Waiting-as-a-Service

Handling ash crowds in dynamic online shopping environments with limited, complex stocks in a fair manner

Pluijmaekers, M.H.L.

Award date:
2020

Waiting-as-a-Service

Handling flash crowds in dynamic online shopping environments with limited, complex stocks in a fair manner

M.H.L. Pluijmaekers, BSc.
m.h.l.pluijmaekers@student.tue.nl
Eindhoven, August 2020

Department of Mathematics and Computer Science
System Architecture and Networking Research Group
Eindhoven University of Technology

Contents

Abstract

Over the last few years, the number of events has grown, and the existing events have been becoming increasingly more popular. The companies providing online platforms for event organizers to sell tickets therefore have to deal with an ever increasing load. The needs Eventix, who offer such a platform, are of specific focus.

Highly popular events attract flash crowds, a large and sudden influx of customers interested in purchasing a subset of the offered products. Resulting in demand for a product far exceeding supply. Tickets have a strict limit on supply, overselling or back ordering tickets is not acceptable. Often tickets are actually a combination of multiple products, each having their own strictly limited stock and are subject to constraints on purchasable amounts.

In this thesis, an architecture is presented which can handle flash crowds and keeping track of stock of compound products according to the values of event organizers. Organizers have varying viewpoints on how to handle a flash crowd, many of which are supported by the architecture. A proof-of-concept of the architecture deployable on a Kubernetes cluster is created and evaluated. The results of the evaluation indicate that even the proof-of-concept functions as expected. The designed architecture should be suitable to solve the issues. The throughput of the proof-of-concept is found to be small. It is recommended Eventix researches methods of improving throughput before implementing a more feature complete implementation.

Preface

Before you lies a thesis titled “Waiting-as-a-Service - Handling flash crowds in dynamic online shopping environments with limited, complex stocks in a fair manner”. It has been written as part of my graduation project for the Computer Science and Engineering Master at the Eindhoven University of Technology.

The project was undertaken at the request of Eventix, my current (and hopefully future) employer. The project itself was difficult, many iterations of both the thesis and the code were needed in order to achieve what I hope is at least a passible graduation project. But my advisors R.H. Mak and C. de Jong provided help and constructive criticism when needed or requested, even when I did not want to hear it. To them both I wish to extend my deepest appreciation and gratitude. Another person deserving of gratitude is N. Sidorova, for being the third member of the assessment committee.

Lastly I would also like to thank my girlfriend, parents, friends, and colleagues that read my thesis, provided feedback, have been supportive, and put up with me (and my rants) over the last few months. You always supported me and kept me motivated.

Mart Pluijmaekers
Eindhoven, August 2020

Chapter 1

Introduction

In this thesis an architecture for handling flash crowds (large, suddenly appearing crowds) and handling products with complex stocks (products containing products) will be presented. First, some context on why this thesis exists is given. Next, an analysis of the problem will be presented. In further chapters, the architecture will be presented and verified. Finally, the conclusions will be presented including recommendations on the next steps to be taken.

1.1 Context

The (online) ticketing industry has evolved into providers servicing any type of organization that want to sell any type of tickets. Whether these tickets are for transportation, cultural events, social events, conferences, or something different. Most providers are equipped to sell anything that can be sold as tickets.

The needs of one specific provider, Eventix, will be of particular focus. Eventix is a fast growing company focused on delivering an automated and easy to use multi-tenant self-service ticketing system. Founded in 2014, it has maintained a steady growth ever since. Eventix strives to create a complete toolbox of features for successful marketing, management, ticket sales, and growth of any event, regardless of the size of the event.

Eventix focuses on automation, scalability, and quality service. Through Eventix's systems, their customers (event organizers) sell tickets for events large and small. They purchase Eventix's ticketing services to service their own customers, the audience. After having setup their events, organizers either embed a shop containing their tickets into their website, or point their intended audience to a url where the shop can be found. Eventix does not offer, nor is interested in offering, any search option to find events, all events for which they sell tickets are marketed by the organizers themselves, Eventix only wants to be the middle-man.

In the system Eventix provides, four distinct groups of stakeholders are recognized.

Audience, those visiting a ticket shop and purchasing tickets.

Organizers, those who organize and manage ticket sales to sell tickets to their audience. They are the ones purchasing Eventix's services.

Administrators, those who manage (scale) and monitor the systems, typically employees of Eventix.

Payment Service Providers (PSPs), entities that handle payments on Eventix's behalf.

Using the dashboard offered by Eventix, Organizers create products which Eventix calls tickets. Within these tickets, sub-products are offered, which Eventix calls products. During the shopping process, customers compose their tickets using the products offered in these tickets. Consequently, different instances of the same ticket can contain completely different sets of products. Validity of a ticket is important, each ticket is only valid once. Otherwise, multiple people could use the same ticket to gain access to some event. To achieve this Eventix adds a unique identifier to each ticket. Even tickets sold with products are still identified by a single identifier. Eventix offers a system which cannot only be used to check whether a ticket has been used, but also whether the products of a ticket has been used.

An example usecase is an event with limited parking availability. When organizers create the ticket which provides access to their event, they can create a product which gives access to parking services once and add it to the ticket. When customers then purchase tickets, they can decide on the number of times they want to park their car and add the "parking product" the appropriate number of times. When a customer purchases multiple tickets, one for each of their friends for instance, the "parking product" can be added a different amount of times per ticket. Consequently, it is important to keep track of specific instances of tickets and for each ticket instance the specific amount of products of each product type attached to the ticket instance.

The created tickets can then be linked to organizer created shops. Eventix does not impose any limits on the number of tickets attached to a shop or to the amount of shops offering the same ticket. Nor does it impose limits on the products. These shops operate like a frontend to the virtual warehouse of the event organizers's tickets.

There is no homogeneous manner in which organizers setup their shops. In general, there are three ways organizers setup their shops:

- Few large shops, through which tickets for multiple events are sold. When new tickets (other than those already on offer) need to be offered, these shops get updated. These are generally used for yearly recurring or one-off events, like music festivals.
- Many small shops, which get discarded after they have served their purpose. These are generally used for often recurring events, like touring theater companies that move around to different locations.
- One small shop, for one-off events where there is only a single or very limited amount of ticket types available.

There are also many organizers which use a mix, e.g. having many smaller shops which they actively update. This might be due to selling many events where every once in a while a new type of ticket becomes available for sale or simply personal preference. According to Eventix, this is one of their main features. This has resulted in an architecture where all shops are sharing the same resources, making it a multi-tenant system. For stock handling, Eventix currently relies on a non-atomic procedure for issuing reservations consisting of the following actions:

1. Create a reservation for the ticket, decreasing availability.
2. Check availability for the ticket.
3. If availability does not suffice, release ticket.
4. Recursively execute this procedure for all required sub-products.
5. If for any sub-product the availability does not suffice, release all reservations.
6. When all availabilities suffice, hand out all reservations.

Because Eventix, in contrast to some of their competitors, chose to focus on just ticketing, it uses external entities for handling payments, the PSPs (payment service providers). Therefore, they rely on these parties when servicing the audience on behalf of organizers.

There are multiple issues with the current architecture, mostly related to reliability and scalability. These become painfully apparent when Eventix is experiencing a flash crowd. For these descriptions a "customer" will denote the shop visitor trying to purchase tickets.

Payments

As is to be expected with any service, degraded performance and even downtime is to be expected. Currently, when a PSP is experiencing degraded performance or downtime Eventix cannot prevent customers from accessing the payment services, resulting in an apparent performance degradation or downtime for Eventix's system. This is a main cause of the apparent unreliability of the system.

Reservations

When the number of concurrent requests, requesting a reservation for a ticket exceeds availability, the current system will show nondeterministic behavior. The current process of issuing a reservation consists of decrementing the stock, checking whether the remaining stock is greater or equal to zero. If it is greater than zero, a reservation is generated and returned to the customer. If it is less than zero, the stock is incremented and an error is returned to the customer.

Depending on the exact interleaving of the described actions, customers can experience varying behavior. For instance, an interleaving exists where no customer is able to obtain a reservation for a product with a stock lower than the demand: for all requests the first action has been executed (lowering the stock to below zero), when the second action (the stock check) occurs, it will result in no issued reservations since for no customer the stock of the ticket was high enough. While in a different interleaving, it can occur that the number of reservations handed out is exactly equal to the stock: when every request happens to be handled in some neat order where no requests are interleaved. Many more interleavings exist, resulting in nondeterministic behavior.

Admission

Currently, no admission proper control exists, any customer that arrives at Eventix's systems is admitted immediately, regardless of current load. A system is in place which admits customers in a non-intelligent fashion, but this system needs to be manually operated. Since the systems are multi-tenant, one customer experiencing degraded performance due to load implies many if not all customers are experiencing degraded performance. The only way to currently mitigate these issues is by increasing total server capacity.

In general a sudden, large influx of customers (called a flash crowd) only occurs when new tickets become available, making manual flash crowd mitigation manageable, since it is known beforehand when tickets become available allowing for manual scaling. In general, shop owners (organizers in this case) only want a relatively small number of concurrent shoppers, to prevent the issues described in Section 1.1. Making flash crowd mitigation by manually increasing server capacity to handle the increase in customers instead of the desired number of concurrently shopping customers both undesirable and wasteful of resources.

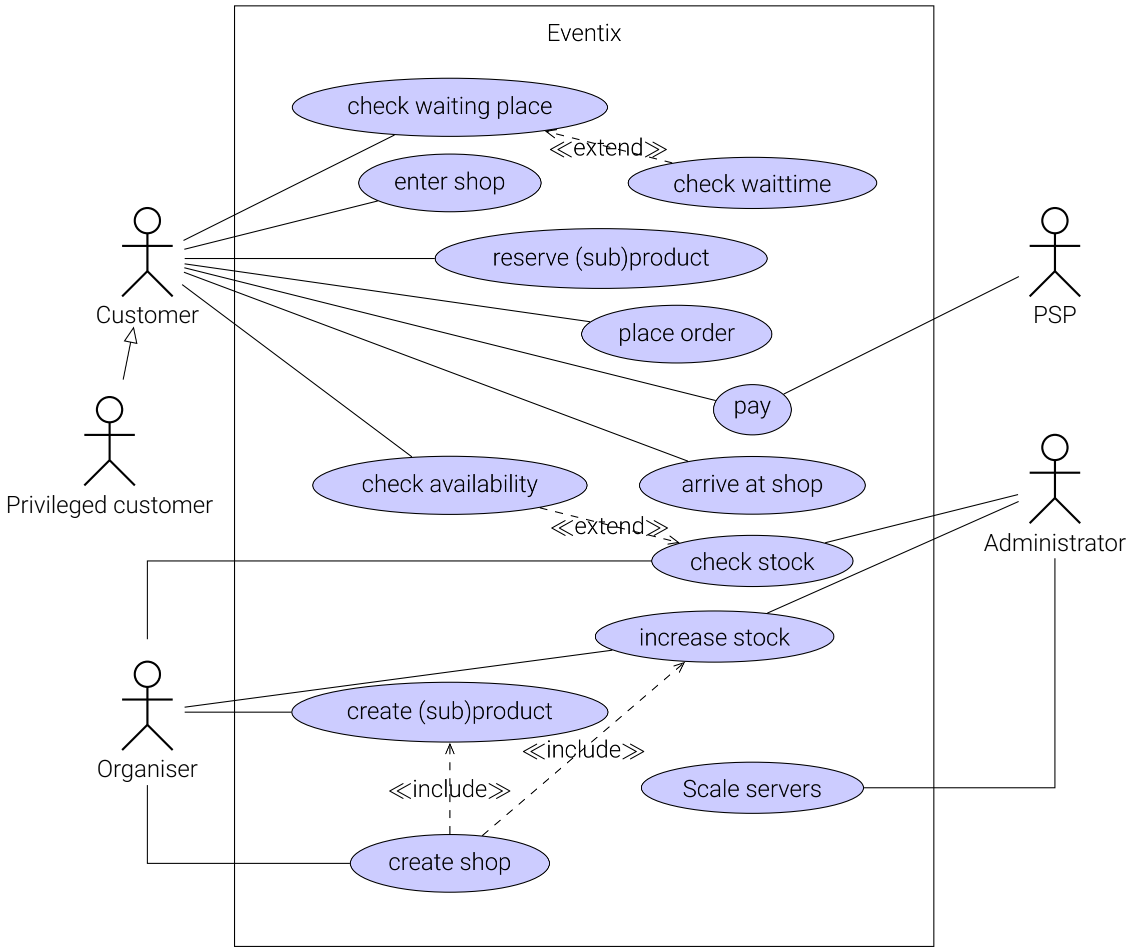


Figure 1.1: Usecase diagram for the current Eventix system.

1.2 Domain analysis

For physical shops, the structure of a shop visitation (shopping run) is commonly known and can be roughly described by following procedure:

1. Arrive at the shop.
2. Enter the shop, becoming an "active" customer.
3. Choose (browsing for) products to purchase.

4. Place an available chosen product in the (virtual) basket (reserving of product).
5. Go to checkout to attempt payment.
6. Pay for the chosen products.
7. Receive purchased products or leave without products.

Steps 3 and 4 are repeated until all desired products have been handled. In every step, the customer is receiving service, either by physical attributes of the shop like layout and product availability or by interactions with staff like asking for advice or paying. Figure 1.2 describes this process. The flowchart shows only the basic process, a more extensive process would, for instance, include placing back product from the basket and a decision on whether to return product to stock after a number of failed payments. All dashed diamonds are decisions that are taken by either the customer or an external entity, whereas all solid diamonds are decisions taken by the shop operator.

Online shops try to emulate the shopping experience of physical shops and in doing so they have one main advantage over their physical counterparts, they (seem to) have unlimited stock. This enables customers to shop without worrying whether they will receive their desired goods. A customer can browse the offered goods, place them in a (virtual) shopping basket, submit some personal information like name and shipping address, and finally, pay. When products are in stock, the experience is the same as in physical shops, the product can be shipped immediately. When stock for an item has been depleted however, an order will be placed on backorder and the goods will be shipped after stocks have been replenished. This works due to most online shops selling goods they can restock.

Other shops, however, are different, they sell goods that cannot be restocked. These include shops selling: (airline-, theater-, festival-, etc.) tickets, sneakers, high fashion, etc., any business selling items with a limited supply. This situation is exacerbated due to some of these industries having to deal with “flash crowds”, a very high interest in offered products, which likely exceeds supply.

In these cases, they have to regulate the shopping behavior of their customers, which explains why the flowchart contains two decision boxes that limit shopping, one from the perspective of the customer and the other from the perspective of the shop. The solid “Can continue shopping?” decision box is for the shop/system to decide whether a customer can continue shopping. The dashed “Finished shopping?” box is for customers to signal they have finished browsing.

A recent development, especially in the ticketing industry, is selling compound products. These are products that a customer can compose from multiple sub-products, and therefore have to take into account not only their own stock, but also stock of the sub-products they encompass. An example of this can be found in Table 1.1. The example depicts three products a, b, c, each encompassing two sub-products $p_i, 1 \leqslant i \leqslant 3$, with only a, b, c available for direct purchase. The table shows both the stock and the maximum number of available product, since any product has a certain stock which cannot be exceeded, the maximum availability is the maximum amount of this product that can be purchased i.e. the minimum of the stock of the product and all of its sub-products.

Table 1.2 shows that purchasing c, 25 times in this case, also influences the number of available a, since both contain p_1. Furthermore, the total number of sellable products can vary wildly, for example: purchasing either $5a + 65b$ or $70b + 5c$ extra ensures that no further products are available anymore (Tables 1.3, 1.4) even though stocks still might not have been depleted. This shows that simply counting stock and decrementing availability is not enough, since stocks are not fully independent.

The previous example is still quite simple, since the sub-products are required. When optional sub-products are allowed, products which can be optional in one product but that might be required in another, the demonstrated stock counting issues worsen. Table 1.5 shows an example of this effect, it shows two distinct options when purchasing x, since q_2 might be added.

x without q_2, only a_x, a_{q_1} decrease.

x with q_2, both a_x, a_{q_1} and a_y, a_{q_2} decrease.

Furthermore, payment success is not guaranteed for any shop, but especially problematic for online shops. These need to make a choice, either wait for payment success before checking availability and backorder products with insufficient stock, refund customers when stocks are insufficient, or reserve products before the payment is attempted and hand back the reservations on payment failure. Different situations fit different options, this is dependant on the exact requirements of the shop.

Another complication is dealing with stock increases. During normal sales, stock is only expected to decrease from the time the shop opens until it closes or has sold out its products. Stock increases can occur, however,

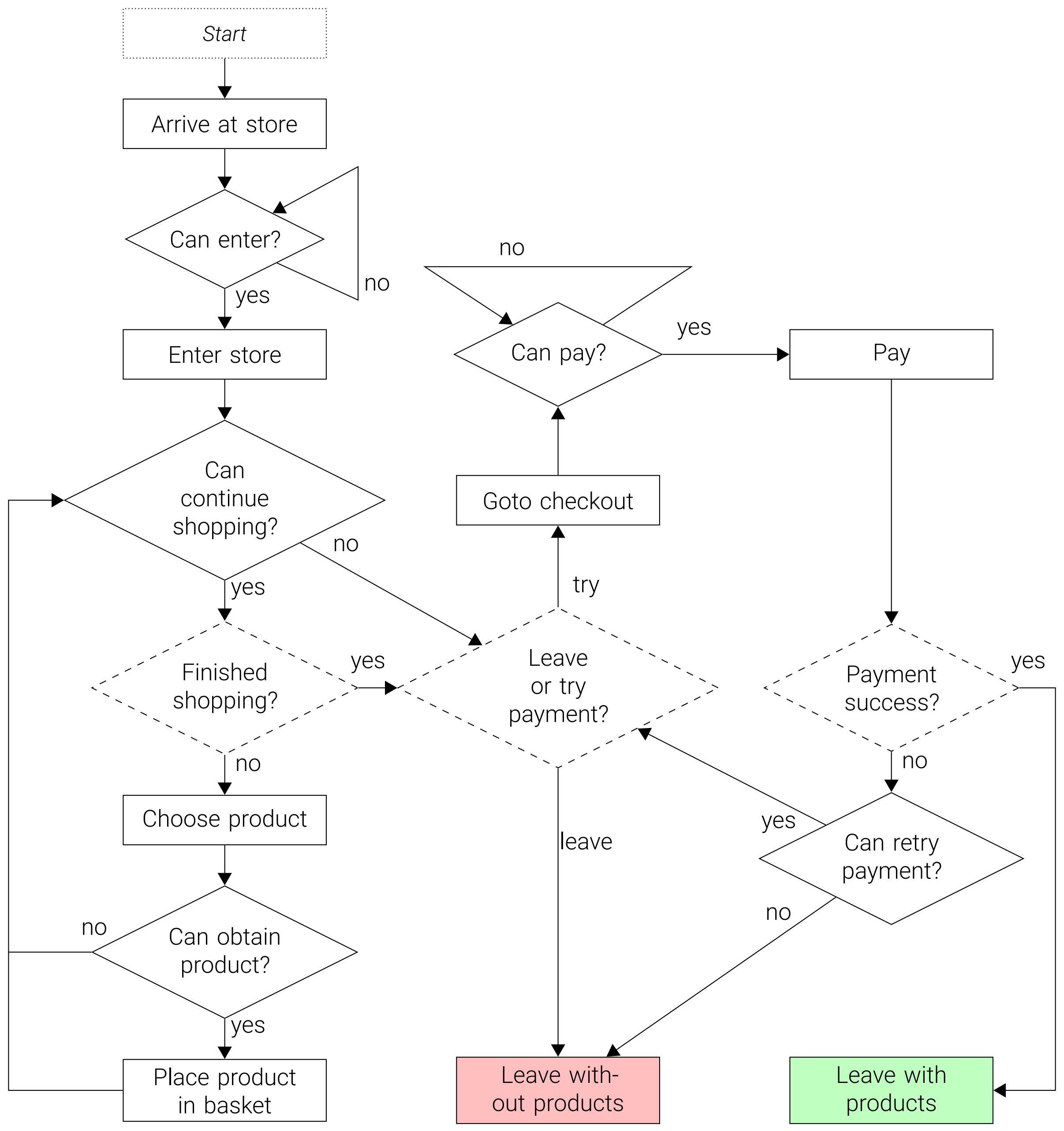


Figure 1.2: Flowchart showing basic customer actions in a shop.

because of two main reasons. Stock increases due to:

- Human error or more stock becoming available. e.g. stock was incorrectly setup or a shipment arrived respectively.
- Reserved products being placed back or unpaid product being returned to stock.

These stock increases also occur in physical shops, but do not specifically need to be accounted for, since there is no way of possible overselling or underselling when changes are not tracked properly, due to products still being physical items.

As discussed previously, every product has a stock, whether it is being sold in an online or in a physical shop. For chains of shops, that get supplied through a central location, stocks can be somewhat distributed. When a single shop runs out of a certain product, it can get resupplied from a centralized warehouse or even from a different shop. For online shops a similar mechanism exists. These shops are a frontend to the same warehouse, like the Dutch company Coolblue used to be. Coolblue once offered many different “stores”. Each of these stores was specialized in a single category of electronics products. Products were offered in multiple stores. These stores served as the frontend to the main Coolblue warehouse. Shopping around between frontends, when one has an out-of-stock product, is useless since frontends selling the same product are subject to the same centralized stock. Meaning stock checks are not necessarily limited to single shops, but to collections of shops.

Product	**Sub-products**	**Stock** (s)	**Max available** (a)
p_1	$\{\}$	50	50
p_2	$\{\}$	100	100
p_3	$\{\}$	200	200
a	$\{p_1, p_2\}$	100	$\min(s_a, s_{p_1}, s_{p_2}) = 50$
b	$\{p_2, p_3\}$	100	$\min(s_b, s_{p_2}, s_{p_3}) = 100$
$c.$	$\{p_1, p_3\}$	40	$\min(s_c, s_{p_1}, s_{p_3}) = 40$

Table 1.1: Example stocks for compound products.

Product	s	a	$\Delta_{s(P)}$	s'	a'
p_1	50	50	$\text{-}20_a + \text{-}25_c$	5	5
p_2	100	100	$\text{-}20_a + \text{-}10_b$	70	70
p_3	200	200	$\text{-}10_b + \text{-}25_c$	165	165
a	100	50	$\text{-}20_a$	80	$\min(s'_a, s'_{p_1}, s'_{p_2}) = 5$
b	100	100	$\text{-}10_b$	90	$\min(s'_b, s'_{p_2}, s'_{p_3}) = 70$
c	40	40	$\text{-}25_c$	15	$\min(s'_c, s'_{p_1}, s'_{p_3}) = 5$

Table 1.2: Purchase scenario: $P = 20a + 10b + 25c$.

Product	s	a	$\Delta_{s(P)}$	s'	a'
p_1	5	5	$\text{-}5_a$	0	0
p_2	70	70	$\text{-}5_a + \text{-}65_b$	0	0
p_3	165	165	$\text{-}65_b$	100	100
a	80	5	$\text{-}5_a$	75	$\min(s'_a, s'_{p_1}, s'_{p_2}) = 0$
b	90	70	$\text{-}65_b$	25	$\min(s'_b, s'_{p_2}, s'_{p_3}) = 0$
c	15	5	0	15	$\min(s'_c, s'_{p_1}, s'_{p_3}) = 0$

Table 1.3: Purchase scenario: $P = 5a + 65b$, after purchase scenario of Table 1.2.

Having just low or nonreplenishable stock is not necessarily problematic. When stocks are low, but demand is low as well, selling products still happens gradually. When there exists a low demand to stock ratio, "competition" for products is low. It can be the case that there is a sudden, large influx of demand for a product. This can be at any point in time, even before the shop has opened and therefore unknown to the shop operator! Shops announce they are selling certain products, either by an actual announcement, or through common knowledge about the types of products offered by a shop. Customers logically pick up on these announcements, so it is actually the announcement which causes demand! Due to this large unexpected influx of demand, competition for products gets fierce. The occurrence of these large crowds, called "flash crowds", at physical shops is manageable, since physical shops have physical limitations and a limited service capacity. For every customer action in Figure 1.2, a shop needs to provide a service. A customer can only purchase products that a shop offers, and a customer cannot pay for the chosen products when the shop has no cash register available. These different services can have different capacities and for some of them capacity can be increased at run time, but only to a specific limit. An example of a scalable service is paying: when more customers want to pay, a physical shop operator can make more cash registers available to customers. But at some point there are no more physical registers to make available that were previously unavailable. Conversely, shop capacity is an example of a service (concurrently serving a certain amount of customers) that cannot be increased by physical shop operators.

Consequently, not all customers can receive service at the same time, waiting is bound to happen. When flash

Product	s	a	$\Delta_{s(P)}$	s'	a'
p_1	5	5	-5_c	0	0
p_2	70	70	-70_b	0	0
p_3	165	165	$-70_b + -5_c$	90	90
a	80	5	0	80	$\min(s'_a, s'_{p_1}, s'_{p_2}) = 0$
b	90	70	-70_b	20	$\min(s'_b, s'_{p_2}, s'_{p_3}) = 0$
c	15	5	-5_c	10	$\min(s'_c, s'_{p_1}, s'_{p_3}) = 0$

Table 1.4: Purchase scenario: $P = 70b + 5c$, after purchase scenario of Table 1.2.

Product	**Required sub-products**	**Optional sub-products**	**Stock** (s)	**Max available** (a)
q_1	$\{\}$	$\{\}$	50	50
q_2	$\{\}$	$\{\}$	100	100
x	$\{q_1\}$	$\{q_2\}$	100	$\min(s_x, s_{q_1}) = 50$
y	$\{q_2\}$	$\{\}$	100	$\min(s_y, s_{q_2}) = 100$

Table 1.5: Example stocks for compound products with optional.

crowds occur, a physical shop can use limits on service capacity to handle the crowd and impose limits on its behavior. In particular, making customers wait for a service in an orderly manner. Figure 1.2 provides logical places where waiting can occur, at the solid diamonds. The reasons why waiting occurs here are either by design or physical limitations. Whenever a service is temporarily not available, whether intentional or not, the group of customers waiting is called a “waiting group”. Every waiting group needs to be handled in some way to ensure customer can complete its order.

The waiting groups focussed on are listed below, the list also contains a description or example of why waiting will occur here.

- **w1** `Can enter`, shops can have limited capacity preventing more customers entering than the building capacity.
- **w2** `Can continue shopping`, shops can have policies limiting customer shopping behavior (e.g. max. 5 items).
- **w3** `Can pay`, a physical shop has a limited number of available cash registers or the cash registers can be broken.

Note, `Can obtain product` and `Can retry payment`, are the remaining solid diamonds that are not waiting groups, due to them not being a place where waiting can occur. Put simply, these are a decision that the system can take immediately based on metrics like availability, contents of basket, and how many times payment has been attempted.

For low competition products large crowds are simple to deal with for online shops. By simply increasing server capacity, the same service can be provided to a greater amount of customers. This holds no matter the total amount of customers in the shop, as long as the competition per product is low enough. For high competition products however, these online shops cannot make use of physical constraints, like the physical shop can, to ensure that service is provided without unintentional favoritism.

