Universiteit Utrecht

bol.com

Automating Resilience Tuning

Computing Science Thesis

Author:
Lars Hartmann (3344835)
l.b.l.hartmann@students.uu.nl

Supervisors Utrecht University:
Marjan van den Akker
Jan Martijn van der Werf

Supervisor Bol.com:
Carst Tankink

Abstract

Resilience frameworks are often used in microservice software ecosystems to ensure that failure in a service does not propagate throughout the rest of the ecosystem. These frameworks need to be tuned for each connection they are applied to, which is a time-consuming task. In this work, the possibilities of using discrete event simulation to help automate the tuning process of resilience frameworks in microservice software ecosystems is studied. A simulation model is designed, based on the microservice ecosystem at bol.com, a large online retailer in the Netherlands. The model simulates a connection between two services in the bol.com ecosystem. The model is improved upon iteratively during the project as a result of new insights gained from interviews with domain experts at bol.com and results of experiments. Using system logs available at bol.com, an input analysis is performed. In collaboration with domain experts at bol.com, performance measures are designed. Finally, this simulation model is then used by an optimization heuristic which tests different configurations. The results of the experiments show some interesting traits of the configuration parameters and relations they may or may not have with the effectiveness of the resilience framework.

Contents

1 Introduction

Websites are often backed up by services. These services are pieces of software that run on dedicated machines, and supply the website with the data necessary to function. When a service needs data from another service in order to operate correctly, it is called dependent on another service. In order to retrieve this data, a service can send a request for it to another service. The service that receives the request then responds with the requested data.

In large software projects, often developed by multiple teams, it is often beneficial to move from monolithic systems, where one service does all of the work, to an ecosystem of microservices, where the work is divided into smaller parts and distributed among these microservices. These microservices are easier to build and manage independently of each other. This allows multiple teams of developers to build and maintain different parts of the ecosystem more easily.

Another benefit of microservices is that failures are better contained. A failing service is a service that experiences reduced functionality. When a large monolithic service fails, it will take the entire system down with it. When a microservice fails, only a small part of the ecosystem fails, while the rest is free to continue operation as normal. Other services that depend on the failing service might experience some reduced functionality themselves, but the overall ecosystem will continue to operate. However, failure is "infectious" and can spread from a single microservice to the entire system.

When a service starts failing or timing out it will naturally affect other services that depend on it. However, it can also affect unrelated services by claiming all available resources (like CPU time), leaving none for other services. This in turn means the failures or delays can spread to unrelated parts of the ecosystem. All these unrelated parts can then propagate failures even further throughout the system until the entire ecosystem fails.

This is why services should be made *resilient*. In this context, resilience is defined as the ability of a network to continue operating on an acceptable level even when challenged by disruptions, and the ability of the network to restore itself to its original state of operation without help from outside of the network. Resilience is a property of services that can help prevent infectious failures. By recognizing failing services and isolating them, the rest of the ecosystem remains (relatively) unaffected. An added benefit is that the failing service gets some "breathing room", due to it being isolated. This might help the failing service recover on its own.

At bol.com, a large online retailer in the Netherlands, the webshop is an application that consists of a front-end backed up by a large number (40+) of back-end microservices. In an ideal world, these services are available at all times. In practice, services sometimes run into problems. In order to keep the webshop open, the entire webshop has been made resilient. This means that whenever a service fails, or has reduced functionality, use of the service is automatically disabled. The most important tool that is being used for this is Hystrix [17].

1.1 Hystrix

Hystrix is a framework, developed by Netflix, which adds resilience against disruptions that increase the time it takes for a service to respond to a request (latency), and disruptions that cause services to run into errors when responding to a request (faults). Hystrix improves the resilience of a software ecosystem in a number of ways. This section explains how Hystrix adds resilience to microservice ecosystems, starting with how Hystrix is integrated with the ecosystem.

Hystrix is integrated into the software via the code that creates the requests for data to other services. Each of these requests gets wrapped in a Hystrix object. This object means that the request is made via Hystrix, and allows Hystrix to provide resilience strategies for this request. A connection wrapped in Hystrix like this is called a *circuit*. Hystrix has a number of mechanisms in place to make the services it is embedded in more resilient.

When Hystrix determines that the response for a request takes too long, it can decide to timeout the request. A timeout is an error that states that a response was not received in a timely manner. This way services do not end up waiting too long for a response that may never come (long response times are frequently a sign of failing systems).

Another way that Hystrix provides resilience is by bulkheading. This means that Hystrix limits the amount of concurrent requests a service can send to each service that it depends on. By limiting this number, Hystrix ensures that in the case of a service failure, the number of requests sent to the failing service are limited. If the bulk-head is full, any new requests to that service will be immediately rejected. This way, resources to send requests to other non-failing services do not get claimed by requests to a failing service.

Hystrix also measures the health of a service. This is done by measuring the amount of requests that:

Succeed Successful requests that get a response in a timely manner.

Fail Requests that fail due to an error in the responding service.

Timeout Request that do not receive a response in a timely manner, and are therefore interrupted by Hystrix.

Rejected Requests that get rejected because the bulk-head is full.

Based on this health measure, Hystrix can decide to open a *circuit breaker*. A circuit breaker is a piece of code that, when opened, stops all requests to a particular service for a period of time. After this period of time, the service health is re-evaluated and Hystrix can decide to close the circuit breaker, allowing requests to reach the service, or to keep the circuit breaker open for a longer period of time.

When a request that is handled by Hystrix does not receive a desired response for any of the reasons stated above, Hystrix will attempt to respond with a fallback response. The fallback response is an emergency response that lets the service making the request know that something went wrong, but also allows the service to operate, possibly with reduced functionality. When a request that is handled by Hystrix does receive a response in a timely manner without anything going wrong, the response is handed to the requesting service.

Finally, Hystrix also gathers and reports metrics on the traffic that passes through it and the health of the services it connects to. These metrics are gathered and reported in near real-time. Hystrix also monitors configuration changes in near real-time. This allows engineers to reconfigure Hystrix and see how their changes affect the system in near real-time.

1.2 Problem statement

When using Hystrix (or any other resilience frameworks, such as Istio [8]), it is important to make sure it is configured correctly. Hystrix has many different properties that need to be configured correctly for each different service. A badly configured Hystrix command can lead to false positives or false negatives.

False positives happen when a resilience framework determines that a healthy service is failing, and starts to throttle incoming traffic to that service. This leads to an unnecessary reduction in service. False positives can happen when the configuration of a resilience framework is tuned too strictly, and takes action despite the service being healthy.

False negatives happen when a resilience framework does not notice that a service is failing, and lets all traffic through unimpeded. This means the failing service does not receive any breathing room, and the resilience framework is not doing what it is meant to do. This can also lead to a failing service slowing down all other services that depend on it, for example because requests take much longer to fulfill. False negatives can happen when the configuration of a resilience framework is tuned too liberally, and does not take any action despite the service failing.

Tuning the configurations of these resilience frameworks is a job that is not trivial to do. This is because the tuning depends on a lot of outside parameters. According to Netflix [17] and interviews with domain experts at bol.com [10] important parameters for tuning are peak traffic, mean response time, and accepted error rates. These parameters and their values can differ for each connection between two services.

