation 6.7).

$$\frac{\# \text{ ordered quantity} \geq \text{ available quantity}}{\# \text{ transactions}} \quad (6.7)$$

For us to have an average of 6.16 transactions per product, it should only occur once in 6.16 times that a buyer orders the available amount or more. Approximately 8.44 per cent of the time, buyers tried to order more than available. Another 7.99 per cent of the time, the buyer bought precisely the maximum amount. Together, a buyer bought a quantity that resulted in a sold out product 16.43 per cent of the time. This process can be seen as a geometric distribution [23]: on successful trial, the product is sold out. With a 16.43 per cent chance, we sell out each time. The geometric distribution tells us that we need a total of $1/p$ trials, where the last one is a successful trial, and we have the probability p . When we fill in $p = 16.43$ we get $1/0.16 \approx 6.09$. So, to preserve this property of being sold 6.16 transactions on average per product, it would be approximately correct to order a quantity of the available amount and the available amount plus one, about 7.99 and 8.44 per cent of the time respectively. In fig. 6.3, we see these two jumps.

In 17.26 per cent of the time, a buyer ordered less than the minimum quantity. However, since the minimal quantity can be one, setting it to the minimal quantity minus one would cause a fault in the system about 17.26 per cent of the time. That is why we fill up the rest of the cases with the minimal quantity: This way, we do not initiate a special case of ordering less than the minimum amount nor do we sell out on accident, causing the average number of transactions per product to drop.

6.2.3. Distribution of the increase price**Found distribution**

The found distribution is dependent on both $x \sim U(0, 1)$ and the paid price *paidPrice*:

$$increasePrice(x, paidPrice) = paidPrice \cdot (factor(x) + 1) \quad (6.8)$$

With the factor:

$$factor(x) = \begin{cases} \text{gamma}^{-1}(1.2862, 0.7252) & x \in [0, 1] \\ 0 & \text{else} \end{cases} \quad (6.9)$$

Used data

For this distribution, the columns *increase position* and *bidding price* of dataset 1 (see appendix D.1) were used. As mentioned in section 6.2.1, we will only look at relative prices for the same reasons.

Data processing approach

From the increase position and bidding price, we calculated the difference as a factor with eq. (6.10). Here, we calculate the additional percentage of the bidding price relative to the increase position.

$$factor = \frac{increase\ position}{bidding\ price} - 1 \quad (6.10)$$

In table 6.2, we have grouped the factor in bins with size 0.2 and counted the occurrences. Then we plotted the cumulative chances in fig. 6.4a and saw it resembled a gamma distribution. To find out the right parameters for the gamma distribution, we first laid an gamma distribution with $\alpha = 2$ and $\beta = 1$ on top in fig. 6.4b. After this, we calculated the sum of standard errors (SSE) with formula 6.4. After this, we used Microsoft's Excel add-in program 'solver' to minimize the SSE [22]. This gave us the parameters $\alpha = 1.2862$ and $\beta = 0.7252$ with a SSE of 210.95 and an absolute error of 1.63% on average, shown in fig. 6.4c.

6.2.4. Distribution of simultaneous buyers**Found distribution**

The found distribution is dependent on both $x \sim U(0, 1)$ and optional parameter *watchers*. *Watchers* defaults to 122, the average number of watchers in the data we have analysed.

$$coBuyers(x, watchers) = factor(x) \cdot \frac{watchers}{122} \quad (6.11)$$

With the factor:

$$factor(x) = \begin{cases} 0 & x < 0.5227 \\ 1 & 0.5227 < x < 0.8021 \\ 2 & 0.8021 < x < 0.9223 \\ 3 & 0.9223 < x < 0.9636 \\ 4 & x > 0.9636 \end{cases} \quad (6.12)$$

Used data

For this distribution, the columns *watchers* and *co-buyers* of dataset 1 were used.

Data processing approach

One assumption one can make is that there is a correlation between the number of watchers and co-buyers. To research this, we compared the individual distribution of the number of co-buyers of several sections of watchers in table 6.3. The last row gives an overview of the distribution of all the data.

Table 6.3 shows that the number of watchers does not have a significant influence on the number of co-buyers: every row, which represents a bin of watchers, has roughly the same distribution of co-buyers. Also, as we have more watchers in lower rows, we do not see an increase in percentages of the higher number of co-buyers. Therefore, we will use the overall data distribution since this is easier implemented.

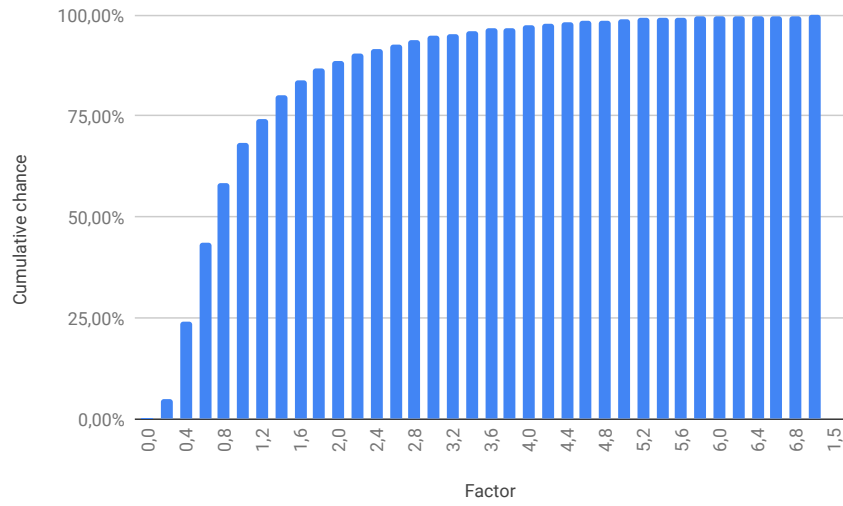
However, since we might want to stress test the system, we will also allow a scale-up factor. With an optional parameter of the number of watchers, one can scale up the number of co-buyers with respect to the average number of watchers in the data (122).

6.3. Scheduler

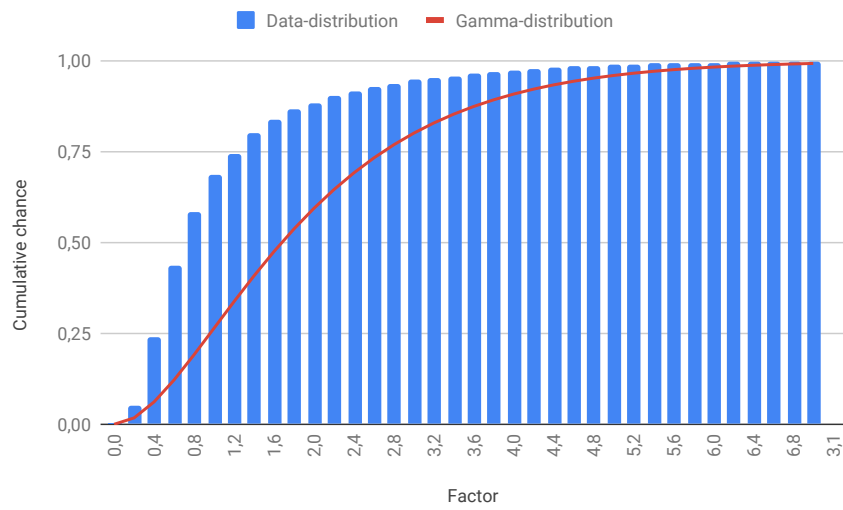
The scheduler is used by the orchestrator to generate behaviour for both the auctioneer and buyer clients. The total number of buyer clients that will participate in the simulation can be specified in advance. The behaviour of a buyer client is defined simply by a set of bids that it should make. Each bid has the following attributes:

Bin	Occurences	Chance according to data	Cumulative chance according to data	Cumulative chance according to gamma
0.0	0	0.00%	0.00%	0,00%
0.2	122	5.04%	5.04%	1,75%
0.4	458	18.93%	23.97%	6,16%
0.6	476	19.67%	43.64%	12,19%
0.8	360	14.88%	58.51%	19,12%
1.0	242	10.00%	68.51%	26,42%
1.2	142	5.87%	74.38%	33,74%
1.4	137	5.66%	80.04%	40,82%
1.6	90	3.72%	83.76%	47,51%
1.8	74	3.06%	86.82%	53,72%
2.0	42	1.74%	88.55%	59,40%
2.2	44	1.82%	90.37%	64,54%
2.4	33	1.36%	91.74%	69,16%
2.6	26	1.07%	92.81%	73,26%
2.8	26	1.07%	93.88%	76,89%
3.0	23	0.95%	94.83%	80,09%
3.2	9	0.37%	95.21%	82,88%
3.4	16	0.66%	95.87%	85,32%
3.6	17	0.70%	96.57%	87,43%
3.8	8	0.33%	96.90%	89,26%
4.0	15	0.62%	97.52%	90,84%
4.2	10	0.41%	97.93%	92,20%
4.4	8	0.33%	98.26%	93,37%
4.6	4	0.17%	98.43%	94,37%
4.8	7	0.29%	98.72%	95,23%
5.0	5	0.21%	98.93%	95,96%
5.2	6	0.25%	99.17%	96,58%
5.4	5	0.21%	99.38%	97,11%
5.6	2	0.08%	99.46%	97,56%
5.8	2	0.08%	99.55%	97,94%
6.0	2	0.08%	99.63%	98,26%
6.2	4	0.17%	99.79%	98,54%
6.4	1	0.04%	99.83%	98,77%
6.6	1	0.04%	99.88%	98,97%
6.8	0	0.00%	99.88%	99,13%
7.0	3	0.12%	100.00%	99,27%

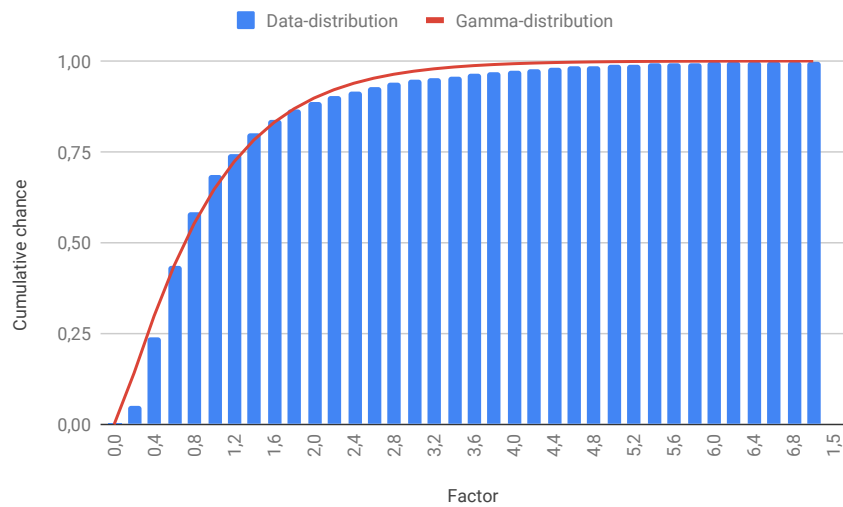
Table 6.2: Bins of the factor of increase position and its properties



(a) The cumulative data distribution



(b) First try to find theoretical distribution



(c) The final distribution on top of the data

Figure 6.4: Distributions of the increase position factor

Watchers	Co-buyers					Samples
	0	1	2	3	4	
<86	54%	13%	29%	4%	0%	24
86-90	36%	36%	0%	14%	14%	14
91-95	59%	41%	0%	0%	0%	17
96-100	45%	20%	30%	5%	0%	20
101-105	60%	26%	9%	3%	3%	77
106-110	59%	26%	9%	2%	4%	179
111-115	63%	25%	9%	2%	1%	485
116-120	51%	30%	13%	3%	3%	450
121-125	53%	27%	11%	5%	4%	256
126-130	44%	34%	12%	6%	3%	243
131-135	44%	28%	16%	5%	8%	226
136-140	42%	34%	12%	7%	6%	179
141-145	48%	23%	14%	10%	5%	125
146-150	51%	28%	18%	3%	0%	68
151-155	46%	23%	21%	8%	3%	39
>156	61%	22%	6%	6%	6%	18
All data	52,27%	27,93%	12,02%	4,13%	3,64%	2420

Table 6.3: Bins of number of watcher and the respective occurrences of number of co-buyers

BID	
auction item ID	The ID of the auction item on which to place the bid.
bidding session number	Indicates after how many rounds of bidding (on the specified auction item) the bid should be placed.
price	The price point the buyer aims to place a bid at.
quantity	The quantity the buyer should buy.
reaction time	The time (in ms) between the clock reaching the desired price and the client making a bid. The winning bid should have the lowest reaction time.
winner	Can be either true or false. If true, this bid should be a winning bid.

A set of actions defines the behaviour of the auctioneer client. The point at which to execute these actions is defined by the auction item ID and bidding session number attributes. The scheduler produces two types of actions. The first and most prevalent action is the INCREASE action, which tells the auctioneer client to increase the clock to a particular price. This action has the following attributes:

INCREASE	
auction item ID	The ID of the auction item that is being sold when this action should be executed.
bidding session number	Indicates the bidding session at the start of which this action should be executed.
price	The price to increase the clock to.

The second type of action produced by the scheduler is SET STEP SIZE. This action tells the auctioneer client to set the step size of the clock to a particular value. It has these attributes:

SET STEP SIZE	
auction item ID	The ID of the auction item that is being sold when this action should be executed.
bidding session number	Indicates the bidding session at the start of which this action should be executed.
step size	The step size to set.

The bids and auctioneer actions are chosen based on a real auction schedule from the Clock API. For each auction item, the scheduler draws several values from the *bought quantity* distribution, such that the sum of these quantities is exactly equal to the total quantity that is for sale. These quantities specify how many units of the current auction item are bought during each bidding session. For each bidding session, the scheduler draws a number N from the *simultaneous buyers distribution* and a bidding price from the *bidding price* distribution. Then it randomly picks N unique clients for which it generates BID objects with the aforementioned price and quantity. One of the N bids is chosen to be the winning bid. Its `winner` attribute is set to true and the scheduler ensures that its `reaction time` is the lowest of all N bids.

For each bidding session, an INCREASE action is generated, drawing a value for the `price` attribute from the *increase price* distribution. The scheduler also guarantees that the number of clock rounds never exceeds two rounds during the increase clock action, by adding appropriate SET STEP SIZE actions where necessary. For example, imagine the current clock price is 87 cents and the step size is 1 cent. If the value to increase the clock to during the next bidding session is high, for example, 515 cents, 5 rounds would be needed with the current step size of 1 cent. Instead, the scheduler adds a SET STEP SIZE action to the auctioneer behaviour, setting the step size to 5 cents. With this step size, increasing the clock to 515 cents requires only 1 clock round.

Using the information in the schedule (bids and auctioneer actions) generated by the scheduler, a simulation can be performed that is based on a real auction schedule and statistics from real buyer and auctioneer behaviour. The `winner` attribute of the bids allows us to compute the percentage of bids which ought to win according to the schedule, that actually win during the simulation. This is a valuable metric which can help assess the trustworthiness and usefulness of the Digital Clock system.

6.4. Results

The simulation outputs a JSON file, as seen in listing E.1. The simulation collects lots of data, but we have filtered it to data concerning pings. In our data, we consider a single ping to be half the round-trip time. This time is obtained by sending a single call to the RTCS, the simulated client is connected to, and measuring the time it takes to get an answer. As mentioned in section 6.1.2, only the 'first come, first serve' strategy is taken into account. Therefore, if the bid that was supposed to win according to the schedule does not win, this can have two causes: for one, the simulation system could be at fault, either due to failing buyer clients or scheduling problems. The second cause could be that the digital clock system itself is performing inadequately, which we will investigate here by analyzing the ping data.