Shop operators can enact multiple different policies to ensure no favoritism is being given, but the exact policies to enact depend on the shop operators’s personal viewpoints and goals. In Section 1.2.1, a number of policies are laid out that can be used by shop operator to handle customer flow and behavior in their shops.

Note, service without unintentional favoritism does not imply that all customers receive the same service. Multiple classes of customers may exist, each with a different level of privilege, e.g. passengers of an airline for a single flight are divided in different classes: first, business, economy, etc. Within each class, service should be provided without favoritism, while between classes service can be different. This can be a way of handling large crowds, allowing or disallowing access to service depending on the customer’s class, although this requires tagging of customers, but basing this on certain attributes (e.g. approximate location) is also an option.

Another method of flash crowd mitigation in offline shops is limiting the number of items that can be purchased by a single customer, so that more customers can get a share of that product. Simply put, if a product has a limited stock of 1000, allowing a maximum of 10 items per customer guarantees that at least 100 customers should be able to purchase the product.

Sometimes, the exact opposite is required, choosing a minimum amount of (sub-) product or only being able to choose a multiple of a certain (sub-) product. Examples of this would be respectively: airplane ticket with at least one item of checked luggage and "buy-one-get-one-free" type products, where only multiples of a product can be purchased.

When flash crowds occur at physical shops, it seems intuitive that the crowd gathers for that specific shop. However, for the most part just a few products are highly desirable and will be the cause of the flash crowd. To illustrate, ample video examples exist showing crowds scrambling to enter shops on events like "Black Friday" and "Cyber Monday" and fighting over a few heavily discounted products. These few items were the cause of the flash crowd.

Unfortunately, products cannot be considered the source of flash crowds. Ample examples exist where the actual shop is the source of attraction. Specifically, popup shops, where a shop "pops-up" with little to no announcement to create a fear of missing out on potential deals. In these cases, crowds gather not for specific products, but for the shop, that might sell desirable products.

The last stock issue which needs consideration is what will be called the "final-items problem". When an item has certain limited stock while total demand for this product exceeds the availability, a way of determining who will get the products is needed. As explained before, if the competition is low, mitigation is possible. However, if it is assumed all customers currently in a shop suddenly are interested in the same product and rush over to add it to their basket, increasing competition, the issue manifests. This, however, is just an extension of normal stock handling. It becomes an issue when combined with sub-products and/or product constraints need to be taken into account.

Assume only two products are available, but to be a valid combination a customer needs to purchase both products (forced "buy-one-get-one-free") to be valid. When in a physical shop two different customers take only one instance of the product they might decide that one of them can get both. For online shops, inter-customer communication is impossible so there is a need for a tiebreaker to solve these issues. The easiest solution is that neither customer gets both products, both need to wait a certain time (backoff) and then might try again.

For sub-products, this entails having to hand back sub-products that already have been picked to ensure that another customer can get the full product.

1.2.1 Policies

In this section different policies will be presented which can be used by shop operators to compose a shopping experience for their customers. The policies will be converted to requirements in Chapter 2. Multiple different frameworks of specifying requirements exist. For this thesis, the SMART framework[1] was chosen. SMART originates from time management courses and aids with setting attainable goals. The paper describes how to generate requirements in a SMART manner. A requirement is SMART if it is: Specific, Measurable, Attainable, Realizable, and Traceable. In the context of policies it is interpreted to mean:

S Specific, a policy must say what is required.

M Measurable, when possible verify that the policy is implemented.

A Attainable, the system can actually exhibit the described policy.

R Realizable, the policy can be implemented given the known constraints.

T Traceable, ability to trace the policy to other parts of the process: requirements, design, implementation, and tests.

For these policies, two distinct classes of customers are recognized: normal, and privileged customers. Privileged customers can do anything that normal customers can, but might be able to receive service when normal customers cannot. These privileged customers might be identified before the start of and/or during their shopping run. In certain industries, the privileged customer class can be separated further. Previously, three waiting groups (**w1**, **w2**, and **w3**) were identified where waiting might need to occur. For each of these some policies will be described. Furthermore, policies regarding the final-products tiebreaker will be listed. Some listed policies are more usable than others, the provided lists are intended to provide an overview, not to be exhaustive. These policies have been distilled from interviews with employees of Eventix.

w1, Can enter

The policies described in this section pertain to admittance into a shop, they not only deal with admission, but also with (not) providing information to the customer.

Customers are admitted ...

`ce1` immediately.

`ce2` eventually, admission is temporarily held.

`ce3` only iff privileged.

`ce4` in (near) arrival monotonic order, i.e. if customer a arrives before customer b, a will be admitted before b.

`ce5` in random order.

`ce6` only while the capacity of the shop is not exceeded.

`ce7` with a constant flow, e.g. 5 per minute.

Customers ...

`ce8` abandoning shopping runs does not lead to changes in service.

`ce9` can know when they approximately will be admitted.

`ce10` can know how many other customers are also awaiting to be admitted.

`ce11` cannot know when they approximately will get admitted.

`ce12` cannot know how many other customers are also awaiting to be admitted.

w2, Can continue shopping

The policies described in this section pertain to a customer being able to choose a product.

Any customer can ...

`cs1` immediately choose a product.

`cs2` limit reservation burstiness, allow choosing a limited amount of product per interval, increasing the number of different customers that can buy the same product.

`cs3` choose up to a strictly limited amount of product.

`cs4` choose up to a strictly limited amount of a single product.

`cs5` choose products within a certain fixed shopping window. e.g. a customer has 30 minutes to complete filling its basket.

`cs6` only choose product that was not shown as sold-out to any customer.

Note, for `cs5` another option is keeping track of the age of products in the shopping basket. This however, is undesired since it allows for the following to occur:

- A customer picks a product p_1 and places it in the basket.
- Some time later, the customer picks another product p_2 and places it in the basket.
- The customer returns p_1.
- Repeat

This would result in the ability for a customer to postpone completing its order indefinitely even though a maximum shopping-time is desired.

Can obtain product

The policies described in this section pertain to a customer being able to place a product in its basket.

Any customer can get ...

`op1` up to the amount of product that has not been purchased yet.

`op2` up to the amount of product that has neither been purchased nor is awaiting payment success.

`op3` up to the amount of product that has neither been purchased, is awaiting payment success, nor has been placed in baskets yet.

`op4` up to a certain amount of product, allowing for some overselling percentage.

w3, Can pay

The policies described in this section pertain to a customer being able to pay for its order.

`cp1` any customer can pay.

`cp2` no customer can pay (temporarily).

`cp3` in a pre-/yet to- be determined order.

`cp4` only privileged customers can pay.

Note, in general these policies can, or need to be, combined to achieve a feasible shopping experience. Most are composable, but others are not. Within the same category mutual exclusive policies exists, for instance: neither: `ce3` and `ce4`, nor `cp1` and `cp3` are composable. These lists also show that it might be desirable to alter the applied policies depending on the situation. For instance, when a shop is experiencing a high, load customers might need to be prevented for paying for their orders (`cp2`) until enough capacity can be created to handle all the payments. While others might only need to be altered, like `cp3`.

Since the entire system is multi-tenant, a method needs to exist to ensure that for different shops different compositions of policies can be applied. Different shops can have different policies depending on the exact outcome the shop operator desires.

Tiebreakers

As described in the previous section, different policies can exist to deal with tiebreakers regarding the final-products problem.

`tb1` no customer gets a final product unless the complete product can be reserved.

`tb2` random customers will get the final products.

`tb3` customers in the shop the longest will get the final products.

`tb4` customers in the shop the shortest will get the final products.

`tb5` customers having the highest value in their basket will get the final products.

1.3 Related work

This project deals with handing out instances of products (reservations) with a strictly limited stock and proper admission control even when encountering flash crowds. Both of these topics already have been subject of some research. However much is only tangentially relevant to this project.

Listed below are a number of papers which are at least somewhat relevant.

1.3.1 Reservations

Research on reservations systems pertains mainly to low-level resource reservations or when relevant to this project are quite old. Usually describing centralized systems.

In an interview back in 1984, the TWA reservation system is described[2]. In a slightly more recent publication, Eklund described the history of another such reservation system[3] dating back to the 1940's. Even though these publications are old and describe even older systems, they provide nice insights into data recovery procedures, basic database designs, and how to operate such a system in an even further constrained environment as the airline industry.

More modern descriptions of such systems exists, like the work by Oloyede, Alaya, and Adewole[4]. They describe the construction of a system for reserving bus tickets. They provide a very practical view of what a reservation system might look like.

The older systems are usually centralized since that is a comparatively easy way of ensuring data is consistent. Since a distributed architecture will be presented, consistency will play a large role in ensuring reservations can be handed out while ensuring stocks are not exceeded. Bermbach and Kuhlenkamp provide a comprehensive overview of different data consistency models and viewpoints[5]. Including detailed discussions on usability and usecases of these models.

1.3.2 Admission control and flash crowd handling

Some work is done on admission control in relation with flash crowds. But in general, research no flash crowds is how to properly handle content delivery.

Elson and Howell present an overview of different techniques that can be used for dealing with flash crowds[6]. Including content delivery using different tiers serving the content, load balancing techniques, and dynamic scaling.

Another overview of different techniques is given by Hamilton[7]. The paper lists many different recommendations on how to design internet scale applications and therefor how to potentially deal with flash crowds. These recommendations provide ideas on how to structure the architecture and how to implement certain aspects of the proof-of-concept.

Chen and Heideman presented a novel way of detecting and mitigating the effects of flash crowds by aggregating requests and responses when a flash crowd is detected[8]. The method of dealing with flash crowds is by selectively allowing requests to pass at a router level.

In 2004, Zeng and Veeravalli provide an overview of, and described a new dynamic load balancing algorithm[9]. This algorithm strives to divide members of a flash crowd between components in the system in such a manner as to distribute the load between components evenly.

Fischer, Lynch, Burns, and Borodin presented a method for maintaining a distributed FIFO queue[10]. The queue contains processes, and admission from that queue means the process is allowed to access its critical section. Only allowing for a certain maximum number of processes to access their critical section concurrently. This is very similar to what is needed for this project, though their technique does not aid with arrival monotonic adding to the queue.

1.4 Problem statement, deliverables, and organization

In this chapter we described two categories of problems encountered by shop operators and ticketing platform administrators. A method needs to be devised to deal with flash crowds and compound products with limited stocks. Different shop operators can have different viewpoints on how to admit flash crowds and handle compound products. Policies that can be used to express these viewpoints were laid out in Section 1.2.1.

The problem this project solves, is how to admit flash crowds and issue instances of compound products using policies applied by shop operators. Ensuring, neither the applied policies, nor the load on any shop influences the operations of other shops. Each shop should operate fully independent of other shops. Even when shops are hosted by the same multi-tenant system.

The main result achieved in this project is a distributed architecture that can be used to construct a system that provides a solution to the stated problems. The architecture should be designed such that systems implementing this architecture can be scaled dynamically, are deployable to a Kubernetes cluster, and are resilient to failures ensuring no data loss occurs. Furthermore, for the designed architecture a proof-of-concept implementation is created to verify whether the architecture is suitable to solve the stated problem.

For creating this architecture, first requirements engineering is needed. The gathered requirements, which the architecture must satisfy, are presented in Chapter 2. When applicable these requirements receive a formalism to make them measurable. These requirements are used to devise an architecture, which is presented in Chapter 3. To validate the architecture, a proof-of-concept implementation is constructed and described in Chapter 4. This chapter also describes how the proof-of-concept can be deployed on a Kubernetes cluster. Two functional tests and their results are presented in Chapter 5. These functional tests verify whether the proof-of-concept performs as expected. In Chapter 6, the performance of the proof-of-concept implementation is discussed, future work is laid out, and recommendations are made for continuation projects.

Chapter 2

Requirements

In the previous chapter an introduction to the problems was stated and listed a number of policies on how to deal with these problems. In this chapter the requirements of the system (and thus the policies) will be described.

From the policies in Section 1.2.1 it becomes clear that there exist at least two distinct groups of requirements. One dealing with the life cycle of a shop visitor, the other dealing with the lifecycle of products. The first pertaining to dealing with stock and availability (reservations), while the second deals with admission to the shop and payment services (waiting). Both parts will be treated as distinct categories, to be specified separately. Furthermore, some requirements will be presented for which no policies have been defined in the previous chapter, but which are important nevertheless. These extra requirements (as are the policies) are based on interviews with Eventix employees.

Each of the policies laid out in Section 1.2.1 will be translated into requirements directly. The identifiers of each of these requirements directly corresponds with the policies laid out in Section 1.2.1. The interviews also provided a second set of requirements for each part of the system.

Not all requirements are equally important. In order to make this distinction, a priority system is needed. For this project `MoSCoW`[11, 12] was chosen. The main advantage of `MoSCoW` is its simplicity, since it only uses four levels of importance. Furthermore, Eventix had experience with this system which eased setting the priorities of each of the requirements.

`MoSCoW` is an acronym for:

- M **Must have**, critical requirements which must be present for the system to be a success.
- S **Should have**, important but not critical requirements, often as important as **Must have** requirements, but not as critical for the current delivery.
- C **Could have**, desirable but not necessary requirements, often enhance user experience or performance.
- W **Won't have**, requirements that are (currently) undesirable.

2.1 Scope limitations

The problems described below are not considered for this project, even though they heavily influence the perceived operations of the system. Their inclusion would increase the scope of the project to an unmanageable size.

2.1.1 Worldwide support

In an ever more connected world, more and more shops draw interest from an ever more geographically disperse group of customers. Consequently, distances between customers, and distances between customers and the shops increase. This equates to the servers of online shops having an (on average) greater distance to the shop's customers, impacting loading time of the shop due to networking delays. To keep the same level of loading time all over the world, a system might need to be globally distributed in some manner.

However, this will be ignored since the shops that would benefit from having these issues resolved likely attract mostly local interest. This is due to:

- An event needs to be traveled to. As distances increase, travel costs also increase, decreasing the number of attendees from further away.
- The further an item needs to be shipped, the higher the shipping costs will be. Making it more cost efficient to shop locally.

For both of these reasons, it is also likely that customers are still motivated enough to purchase the products and are content with higher loading times.

2.1.2 Content delivery

For serving online stores, content delivery also needs to be resilient to flash crowds. When a flash crowd occurs, the content still needs to be served properly. Serving content is well understood. Many different techniques for content delivery are known and have even permeated into enthusiast level applications[6].

Content delivery is an integral part of a full-fledge system. If the shop interface cannot be served to customers, no shopping can occur. The cited paper lists many techniques that can be used to construct a robust content delivery system, showing the breadth of this topic. This breadth is the reason content delivery will not be taken into account for this project.

2.2 Formal requirements

In this section the requirements will be presented and, whenever reasonable, a formal definition of the requirement will be stated. To facilitate these formal definitions, multiple (total) partial functions and sets need to be defined. For all of these requirements only a single shop is considered, since each shop operates independently from all others.

The requirements listed in Sections 2.2.2 and 2.2.3 are direct translations of the policies specified in Section 1.2.1, except for the set of requirements about **Sub-products**. Since the policy proposals from Section 1.2.1 are not always unambiguous more stringent definitions of these requirements are laid out either in text or in the formalizations.

2.2.1 Definitions

Listed below is an initial set of definitions which will be used in all requirement formalizations. This is not the full list, in Sections 2.2.2 and 2.2.3 more definitions will be given.

$\mathcal{T}$ is the set of positive real numbers, representing semi-infinite time domain (2.1)

$\mathcal{P}$ the set of products sold in the shop (2.2)

$\mathcal{C}$ the set of customers who want to enter the shop (2.3)

$\mathcal{C}_P \subseteq \mathcal{C}$ subset of customer having some privilege (2.4)

$a : \mathcal{C} \to \mathcal{T}$ the arrival times of customer at a shop (2.5)

$e : \mathcal{C} \to \mathcal{T} \cup \{\infty\}$ admission times of customers into a shop (entry) (2.6)

$p : \mathcal{C} \to \mathcal{T} \cup \{\infty\}$ the time when a customer is ready to pay (2.7)

$d : \mathcal{C} \to \mathcal{T} \cup \{\infty\}$ shop departure time for a customer (2.8)

These yield the following invariant, which must always hold for all customers due to the flow through a shop displayed in Figure 1.2:

$$\forall_{c \in \mathcal{C}} \;\; a(c) \leqslant e(c) \leqslant p(c) \leqslant d(c) \tag{2.9}$$

Note, that one scenario breaks this invariant, the accidental departure from a shop by a customer. Taking this into account, the invariant becomes:

$$\forall_{c \in \mathcal{C}} \;\; a(c) \leqslant e(c) \leqslant p(c) \leqslant d(c) \;\Rightarrow\; a(c) \leqslant d(c) \tag{2.10}$$

2.2.2 Reservations

The flowchart in Figure 1.2 shows that shopping customers over time collect a number of products, which thereby may become unavailable to other customers. The state of these collections is captured using the following function:

$$r_c : \mathcal{T} \times \mathcal{P} \times \mathcal{C} \to \mathbb{N} \tag{2.11}$$

Where $r_c(t, p, c)$ is the size of the collection of products of type $\mathcal{P}$ that customer c has collected at time t. Depending on where c is in the flowchart of Figure 1.2, the meaning of this number is slightly different. The size of the collection of products depicted by $r_c(t, p, c)$ is called:

- reserved, when the customer has been admitted into the shop but is still shopping ($e(c) \leqslant t < p(c)$)
- pending, when the customer has gone to checkout but has not yet left the shop, ($p(c) \leqslant t < d(c)$)
- bought, when the customer has left the shop ($d(c) \leqslant t$)

Note, only while the customer is shopping, the value of r_c changes. After the customer has finished shopping but before the customer has left ($p(c) \leqslant t < d(c)$), the value of r_c does not change. After the customer has left ($d(c) \leqslant t$), the value of r_c for any customer c and any product p is as follows:

$$r_c(t, p, c) = \begin{cases} 0 & \text{when the customer leaves without products} \quad (2.12a) \\ \lim\limits_{t' \uparrow d(c)} r_c(t', p, c) & \text{when the customer leaves with products} \quad (2.12b) \end{cases}$$

This is due to the products either being bought or returned to stock. This difference is due to the fact that when a customer leaves, the products have to be handed back, while when they have not yet $r_c(t, p, c)$ becomes zero when no successful.

Note that r_c is a left continuous piece-wise constant function (see Figure 2.1), where left continuous means that r_c seen as a function of time satisfies $r_c(t, p, c) = \lim_{t' \uparrow t} r_c(t', c, p)$.

Next consider the function r_i defined by:

$$r_i(t, p, c) = \lim_{t' \downarrow t} r_c(t', p, c) - r_c(t, p, c) \tag{2.13}$$

where $r_i(t, p, c)$ is the number of products p placed in or removed from the basket of customer c at time t.

So, for all $c \in \mathcal{C}$ and $p \in \mathcal{P}$, r_i is almost always zero, except for a finite number of moments $t \in \mathcal{T}$.

If a reservation is placed at t, it is assumed to be handled immediately. Furthermore, it is assumed that at any point in time there is either an increase or decrease in the number of reservations, not both.

Figures 2.1 and 2.2 depict an example for both functions for the following scenario for a customer and a product:

t_1	The customer reserves two products
t_2	The customer reserves three products
t_3	The customer releases two reservations
t_4	The customer reserves another product

Figure 2.1 shows the cumulative number of reservations, while Figure 2.2 shows the steps taken at any point in time.

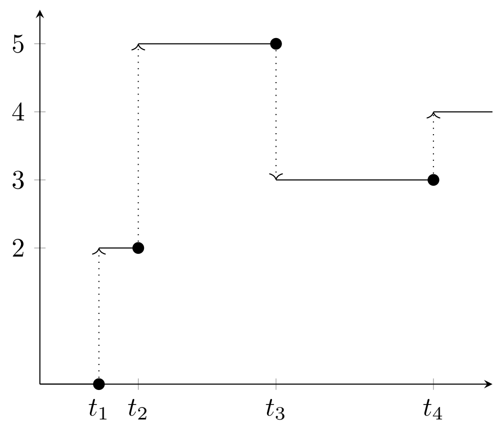


Figure 2.1: Cumulative number of reservations (r_c) over time

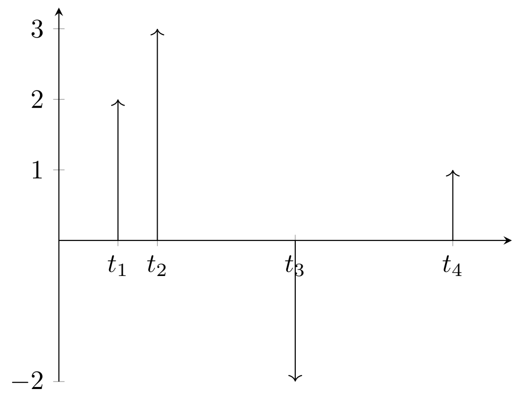


Figure 2.2: Graph visualizing the reservation increases and decreases (r_i) over time

The following sets and (total) functions aid with the formalisms listed below.

$T_{in}(p, c) = \{t \in \mathcal{T} \mid r_i(t, p, c) > 0\}$ the points in time when reservations were added (2.14)

$T_{out}(p, c) = \{t \in \mathcal{T} \mid r_i(t, p, c) < 0\}$ the points in time when reservations were released (2.15)

$T_{ms} \in \mathcal{T} \cup \{\infty\}$ the maximum amount of time a customer can be choosing products (2.16)

$S_m \in \mathbb{N}^+ \cup \{\infty\}$ the maximum number of products that can be purchased (2.17)

$T_b \in \mathcal{T} \cup \{\infty\}$ burst interval, the interval in which a high number of reservations can be placed (2.18)

$S_{mb} \in \mathbb{N}^+ \cup \{\infty\}$ the amount of reservations that can be placed in T_b (2.19)

$P_s : \mathcal{T} \times \mathcal{P} \to \mathbb{N} \cup \{\infty\}$ the stock of a product at a certain point in time (2.20)

$P_a : \mathcal{T} \times \mathcal{P} \to \mathbb{N} \cup \{\infty\}$ the availability of a product at a certain point in time, this is more specifically laid out in Section 2.2.2 (2.21)

$\mathcal{P}_{sr} : \mathcal{P} \to 2^{\mathcal{P}}$ the set containing the required sub-products of a product (2.22)

$\mathcal{P}_{so} : \mathcal{P} \to 2^{\mathcal{P}}$ the set containing the optional sub-products of a product (2.23)

$\mathcal{P}_{opt}(p) = \mathcal{P}_{so}(p) \cup \mathcal{P}_{sr}(p)$ the set containing all sub-products of a product (2.24)

$r_r(t, p) = \sum_{c \in \mathcal{C},\ t < p(c)} r_c(t, p, c)$ the total number of products all customers reserved prior to a point in time (2.25)

$r_p(t, p) = \sum_{c \in \mathcal{C},\ p(c) \leqslant t < d(c)} r_c(t, p, c)$ the total number of products all customers have pending payment success prior to a point in time (2.26)

$r_b(t, p) = \sum_{c \in \mathcal{C},\ d(c) \leqslant t} r_c(t, p, c)$ the total number of products all customers bought prior to a point in time (2.27)

Invariants

For the proposed functions and sets a few invariants can be defined to ensure that they can be used to properly formalize the desired shop behavior.

No customer can start reserving products before they have been admitted:

$$\forall_{c \in \mathcal{C}}\ \forall_{t \in \mathcal{T}}\ t < e(c) \Rightarrow \forall_{p \in \mathcal{P}}\ r_c(t, p, c) = 0 \tag{2.28}$$

To prevent exceeding of stock, the sum of r_c for all customers and products at any point in time cannot exceed the product stock:

$$\forall_{p \in \mathcal{P}}\ \forall_{t \in \mathcal{T}} \sum_{c \in \mathcal{C}} r_c(t, p, c) \leqslant P_s(t, p) \tag{2.29}$$

Ensuring stocks are not exceeded is not enough since stocks and availability have a more complex relation, as described in Section 2.2.2. Therefore, it must hold that the value of r_i can be at most P_a at any point in time for any customer that has not yet finished shopping. Note, due to Definitions 2.13 and Invariant 2.28 only the point in time customers have finished shopping needs to be defined.

$$\forall_{c\in\mathcal{C}}\ \forall_{p\in\mathcal{P}}\ \forall_{t\in\mathcal{T}}\ t < p(c) \Rightarrow r_i(t,p,c) \leqslant P_a(t,p) \tag{2.30}$$

Lastly, after a customer has finished shopping, no changes to the amount of reserved products is allowed. One exception being that all products can be put back, when the customer leaves the shop without products, due to unsuccessful payment for instance.

$$\forall_{c\in\mathcal{C}}\ \forall_{p\in\mathcal{P}}\ \forall_{t\in\mathcal{T}}\ r_i(t,p,c) \neq 0 \Rightarrow \Big(t < p(c) \vee r_i(t,p,c) = -r_c(t,p,c)\Big) \tag{2.31}$$

Can continue shopping

Listed below are the requirements for determining when customer must be done shopping, based on the policies defined in Section 1.2.1.

`cs1` any customer can immediately choose a product [MUST]
Meaning that no limits will be imposed on the shop flow after a customer has been admitted. But a customer that has not yet been admitted is not allowed to have any reservations. Since this is the initial condition listed in Equation 2.28, a formalization is not needed.

`cs2` limit reservation burstiness, any customer can choose a limited amount of product per interval [COULD]
The intent of this requirement is to limit the number of items that can be reserved within a certain interval (burst protection), to prevent customers from being able to issue too many actions in too short a time. This ensures that there is enough total stock for all customers. To achieve this, the total number of reservation increases will be limited. This is done since it cannot be known beforehand which products have a limited availability due to (intentional) low stock or have low availability due to a large number of reservations. Therefore, looking at the actual availability of products is not a good metric of determining popular products. Two possible formalizations can be formulated to achieve the desired behavior, both having different implications.

The first would be to impose a lower bound on two subsequent reservation increases, by only allowing a single increase per interval. Another, preferable, way is to limit the number of reservations for any interval of a certain length, ensuring that a customer wanting to purchase a certain amount (fewer than the bound) of products is not forced to wait for a certain amount of time before the next reservation can be placed.