In the Hystrix documentation [17], Netflix suggests starting with a quite liberal configuration, letting the system run in a production environment for a day or so, and to then tune the configuration more strictly based on the latency percentiles and traffic volume gathered by Hystrix's logging of the past day. Netflix does not give a concrete strategy on how to tune the configuration more strictly, but they do offer some guidelines. In practice, developers of different teams at bol.com note [10] that the initial configuration rarely gets updated after some time to be more strict. This in turn results in disruptions not being detected as efficiently, which means that a failing service consumes more resources than they would with a stricter configuration. The reason for this is that, at the moment at bol.com, tuning the Hystrix commands is done manually, and as such is a difficult and time-consuming task.

In order to evaluate different resilience configurations, their effects on a production environment need to be monitored. This makes testing different resilience configurations difficult, because that means they need to be tested in a production environment, where they could possibly disrupt real traffic. Other environments simply do not have traffic like a production environment, which means they can not give an accurate display of how the configuration interacts with the real traffic. An alternative would be to create a simulation of a production environment, including production traffic profiles. This way, testing and evaluating different configurations can be done without disrupting the production environment.

There are no clear strategies available to tune Hystrix configurations, and the teams at bol.com have no way of testing new configurations under production-like circumstances without actually testing on real production servers. They do have an acceptance environment which is set up to be a copy of the production environment, including similar data, but that does not have the same traffic profile as the real production environment. Unfortunately, traffic is one of the most important factors when configuring Hystrix.

With this in mind, the aim of this project is to obtain knowledge and insight concerning the evaluation of resilience configurations by creating a simulation of (a part of) the microservice ecosystem at bol.com (which is connected via a resilience framework), and testing and evaluating different configurations within this simulation model. Interviews with domain experts and data from production system logs will be used to help the development of the simulation model.

1.3 Discrete event simulation

Simulations are used to model the real world operation of a system on a computer, they allow many "what if?" scenarios to be tested without affecting real world systems. A simulation generally consists of a simulation model, the simulation clock, and the simulation state. The simulation model is a representation of the real world system that is being simulated. It describes how different parts of the system interact with each other. The simulation clock keeps time for the simulation. Simulations can process time faster than real time, which is why a simulation clock is necessary to keep track of the simulated timespan. The state of a simulation is a set of variables that describes the current state of the simulation.

Discrete event simulation is a type of simulation that assumes that the simulated system does not undergo any changes in between distinct events, therefore it can skip from event to event. Each event happens at a certain point in time, and these events are the only things that are capable of changing the state of the simulation. This also means that every change to the system (such as incoming requests, or service goes down/comes back up) needs to have a corresponding event. Jumping in time from one event to the next significantly reduces the runtime needed to simulate a timespan. In order to keep track of all upcoming events an event list is needed in which all upcoming events are ordered by the time the event occurs. This is typically implemented with a priority queue sorted by event time. It allows the simulation to obtain events in a chronological order, even if they were not inserted into the queue in a chronological order.

The advantages of using a simulation to experiment with real world systems are numerous, a few advantages are listed below:

Cost The cost of creating a simulation of a real world system are often quite low compared to the cost of the actual real world system;

Safety Experiments run in a simulation can not harm or influence the operation of a real world system.

Variable Simulations allow experiments to be run under different circumstances or different variants of a real world system.

Repeatability Simulations are able to repeat the exact same experiment, with the exact same environment, multiple times.

Using a simulation can also bring with it some disadvantages. Some of the disadvantages are listed below:

Cost Building and running a simulation can be very expensive (in terms of time and resources) compared to other research methods.

Information required Building a simulation requires a lot of in-depth knowledge and information about the system.

Abstraction Building a simulation requires abstraction of the system. It is possible that this abstraction oversimplifies the system, which could make the results of the simulation unreliable.

2 Literature study

The subject of this project is very specific. It is about using a specific tool (discrete event simulation) to attempt to solve some problem (evaluating resilience framework configurations) in a certain environment (microservice software ecosystem). Because of this, previous work in this very specific field was not found. However, previous work in related fields has been found.

In this chapter all the identified research is listed and described, as well as the takeaways that are relevant to this project. This chapter will start with identifying why simulation could be used, followed by research on how simulation can be used. Next, a number of studies showcasing different simulation tools will be reviewed. Following this, research on how to setup a simulation model and the challenges in the field of simulation are reviewed. Finally, work that will be helpful in defeating these challenges is identified.

The big question is: Why use simulation for this? In the paper "Software process simulation modeling: Why? What? How?" [15] the purpose of simulating a software development process is clustered into 6 categories:

- Strategic management;
- Planning;
- Control and operational management;
- Process improvement and technology adoption;
- Understanding;
- Training and learning.

The most interesting purpose, with regards to this project, is understanding. The goal of this project is to understand more about the impact of different resilience configurations. The authors talk about simulating software development, but this can be extended to simulating actual software. The category control and operational management seems interesting for this project as well, however this category encompasses the project management side of a simulated process, such as development time. In this sense, simulations can help managers determine if corrective action is needed for a development that seems to struggle with certain issues.

The most closely-related work is a study by Schaeffer-Filho, Smith & Mauthe on policy-driven network simulation [18]. In their paper, the authors propose a combination of a network simulator and a policy management framework to develop a network resilience simulator. The policy management framework acts as a resilience framework, in that it can be used to enact certain predefined resilience policies. They demonstrate a basic functionality of their model, namely that policies can be altered and administered during runtime. This is demonstrated by limiting the bandwidth of a connection between two servers during operation. Their work focuses on network infrastructures and how they can be combined with policy management frameworks to manage different amounts of traffic. Furthermore, the authors have combined two pre-existing tools and mainly focus on how they managed to do that. Related to this study, no valuable insights, such advantages and disadvantages of their work are given. The takeaway from their work for this project is that there are network simulation tools available and that they can be combined with policy management framework in order to simulate resilience frameworks. However, the tools showcased in this work focus mainly on the network infrastructure part (routers), whereas this project focuses more on the software side (services). Also, the development for the chosen network simulator (SSFNet) has been discontinued in 2004. It would be interesting to see if up-to-date network simulation tools exist that also allow simulation of higher-level software systems without worrying about the underlying network architecture and/or topology.

2.1 Simulation tools

The following literature on different network simulation tools has been identified:

An article on **Advances in Network Simulation** [3] talks about the Virtual InterNetwork Testbed (VINT) project, which is a simulation tool for network researchers. This tool focuses on network infrastructure and topology, whereas this project focuses more on the software of a microservice ecosystem.

A study on **SimFlex** [9] details how this tool can be used to simulate a full system, up to and including the complete instruction set architecture and all connected peripherals. This tool is built to simulate hardware systems, but this project aims to simulate software systems.

In **Gremlin: Systematic Resilience Testing of Microservices** [13], a different approach from simulation is offered for resilience testing. The authors have developed a tool that functions as a sort of man-in-the-middle. All services interact with each other via this Gremlin tool, which then allows researchers to fake disruptions in the network. This allows the researchers to observe the behaviour of the affected systems. Gremlin is also able to gather metrics from all traffic it handles. Using a tool like this means that you are disrupting your business to research the effect on disruptions on your system, which is the exact issue that simulating the environment tried to avoid. Furthermore, engineers at bol.com have experience with these kind of tools, and note that these kinds of tools seem to scale very poorly, making them less useful for larger systems [10].