From the data, several aspects related to the pings were calculated. Before we can say anything about the results, we should know which results are acceptable and which are not. Royal FloraHolland, considers a connection with a ping of 30 milliseconds or less to be a good connection. This is necessary, because in their configuration for clocks on one location, the clock updates every 37 milliseconds. To be on the safe side, we will consider their 30 milliseconds as leading for a price change. To be able to say something about the variance, we will take 15 milliseconds as a threshold. If the maximum variance is higher, it will be considered a bad connection. Although this is an arbitrary number, we do have some reasoning behind it: With a variance of 15 milliseconds, we know for sure that you do not end up in the interval. For instance, a buyer places a bid that was meant to arrive in the third millisecond of the interval x . With a variance of 15 milliseconds, it could now arrive in the interval $x - 1$. The only edge case is when a message was meant to arrive at precisely the fifteenth second, but this chance is arbitrarily small. We are will thus be looking for a maximum average ping of 15 milliseconds (30

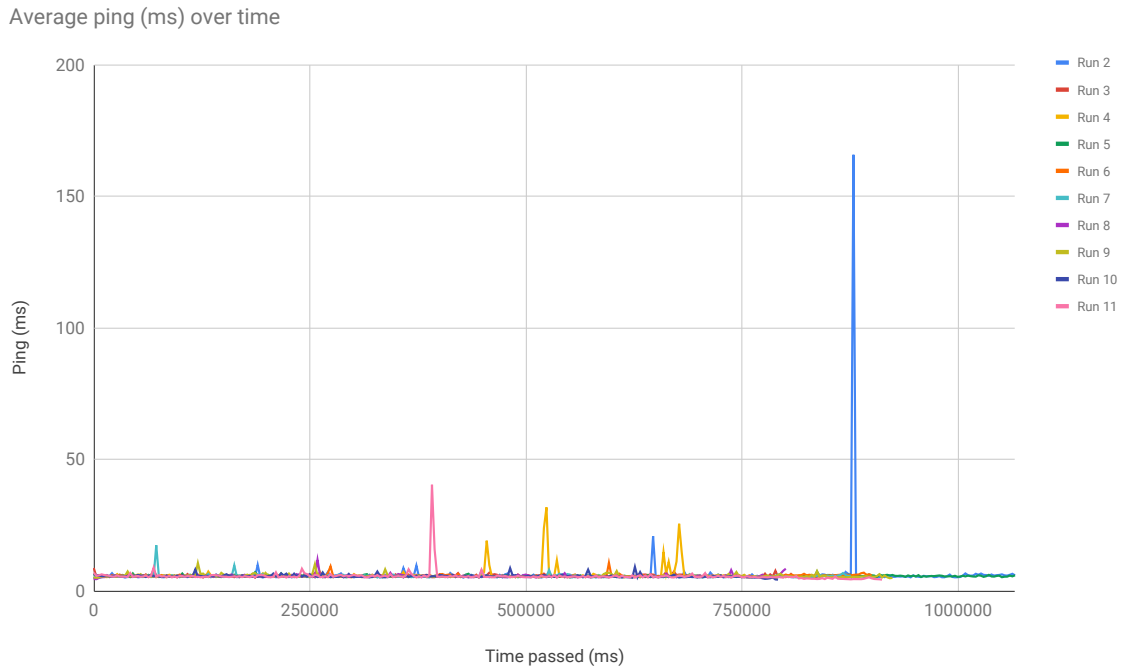


Figure 6.5: Average ping (ms) over time

milliseconds max, minus 15 milliseconds variance) and a maximal variance of 15 milliseconds.

We first looked at the average ping over time. For this, we grouped all pings in time windows such that they were not more than 3000 milliseconds apart. This number was chosen based on the frequency of the client pings (see [section 6.1.2](#)). We can thus ensure that no client has more than one ping in each bin. The average ping over time is plotted in [fig. 6.5](#). There is a clear baseline, centered around 5 to 6 milliseconds, which is below the 15-millisecond threshold. However, there are five peaks that do not comply with this threshold.

Another useful graph is the average ping per client, shown in [fig. 6.6](#). All the averages, with the exception of run 2 and 4, seem to concentrate around a ping of 2 to 4 milliseconds. Furthermore, all averages are below the 15-millisecond threshold.

To see if our system is reliable, we also look at the standard deviation (STDev) and the median absolute deviation (MAD) [24] of the client pings. MAD, unlike STDev, is not as sensitive to outliers and helps us get a complete picture of the variance in the ping.

Although we have too little data samples per client, namely 10, to know if they are normally distributed, we will work them out if they were, to have a feeling for the numbers. The chances this will produce, however, are not necessarily applicable to the data.

In [fig. 6.7](#), we can see the STDev per client. Mostly, the points fall between 0 and 30. When we exclude run 2, almost all points fall between 0 and 15 milliseconds.

87.6% of the points lay on or below 3, which means that they have a standard deviation of 3 milliseconds. A maximum variance of 15 milliseconds is thus equal to 5 standard deviations. The chance of not falling in the interval $[\bar{X} - x\sigma, \bar{X} + x\sigma]$, with mean \bar{X} and number of standard deviations x , can be calculated with the well known function [eq. \(6.13\)](#). Thus, the chance of not falling in this interval is $5.73E - 5$ per cent. In 95,8 per cent of the time, we have a standard deviation of 10 milliseconds or less. A maximum variance of 15 milliseconds is thus equal to 1.5 standard deviations. The chance of not falling in this interval is 13.4 per cent. Together, this gives us $0.876 \cdot 5.73 \cdot 10^{-5} + (0.958 - 0.876) \cdot 0.134 = 0.0110$. There is thus an about 1.15 per cent chance that we have a variance that is too high if this data were normally distributed. When the data would be more uniform, this would probably be higher.

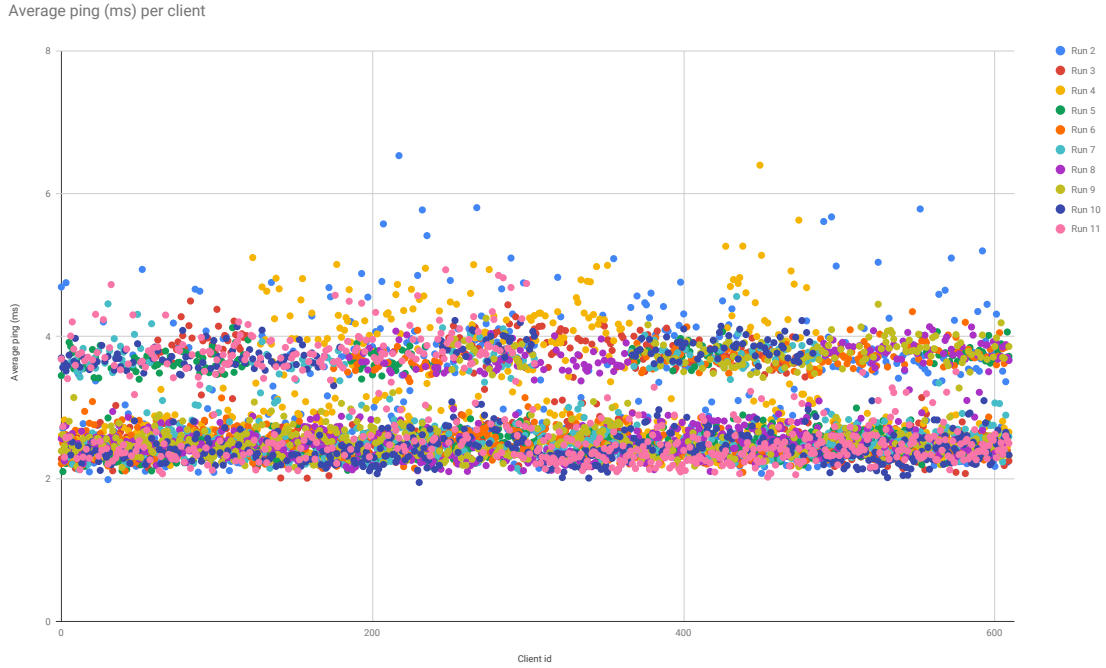


Figure 6.6: Average ping (ms) per client

$$chance = \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right) \quad (6.13)$$

where:

erf = The well known error function
 x = Number of standard deviations

For each client, we have plotted the MAD in [fig. 6.8](#) according to [eq. \(6.14\)](#). Since they are less prone to outliers, we can immediately see that the deviation is very low. Not taking into account 10 out of 6100 samples, the maximum value is 1.5. If this data were normally distributed, this means that we can handle 10 times this deviation, before it becomes problematic. [\[25\]](#) shows that the probable error, which is equivalent to MAD, can be calculated according to [eq. \(6.16\)](#) with reverse [eq. \(6.17\)](#). This last formula gives us $\sigma = 2.23$. A maximum variance of 15 milliseconds is thus equal to 6.745 standard deviations. The chance of not falling in this interval is $1.53E - 11$ per cent.

$$MAD = \operatorname{Median}(|X_i - \tilde{X}|) \quad (6.14)$$

with

$$\tilde{X} = \operatorname{Median}(X) \quad (6.15)$$

$$\text{Probable error} = 0.6745 \cdot \sigma \quad (6.16)$$

$$\sigma = \frac{\text{probable error}}{0.6745} = \frac{MAD}{0.6745} \quad (6.17)$$

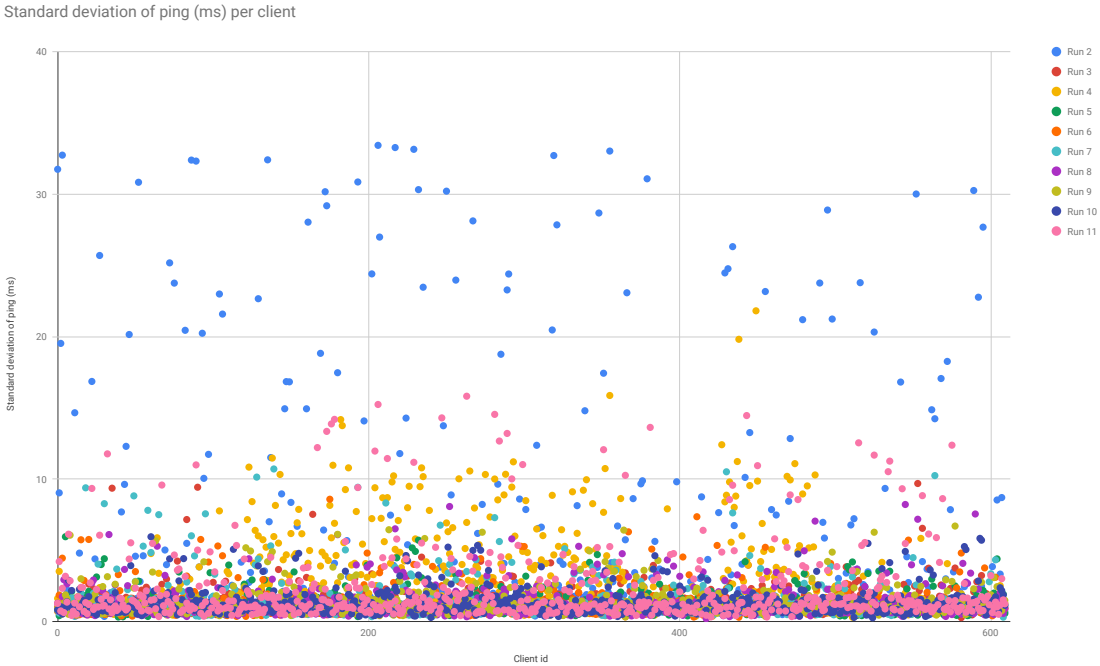


Figure 6.7: Standard deviation of ping per client

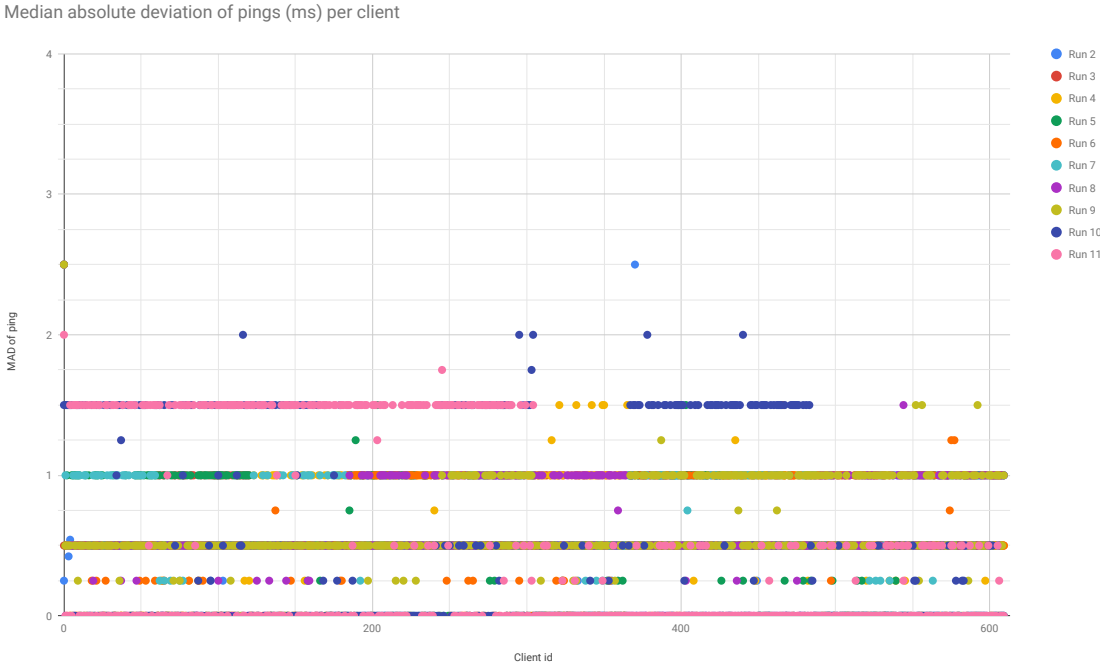


Figure 6.8: Median absolute deviation of ping per client

7

Evaluation

In this chapter, we will evaluate the results of this project. We will first discuss the process around the project in [section 7.1](#). We will then examine how the product turned out in [section 7.2](#). Finally, in [section 7.3](#) we will then review the performance results found in [section 6.4](#).

7.1. Process

In this section, we will evaluate our process of developing the software. We will describe our workflow in [section 7.1.1](#). [Section 7.1.2](#) describes our interaction with the clients.

7.1.1. Workflow

During the project, we made use of an Agile methodology [26]. This meant that we tracked our progress using a Scrum board on Trello [27], worked using weekly sprints and did daily stand ups. We used Azure DevOps [28] as our source control platform, which offered continuous integration services and neatly integrated with our continuous deployment to Azure services [29]. Furthermore, we worked using pull-based development with feature branches. Merging branches to the *master* branches required the continuous integration pipeline to pass successfully and at least one positive code review of a member that has not contributed to the branch.

Each team member was responsible for a different part of the project. The exact responsibilities are described in [appendix H](#). However, having these specific responsibilities does not mean that there was no discussion about the design and implementation details with other team members or that other team members never contributed to those parts of the code base.

7.1.2. Client interaction

In principle, we were present at the JEM-id office forty hours a week. However, the group members had the liberty to work remotely from their homes. Most time was still spent at the office to maximize client interaction.

During the project, we had a multitude of meetings with employees of the client. Additionally, in the early stages of the project, we had numerous meetings with employees of Royal FloraHolland regarding the current auction process. Later on during the project, we have given demonstrations of the progress of the project to various JEM-id and Royal FloraHolland stakeholders. Halfway during the project, there was a meeting with both our TU Delft supervisor and JEM-id supervisors to demonstrate and discuss the progress and future expectations of the project. During the aforementioned meetings and presentations, the feedback has generally been very positive.

7.2. Product

In this section, we will evaluate the project. We will discuss what features and requirements did not make it into the final product, as well as features and requirements that were originally not planned for

inclusion, but were included after all. We will also talk about the satisfaction of our client, JEM-id, and our client's client, Royal FloraHolland. Finally, we will examine the code quality in [section 7.2.3](#).

7.2.1. Features and requirements

We have strived to implement all functionalities and requirements that were decided on before we started programming. Despite this, we have chosen not to implement some features, and we have also decided to add some that were not listed. We will discuss these changes here.

We are happy to report that we have implemented nearly all functional requirements that were prioritized as *Must have* and *Should have*. One *Should have* has not been implemented, and it is the one asking us to store transactions in a central system. Since this central system does not have any support for storing transactions, we have chosen to ignore this feature. Moving to features with a lower priority, we find that the following features have not been implemented (see [section 4.1](#) for a full reference):

- Standard auctioneer messages: we have discussed this idea with Royal FloraHolland, and they considered a chat function to be a bad idea. As such, we have chosen to omit the feature fully.
- Numerical view: we have kept the default clock layout that is used in the current system. A numerical would not add much to our feasibility study and would be trivial to implement. Therefore, we have not included it.
- Alarm functionality: due to time constraints, we have not been able to add an alarm system that alerts users when an item they selected is about to be auctioned off.
- Spectator mode: Although users can always view the auction, we have not built a switch that disables all buying options (and thus enables spectator mode).

All functional requirements that were considered *Won't have* have not been included, although one of these requirements might be debatable. We have stated that we would not be building an artificial intelligence system that is able to control auctions automatically. However, we have built a simulation engine that is able to use real-world data to simulate an auctioneer in a realistic fashion. So although we do not incorporate any artificial intelligence in this system, we have built software that can control this auction system in a fully automated manner.

Similarly, we have achieved all non-functional requirements that were set for this project and were categorised as *Must have* or *Should have*. For example, we have achieved fairness for all buyers by implementing several arbitration algorithms that pick one or more winners from a list of bids. The clock administrator, which will in practice be Royal FloraHolland, can then choose the algorithm they would like to use. A slightly debatable requirement is the one dictating that the system should have high availability. Although we have built auto-reconnect features when a connection is lost, for example, we are still reliant on cloud computing providers, meaning that this is somewhat outside of our control.