Since the second version is preferable, this formalism will be used to limit the number of reservations in a interval.

$$\forall_{t\in\mathcal{T}}\ \forall_{c\in\mathcal{C}}\ \Big(\sum_{p\in\mathcal{P}}\ \sum_{t'\in T_{in}(p,c),t-T_b<t'\leqslant t} r_i(t',p,c)\Big) \leqslant T_{ms} \tag{2.32}$$

`cs3` any customer can choose up to a strictly limited amount of product [MUST]

$$\forall_{t\in\mathcal{T}}\ \forall_{c\in\mathcal{C}}\ \sum_{p\in\mathcal{P}} r_c(t,p,c) \leqslant S_m \tag{2.33}$$

`cs4` any customer can choose up to a strictly limited amount of a single product [MUST]
The total number of each product type that can be reserved by any customer at any single point in time needs to be limited. It is assumed a function f exists which maps each product to the maximum amount of times it can be purchased.

$$\forall_{t\in\mathcal{T}}\ \forall_{c\in\mathcal{C}}\ \forall_{p\in\mathcal{P}}\ r_c(t,p,c) \leqslant f(p) \tag{2.34}$$

`cs5` any customer can choose products within a certain fixed shopping window [MUST]

$$\forall_{c\in\mathcal{C}}\ \ e(c) < \infty \Rightarrow p(c) - e(c) \leqslant T_{ms} \tag{2.35}$$

`cs6` any customer can only choose product that was not shown as sold-out to any customer [MUST]
When a customer at a previous point in time saw that a product was unavailable, it has to remain unavailable even when stock has become available.

$$\forall_{t\in\mathcal{T}}\,\forall_{p\in\mathcal{P}}\left(\exists_{t'\leqslant t}\,P_a(t',p)=0\right)\;\Rightarrow\;P_a(t,p)=0 \tag{2.36}$$

Can obtain product

All these requirements are formalized as an image of P_a for values of $t\in\mathcal{T}, p\in\mathcal{P}$. Since P_a defines the availability of any product. Since usable stock is not only determined by P_s but also by the availability of its required sub-products, the following recursive function will be used to depict the usable stock of a product:

$$P'_s(t,p)=\min\left(P_s(t,p),\min_{p'\in\mathcal{P}_{sr}(p)}\left(P'_s(t,p')\right)\right) \tag{2.37}$$

`op1` any customer can get up to the amount of product that has not been purchased yet [MUST]

$$P_a(t,p)=P'_s(t,p)-r_b(t,p) \tag{2.38}$$

`op2` any customer can get up to the amount of product that has neither been purchased nor is awaiting payment success [MUST]

$$P_a(t,p)=P'_s(t,p)-r_b(t,p)-r_p(t,p) \tag{2.39}$$

`op3` any customer can get up to the amount of product that has neither been purchased, is awaiting payment success, nor has been placed in baskets yet [SHOULD]

$$P_a(t,p)=P'_s(t,p)-r_b(t,p)-r_p(t,p)-r_r(t,p) \tag{2.40}$$

`op4` any customer can get up to an amount of product, allowing for some overselling percentage [COULD]
Assume $P'_a, O\in\mathbb{R}, |O|\leqslant 100$, to be the presented availability to the customer and the overselling percentage respectively. This value can be negative to also support "underselling"

$$P'_a(t,p)=(1+\frac{O}{100})*P_a(t,p) \tag{2.41}$$

Sub-products

For sub-products reservation, another function is needed to keep track of the exact number of sub-product reservations per reservation. Since a reference to a product reservation (using r_i and T_{in}) can be generated, it can be used to uniquely track the number of reserved sub-products per product reservation.

$$sub: T\times\mathcal{C}\times\mathcal{P}\times\mathbb{N}^+\times\mathcal{P}\rightarrow\mathbb{N} \tag{2.42}$$

Since any non-zero value of $r_i(t,p,c)$ is either an increase or decrease, not both. All reservations can be identified since $r_i(t,p,c)$ gives the exact number of reservations increases/decreases. Since only reservation increases are needed, just points in time present in T_{in} will be considered. For every sub-reservation the following holds: the amount of sub-reservations for any products has to be zero when there is no reservation the sub-product can belong to.

$$\forall_{t\in\mathcal{T}}\,\forall_{c\in\mathcal{C}}\;\forall_{p\in\mathcal{P}}\;t\notin T_{in}(p,c)\Rightarrow\forall_{n\in\mathbb{N}}\,\forall_{p'\in\mathcal{P}_{opt}(p)}\,sub(t,c,p,n,p')=0 \tag{2.43}$$

`sp1` A product can have either a strictly limited, or unlimited stock [MUST]
Stock might be limited due to a finite number of items that can be sold or infinite when a product might be backordered. The definition for the reservation function already covers this. This is already provided by the specification of P_s

`sp2` A product contains multiple sub-products [MUST]
This has been resolved using the specification of $\mathcal{P}_{opt}$.

`sp3` A product contains required sub-products [MUST]
This has been resolved using the specification of $\mathcal{P}_{sr}$.

`sp4` A product contains optional sub-products [MUST]
This has been resolved using the specification of $\mathcal{P}_{so}$.

`sp5` A product contains a set of sub-products from which a minimal amount of products need to be chosen. i.e. from 10 sub-products, at least two must be chosen [COULD]
It is assumed a function f exists which maps products to the minimal amount of sub-products that need to be chosen.

$$\forall_{c\in\mathcal{C}}\ \forall_{p\in\mathcal{P}}\ \forall_{t\in T_{in}(p,c)}\ \forall_{n\in[1,r_i(t,p,c)]} \sum_{p'\in\mathcal{P}_{opt}(p)} sub(t,c,p,n,p') \geqslant f(p') \tag{2.44}$$

`sp6` A product contains a set of sub-products from which a maximum amount of products can be chosen. i.e. from 10 sub-products, at most five can chosen [COULD]
It is assumed a function f exists which maps products to the maximum amount of sub-products that can be chosen.

$$\forall_{c\in\mathcal{C}}\ \forall_{p\in\mathcal{P}}\ \forall_{t\in T_{in}(p,c)}\ \forall_{n\in[1,r_i(t,p,c)]} \sum_{p'\in\mathcal{P}_{opt}(p)} sub(t,c,p,n,p') \leqslant f(p') \tag{2.45}$$

`sp7` A product contains sub-products which can be purchased a limited amount of times, i.e. when composing a product, certain sub-products can be added a bounded number of times [COULD]
It is assumed a function f exists which maps products to the maximum amount of a single sub-product type that can be chosen.

$$\forall_{c\in\mathcal{C}}\ \forall_{p\in\mathcal{P}}\ \forall_{t\in T_{in}(p,c)}\ \forall_{n\in[1,r_i(t,p,c)]}\ \forall_{p'\in\mathcal{P}_{opt}(p)}\ sub(t,c,p,n,p') \leqslant f(p') \tag{2.46}$$

`sp8` For some $n, m \in \mathbb{N}, n > m$, when n customers want to reserve the same (sub-) product p, with $a_p = m$, a tiebreaker exists to determine who will get the (sub-) products [COULD]
A formalization for this requirement cannot be given since the exact workings of this tiebreaker is not given.

`sp9` For some $n, m \in \mathbb{N}, n > m$, when n customers want to reserve the same product p having required sub-products, with $a_p = m$, a tiebreaker exists to determine who will get the products and the sub-products [COULD]
A formalization for this requirement cannot be given since the exact workings of this tiebreaker is not given.

For sp8, assume that for a product p, $a_p = 1$ and two different customers want to place a reservation for p. When a tiebreaker exists, it can be ensured that one of the customers get this product.

2.2.3 Waiting

For the formalization of these requirements more sets and functions are needed, specifically:

$\mathcal{W}(t) = \{c \in \mathcal{C} \mid a(c) \leqslant t < e(c)\}$ the set of customers waiting for admission (2.47)
$\mathcal{B}(t) = \{c \in \mathcal{C} \mid e(c) \leqslant t < p(c)\}$ the set of customers choosing products (browsing) (2.48)
$\mathcal{D}(t) = \{c \in \mathcal{C} \mid p(c) \leqslant t < d(c)\}$ the set of customers who have finished shopping (2.49)
$\mathcal{E}(t) = \{c \in \mathcal{C} \mid d(c) \leqslant t\}$ the set of customers who have departed the shop (2.50)
$\mathcal{E}_s(t) = \{c \in \mathcal{C} \mid d(c) \leqslant t\}$ the set of customers who have departed the shop and successfully paid for the products (2.51)
$T_o \in \mathcal{T} \cup \{\infty\}$ the point in time when customers first are admitted into the shop (2.52)
$S_r \in \mathbb{N} \cup \{\infty\}$ the rate in which customers are admitted into the shop (2.53)
$S_{qt} \in \mathcal{T}, S_{qt} \leqslant T_o$ when customers are accepted into a waiting group for the shop (2.54)
$S_c \in \mathbb{N}^+ \cup \{\infty\}$ shop capacity, the maximum amount of people that are allowed to be choosing products (browsing) concurrently (2.55)

Can enter

`ce1` customers are admitted immediately [MUST]

$$\forall_{c\in\mathcal{C}}\ e(c) = a(c) \tag{2.56}$$

`ce2` customers are admitted never [MUST]

$$\forall_{c\in\mathcal{C}}\ a(c) = \infty \tag{2.57}$$

`ce3` customers are admitted iff privileged [COULD]

$$\forall_{c\in\mathcal{C}}\ e(c) < \infty \Rightarrow c \in \mathcal{C}_P \tag{2.58}$$

`ce4` customers are admitted in (near) arrival monotonic order [SHOULD]
There is no need for exact arrival monotonic admission of customers, since only the impression of arrival monotonic admission order needs to be given. Therefore, the following process wil be used

When a customer c_1 arrives more than some amount of time t_a before a second customer c_2, c_1 should be admitted before c_2. When $t_a = 0$ monotonic admission is achieved. In Section 2.2.5 bounds on t_a are laid out.

$$\forall_{c_1,c_2\in\mathcal{C}}\ a(c_2) - a(c_1) > t_a\ \equiv\ e(c_1) < e(c_1) \tag{2.59}$$

`ce5` customers are admitted in random order [COULD]
This would be used to randomize customers that attempt to have the lowest place possible in a waiting group as possible, even days or perhaps weeks in advance. The time after which customers will be admitted into a waiting group will be constrained, this time should be the shop opening time at the latest. This would mean that all customers will have to rejoin the waiting group. When the waiting group admission time and shop opening time are the same, randomization occurs. This will be achieved due to external factors like reaction time and network speed. During normal opening-time there is no need for random admission, this was only identified after the policy requirement had been given. Therefore, only the set of customers arriving before S_{qt} are randomized.

$$\forall_{c\in\mathcal{C}}\ a(c) < S_{qt}\ \Rightarrow\ e(c) = \infty \tag{2.60}$$

`ce6` customers are admitted only while the capacity of the shop is not exceeded [MUST]

$$\forall_{t\in\mathcal{T}}\ |\mathcal{B}(t)| \leqslant S_c \tag{2.61}$$

`ce7` customers are admitted with a constant flow [SHOULD]
Unlike cs2 bursts are not allowed, but actually a fixed rate of customers admitted into the shop, meaning the entry-time between two customers has a certain minimum.

$$\forall_{c_1,c_2\in\mathcal{C}}\ e(c_1) \leqslant e(c_2) < \infty \Rightarrow e(c_2) - e(c_1) < \frac{1}{\mathcal{S}_r} \tag{2.62}$$

`ce8` customers abandoning shopping runs does not lead to changes in service [SHOULD]
There are two causes for customers to abandon a shopping run, either accidental or on purpose. When on purpose, the customer is unlikely to return since the action was intentional. Accidental leaving, including accidental refreshes and networking issues, should not result in a “punishment” for the customer. Neither should intentional leaving and returning result in an improvement of service. Since the difference between accidental and purposeful abandonment cannot be detected all abandonment should be handled in the same way.

Any action by the customer leaving the shop, except explicit leaving using the **Leave without products** or **Leave with products** actions, need to be ignored (Figure 1.2). No changes to the (eventual) admission time (e) or time that the customer needs to finish browsing T_{ms} will be made.

`ce9` customers can know when they approximately will be admitted [SHOULD]
Showing the customer the amount of customers in front of the current customer (customers that would be admitted before the current customer) is a nice way to show the customer how long the approximate wait would be. When this can update automatically, it also serves to show progress is being made.

`ce10` customers can know how many other customers are also awaiting to be admitted [COULD]
Showing the amount of customers behind the current customer (customers that would be admitted after the current customer) can aid with customers deciding they do not want to remain in the waiting group.

`ce11` customers cannot know when they approximately will get admitted [COULD]
The opposite of ce9, when it is desirable to give the queuing customer as little information as possible, to keep interest as high as possible.

`ce12` customers cannot know how many other customers are also awaiting to be admitted [COULD]
The opposite of ce10, when it is desirable to give the queuing customer as little information as possible, to keep interest as high as possible.

Can pay

These requirements pertain to the ability of customers to precede to payment, these are analogous to the requirements for **Can enter**.

`cp1` any customer can pay [MUST]
No limit should be imposed on the customers that want to pay, comparable to ce1. But since a customer needs to initiate payment, a stricter bound cannot be given.

$$\forall_{t \in \mathcal{T}} \forall_{c \in \mathcal{D}(t)} \; p(c) < \infty \tag{2.63}$$

`cp2` no customer can pay [MUST]
Whichever customer is done shopping, a payment attempt should not be allowed, comparable to ce2

$$\forall_{t \in \mathcal{T}} \forall_{c \in \mathcal{D}(t)} \; p(c) = \infty \tag{2.64}$$

`cp3` in a pre-/yet to- be determined order [COULD]
It is assumed a function f exists which maps customers to the order in which they are allowed pay.

$$\forall_{t \in \mathcal{T}} \forall_{c_1, c_2 \in \mathcal{D}(t)} \; f(c_1) < f(c_2) \; \Rightarrow \; p(c_1) < p(c_2) \tag{2.65}$$

`cp4` only privileged customers can pay [COULD]

$$\forall_{c \in \mathcal{C}} \;\; c \in \mathcal{C}_P \equiv p(c) < \infty \tag{2.66}$$

2.2.4 Tiebreakers

Id	Requirement	Priority
`tb1`	no customer gets a final product unless the complete product can be reserved	MUST
`tb2`	random customers will get the final products	COULD
`tb3`	customers admitted the earliest get the final products	COULD
`tb4`	customers admitted the latest get the final products	COULD
`tb5`	customers having the highest value of products in their basket will get the final products	COULD

2.2.5 Near monotonic arrival

For near monotonic arrival, different customers in the same waiting group will be compared by their arrival and admittance times $a(c_i), e(c_i)$. The only waiting groups where admittance is important is **w1** (`Can enter`), since this influences the potential availability of products. For this, it is assumed that the unit of $\mathcal{T}$ is milliseconds.

For any two customers ($\forall_{t \in \mathcal{T}} \; c_1, c_2 \in \mathcal{W}(t)$), c_1 is admitted before c_2 iff:

`ma1` c_2 arrives more than 1 second after c_1 [MUST]
When there is more than 1000 milliseconds between the arrival of c_1, c_2, c_1 will be admitted before c_2.

$$\forall_{c_1, c_2 \in \mathcal{C}} \; a(c_2) - a(c_1) > 1000 \; \equiv \; e(c_1) < e(c_2) \tag{2.67}$$

`ma2` arrives more than 250 milliseconds after c_1 [SHOULD]
When there is more than 250 milliseconds between the arrival of c_1, c_2, c_1 will be admitted before c_2.

$$\forall_{c_1, c_2 \in \mathcal{C}} \; a(c_2) - a(c_1) > 250 \; \equiv \; e(c_1) < e(c_2) \tag{2.68}$$

`ma3` arrives more than 100 milliseconds after c_1 [COULD]
When there is more than 100 milliseconds between the arrival of c_1, c_2, c_1 will be admitted before c_2.

$$\forall_{c_1, c_2 \in \mathcal{C}} \; a(c_2) - a(c_1) > 100 \; \equiv \; e(c_1) < e(c_2) \tag{2.69}$$

2.3 Other requirements

2.3.1 Deployment requirements

Due to the erratic nature of flash crowds making their occurrence unpredictable, the system needs to be elastic, i.e. scale in or out depending on load. Since flash crowds can occur very quickly, this scaling must occur quickly as well. Two deployment platforms where evaluated, running directly within an OS, virtualized or bare-metal, and containerized. Containerization was chosen because of the following reasons:

- Increased portability, regardless of the platform and cloud, containers can run almost anywhere as long as a runtime is available.
- Increased security, through application isolation from the host- and other- systems.
- Better elasticity, through deploying more or removing containers.
- Faster start-up.
- Automatic load balancing.
- Build in high availability, depending on the orchestrator automatic container restarting on failure might be available.

The advantages of these components is that when a container crashes, an orchestrator can start a new instance of that container, increasing fault tolerance of the entire system. If the system was designed to run on a bare-metal/virtualized platform scaling and crash recovery would need to be specifically implemented, while for containerized solutions ready made solutions exist.

This also ensures that integration of the system in existing system should be comparatively easy, since the services should be as self contained as possible.

Multiple container orchestrators exist, Kubernetes, Swarm, Mesos, and others. Kubernetes was chosen since, it is one of the big three and Eventix already deploys part of its system on Kubernetes clusters.

Id	Requirement	Priority
`dp1`	application runs on Linux	MUST
`dp2`	application runs on Windows	WON'T
`dp3`	application is containerized	SHOULD
`dp4`	application runs in a Kubernetes cluster	SHOULD
`dp5`	application scales with load	SHOULD
`dp6`	application is failure resilient	MUST

Multi-tenantcy

As described in Section 1.1, Eventix's system is a multi-tenant system. Many of their customers and visitors make use of the same set of resources, regardless of size.

Id	Requirement	Priority
`mt1`	application handles a single shop	MUST
`mt2`	application handles multiple shops	SHOULD
`mt3`	when a single shop is experiencing a flash crowd, other stores should remain unaffected depends mt1	SHOULD
`mt4`	shop operators can have a guaranteed number of customers in their shop	SHOULD
`mt5`	shop operators can have a guaranteed limit on waiting times	WON'T

Limits

Id	Requirement	Priority
`li1`	application aids content delivery	WON'T
`li2`	application is globally distributed	WON'T

Metrics

Id	Requirement	Priority
`me1`	application works without supervision, based on some pre-specified operation policies	SHOULD
`me2`	application keeps track of metrics	MUST
`me3`	application keeps track of total number of waiting customers depends me2	MUST
`me4`	application keeps track of total customer influx depends me3	SHOULD
`me5`	application keeps track of total customer outflux depends me3	SHOULD
`me6`	application keeps track of number of waiting customers per shopdepends me2	SHOULD
`me7`	application keeps track of customer influx per shop depends me6	SHOULD
`me8`	application keeps track of customer outflux per shop depends me6	SHOULD

2.3.2 Miscellaneous requirements

Id	Requirement	Priority
`mc1`	Products can only be purchased from stores they are offered in	MUST
`mc2`	Customers have a limited amount of retries for achieving a successful payment	SHOULD

Chapter 3

System architecture

In this chapter an architecture will be presented for handling both flash crowds, and complex stocks. Firstly a decomposition of the requirements into separate groups is given, in such a manner that each group has a coherent set of responsibilities. These groups will then be combined into different components. For this project, we will be using the definition of a component by Szyperski[13]. This definition was chosen since it provides a clear method of specifying components. A component is a unit of composition providing explicit interfaces to other components and depending on interfaces provided by other components. After the components have been defined, a composition is presented making use of the architectural styles presented by van Steen and Tanenbaum[14].

One of the main considerations for the architecture was the desired deployment environment. Since multiple different deployment environment were specified, Linux and Kubernetes, focus was put on the most restrictive deployment environment: Kubernetes. Kubernetes is an extensible system that can be used to manage and deploy containerized applications and systems[15]. Containers are a way to package applications in such a manner that they contain everything in order to run. Containerized application should be designed using cloud-native patterns. The 12-factor app methodology[16] was specifically chosen as it has seen wide adoption in industry by (for instance) Heroku[17].

The 12-factor app recognizes 12 factors which when used correctly yield a well defined, easy to deploy, and easy to scale application. From these 12 factors the following are most relevant for the architecture:

- **Backing services**: no distinction should be made between local and third party services.
- **Processes**: components should be run as processes and be stateless when possible.
- **Port binding**: communication between components should happen over the network.
- **Disposability**: a component should be able to start and stop as quickly as possible.

Consequently, the architecture will be designed subject to these factors. i.e. all components are stateless when possible, boot and terminate as quickly as possible, and communicate with other components over the network. Obviously, state needs to be kept, statelessness in this context means that for every request, the needed state is retrieved from a datastore, and the resulting state is stored in a datastore before the request is finalized. From this follows that many of the requirements about persistence and resiliency should now be taken into account when choosing datastores.

Since the system as a whole needs to be scalable (dp5), it is already taken into account that for any component multiple instances might be running at the same time. All running instances of a component will collectively be referred to as “nodes” of that component.

3.1 Overview

The final system is performing multiple tasks, roughly categorized into maintaining an orderly shopping run, and ensuring stocks are properly kept. Since these two categories’s responsibilities are orthogonal to each other it would be best to separate these tasks into separate modules. Since each module needs to perform several tasks, each module will be further separated into components. Figure 3.1 contains an overview of the modules and the components in each module. For simplicity the datastores and communication to the datastores has been left

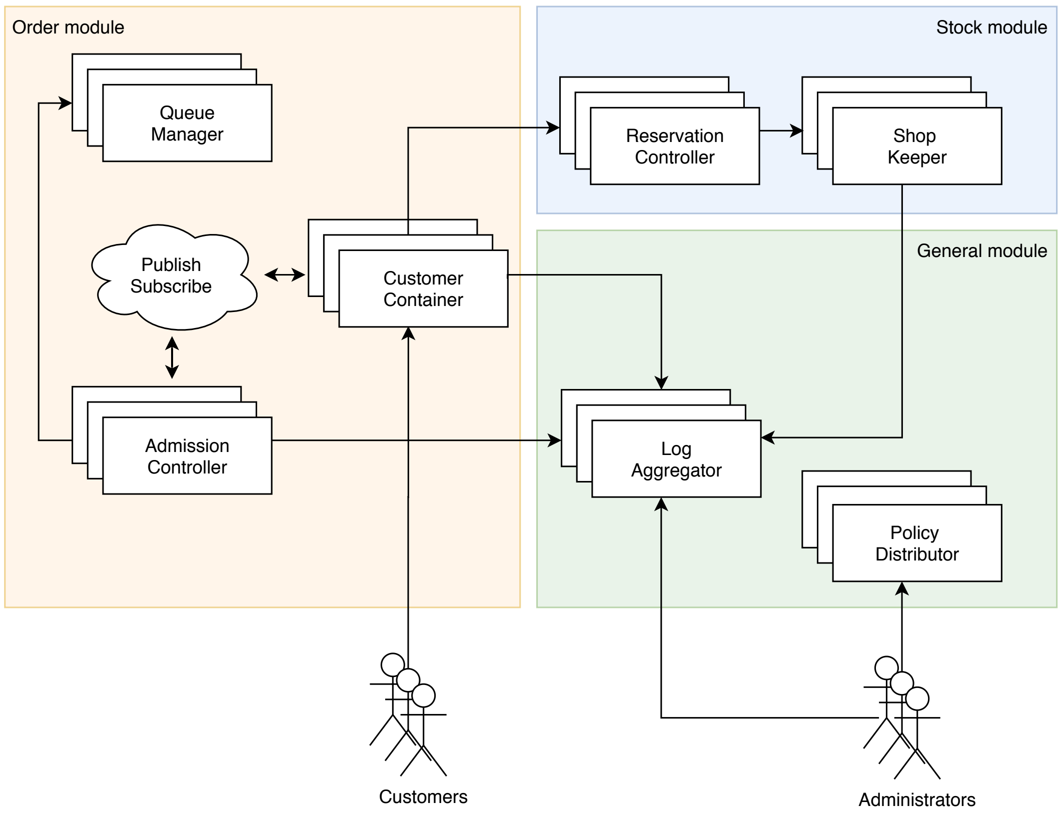


Figure 3.1: Diagram showing communication between components, and actors and components.

out. In further sections each of these components will be discussed. Section 3.3.3 contains examples of the actual communication between components.

In total three modules exist, each with their own tasks and responsibilities, the order, stock, and general modules. The order module is tasked with ensuring customers transition properly between phases of their shopping runs. The stock module is tasked with ensuring everything related to products and their stocks are handled properly. The general module provides services that other modules need, and contains the services that do not fit in the other modules.

The modules will exist of multiple components, each of which encompass a coherent set of responsibilities. The following list contains the modules, the components per module and the general responsibilities of each of the components:

- Order module:
  - Queue Manager: keeps track of an ordering of customers for every shop.
  - Admission Controller: decides when to allow which customers based on the ordering kept by the Queue Manager.
  - Customer Container: the entrypoint for customers into the system. When customers issue a request to the Customer Container, it ensures this request is permissible and gets forwarded to the correct component.
- Stock module:
  - Shop Keeper: keeps track of the number of products per phase of the product lifecycle.
  - Reservation Controller: keeps track of product instances and which customer they are issued to and is responsible for handing out these instances.
- General module:
  - Log Aggregator: tracks all generated log items and makes them available to administrators and other components.
  - Policy Distributor: distributes policies/settings provided by administrators to all components that need settings.

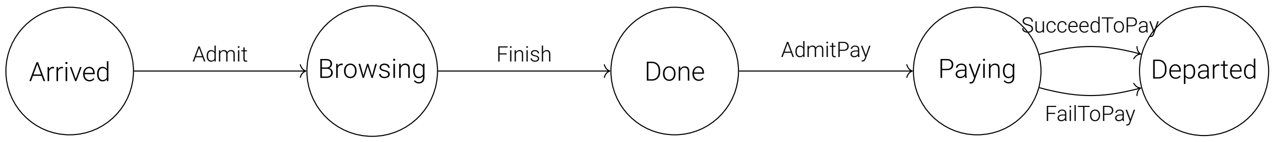


Figure 3.2: The customer lifecycle.

Figure 3.1 contains an overview of how these components are communicating, an arrow from component A to component B means that component A depends on an interface provided by component B.

All components in the system adhere to the definition of components provided by Szyperski[13]. All components have the following properties:

- a component is a unit of independent deployment
- a component is a unit of third-party composition
- a component has no (externally) observable state

A component provides contractually specified interfaces while consuming explicit context dependencies, i.e. a component provides interfaces to other components, while consuming interfaces provided by other components. For every presented component, the interfaces and dependencies are laid out.

In the next sections an overview of the components and the functionality they provide will be presented. Finally some examples on how the components interact are provided.

3.1.1 Order module

If it were up to the customers, their shopping runs would be as advantageous to themselves as possible without any regard for the needs and desires of other customers. The order module exists to ensure that order is kept and that all customers within the same privilege level are treated equally.

A customer's shopping run can be seen as the lifecycle of that customer within a store. Specifically, a shopping run consists of five phases:

- Arrived: the customer has arrived at the shop, but has not been admitted. Since the system does not perform any serving of content, admission is not trivially determined. How the customers's arrival is determined, is laid out later in this section.
- Browsing: the customer has been admitted and is selecting the desired set of products for purchasing.
- Done: the customer has finished browsing, but is not yet admitted to payment services.
- Paying: the customer has been admitted to payment services and is attempting payment for the chosen products.
- Departed: the customer has departed the store, either with products (after successful payment) or without.

A customer transitions into a new lifecycle phase iff some event occurs. Specifically the following events:

- Arrive (at store): customer arrives at store and the system gets notified of the arrival. Customer moves to "Arrived".
- Admit (into store): the customer is admitted into the store. Customer moves to "Browsing".
- Finish (browsing): when either the system or the customer has determined the browsing phase is over and the customer has now finished browsing. Customer moves to "Paying".
- AdmitPay: when the customer has been admitted to payment services. Possibly directly after finish.
- SucceedToPay: customer has attempted payment, which succeeds. Customer moves to "Departed".
- FailToPay: customer has attempted payment, which fails. Customer moves to "Departed".