The authors of **Network-Oriented Full-System Simulation using M5** [1] showcase M5, a simulation tool that can deal with the challenges of ever increasing I/O data rates. It provides a detailed performance model of all I/O activity during a simulation. This tool focuses on I/O, which is out of scope for this project.

Model-driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches [6] is a study in which another alternative to simulation is suggested. The authors have created a tool that can mirror an existing software ecosystem, and allow researchers to perform tests and experiments on that mirror copy. The copy on which to experiment will run in real time, whereas simulations can run faster than real time. This means that this tool will take significantly more time to simulate a timespan.

2.2 Simulation design

With none of the tools identified above being suited for the aims of this project, the alternative is to develop a dedicated simulation. The next step, with or without simulation tooling, is designing the simulation model. The authors of the paper Software process simulation modeling: Why? What? How? [15] also talk quite extensively about what to model in a simulation. The key aspects are model scope, result variables (output metrics), process abstraction, and input parameters. Figure 1 demonstrates the relationship between these aspects and the purpose of the model. This model will be especially useful when designing the simulation model.

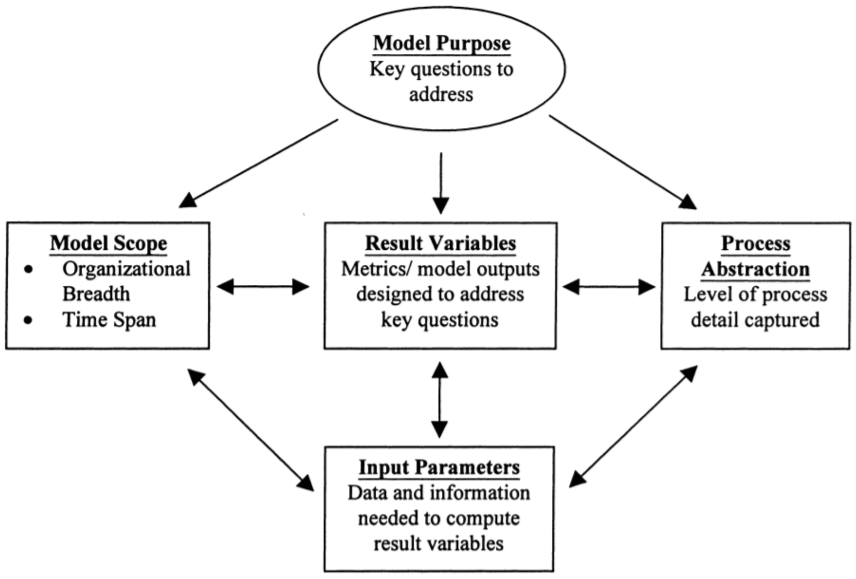
Model Purpose
Key questions to address
Model Scope
• Organizational Breadth
• Time Span
Result Variables
Metrics/ model outputs designed to address key questions
Process Abstraction
Level of process detail captured
Input Parameters
Data and information needed to compute result variables

Figure 1: Relationship between aspects and purpose of a simulation model

Of course there are challenges when trying to simulate microservice ecosystems. The authors of Performance Engineering for Microservices: Research Challenges and Directions [12] talk about some of these challenges. The three main areas in which they describe these challenges are performance testing, monitoring, and performance modeling. One of these challenges is finding an appropriate modeling abstractions. This study will be helpful in identifying the relevance of this project.

In the paper Formalizing software ecosystem modeling [2], the authors talk about the need for a formal modeling standard for software ecosystems and the environments in which they live. They propose a set of rules that help identify relationships between different parts of the ecosystem, which in turn can be developed into a model of the software ecosystem. The model that the authors describe in this paper is a model that can be used to explain the software ecosystem in boardroom meetings with (primarily non-technical) stakeholders. The proposed method is an interesting take on modeling software ecosystems, but it is not suited for creating simulation models of a software ecosystem.

Finally, authors of a paper on network resilience evaluation [19] talk extensively about what resilience is, what different types of resilience there are, and how resilience can be quantified. Methods of quantifying resilience in terms of operational state and service level of the system, and on the temporal aspects of resilience are identified. Although the majority of the paper focuses on the physical infrastructure of networks and how they can be made more resilient, the quantification of resilience will be useful to help design the performance metrics for this project.

3 Research Question

The goal of this project is to obtain insight and knowledge concerning the evaluation of different configurations for resilience systems by creating a simulation of (a part of) the microservice ecosystem at bol.com. To reach this goal, the following main research question has been defined:

"How can discrete event simulation be used to support the decision-making process of tuning resilience framework configurations in a microservice software ecosystem?"

To help answer this question, the following sub-questions have been defined:

1. How can a microservice software ecosystem that uses a resilience framework be modelled as a discrete event simulation model?
2. How can different resilience configurations be evaluated and compared?
3. What different resilience configurations are interesting to compare?
4. Do the results of the experiments offer valuable insights into tuning resilience framework configurations?

The designed simulation model will be implemented, and different resilience configurations will be evaluated. These experiments and their results are also an important goal of this project. After the implementation of the simulation model and the experiments, the sub-questions can be answered, and subsequently the main research question can be answered. The design of the simulation model will be based on the microservice ecosystem back-end of the bol.com webshop, which uses the Hystrix framework for resilience.

4 Analysis of the system

The system being simulated is a part of the microservice ecosystem at bol.com. It consists of two services and the connection between them. This connection is wrapped in a Hystrix command. The simulated services are the Inspire-to-Sell (I2S) service and the Sellingoffer-Integration-Service (SLI). The Inspire-to-Sell service shows advantages of ordering a product at bol.com, such as delivery options and times. To display this for these products, it requires pricing and delivery data from the Sellingoffer-Integration-Service. These services and the connection between them are used as data providers for the simulation. The data supplied by these services are used to transform the abstract simulation implementation into a concrete representation of a connection in the bol.com microservice ecosystem.

These services are called service A (I2S) and service B (SLI) in the simulation. The I2S service (service A) receives a request that requires it to talk to the SLI service (service B). Service B can handle requests from service A, but needs some time to do so. In order to protect itself, service A has wrapped its calls to service B in a Hystrix wrapper.

In Figure 2 the flow of a request through this system is shown. Upon receiving a request, service A asks Hystrix to send a request to service B. If Hystrix allows this, the request is sent and a timer is set. If the timer reaches zero before Hystrix has received a response from service B, Hystrix will cut the processing of the request short and tell service A it was unable to get a response from service B. If something goes wrong during transmission of the request to service B, the HTTP driver will notify service A. If the request does reach service B, it is offered to the request queue of service B. If the queue is full, it will reject the request, and service B sends an error response to service A immediately. This process is similar to the bulkheading process in Hystrix, the difference is that the Hystrix bulkhead protects the client (service A), and this request queue provides a limit to service B. Hystrix has no insight in the state of the request queue of service B.

If the queue accepts the request, it will wait in the queue until service B is idle and takes a new request from the queue. When the request is taken from the queue, service B will start processing it. When service B finishes processing the request (successfully or unsuccessfully), it will send a response to service A with the result. Finally, this allows service A to finish processing the request.

For bol.com it is interesting to see how Hystrix behaves when a part of the system malfunctions, in particular when service B malfunctions. Service B can malfunction in two different ways:

1. It breaks completely. Service B drops all requests in the queue and no longer accepts new requests. The service becomes completely unresponsive.

2. It becomes very slow. Service B keeps operating as it normally would, except that it takes much more time than normal to process a request.