Furthermore we have also fulfilled some *Could have* non-functional requirements. The first one is support for Docker so that the application can be easily deployed across all platforms that support Docker. Although we have not implemented ourselves, this is relatively easy to do using an official tutorial [30]. The second one is documentation enforced by automatic tooling, which we have used from the start of the project.

7.2.2. Client satisfaction

During the project, we have had a number of feedback moments where we have conferred with at least one of our supervisors to discuss our progress on the project. In this section, we will describe our general impressions.

In the fifth week of the project, we had a midterm meeting with our supervisor from TU Delft, Dr. A. Panichella. During the meeting, we have demoed our system and described some of our choices. Overall, he found that we were "clearly in the green", indicating that we were well on track with our project.

The week after, week six, we had another meeting with some technical staff from Royal FloraHolland. Although we originally went to learn more about the online clock system currently in use, we ended up

Metric	Score CA	Score RTCS
Code Smells	0 issues	0 issues
Line Coverage	92.8%	73.9%
Lines of code	1887	660
Reliability	0 issues	0 issues
Security	0 issues	0 issues
Duplications	0 issues	0 issues

Table 7.1: SonarQube results.

demoing our progress to them. Their overall impression was positive, with them mostly being impressed that the system worked as well as it did at the time.

A few days after that meeting, we had a more formal presentation where we were allowed to demo our system to Royal FloraHolland employees. This presentation was more focused on the business needs of the application and less so on the technical side of the project. As such, we mostly discussed features and other capabilities rather than technical details. Although no judgement of the application was explicitly given they did start discussing where they would be testing the application, and remarked how it might be integrated with current systems. This leads us to assume they were happy with the system, and consider it a viable option to use in the future.

7.2.3. Code quality

We regard code quality as an important part of software development. Thus we have taken special care to use design patterns, such as adapters, factories, and builders when applicable. Additionally, we have used various static analysis tools to maintain a uniform code style and detect code smells. Besides these tools, we have used SonarQube [31] to keep track of our code quality in our different backend repositories. The results found by SonarQube are given in [table 7.1](#). These results require some clarification. The relatively low line coverage score of the RTCS can be attributed to the fact that the Startup class used by ASP.NET Core is relatively hard to test, since it sets up the entire web server in a few function calls, given some arguments that are difficult to mock. Furthermore, it must be noted that a low number of issues does not guarantee bug-free code.

Furthermore, halfway during the project, we received feedback on our code maintainability from the Software Improvement Group (SIG) [32]. SIG rates the maintainability based on other code projects they store in their database. Our project has received an above average score. We tried tackling the issues they raised. However, due to the low severity and time constraints, we were not able to resolve all of them. The full feedback is provided in Dutch in [appendix G](#).

7.3. Performance

During the development phase of our software, we have often experimented with our system to see if it is able to perform basic functionalities, such as increasing the clock, starting the clock, placing bids, and more common actions that are repeated in quick succession. After we had optimized aspects of both the back-end and front-end systems, we felt that the performance was fairly good, and the system felt responsive in general. Of course, such feelings cannot be adequately quantified, and to prove that a system works and is performant, it needs to be tested.

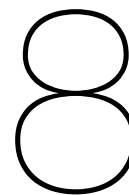
We have run several tests on our system. Early on, we decided that in order to test our system thoroughly, we would need to simulate the auction process, to ensure that the system can handle many users and that it still operates correctly. The methodology and results of these tests are described in [chapter 6](#). In this section, we will aim to contextualise these results and elaborate on them.

We have run our simulation eleven times in total. The first run has been omitted from the results, as it was a test run to verify that all systems were able to communicate as they should. Subsequent runs are included with the results. A first note we would like to make is that the second and fourth run (*run 2* and *run 4*) are often a bit of an outlier in the ping graphs displayed in [section 6.4](#). While we cannot give any reason with certainty, we assume that this run was a bit flakier because all the services were started again in these runs. This initialisation process was already done in the other runs, possibly

speeding it up. Furthermore, for the second run, one of the test devices was connected over a wireless connection. In later runs, this device was connected to an Ethernet cable, potentially allowing it to use a faster network connection. This might explain why the second run is even flakier than the fourth one. Further discussion of the results gathered will take place in [chapter 9](#).

When looking at the average ping over time graph ([fig. 6.5](#)), we find that the average ping time is almost always below 15 milliseconds. Only 0.0820 per cent of the time, it was higher (which corresponds to the five peaks the section talks about). As per the definition set by Royal FloraHolland, the clients have a fast connection. Although the graph does show some spikes, we still consider this result to be decent in general. Since these spikes only last for a very small amount of time, the chance that users will notice them is very slim. The results found in [section 6.4](#) support this thought.

Firstly, every client has an average ping of fewer than 8 milliseconds, which is far below the self-appointed threshold of 15 milliseconds. Secondly, in 87.6 and 95.8 per cent of the time, the system can handle up to 5 and 1.5 standard deviations, respectively. In many distributions, this will map to high chances of falling into this interval of deviations. In most cases, the standard deviation holds up to our self-appointed threshold. Thirdly, the median of absolute deviations is noticeably low. Only 0.164 of the time, it is higher than 1.5 (10 out of 6100 samples). This would mean that our system is reliable and consistent almost all of the time when we are within 10 MAD's of the average. The chance of not being in this interval is almost irrelevant in the case of normally distributed data. We can thus imagine this being the case in other data as well. Therefore, we believe our simulation results are mostly accurate.



Ethics

In this chapter, we will describe some ethical issues that we have encountered during our project. In [section 8.1](#) we describe the possible implication the proposed auction system can have on the jobs of people working at the auction. We continue by discussing the financial aspects in [section 8.2](#). Finally, [section 8.3](#) describes how we obtained and handled the data we used.

8.1. Jobs

Our project potentially provides Royal FloraHolland with a new auction system. One of the design goals we set for the project was that it would be easy to extend the project with new functionalities. This means that, in the future, an automated system could be created that is able to control the auction, essentially taking over the job of the auctioneer.

In fact, we have already created such a system able to control the auction as part of our simulation component. We have created an auctioneer client that is able to increase the clock to a specific price, starts it and then waits for a client to place a bid. Although this is a relatively simple bot (it follows a generated schedule), it does lay the foundation for future improvements, that could, in the end, fully automate an auctioneer's job.

Similar reasoning can be applied to buyers. Some large buyer organisations have workers dedicated to buying the correct products at the right price. We have repeatedly heard from Royal FloraHolland staff that experienced buyers tend to perform better than those who are relatively new to the scene. This means that employing the right people can be a great advantage to bigger organisations. Should they decide to create a bot similar to the buyer clients we have created for our simulations, these dedicated buyers could potentially find themselves without a job.

8.2. Financial aspects

The auction deals with large sums of transactions on a daily basis. Therefore it is crucial that all transactions made by the system are correct. This means that special care was taken to ensure that there are no possibilities for invalid transactions (e.g., due to rounding errors with floating point numbers) that could potentially harm companies financially.

Additionally, it is important that every buyer has an equally fair experience. However, due to external factors such as network delay, this is not always possible. Because of this, we have implemented multiple arbitration algorithms that can select winning bids based on different fairness constraints. Furthermore, we have made sure that this system is easily extensible if the fairness criteria change in the future.

8.3. Data

The data used for the simulation were given by Royal FloraHolland. The data contained sensitive information, such as client identification numbers. All sensitive information has been filtered out, rendering it completely anonymous. We have only used the data we needed. Furthermore, we decoupled data which did not need to be coupled. With all of these precautions, we ensured the privacy of the users of Royal FloraHolland and the confidentiality they have entrusted us.

9

Discussion and recommendations

This chapter provides a discussion on our findings during the project. First, we will discuss the clock system in [section 9.1](#). Then, we will discuss our simulation results in [section 9.2](#).

9.1. The clock system

For the clock, we believe we have made the right decision of language and framework. The choice of ASP.NET allowed us to integrate with the existing services quickly and painlessly. We believe using SignalR core for WebSocket communication was also the right decision. However, we had a harder time with SignalR because of its APIs. The reason for this is that its core-version APIs are still in development and do not allow the code to be as neat as we would like it to be, either requiring one to register callback functions or use reflection.

During architecture design, we quickly noticed that RTCS services would not do much more other than message passing in our application. In the future, one may require RTCS services to keep their own beliefs about the state of the clock, such that they may be queried for it, relieving the clock service. We also found, during the development phase, that such a message passing service already exists in the Azure cloud. Azure SignalR scales SignalR WebSocket connections using only one app service application [33]. In the end, we left space for using Azure SignalR instead of RTCS services if that were to be preferred.

9.2. Simulation

As mentioned earlier in [section 7.3](#), we are satisfied with the results found by our simulation. However, the performed simulations have some limitations that may have affected the outcomes.

Firstly, we were not able to simulate every possible configuration of the auction while doing our simulations. For example, we have only used the *first* algorithm as described in [section 5.2](#). This means that we are uncertain of the performance for different winner selection algorithms. Additionally, we have only performed the simulation for an auction with fifty items, while real auctions can have up to hundreds of different items. Thus, we can not be certain that the found performance is maintained if the auction is prolonged.

Furthermore, we were only able to host all different services on a single Azure plan. This meant that all services shared the same resources, which could have affected the found performance. Additionally, all 610 headless clients were hosted on four different computers, connected from two different networks. This is a highly unrealistic situation since generally, one computer will most likely have one connected client. These 610 clients were split over only 2 RTCS instances sharing the same resources (as was mentioned earlier). We suspect that it is possible to obtain better results when using more RTCS instances that do not share the same resources.

Moreover, we have made the assumptions that the results were normally distributed when analyzing them. However, the sample size was not large enough to be fully certain that these results were

normally distributed.

Finally, we think it can be useful to perform more simulations in the future. These simulations can have different settings and the different applications used in the entire system should ideally be spread out over more different systems for a more realistic situation.

10

Conclusion

At the start of this project, as instructed by JEM-id, we set out to show that a web-based version of the auction clock system that is currently in use by Royal FloraHolland is feasible. We aimed to show that with modern software frameworks and tools, such an application can be built within seven weeks, while being easily extendable for future work. Furthermore, we intended to prove that this application can handle a similar, if not higher load than the current auction system employed by Royal FloraHolland.

Indeed, our final product is an in-browser auction application that has all the functionalities that are critical to running a digital Dutch auction. It allows users to place bids on auction items, select desired quantities, view essential product information and keep track of transactions made. This is all done within a GUI that resembles the original auction system to ensure user familiarity, while also being updated to fit modern design standards and tooling.

A separate view within the browser is used to control the auction process. In this view, the auctioneer can set up all the controls and hotkeys as they desire. Our application facilitates auctioneer actions to control the auction clock, as well as the ability to skip auction items and pause the auction process.

As our system was built using modern frameworks and was designed to be flexible and easily extendable, we think it should be easy to add new features. We think our own pace of development is in support of this claim.

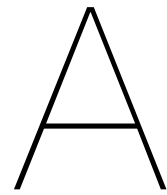
Behind the scenes, we built our system to be scalable with respect to the number of connected users. This is achieved by distributing the load on the auction clock server over intermediary communicators called Real Time Connection Services (RCTSs). The number of RCTSs is variable to allow more instances to be deployed if need be.

We created a simulation scheme with which we performed load testing on our auction system, using autonomous buyer and auctioneer clients. We recorded ping data from these autonomous buyers. Upon careful inspection, we conclude that our product can easily handle a load five times that of a typical auction. The clients had an average ping lower than 30 ms, which shows that a great number of fast connections can exist within our system simultaneously. Based on the results we have obtained through our simulations, we find that the connections are stable enough for participating in the auction in a competitive setting.

By allowing an auction administrator to set the winner selection algorithm, we offer different kinds of fairness. For example, they can choose that the first bid that comes in will always be selected as the winner. Another option is that the bid with the highest price will win. Bids are then accumulated for the duration of the grace period, but this means that this grace period must account for network speeds of remote users as well. Since each method appears to have advantages and drawbacks, we have chosen not to make this decision.

However, our simulation had certain limitations. Most autonomous clients were deployed from within household networks, risking delays as large streams of data had to go through a single point of congestion (the local router). Some simulation runs showed one or more simultaneous spikes in ping for a considerable number of clients. We suggest that setting up a more realistic simulation environment could give insight into whether these spikes were due to local network congestion or the failure of the auction system.

To conclude, we think the project is a success. We have shown that it is possible to hold a fast-paced Dutch auction through a web-based application. Our system can easily handle typical loads and can accommodate even greater loads. Connections are fast and stable. Furthermore, we developed this product within a period of seven weeks, demonstrating the power of modern frameworks. Our system is also easy to build upon, but we advise our clients to perform more rigorous load-testing simulations before employing it in the real world.



Original problem statement

The content in this appendix is a direct copy of the BEP proposal originally listed on BEPSys (<https://bepsys.ewi.tudelft.nl>). Although the content remains identical, minor changes have been made to reflect the visual style of this document.

Creating an online auctioning clock

In the current situation, traders of plants and flowers can buy and sell their goods through the 'flower auction' of Royal FloraHolland. Nowadays, the flower auction is becoming less and less popular with traders, due to the fact that traders have to be physically present at the flower auction, at a set time. In this day and age where time is limited (time = money), traders tend to prefer online solutions over physical tradings. In recent years this has led to specific online solutions ('kopen op afstand') which however still have to conform to the boundaries (and legacy systems) that are enforced by the usage of this physical auction clock.

Given the current state of technology and the fact that one does not have to take into account any legacy code or tech-stack, how different would this be if a new online solution would be designed. New and yet unknown opportunities arise which can ultimately lead to a new concept that can be applied to horticulture's auctioning mechanisms. The most prominent technical challenges will lie in the areas of time-synchronisation algorithms of the real-time auctions while serving N clients (easily 1000+) from different countries, time-zones, using different devices, and so on. The solution needs to be highly performant and very easy to scale in order to serve the increasing and unpredictable number of concurrent clients.

The goal of this project is to develop a complete new concept for online auctioning clocks that implement the 'Dutch auction'. The concept should ultimately be capable of incrementally replacing the existing, physical, auctioning clocks at Royal FloraHolland. With a solid, performant and scalable solution in place, different new features can be developed such as the integration of different sorts of purchase-agents that can be deployed.

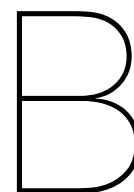
Challenges include:

- Time synchronisation between backend systems en multi-platform clients (i.e. desktop, mobile, ...)
- Performance and scalability are critical success factors
- Possibly different AI components when developing extra features.
- The floriculture sector is known for being slow to adapt to technology. The product therefore needs to be of high quality to convince people to use it.

Problems:

- The students must decide which tools and technology will be used.
- The students must design an architecture for the online auction clock.
- High performance must be ensured for all users.
- The solution must be able to scale easily, which will give problems due to the nature of distributed systems.
- Anyone should be able to use the online auction

Students can work on-site and will work with real data derived from the current auctioning clock processes at Royal FloraHolland. The project group will ultimately conduct a 'Proof of Concept' that will be presented to the CDO of Royal FloraHolland.



Product Plan

All content in this section is directly taken from our original product plan. Although the content remains identical, minor changes have been made to reflect the visual style of this document and to incorporate feedback from our supervisors.

B.1. Introduction

The Netherlands is the largest exporter of decorative flowers in the world [1]. This billion euro industry is powered by relations between transporters and growers. Traditionally, transactions between growers and transporters or exporters are set up by a third party through an auction. The auction we will be working to improve is the Dutch auction set up by Royal FloraHolland (RFH).

Royal FloraHolland has more than a hundred years of experience with Dutch auctions or clock auctions. The company has been improving its capacity, number of auction houses and efficiency significantly, allowing it to grow to the most successful of its kind. The main goal of the cooperation is to establish trade connections between growers and buyers of plants and flowers. To do so, RFH offers a number of services and software products to ease this process. The clock auction has been modernised by RFH ever so slightly over the years to help accomplish this. For instance, growers can now send a list of the products they want to put up for auction over internet and buyers may attend the auction, similarly, online.

However, even with modernising changes, the popularity of the traditional auction format has been in decline for the last decade. In 2018, the total clock trade share was 42% against direct trade [34]. In 2014, this share was nearly 49% [35]. The percentage of transactions originating from the clock is dropping slowly. The function of auction trade is becoming price setting rather than establishing transactions. Establishing transactions is, instead, done through direct trade with set prices. The reason for this shift is found in other modernisation tactics applied by RFH. It is becoming easier and cheaper for buyers to partner with trusted growers and set up direct transactions. The price for these transactions is determined by buying smaller amounts at the clock auction.

This brings us to our project. Our project is provided by JEM-id, a company developing software for the agricultural sector. JEM-id has partnered with RFH to develop a software platform where all kinds of services can be offered to growers and transporters called Floriday. We have been tasked with rebuilding the modernised clock such that it may be integrated into Floriday and, more importantly, such that it can support a growing world-wide list of buyers simultaneously. The challenge is to maintain the fairness and incredible speed that are associated with the auction in an online format, while being maintainable and modular, which the current clock is not.