These phases and events correlate to elements of the flowchart depicted in Figure 1.2 and the lifecycle with the transitions can be found in Figure 3.2.

Some events are generated by customers, while others are generated by the system. The arrive events are generated by customers. The finish event is generated by both the system and the customer since either can decide when a shopping run is finished. All other events are generated by the system.

Note, transitions to departed from other phases have been left out since that would make the figure confusing. Furthermore, any customer can decide to simply depart but not inform the system of the departure. Therefore, it must be ensured an upper bound exists on the amount of time a customer can be browsing or attempting payment. When this upper bound is reached, the customer should be forced to transition to the next phase in the lifecycle. This to guarantee progress will be made in every shopping run.

Ensuring order is kept consists of two distinct parts, keeping track of the actual order of customers and making use of that ordering to perform useful tasks. Since the behavior of these parts is orthogonal to each other these will be separated into distinct components. The Queue Manager will be responsible for actually keeping track of the ordering of customers, while the Admission Controller is responsible for ensuring that ordering is used for admitting customers.

Queue Manager

The Queue Manager will be providing two functions, registering customers into queues and when requested retrieve, return, and delete the head of a specific queue in order to facilitate admission of customers.

The Queue Manager stores the order of customers. To achieve this, the Queue Manager will store queues for customers and enqueue customers when requested. Specifically, enqueueing should be done in such a way that the customers in the queue are in arrival monotonic order, i.e. using a FIFO queue. However, when customers have different levels of privilege the situation gets more complex. In the case of queueing, for any two enqueued customers having different levels of privilege, customers with a greater level of privilege should experience a shorter wait compared to customers with a lower level of privilege. Simply appending customers to the end of a list is no longer a suitable solution when customers have different privilege levels. A different data structure needs to be chosen.

Any data structure keeping track of queues should have the following properties:

- Fast head removals: to remove the customer at the head of the queue for admission.
- Fast insertion: to add a customer to the appropriate place in the queue.
- Quick value queries: to find the appropriate place where to put new customers in the queue.

Preferably all operations have the smallest time complexity as possible, preferably $O(1)$. Two immediately suitable options exist for these queues, lists and trees. When the number of privilege levels is small, one list per privilege level can be created. Admission is as simple as storing a pointer to the first non-empty queue with the highest privilege, dequeuing a customer from that list and update the pointer when needed. When the number of privilege levels is large, trees are more suitable.

When using trees as the queue data structure, the privilege level needs to be used to determine where the customer needs to be inserted. When multiple customers have the same privilege level, the newly arrived customer needs to be added to the queue in such a manner that he/she gets admitted after all customers of equal privilege that are already present in the queue.

Trees that are suitable are either efficient in memory datastructures, or should easily serialize into the used datastores. Regardless of the tree chosen, the time complexity of operations must be as low as possible.

Every shop requires two queues, one for admission into the store and one for admission to payment services. From the queueing perspective, there is no difference in the operation of both queues. Any queue implementation that is agnostic to the lifecycle phase the customers it contains are in suffices as an implementation for the shop queues.

The two functions described at the beginning of the section can be performed in a stateless manner. Data structures like linked lists or trees can be stored in most databases and queries on these data structures should therefore be easily executable in a stateless manner as well.

Provided interfaces The Queue Manager provides the following interfaces to other components:

- `AddCustomer(ShopIdentifier, Phase, CustomerIdentifier)`
- `AdmitNext(ShopIdentifier, Phase) : CustomerIdentifier`

Required interfaces The Queue Manager requires no interfaces.

Admission Controller

With the Queue Manager being responsible for ensuring customers are placed in queues, the Admission Controller needs to ensure customers actually get added into, and admitted from these queues. The main issue with this is that potentially a large number of people needs to be admitted within a certain period. Suppose the administrator desires that 1000 people are admitted per minute, either the Admission Controller needs to be triggered 1000 times per minute, or when this is impossible, admit customers in groups. For instance, admitting 50 customers every three seconds, still achieving 1000 admitted customers per minute. Even though this grouping lowers overhead, it comes at the cost of a greater jerk in admission.

Admission policies consist of two parts: maximum number of customers per shopping phase and a rate-limit on the number of customers that can be admitted within a certain period. To ensure the upper bound on customers does not get exceeded the Admission Controller is interested in the following events:

- A shopping customer finishes shopping: decreasing the number of shopping customers.
- A shopping customer departs: i.e. a customer leaves with or without product, decreasing the number of shopping customers.
- A customer arrives: when the maximum amount of browsing customers has not yet been reached, this new customer can be admitted immediately.

When no rate-limit is set, customers can be admitted when a departure occurs or immediately after arrival when the maximum bound has not yet been reached. When shop admission is rate-limited it is insufficient to only listen for these events, since these only pertain to customer lifecycle events and arrive irrespectively of any rate-limit the administrators want to impose on the shop. Consequently, rate-limiting needs a periodic event generator that reliably signals when a specific amount of time has passed, a ticker. Assuming the rate-limit is 60 admitted customers per minute, the ticker has to send an event every second and the Admission Controller then checks whether to admit the next customer. However, since the rate limit can be very high, it might still be needed to admit customers in batches, to reduce the number of ticks. So, for the example, it can be advantageous to only tick once every five seconds and then admit up to five customers. This lowers the ticking overhead without decreasing customer throughput though, as described before, increasing jerk. Therefore, the choice of batch size is important to ensure a smooth admission is achieved. When a rate-limit is needed, only the ticker events should admit customers, all other events should only be used for accounting.

When some event, either ticker based or lifecycle based, is received by the Admission Controller, it needs to determine whether to admit the customer. For this, the Admission Controller needs to count the number of customers in each phase of the lifecycle, the occupancy statistics. For ticker events, the Admission Controller needs to use these counts to determine whether to admit the next customer. For lifecycle based events, the Admission Controller needs to use the counts to determine whether to admit the customer immediately, or whether to enqueue the customer.

Due to the fact that a ticker based approach is needed, it is best to not make the Admission Controller fully stateless, since these tickers potentially tick many times per minute, or even per second, to achieve the desired smoothing effect of the rate-limit. Making this component stateful means that a different approach to scaling is needed.

The Admission Controller can make use of a polling architecture and partition every shop between all nodes in such a way that no two Admission Controller nodes are admitting customers for the same shop. Every Admission Controller node, regularly checks all shops within their partition for updates to the admission policies. By querying the datastore and using a (preferably) simple consistent hashing algorithm, determine which policies are in their partition. When a Admission Controller node detects a new shop, it needs to retrieve the occupancy statistics for that shop since it could be possible that another Admission Controller node already processed customers for that shop.

Requirement ce1 states that customers can be admitted immediately upon arrival. When new customers arrive, the Admission Controller is informed by the Customer Container of arrival. The Admission Controller must then decide whether to immediately admit the new customer, or to add it to a queue. Only after the Admission Controller has either decided on direct admission, or has received confirmation from the Queue Manager that the customer has been added to the queue, can it inform the Customer Container of the new status of the customer. Since the Admission Controller decides what happens when a customer arrives, it is the authority which decides when a customer has arrived.

Any change requested by the customer to the Customer Container cannot be applied directly, the Customer Container must await the response of the Admission Controller.

Provided interfaces The Admission Controller provides the following interfaces to other components:

- `ChangePhase(CustomerIdentifier, ShopIdentifier, LifecyclePhase)`

Required interfaces The Admission Controller requires the following interfaces:

- `Queue Manager :: AddCustomer(ShopIdentifier, Phase, CustomerIdentifier)`
- `Queue Manager :: AdmitNext(ShopIdentifier, Phase) : CustomerIdentifier`
- `Customer Container :: ChangePhase(CustomerIdentifier, LifecyclePhase)`
- `Log Aggregator :: CustomerPhases(ShopIdentifier) : map[Phase]Integer`

Customer Container

The Customer Container is the entrypoint of customers into the system, responsible for taking in all requests issued by customers, checking whether they are permissible to be executed with respect to the customers's lifecycle phases and issue requests to other components.

As the Customer Container is the entrypoint for customers into the system, a method is needed to detect the arrival of customers. Since content delivery is explicitly left out of this project's scope, there is no immediate way of detecting customer arrival. If content-delivery was taken into account, the retrieval of the content used to serve the customer user interface could be used to detect customer arrival.

It is expected that any implementation of the system is informed by the user interface of arrival. Interfaces will be specified which deal with customer interaction. One advantage of this approach is that it makes the system user interface agnostic. Any user interface adhering to the specified interfaces can make use of the services provided by the entire system.

It is desirable to know the amount of customers that are actually currently shopping, i.e. the amount of connected and admitted customers. When only the status of customers's shopping runs is stored, only an upper limit on the amount of customers per shop can be given, even though a more reliable metric is desired. Users (customers) cannot be trusted to reliably inform the system of their departure from the shops. Since they could simply have departed without sending a notification of departure or due to hardware failure cannot continue their shopping runs. This is an extension of the argument that, in general, user input cannot be trusted, as suggested by Fowler[18].

Therefore, a mixed stateful and stateless approach is needed to handle customer sessions. All customers need to remain connected to the system and send liveness probes to show they are still connected. This liveness check can be achieved through a myriad of approaches, for instance sending regular heartbeats over a persistent TCP/Web socket or other networking technology. When either the heartbeat stops or the connection is dropped/-closed the customer has disconnected. This however does not mean that the customer has consciously departed the shop, since it is possible the customer is experiencing hardware failure. This persistent connection also ensures that the Customer Container can serve as a hot cache for the customers's sessions, lowering the request duration for every customer request, since the session of that customer does not need to be loaded from the (persistent) datastore for every request. Nevertheless, all changes resulting from customers's actions should be persisted immediately to ensure these changes are not lost. When a customer reconnects after a failure, that customers's session can be retrieved and the customer can continue issuing requests.

Part of the recovery of sessions for disconnected customers is that the contents of the basket must be retrievable. Even though customer user interfaces could store the basket contents, ensuring the recovery procedure sends the current basket (as stored by the Reservation Controller) ensures that any session using any user interface can recover.

As a consequence of the need for recovery, customers's sessions need to be uniquely identified, so immediately after customers have arrived, they should be issued a unique identifier which is linked to their session.

The Customer Container is also responsible for sending status changes of customers to the Log Aggregator since this component registers the changes of customers. Another entity interested in certain phase changes is the Reservation Controller. Since it needs to ensure that when the customer lifecycle phase changes, that customers's shopping basket is updated. Specifically, for the transitions in both lifecycles that have the same name. The product lifecycle is presented in Section 3.1.2.

Provided interfaces The Customer Container provides the following interfaces to other components:

- `ChangePhase(CustomerIdentifier, LifecyclePhase)`
- `PaymentSuccess(CustomerIdentifier)`

The Customer Container also needs to provide an interface to the customers with the following functionality:

- `Arrive(ShopIdentifier) : SessionIdentifier`
- `Return(SessionIdentifier)`
- `Reserve(Product) : Reservation`
- `Release(Reservation)`
- `Finish()`
- `Leave()`

The Customer Container also needs a way to signal the customer of changes, most notably of status changes for when the customer is forced to attempt payment since its browsing window has expired.

Note, this does not contain any methods to handle payments, payment services are not taken into account for this project. It is assumed an implementation sends a customer to payment services and receives proper feedback on payment success.

Required interfaces The Customer Container requires implementation of the following interfaces:

- `Log Aggregator :: CustomerStatusChange(ProductIdentifier, PhaseChange) : Boolean`
- `Reservation Controller :: MakeReservation(ProductIdentifier, ShopIdentifier) : Reservation`
- `Reservation Controller :: UndoReservation(Reservation) : Boolean`
- `Reservation Controller :: Extend(ProductIdentifier, ShopIdentifier, Reservation) : Reservation`
- `Reservation Controller :: FinishBrowsing(CustomerIdentifier)`
- `Reservation Controller :: SucceedToPay(CustomerIdentifier)`
- `Reservation Controller :: FailToPay(CustomerIdentifier)`
- `Reservation Controller :: RetrieveReservations(CustomerIdentifier) : [Reservation]`
- `Admission Controller :: ChangePhase(CustomerIdentifier, ShopIdentifier, LifecyclePhase)`

3.1.2 Stock module

The stock module concerns itself with ensuring instances of products can be issued to customers, while ensuring that policies setup by administrators do not get violated and most importantly stocks of products are not exceeded. For this, the Stock module will try to emulate shopping baskets. In a physical shop, instances of products can be picked up from shelves or a different warehousing system. In online shops, physical products do not exist, only descriptions of products exist which makes keeping track of specific instances of products non-trivial.

As with the shopping runs, product instances also encounter a lifecycle of sorts. Assume a virtual shop exists containing all instances of products, all products in this shop would be called “available”. When a customer, after browsing the shop, has decided on a product instance, it is placed in the customer’s basket, making that product instance “reserved”. After the customer has finished shopping and has attempted to pay for the product instances (reservations) in their basket, the status of all product instances in that customer’s basket change from “reserved” to “pending”. When the initiated payment succeeds the status of the product instances change from “pending” to “sold”. This lifecycle can be seen in Figure 3.3.

It is also possible for the state of products to move back to a previously held state. For instance, when a customer changes his mind and wants to return a “reserved” product instance, that instance becomes “available” again. Similarly, when a payment fails, all “pending” product instances which could have been paid through that payment will become “available” again.

The handling of stocks consists of two distinct responsibilities: keeping track of the number of product instances per phase of the lifecycle, and keeping track of the actual instances of products and deciding whether new instances can be reserved by customers. These two responsibilities will each be handled by different components, the Shop Keeper will keep track of the number of products per lifecycle, while the Reservation Controller will be responsible for keeping track of the product instances and for handing out new product instances.

It might seem that the Reservation Controller has too many responsibilities. However, splitting the storing of product instances and the issuing of new instances into two components would result in the component storing

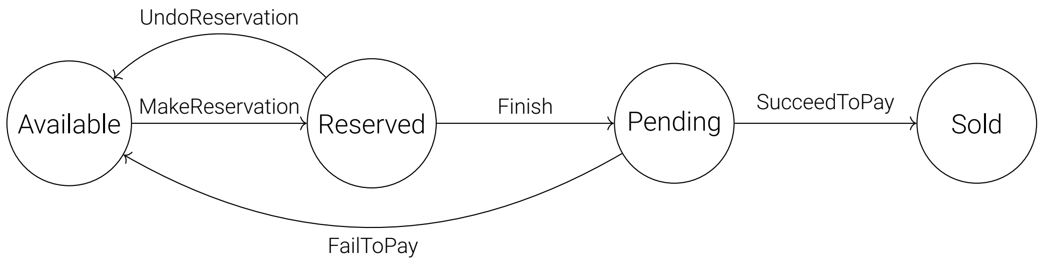


Figure 3.3: The product lifecycle.

the instances having too little responsibility. Furthermore, for every newly issued product, the entire shopping basket for the requesting customer needs to be retrieved, which makes the issuing component too dependent on the storing component to justify keeping them separate. Keeping these two responsibilities together was deemed the best solution.

Shop Keeper

The Shop Keeper is concerned with keeping track of products and the statuses of those products, and ensure stocks are not exceeded pertinent to the requirements setup by the administrators. There is no need for this component to be stateful, since for every request it can retrieve the applicable policies and number of products in each class. As described in the introduction, the system needs to recognize the following lifecycle phases:

- Available: not yet purchased, nor put in basket.
- Reserved: added to basket by customers.
- Pending: products for which payment is attempted, but the payment did not yet succeed.
- Sold: products for which a successful payment exists.

The lifecycle and the transitions between lifecycle phases can be found in Figure 3.3.

In a physical shop all products are uniquely identifiable since they are physical products and therefore are countable. For an online shop - where products might be converted into physical products eventually - there is no need to link the physical products (for instance, products stored in warehouses) to digital instances of products. Therefore, the system only needs to keep count of the number of products given out per phase.

The Shop Keeper will keep track of these counts and when the Reservation Controller requests a change, the Shop Keeper checks whether stocks suffice and either process or deny the request when no policies get violated and stocks are sufficient.

For instance, when for a product all phases need to be considered and a request comes in for reserving n product p, the Shop Keeper can simply retrieve the stock s and number of reserved r, pending p, and sold q stock for p. Shop Keeper should only process that request when $s - r - p - q \geqslant n$, note the structure of this inequality is dependent on the applied policies. Processing in this case would mean increasing r by n. Shop Keeper should be able to handle the following phase changes, where each phase change is applied to a single product for some number n of that product:

As explained in the introduction, when possible every component should perform in a stateless manner. For every phase change request, the Shop Keeper can check the number of products per phase, and the policies subject to that product whether that change request can occur. This operation can yield race conditions when not implemented carefully. Therefore, it is required that either the datastore supports long-lived transactions such that both the retrieval of the data and the (potential) update of the data occurs atomically. Or another locking mechanism must be implemented to ensure that data does not change. A high level of consistency is needed. Consistency models are discussed in a later section.

Just handling these phase changes is not enough, the Shop Keeper also needs to ensure that the statistics are kept up to date. A different component (Log Aggregator) is responsible for aggregating changes in lifecycle phases, so the Shop Keeper should send these changes to the Log Aggregator. There are two main different techniques that can be used to send the number of products that changed per phase to the Log Aggregator: immediately or deferred.

Sending the changes immediately when they occur yields the highest accuracy, but at the cost of performance. Sending them in a deferred manner (on a fixed period, or when a threshold value is reached) has two drawbacks.

The first is that large changes can get averaged out within a time period. Both a large increase and an equally large decrease within a single period will not result in a reported change, or at least will the reported change not be as large as the actual changes. This can be mitigated by also defining a threshold value, which is used to send the changes when the current change is greater than the defined threshold. Therefore, choosing period length or threshold value becomes very important. The second drawback is that now state is being kept. When the Shop Keeper experiences a crash, it might not have sent all the changes to Log Aggregator, resulting in a need for recovering from this failure by comparing the actual values for each product and each phase with the values reported to Log Aggregator, resulting in a more complex interaction between Shop Keeper and Log Aggregator. Therefore, it is advised to immediately report the values, since if the Shop Keeper is a stateless component, the number of deployed nodes can be scaled up, in order to mitigate the performance penalties.

The following phase changes and associated actions should be recognized by the Shop Keeper to ensure that the customer lifecycle can properly influence the product lifecycle. Note, the Shop Keeper itself is not responsible for ensuring these lifecycles are kept in sync. This has to be done by the Customer Container and Reservation Controller. The Shop Keeper only keeps track of products and the amount of products in each of the product lifecycle phases.

- `MakeReservation`: increase the reserved count.
- `UndoReservation`: decrease the reserved count.
- `FinishBrowsing`: decrease the reserved count and increase the pending count.
- `SucceedToPay`: decrease the pending count and increase the sold count.
- `FailToPay`: decrease the pending count.

For every change the number of products that needs to change can be included. This is to reduce overhead. When customers finish shopping, all products in the customers's basket must be transitioned from "reserved" to "pending" to ensure that the product lifecycle is kept in sync with tho customer lifecycle. When a customer, for instances, finishes shopping instead of having to request a transition for every reservation seperately, the Reservation Controller can now request the Shop Keeper to change all product instances of the same type at once.

Provided interfaces The Shop Keeper provides the following interfaces to other components:

- `ChangeProduct(ProductIdentifier, PhaseChange, Integer) : Boolean`

Required interfaces The Shop Keeper requires the following interfaces:

- `Log Aggregator :: ProductPhaseChange(ProductIdentifier, map[Phase]Integer)`

Reservation Controller

In order to keep track of product instances, the Reservation Controller will be generating "reservations" for products, and using the interfaces provided by the Shop Keeper to ensure that stocks are not exceeded.

In physical shops, product instances are easily tracked, since these are physical and can be made unavailable to other customers by simply taking them and putting them in - for instance - a shopping basket. As stated in Section 1.1 for every ticket (or product) sold, Eventix needs to know the exact products that where sold within that ticket. This is an important consideration for the design of the Reservation Controller.

For physical shops, a similar situation would not be an issue. Combined products can be physically linked together ensuring there is an automatic relation between product instances and sub-product instances. For online stores, there is no obvious way to track any product instances. One possible technique is to keep count of the number of instances of all products customers have put in their basket. This is a good solution as long as products are not compound. For compound products, this idea can be extended to include keeping track of the specific number of sub-products per product. As long as products can be nested a small number of times this is a good solution. However, when the level of nesting increases, a better solution is to issue and keep track of "reservations", random identifiers linked to a specific product. When the reservation is for a sub-product, the reservation can also be linked to the reservation it belongs to.

Sub-products had the largest influence on the design of this component and the designed interface. Two problems need to be taken into account when dealing with (sub-)products. The first is determining validity of a reservation subject to the setup constraints on the associated sub-products. The second problem is how to resolve conflict between customers. Customers are said to be in conflict when they want to place n reservations of a product with an availability a where $a < s$. Individually, these problems are not difficult to solve, but they combine in a subtle fashion in such a way that the combination is much more difficult to solve.

To ensure that sub-products reservations adhere to the bounds setup by the administrators, a method needs to be devised to issue reservations subject to these bounds. Multiple different techniques can be considered, each having different advantages and disadvantages. Selecting the best technique is heavily dependent on the specific way the bounds are setup. Possible techniques:

1. Issue "complete" reservations, a reservation containing not only the reservation of the product, but also the reservations for all sub-products. When requesting the reservation, the requesting customer must provide the structure of sub-products that should be part of the reservation. A "complete" reservation can only be handed back in its entirety.
2. Automatically issue reservations, using some automated process to decide which sub-products to issue reservations for in order to satisfy all bounds.
3. Use an iterative process, the customer first needs to reserve some product, and then using that reservation can add sub-products to that original reservation.

For handing back reservations, similar techniques can be devised. Note, regardless of the chosen technique the entire shopping basket and all relationships between reservations need to be stored.

When customers are in conflict the pigeonhole principle states that only a subset of all requested reservations can be issued. A method needs to be devised to determine which customers are awarded the reservations. These so called tiebreakers are specified in requirements tb1, tb2, tb3, tb4, tb5. Since most of these tiebreakers require communication to determine which customers are awarded reservations, these are left as future work due to this complicating the (stateless) requests considerably. Only one tie breaker will be considered, tb1.

Releasing any reservation is trivial since this will only increase the availability of products. Worst case, handing back a reservation will break one or more of the defined bounds on the number of products. For requesting new reservations however, caution must be taken on how these reservations get issued. Resolving competition by not awarding any reservation to any of the competing customers reduces the complexity slightly, since it should be easy to implement.

Due to the choice of this tiebreaker handing out exactly the stock in number of reservations should take longer than strictly necessary. As long as stocks are higher then concurrent demand for a product, meaning no competition occurs, reservations can be issued when requested. However, as soon as competition occurs, the rate of handing out reservations slows down compared to the number of requested products. This is not a major issue, since reservation requests occur in a probabilistic fashion and therefore the number of issued reservations should converge on the available stock eventually.

The process chosen for issuing reservations influences the duration of this convergence. When an automated process is used, the number of products that two customers can potentially be in competition over increases to the number of products the automated process is trying to reserve. Therefore, only the iterative process is deemed viable when using the chosen tiebreaker. This enforces that for any reservation request, customers can only be in conflict over a single product. Another advantage to this approach is that customers are now in full control over the issued reservations. This is useful since depending on the structure chosen for defining the bounds on sub-products, it is possible that automated systems generate a set of reservations the customer does not want, resulting in many unnecessary reservation undo requests issued by the customer.

When the Reservation Controller is notified by the Customer Container of a customer's lifecycle change, it needs to signal the Shop Keeper to ensure the counts can change for the reserved product instances. The phase changes defined by the Shop Keeper can be used to handle these changes. These phase changes were already chosen in such a way that they can be used to ensure that the counts for the number of products per lifecycle phase can transition together with customer lifecycle changes. Since the Shop Keeper does not have the notion of reservation, or of compound products, an extra operation needs to be defined, the `ExtendReservation` operation. The entire list of operations:

- `MakeReservation`: when customers want to place product in their basket.
- `UndoReservation`: when customers want to return product from their basket.
- `ExtendReservation`: when customers want to extend a reservation.
- `FinishBrowsing`: when customers finish browsing and wants to attempt payment.
- `SucceedToPay`: when customers successfully pay for the products in their baskets.
- `FailToPay`: when customers fail to pay for the products in their baskets.

Provided interfaces The Reservation Controller provides the following interfaces to other components:

- `MakeReservation(ProductIdentifier, ShopIdentifier) : Reservation`
- `UndoReservation(Reservation) : Boolean`
- `Extend(ProductIdentifier, ShopIdentifier, Reservation) : Reservation`
- `FinishBrowsing(CustomerIdentifier)`
- `SucceedToPay(CustomerIdentifier)`
- `FailToPay(CustomerIdentifier)`
- `RetrieveReservations(CustomerIdentifier) : [Reservation]`

Required interfaces The Reservation Controller requires the following interfaces:

- `Shop Keeper :: ChangeProduct(ProductIdentifier, PhaseChange, Integer) : Boolean`

3.1.3 General module

The general module consists of services which do not fit within any of the other modules, or provide services to components in both of the other modules.

Log Aggregator

Two components (Shop Keeper and Customer Container) generate log events which need to be used to generate statistics. The Log Aggregator aggregates these log events and stores them in such a way that (complex) statistics can be generated from them. Since the Admission Controller requires a specific statistic for its operation (the number of customers in each lifecycle phase for each shop), the Log Aggregator will also disclose this specific statistic. Other statistics need to be disclosed using a separate component. Since a plethora of different statistics can be conceived of and these statistics have not been specified, therefore the current prototype system does not contain a component (or multiple components) that provide such statistics. In Section 4.5.3 however, a tool will be presented to disclose statistics to the Administrators, but only to show that statistics can be generated.

The choice of both aggregating the log events and disclosing the statistics on shop occupancy in the same component was made to ensure all components are somewhat balanced in terms of complexity and responsibility. This component can also easily be expanded to encompass more responsibilities and even functionalities like resolving conflicts with customers or shop operators whether certain transactions/operations happened, although an exploration of this is left as future work.

As with other components, the Log Aggregator can and should be implemented in a stateless manner. i.e. immediately storing all received statistics in a persistent datastore. The log events received are phase changes for customers and for products.

The interfaces provided by the Log Aggregator contain a mapping from a customer or product lifecycle phase to an integer. This abstraction is made to keep the interface specification concise.

Provided interfaces The Log Aggregator provides the following interfaces to other components:

- `CustomerStatusChange(ShopIdentifier, map[Phase]Integer)`
- `ProductPhaseChange(ProductIdentifier, map[Phase]Integer)`
- `CustomerPhases(ShopIdentifier) : map[Phase]Integer`
- `ProductPhases(ProductIdentifier) : map[Phase]Integer`

Required interfaces The Log Aggregator requires no interfaces.