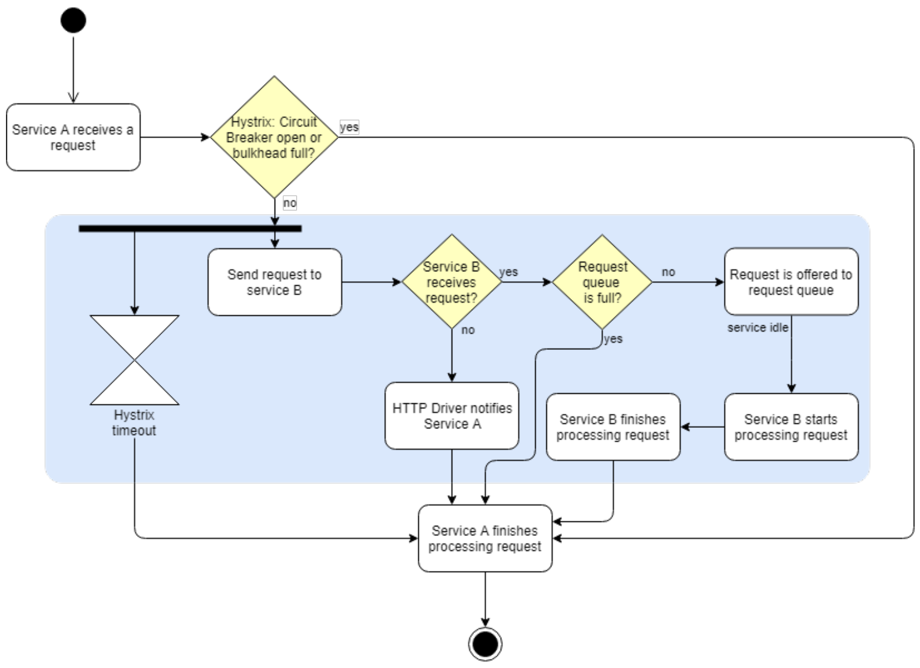
Service A receives a request
Hystrix: Circuit Breaker open or bulkhead full?
yes
no
Send request to service B
Service B receives request?
yes
no
Request queue is full?
no
yes
Request is offered to request queue
service idle
Hystrix timeout
HTTP Driver notifies Service A
Service B finishes processing request
Service B starts processing request
Service A finishes processing request

Figure 2: Flow of a request through the analyzed part of the system

Hystrix acts in accordance to its configuration. According to domain experts at bol.com [10], most of the services at bol.com use the default Hystrix configuration. A Hystrix configuration consists of the 5 following parameters:

Sleep period When Hystrix determines a circuit is unhealthy, it blocks all traffic over this circuit for a period of time before checking to see if it has recovered. The default value is 5 seconds.

Volume threshold Hystrix considers the circuit health a valid metric only if a minimum number of requests has been made in the last 10 seconds. The default value for this threshold is 20.

Error rate threshold A Hystrix circuit is deemed unhealthy when the error rate exceeds this threshold. By default, the error rate has to exceed 50%.

Bulkhead size The size of the bulkhead for this circuit is determined by this parameter. The default configuration for this parameter is 10.

Timeout delay This parameter specifies how long Hystrix allows a request to go on before it interrupts the request for taking too long. This is set to 1 second by default.

5 The simulation model

This chapter aims to give an overview of the simulation model, which is based on part of the bol.com microservice ecosystem back-end and its implementation of the Hystrix resilience framework. More specifically, the model is based on the Inspire-to-Sell (I2S) service, the Sellingoffer-Integration-Service (SLI), and the connection between these services. Section 5.1 describes the assumptions underlying the simulation model. In section 5.2 the performance measures are described. Furthermore, the state of the simulation model is detailed in section 5.3, and in section 5.5 a specification of the events of the simulation model and an overview of their corresponding event handlers is given.

5.1 Assumptions

All simulations are modelled with certain assumptions in mind. These assumptions help define the scope of the simulation. The following assumptions were made during the development of the simulation model:

- Any failure in processing a request is not the fault of service A. Only service B can fail during processing a request.
- A request is fully processed once service A returns a response of any kind. This means that requests are also fully processed if the response contains a non-successful result.
- Service A receives only valid requests.
- Service A needs no additional time to process a request beyond the time it takes to get a response from service B.
- The network between services works flawlessly and instantly. This means that the probability of service B not receiving the request is 0. In this case, the HTTP driver only notifies service A of an error if a request is sent to service B while it is broken down completely.
- The Hystrix fallback response is instantaneous, satisfying, and identical for all requests that require it.

Most of these assumptions are made to scope the simulation to those parts that Hystrix can influence.

5.2 Performance metrics

The performance of the system is based on the following performance metrics:

Number of requests Split into categories for each possible result of a processed request. This means that, alongside numbers for successful requests, separate metrics for each different error type are tracked (circuit-breaker, bulkhead, Hystrix timeout, HTTP error, and service B error).

Average processing time This metric gives insight into how long requests take on average to fulfill. An average over all requests is available, as well as averages for only successful and for only failed requests. The processing time for a request is the time that service A needs to send a response to the request. It is measured by the time difference between the moments service A receives a request, and the moment service A send a response to a request (the moment the request is fully processed). These values are tracked by a variable that holds the sum of the processing times of all (successful, failed, and all) requests. Finally, this sum can be divided by the number of (successful, failed, or all) requests to get an average value.

Error rate The fraction of all requests that result in an error. This metric shows the health of the system.

False negative time Time during which the Hystrix circuit is closed (letting requests through), but the service has broken down (or become latent). This measures how quickly Hystrix reacts to a failing service.

False positive time Time during which the Hystrix circuit is open (stopped requests), but the service is healthy. This measures how quickly Hystrix reacts to a recovered service.

Service B downtime The amount of time service B is broken down.

The false positive and false negative time metrics are based on the work by Sterbenz et al. on network resilience evaluation [19]. In their paper, a model for measuring resilience is proposed, based on the level of service provided and the operational state of the system. This model is used as the inspiration for the metrics in this simulation. The level of service for a single request provided in this simulation is a binary variable, either the request returns successfully, or it returns a fallback response (which is assumed to be equal between all requests). Note that an identical service level for all errors (a fallback response) does not mean all errors are identical. It only means that the response sent back when an error occurs is the same for all errors. Internally, the different errors are still tracked separately.

The operational state is also modelled as a binary variable, the service is either healthy, or it is not (either it is broken completely, or it has become slow). Because no further progression is simulated between the unhealthy operational states (for example, a service cannot progress from being slow to breaking down entirely), both these states can be seen as simply unhealthy.

In their work, the operational state is said to decrease over time when a service starts to break down. However, in this simulation the decrease in operational state is immediate and binary. Service B is either healthy or unhealthy, and this transition happens in a single moment. Similarly, the service level of the response is binary as well, either a successful or a fallback response is returned. With the proposed model and these differences in mind, the false positive time and false negative time metrics have been designed. The idea is to measure how fast Hystrix is able to react to a failing service, with the aim of minimizing the time Hystrix takes to react. In Figure 3 this model is visualized.