B.2. Purpose

B.2.1. Research phase goals

During the research phase, we are interested in two main points. First off, we need to know what the stakeholders in this project want.

The second question is that we will have to build a realtime communication system. Some of these may exist, but we will have to look at the different features of these libraries.

B.2.2. Product goals

During the programming phase, we will attempt to realise the product. For this, we will mainly look at the functional requirements as described in [appendix B.2.3](#). It is our goal to have a satisfied client. This means that we have to implement the *must haves* from [appendix B.2.3](#). In other words, we have to at least make a dynamically scalable clock system, which is relatively fair and simulatable.

B.2.3. Functional requirements

Must have

- The system must be simulatable.
- The system must be able to scale dynamically to support a large number (1000+) of simultaneous client connections.
- There must be a graphical user interface which sufficiently resembles the existing clock system, to ensure user familiarity.
- A buyer must be able to place a bid on the clock.

Should have

- The system should have a graphical user interface for the auctioneer which sufficiently resembles the existing user interface.
- The clock bounce should be configurable by the clock administrator.
- The clock speed should be constant and be configurable by the clock administrator.
- The graphical user interface should be fully responsive to enable interactivity on any device size.
- The clock data should come from a central system that is able to manage all clock system instances.
- The results of the auction should be stored in this same central system.
- The user interface should have an option to switch between clocks.

Could have

- The system could have a standardised set of messages that can be broadcast to the auctioneer. These messages can be sent by the buyer if they make a mistake.
- Product information that is relevant to buyers, including photos, could be displayed in the live auction feed.
- The user could be able to switch between a clock view and a numerical view.
- The graphical user interface could have support for multiple languages.
- The system could have an alarm functionality that allows buyers to receive a notification once a product is about to be auctioned.
- It should be possible for the users to pause participation in the auction process (switch to spectator mode).

Won't have

- The system won't have a chat function.
- The system won't have voice-over-IP (VoIP) functionality.
- The system won't have an artificial intelligence system to automatically control auctions.
- The realtime communication systems won't be started in multiple geographical locations.
- The system won't have wish-list functionality that automatically updates when a buyer purchases something.
- The system won't have grower reviews.

B.2.4. Nonfunctional requirements**Must haves**

- The back-end code must target a .NET Standard-compliant environment.
- The system must be cross-platform deployable.
- The system must be deployed on a cloud computing service.
- The system must be maintainable for developers after the project is finished.
- The system must be as fair as possible to all parties. By this we mean that all auction buyers are treated equally, regardless of non-excessive network delay or physical location.

Should haves

- The back-end should be written in C#.
- The source code of the system should follow the C# coding conventions. [REF]
- The front-end of the system should be written in TypeScript.
- The front-end of the system should be written using the React front-end library.
- Front-end styling should be embedded in the TypeScript source code.
- The system should have no compilation warnings.
- The realtime communication service should be scalable on multiple cloud computing instances. If load is heavy, it should then be possible to add more computing power so that the load is distributed.
- The functionings of the system should be documented.
- The front-end will use the Floriday UI package to reuse UI functionality.
- The system should have a high performance, so that its response times are less than one second.
- The system should be secure, so that no data belonging to one user is sent to another.
- The system should have a high availability, so that a user can always use the system as expected without issues.

Could haves

- The system could run in a Docker container to enhance portability.
- Automatic tooling could be used to enforce proper documentation.

B.2.5. Deliverables

At the end of the project we will deliver three different separately deployable applications:

- The graphical user interface (this includes the interface for the buyers, auctioneers and administrators) as a web application.
- A real-time communication service (RTCS) capable of maintaining a part of the connected clients to a clock instance.

- A clock service capable of performing all the computations necessary to host an auction and to dynamically start RTCS instances for extra users when the need arises.

JEM-id will remain full owner of the produced source code and this code will not be published.

B.3. Process

B.3.1. Communication

In this section, we will describe how we will communicate with each other during the project. We won't go into detail about team problems, but rather address that in [appendix B.4.1](#).

For this project, we have several communication channels. The first is the daily stand-up meeting, in which we get an overview of what each team member is doing. Our client coach will attend these meetings, so that he will be up to speed on our current process all the time. This also makes it easier to ensure we produce what the client needs.

The second communication channel is a Slack channel, in which we can discuss all things. Our client coach is also a member of this Slack channel, so that we can contact him any time too with small questions or request an additional meeting.

The third communication channel is a WhatsApp group with only the team members. This channel is rather informal, and it's main use is to inform other team members of delay in transport or other small questions.

The final communication channel is Mattermost. All team members have an account here, which can be used to contact our TU Delft coach directly. This is only used to set-up meetings or ask very small questions for which a real meeting is not needed.

Besides these communication channels, we all have been provided a desk next to each other. These are placed in a silent room where we can work without distractions. If we want to discuss things, we have permission (and are therefore also asked) to book a meeting room. To further promote good communication, we have agreed to be present at least four out of the five weekdays. When taking a (part of a) day off, the team should be informed at least a day prior and it should be marked as 'not at work' in the corresponding Google Calendar. Hours not made during the week, should still be made in the weekend instead. We believe in personal responsibility, meaning that each person should be an adult and make sure they make their hours. If other team members suspect another team member is seriously slacking, they should take it up with them as described in [appendix B.4.1](#).

B.3.2. Workflow

For our project we will be using git repositories hosted on Azure DevOps as our source control system. Additionally, we will run our Continuous Integration (CI) on Azure. We will use a pull-based development strategy, meaning that we will not push to the master branch directly (or even be allowed to do so), develop new features on feature branches and merge branches into the master branch only when reviewed and greenlit by our CI server.

The checks that our CI performs will include:

- Unit testing
- Code metrics analysis (such as SonarQube)
- Some form of style checking (e.g. StyleCop)
- Some form of code analysis (e.g. FxCop)

B.4. Risks

As in any project, there are several risks. In this section, we will identify some risks that may occur at any time and propose a solution for these risks.

B.4.1. Team problems

There are several cases, in which team members might have issues with one another. When this happens it is important to always be respectful towards one another. A team member is responsible for their own actions and thus must take appropriate actions. This might be talking to the team member they are having issues with, or take it up with the team. If it reaches the point that there is visible tension, the team coach should be asked to mediate. If this doesn't lead to a solution, another one must be found. In the worst case, this can be the removal of a team member.

B.4.2. Expected design is infeasible

It can turn out that the wishes of the client are infeasible with the resources and time we are given, or at all given the current level of technology in the world. If this is the case, it is important to schedule a meeting with the client as soon as possible. In this meeting, we have to let the client know which part of the assignment is infeasible, and why.

If it's due a time constraint, it should be discussed how we can leave the project so that they might either continue it after our time is over, or have a smaller version of it.

If it is due a problem with resources, we should discuss if we can either get more, or if a lower quality of performance is allowed.

Finally, if the expected design is infeasible due technological advancements not yet available, we must discuss reducing the requirements to either make it feasible within current technological standards, or leave out other requirements so we can research this part. In the last case, we should inform them of the risk that research not always leads to a solution given finite time and resources.

B.4.3. Inadequate research

During this project, the project team is expected to first do research. This research consists of getting to know the problem, finding out who the stakeholders are, and coming up with a solution to the problem. All this must be done while balancing the needs of the involved stakeholders. To find this solution, it is necessary that we research the relevant techniques and technologies so that we can build a solution that works well, is maintainable and is usable.

Although we will attempt to do the research as thoroughly as possible, we may be required to adjust or ignore our findings later. It might happen that we are unable to use the solution that was originally thought to be viable. In that case, we will attempt to do more research so that we can build another solution to the problem that works better.

B.5. Planning

In this section, we will discuss our planning for the project.

Delft University of Technology (TU Delft) has given us several deadlines, displayed in [table B.1](#). These deadlines are minimal, which means more may be added in correspondence with the TU Delft coach. Since they won't be official either way, we will not mention them here.

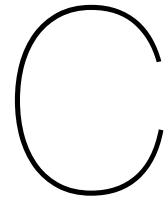
Week	Date	Description
Week 1	April 26, 2019	Project plan due
Week 2	May 3, 2019	Research project due
Week 6	May 31, 2019	Submit code to SIG (1)
Week 9	June 21, 2019	Submit code to SIG (2)
Week 10	June, 25 2019	Submit final report
Week 11	July 1-5, 2019	Final presentation

Table B.1: Deadlines given by TU Delft

Besides these deadlines, we have made a roadmap as seen in [fig. B.1](#) (on the next page). In here, we give a loose plan which functionalities will be worked on when. If a functionality is in the same row as another, but later in time, it is dependant on the other functionality. It can thus not be started before the other is finished.

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10
Deadlines	April 26, 2019: Project plan	May 3, 2019: Research project				March 31, 2019: Code metric analysis 1			June 21, 2019: Code metric analysis 2	June 25, 2019: Final report
Phases	Research		Programming and Testing						Finishing touches, start final report	Final report
Research	Project plan	Research plan								
Clock UI			Learn TypeScript and React							
			A view with a countdown of the product price	Create buyer view		Create auctioneer view		Create numeric view		
			Set up project tooling	Retrieve clock CA server	Connect to RTCS			Support multiple languages		
			Set up deployment							
Clock API			Learn C#	Connect to RTCS			Spawn new RTCS instances			
			Set up basic API server			Process RTCS results				
			Have an authentication mechanism							
			A clock mechanism is present			Auctioneer controls are present				
RTCS			Connect to server		Sync/fairness mechanism					
				Connect to clients						
Simulation			Simulation							
				Synthetic data generator						

Figure B.1: Roadmap of the project



Research Report

All content in this section is directly taken from our original research report. Although the content remains identical, minor changes have been made to reflect the visual style of this document and to incorporate feedback from our supervisors.

C.1. Introduction

The Netherlands is the largest exporter of decorative flowers in the world [1]. This billion euro industry is powered by relations between transporters and growers. Traditionally, transactions between growers and transporters or exporters are set up by a third party through an auction. We will be working to improve the *Dutch auction* set up by Royal FloraHolland (RFH), the largest flower auction worldwide [2]. In a *Dutch auction*, the auctioneer sets a starting price for the product, which is then gradually lowered until a buyer accepts the price or the price falls below some threshold. A clock-like interface, a vestige of the analogue and mechanical clocks used in the past, visualises the current price.

Our client is JEM-id, a company that develops software for the agri- and floriculture sector [3]. JEM-id has partnered with RFH to develop the software platform Floriday, which offers all kinds of services to growers and transporters [4].

Currently, it is possible to bid on auctions remotely, using KOA (Kopen Op Afstand — “buying remotely”) [5]. However, this system requires the installation of special software. Furthermore, the code-base is old, and it is difficult to add new features. This project aims to develop an online web-based version of the clock that can be integrated into Floriday. The system should be scalable and as fair as possible: It should alleviate the disadvantages that stem from remote localisation of buyers compared to on-site buyers.

In this research report, we will first give a thorough analysis of the problem in [appendix C.2](#), reviewing the current auction system and its different types of users. Additionally, we discuss our client’s wishes and requirements. As our client is a software company, they already had some proposals for the UI and architecture of the system, which we discuss in [appendix C.2.2](#). Finally, in [appendix C.3](#), we will argue which technologies we think are most suited for building the system.

C.2. Problem analysis

This section will first review the current situation. After that, it will introduce our client and their wishes, followed by a short introduction of one of their systems. Finally, we discuss some differences in view that JEM-id and the auction have.

C.2.1. Current auction

This section describes the current auction and analyses every aspect that will have to be replaced by our application. Besides that, it looks at all external influences and data sources our application will need to process and take into account.

The clock

The process of a Dutch auction is the same for each product that is auctioned off. Each auction takes place with a big clock as its centrepiece. The clock counts down for each product and stops when a buyer presses a button to buy the current product. The first buyer to press the button wins the auction and purchases the current product for the price the clock indicates at that time. If there is stock left, the auctioneer raises the price, and the auction is resumed. If none of the buyers presses the button before reaching a minimum price, the auction of the product closes.

Buyers

Almost two decades ago, Royal FloraHolland's precursors introduced KOA (Kopen op Afstand — "buying remotely") to make their clock auctions accessible to anyone [6]. Even buyers who do not live nearby the auction can now access it. For some auctions, there is no physical room associated with the auction due to KOA becoming the more popular option. We will distinguish between buyers buying through the physical auction room and buyers buying through KOA due to their essential differences.

Buyers physically present in the auction room have an advantage over KOA buyers in a few ways. First of all, each table in the auction room is directly wired to the clock system, meaning button presses from on-site buyers may potentially reach the auction system faster. Secondly, buyers physically present may have the opportunity to see the state of each product in real life instead of from a picture. This opportunity gives a great advantage when the taken picture is not of very high quality. Finally, depending on the auction room, the room might provide specialised hardware with a buy-button for each typical quantity. If provided, buyers can efficiently indicate the number of containers to buy in a single button press.

KOA buyers have one obvious advantage over on-site buyers: KOA buyers do not need to be at any specific location to buy from that location. This advantage implies that KOA buyers may buy from many different clock auctions at once, watch all prices and buy from the cheapest clock that day. Being on-site and using KOA can be combined to get the best of both worlds: an on-site buyer may bring their laptop to keep an eye on the other auctions through KOA.

For both KOA buyers and on-site buyers, the same rules apply: Each buyer can only perform a few actions. First, a buyer can buy (a part of) the offer for the current clock price. Secondly, when a buyer made an incorrect purchase, they can communicate this to the auctioneer by stopping the clock. Stopping the clock is only allowed when the bought products are still on the clock. After stopping the clock, the auctioneer will contact the buyer and correct if needed. The auctioneer can also contact a buyer when they find the price unreasonably high as a precaution.

The buttons for these actions may differ per location, but every location always includes a button that both stops the auction and starts the transaction. Some locations provide specialised hardware (as mentioned earlier) and complete the transaction within one button press. Other locations use a voice-chat in which a buyer indicates the quantity they want to buy after winning.

Auctioneers

Each clock auction is overseen by an auctioneer. Auctioneers fulfil this role by observing all bids and noticing price changes during the auction while simultaneously keeping the auction running. Their job is to act on changes and irregularities during the auction. An auctioneer can influence many things: the speed of the auction, the starting position of the clock, whether a transaction is approved, the minimum quantity to buy and so forth. Auctioneers have the power to ensure every grower gets a similar price for their goods, independent of what time their product is up for auction.

An auctioneer has to perform these actions swiftly. For instance, changing the minimum amount per transaction may happen between auction rounds, which usually last only a couple of seconds. For this, the auctioneer has a special keyboard (see [fig. C.1a](#)). This keyboard allows them to change the minimum purchase quantity of an item in a single button press, amongst other things. The advantage of this is that an auctioneer can do many actions with little button presses, making the auction run smoothly.

Auctioneers have a specialised view of the auction (see [fig. C.1b](#)). This screen displays many essential pieces of information to the auctioneer. In the top-left, the next six items up for auction, and a detailed

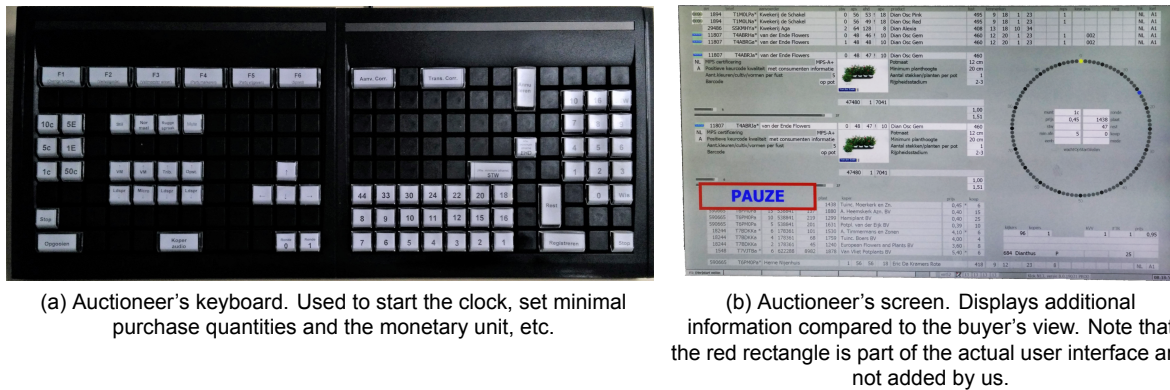


Figure C.1: The tools used by an auctioneer to lead the auction.

overview of both the next and the current item are displayed. These detailed descriptions include information about the product and its grower. A common point of interest is the quality of both the product and its grower, and the features of the product such as colour, length of the stalk and size of a container. In addition to the information specifically about the product, the auctioneer also gets to see information about containers sold during presale in the bottom-right of the screen. The auctioneer has the same clock element as the buyers. Finally, the auctioneer has access to all previous transactions made during the auctioning of the current product in the bottom-left. When a buyer reports a mistake, the auctioneer can use this information to judge and overrule the faulty transaction. Once the next product in the queue is auctioned, these transactions can no longer be amended.