Policy Distributor

The Policy Distributor serves the purpose of ingesting settings from the administrators and distributing these to all components in the entire system. Since almost all components perform in a stateless fashion and retrieve needed settings from datastores, interactions with these components by storing the settings can also be performed by interactions with the datastores. The Policy Distributor will simply register administrator specified policies in the store and the components in need of policies will retrieve them from the datastore.

All settings are rooted in the concept of shop. Therefore, when a shop is no longer needed, settings for products sold through, and admission policies of that shop are no longer needed. Furthermore, these updates only occur intermittently, reducing the need for having separate updates and deletions for the policies and products. Consequently, the create/update and delete operations can create/update and delete the entire shop at once. Since deleting and retrieval of shop settings is not required by the requirements, they are not added to the interfaces.

Provided interfaces The Policy Distributor provides the following interfaces to administrators:

- `SetStore(ShopIdentifier, [Product], AdmissionPolicy, PaymentPolicy)`

Required interfaces The Policy Distributor requires no interfaces.

3.2 Scaling

As specified, the system needs to be scalable this was one of the driving factors behind designing components to be stateless. Since stateless components can be scaled by simply spawning more instances of the same type. They are limited by the throughput on the datastores, since for every request the state needs to be retrieved from, and when needed updated in the datastores.

Each component has a maximum capacity, an upper bound on the number of tasks it can perform in a set period. Since most components are stateless, a node of a component operates independently from other nodes. Linearly changing the number of running nodes for a component, results in a linear change of capacity of that component. Since each component belongs to exactly one module, changing the capacity of any component, changes the capacity of that component's module.

When the load of the system changes, the capacity of the system must also change to deal with the load. Depending on the place where the load changes, different components must be scaled. If there is a change in the number of arriving customers, the capacity of the order module should change accordingly. For instance, when the system is encountering a flash crowd, or a flash crowd event has been resolved, the capacity of the order module needs to be changed. Increasing the capacity of the stock module will not result in an increase in the ability of coping with a flash crowd. Vice versa, when the amount of concurrently browsing customers changes, the stock module's capacity needs to change.

Logically, an upper bound on the number of instances per component that can run simultaneously exists, or more precisely an upper bound on the maximum load the system can support exists. This bound is now determined by the throughput and maximum number of simultaneous connections of the datastore.

3.2.1 Resiliency and consistency

Since all state changes get stored immediately, datastores need to be constructed in such a way that they achieve the desired resiliency and consistency requirements per component. Many different viewpoints on consistency and consistency models exist, each having its own advantages. An overview of these models is presented by Bermbach and Kuhlenkamp[5], and this overview is used to determine which models to choose.

A dichotomy exists between the required consistency models for all components. Some are sufficiently serviced with a very weak consistency model, while others require a greater level of consistency.

For every component the impact of encountering stale data is different:

- Customer Container: low impact. Every customer is connected to exactly one Customer Container node. All updates are stored in memory and are persisted. Stale data should therefore only be encountered when the system is trying to recover the session of a disconnected customer. When session data is stale, either the entire session cannot be found, or the retrieved lifecycle phase is wrong. When the entire session is missing the entire recovery procedure can be retried. When the lifecycle phase is wrong, in the worst case the customer might have to wait to be admitted again.
- Admission Controller: low impact. The Admission Controller polls the database for changes to policies and applies them to the local state. As long as the data is updated eventually, these policy changes will also be applied eventually. It can be assumed, that all policy settings in the database were valid at one point in time. Even though it is unfortunate the Admission Controller is applying stale settings, it is applying settings that were valid once.

- Queue Manager: high impact. If the Queue Manager would encounter stale data it might add a customer in the wrong spot in the queue, or return customers to the Admission Controller for admission that have already been admitted.
- Reservation Controller: medium impact. The Reservation Controller deals with everything related to reservations with respect to constraints on products. Taking issuing a reservation as an example, when there is stale data the Reservation Controller might decide incorrectly that some reservation can or cannot be issued. Not issuing a reservation poses no problems, since the request can simply be retried by the customer. A reservation that is issued when it should not have, can be fixed by checking the entire customers's shopping basket again when placing the order.
- Shop Keeper: high impact. The Shop Keeper must ensure stocks are kept. Out of the four fields the Shop Keeper tracks per product (stock, reserved, pending, sold), only stale stock should not result in operational problems by the same reasoning as how the Admission Controller handles stale data. For the other fields however, stale data can be detrimental to the operations of the Shop Keeper. Taking the reserved count as an example, an under or over reporting of this field, will result in an under or over reporting of the product's availability. Possibly resulting in an over or under sell of that product. The same reasoning applies to the other two fields.
- Policy Distributor: low impact. The Policy Distributor only writes data to the datastores. Writing data is idempotent, since the Policy Distributor sets all settings for a shop at once.
- Log Aggregator: low impact. All log items received should be logged. But this data is only appended, there is no stale data to be read.

The impact of stale data on all components except the Reservation Controller, Shop Keeper and Queue Manager is low. They therefore require comparatively little in terms of consistency, since all data stored only eventually needs to be present and available to all nodes. For these components comparatively weak consistency is sufficient, since there are no hard requirements on the data they write, if they write data at all.

Taking a data-centric viewpoint, most components are adequately served by eventual consistency since either only a single node needs the data, is updating the data, or is only appending data.

However for the Reservation Controller, Shop Keeper, and Queue Manager the situation is different. These need a high level of consistency guarantees. Therefore, the datastore of these components is at least sequentially consistent and preferably provides linearizability of operations. When some of the stricter requirements like monotonic arrival are relaxed, serializability of operations could also be suitable.

The CAP theorem[19] states that it is impossible for distributed data to have more than two of the following three properties:

1. Consistency: the data that is read, is the data that was written last.
2. availability: every request receives a response, without guarantees on staleness of data.
3. Resistance to network partitions: the system containing the data continues to operate even when communication is interrupted between nodes in the system.

By consequence, the stronger consistency models sacrifice the availability. However, having low availability can result in highly consistent and network partition resilient datastores. Both very desirable features for the datastores for the Reservation Controller, Shop Keeper, and Queue Manager. Unfortunately, since interactions with datastores are mostly write heavy, the throughput of components can be quite limited. When components are made to be stateful, or data storage patterns are carefully chosen, it can be possible to achieve a higher throughput.

For instance, ensuring stocks are kept (Shop Keeper), if the available stock was "distributed" between Shop Keeper nodes, in such a manner that each of n nodes can give out $\frac{1}{n}$ of the total stock, congestion on the datastore would be lowered since only intermittent updates on stock changes have to be communicated. Research on these kinds of solutions, to increase possible throughput in the system, is left as future work.

3.3 Composition

Up to this point the components are only presented as separate entities. In the introduction, it was mentioned that the chosen architectural style is a mixed SOA and Publish-Subscribe approach. This choice was made since the order and stock modules have different requirements on communication when multiple nodes of every component are running. Figure 3.1 provides an overview of the modules, the components, the interaction between components, and interaction between the actors (customers and administrators) and the components. For simplicity, the datastores and interactions with the datastores have been left out.

3.3.1 Order module

The order module has to ensure that shopping runs for customers are managed properly. The main driving force for the architectural style of the order module is the behavior of the Admission Controller and the Customer Container. Every store is assigned to a single Admission Controller node and every customer is assigned to a single Admission Controller node. These nodes therefore operate in a referentially decoupled manner. Customer Container nodes are not aware of shops, so are neither aware which Admission Controller node takes care of admission into the shop of any of the Admission Controller nodes's customers. Vice versa, an Admission Controller node is not aware of customers, so neither which Customer Container nodes takes care of communication with that customer. A Publish-Subscribe interaction style[14] is the standard solution for such a solve this.

When a customer arrives, the Admission Controller must always decide what happens with the customer. In case immediate access is needed, the customer must immediately be admitted. When admission is not immediate, the customer must be added to a queue. Consequently, when it is decided a customer must be added to a queue, the Admission Controller must contact the Queue Manager to add this customer to the queue. When the Admission Controller decides a customer can be admitted, it requests the Queue Manager to retrieve the head of the queue, return it to the Admission Controller, and remove the head. To ensure that customers are not admitted multiple times, the retrieving and removal of the head must occur atomically.

This structure also decreases the amount of race conditions regarding admissions into and from queues. It is impossible for a customer to be admitted from a queue before being added to the queue, since both actions are initiated by a single Admission Controller node for any single shop.

In principle, any event generated by any customer of any store can be sent to all Admission Controller and Queue Manager nodes. This is highly inefficient, since not every node might be interested in that event. Therefore, these events need to be scoped somehow. For instance, admission for any shop is handled by at most one Admission Controller node, sending all these events to all Admission Controller nodes will result in unneeded message processing. Since customers are only shopping at a single shop, all events generated by changes in any customers's shopping run can therefore be scoped to both the customer and the shop. Both of these objects have identifiers which are good candidates to be used to scope the messages.

When using the customers's identifier as the topic, discovery of new customers becomes an issue. When customers arrive the Admission Controller cannot be aware of their arrival until it is notified. This discovery message could be sent over the same Publish-Subscribe middleware using, for instance, the shop identifier as the topic. Since shop identifiers are the second option as potential topics, it will be simpler to only use shop identifiers as the topics and send all messages related to changes in customers's shopping runs for that shop using the shop identifier as the topic. An extra advantage is that this works well with how the shops are distributed between Admission Controller nodes. They can simply subscribe to the relevant topics by using the identifiers of the shops they are admitting customers for. The Admission Controller thus does not need a separate mechanism for the discovery of new customers.

Another advantage of using shop identifiers as topics is that this ensures a predictable amount of topics, since the amount of shops is known. If the customers's identifiers would be used the number of topics is unpredictable. Since all published messages pertain to changes in the lifecycle of some customer, the used topics could be a combination of both the shop identifier and the lifecycle phase messages sent on that topic will be about. This is a viable option, but deemed unnecessary since the amount of messages per customer is low and the overhead incurred from dealing with these extra messages is negligible.

The data sent over the Publish-Subscribe middleware will be an abstraction of the structure provided by the interfaces, specifically sent message consist of the following attributes:

- The identifier of the shop (the topic)
- The identifier of the customer
- The phase of the shopping run the customer will now be in, e.g. "shopping" when the customer gets admitted

Even though the data is available in the datastores and thus in a shared data space, the messages should still include the data as described above, to ensure that the relevant components can act as quickly as possible without having to execute lookups in the datastores.

The interfaces listed in Section 3.1.1 are specified such that direct communication between these components can occur. The Queue Manager and Admission Controller are communication using some Publish-Subscribe middleware so some of these interfaces are implemented as messages sent over the middleware. Specifically:

- `Admission Controller :: ChangePhase(CustomerIdentifier, ShopIdentifier, LifecyclePhase)`
- `Customer Container :: ChangePhase(CustomerIdentifier, LifecyclePhase)`

In all cases, both the Customer Container and Admission Controller will be subscribing to and publishing lifecycle changes for customers within shops. Even though the communication between the Customer Container and Admission Controller is handled by Publish-Subscribe middleware, the contents of their communication follows a request-response structure.

For a change requested by the customer (specifically: arrive, finish) the Customer Container sends the event and awaits the reply (itself is a message) from the Admission Controller on what the actual new phase of that customer will be before applying and persisting the lifecycle phase change.

This structure must be followed since it must be possible for a newly arrived customer to "skip" a phase. It must be possible for a newly arrived customer to be admitted immediately, skipping "Arrived". Or after having finished browsing be admitted to payment services immediately, skipping "Done".

3.3.2 Stock and general module

For issuing reservations, a very clear request-response structure exists. For any change regarding stocks, something is the source of the change, for instance: a customer requesting a reservation, or the Customer Container signaling that a customer is attempting payment.

Each component handling requests in turn call other components, thus encapsulating their services, a quintessential attribute of a Service Oriented Architecture (SOA) as described by van Steen and Tanenbaum[14]. A different composition style, like Publish-Subscribe was considered, but a SOA was deemed a better fit since there is no referential coupling in the communication between the components.

Other advantages like reduced overhead (since a SOA does not require a message broker like a Publish-Subscribe architecture) and the loose coupling of components provided the final push. Due to the same reasoning also holding for the general module, it will also be composed as a SOA.

For most calls it is impossible or undesirable to add all data needed to handle the request to the request. For instance, when issuing a new reservation. When issuing new reservations, the entire contents of the shopping basket needs to be taken into account in order to determine whether the request is permissible. When this information would be included in the customer's reservation request, some manner is needed of ensuring the request is not manipulated. Some requests do contain all information needed for processing like updating shops, making a REST architecture a viable alternative to SOA. However, for all requests issued by the customer, it is undesirable to use this data since there is no way outside of comparing it to persisted state to guarantee that the customer did not maliciously modify its local state.

It might appear that it is preferable to have the Log Aggregator ingest log items from the Publish-Subscribe middleware since no explicit notifications are needed. However, this would require the Log Aggregator nodes to be aware of all shops, in order to subscribe to the appropriate topics and perform some coordination to ensure each message is only handled once, since each Log Aggregator would receive each message sent.

3.3.3 Examples

In the following section three examples will be provided to give an overview on how the components work together to achieve the desired functionalities. Specifically, an example of a shopping run and reservation of multiple products will be shown. To simplify, all interactions with datastores have been left out. Communication between the Customer Container and Admission Controller always uses the Publish-Subscribe middleware. Sent messages are depicted as direct communication between these two components.

Each example will be accompanied by a Message Sequence Chart (MSC). Within these MSCs, the names of the components will be abbreviated by taking the first letter of each word in the name, e.g. a component named "Better Component" will be abbreviated to "bc".

Shopping run

This example will focus on showing how customer's basic shopping runs work. This example shows the main interactions between the Customer Container, Queue Manager, and Admission Controller without all the details regarding stock and statistics. The following scenario is laid out and visualized in Figure 3.4:

1. Two new customers c, c' arrive at shop s.
2. The Customer Container sends messages over the Publish-Subscribe middleware to notify the appropriate Admission Controller of the arrivals.
3. The Admission Controller determines c can be admitted immediately, but c' must be queued.
4. The Admission Controller requests the Queue Manager to add c' to the admission queue for shop s.
5. The Admission Controller notifies the Customer Container that c is admitted, and c' is arrived (enqueued).
6. After some time the Admission Controller determines the next customer can be admitted.
7. The Admission Controller requests the Queue Manager to return the customer at the head of the queue and to remove that customer from the queue.
8. The Admission Controller sends a message over the Publish-Subscribe middleware to notify the admission of c'.

Note, to reduce complexity of the diagram and make it more concise, the second queuing step for the payment services has been left out since it is identical to the admission into the shop, except it admits to payment services instead.

Reserving of product

This example shows the interaction between components when interacting with the stock module. It specifically shows the interactions between the Customer Container, Reservation Controller, and Shop Keeper. Showing clearly the link between the product and customer lifecycles. The following scenario is visualized in Figure 3.5.

1. A customer c gets admitted into shop s.
2. c reserves a number of product p, and gets returned all reservations.
3. c then extends a single of the previously generated reservations, r_2.
4. Payment is then attempted, updating all products at the Shop Keeper. This is done by changing all n instances of each reserved product type at once.
5. Payment (not shown) fails.
6. After the failure occurs, all products are placed back.
7. c can only depart, no further attempts on payment can be made.

Generating statistics

This example will show how the statistics are generated. It specifically shows the interactions between the Customer Container, Shop Keeper, and Log Aggregator. Showing how the events propagate from the Customer Container and Shop Keeper to the Log Aggregator. To make the MSC more readable, the parameters for the CustomerPhaseChange and ProductPhaseChange have been left out. The following scenario is visualized in Figure 3.6.

1. A customer c gets admitted into shop s.
2. The admission event is then sent to the Log Aggregator. Note, the arrival event should also have been sent.
3. c then places a number of reservations, where each reservation event gets sent to the Log Aggregator.
4. c finishes shopping (not pictured), which gets sent to the Log Aggregator.
5. For all reservations c placed, the Shop Keeper changes the phase and notifies the Log Aggregator.

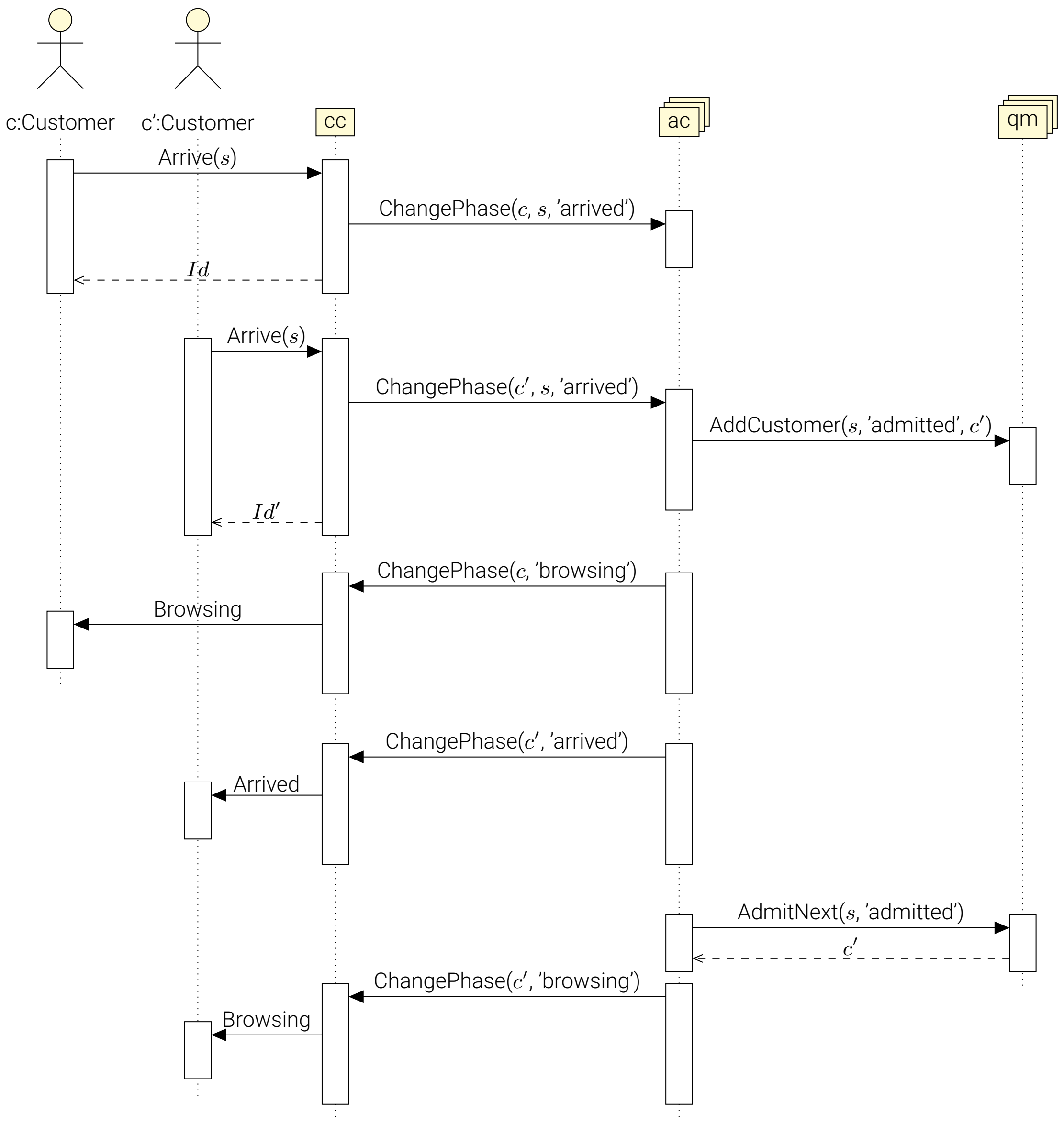

c:Customer
c':Customer
cc
ac
qm
Arrive(s)
ChangePhase(c, s, 'arrived')
Id
Arrive(s)
ChangePhase(c', s, 'arrived')
AddCustomer(s, 'admitted', c')
Id'
ChangePhase(c, 'browsing')
Browsing
ChangePhase(c', 'arrived')
Arrived
AdmitNext(s, 'admitted')
c'
ChangePhase(c', 'browsing')
Browsing

Figure 3.4: Message Sequence Chart describing part of a basic shopping run.

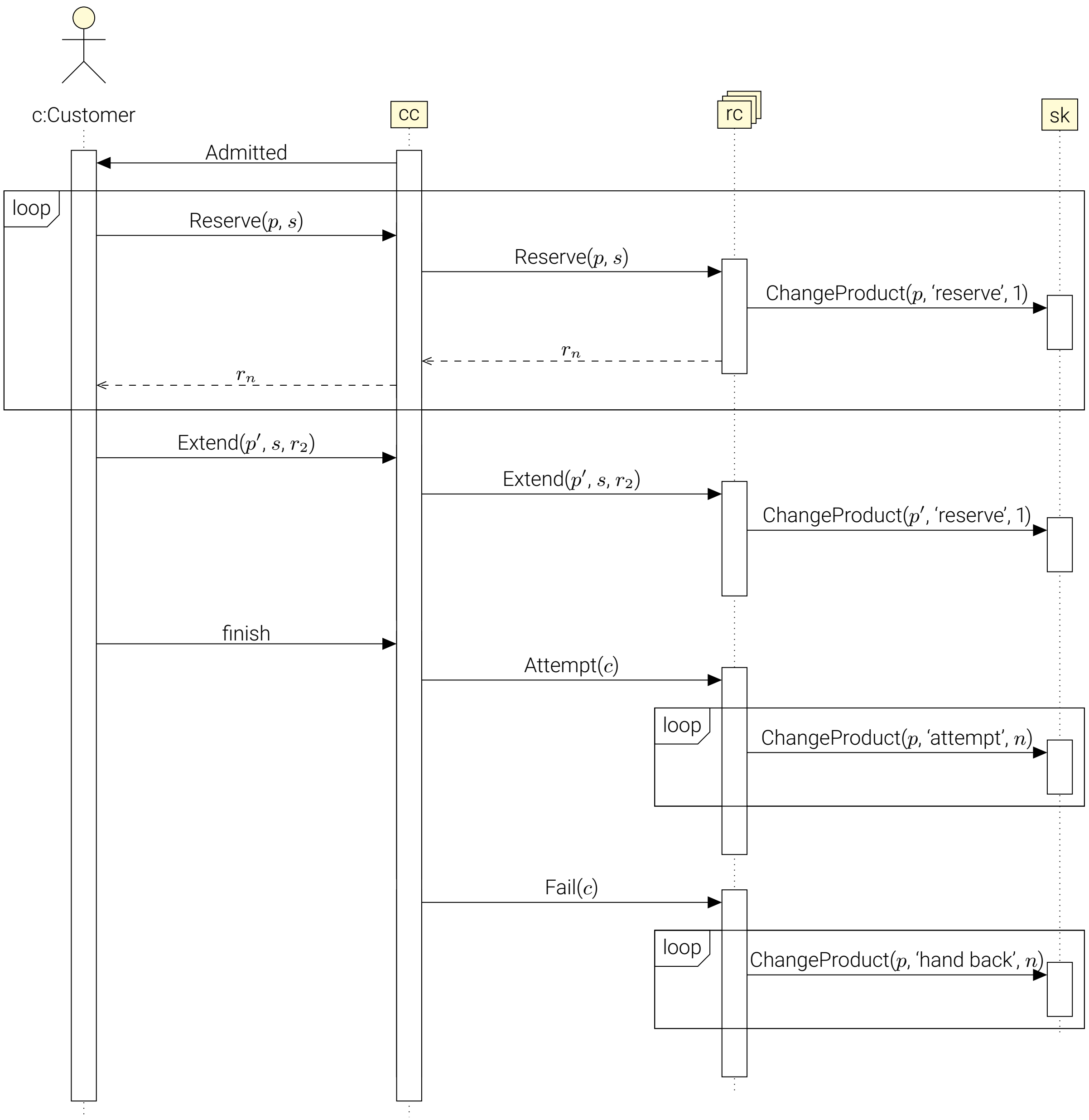


Figure 3.5: Message Sequence Chart describing reservation flow.

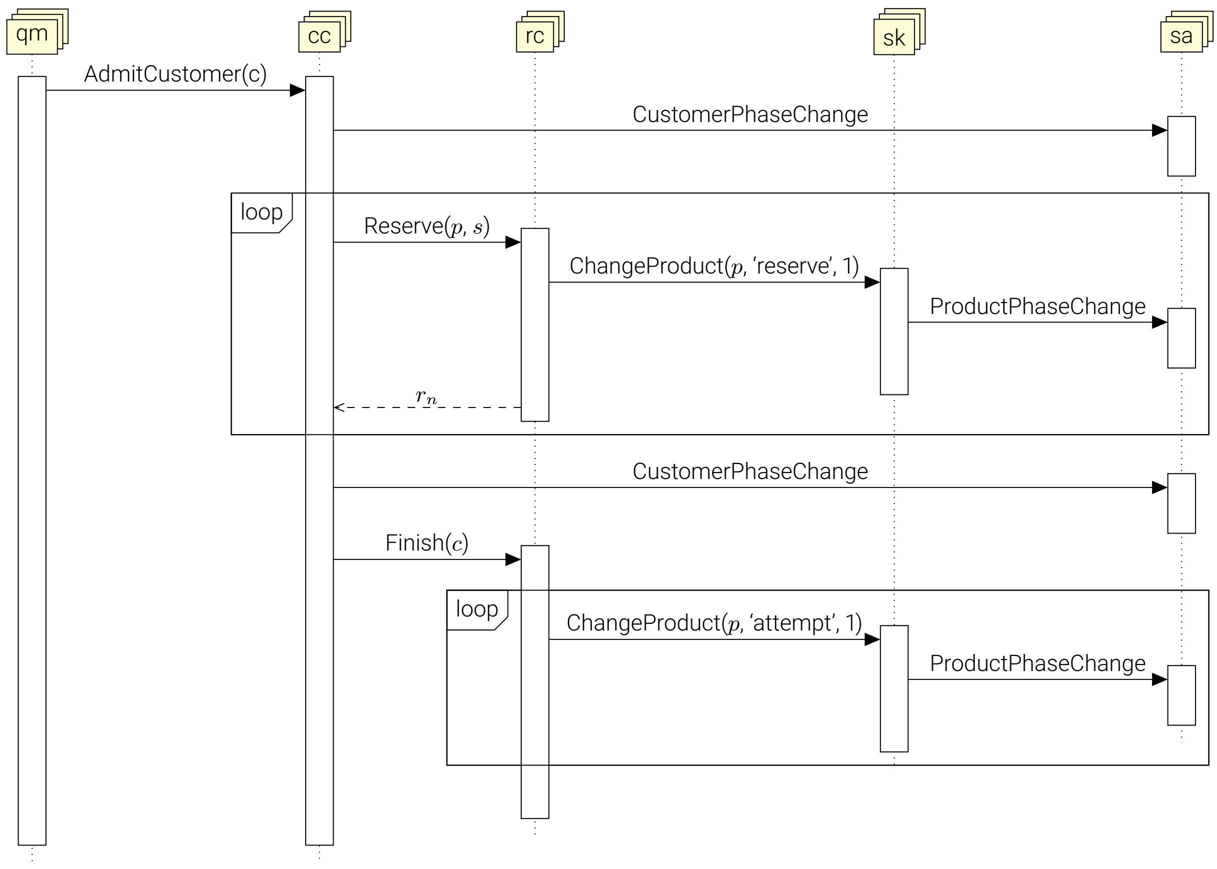


Figure 3.6: Message Sequence Chart describing the flow of statistics to the Log Aggregator.