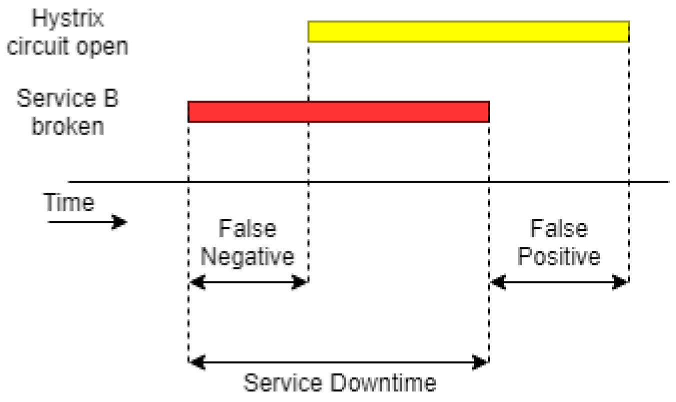

Figure 3: Visualization of the proposed performance metrics: false negative time and false positive time.

These three metrics (false negative time, false positive time, and service B downtime) can be calculated by keeping track of the most recent times that service B breaks down and recovers, and the most recent times that the Hystrix circuitbreaker opens and closes (this signifies that Hystrix has determined that service B is unhealthy). When any of these four things happens, the timestamp is saved. In the case of a service B recovery happening, the service B downtime metric is updated with the difference between the timestamp that service B broke down and the time at which service B recovered. Next, the State is checked to see if a period of false positive time or false negative time has occurred. If it has, the length of this period is calculated by determining the difference between the current time and the start of the period. The start of the period is determined by the largest of two parameters, which are different based on whether false negative time or false positive time is being calculated. In the case of false negative time, the last time that service B broke down or the last time that the Hystrix circuitbreaker closed signify the start of the period. In the case of false positive time, the last time that service B recovered or the last time that the Hystrix circuitbreaker opened signify the start of the period. Algorithms 1 through 4 show the pseudo-code the metrics use to determine calculate these periods.

Algorithm 1 Update false negative time metric when Hystrix circuitbreaker opens.

```
Set lastHystrixOpenTime = time
if Service B is not healthy then
  Update falseNegativeTime += (time - max(lastServiceBBreakdownTime, lastHystrix-CloseTime))
end if
```

Algorithm 2 Update false positive time metric when Hystrix circuitbreaker closes.

```
Set lastHystrixCloseTime = time
if Service B is healthy then
    Update falsePositiveTime += (time - max(lastServiceBRecoveryTime, lastHystrixOpenTime))
end if
```

Algorithm 3 Update false positive time metric when service B breaks down.

```
Set lastServiceBBreakdownTime = time
if Hystrix circuitbreaker status is OPEN then
    Update falsePositiveTime += (time - max(lastServiceBRecoveryTime, lastHystrixOpenTime))
end if
```

Algorithm 4 Update false negative time and service downtime metrics when service B recovers.

```
Set lastServiceBRecoveryTime = time
Update serviceBDowntime += (time - lastServiceBBreakdownTime)
if Hystrix circuitbreaker status is not OPEN then
    Update falseNegativeTime += (time - max(lastServiceBBreakdownTime, lastHystrixCloseTime))
end if
```

5.3 State

The state represents the reality of the simulation at a certain point in time. This reality consists of different relevant variables that are manipulated by the events of the simulation. In this section an overview of the different state variables is given. The state consists of the following elements:

Hystrix Tracks the variables Hystrix needs to monitor the circuit during the simulation. These variables are:

- Current number of concurrent requests
  To determine if the bulk-head is full or not, the current number of concurrent requests is needed. This number indicates how many requests have been sent to service B and have not yet received a response of any kind.
- Circuit health
  Keeps track of the health of a circuit. Hystrix opens and closes the circuit breaker based on the health of a circuit. The circuit health consists of 10 buckets. Each bucket keeps track of the number of failed and successful requests that have happened in a second. It keeps track of separate counters for the different kind of failed requests (due to error in service B, Hystrix timeout, bulkhead rejection, circuitbreaker rejection, or bad arguments). If the health of a circuit drops below a set threshold, Hystrix opens the circuit breaker. Note

that this is a measure that Hystrix uses to determine the health of a service. It is not the same as the service level of a service.

- Circuit breaker status
  The circuit breaker has three different possible states. The *CLOSED* state indicates that the circuit works as normal, letting requests through. The *OPEN* state breaks the circuit, meaning that Hystrix rejects all requests. Finally, the *HALF OPEN* state allows one request through, to test if a circuit has recovered.
- Circuit opened timestamp
  Keeps track of when the circuit breaker was opened, if it is currently open. If it is not, the timestamp is set to -1.

List of requests The progress of all requests that are currently being processed needs to be registered. In order to reduce the required memory to run a simulation, requests that have no future events scheduled anymore are removed from the State. Variables to be saved are:

- Request ID
  An ID to identify the request.
- Request status
  This variable indicates where in the process this request is. Possible values are: *RECEIVED_A*, *HYSTRIX_TIMEOUT*, *HTTP_ERROR*, *RECEIVED_B*, *SUCCESS_B*, *ERROR_B*, *HYSTRIX_CIRCUITBREAKER*, *HYSTRIX_BULKHEAD*, *PROCESSING_B*, *PROCESSED_A*. These values correspond to the different stages a request can go through during processing. It is used to determine the type of response sent back by service A.
- Request received timestamp
  A timestamp that indicates when the request was initially received.
- Request path status
  Request paths are paths through the request flow that the request can walk. This means that while the request is walking any of these paths, the request needs to be kept in the State. The three paths are *HttpErrorPath*, *HystrixPath*, and *ServicePath*. Each path variable takes one of three possible values: *UNUSED*, *UNFINISHED*, or *FINISHED*. When none of the path status variables are *UNFINISHED*, the request can be deleted safely from the State. The purpose of these statuses is to determine if a request can be deleted from the State safely. The *HttpErrorPath* is the path through the *HTTP error received* event. The *HystrixPath* is the path with the *Hystrix timeout* event. The *ServicePath* is the path that goes through the different service B events.

Service B Contains the state of Service B. This in turn keeps track of the following variables:

- Request queue
  The request queue is a first-in-first-out queue where incoming requests are stored until they can be processed. When the service is free to process a request, it polls the queue for the next request and starts processing it. The queue also has a maximum size. When the number of incoming requests is equal to its maximum size, the queue is filled and any other incoming requests get rejected instantly.
- Idle status
  Tracks if the service is currently processing a request, or if it is free to start processing a new request. Possible values are: *True* indicates the service is idle, *False* indicates the service is processing a request.
- Service status
  Tracks the current status of the service. The service can be either *HEALTHY*, *BROKEN*, or *LATENT*.

5.4 Input parameters

The simulation model requires a number of input parameters to realistically simulate the real world system. This section lists the input parameters required by the simulation. In chapter 6 an analysis for some of these parameters is performed.

Request interarrival time The time between two sequential requests reaching service A.

Probability of network failure The probability that requests are lost in the network during transfer between service A and service B. Since the network is assumed to work perfectly, this probability is 0.

Network latency The time it takes to send a request over the network between service A and service B. Again, since the network is assumed to work perfectly, this latency is also 0.

HTTP driver time to error Time it takes for the HTTP driver to determine that a request is lost in the network. At bol.com, the default value for this parameter is 1 second [10].

Probability of service B processing error The probability of encountering an error when service B is processing a request from service A.

Service B request processing time The time service B needs to process a request from service A.

Hystrix configuration A Hystrix configuration as defined in chapter 4.