Other kinds of transactions

Besides the auction, there are other channels through which growers can sell their stock. According to Royal FloraHolland's annual reports, the clocks sales percentage is ever decreasing while direct sales are increasing [36]–[38]. The clock has not become irrelevant, however, since it has acquired a status as a price determining mechanism.

One such channel, as stated earlier, is direct sales. With direct sales, a grower finds buyers and sells directly to them. For this, they often use platforms like FloraXchange, FloraMondo and Floriday (see [appendix C.2.3](#)).

Another channel is the presale the auction offers. Through this medium, the grower might offer up to half of the containers they entered into the clock auction for clock presale, where a buyer may purchase it for a fixed price. This price is usually a bit higher, but it also gives a guaranteed transaction to the buyer, whereas clock sale depends on buyer performance.

C.2.2. JEM-id

JEM-id is a software company that develops software for the agricultural sector. JEM-id proposed the request to build an online auction clock. This online auction clock would be a great addition to their current software, especially Floriday (see [appendix C.2.3](#)): integrating the clock with their existing tools will provide easier access for growers and buyers using the platform. JEM-id has a clear vision of the product and how its architecture should be designed to make for an efficient and integrated auction.

JEM-id has proposed the model displayed in [fig. C.2](#). Here, Floriday is external software, to which we can send API requests. Floriday has prices and pictures of plants and flowers, and information on the stock of growers, which the clock can use. To the left of Floriday, we can find the CMA, which stands for Clock Management API. This service should manage clocks and be allowed to create and destroy clock instances. CA, which stands for Clock API, represents the actual clock auctions. There can be many of these, all managed by the same CMA. Every CA has multiple RTCSs, which stands for Real-Time Communication Service. This RTCS manages a connection with the actual client, who is represented by the UI. When an RTCS fills up, the CA requests a new one. See also [appendix C.3.3](#)

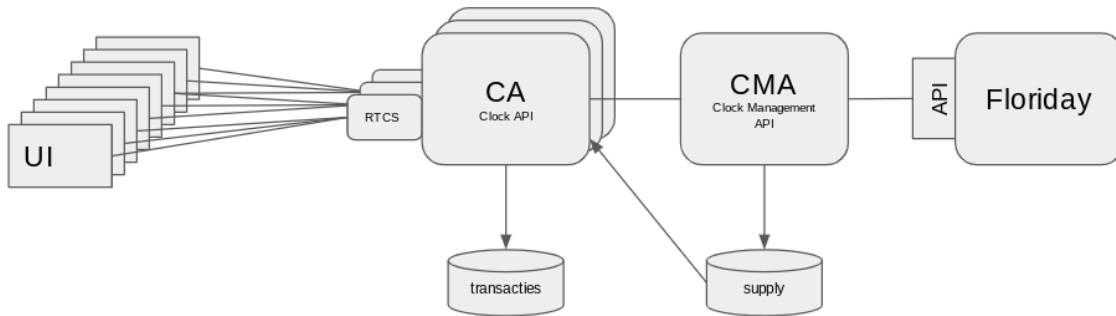


Figure C.2: The model as proposed by JEM-id.

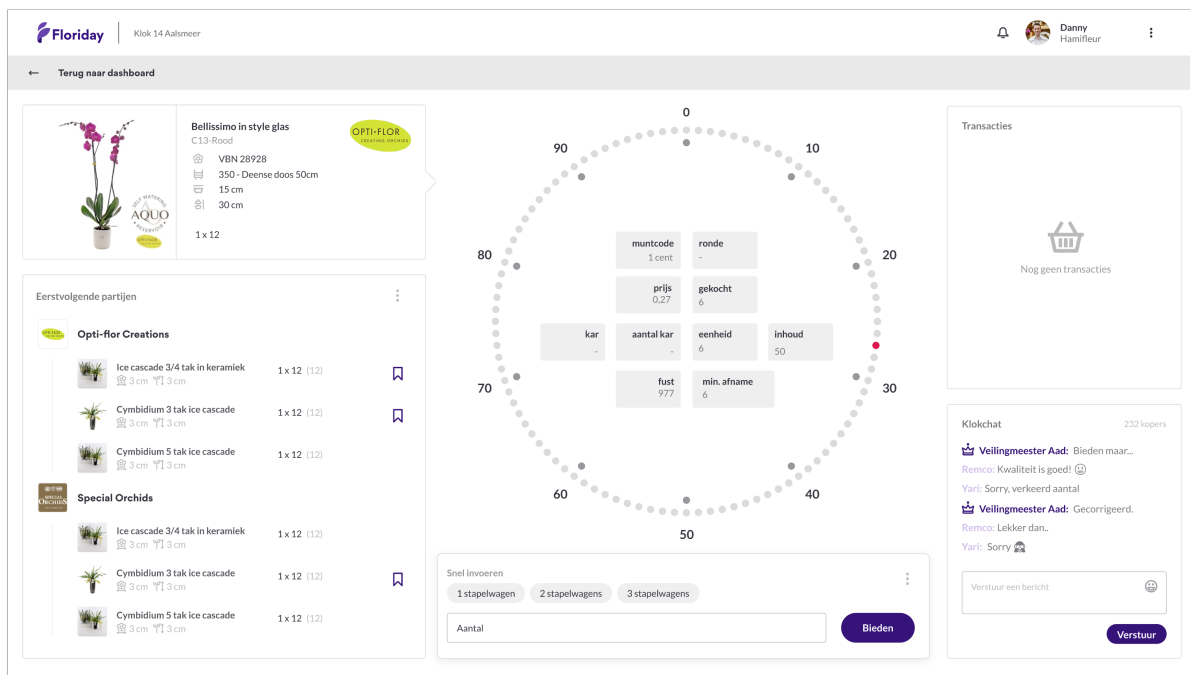


Figure C.3: The new clock layout proposed by JEM-id.

for more details.

JEM-id has tasked us with implementing the UI, RTCS and CA, which is the clock and its communication with clients and the CMA.

The UI

JEM-id proposed the UI for a buyer to be [fig. C.3](#). From the image, we can already see several requirements proposed by JEM-id.

In the left part of their design, we can see the details of the current auction. Below, we see the upcoming products. In the middle, we have the clock, displayed as a circle consisting of smaller circles. The current price is coloured red. Additional sale information, such as the minimum number of products to buy, is displayed inside the circle. Underneath is the buy menu, where a buyer first has to enter how much they want to buy before they can place a bid. On the right, we see the user’s previous transactions. Beneath it, we can find a ‘clock chat’, in which a buyer can talk to the auctioneer. This chat comes into play when a buyer places a faulty bid.

Back-end features

Besides the features that are apparent from the UI, there are other features beneath the surface. One such feature is the decision process on who is the winner. The auction would like this to be as fair as possible, which means that, ideally, the first bidder should be the winner.

Another feature is for the CA to send all transactions to the CMA, so that Floriday and internal databases may be used to further process transactions.

When a buyer places a bid, a so-called 'grace period' is started. In this period, which lasts several hundred milliseconds, bids can still be placed. Because of this grace period, people who have a disadvantage due to delay can be compensated by evaluating all incoming requests. When the period has passed, rules defined by the auction will be used to pick a winner. For example, we could use the one that was ultimately placed first or the one with the highest price per item.

A final, somewhat hidden feature is client alerts. Clients should receive an alert on several occasions like winning (or losing) an auction or reaching the minimum price (which means closing the auction).

Future expansions

In the future, JEM-id would like to expand the digital clock. They would like us to take such future expansions into account already. Designing the system in an extensible way will allow later development to be smoother. Some of these features that we will keep in mind but not pursue include:

- Managing several clocks using the CMA. This CMA gives information about the supply to the different CAs. We should thus anticipate the CMA being a standalone API.
- Internationalisation. It would be useful if people all over the world can use the system in their own language. Therefore we should build the system so that it has support for multiple languages.
- New clock layouts. The current layout is based on a rather old layout. Revising this layout in the future might be useful. Therefore, the UI should be extensible so that different UIs are interchangeable.
- Floriday authentication. Users should eventually be able to sign into the system using their Floriday account. See also [appendix C.2.3](#).

C.2.3. Floriday

Quoting the website, 'Floriday is the worldwide digital platform for the floriculture sector that helps the grower to organise their commercial processes efficiently' [7]. It is an application developed by JEM-id and Royal FloraHolland, launched in 2018. Its purpose is to help growers organise their stock, orders and logistics. They can also direct their stock to several channels, like the (analogue) clock.

This system could be connected to our digital clock API later on (as mentioned in [appendix C.2.2](#)). Floriday already allows growers to set different prices for each buyer, so it would be easy to add a minimum price for the clock auction to this data model. Floriday itself is easy to connect to since it is built using a microservice architecture.

C.2.4. Conflicting requirements

Where JEM-id is our client, the auction is their client. We should thus try to satisfy both as much as possible. We have encountered several differences in their wishes and will address them here.

The first difference we noticed is the minimum price for a given product. Where JEM-id wanted to hide this information, the current auction actively shows this with a blue dot.

Another disagreement is the chat proposed by JEM-id. Currently, a buyer can stop the auction, and by doing so, inform the auctioneer of the buyer's mistake. After this, the auctioneer makes contact with the buyer. This system could be seen as an active phone call with hundreds of people in it. JEM-id proposes a chat to replace this system. The auction, however, saw several problems with this, the first of which is be message overload. Buyers might feel less of a boundary sending a message because a chat is more informal. The chat would then go to fast for the auctioneer to read and act upon. Another is the slower speed of typing compared to talking, which means that correcting a wrong order might

take longer. Instead of a chat, the auction proposed a standardised set of messages instead, from which the buyers can choose. The advantage of this system is that it keeps it formal. One can also restrict the messages a particular buyer can send. Finally, if needed, a private chat can be opened to get more information about the situation. It was clear, however, that the standardised set of messages was still not considered optimal by the auction.

A final difference was the power of the auctioneer (and administrator). Where JEM-id gave the auctioneer authority to alter the minimum price for a product, the auction does not. According to the auction, this is even legally prohibited.

C.3. Technologies

The architecture proposed in [appendix C.2.2](#) consists of three components that we will develop. These different components might require different technologies to be used, due to differences in nature of the components (API vs GUI) or other constraints. In this section, we will describe the different technological options available, which ones we decided to use, and the reasons behind these decisions. Before describing each technology separately, we will first describe some choices we have made for all technologies.

C.3.1. Tooling and deployment

First of all, we would like to be able to deploy our code on as many platforms as possible. The software should therefore not be tied to any specific vendor or platform. It must also be possible to deploy the code on public cloud services such as Microsoft Azure, Amazon Web Services, or any other cloud provider. We would thus like to be able to package the application in such a way that it can be executed on any of these platforms. To do this, we want to use Docker containers, possibly combined with Docker orchestration services such as Kubernetes, which can automatically deploy the software and manage how it is scaled [\[39\]](#).

We also want to use Continuous Integration (CI) and Continuous Delivery (CD) on our projects, so that every commit is automatically built and tested. For this, we will use the CI service integrated with the source control system given by the company, which is Microsoft Azure DevOps [\[40\]](#). The CI/CD service is named Azure Pipelines, which allows developers to configure a CI/CD pipeline. This pipeline would ideally first test the code to ensure that it works correctly, and then package it into a format that can be easily distributed, such as a Docker image.

We will also be enforcing a standard code style using style checking tools. If the code style is incorrect, the build should fail during the CI process. We also plan to use Swagger/OpenAPI tooling for our API services. This tooling allows us to develop interoperable APIs more easily. It also enables us to automatically generate a client that is capable of communicating with our API service, so that we do not need to write code to connect to our API endpoints manually.

C.3.2. Clock API

Web frameworks

The Clock API is a Web API. For its underlying server we have considered the following frameworks: Express, ASP.NET, ASP.NET Core. Express is a commonly used Node.js framework for making web applications. It provides capabilities for creating RESTful APIs [\[41\]](#). ASP.NET and ASP.NET Core are both open source products owned by Microsoft [\[42\]](#). ASP.NET Core is a modern re-implementation of the ASP.NET web framework on the cross-platform .NET Core framework. Currently, most newer products of JEM-id use ASP.NET Core, while their older products tend to use ASP.NET.

When it comes to platform support, both Express and ASP.NET Core work on multiple platforms (Windows, Linux and Mac). ASP.NET on the other hand, only runs on Windows [\[43\]](#) and is therefore not suited for our purposes.

On the performance side, ASP.NET Core outperforms ASP.NET [\[43\]](#). When we compare the performance of ASP.NET Core and Express, we find that ASP.NET Core has higher performance on cloud-based systems [\[44\]](#)–[\[46\]](#).

In conclusion, we have decided to use ASP.NET Core, because it is the current go-to framework of our

client, its cross-platform capabilities and its high performance.

Programming languages

In principle, ASP.NET Core supports any language which is supported by the .NET Core framework. We will focus on the three officially supported languages: C#, F# and VB.NET and thus we will not touch upon third party supported languages such as IronPython [47], IronJS [48] and IronRuby [49] or languages that have previously dropped support for the .NET Framework such as Scala [50].

C# is the most commonly used language of the three [51]–[53]. It is a multi-paradigm language which mostly supports imperative programming styles. VB.NET mainly provides a different syntax for the same capabilities as C# does. F#, on the other hand, is a multi-paradigm language which emphasises functional programming constructs [54]. However, we do not deem it necessary to use functional programming for this project and have thus decided to use C#.

Testing frameworks

The .NET Core ecosystem knows several commonly used unit testing frameworks. The oldest testing framework is Microsoft's own MSTest [55]. Additionally, the most popular unit testing framework is NUnit [56]. Finally, the newest unit testing framework, created by the original creator of NUnit, is xUnit [57]. All these frameworks share the same features, although they do not share the same syntax [58]. We have decided to use xUnit because it has gained a lot of popularity, even though it is the youngest framework.

For mocking, there exist numerous libraries. The most used one is Moq, which is free and open source. However, it requires marking functions as `virtual`¹ or implementing an interface that declares the functions that should be mocked. There does exist a more powerful mocking library, TypeMock, which can mock/stub any arbitrary class and function. However, this library has licensing costs [59] and will therefore not be used. RhinoMocks previously used to be the most popular mocking framework for .NET Framework. However, it has not received an update since 2014 [60] and thus does not support .NET Core. Another possibility is Foq [61], which focuses on mocking in F# projects, which is not the language we decided on in [appendix C.3.2](#). Thus, we will use Moq for our mocking purposes.

Static analysis

For the purpose of code style enforcement, we will introduce Microsoft's StyleCopAnalyzers to our CI pipeline. StyleCopAnalyzers allows specifying StyleCop rules which enforce a given code style [62], [63]. We have decided to use StyleCopAnalyzers because Microsoft supports it and it integrates nicely with the tooling Microsoft provides.

For all other static analysis purposes, we will use Microsoft's FxCopAnalyzers in our CI pipeline. FxCopAnalyzers uses FxCop rules [64] to check for possible issues in C# code [65]. We chose to use FxCopAnalyzers because it is supported by Microsoft.

Database

The Clock API will need a database to store properties about itself, transactions and possibly other gathered data. It is possible to store this data in memory, but if the service unexpectedly exits, all transactions would be lost if they had not been flushed to the clock manager earlier.

The type of database is not very relevant. The preferable option would be a relational one since these tend to have properties that are desirable for transaction records (ACID) [66]. The exact vendor does not matter that much. Cloud hosts, such as Amazon Web Services and Microsoft Azure, can host all types of databases. These databases are then reachable using the respective platform's APIs. By using some abstraction layer, it would be trivial to implement connections to another database type. Since our client has historically used Microsoft's SQL Server product, it is likely that we also will use this system.

The chosen database does not need to scale a lot, because the only data stored would be transaction data that is supposed to be emptied for the next auction day, thus we will not expect more than a few thousand entries stored in total. As such, a conventional SQL-based database should be sufficient and more than capable. A NoSQL (non-relational) solution is not necessary. We will therefore not

¹In C#, the keyword `virtual` denotes a function that can be overridden in a subclass.

investigate the different non-relational database systems for storing the data gathered by the Clock API service.

C.3.3. Realtime communication service

The real-time communication service would require the possibility to push messages from the server to the client in a timely manner. Using regular HTTP responses would thus most likely not be suitable, because this requires a lot of polling from the client side. WebSockets do provide this needed functionality [67], [68].

SignalR is an abstraction layer on top of WebSockets, which has a fallback to other means of achieving bidirectional communication between client and server [69], [70]. ASP.NET Core natively supports SignalR. However since it is an open standard, implementations exist for many web frameworks, including Node.js [71].