Chapter 4

Implementation

In this chapter an implementation of the architecture described in the previous chapter is presented. This implementation provides most of the required features, but must be seen as a proof-of-concept and is definitely not feature complete. At first the commonalities between all parts of the system will be described, including communication, and database technologies. Then the implementation of all components will be discussed. Lastly, the specifics of how to deploy the system on a Kubernetes cluster will be presented.

4.1 Environment

All components are implemented in the Go (or Golang) programming language. Go is a C like programming language released by Google on November 10, 2009. It is designed specifically to create server programs deployable on multiprocessor architectures. The Go version used is 1.14.2 dated April 8, 2020. Go uses a concept called Goroutine, which is a function that can run concurrently with other Goroutines. Goroutines can be thought of as lightweight threads. Goroutines communicate using a concept called "channel". Channels are strongly typed synchronized conduits through which data can be sent. These channels can be written to by multiple producers and read by multiple consumers to facilitate creating of complex structures. Every piece of data can be read only once. When multiple consumers are reading data from the channel, sending data over that channel will be delivered to at most one consumer. As long as there is at least one consumer, delivery is guaranteed.

4.2 Deployment

The deployment environments specified in the requirements were of great influence on the chosen architecture. Consequently, the deployment environments were also of great influence on the implementation. For the implementation the task was set to not only provide a working implementation, but also determine how to properly deploy it on a Kubernetes cluster.

Kubernetes clusters provide abstractions to the infrastructure they are deployed on. If a system can be deployed to one Kubernetes cluster it should be deployable on other Kubernetes clusters with at most minor modifications to the deployment configuration. Eventix uses DigitalOcean as their PaaS provider. It turned out DigitalOcean recently started offering hosted Kubernetes clusters, making deployment of a Kubernetes cluster easy. Even though it does not matter who provides the Kubernetes cluster, the system was still deployed to a managed DigitalOcean Kubernetes cluster.

To ensure enough capacity exists to run all the tests, and to mimic an actual production environment, a three node cluster was created. In the cluster every node consists of 3 virtual CPUs, 8GB of memory and 120GB disk, resulting in a total cluster capacity of 12 virtual CPUs, 24 of memory and 480GB of disk. The components do not need to store much data themselves. Almost all data is stored in databases. Kubernetes provides a disk abstraction called "Persistent Volume", which in the case of a DigitalOcean managed Kubernetes cluster always get mapped to block storage external to the cluster.

4.2.1 Containerization

As per the Kubernetes documentation, many different container runtimes are supported[20]. At first it appeared that the runtime the Kubernetes cluster uses impacts the tools needed to construct the images. However, further inspection of the available documentation showed that the deployed runtime is only of minor influence on which types of images to run.

Docker[21] was chosen as the containerization platform since Eventix already used Docker and Docker images can be used on DigitalOcean Kubernetes clusters.

Using Docker forces requirements on building the image, namely that the containers must be based on Docker images. For building Docker images, a so called "Dockerfile" needs to be provided which contains build instructions for a Docker runtime to create the actual images. By ensuring the structure of the components is similar, only a single Dockerfile was needed. The used Dockerfile can be found in Appendix A.

After these images have been created, they need to be made available to the cluster using what Docker calls a registry. This registry contains a collection of uploaded images. As long as the registry is accessible by the Kubernetes cluster, it can then pull images from this registry. A public registry exists, called Docker hub. But since Eventix preferred not to have these images public, a private registry was used, also hosted on DigitalOcean infrastructure.

4.3 Communication

Since the system consists of a mixed Service Oriented and Publish-Subscribe architecture, the communication is also mixed. Services are either called directly or are sending messages over the Publish-Subscribe middleware. In order to keep the structure of exchanged data consistent, Flatbuffers[22] was chosen as the data interchange for both the middleware messages and as the format of data sent in the requests and responses when calling other services.

4.3.1 Publish-subscribe middleware

For the Publish-Subscribe middleware an existing project called NATS Streaming[23] is used. NATS Streaming provides a Publish-Subscribe middleware layer with strong guarantees, specifically at least once delivery, ensuring that each message is guaranteed to be delivered to every interested party. NATS Streaming was chosen since it provides high quality libraries for Go, provides an at-least-once delivery guarantee, and is easily deployable on a Kubernetes cluster using a configuration provided by the NATS developers.

4.3.2 Service communication

The communication between services occur using a Remote Procedure Call mechanism, many different RPC technologies exists. In this project, gRPC[24] (using a Flatbuffer as serialization technology) is used. gRPC was chosen over other prevalent RPC protocols, like JSON-RPC[25], since gRPC has good library support for easy integration in Go. Due to reduced overhead it provides higher throughput and quicker response times. The overhead is reduced due to efficient data (de)serialization, and the usage of HTTP/2 over HTTP for efficient transport[26]. At first, the benefits of using HTTP/2 over HTTP were considered to be negligible, since the cited source focussed on website load times and found HTTP/2 only could potentially improve these load times. After verifying the performance however, it was found that HTTP/2 outperformed HTTP for our usecase. Combining this speed increase with the more efficient (de)serialization solidified the choice for gRPC.

gRPC services require two specifications in an IDL: the format of the data transferred, and the specification of the interfaces. Appendix F contains an example of how these are structured.

4.4 Databases

As discussed in Section 3.2.1 different components have different requirements for the datastores. For simplicity sake and ease of implementation, the fewest number of different datastores (databases) was used. Two different databases, ClickHouse[27] for the Log Aggregator and Dgraph[28] for the rest are used. ClickHouse is a columnar storage OLAP database which can be queried using a SQL like language. Dgraph in turn, is a fast and scalable graph database providing distributed ACID transactions.

The advantage of both databases is that they can natively run on Kubernetes using a configuration provided by the developers of each project. However, the provided configuration of ClickHouse assumes the cluster is hosting ClickHouse to be used by services external to the cluster. Access from services internal to the cluster did not perform as expected. Therefore, ClickHouse was installed on a machine external to the cluster. Dgraph however, was installed on the Kubernetes cluster using the developer provided configuration.

ClickHouse, was originally developed by Yandex and released as open source software on June 15, 2016. It is specifically intended for low data consistency, high throughput, read heavy data processing. In other words, it is well suited for generating statistics from raw data. Each event the Log Aggregator receives is added to the appropriate table containing all events of that type, which then automatically get aggregated into tables in a format such that the desired statistics can be quickly queried. Appendix B.2 contains the schema of the tables. How the schema is used is explained in Section 4.5.3. Since the Log Aggregator also needs to provide metrics to administrators, a dashboard solution was implemented, specifically Grafana. This will be further explained in Section 4.5.3.

Dgraph, is an open source graph database created by Dgraph Labs and published in December 2015. The main reason Dgraph was chosen was the potentially complex nested structure of products, sub-products, and their constraints. Since all other data: queues, lists of reservations, and settings can also be stored as a graph. No other database was considered for storing these types of data. Dgraph uses the Raft consensus algorithm[29] to achieve consensus. Raft ensures consensus exists on which transactions to commit in which order, thus ensuring linearizability of operations is achieved.

Dgraph queries are declarative and describe the desired structure and contents of the response. For modifying data, Dgraph can use either RDF N-Quads or JSON to specify the new data. Only RDF N-Quads are used. A data change request in Dgraph consists of multiple queries and multiple mutations. The result of the query will always be the state of the data before the mutations are applied. This enables the usage of the result of the query in the mutations as well as retrieving data relevant to the mutation since the result of the query is returned after the change request has been executed.

Every query, both change requests and simple retrieve queries, need to be wrapped in a transaction. A single transaction can contain multiple queries and change requests. To achieve Linearizability of operations, Dgraph only allows one transaction to mutate the same data concurrently. If two transactions are mutating the same data concurrently, one of the transactions is automatically aborted. In order to deal with this behavior, an exponential randomized backoff retry procedure is implemented. Still this does not guarantee the transaction is executed eventually. Therefore, a transaction needs to be attempted for a limited number of times or for a limited amount of time.

Go has a concept called “context”. This context is generally used for request-response interactions and is used to provide context to the requests. Contexts are an in-memory key-value store and can be “cancelled”. When a context is cancelled the request it is associated with should be viewed as cancelled as well. Contexts can be either manually or automatically cancelled. Automatic cancellation can occur when a deadline has been reached. Any interaction with Dgraph requires a context object. When available an existing context with deadline is used, or a new one is created and sent along with the transaction. When the context deadline is exceeded the exponential backoff will terminate. In order to deal with the issue of unreliable transaction commits, while still having some guarantee on the maximum duration of any request, the deadline of these contexts is set to 15 seconds by default. Any request, including queries to Dgraph, can therefore take at most 15 seconds.

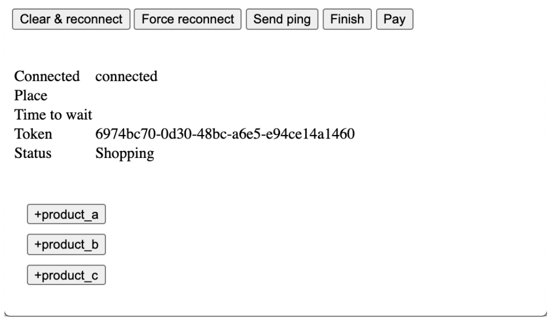
Both databases are only internal facing, only services from inside of the cluster can execute queries and request mutations on the databases. Components can access these databases directly instead of having to access them through other components. By accessing the data directly overhead on data access is removed. Exposing the databases directly to customers poses security risks since it allows any customer, including malicious customers, to modify data. To still enable customer interaction, all customer requests are issued by customers to the Customer Container. The Customer Container then forwards these requests to the appropriate components. The components then modify the data directly when needed. No database is thus exposed to customers or administrators directly.

4.5 Components

In the following section a proof-of-concept implementation of each of the components described in the previous chapter is presented. For this proof-of-concept a very optimistic approach is taken, it is assumed errors rarely happen. Therefore, error recovery and verification are not implemented.

4.5.1 Customer Container

As described in Section 3.1.1, an implementation of the Customer Container requires a persistent connection to each customer. For the connection with the customer, the proof-of-concept will make use of WebSockets since an efficient implementation was available, and WebSockets are supported by many modern browsers.



Connected connected
Place
Time to wait
Token 6974bc70-0d30-48bc-a6e5-e94ce14a1460
Status Shopping

Figure 4.1: Screenshot of the interface.

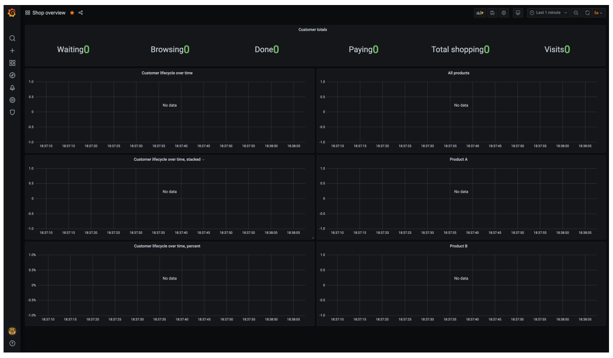


Figure 4.2: Basic dashboard built using Grafana

As described, messages sent over the Publish-Subscribe middleware are sorted by topics, where the shop identifier is used as the topic. When a new customer connects to the Customer Container, it checks whether it has a subscription to the topic belonging to that shop. Iff it does not, it subscribes to that topic.

Every request placed by a customer, is first verified whether the customer is at a point in their shopping run where placing this request is permitted before it is forwarded to the correct component. Before the response is returned to the customer, it is first persisted to ensure it cannot be lost. To ensure customers can actually recover from networking issues. The recovery procedure is implemented as described in Section 3.1.1. The only difference is that the reservations are always retrieved. Not just when session recovery is attempted.

This is done to ensure that the frontend application does not need to store any state except the identifier. Simplifying any (eventual) frontend application. For the demo frontend a very small library was written, which provides an abstraction to the actual communication. This library can be used as a base for a full implementation.

User interface

Even though content delivery was explicitly not taken into account, an interface still needed to be served for demo and testing purposes. This interface can be seen in Figure 4.1.

4.5.2 Policy Distributor

The Policy Distributor is tasked with ingesting settings provided by the administrators and ensuring these settings get stored somehow. Since this component is one of the few external facing components, it only receives requests intermittently, and it needs to interface with other systems a different request format was used. Flatbuffers, though efficient, are not readable due to Flatbuffer being a binary format. Furthermore, Flatbuffers are not commonly used as a messaging format between different systems, since the format of the messages is very strict and needs to be known to both systems.

A very popular option is JSON. Requests and responses in JSON can contain more data than the other system needs. Furthermore, the structure of the data is less strict, the order of keys with the same nesting does not matter. Any extra data can be ignored by the other system. These features make JSON well suited for a data interchange language for interoperable systems. The Policy Distributor provides a web-server with a single endpoint, a POST endpoint. This endpoint accepts a JSON object which contains all shop settings including products, stocks, and admission policies. The deletion and retrieval endpoints were omitted since they were not required by any of the requirements, even though they are trivial to implement.

For every request, the Policy Distributor marshals the JSON object into RDF N-Quads for storage in Dgraph. The schema of the data stored in Dgraph can be found in Appendix B.1. Requests are only verified to contain a JSON object in the correct structure.

4.5.3 Log Aggregator

The Log Aggregator is performing two tasks: taking in log events and storing them in a database, and disclosing a basic statistic to the Admission Controller. As explained in Section 4.4, ClickHouse is used to store the number of product instances and customers in each of their respective lifecycles. This is achieved by logging the differences in amounts per lifecycle phase.

Taking a customer that is admitted as an example. The Log Aggregator will insert a single tuple into ClickHouse, containing a timestamp and that the amount of "waiting" customers has decreased by one, while the amount of browsing customers has increased by one. These changes are then aggregated automatically by ClickHouse.

ClickHouse has a limitation however, for performance reasons it requires bulk inserts. Strictly speaking, each event can be inserted separately, but this has a high performance penalty.

In order to deal with this all inserts are chunked and regularly inserted into ClickHouse. Since there is a possibility that between insertions the Log Aggregator experiences a fatal error, log messages could be lost. To mitigate this issue, all events are first stored in a local key-value store, Badger[30]. Badger is part of Dgraph and is used to actually store the graph data on each node in the Dgraph cluster. In the Log Aggregator, it is only used as a quick persistent datastore to store the log events until they are inserted into ClickHouse. After events have been inserted into ClickHouse they are removed from Badger.

On boot the Log Aggregator starts a periodic process. This process is responsible for generating and committing ClickHouse transactions. This process has a period of five seconds. The process attempts to commit any running transactions. When successful and Badger is cleared. Finally, new transactions are started. The process starts two transactions, one for inserting customer lifecycle log-items, and one for inserting product lifecycle log-items.

When the Log Aggregator receives a log-item it uses one of the running transactions to insert the data, as well as inserting it into Badger.

Crash recovery consist of inserting all events contained in the key-value store into ClickHouse. ClickHouse inserts are idempotent. Little care has to be taken to ensure duplicate events are not inserted.

Appendix B.2 contains the schema stored in ClickHouse. It consists of two tables to store all log events, and two views that aggregate the results which can be used by the Admission Controller and Grafana as a basic dashboard.

Provided statistic

One of the responsibilities of the Log Aggregator is to provide a basic statistic to the Admission Controller. This is not supposed to be an important responsibility of the Log Aggregator and the implementation reflects this. This statistic is implemented as a simple query which the Admission Controller executes itself directly. The Log Aggregator implementations does not provide a gRPC service to query this data.

Dashboard

Even though it was not strictly specified, a basic dashboard was created to visualize the number of customers in each lifecycle, and track the number of instances of products. This only serves to show the potential of the stored log entries and the derivable statistics, not to be a complete dashboard. A screenshot of this dashboard is shown in Figure 4.2.

4.5.4 Shop Keeper

The Shop Keeper needs to store the number of instances per phase of the lifecycle per product and use this information together with the deployed settings to determine whether stock can be handed out. For every lifecycle phase except "available" and every product, an integer is stored depicting the number of product instances that are currently in that phase, i.e. counters for the number of products in each phase.

For each of the product lifecycle transitions, as shown in Figure 3.3 and described in Section 3.1.2, some values need to be incremented or decremented on every request:

For the defined phase changes the following needs to happen:

- `MakeReservation`: increment the reserved products counter.
- `UndoReservation`: decrement the reserved products counter.
- `FinishBrowsing`: decrement the reserved products and increment the attempted products counters.
- `SucceedToPay`: decrement the attempted products and increment the sold products counters.
- `FailToPay`: decrement the attempted products counters.

As can be seen, the availability is neither incremented nor decremented. The availability is computed from the other fields and the policies setup. The policies determine whether the reserved, pending, and sold counts need to be taken into account when determining the availability. It is therefore possible for two products having exactly the same stock, and reserved, pending, and sold counts to still have a different availability.

These counts are stored in Dgraph. When these counts need to be updated, it is important all updates occur atomically. Otherwise products can oversell. Assuming a customer has finished shopping, all of his reserved products must transition to "pending". If at first, the reserved count was decremented before the pending count was incremented, the availability of that product (temporarily) increases by the difference. Therefore, the increases and decreases should occur at once, atomically.

A distinction is made between "MakeReservation" and all other changes. For "MakeReservation" it is important to verify whether enough product is available. For all other changes, this check is not needed since these requests never decrease the actual availability. It might be the case that they decrease the reported availability, since this is derived from the stock and a combination of the "reserved", "pending", and "sold" counters.

In order to implement all requirements pertaining to availability, three booleans are stored per product. A boolean for expressing whether the number of reserved, pending, or sold products should be taken into account when determining the availability of a product. When requesting a reservation, a single change request is issued containing a query that calculates the appropriate value for the availability. As specified in the architecture, products are reserved one at a time. Consequently, it is sufficient to check whether the availability is greater than zero.

When the availability is greater than zero, the reservation count is incremented. Otherwise this count is not incremented.

For the other phase changes, it is first determined which field needs to be incremented, and which field needs to be decremented. A query is then constructed which updates the relevant counters.

4.5.5 Reservation Controller

In the basic case, the Reservation Controller receives requests from the Customer Container, calls the Shop Keeper, generates a random identifier (the reservation), stores the results, and returns the result to the Customer Container. As soon as sub-products get involved, the situation gets quite a bit more complex. For this proof-of-concept an implementation of sub-products could not be completed due to implementation problems. Implementing a Reservation Controller that can handle sub-products and issue reservations using the architecture presented in Section 3.1.2 is left as future work.

Due to this omission, the implementation of the Reservation Controller is quite simple. For a new reservation request, the Reservation Controller requests the Shop Keeper whether stock is available. Iff so, a reservation is generated and linked to the requesting customer.

For any of the customer lifecycle changes, the Reservation Controller retrieves all issued reservations for that customer, derives the number of reserved instances per product, and requests the Shop Keeper to change that amount for each of the products.

4.5.6 Queue Manager

As stated previously, the Queue Manager provides two services: adding customers to queues, and admitting customers from the queues.

As an underlying datastructure the Queue Manager makes use of (singly) linked lists, where every element of the list points to its successor. The linked list was chosen since it was the easiest datastructure to serialize in Dgraph, where it is persisted. Privilege levels where not implemented, but could be implemented by altering the queries used for adding to and admission from queues.

Only arrival monotonous queue admission is implemented. This is enabled by the strong consistency provided by Dgraph. Because of this choice the weaker admission requirements (ma1,ma2,ma3) are implemented as well.

Both the adding and removing of customers to queues are implemented as a single transaction containing a single mutation. When combined with only a single Admission Controller handling a shop results in adding customers in arrival monotonic order.

In total, two references to elements in the linked list are stored. A reference to the head and a reference to the tail. These serve to aid with quick admission from and admission into the queues respectively. For admission into the queues, the tail element is needed, if this was not implemented, the entire list needs to be traversed in order to find the last element. Storing it directly is easier. Admission of a customer is simply adding a successor to the current tail of the queue and updating the tail reference to the newly added element.

Admission from a queue is exactly the opposite. The head of the queue is replaced by its successor and the customer contained in the old head is returned to the Admission Controller. Appendix C contains the query and mutations used for admitting a customer from a queue.

4.5.7 Admission Controller

Previously, it was explained that stores should be distributed over all Admission Controller nodes in such a way that every store has a single Admission Controller node responsible for admission. Due to the way the Admission Controller is deployed, each Admission Controller node has a sequential integer identifier. This identifier starts from zero. Using this index (i), the total number of Admission Controller pods (n), and the uid of stores (u) a simple method is used to distribute the stores over all Admission Controller nodes. A uid is a unique integer identifier assigned by Dgraph to every node in the graph. An Admission Controller node is responsible for a store iff $i \equiv u \ (mod\ n)$. Even though this procedure works reasonably well for distributing the stores initially (on boot). It does not take into account possible load caused by the shops. For a full-fledge implementation a better distribution scheme needs to be implemented.

In order to admit customers, the Admission Controller must be aware of the amount of customers per lifecycle phase per store. It simply tracks these numbers using integers. When it receives a message over the Publish-Subscribe middleware notifying a lifecycle change, it increments and decrements these counters where needed. For any lifecycle change, it decrements the counter for the previously held lifecycle phase, while incrementing the counter for the new lifecycle phase.

When an arrival message is received, it first checks whether the customer can be admitted immediately. Iff so, it immediately sends a message over the Publish-Subscribe middleware that the customer's new status is "shopping".

When the customer cannot be admitted immediately, it requests the Queue Manager to add the customer to a queue. Then it sends a message over the Publish-Subscribe middleware that the customer's new status is "arrived".

Regardless of what the Admission Controller will reply, the Admission Controller always register a customer has arrived. Only in the case the customer has been admitted immediately, it will also register a customer has been admitted. Consequently, regardless of the response, the counts are always accurate. The procedure is applied when customers finish shopping and want to be admitted to payment services, only using different lifecycle phases.

In order to achieve this, the Admission Controller needs the policies to contain the following four variables:

- Phase: which phase this policy is applied to.
- Max: the maximum amount of customers that can be in the phase at the same time.
- AdmitPerMinute: the amount of customers that can be admitted per minute.
- Admission: who to admit, since privilege levels are not implemented only "any" and "none" are used. When none is chosen, all customers are immediately added to the queues and admission is halted.

These policies are cached locally by the Admission Controller. Updates to these policies can occur at any point in time. Therefor all Admission Controller nodes start polling the database at boot. They check whether the set of shops distributed to them has changed, applies any changes in the admission policies, starts admission for new stores, and stops admission for stores that are no longer in the set.

To limit the rate of the number of customers that can start browsing, tickers are started which send an event (tick)
`AdmitPerMinute` times per minute. For every tick, an event is sent over a channel. A pool of workers listens to these events and when an event is received checks whether customers are awaiting admission. If so, the customer at the head of the queue is retrieved from the Queue Manager (which updates the queue by removing the head) and an admission message is sent for that customer over the Publish-Subscribe middleware. To ensure monotonic admission, separate Goroutines handle the admission into the shop and admission to payment services.

Since NATS streaming provides at-least-once delivery, the Admission Controller must be able and cope with duplicate messages. This is implemented by simply keeping track of which customer and lifecycle combinations have been seen. If a duplicate message is detected, it is ignored.

Since it is not guaranteed that a query in Dgraph will succeed within the allotted time.

The requirements for store admission are a superset of the requirements for payment services admission. Therefore, the implementation is agnostic to which phase in the lifecycle it is handling. Thus both of the lifecycle phases in need of admission are handled by the same implementation.

4.6 Kubernetes deployment specifics

One of the main concepts behind Kubernetes is that all infrastructure resources, collectively called resource objects, can be declaratively specified and then automatically deployed in a platform agnostic manner. These resource objects not only include deployed containers, but also (persistent) storage and specifics on network communication. The purpose of this project is to not only create a solution that solves the issues but which also can be run on a Kubernetes cluster.

Two main parts of deploying the entire system to a Kubernetes cluster will be of particular focus: container deployment and container communication.

4.6.1 Container deployment

In Kubernetes, the unit of deployment is called a "pod". A pod consists of multiple containers that are highly coupled and share resources. It is also possible to assign containers to perform startup tasks before the main containers start in order to prepare the local environment for the application containers. Kubernetes uses controllers to actual control the deployment of these pods, all of which are tailored to different workloads. The following is a summary of the Kubernetes documentation:

- **ReplicaSet:** maintains a stable set of pods at any given time.
- **ReplicationController:** provides similar functionality to the combination of a Deployment+ReplicaSet.
- **Deployment:** provides updates to Pods and ReplicaSets. Used to declaratively describe desired state and let the cluster determine the specifics.
- **StatefulSet:** same as a Deployment, except each pod gets a sticky identity, providing stable network identifiers and re-attachable storage, as well as ordered and graceful scaling. Re-attachable means, that when a pod called "POD-2" crashes, and a new pod with the same name is started, all disks attached to the old "POD-2" are now attached to the new pod.
- **DaemonSet:** ensures every node (machine) in the cluster runs an instance of the specified pod.
- **Jobs:** deploys a certain number of pods which perform a task that should terminate and ensures that a set amount of them terminate successfully.
- **Garbage Collection:** used to delete orphaned resources, for instance Pods, whose parents (a ReplicaSet for instance) already have been removed, but are still running themselves.
- **TTL Controller:** creates Jobs and automatically removes resources after some Time To Live (TTL) has passed.
- **CronJob:** creates Jobs on a repeating schedule.

From these controllers, two are used: the Deployment and the StatefulSet. The developer provided configuration for Dgraph utilizes a DaemonSet to ensure that all cluster nodes have a local instance available to them to ensure that all pods on that node have access to a local copy of the graph database.

The Admission Controller and Log Aggregator are deployed using StatefulSets, while all other components make use of Deployments. StatefulSets where chosen since these components are stateful.

Even though the Log Aggregator operates in a stateless manner, it still is implemented as a StatefulSet due to the fact that it temporarily stores the log-items in the key-value store. By utilizing a StatefulSet, the Badger database can be stored on a persistent disk. When pod crashes and a new pod is deployed the disk of the crashed pod is attached to the new pod. The Log Aggregator can then recover the data by simply retrieving the data in the database and inserting it into ClickHouse.