Breakdown scenario Introduced in iteration 4, the breakdown scenario lists a number of breakdowns that happen during simulation. Each simulation specifies when the breakdown occurs, how long until service B recovers, and what the delay caused by the breakdown is. A delay of 0 indicates that service B breaks down completely instead of just becoming slow.

In chapter 6 an analysis is performed for the request interarrival time, the probability of service B processing error, and the service B request processing time.

5.5 Events and event handlers

Events are what actually drives the simulation forward. They are the only way to change the state of the simulation. Each event happens at a certain time, and all events are handled in order of time (events that happen first, get processed first). The events will be explained in detail in the following subsections. But first, a brief overview of all events:

Request received A The handler for this event is by far the most complicated of the entire simulation. This event signifies the arrival of a request from outside the system to service A. The handler performs a number of checks that determine how the request will be serviced.

Request received B This event is used to get the request from service A into the request queue of service B.

Request processing start B Checks if service B is currently idle and if there is a request waiting to be processing. If so, it will take the request from the request queue and start processing it.

Request processed B Models service B finishing processing a request and forming a response for service A.

Hystrix timeout Signifies Hystrix determining that a request is taking too long and interrupting the processing of the request.

HTTP error received When something goes wrong sending a request to service B, the HTTP driver will give an error. This event also occurs when service B is broken down and unable to respond to incoming requests at all.

Request processed A Service A is finished processing the request and is ready to send back a response.

Service B breakdown Service B breaks down because of some outside force. The input parameters specify if service B breaks down completely or if it becomes latent. A service B breakdown has a start time, duration, and type of breakdown. Breakdowns should not overlap.

Service B recovery Service B recovers thanks to some outside force and resumes normal operation.

Figure 4 shows the event graph for the proposed model. The *Request received A* event is the entry point. An initial *Request received A* event is scheduled upon initialization of the simulation, and from that initial event all future events (except *Service B breakdown* and *Service B recovery* events) are scheduled. The *Service B breakdown* events for each breakdown specified in the input parameters is also scheduled upon initialization of the simulation. These events schedule their *Service B recovery* events when they occur.

The *Request received B*, *Request processed B*, *Hystrix timeout*, *HTTP error received*, and *Request processed A* events all contain the ID of the request they correspond to. The *Request processing start B* event determines which request it is going to affect itself, so it does not contain a request ID. The *Service B breakdown* and *Service B recovery* events do not correspond to requests, and as such do not contain a request ID either.

Shown in the event graph are the different paths each request can take. The *Request received A* event is able to schedule all other events, except for *Request processed B*, *Service B breakdown*, and *Service B recovery.* It is possible that the *Request received A* event schedules multiple new events, thereby creating different paths through the graph for a particular request. These multiple paths eventually all converge back to the *Request processed A* event, but not at the same time. For this reason, the simulation keeps track of which paths a request is walking and if they have completed any path. A request is only safe to remove from the State when the simulation knows for certain that there are no more events that affect that particular request, which is when all of the paths it walks are finished. The *Service B breakdown* and *Service B recovery* events model the breakdown and recovery of service B by an outside force.

Note that latency is modelled in the event graph, however, due to the assumptions and specified input parameters, the latency between events is set to 0, effectively eliminating it from the simulation. It is, however, possible to introduce latency via the input parameters, which is why it is modelled in the graph.

Final note is that event handlers are only allowed to change the Request Status variable if this status is not *PROCESSED_A*. This means that service A has not yet sent back a response for this request, which in turn means the type of response sent is also not yet determined.

5.5.1 Request received A

This event is the entry point for the process. Ultimately this request has to be responded to in a timely fashion and with an acceptable service level. In addition to handling the event, the handler also schedules another *Request received A* event with regards to the interarrival time.

The handler for this event models the Hystrix process that happens when making a request from service A to service B wrapped in Hystrix. New requests are registered to the State. Then the event handler will ask the Hystrix module if it allows the request to go through. The Hystrix module answers with the states of the circuit breaker, which is affected by the circuit's health, and the

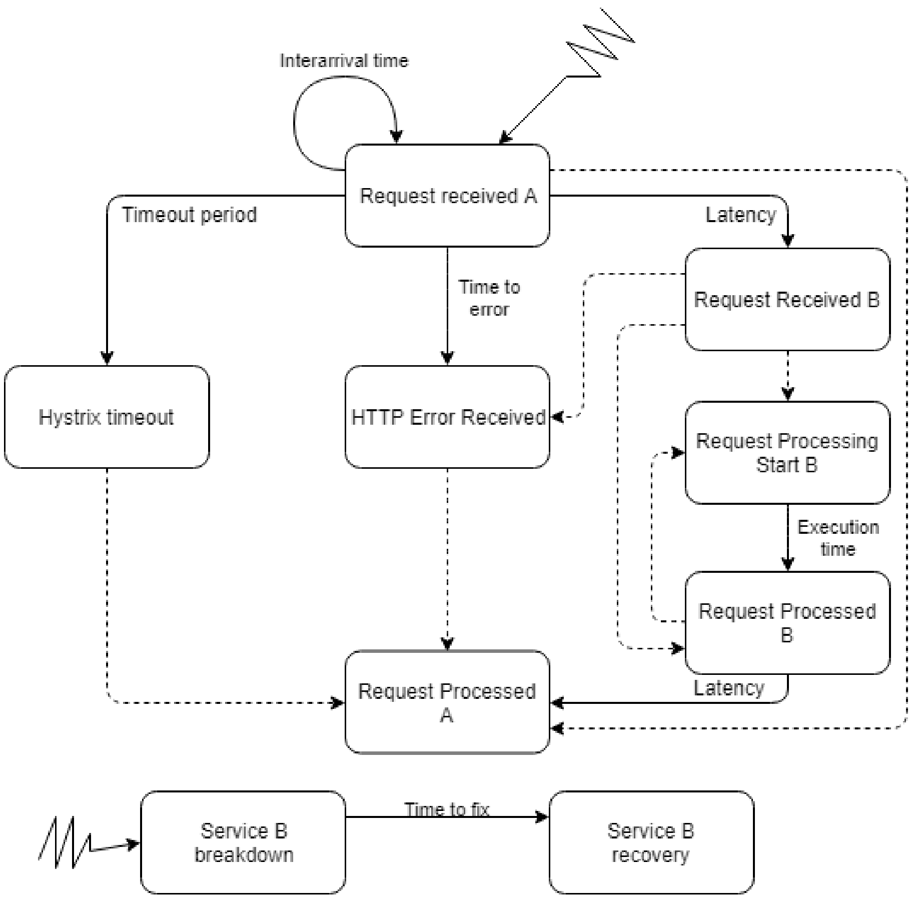

Figure 4: Event graph of the proposed simulation model.

bulkhead, which limits the number of concurrent requests to a service.

Based on the the answers the Hystrix module gives, the event handler schedules a *Request received B*, *Hystrix timeout*, *HTTP error received*, and/or *Request Processed A* event(s). The *Request received B* event is scheduled based on the latency between service A and service B. The *Hystrix timeout* and *HTTP error received* events are scheduled based on input parameters of the simulation, namely the configured Hystrix timeout value and the HTTP driver time to error value. The request path statuses are then updated according to the paths that the request will take. Algorithm 5 shows the pseudo-code for this event's handler.