We have decided to use ASP.NET Core for the real-time communication service because of five reasons. Firstly, we are already using ASP.NET Core for the Clock API. Secondly, there is no evidence that ASP.NET Core cannot achieve the performance goals we have set. Thirdly, we think it would be beneficial to keep our development stack uniform and straightforward. It would also allow us to use the same frameworks/libraries for testing and static analysis as discussed in [appendix C.3.2](#). Finally, it is also possible to compile the project ahead-of-time into native code using CoreRT [72] for higher performance.

C.3.4. Graphical user interface

This section describes how the front-end will be built. We will first discuss what type of website we need to build. We will then decide on the programming languages and libraries used. Finally, we will discuss how this system will be tested.

Application type

The front-end of the system will be a web document that can be displayed in a user navigator (also known as a web browser). Some constraints to this document are that it should be interactive, highly responsive and fetch real-time data. One model that is very suitable to achieve this goal is a Single-Page Application (SPA) [73]. This type of web application only loads the necessary resources once and retrieves data from a server to display to the user. In this process, the page is never fully reloaded.

Another, more traditional method is Multi-Page Application (MPA). In this method, a new web document is retrieved every time the user navigates to a new location. Both methods have their pros and cons. For example, the SPA can more easily expose a user interface (UI) with many features, that connects to an API to fetch data. An MPA is more suitable for displaying static data and does not necessarily require JavaScript to execute code in the client navigator.

We would like to have an application that is able to perform real-time communications with a server while displaying data to the user. Therefore, it would be more beneficial to use SPA technologies for the UI rather than MPA technologies. A disadvantage is that we will now have to load all application resources upon loading the web page, making the page load slightly slower. In addition, we are dependent on the user having enabled JavaScript. An advantage is that using JavaScript does allow us to use client resources in a more optimal fashion, increasing the scalability of the system.

Programming languages

The application should be able to run application code in the client navigator in order to build a Single-Page Application. A programming language that is widely supported, though in varying degrees, is JavaScript (standardised as ECMAScript) [74]. JavaScript does not have static types, although such static analysis is useful for catching common programming errors early. Furthermore, type information is useful for program comprehension [75]. As such, we would like to use a typed language which can be transformed into JavaScript code.

A programming language that is useful for adding type information to JavaScript is TypeScript. TypeScript is syntactically a superset of JavaScript, adding type annotations and new functionalities that can be transpiled² to common JavaScript [76], [77]. Type checking enables us to have code that is

²Transpilation is the process of transforming one type of source code to some other source code automatically.

less error-prone while still offering the same functionality as JavaScript. We will be using linting tools such as ESLint and TSLint to ensure that a common and readable style is used. These tools can also provide some level of static analysis, such as detecting trivial cases of conditions that can never be true.

Front-end library

Several frameworks and libraries have already been developed to realise SPAs. Some popular and widely used examples include *Angular*, *Meteor*, *React* and *Vue.js*. Each library has some advantages and disadvantages. In this section, we will be comparing these libraries based on their features, tooling and flexibility. All candidates are widely adopted and have many plugins and extensions available.

Angular is a commonly-used framework to develop front-end applications. It uses a Model-View-Controller style architecture to construct the user interface. For this, the developer specifies the data model, the controller, and the UI. Angular then renders this data into the UI using bi-directional data binding.

Meteor is a web framework that automatically synchronises data between the client and the server. It uses MongoDB as its primary data storage solution. Meteor requires that the entire software stack is designed around meteor usage. It can then use any other front-end library to actually render this data to the user.

React is a widely used front-end library that is designed to build a user interface within the web browser's Document Object Model (DOM). It does not enforce any specific way of performing actions within the app, and as such, it is merely a library for rendering a graphical user interface.

Vue is similar to React as it is also often used to build user interfaces. Vue is slightly younger than React and has incorporated some features from both React and Angular in its design. It also enforces the use of a model and is consequently more opinionated than React, but less so than Angular.

All frameworks support TypeScript, so tooling would not be an issue. Since Meteor demands that the server is built in a specific way, it is not the right solution for us. We want to keep the front-end as flexible as possible, which makes React a more suitable candidate. Eventually, this application will become a part of *Floriday*, which already uses React. As such, we will also use React to build the front-end. This also allows us to use *Floriday UI*, a UI package that allows us to style the user interface in a fashion similar to the main *Floriday* application.

GUI Testing Frameworks

This section will explain which GUI testing framework will be used. To decide which frameworks to consider, we looked at their popularity on GitHub (according to the number of stars), and the number of npm installs [78]. For obvious reasons, a framework with a large active community is preferred. This project will use Azure Pipelines as CI environment (see [appendix C.3.1](#)). Therefore, the GUI testing framework of choice must be compatible with Azure Pipelines. Furthermore, we will also be using React to build our UI (see [appendix C.3.4](#)), so testing frameworks aimed at Angular applications, such as Protractor, were not considered.

From this selection process, two main contenders emerged: Selenium WebDriver with over 14 thousand stars and Cypress with over 11 thousand stars. Both Selenium WebDriver and Cypress have an active community and are actively supported and developed. Both are open source [79], [80]. There are, however, many points on which they differ.

Selenium WebDriver focuses more on experienced programmers and is very flexible [81]. As a consequence, there is a steeper learning curve [82], [83]. Since time is of the essence for our project, this is a major drawback. Cypress, on the other hand, focuses on simplicity [84]. It is an all-in-one solution for end to end testing of web applications. It is easy to install and use, but has some limitations compared to Selenium WebDriver. It currently works only for Chrome, though it is on the roadmap to expand this to other browsers [85]. Additionally, the only testing language that is supported is JavaScript.

Another difference is how the tests are run. During testing, Selenium's components communicate over HTTP, using the WebDriver protocol [86], even when run locally, which can cause delays. Cypress,

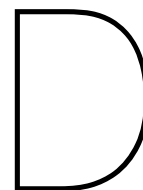
on the other hand, runs entirely within the browser thanks to their own DOM manipulation technology. According to Cypress, this feature has gained them more accuracy and makes tests run faster [84]. Cypress is also already used by RFH (a partner of JEM-id).

Taking all these factors into account, we will use Cypress as our GUI testing tool. Since our project has a considerable time constraint, it is in our best interest if our learning curve is as small as possible. We also think that the freedom Selenium WebDriver offers does not add much value; the choices Cypress has made for us, like the assertion library, seem reasonable. Finally, Cypress is already used by RFH, which might also expand on our project in the future.

C.3.5. Simulations

Before using a system, it is important to know how well it performs with a lot of input. Currently, around one thousand people are using a clock simultaneously. However, the system should be able to handle even more concurrent clients. To verify that our system is capable of dealing with such high loads it is useful to perform load tests and stress tests. Load tests aim to ensure that the system can operate on a large scale, while stress testing attempts to break the system by overloading it to find bottlenecks and failures [87].

However, it is not feasible to perform these load and stress tests using thousands of physical computers. Therefore we will simulate the users. For this, we plan on using a similar architecture as described by Noor *et al.* [21]. This architecture describes a single orchestrator service which sets up all clients and allocates test scenarios to the clients. Instead, we will use several cloud services, which each have multiple clients. We will use both real data provided by Royal FloraHolland and synthetic data that we generate ourselves. Such synthetic data may include for example bogus bids, to simulate multiple users bidding on an auction at the same time.



Simulation distributions data

D.1. Dataset 1

The first dataset contains $n = 3165$ samples of auction data from 27 August 2018. This dataset was needed in combination with dataset 2, since it did not contain information about the windup position.

D.1.1. Filtered data

The data was filtered in two ways: The non-usable columns were removed and the unrelated samples were deleted. The following columns were marked as non-usable:

All samples that weren't a purchase agreement, were deleted. After that, we removed the row, where the user was a test account. After this, $n = 2420$ samples remained.

D.1.2. Columns

In this section, we will describe all columns we used. We will not mention unused columns.

Action specification. The action specification of the action. For example, when the action was register, the specification can be clock purchase agreement. Used to filter out non clock purchase agreements.

User. The user that initiated the action. Used to filter out fake buyers.

Auction number. The number representing a product.

Increase position. The price at which the auctioneer stopped increasing, and the auction began.

Bidding price. The price for which the buyer bought the product.

Minimum quantity. The minimal quantity that a buyer ought to order.

Available quantity. The available quantity of the product.

Ordered quantity. The requested quantity.

Watchers. The number of people that watched the auction and could place an order.

Co-buyers. The number of people that placed an order simultaneously with the buyers.

D.2. Dataset 2

The second dataset contains $n = 157466$ samples of auction data from 15 April 2019. This dataset was needed in combination with dataset 1, since it did not contain information about minimal price.

D.2.1. Filtered data

The data was not further filtered, since the data contained no information about test users or types of transactions. Therefore, all samples are assumed to legitimate clock purchase orders.

D.2.2. Columns

In this section, we will describe all columns we used. We will not mention unused columns.

Bidding price. The price for which the buyer bought the product.

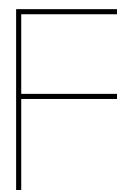
Minimal price. The minimal price the buyer can order the product for.



Simulation result data

```
{
  "winningBidResults": [
    {
      "timeStamp": {
        "time": string,
        "auctionItemId": number,
        "biddingSessionId": string,
        "quantityRemaining": number
      },
      "clientId": number,
      "shouldBeWinner": boolean
    }
  ],
  "skippingBidSessionResults": [
    {
      "timeStamp": {
        "time": string,
        "auctionItemId": number,
        "biddingSessionId": string,
        "quantityRemaining": number
      },
      "clientId": number
    }
  ],
  "pingResults": [
    {
      "timeStamp": {
        "time": string,
        "auctionItemId": number,
        "biddingSessionId": string,
        "quantityRemaining": number
      },
      "clientId": number,
      "roundTripInMillis": number
    }
  ],
  "clientsChangedResults": [
    {
      "timeStamp": {
        "time": string,
        "auctionItemId": number,
        "biddingSessionId": string,
        "quantityRemaining": number
      },
      "clientId": number,
      "amount": number
    }
  ]
}
```

Listing E.1: JSON format results






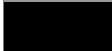
Ping Heatmaps

Using the `pingResults` from the simulation result data (see [appendix E](#)) we can create “ping heatmaps”. These are plots that show for each client (on the vertical axis) and each time interval Δt (on the horizontal axis) the highest ping value that was recorded within that time interval. The number of time intervals N was set to 100 for the creation of the plots in [figs. F.1 to F.5](#).

Let t_{min} be the lowest timestamp recorded and t_{max} be the highest timestamp recorded. Then Δt is determined as follows:

$$\Delta t = (t_{max} - t_{min})/N \tag{F.1}$$

The plots also show in lilac when no ping data was available. There are large chunks of lilac at the start and end of each simulation because we started and stopped batches of headless clients manually. In some cases, a client stopped producing pinging data midway through the simulation. We suspect that the headless client has a bug where the process that is responsible for pinging the RTCS can sometimes crash. This does not necessarily mean that the client crashed completely and was unable to participate in the auction. Due to the time constraints associated with this project, we have not fixed this issue.

Legend	
	No ping data available.
	Ping < 30 ms.
	Ping < 100 ms.
	Ping \geq 100 ms.