Another advantage of StatefulSets is that the deployed pods receive incrementing integers as identifiers. This information is mainly useful for the Admission Controller, which uses this identifier and the total number of deployed pods in the StatefulSet to determine which shops are in their partition.

```
apiVersion: v1
kind: Service
metadata:
name: cc-ext
spec:
type: NodePort
ports:
- name: "sock"
  port: 8000
  targetPort: 8000
  protocol: TCP
- name: "web"
  port: 8001
  targetPort: 8001
  protocol: TCP
selector:
  app: waas
  tier: cc
```

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
annotations:
  nginx.ingress.kubernetes.io/rewrite-target: /$1
  nginx.ingress.kubernetes.io/configuration-snippet: |
    proxy_set_header l5d-dst-override $service_name.$namespace.svc.cluster.local:$service_port;
    grpc_set_header l5d-dst-override $service_name.$namespace.svc.cluster.local:$service_port;
name: ingress
spec:
rules:
  - host: waas.openticket.tech
    http:
      paths:
        - path: /(pd/.*)
          backend:
            serviceName: pd-ext
            servicePort: 8003
        - path: /(sock/.*)
          backend:
            serviceName: cc-ext
            servicePort: 8000
        - path: /(cc/.*)
          backend:
            serviceName: cc-ext
            servicePort: 8001
```

Listing 4.1: Example of a Kubernetes service (left) and ingress (right) specification

All other components are deployed using Deployments, all using ReplicaSets.

An example of a Deployment and StatefulSet specification can be found in Appendix D.

4.6.2 Container communication

For container communication Kubernetes uses two different resource objects: “Service” and “Ingress”. A service is a named combination of a “pod selector” and a set of ports. Pods can be labelled with metadata, a “pod selector” then uses this metadata to match a set of pods. A Service then makes the pods and those ports available to other pods, using the provided name. An example service can be seen in Listing 4.1. This service defines that ports 8000 and 8001 of the pods matching the labels defined in the “selector” field (“app” equals “waas” and “tier” equals “cc”) are reachable from the common names, “cc-ext”.

Since Kubernetes uses DNS to resolve the pod to contact and gRPC only resolves this address once, load-balancing does not exist by default. If for every request the hostname is resolved, Kubernetes will ensure that the traffic gets load-balanced between all pods adhering to the service selector. Since it is preferred to have load-balancing, the proof-of-concept makes use of a so called service mesh[31]. A service mesh is an infrastructure layer that allows for control on how pods communicate. It can be used for structures like circuit breakers, where pods are made unavailable to other pods when the component is throwing too many errors. Another feature of some service meshes is support for canary releases. Many different service meshes exist. These service meshes were only given a cursory examination. When choosing a service mesh the ability of load-balancing gRPC requests was the main consideration. The service mesh that enabled this behavior with the least configuration was Linkerd[32], thus Linkerd was chosen and used as the service mesh.

A service by itself only creates a communication channel between pods within the cluster. It still is not possible to contact any of the pods from outside of the cluster. For this Kubernetes makes use of another resource object, called “ingress”. An ingress is a resource object describing how a specific service can be accessed from outside of the cluster. Listing 4.1 contains the ingress specification for all services requiring an ingress including the service of Listing 4.1.

An ingress is controlled by an Ingress controller, typically a web-server or an OSI layer 4/7 software load-balancer. These listen to incoming requests from outside of the cluster and forward them to the appropriate services. For the deployment of the proof-of-concept different ingress controllers were investigated. Specifically NGINX[33], a high performance webserver and reverse proxy, and Traefik[34], which is a OSI layer 7 load balancer, were tested and considered as the ingress controller. Even though Traefik provided more features over NGINX, the final choice was still NGINX. Since it provided a simpler solution, the project is more mature, and Eventix was already familiar with NGINX.

The ingress of Listing 4.1 allows for three routes at the `http://waas.openticket.tech` domain. The following services are made available:

1. `pd-ext:8003` is made available at the path: `/pd/`, used for storing settings at the Policy Distributor.

2. `cc-ext:8000` is made available at the path: `/cc/`, serving the simple interface provided by the Customer Container.
3. `cc-ext:8001` is made available at the path: `/sock/`, serving the websocket server for the Customer Container.

Chapter 5

Validation

In this chapter, the proof-of-concept implementation will be validated to work as expected. At first, requirements are listed that were not implemented in the proof-of-concept. Then the performance of the databases will be discussed, whether they performed as expected. Lastly, the operations of the proof-of-concept will be verified. This verification is done by executing two functional tests to verify whether requirements are properly implemented.

5.1 Unimplemented requirements

In this section unimplemented requirements will be discussed. These can be broadly grouped in four different "features". In Section 6.2 possible implementations for these "features" will be presented.

The statistics are not present in this list. They were specified in the requirements to indicate their relevance to a full-fledge system. But for any full-fledge system, these requirements should be expanded. Determining the statistics to generate is not an architectural problem. Although the architecture might need to be redesigned in order to be able to generate the new statistics. Since these expanded statistics requirements are currently unknown, no plans can be presented on how to implement them.

Sub-products

Relevant requirements: sp2, sp3, sp5, sp6, sp7, sp8, and sp9

As stated in the previous chapter, sub-products could not be implemented. For the proof-of-concept implementation modeling the products and constraints as a DAG was attempted, which resulted in a complex process of issuing reservations. Even when the iterative approach was used. The work on this approach could not be finished.

Multiple shops

Relevant requirements: mt3

In the proof-of-concept, it is possible for shops to experience degraded performance due to the distribution of shops not taking (the expected) load into account. This choice was made to ensure shopping runs could take place and the Admission Controller could start admitting customers.

Customer feedback

Relevant requirements: cs5, ce9, ce10, ce11, and ce12

Currently, no customer feedback is implemented. After arriving, customers receive no feedback on their current place in the queue and possible wait times. Furthermore, currently no limits are imposed on the amount of time customers can spend in any phase of their lifecycles. The lifecycles of customers who have departed the shop without notifying are currently not forced to progress to latex phases.

These requirements are not implemented due to setting different priorities.

Tie-breakers

Relevant requirements: tb2, tb3, tb4, and tb5

For this proof-of-concept only a single tiebreaker was considered. No method could be devised to implement any of the other tiebreakers without either sacrificing request duration and throughput of the Reservation Controller dramatically, or making the Reservation Controller stateful.

It was deemed more important to keep request durations as short as possible and to ensure as many components operate in a stateless manner, than to implement the other tiebreakers.

5.2 Database evaluation

Due to the choice of database being very important, it cannot be omitted from the evaluation of the proof-of-concept. Two databases were chosen, ClickHouse and Dgraph, each of which will be discussed in separate sections.

5.2.1 ClickHouse

The choice for ClickHouse is deemed a good choice. Analysis of the Log Aggregator logs shows that everything that was sent to the Log Aggregator was inserted into ClickHouse. No issues where encountered while using ClickHouse.

Clickhouse was not tested directly, but was tested through a load test of the Log Aggregator. A script was written to attempt sending as many log items as possible sequentially and record the response times. The test was executed locally on a development machine, but the ClickHouse server was the same as the one used by the Log Aggregator deployed to the Kubernetes cluster.

The connection to the ClickHouse database was quite bad. This was done on purpose to provide a pessimistic scenario. The networking delay was tested using the ping command for 100 pings at an interval of 0.1 seconds. It reported a minimum, maximum, average round trip times and the standard deviation of: 23.340, 69.932, 29.506, 7.180 milliseconds respectively. To make matters worse, the (locally running) Log Aggregator was connected to ClickHouse using an SSH port-forward, thus incurring extra overhead and increasing latency.

The load test attempted to send as many log-items to the Log Aggregator as possible for 60 seconds. The following minimum, maximum, average response times, and standard deviation were found: 133, 34,2234, 468, and 4,450 microseconds respectively. The total number of requests placed was 144,674. These numbers are deemed more than sufficient for the proof-of-concept.

The expectation was that when the transaction commits, a slowdown would be detected. Since the Log Aggregator commits every five seconds, between 11 and 13 periods with slow requests were expected.

Inspection of a plot of the request durations did not clearly show where the commits had taken place. Slow requests where not grouped together.

5.2.2 Dgraph

Even though Dgraph provided the needed consistency constraints, it did not perform as expected. Dgraph performs very well when reading data, i.e. when data access is read-heavy. Unfortunately, the interaction of the proof-of-concept with Dgraph is write-heavy. Only recovery of lost sessions, and retrieving the a customers's reservations (for product lifecycle changes) only reads data, all other interaction also mutate data.

When Dgraph detects multiple concurrent mutations of the same data, transactions are aborted automatically. To deal with this, two different retry schemes were tested: immediate retry and random exponential backoff. Immediate retry only exacerbated the problem of aborting transactions, since more mutations were taking place at the same time. The random exponential backoff resulted in much better performance. But did, of course, not fully eliminate the issue. Due to the limited throughput and the retrying of transactions, the performance of the system is not as expected.

Even with the exponential random backoff, throughput was 80 to 100 mutations per second at best. Which is not enough to test an actual flash-crowd scenario.

Measure	Raw measurements	Without outliers
Number of measurements	10,000	9,980
Minimum rtt	0.333 ms	0.333 ms
Maximum rtt	4.08 ms	0.767 ms
Average rtt	0.495 ms	0.492 ms
Standard deviation	0.113 ms	0.082 ms

Table 5.1: RTT results between the test host and the cluster load balancer

5.3 Verification

Two different scenarios will be tested. Each scenario will verify whether one of the more important responsibilities of the system works as required. The first scenario tests admission control, ensuring order is kept. The second scenario tests whether stocks are kept properly, ensuring stocks are not exceeded.

5.3.1 Testing setup

For the test, the proof-of-concept was deployed on a DigitalOcean managed Kubernetes cluster. This cluster consists of three nodes, each having 4 (virtual) CPUs and 8GB of memory.

The tests executed are functional tests. Proper load tests cannot be run, since even the functional tests achieve the maximum throughput of Dgraph.

The tests have been run on a single VPS, located in the same DigitalOcean data center as the Kubernetes cluster. DigitalOcean provides guarantees on resource reservation. When a VPS is created, CPU and memory resources are reserved for that VPS. This should result in consistent VPS performance. Testing using a DigitalOcean VPS was chosen in order to achieve a high reliability of the network. Fewer hops between the machine running the test and the Kubernetes cluster should result in smaller and less varying network overhead. This was measured by testing the latency of the network connection by using the `ping` command which measures the round-trip-time (rtt) between two network hosts.

The test involved measuring 10,000 rtts, at an interval 0.1 seconds using:
`ping -c 10000 -i 0.1 waas.openticket.tech`. From these raw results outliers where filtered by first calculating the quartiles and the Inter Quartile Range (IQR) between the first and third quartiles and then removing measurements with a difference to the mean greater than 1.5 times the IQR. The following values where found for the first and third quartiles respectively: $q_1 = 0.42$, $q_3 = 0.559$. The IQR derived from these quartiles is $IQR = q_3 - q_1 = 0.139$. This test was run three times. Appendix G contains all the results. The results presented are the worst of the three tests. “waas.openticket.tech” was registered in DNS to point to the cluster’s load balancer.

Unfortunately, a deeper investigation in the network behavior could not be performed. The DigitalOcean provided load balancers is not accessible and networking within the cluster is handled by a combination of Kubernetes services and the service mesh. This makes the networking characteristics behind the load balancer difficult to investigate.

The minimum, maximum, and average rtts, as well as the standard deviation are presented in Table 5.1 for both the raw and cleaned measurements.

These results show that the delays between two hosts are comparatively small, and the network is performing quite consistently.

5.4 Functional tests

In this section two functional tests and their results are presented. The two tests focus on two of the more important responsibilities of the system: admission control and stock keeping.

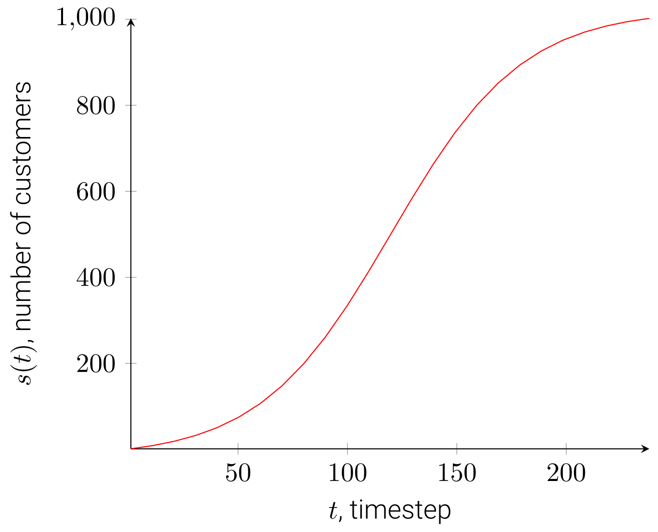


Figure 5.1: Plot of the sigmoid function used to determine the number of customers

5.4.1 Admission control

This test will determine whether the system can admit customers in the same order as they arrived in, arrival monotonically.

The scenario tested is described by the following steps:

1. A customer arrives.
2. After being admitted into the shop, the customer browses for three seconds. No reservations are placed.
3. The customer signals he/she has finished shopping.
4. After being admitted to payment services, the customer attempts payment for five seconds.
5. The customer signals he/she will depart.

Anecdotal evidence from Eventix states that the total number of customers over time during a flash-crowd event can be modeled using a sigmoidal function. The number of arriving customers is low at first, increases quickly, and decreases gradually after the peak in arriving customers has been reached.

For this test the following sigmoidal function was used:

$$s(t) = 520 * (tanh(\frac{t - 120}{60})) + 502$$

A plot of the sigmoidal function can be found in Figure 5.1.

A script was created that executes a scenario a number of times for a certain amount of timesteps. For this test the timestep chosen was one second. The amount of new scenarios started per timestep was given by $\lfloor s(t) - s(t-1) + 0.5 \rfloor$ for any step $t \in \mathbb{N}^+$, i.e. the nearest integer rounding of the difference between two timesteps.

The script was run for 232 timesteps, $1 \leqslant t \leqslant 232$, resulting in the scenario having been played 1000 times, with a maximum of nine started scenarios in a single second.

Four Queue Manager nodes where deployed. Only a single Admission Controller node was deployed since only a single store will be part of the tests. For all other relevant components enough capacity was ensured. Specifically four Customer Container and four Log Aggregator nodes were deployed.

Since the implementation of the admission into the shop and admission to payment services is the same, only store admission will be verified. For the test, admission to payment services is setup to be immediate.

The policies setup for the test allow for an absolute maximum of 20 shopping customers and admits customers at a rate of 300 customers per minute. No customer is admitted immediately. Every customer is only admitted at a schedule of 300 per minute and until the maximum of 20 has been reached.

Results

There are two places where arrival can be determined: at the Customer Container or at the Admission Controller. Different Customer Container nodes can be experiencing different load. Using the Customer Container to determine the arrival order of customers is therefore not very reliable. Furthermore, only log items can be used to reliably determine when a customer was admitted. Since multiple Customer Container nodes are generating these log items, determining the precise order in which they were generated is not feasible with the current implementation.

Therefor the Admission Controller was chosen to provide the information on when customers arrived and when they were admitted. The Admission Controller requests the Queue Manager to add customers to queues and request the Queue Manager to admit customers from queues. This makes the Admission Controller the ideal candidate for determining whether the order is being kept.

When comparing the arrivals and departures of customers, only one discrepancy could be found. All other customers were in the correct order.

This one discrepancy was a scenario that was aborted due to Dgraph not being able to add this simulated customer to a queue. Earlier attempts at running this test found all 1000 customers were added to queues and admitted from queues successfully. These results prove that the proof-of-concept implementation can admit customers in arrival monotonic order.

Analysis of the log files, and the entries stored by the Log Aggregator show however that the Admission Controller node was not keeping up. Many scenario instances were waiting minutes before receiving the confirmation that they were added to a queue. This also showed an error in the Log Aggregator implementation, since at some point the Log Aggregator reported more than 20 browsing customers.

This can be explained by the fact that the Log Aggregator rounds the timestamp of the logged items to the nearest second. The Customer Container notifies the Log Aggregator of lifecycle changes. Since different customers are connected to different Customer Container nodes, it is possible the “finish” event was logged after the “browsing” event, resulting in an over reporting of the number of browsing customers.

5.5 Exceed stocks

This test will verify whether stocks are not exceeded when placing reservations and whether products follow the correct lifecycle. To do this the following scenario will be tested:

- A customer arrives.
- On admission, the customer attempts to reserve two instances of the same product.
- After the second successful attempt, the last reservation is released.
- The customer signals he/she has finished browsing.
- The customer signals he/she has departed the store.

Departing a store is counted as having paid for the reserved products.

As stated before, Dgraph does not support multiple transactions mutating similar data. Multiple instances of this scenario will not be run in parallel. In almost all preliminary tests, running multiple concurrent scenarios resulted in too many aborted transactions to make the results usable. The scenario script could have been made more robust, in order to deal with these abortions. Dealing with these abortions would mean even more requests and transactions would be created, only exacerbating the problem of aborting transactions.

Instead of running the scenario in parallel, 50 of them were run sequentially. When a scenario has finished, the next scenario starts. All scenarios are trying to reserve the same product with a stock of 80, ensuring some reservation attempts must fail. Furthermore, for the test it is required that all lifecycle counters are taken into account when determining whether a product is available.

The values reported to the Log Aggregator are used to determine whether stock is exceeded. At no point in time can the number of reserved+pending+sold products be greater than 80.

As with the other scenario, it must be ensured enough capacity exists in other components to not be of influence on the scenarios. For the two components involved with this test, the Reservation Controller and the Shop Keeper, four nodes where deployed. The load on the Log Aggregator and the Customer Container also impact this test.

Four Customer Container and four Log Aggregator nodes were deployed to ensure enough capacity existed to not be a bottleneck in the test.

The Admission Controller was setup in such a way that immediate admission was allowed. No queueing was needed in this scenario.

Results

It can be verified in two ways whether stocks are not exceeded. The first, is by counting how many reservations were received and how many were handed back by the test script. Calculating the running sum shows that the total number of issued stock never exceeds 80.

The second way, is by checking the logs aggregated by the Log Aggregator. By, again, calculating a running sum, the number of products in each lifecycle phase at each point in time can be determined. This also showed that the stocks were never exceeded.

Chapter 6

Conclusion

In this project, an attempt was made to solve the problems of handling flash-crowds and issuing reservations for products having complex stocks. In this chapter, the architecture and the proof-of-concept implementation will be evaluated. Another look will be given to the requirements. Analyzing them to see whether relaxing certain requirements could improve the quality and usability of the project.

At first, the performance found in the tests described in the previous chapter is discussed. Then possible future work is laid out. Specifically, four unimplemented features are discussed and possible solutions are presented. Finally, recommendations on possible next steps for continuing this project are given.

6.1 Performance discussion

The results of the functional tests described in Chapter 5 show that the proof-of-concept implementation is functional. However, it is also clear that the throughput is quite low. A maximum of nine customers arriving in a second cannot be called a flash crowd. Neither is only being able to reliably handle a single concurrently shopping customer.

Therefore, it cannot definitively be concluded that the designed architecture can properly deal with flash crowds. Many attempts of improving throughput were attempted, but without the desired results. In the attempts it became obvious, and as stated in Chapter 5, that the throughput of Dgraph is the main bottleneck. Choosing a different database which provides both the needed consistency guarantees and a higher throughput should result in throughput improvements.

Another method of improving throughput is by changing the data access. Listed below are ideas on how throughput can be increased in the order and stock modules.

Order module

The order module needs to ensure that order is kept and customers are added to queues when appropriate.

By weakening the requirements for direct admission (ce1, cp1), the responsibilities of the Admission Controller can be reduced to only admitting customers on a fixed schedule. This removes much of the load of the Admission Controller and thus increasing the maximum number of customers that can be admitted per minute by a single Admission Controller.

Another possible optimization is to split the complete queue in smaller sub-queues and joining the sub-queues periodically. This would decrease the number of mutations that are updating similar data, increasing the total throughput.

Stock module

As stated in Chapter 3, for keeping stock very strong consistency is needed. This will also need to hold when dealing with stocks that cannot oversell.

By making the Shop Keeper stateful and dividing the stock between all Shop Keeper nodes, throughput might be increased. Assume a stock of n, with c deployed Shop Keeper nodes. Each Shop Keeper node can then be responsible for $\frac{n}{c}$ of the total stock. Each Shop Keeper should then ensure that locally the stock is not exceeded.

The Reservation Controller can then iterate over some set of Shop Keeper to see whether stock exists for reservations before issuing a reservation. Dynamic policies can be created that merge multiple Shop Keeper nodes when their stock drops below a certain threshold.

Since this (as with the optimizations for the order module) decreases the amount of possible transaction conflicts, total throughput should rise.

6.2 Future work

In this section, possible solution to three of the four missing features described in Section 5.1 are presented.

Sub-products

For implementing sub-products, two approaches might be possible. The simplest approach is to restrict nesting of sub-products and simply counting the number of attached products per level for every reservation. Determining whether to issue a reservation with respect to the constraints only requires checking these counts.

Another more complex approach, is to use a generic structure of storing the products and the constraints. For instance, modeling the products, sub-products, and the constraints between them as a tree or a DAG. Using a DAG is not advisable even though it can express very complex constraints, when nodes in the graph are either products or constraints. For the proof-of-concept, an implementation using a DAG was attempted, but this proved to have too many edge cases to implement properly in the time allotted.

Multiple shops

Two possible solutions are presented to ensure shops operate without degraded performance when one shop is experiencing a flash-crowd. The first solution is to create a better distribution scheme. Preferably a distribution scheme that also works well on runtime.

The second solution would be to use more intelligent load balancing. Currently, all customers are handled by the same set of Customer Container nodes. If shops had a bias towards a certain subset of the Customer Container nodes performance of other shops should not degrade. This could be applied to all components. This solution also requires improvements on the distribution of shops to ensure load is balanced better.

Customer feedback

A possible solution to providing an approximate “place in queue” and “time to admission” to customers, is to let the Customer Container derive the required values. This can be implemented, by only storing one extra integer with all arrived customers. This integer would contain the number of customers that have arrived before the currently arriving customer. Resulting in each customer for a shop receiving an incrementing integer value.

On admission, the Admission Controller must then add this integer. Since all Customer Container nodes have a subscription to the stores of all customers it has connections with. It can simply calculate the spot in the queue for each of its customers. From this spot, it can then derive the approximate wait time.

The following is a possible implementation for forcing customers to the next lifecycle phase. After arrival, the Customer Container can retrieve the maximum durations of each of the lifecycle phases and derive deadlines from these durations. Using these timers, the Customer Container can always force lifecycle phase changes for customers exceeding those deadlines.

6.3 Recommendations

Many lessons were learned from designing the architecture subject to the requirements and the subtle interactions between these requirements. From the lessons learned, improvements can be distilled that would benefit the project.

- Evaluate the requirements. Try to weaken them where possible. Of specific focus should be requirements dealing with admission, sub-products, and tiebreakers. These heavily influence the architecture. Restating these requirements and altering the architecture accordingly can result in improvements in the operations of the system.

- Research a different method of storing data. Many of the issues experienced while implementing were caused by the way data was stored. A mixed stateless/stateful approach as listed in Section 6.2. Using Dgraph is not specifically advised against, but looking into other databases might result in easy improvements in terms of reliability and predictability of the system.
- Research how to improve the Kubernetes deployment by researching available tooling. Many tools and techniques exist to facilitate running workloads on Kubernetes. Each have different advantages and disadvantages. A set of tools was chosen that provided satisfactory functionality and behavior, but did not attempt to find the "best" tools and techniques. Researching different tools and techniques can result in a better deployment, making partitioning of components easier.

6.4 Conclusion

In this thesis, an architecture designed for handling flash crowds and issuing reservations is presented, each subject to many different policies. An implementation of this architecture solves major issues for companies selling popular products with strictly limited, unreplenishable stock.

A proof-of-concept of this architecture was created and verified whether it could solve the issues of properly handling flash crowds and issuing reservations. Even though the performance was deemed insufficient, it was shown to have implemented the desired behaviors.

The next steps should be to evaluate the outcome of this project, whether the ideas behind the architecture and the implementation are deemed correct and suitable, evaluating the requirements, altering requirements where needed, and perform more research on Kubernetes tooling.

Bibliography

[1] M. Mannion and B. Keepence, “Smart requirements,” *SIGSOFT Softw. Eng. Notes*, vol. 20, no. 2, p. 42–47, Apr. 1995. [Online]. Available: https://doi.org/10.1145/224155.224157 1.2.1

[2] D. Gifford and A. Spector, “The twa reservation system,” *Communications of the ACM*, vol. 27, no. 7, pp. 650–665, 1984. 1.3.1

[3] J. Eklund, “The reservisor automated airline reservation system: Combining communications and computing,” *IEEE Annals of the History of Computing*, vol. 16, no. 1, pp. 62–69, 1994. 1.3.1

[4] M. Oloyede, S. Alaya, and K. Adewole, “Development of an online bus ticket reservation system for a transportation service in nigeria,” *Development*, vol. 5, no. 12, 2014. 1.3.1

[5] D. Bermbach and J. Kuhlenkamp, “Consistency in distributed storage systems,” in *Networked Systems*, V. Gramoli and R. Guerraoui, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 175–189. 1.3.1, 3.2.1

[6] J. Elson and J. Howell, “Handling flash crowds from your garage,” 01 2008. 1.3.2, 2.1.2

[7] J. R. Hamilton *et al.*, “On designing and deploying internet-scale services.” in *LISA*, vol. 18, 2007, pp. 1–18. 1.3.2

[8] X. Chen and J. Heidemann, “Flash crowd mitigation via adaptive admission control based on application-level observations,” *ACM Trans. Internet Technol.*, vol. 5, no. 3, p. 532–569, Aug. 2005. [Online]. Available: https://doi.org/10.1145/1084772.1084776 1.3.2

[9] Z. Zeng and B. Veeravalli, “Rate-based and queue-based dynamic load balancing algorithms in distributed systems,” in *Proceedings. Tenth International Conference on Parallel and Distributed Systems, 2004. ICPADS 2004.* IEEE, 2004, pp. 349–356. 1.3.2

[10] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin, “Distributed fifo allocation of identical resources using small shared space,” *ACM Trans. Program. Lang. Syst.*, vol. 11, no. 1, p. 90–114, Jan. 1989. [Online]. Available: https://doi.org/10.1145/59287.59292 1.3.2

[11] D. Clegg and R. Barker, *Case Method Fast-Track: A Rad Approach*. USA: Addison-Wesley Longman Publishing Co., Inc., 1994. 2

[12] P. Achimugu, A. Selamat, R. Ibrahim, and M. N. Mahrin, “A systematic literature review of software requirements prioritization research,” *Information and Software Technology*, vol. 56, no. 6, pp. 568 – 585, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584914000354 2

[13] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. 3, 3.1

[14] M. van Steen and A. Tanenbaum, *Distributed Systems*, 3rd ed., 2 2017, self-published, open publication. 3, 3.3.1, 3.3.2

[15] “What is kubernetes?” Jun 2020, accessed: 2020-06-29. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ 3

[16] A. Wiggins, *The Twelve-Factor App*, accessed: 2020-06-01. [Online]. Available: https://12factor.net/ 3

[17] P. b. R. Daigle, “Building twelve factor apps on heroku,” Aug 2016, accessed: 2020-06-09. [Online]. Available: https://blog.heroku.com/twelve-factor-apps 3

[18] M. Fowler, “The basics of web application security,” accessed: 2020-06-29. [Online]. Available: https://martinfowler.com/articles/web-security-basics.html 3.1.1

[19] A. Fox and E. A. Brewer, "Harvest, yield, and scalable tolerant systems," in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*. IEEE, 1999, pp. 174–178. 3.2.1

[20] "Container runtimes," Jun 2020, accessed: 2020-07-09. [Online]. Available: https://kubernetes.io/docs/setup/production-environment/container-runtimes/ 4.2.1

[21] "empowering app development for developers," accessed: 2020-07-09. [Online]. Available: https://www.docker.com/ 4.2.1

[22] "Overview," accessed: 2020-07-10. [Online]. Available: https://google.github.io/flatbuffers/ 4.3

[23] "Introduction," accessed: 2020-07-10. [Online]. Available: https://docs.nats.io/nats-streaming-concepts/intro 4.3.1

[24] "grpc - a high-performance, open source universal rpc framework," accessed: 2020-07-14. [Online]. Available: https://grpc.io/ 4.3.2

[25] "Rpc 2.0 specification," accessed: 2020-07-14. [Online]. Available: https://www.jsonrpc.org/specification 4.3.2

[26] H. de Saxcé, I. Oprescu, and Y. Chen, "Is http/2 really faster than http/1.1?" in *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2015, pp. 293–299. 4.3.2

[27] "Clickhouse dbms," accessed: 2020-07-10. [Online]. Available: https://clickhouse.tech/ 4.4

[28] D. Labs, "A distributed, fast graphql database with graph backend," accessed: 2020-07-10. [Online]. Available: https://dgraph.io/ 4.4

[29] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro 4.4

[30] D. Labs, "Badger db," accessed: 2020-07-21. [Online]. Available: https://dgraph.io/badger 4.5.3

[31] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019, pp. 122–1225. 4.6.2

[32] "Homepage," accessed: 2020-07-21. [Online]. Available: https://linkerd.io/ 4.6.2

[33] "High performance load balancer, web server, & reverse proxy," Jun 2020, accessed: 2020-07-12. [Online]. Available: https://www.nginx.com/ 4.6.2

[34] C. Team, "Traefik, the cloud native edge router: Containous," accessed: 2020-07-12. [Online]. Available: https://containo.us/traefik/ 4.6.2

Appendix A

Dockerfile

This appendix contains the Dockerfile used to build the images for the seven components. A Dockerfile contains a number of steps that need to be taken in order to generate a Docker image. The steps in this Dockerfile are split into two groups: building a binary, and configuring an image.

Docker images tend to be quite large since they often contain many tools. Images need to be small for ease and speed of deployment. To achieve this, a common image called “golang:1.14” is used to build the binary and then generate an image based on a much smaller base image. The “golang:1.14” is quite large at 809MB. The Go compiler generates a statically linked binary. Therefore, the binary can be added to on an “empty” image. Resulting in an image containing just the binary. Instead of using an “empty” base image, “alpine” was chosen as a base image. It is possible to execute programs in a running container. Of course, only programs present in the image the container is based on can be used. Being able to execute other programs in containers, like a basic shell, is a good debug tool to have. Since the images are based on “alpine” instead of “scratch” (the “empty” image) all tools offered by the alpine Linux distribution can be used.

The first steps in the Dockerfile describe how to build the binary. By default the Go compiler adds debug headers to the binary. Since these headers are not needed, they are stripped away. To build the actual image, a new base is selected, “alpine”, and the binary is copied over to it. Finally, some environment variables are set, to facilitate deployment without much configuration.

The images resulting from these steps where between 16MB and 21MB large.