Note that determining if service B will receive the request may seem like looking into the future. The reason that this is determined here is that the time of the next event scheduled is based on the result of this check. Furthermore, the State changes performed after this check are purely administrative, and serve only to help reduce the memory needed to run the simulation.

Algorithm 5 The *Request received A* event handler

```
Register new request to State
Generate interarrivalTime = time until the next request is received
Schedule new Request received A event at (current time + interarrivalTime)
Determine circuitbreakerRejects = True if the Hystrix circuitbreaker does not allow the
request through, False otherwise
Determine bulkheadRejects = True if the bulkhead is full, False otherwise
if circuitbreakerRejects OR bulkheadRejects then
  if circuitbreakerRejects == True then
    Set request status to HYSTRIX_CIRCUITBREAKER
  else
    Set request status to HYSTRIX_BULKHEAD
  end if
  Set status for all request paths of this request to UNUSED.
  Schedule new Request processed A event at current time
else
  Increment Hystrix current number of concurrent requests by 1
  Schedule new Hystrix timeout event at (current time + hystrix timeout value)
  if The network delivers the request to service B successfully then
    Set status of request path HttpErrorPath to UNUSED for this request
    Set status of request path HystrixPath to UNFINISHED for this request
    Set status of request path ServicePath to UNFINISHED for this request
    Generate latency = time needed to transfer request over the network
    Schedule new Request received B event at (current time + latency)
  else
    Set status of request path HttpErrorPath to UNFINISHED for this request
    Set status of request path HystrixPath to UNFINISHED for this request
    Set status of request path ServicePath to UNUSED for this request
    Generate timeToError = time it takes HTTP driver to throw error
    Schedule new HTTP Error received event at (current time + timeToError)
  end if
end if
```

5.5.2 Request received B

The *Request received B* event signifies that service B has received the request and offers it to the request queue. As shown in Algorithm 6 the handler for this event checks if service B is operational, if so it offers the request to the queue, and based on whether the queue accepts the request or not, it schedules one of two possible events. If the queue accepts the request, a secondary check is performed to see if the service is currently idle. If it is, a *Request processing start B* event is scheduled. If the queue rejects the request, that indicates the queue is full, and the request will get rejected immediately. In this case, a *Request processed B* event is scheduled. A rejection from the request queue results in the same error as when processing the request in service B fails. If service B is not operational, then this is as far as this request will walk the *ServicePath*. As such, the handler will only schedule a *HTTP Error received* event, update the status of the *ServicePath* request path, and delete the request from the State if it is safe to do so.

Algorithm 6 The *Request received B* event handler

```
Retrieve request from State by request ID
if Service B is not broken then
    if Request status is not PROCESSED_A then
        Set request status to RECEIVED_B
    end if
    Offer the request to the request queue
    if Request queue size < request queue maximum size then
        Request is accepted into request queue
        if Service B idle status == True then
            Schedule new Request processing start B event at current time
        end if
    else
        Schedule new Request processed B event at current time with successFlag = False
    end if
else
    Schedule new HTTP Error received event at current time
    Set status of request path ServicePath to FINISHED for this request
    if Request is safe to delete from State then
        Delete this request from State
    end if
end if
```

5.5.3 Request processing start B

This event is named this way because it is used to attempt to have service B start working on processing a request in its request queue. Algorithm 7 shows that the event handler checks if service B is operational, idle, and if there is a request waiting in the queue (note that it only reads from the State at this point). If it passes these checks, it updates the state of the service and the request (if necessary), and after that it determines if service B will be successful in processing the request. Based on this, a new *Request processed B* event is scheduled with a flag signifying the success or failure of processing the request. An additional delay is taken into account when scheduling the new events if service B is latent. If service B is not operational, this event handler does nothing.

The checks at the start are there to prevent the simulation from edge cases where a *Request processing start B* event is scheduled at the same time a *Service B breakdown* event or another *Request processing start B* event is scheduled.

Again it may seem that the event handler is looking into the future when determining if service B will process the request successfully. As with the *Request received A* event, the reason for this is that the time of the next event depends on whether or not service B is successful in processing the request.

5.5.4 Request processed B

The *Request processed B* event signifies that service B has processed the request (successfully or unsuccessfully, as indicated by a flag), and is ready to send back a response to service A.

Algorithm 7 The *Request processing start B* event handler

```
if Service B status is not broken then
  if Service B is idle then
    if There is a request waiting in the request queue then
      Retrieve the first request waiting in the queue
      Update service B idle status = False
      if Request status is not PROCESSED_A then
        Set request status to PROCESSING_B
      end if
      Set delay = 0
      if Service B is latent then
        Generate delay = additional delay in execution due to service latency
      end if
      if Service B will process the request successfully then
        Generate executionTime = time service B needs to fulfill request
        Schedule new Request processed B event at (current time + executionTime + delay) with successFlag = True
      else
        Generate timeToError = time service B runs until it throws an error
        Schedule new Request processed B event at (current time + timeToError + delay) with successFlag = False
      end if
    end if
  end if
end if
```

In Algorithm 8 the pseudo-code for this event's handler shows that this event determines if service A receives the response sent from service B. If it does, a *Request processed A* event is scheduled. If it does not, no new event is scheduled and the State is updated. If allowed, the Request is removed from the State.

Additionally, if the previous event was a *Request processing start B* event, that indicates that the service has been working on processing the request. In this case, the state of the service needs to be set to idle, and if the request queue is not empty, a new *Request processing start B* event is scheduled.

If the previous event was not a *Request processing start B* event, that means it was a *Request received B* event (as seen in the event graph in Figure 4, the *Request processed B* can only be created by these two events). This in turn indicates that the request was rejected from the request queue, and as such, service B was never working on processing it.

All of this only happens if service B is operational (in other words, not broken down completely) at this time. If it is broken down completely, the *ServicePath* status of the request is updated and no further events are scheduled for this request, signifying that the processing of this request was lost when the service broke down.

If no further events are scheduled for this request (because service A does not receive the response, or because service B is broken), the *ServicePath* status of the request is set to *FINISHED* and a check is performed to see if the request can be deleted safely from the State. Normally, this is done in the *Request processed A* event, but this path will never reach that event in this case.

Algorithm 8 The *Request processed B* event handler

```
Retrieve request from State by request ID
if Service B is not broken then
    if Request status is not PROCESSED_A then
        if successFlag == True then
            Set request status to SUCCESS_B
        else
            Set request status to ERROR_B
        end if
    end if
    if Service A will receive the response then
        Generate latency = time needed to transfer response over the network
        Schedule new Request processed A event at (current time + latency)
    else
        Set status of request path Request path to FINISHED for this request
        if Request is safe to delete from State then
            Delete request from State
        end if
    end if
    if The previous event was a Request processing start B event then
        Update service B idle status = True
        if If the request queue is not empty then
            Schedule new Request processing start B event at current time
        end if
    end if
else
    Update service B idle status = True
    Set status of request path Request path to FINISHED for this request
    if Request is safe to delete from State then
        Delete request from State
    end if
end if
```

5.5.5 Hystrix timeout

The Hystrix timeout event happens when Hystrix determines something has gone wrong with the request to service B, and decides to cut the processing of the request short. Algorithm 9 shows the pseudo-code for the event handler for this event. This event sets the status of the request (if allowed) and schedules a *Request processed A* event.