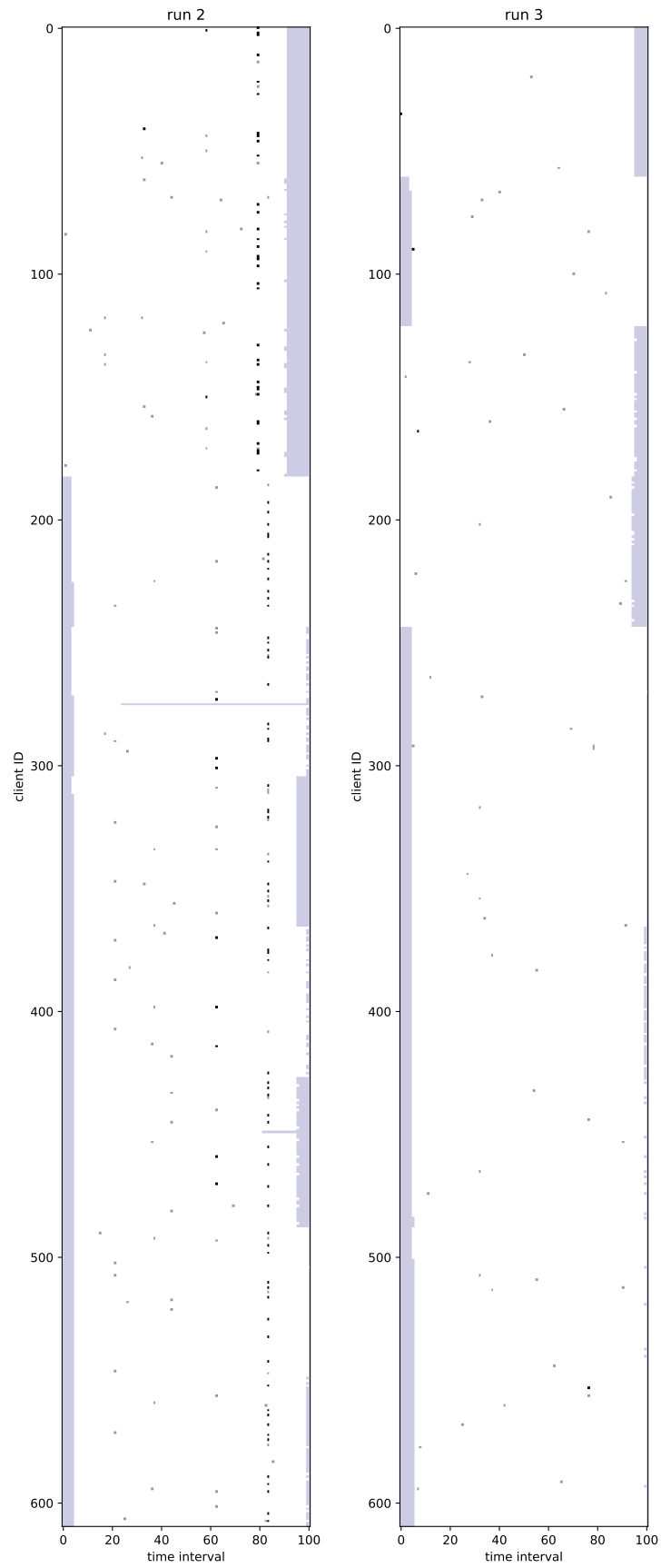


Figure F.1

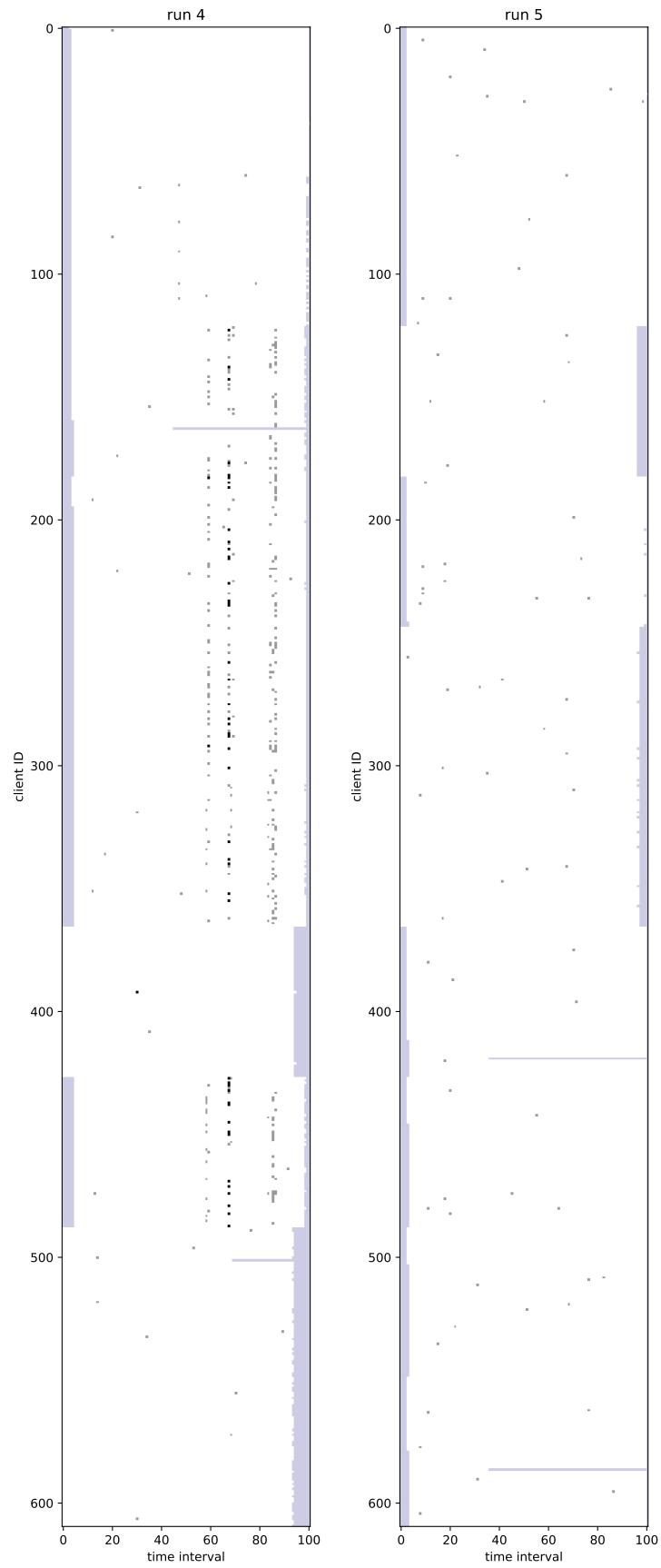


Figure F.2

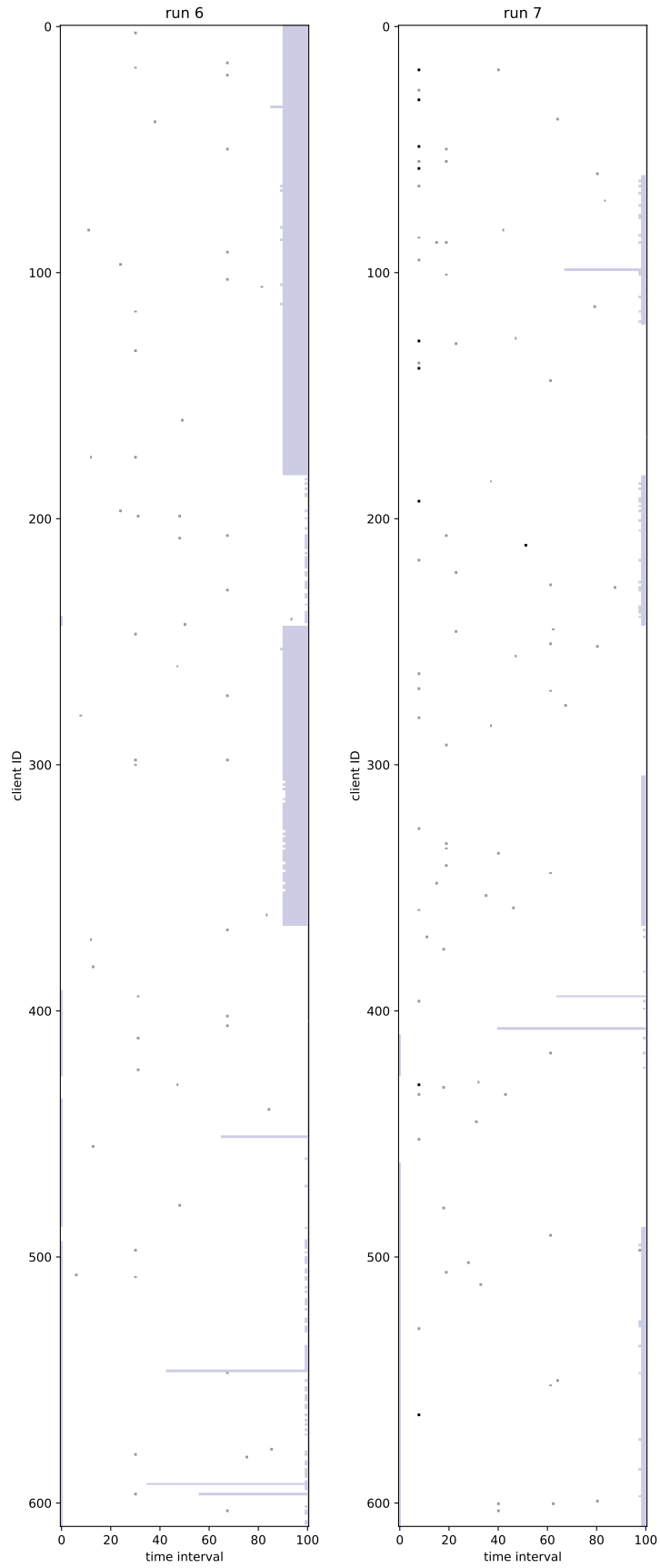


Figure F.3

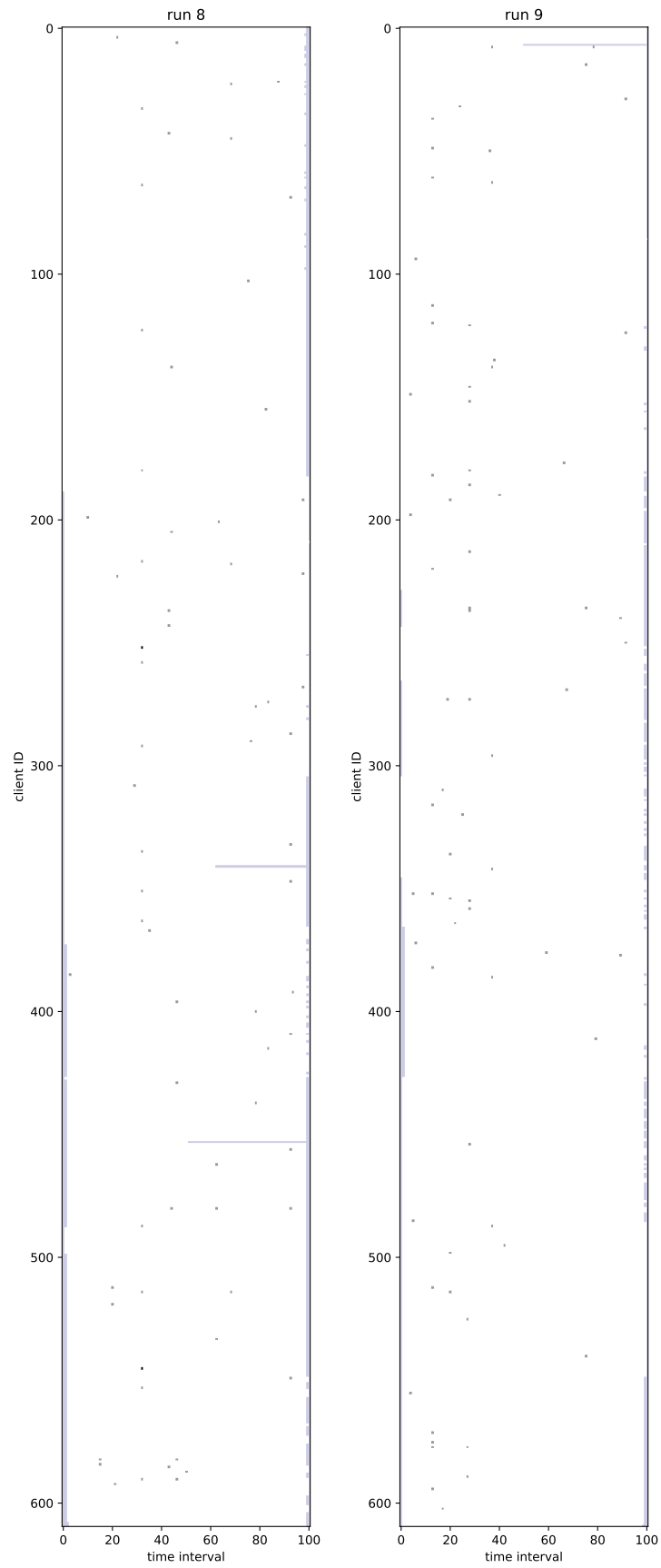


Figure F.4

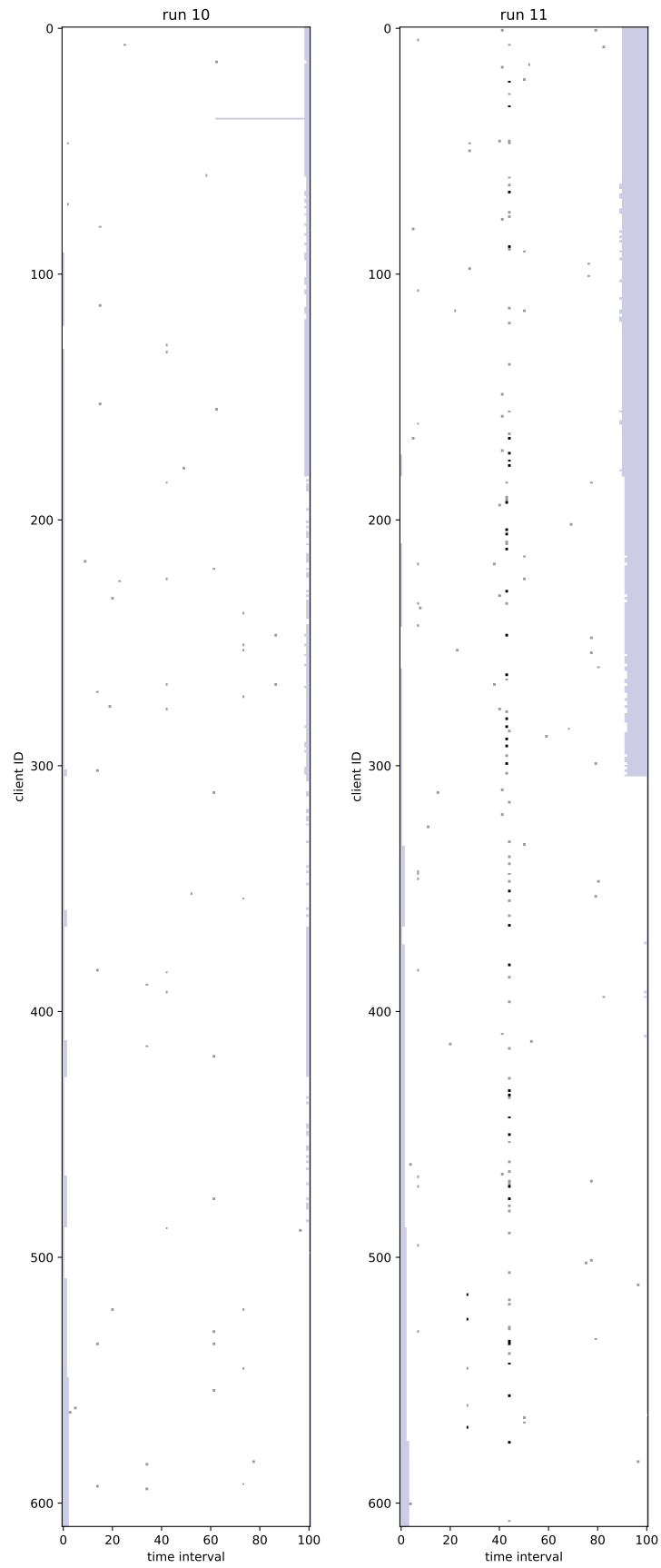


Figure F.5



Feedback SIG

Below is the feedback we have received from SIG (Software Improvement Group) listed verbatim. Although the contents are identical, changes have been made to typeset this text in \LaTeX properly. At the time of writing this report, we have not yet received the second round of feedback from SIG.

Beste,

Hierbij ontvang je onze evaluatie van de door jou opgestuurde code. De evaluatie bevat een aantal aanbevelingen die meegenomen kunnen worden tijdens het vervolg van het project. Bij de evaluatie van de tweede upload kijken we in hoeverre de onderhoudbaarheid is verbeterd, en of de feedback is geadresseerd. Deze evaluatie heeft als doel om studenten bewuster te maken van de onderhoudbaarheid van hun code, en dient niet gebruikt te worden voor andere doeleinden.

Let tijdens het bekijken van de feedback op het volgende:

- Het is belangrijk om de feedback in de context van de huidige onderhoudbaarheid te zien. Als een project al relatief hoog scoort zijn de genoemde punten niet 'fout', maar aankopingspunten om een nog hogere score te behalen. In veel gevallen zullen dit marginale verbeteringen zijn, grote verbeteringen zijn immers moeilijk te behalen als de code al goed onderhoudbaar is.
- Voor de herkenning van testcode maken we gebruik van geautomatiseerde detectie. Dit werkt voor de gangbare technologieën en frameworks, maar het zou kunnen dat we jullie testcode hebben gemist. Laat het in dat geval vooral weten, maar we vragen hier ook om begrip dat het voor ons niet te doen is om voor elk groepje handmatig te kijken of er nog ergens testcode zit.
- Hetzelfde geldt voor libraries: als er voldaan wordt aan gangbare conventies worden die automatisch niet meegenomen tijdens de analyse, maar ook hier is het mogelijk dat we iets gemist hebben.

Mochten er nog vragen of opmerkingen zijn dan horen we dat graag.

Met vriendelijke groet,
Dennis Bijlsma

[Feedback]

De code van het systeem scoort 4.1 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door lagere scores voor Unit Size en Unit Interfacing.

Bij Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Dit kan verschillende redenen hebben, maar de meest voorkomende is dat een methode te veel

functionaliteit bevat. Vaak was de methode oorspronkelijk kleiner, maar is deze in de loop van tijd steeds verder uitgebreid. De aanwezigheid van commentaar die stukken code van elkaar scheiden is meestal een indicator dat de methode meerdere verantwoordelijkheden bevat. Het opsplitsen van dit soort methodes zorgt er voor dat elke methode een duidelijke en specifieke functionele scope heeft. Daarnaast wordt de functionaliteit op deze manier vanzelf gedocumenteerd via methodenamen.

Voorbeelden in jullie project:

- `ArbitrationFactory.Create(string)`
- `ClockService.EndGracePeriod()`
- `ClockService.PlaceBid(string, string, DateTime, int, int)`
- `useInternalClockData.ts.useInternalClockData`
- `useClock.ts.useClock`

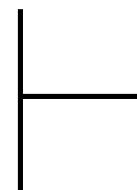
Voor Unit Interfacing wordt er gekeken naar het percentage code in units met een bovengemiddeld aantal parameters. Doorgaans duidt een bovengemiddeld aantal parameters op een gebrek aan abstractie. Daarnaast leidt een groot aantal parameters nogal eens tot verwarring in het aanroepen van de methode en in de meeste gevallen ook tot langere en complexere methoden. Dit kan worden opgelost door parameter-objecten te introduceren, waarbij een aantal logischerwijs bij elkaar horende parameters in een nieuw object wordt ondergebracht. Dit geldt ook voor constructors met een groot aantal parameters, dit kan een reden zijn om de datastructuur op te splitsen in een aantal datastructuren. Als een constructor bijvoorbeeld acht parameters heeft die logischerwijs in twee groepen van vier parameters bestaan, is het logisch om twee nieuwe objecten te introduceren.

Voorbeelden in jullie project:

- `ClockService.ClockService(IOptionsMonitor, IAuctionCacheService, IArbitrationService, IMessagingService, ILogger)`
- `PriceHelper.GetPriceAtTime(DateTime, DateTime, int, int, int)`
- `clockApiClient.ts.throwException`

De aanwezigheid van testcode is in ieder geval veelbelovend. De hoeveelheid testcode ziet er ook goed uit, hopelijk lukt het om naast toevoegen van nieuwe productiecode ook nieuwe tests te blijven schrijven.

Over het algemeen scoort de code dus bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase.



Info sheet

The following page contains the info sheet used to provide a short overview of the client company, the project and the team members.

(the remainder of this page is intentionally left blank)

Title of the project: Creating an online auctioning clock

Name of the client organization: JEM-id

Date of the final presentation: 2nd July 2019

Description

JEM-id is a software development company which focuses on the agricultural sector. We were tasked with creating a proof of concept of a new online auction for Royal FloraHolland as a web application by JEM-id. The pre-existing digital auction is not available as a web application and has generated technical debt over the past twenty years of its existence.

The main challenge of the project was to make sure the application is capable of sufficiently handling the current load of the auction while maintaining a similar performance (<30 ms stable round trip time for clients within the Netherlands). On top of that, the system had to be scalable to support higher numbers of buyers in the future. We used a microservice architecture able to balance the load over several servers to resolve this. An unexpected challenge arose when we wanted to start automatically deploying certain services when they were required. We lacked the time and resources to solve this issue correctly, so we decided to use continuous deployment for the production services and manual deployment for the simulation services.

During the research phase, the main focus was finding the best suitable technologies for creating the different services and the communication between them. This resulted in us using ASP.NET Core for all backend servers, using HTTP requests for getting the initial state of the auction and using SignalR for all subsequent communication.

The created product contains multiple backend services and a web frontend. Additionally, we created service and client applications, which can simulate an entire auction on a running clock instance with realistic buyer and auctioneer behaviour. The simulation was used to test the correctness and performance of the final product. This final product has most of the functionalities of the pre-existing digital auction and could be developed further to include missing features. We concluded that it is feasible to replace the current digital auctioning clock with a web application.