```
FROM golang:1.14.2 AS build

# Copy the current folder contents
WORKDIR $GOPATH/src/opentickets.tech/$BUILD_NAME
ADD . .

USER root

# Run the installer
RUN GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build -mod vendor -ldflags="-w␣-s" \
    -o /go/bin/docker_app


# Build a new image, copy everything over and set the entrypoint, use alpine
FROM alpine
COPY --from=build /go/bin/docker_app /app
EXPOSE 5000/tcp

# Spec required environment variables
ENV LOG_LEVEL=info \
    GRAYLOG_LOCATION='' \
    GRAYLOG_LOG_LEVEL=info \
    LOG_DISCARD='' \
    AC_TICK_DURATION=50 \
    AC_ADDR=:1236 \
    CC_ADDR=:1235 \
    PD_ADDR=:8003 \
    QM_ADDR=:1234 \
    RC_ADDR=:1238 \
    SK_ADDR=:1237 \
    SA_ADDR=:1239 \
    LISTEN=:8001 \
    STORE_NAME=arstarst

STOPSIGNAL 15

# Set the actual execution
CMD ["/app"]
```

Appendix B

Database specifications

This appendix contains the schemas deployed to the Dgraph and ClickHouse databases. Note, originally shops were called stores. Since this caused confusion with the word datastore, store was replaced by shop. In most parts of the implementation, including the schemas, store is still used instead of shop.

B.1 Dgraph schema

The terminology here will be ambiguous. Dgraph schemas have a notion called "type" and these "types" contain properties that also have a type. When the word type is used in the context of Dgraph "types", they will always be quoted. When type is not quoted, it is related to the type of a property.

Dgraph supports a small set of primitives for properties: string, float, int, geo, dateTime, and bool. All except geo are self explanatory. Properties of type geo depict coordinates and allow for spatial queries to be run on the database.

The only thing needed to get up-and-running with a Dgraph schema, is simply listing a set of properties with a type. Since Dgraph uses a declarative query language the entire structure of the response must be explicitly described in the query. Since this can be quite involved, Dgraph has introduced the notion of "types".

"Types" are simply groupings of properties. When types are properly used, queries can be simplified by adding the keywords `expand(_all_)`. When this keyword is added to a query, all properties of that type are automatically added to the response. It is similar to `SELECT * FROM ...` in SQL.

When a property depicts a relation like the "admission_policy" property, for instance, it does not point to a type, instead it points to a "uid". A uid is an identifier generated by Dgraph and serves as an identifier to uniquely identify a node. Within a "type", the type of a property depicting a relationship can use a "type" as the property type. Within "Store", the type of "admission_policy" is "Policy". Using this information, Dgraph can determine that when a query retrieving is retrieving a "Store" and its "admission_policy" property, the "type" of "admission_policy" is "Policy".

To model arrays, Dgraph requires the type of the property is infixed between brackets. An example of this can be seen in the "reservations" property, which is an array of the type uid.

The schema listed contains the structure originally intended to be used for sub-product handling. Since this implementation was never fully finished it is not fully described in the report. By renaming the "type" called "Node" to "Product" and removing the "children", "min", "max", and "unique" properties, the Schema is a drop in replacement for the current schema and implementation while improving on readability of the schema. This change was not applied in order to reduce the time spent on implementation. All extra properties are ignored by the proof-of-concept implementation.

```
# Define Directives and index
children: [uid] .
reservations: [uid] .
customers: [uid] .
xid: string @index(exact) .

max_size: int .
flow: int .
```

```
admission: int .
admit_per_minute: int .
open_at: DateTime .
admit_at: DateTime .
categories: [int] .
privilege_determination: int .
max_shopping: int .
max_returns: int .

customer: uid .
succ: uid .
head: uid .
tail: uid .

type Link {
    succ: Link
    customer: Customer
}

phase: int .

type Policy {
  xid
  max_size: int
  phase: int
  flow: int
  admission: int
  admit_per_minute: int
  open_at: DateTime
  admit_at: DateTime
  categories: [int]
  privilege_determination: int
  max_shopping: int
  max_returns: int

  head: Link
  tail: Link
}

placement_update_every: int .

opens: DateTime .
closes: DateTime .
min: int .
max: int .
unique: int .
admission_policy: uid .
payment_policy: uid .

type Store {
  xid
  placement_update_every: int
  opens: DateTime
  closes: DateTime

  min: int
  max: int
  unique: int

  admission_policy: Policy
  payment_policy: Policy

  children: [Node]

  customers: [Customer]
}
```

```
privilege: int .
status: int .
store: string .

type Customer {
  xid
  privilege: int
  status: int
  store: string
  reservations: [Reservation]
}

requires_reservation: bool .

stock: int .
reserved: int .
pending: int .
sold: int .

includes_pending: bool .
includes_reserved: bool .
includes_sold: bool .

type Node {
  xid
  children: [Node]
  requires_reservation: bool

  includes_pending: bool
  includes_reserved: bool
  includes_sold: bool

  min: int
  max: int
  unique: int

  stock: int
  reserved: int
  pending: int
  sold: int
}

node: uid .

valid: bool .
modifiable: bool .

type Reservation {
  xid
  node: Node
}
```

B.2 Clickhouse schema

Listed below is the schema used by ClickHouse. Two tables and two “realized views” are used. A table is similar to a table in SQL. A “realized view”, though similar, is different to SQL views, they are somewhat similar to materialized views. Realized views are similar to materialized views in that both are tables that contain the data without having to query the table they are based on. Contrary to materialized views, materialized views are updated automatically by ClickHouse.

The tables store the log items aggregated by the Log Aggregator. Specifically they contain each change in the product and customer lifecycles. For instance, when a customer changes lifecycle phase from “Waiting” to

"Browsing". A tuple is inserted into the "customer_events" table with a -1 for the "waiting" column and a +1 for the "browsing column". This tuple is then aggregated by the "customer_stats" realized view. This view aggregates the sum for each of the columns named after lifecycle phases of the "customer_events" table. Exactly the same approach is used for products, through the "product_events" and "product_stats" table and realized view.

Customers generate events relevant to the status of a shop. However, the events generated by a specific customer are not relevant for a shop, the combination of events generated by all customers of that shop is. Therefore, only the customer identifier is used to group the "customer_stats" by. The "customer" field could have been eliminated from the "customer_events" table, but it proved to be a valuable debugging tool, so it was kept. Furthermore, it can be used to resolve conflicts between customers on who got admitted when.

For products a similar reasoning applies. However, here the aid of having the "customer" field in the "customer_events" table was negligible, so it was omitted. Although it can be easily added in order to resolve conflicts between customers regarding who got reservations.

```sql
CREATE TABLE IF NOT EXISTS customer_events
(
    customer    FixedString(36),
    store       FixedString(36),
    waiting     Int,
    browsing    Int,
    done        Int,
    paying      Int,
    departed    Int,
    action_time DateTime64(6, 'UTC')
)
ENGINE = SummingMergeTree() ORDER BY (customer, store, action_time);

CREATE MATERIALIZED VIEW IF NOT EXISTS customer_stats
            ENGINE = AggregatingMergeTree() PARTITION BY toYYYYMM(action_time) ORDER BY (action_time)
AS
SELECT toStartOfSecond(action_time) as action_time,
       store                  as store,
       sumState(waiting)  AS waiting,
       sumState(browsing) AS browsing,
       sumState(done)     AS done,
       sumState(paying)   AS paying,
       sumState(departed) AS departed
FROM customer_events
GROUP BY action_time, store;

CREATE TABLE IF NOT EXISTS stock_events
(
    product     FixedString(36),

    reserved    Int,
    pending     Int,
    sold        Int,
    action_time DateTime64(6, 'UTC')
)
ENGINE = SummingMergeTree() ORDER BY (action_time, product);

CREATE MATERIALIZED VIEW IF NOT EXISTS stock_stats
            ENGINE = AggregatingMergeTree() PARTITION BY toYYYYMM(action_time) ORDER BY (action_time)
AS
SELECT toStartOfSecond(action_time) as action_time,
       product                  product,
       sumState(reserved)  AS reserved,
       sumState(pending)   AS pending,
       sumState(sold)      AS sold
FROM stock_events
GROUP BY action_time, product;
```

Appendix C

Dgraph change request example

Depicted in this appendix is a Dgraph change request. As explained in Chapter 4, Dgraph change requests consist of queries and mutations. Mutations contain descriptions on the data to mutate, and the data to delete and can make use of the result of the queries. Furthermore, each mutations can have an associated condition. A mutation with an associated condition is only applied iff the condition holds.

The example change request is the admission of a customer from a queue. Queues are implemented as linked lists since these allow fast appends and where easily modelled in Dgraph. Two operations are needed for the linked list. Appending a customer to the end of the list, and removing the head of the queue and replacing it with its successor. This change request deals with the second operation.

For this operation, the customer currently at the head of the queue needs to be retrieved, and the head of the queue must be replaced by the successor of the current head. If the current head does not have a successor, it must follow that the head is also the tail of the list. Since the tail of the list is stored with the head of the list (for admission into the queue) it must also be removed when the head is removed.

Therefore, the change request consists of one query, and two mutations. The query attempts to retrieve the head, the customer contained in, the head's successor, and the tail. The first mutation will replace the head of the queue with the head's successor iff there is a successor. While the second mutation removes both the head and the tail iff the head has no successor. The current head of the queue is returned, since the result of queries before the mutations were executed is returned as part of the response.

As explained, each store has two queues, each subject to their own policies. The two queues for a shop are stored as part of the policy they are subject to. Each policy has been given an identifier to be able and easily find it. This identifier is the shop identifier suffixed by the word "admission", or "payment" for the policies controlling shop admission and admission to payment services respectively.

Properties retrieved by queries can be used in mutations when they are marked as variables. Variables are depicted by an identifier, followed by the keyword "as", followed by the property.

The first mutation handles removing the head, this is simply done by setting the new head to the successor of the current head. The policy (containing the head property) is depicted by `uid(Obj)`, the property to mutate by `<head>`, and the new head by `uid(NHead)`. The second mutation deletes both the head and the tail, since neither is needed anymore.

Conditions on mutations were found to be somewhat limited. They work best when checking the cardinality of the result. Since every element in the linked list has at most one successor, it suffices to check whether the cardinality of the amount of retrieved successors (the `NHead` variable) is equal to either zero or one.

C.1 Query

```
query id ($id: string) {
  h(func: eq(xid, $id)) {
    Obj as uid
    head {
      customer {
        xid
      }
```

```
        succ {
          NHead as uid
        }
      }
    }
  }
```

C.2 First mutation

Condition	`@if(eq(len(NHead),1))`
Set	`uid(Obj) <head> uid(NHead) .`
Delete	-

C.3 Second mutation

Condition	`@if(eq(len(NHead),0))`
Set	-

Delete	`uid(Obj) <tail> * .`
	`uid(Obj) <head> * .`

Appendix D

Deployment and StatefulSet example

This appendix contains the YAML files needed to deploy the Customer Container, and the Log Aggregator. These were chosen since they depict the two different manners used to deploy to Kubernetes, a Deployment and a StatefulSet for the Customer Container and Log Aggregator respectively.

Though these two deployment primitives serve completely different purposes, their definition is similar. Both contain metadata, which is useful to identify a (set of) pod(s). Both contain a specification of the containers they will run, contained in the “spec” field, and this structure is even similar for both the Deployment and the StatefulSet.

A specification consists of a name, the name of the image for which a container must be started, network ports that can be exposed, and environment variables with their values for configuration. The difference (in this case) is in the specification of a “PersistentVolumeClaim” (PVC) using the `volumeClaimTemplates` field of the StatefulSet. A PVC is a persistent disk, potentially external to the Kubernetes cluster, that is attached to a container at some path in its filesystem. The `volumeClaimTemplates` field describes a template to a PVC, to be attached to all deployed containers of the Log Aggregator.

D.1 Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cc
  labels:
    app: waas
    tier: cc
spec:
  replicas: 1
  selector:
    matchLabels:
      app: waas
      tier: cc
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: waas
        tier: cc
    spec:
      containers:
      - image: registry.digitalocean.com/waas/cc
        imagePullPolicy: Always
        name: cc
        env:
        - name: CLICKHOUSE_ADDR
          value: 10.110.16.6:9000
        - name: SOCKET_PREFIX
```

```
          value: sock
        - name: REST_PREFIX
          value: cc
        - name: QM_ADDR
          value: qm:1234
        - name: sk_ADDR
          value: sk:1237
        - name: STAN_ADDR
          value: stan:7222
        - name: NATS_LOC
          value: nats.nats
        - name: DGRAPH_ADDR
          value: dgraph-dgraph-alpha.dgraph:9080
        - name: RC_ADDR
          value: rc:1238
        - name: SA_ADDR
          value: sa:1239
        ports:
        - containerPort: 8000
        - containerPort: 8001
      restartPolicy: Always
```

D.2 StatefulSet

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sa
  labels:
    app: waas
    tier: sa
spec:
  replicas: 3
  selector:
    matchLabels:
      app: waas
      tier: sa
  serviceName: sa
  template:
    metadata:
      labels:
        app: waas
        tier: sa
    spec:
      containers:
        - image: registry.digitalocean.com/waas/sa
          imagePullPolicy: Always
          name: sa
          env:
            - name: CLICKHOUSE_ADDR
              value: 10.110.16.6:9000
            - name: LOG_LEVEL
              valueFrom:
                configMapKeyRef:
                  name: configuration
                  key: log_level
          volumeMounts:
            - name: badger
              mountPath: /db
      restartPolicy: Always
  volumeClaimTemplates:
    - metadata:
        name: badger
      spec:
        accessModes: [ "ReadWriteOnce" ]
        resources:
          requests:
            storage: 1Gi
```

Appendix E

List of requirements

This appendix contains a listing of all requirements. Each requirement is accompanied by a short message describing whether the requirement has been implemented.

`cs1` any customer can immediately choose a product [MUST]
Implemented!

`cs2` limit reservation burstiness, any customer can choose some limited amount of product per interval [COULD]
Not implemented

`cs3` any customer can choose up to a strictly limited amount of product [MUST]
Not implemented

`cs4` any customer can choose up to a strictly limited amount of a single product [MUST]
Not implemented

`cs5` any customer can choose products within a certain fixed shopping window [MUST]
Not implemented

`cs6` any customer can only choose product that was not shown as sold-out to any customer [MUST]
Not implemented

`op1` any customer can get up to the amount of product that has not been purchased yet [MUST]
Implemented!

`op2` any customer can get up to the amount of product that has neither been purchased nor is awaiting payment success [MUST]
Implemented!

`op3` any customer can get up to the amount of product that has neither been purchased, is awaiting payment success, nor has been placed in baskets yet [SHOULD]
Implemented!

`op4` any customer can get up to some amount of product, allowing for some overselling percentage [COULD]
Strictly speaking, this requirement has been implemented. But it needs to be setup manually, by increasing available stock.

`sp1` A product can have either a strictly limited, or unlimited stock [MUST]
Implemented!

`sp2` A product contains multiple sub-products [MUST]
Not implemented

`sp3` A product contains required sub-products [MUST]
Not implemented

`sp4` A product contains optional sub-products [MUST]
Not implemented

`sp5` A product contains a set of sub-products from which a minimal amount of products need to be chosen. i.e. from 10 sub-products, at least two must be chosen [COULD]
Not implemented

`sp6` A product contains a set of sub-products from which a maximum amount of products can be chosen. i.e. from 10 sub-products, at most five can chosen [COULD]
Not implemented

`sp7` A product contains sub-products which can be purchased a limited amount of times, i.e. when composing a product, certain sub-products can be added a bounded number of times [COULD]
Not implemented

`sp8` For some $n, m \in \mathbb{N}, n > m$; When n customers want to reserve the same (sub-) product p, with $a_p = m$, a tiebreaker exists to determine who will get the (sub-) products [COULD]
Implemented!

`sp9` For some $n, m \in \mathbb{N}, n > m$; When n customers want to reserve the same product p having required sub-products, with $a_p = m$, a tiebreaker exists to determine who will get the products and the sub-products [COULD]
Not implemented

`ce1` customers are admitted immediately [MUST]
Implemented!

`ce2` customers are admitted never [MUST]
Implemented!

`ce3` customers are admitted iff privileged [COULD]
Not implemented

`ce4` customers are admitted in (near) arrival monotonic order [SHOULD]
Implemented!

`ce5` customers are admitted in random order [COULD]
Not implemented

`ce6` customers are admitted in some pre-/yet to- be determined order [COULD]
Not implemented

`ce7` customers are admitted only while the capacity of the store is not exceeded [MUST]
Implemented!

`ce8` customers are admitted with a constant flow [SHOULD]
Implemented!

`ce9` customers abandoning shopping runs does not lead to changes in service [SHOULD]
Implemented!

`ce10` customers can know when they approximately will be admitted [SHOULD]
Not implemented

`ce11` customers can know how many other customers are also awaiting to be admitted [COULD]
Not implemented

`ce12` customers cannot know when they approximately will get admitted [COULD]
Not implemented

`ce13` customesr cannot know how many other customers are also awaiting to be admitted [COULD]
Not implemented

`cp1` any customer can pay [MUST]
Implemented!

`cp2` no customer can pay [MUST]
Implemented!

`cp3` in some pre-/yet to- be determined order [COULD]
Not implemented

`cp4` only privileged customers can pay [COULD]
Not implemented

`tb1` no customer gets a final product unless the complete product can be reserved [MUST]
Implemented!

`tb2` random customers will get the final products [COULD]
Not implemented

`tb3` customers admitted the earliest get the final products [COULD]
Not implemented

`tb4` customers admitted the latest get the final products [COULD]
Not implemented

`tb5` customers having the highest value of products in their basket will get the final products [COULD]
Not implemented

`ma1` c_2 arrives more than 1 second after c_1 [MUST]
Not implemented

`ma2` c_2 arrives more than 250 milliseconds after c_1 [SHOULD]
Not implemented

`ma3` c_2 arrives more than 100 milliseconds after c_1 [COULD]
Not implemented

`dp1` application runs on Linux [MUST]
Even though the application is containerized, these containers are linux based, so the system can run natively on Linux as well

`dp2` application runs on Windows [WON'T]
Implemented!

`dp3` application is containerized [SHOULD]
Implemented!

`dp4` application runs in a Kubernetes cluster [SHOULD]
Implemented!

`dp5` application scales with load [SHOULD]
Implemented!

`dp6` application is failure resilient [MUST]
Implemented!

`mt1` application handles a single store [MUST]
Implemented!

`mt2` application handles multiple shops [SHOULD]
Implemented!

`mt3` when a single store is experiencing a flash crowd, others shops should remain unaffected depends ver:mt1 [SHOULD]
Implemented!

`mt4` store operators can have a guaranteed number of customers in their store [SHOULD]
Implemented!

`mt5` store operators can have a guaranteed limit on waiting times [WON'T]
Implemented!

`me1` application works without supervision, based on some pre-specified operation policies [SHOULD]
Not implemented

`me2` application keeps track of metrics [MUST]
Implemented!

`me3` application keeps track of total number of waiting customers depends ver:me2 [MUST]
Implemented!

`me4` application keeps track of total customer influx depends ver:me3 [SHOULD]
Implemented!

`me5` application keeps track of total customer outflux depends ver:me3 [SHOULD]
Implemented!

`me6` application keeps track of number of waiting customers per shop depends ver:me2 [SHOULD]
Implemented!

`me7` application keeps track of customer influx per shop depends ver:me6 [SHOULD]
Implemented!

`me8` application keeps track of customer outflux per shop depends ver:me6 [SHOULD]
Implemented!

`mc1` Products can only be purchased from shops they are offered in [MUST]
Implemented!

`mc2` Customers have a limited amount of retries for achieving a successful payment [SHOULD]
Not implemented

Appendix F

Flatbuffer and gRPC example

This appendix contains an example of the Flatbuffer IDL used by components to communicate and contains a specification of the gRPC service. The listed specification lists a number of “tables” and is followed by the declaration of a gRPC service.

Tables are comparable to structs in C and depict the structure of the data. Flatbuffers require the use of tables. Even bare properties, a string for instance, needs to be encapsulated in a table. The example lists 3 tables which are used for the communication between the Customer Container and the Reservation Controller.

Properties in tables can have many different types, many numeric scalars of different bit lenghts, booleans, strings, vectors of some type, and references to other tables. Vectors are simply the type of the elements of the vector infixed between brackets.

The “rpc_service” block describes a service called “Cc” to be implemented by the Reservation Controller, i.e. these are the methods provided by the Reservation Controller to the Customer Container. The “Reserve” call (for instance) requires an instance of a table called “ReservationRequest” as part of the request and that a “Reservation” table instance is added as part of the response.

The table name prefixed with “services.any.” points to tables listed in a different namespace. The contents of the tables in this namespace are not listed.

gRPC offers support for sending or retrieving multiple tables for a single request. An example of this is also listed below, the “Reservations” call retrieves all reservations for a customer. `(streaming: "server")` describes that the component providing the “Cc” service (Reservation Controller in this case) returns multiple instances of a “Reservation” table. If “server” was replaced with “client”, the client can send multiple `any.services.String`s and get returned a single “Reservation”.

```
include "any.fbs";
namespace services.ccrc;

table ReservationRequest {
    path: [string];

    retrieve: bool;

    customer_id: string;
    store_id: string;

    privilege_cat: int;
    privilege: int;
}

table Reservation {
    id: string;
    reservation: string;
    children: [Reservation];
    valid: bool;
    modifiable: bool;
}
```

```
table ReservationSet {
    reservations: [Reservation];
}

rpc_service Cc {
    Reserve(ReservationRequest): Reservation;
    Reservations(services.any.String): Reservation (streaming: "server");
    Finish(services.any.String): services.any.EmptyResponse;
    Pay(services.any.String): services.any.EmptyResponse;
}
```

Appendix G

Ping tests

In this appendix the results of the ping tests are described. These served to determine whether networking issues could play a role in the perceived performance of the proof-of-concept when testing for correctness.

Each test consisted of 10,000 pings, with an interval of 0.1 seconds, to the load-balancer in front of the cluster. Though this test does not definitively prove that networking delays and related issues are not a major factor in communication with the cluster, the topic cannot be investigated deeper. Since the load-balancer and networking communication within the cluster cannot be investigated.

In total three tests were run, at around 8:30, 12:00 and 18:30 on a Saturday from the same machine as the functional tests were run from. The command used was: `ping -c 10000 -i 0.1 waas.openticket.tech`. These times were selected to see whether this was of influence on the results. The expectation was that at different times of the day, the datacenter where the proof-of-concept is hosted would be experiencing different load, which would show in the results.

For each test, the results are given. For each test the results were also cleaned, all measured rtts with a difference to the mean greater than 1.5 times the IQR were removed.

As expected, the tests executed later in the day showed a slowdown. However, the differences are negligible, the uncleaned 18:30 average rtt is around 4% slower than the uncleaned average at 8:30.

G.1 8:30 results

Measure	Raw measurements	Without outliers
Number of measurements	10,000	9,945
Minimum rtt	0.272 ms	0.272 ms
Maximum rtt	3.97 ms	0.701 ms
Average rtt	0.474 ms	0.471 ms
Standard deviation	0.100 ms	0.073 ms

IQR: 0.115

G.2 12:00 results

Measure	Raw measurements	Without outliers
Number of measurements	10,000	9,925
Minimum rtt	0.283 ms	0.283 ms
Maximum rtt	3.97 ms	0.725 ms
Average rtt	0.489 ms	0.486 ms
Standard deviation	0.095 ms	0.078 ms

IQR: 0.121

G.3 18:30 results

Measure	Raw measurements	Without outliers
Number of measurements	10,000	9,980
Minimum rtt	0.333 ms	0.333 ms
Maximum rtt	4.08 ms	0.767 ms
Average rtt	0.495 ms	0.492 ms
Standard deviation	0.113 ms	0.082 ms

IQR: 0.139