Members

W.J. Baartman (wesley_jay@msn.com)

Interests: Dynamic code generation, software testing, static code analysis, software architecture

Contributions: Clock logic, RTCS logic, headless simulation clients, backend code quality assurance

C. Lemaire (chris.lemaire@telfort.nl)

Interests: Distributed architectures, open APIs, programming languages

Contributions: Data flows, service deployment, simulation coordination

M.J. van Tartwijk (m.j.vantartwijk@protonmail.com)

Interests: Data analysis, algorithmics, artificial intelligence

Contributions: Frontend development, simulation scheduling, preparing presentations

P.W.P. van der Stel (pvdstel@outlook.com)

Interests: Full-stack development, software architecture, software engineering

Contributions: Frontend development, linking frontend and backend, frontend code quality assurance

J.C. de Vries (carendevries@live.nl)

Interests: Statistical analysis, problem solving, design

Contributions: Frontend development, auction data statistical analysis, distributions for simulation

Contact Information

Client	Robin van den Broek	robin@jem-id.nl
	Jill van der Knaap	jill@jem-id.nl
Coach	Dr. A. Panichella	a.panichella@tudelft.nl

The final report for this project can be found at: <https://repository.tudelft.nl>.

Bibliography

- [1] D. Workman. (Oct. 2018). Flower bouquet exports by country, [Online]. Available: <http://www.worldstopexports.com/flower-bouquet-exports-country/> (visited on 04/26/2019).
- [2] Guinness World Records, *Largest flower auction*. [Online]. Available: <https://www.guinnessworldrecords.com/world-records/largest-flower-auction> (visited on 05/01/2019).
- [3] *Jem-id*. [Online]. Available: <https://www.jem-id.nl/en> (visited on 05/01/2019).
- [4] *Floriday*. [Online]. Available: <https://www.floriday.io/en> (visited on 04/29/2019).
- [5] Royal FloraHolland, *Remote buying*. [Online]. Available: <https://www.royalfloraholland.com/en/buying/marketplace/auction/remote-buying-nod4756> (visited on 04/29/2019).
- [6] Bloemenveiling Aalsmeer, Oct. 2002. [Online]. Available: https://web.archive.org/web/20021011101233/http://www.vba.nl:80/nieuws.asp?tal_id=1&MainMenu=Home (visited on 04/29/2019).
- [7] Floriday. (2018). Over floriday., [Online]. Available: <https://www.floriday.io/nl/faq/over-floriday> (visited on 04/29/2019).
- [8] Microsoft Inc., *Framework design guidelines*. [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/> (visited on 06/13/2019).
- [9] B. Ng, N. Ribaldo, loveky, V. Rangaraj, T. L. Hau, P. Boere, L. Smyth, J. Xi, H. Wolff, C. Chan, H. Zhu, and A. Lizurchik, *@babel/preset-env*, 2019. [Online]. Available: <https://babeljs.io/docs/en/babel-preset-env> (visited on 06/24/2019).
- [10] J. Lewis and M. Fowler. (Mar. 2014). Microservices, [Online]. Available: <https://martinfowler.com/articles/microservices.html> (visited on 06/17/2019).
- [11] R. Fielding and J. Reschke, *Hypertext transfer protocol (http/1.1): Semantics and content*, Jun. 2014. [Online]. Available: <https://tools.ietf.org/html/rfc7231> (visited on 06/17/2019).
- [12] Microsoft Inc., *Guid struct (system)*, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/api/system.guid?view=netstandard-2.0> (visited on 06/18/2019).
- [13] Identity Server, *Protecting an API using Client Credentials*, 2016. [Online]. Available: http://docs.identityserver.io/en/latest/quickstarts/1_client_credentials.html (visited on 06/23/2019).
- [14] International Organization for Standardization, *ISO 8601-1:2019*, 2019. [Online]. Available: <https://www.iso.org/obp/ui#iso:std:iso:8601:-2:ed-1:v1:en> (visited on 06/23/2019).
- [15] OpenID, *Welcom to openid connect*. [Online]. Available: <https://openid.net/connect/> (visited on 06/14/2019).
- [16] D. Hardt, *The OAuth 2.0 authorization framework*. [Online]. Available: <https://tools.ietf.org/html/rfc6749> (visited on 06/14/2019).
- [17] M. Jones, J. Bradley, and N. Sakimura, *Json web token (jwt)*. [Online]. Available: <https://tools.ietf.org/html/rfc6749> (visited on 06/14/2019).
- [18] ———, *Json web signature (jws)*. [Online]. Available: <https://tools.ietf.org/html/rfc7515> (visited on 06/14/2019).
- [19] M. Jones and J. Hildebrand, *Json web encryption (jwe)*. [Online]. Available: <https://tools.ietf.org/html/rfc7516> (visited on 06/14/2019).
- [20] N. Barbettini, *Oauth 2.0 and openid connect*. [Online]. Available: <https://github.com/okta/okta-developer-docs/blob/b02736bf7fba31d05bb94c68a6e8e5676fd84f2d/packages/%40okta/vuepress-site/docs/concepts/auth-overview/index.md> (visited on 06/14/2019).
- [21] J. Noor, M. G. Hossain, M. A. Alam, A. Uddin, S. Chellappan, and A. B. M. A. Al Islam, “svLoad: An Automated Test-Driven Architecture for Load Testing in Cloud Systems”, in *2018 IEEE Global Communications Conference (GLOBECOM)*, Dec. 2018, pp. 1–7. DOI: [10.1109/GLOCOM.2018.8647493](https://doi.org/10.1109/GLOCOM.2018.8647493).

- [22] Microsoft, *Define and solve a problem by using solver*. [Online]. Available: <https://support.office.com/en-ie/article/define-and-solve-a-problem-by-using-solver-5d1a388f-079d-43ac-a7eb-f63e45925040> (visited on 06/13/2019).
- [23] Wikipedia, *Geometric distribution*. [Online]. Available: https://en.wikipedia.org/wiki/Geometric_distribution (visited on 06/13/2019).
- [24] C. Leys, C. Ley, O. Klein, P. Bernard, and L. Licata, "Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median", *Journal of Experimental Social Psychology*, vol. 49, no. 4, pp. 764–766, 2013.
- [25] Y. Dodge, *The Oxford dictionary of statistical terms*. Oxford University Press on Demand, 2006.
- [26] Agile Alliance, *Agile 101*, 2019. [Online]. Available: <https://www.agilealliance.org/agile101/> (visited on 06/24/2019).
- [27] Atlassian, *Trello*, 2019. [Online]. Available: <https://trello.com/> (visited on 06/24/2019).
- [28] Microsoft Inc., *Azure DevOps*, 2019. [Online]. Available: <https://azure.microsoft.com/en-us/services/devops/> (visited on 06/24/2019).
- [29] —, *Web Apps*, 2019. [Online]. Available: <https://azure.microsoft.com/en-us/services/app-service/web/> (visited on 06/24/2019).
- [30] K. Havens, S. Boyer, M. Linville, W. Jie, S. Smith, paigehargrave, O. Burov, and C. Floyd, Jun. 2019. [Online]. Available: <https://github.com/docker/docker.github.io/blob/master/engine/examples/dotnetcore.md> (visited on 06/24/2019).
- [31] SonarSource S.A, *SonarQube*, 2019. [Online]. Available: <https://www.sonarqube.org/> (visited on 06/24/2019).
- [32] Software Improvement Group, *SIG | Software Improvement Group*, 2019. [Online]. Available: <https://www.softwareimprovementgroup.com/> (visited on 06/24/2019).
- [33] Microsoft Azure, *Azure SignalR Service*, 2019. [Online]. Available: <https://azure.microsoft.com/services/signalr-service/> (visited on 06/25/2019).
- [34] Royal FloraHolland, "Jaarverslag 2018", 2018. [Online]. Available: <https://jaarverslag2018.royalfloraholland.com/wp-content/uploads/2019/04/Royal-FloraHolland-%E2%80%93-Jaarverslag-2018.pdf> (visited on 04/25/2019).
- [35] —, "Jaarverslag 2014", 2014. [Online]. Available: <https://www.royalfloraholland.com/media/3944533/Jaarverslag-2014.pdf> (visited on 04/25/2019).
- [36] H. (M. Truong, W. Ketter, A. (Gupta, and E. van Heck, "Effects of Pre-sales Posted Price Channel on Sequential B2B Dutch Flower Auctions", Jan. 2018. [Online]. Available: <http://hdl.handle.net/1765/104630>.
- [37] Royal FloraHolland, *Annual report 2018*, 2018. [Online]. Available: http://jaarverslag.royalfloraholland.com/#/feiten-en-cijfers/omzet?_k=d7coyz (visited on 04/29/2019).
- [38] —, *Annual report 2015*, 2015. [Online]. Available: https://www.royalfloraholland.com/media/7940281/Flora-Holland-Jaarverslag-2015-NL_1040538.compressed.pdf (visited on 04/29/2019).
- [39] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes", *ACM Queue*, vol. 14, pp. 70–93, 2016. [Online]. Available: <http://queue.acm.org/detail.cfm?id=2898444>.
- [40] B. Familiar, *Microservices, IoT, and Azure*. Springer, 2015. DOI: 10.1007/978-1-4842-1275-2.
- [41] *Express*. [Online]. Available: <https://expressjs.com/> (visited on 04/29/2019).
- [42] Microsoft, *ASP.NET*. [Online]. Available: <https://dotnet.microsoft.com/apps/aspnet> (visited on 04/29/2019).
- [43] —, *Choose between ASP.NET 4.x and ASP.NET Core*, 2018. [Online]. Available: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/choose-aspnet-framework?view=aspnetcore-2.2> (visited on 04/29/2019).
- [44] X. Mao, "Comparison between Symfony, ASP.NET MVC, And Node.js Express for Web Development", Apr. 2018. [Online]. Available: <https://hdl.handle.net/10365/28191>.

- [45] TechEmpower, *Web framework benchmarks - round 17 results - plaintext*, 2018. [Online]. Available: <https://www.techempower.com/benchmarks/#section=data-r17&hw=cl&test=plaintext> (visited on 04/29/2019).
- [46] —, *Web framework benchmarks - round 17 results - json serialization*, 2018. [Online]. Available: <https://www.techempower.com/benchmarks/#section=data-r17&hw=cl&test=json> (visited on 04/29/2019).
- [47] .NET Foundation, *IronPython*, 2018. [Online]. Available: <https://ironpython.net/> (visited on 06/23/2019).
- [48] F. Holmström, *IronJS*, 2013. [Online]. Available: <https://github.com/fholm/IronJS> (visited on 06/23/2019).
- [49] IronRuby community, *IronRuby*, 2011. [Online]. Available: <http://ironruby.net/> (visited on 06/23/2019).
- [50] P. Phillips, *Expunged .net backend*, 2012. [Online]. Available: <https://github.com/scala/scala/pull/1718> (visited on 06/23/2019).
- [51] *TIOBE Index for April 2019*, 2019. [Online]. Available: <https://www.tiobe.com/tiobe-index/> (visited on 04/29/2019).
- [52] *PYPL Popularity of Programming Language*, 2019. [Online]. Available: <http://pypl.github.io/PYPL.html> (visited on 04/29/2019).
- [53] C. Zapponi, *GitHub 2.0 year 2019 quarter 1*. [Online]. Available: https://madnight.github.io/github/#/pull_requests/2019/1 (visited on 04/29/2019).
- [54] D. Delimarsky, *Visual F#*, 2017. [Online]. Available: <https://msdn.microsoft.com/visualfsharpdocs/conceptual/visual-fsharp> (visited on 04/29/2019).
- [55] Microsoft, *Microsoft test framework mstest v2*. [Online]. Available: <https://github.com/Microsoft/testfx> (visited on 04/29/2019).
- [56] C. Poole and R. Prouse, *What is nunit?* [Online]. Available: <https://nunit.org/> (visited on 04/29/2019).
- [57] *About xUnit.net*. [Online]. Available: <https://xunit.net/> (visited on 04/29/2019).
- [58] *Comparing xUnit.net to other frameworks*. [Online]. Available: <https://xunit.net/docs/comparisons> (visited on 04/29/2019).
- [59] T. Inc., *TypeMock .NET Annual Pricing*. [Online]. Available: <https://www.typemock.com/pricing/net/annual> (visited on 04/29/2019).
- [60] *RhinoMocks*, 2014. [Online]. Available: <https://www.nuget.org/packages/rhinomocks/> (visited on 06/23/2019).
- [61] P. Phillips, *Foq*, 2018. [Online]. Available: <https://github.com/fsprojects/Foq> (visited on 06/23/2019).
- [62] *StyleCop Analyzers for the .NET Compiler Platform*. [Online]. Available: <https://github.com/DotNetAnalyzers/StyleCopAnalyzers> (visited on 04/29/2019).
- [63] M. Rajesh Kulkarni P.Padmanabham, "Improving Software Quality Attributes of PS Using Stylecop", *Global Journal of Computer Science and Technology*, vol. 13, no. 8, 2013, ISSN: 0975-4172.
- [64] *Roslyn analyzers*. [Online]. Available: <https://github.com/dotnet/roslyn-analyzers> (visited on 04/29/2019).
- [65] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y. Parareda, and M. Pizka, "Tool support for continuous quality control", *IEEE Software*, vol. 25, no. 5, pp. 60–67, Sep. 2008, ISSN: 0740-7459. DOI: 10.1109/MS.2008.129.
- [66] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery", *ACM Computing Surveys (CSUR)*, vol. 15, no. 4, pp. 287–317, 1983. DOI: 10.1145/289.291.
- [67] I. Fette and A. Melnikov, *RFC 6455 - The WebSocket Protocol*, Dec. 2011. [Online]. Available: <https://tools.ietf.org/html/rfc6455> (visited on 04/30/2019).
- [68] I. Grigorik, *High Performance Browser Networking: What every web developer should know about networking and web performance*. O'Reilly Media, Inc., 2013.

- [69] Microsoft, *Real-time ASP.NET with SignalR*. [Online]. Available: <https://dotnet.microsoft.com/apps/aspnet/real-time> (visited on 04/30/2019).
- [70] A. Choudhry and A. Premchand, "Real time apps using signalr", *International Journal of Computer Trends and Technology (IJCTT)*, vol. 15, no. 3, pp. 92–96, 2014.
- [71] M. Whited, *Signalr-client*, 2018. [Online]. Available: <https://www.npmjs.com/package/signalr-client> (visited on 04/30/2019).
- [72] *.NET Core Runtime (CoreRT)*. [Online]. Available: <https://github.com/dotnet/corert> (visited on 04/30/2019).
- [73] M. Mikowski and J. Powell, *Single page web applications: JavaScript end-to-end*. Manning Publications Co., 2013.
- [74] J. Zaytsev, *Ecmascript 6 compatibility table*. [Online]. Available: <https://kangax.github.io/compat-table/es6/> (visited on 04/29/2019).
- [75] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for javascript", in *Static Analysis*, J. Palsberg and Z. Su, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 238–255, ISBN: 978-3-642-03237-0.
- [76] G. Bierman, M. Abadi, and M. Torgersen, "Understanding typescript", in *European Conference on Object-Oriented Programming*, Springer, 2014, pp. 257–281.
- [77] B. Kelly, P. Goel, and F. Elmas, *Typescript in 5 minutes*. [Online]. Available: <https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html> (visited on 04/29/2019).
- [78] J. Potter, *Cypress vs karma vs nightwatch vs protractor vs selenium-webdriver*. [Online]. Available: <https://www.npmtrends.com/cypress-vs-karma-vs-nightwatch-vs-protractor-vs-selenium-webdriver> (visited on 04/30/2019).
- [79] cypress-io, *Cypress-io/cypress - fast, easy and reliable testing for anything that runs in a browser*. [Online]. Available: <https://github.com/cypress-io/cypress> (visited on 04/30/2019).
- [80] SeleniumHQ, *Seleniumhq/selenium - a browser automation framework and ecosystem*. [Online]. Available: <https://github.com/SeleniumHQ/selenium> (visited on 04/30/2019).
- [81] —, *Selenium - web browser automation*. [Online]. Available: <https://www.seleniumhq.org/> (visited on 04/30/2019).
- [82] L. Latinov, *Cypress vs. selenium, is this the end of an era?*, Jun. 2018. [Online]. Available: <https://automationrhapsody.com/cypress-vs-selenium-end-era/> (visited on 04/30/2019).
- [83] A. McPeak, *Selenium vs. cypress: Is webdriver on its way out?*, Jun. 2018. [Online]. Available: <https://crossbrowsertesting.com/blog/test-automation/selenium-vs-cypress/> (visited on 04/30/2019).
- [84] Cypress, *How it works*. [Online]. Available: <https://www.cypress.io/how-it-works/> (visited on 04/30/2019).
- [85] —, *Roadmap*. [Online]. Available: <https://docs.cypress.io/guides/references/roadmap.html#Upcoming-Features> (visited on 04/30/2019).
- [86] *Webdriver*. [Online]. Available: <https://www.w3.org/TR/webdriver/>.
- [87] K. Naik and P. Tripathy, *Software Testing and Quality Assurance: Theory and Practice*. Wiley Online Library, 2008, p. 214, ISBN: 978-0-471-78911-6.