Algorithm 9 The *Hystrix timeout* event handler

```
Retrieve request from State by request ID
if Request status is not PROCESSED_A then
    Set request status to HYSTRIX_TIMEOUT
end if
Schedule new Request processed A event at current time
```

5.5.6 HTTP error received

A HTTP error received event is created to model the scenario that something goes wrong during transmission of the request from service A to service B. The HTTP error gets returned to the Hystrix module, which can then further handle the request.

Algorithm 10 shows that the event handler for this event is nearly identical to the event handler for *Hystrix timeout* event. The difference is that the request status is set to *HTTP_ERROR*. Afterwards, a *Request processed A* event is scheduled.

Algorithm 10 The *HTTP error received* event handler

```
Retrieve request from State by request ID
if Request status is not PROCESSED_A then
    Set request status to HTTP_ERROR
end if
Schedule new Request processed A event at current time
```

5.5.7 Request processed A

The *Request processed A* event models the handling Hystrix does with the response it has received upon sending the request to service B. This event is the final event in the request processing procedure.

The event handler for this event updates the state and performance metrics for this request. It also removes the request from the State if allowed, otherwise it marks the request as fully processed. Algorithm 11 shows the pseudo-code for the event handler for *Request processed A* events.

As can be seen, the number of concurrent requests that Hystrix tracks is only decreased when the request status is neither *HYSTRIX_CIRCUITBREAKER* nor *HYSTRIX_BULKHEAD*. This is because those statuses also signify that the request was never added to the current number of concurrent requests (since Hystrix does not allow the request through).

Furthermore, Hystrix also gets notified of the result of the request, so that it can update the corresponding counter in the correct circuit health bucket, and update the circuit health. The performance metrics are also notified of the result of the request, as well as the processing time of the request. This processing time is determined by subtracting the time that service A originally received the request (via the *Request received A* event) from the current time. Note that this only happens if the status of the request is not *PROCESSED_A*, signifying that this is the first time that this request reaches the *Request processed A* event. With these values, the metrics can increment the correct counter (based on the result) and update the corresponding processing time values.

As a result of Hystrix updating the circuit health, it might decide the open or close the circuitbreaker. If this happens, Hystrix will notify the metrics that this has happened so that the performance metrics can be updated accordingly (see section 5.2).

Finally, the path that the request has walked to reach this instance of the *Request processed A* event is marked as *FINISHED*, and the request is deleted from the State if all paths have been walked. If it has not yet walked all paths, the status of this request gets updated to *PROCESSED_A* to mark that a response has been sent for this request.

Algorithm 11 The *Request processed A* event handler

```
Retrieve request from State by request ID
if Request status is not PROCESSED_A then
  Determine result of the request based on the request status
  if Request status is not HYSTRIX_CIRCUITBREAKER and not HYSTRIX_BULKHEAD then
    Decrease Hystrix current number of concurrent requests by 1
  end if
  Update counter in current Hystrix circuit health bucket based on request status
  Determine processingTime = time - time request was originally received by service A
  Update metrics with request result and processingTime
end if
Determine which path the request has walked
Set the corresponding path status to FINISHED
if Request is safe to delete from State then
  Delete request from State
else
  Set request status to PROCESSED_A
end if
```

5.5.8 Service B breakdown

The handler for this event needs to determine if service B is breaking down completely or just becoming latent. The status of the service is updated accordingly. If the service breaks down completely, the request queue is also emptied. Finally, the breakdown is registered to the performance metrics (see section 5.2).

Algorithm 12 shows the pseudo-code for this event handler.

Algorithm 12 The *Service B breakdown* event handler

```
Retrieve data about the breakdown at this time from input parameters
if Service B becomes latent then
    Update service B status to latent
else
    Update service B status to broken
    Empty service B request queue
end if
Register service B breakdown to metrics
Schedule new Service B recovery event (current time + time to recover)
```

5.5.9 Service B recovery

The handler for this event is very simple, as can be seen in Algorithm 13. The event handler only updates the service B status and register the recovery to the performance metrics (see section 5.2). The flow of traffic never stopped, but now service B is able to operate normally again (accepting requests and processing them in a timely manner). All requests that were dropped and/or rejected during the breakdown are lost however.

Algorithm 13 The *Service B recovery* event handler

```
Update service B status to healthy
Register service B recovery to metrics
```

5.6 Optimization

In order to find an optimal configuration for the circuit simulated in this project, the *Simulated Annealing* heuristic [16] is used. This section describes Simulated Annealing and the implementation used.

Simulated Annealing is an optimization heuristic based on the annealing process in metallurgy. The gist of it is that the algorithm evaluates two configurations, an original and its neighbour. Based on the results of these evaluations, a probability is calculated to determine if the neighbour configuration is accepted or rejected. If the neighbour performs better than the original the probability that the algorithm accepts the neighbour is 1 . If the neighbour does not perform better than the original, the probability is less than 1, but more than 0. In this case, the probability of accepting the worse configuration is determined by the function $f(c_o, c_n, T) = e^{\frac{c_o - c_n}{T}}$, where c_o is the cost of the original configuration, c_n is the cost of the neighbour, and T is the temperature. This probability is affected by the difference in cost between the neighbour and the original, as well as the temperature of the algorithm. Lower temperatures result in lower probabilities. Larger differences in costs between an original and a neighbour also result in lower probabilities, if the neighbour has a higher cost than the original. During operation this temperature gradually decreases, until a stop condition is met (either the temperature goes below 1, or no significant improvements have been found in the last 80 evaluations).

Because of the runtime of evaluating a single configuration, a simple optimization algorithm that tries all different configurations is infeasible as it would take too long to find an optimal configuration. Simulated Annealing works in this scenario because it is a heuristic, and thus does not attempt to evaluate every possible configuration. Furthermore, it has the ability to escape local optima.

In order to implement the simulated annealing heuristic, the neighbour function has to be defined. The neighbour function takes a Hystrix configuration (the original) and transforms it into a new configuration that resembles the original, but slightly different. The neighbour function for this heuristic works as shown in Algorithm 14. It requires a parameter to determine which configuration parameter to change.

Algorithm 14 The neighbour function used by the simulated annealing heuristic.

```
Determine which configuration parameter to alter
Determine $delta$ = sampled value from normal distribution with mean = 0, sd = 1
Scale $delta$ with a scaling value specific to the chosen parameter
Calculate $newValue$ for parameter by adding $delta$ to original value of the parameter
Create new configuration using original values and $newValue$ for chosen parameter
Return new configuration
```

The Simulated Annealing algorithm uses the neighbour function to change the same parameter eight times in a row, then switches on to the next one. When it is done with the last parameter, the next time it will start over with the first parameter. The scaling values for each parameter are chosen in such a way that each neighbour only has a small difference to the original. The scaling values are shown in Table 1. These values were found after some experimentation with different values. The idea is that they allow the neighbour function to take small steps away from the original configuration, while still resembling the original configuration.

Sleep period	Volume threshold	Error rate threshold	Bulkhead size	Timeout delay
300 ms	2	3%	2	100 ms

Table 1: Scaling value for each Hystrix configuration parameter as used by the neighbour function of the simulated annealing algorithm.

With the Simulated Annealing heuristic, the number of simulations that are run per configuration is reduced. This reduction came with a caveat, the so-called "lucky ticket" problem. If a certain configuration happens to be very lucky in its evaluation, it might get a score that is impossible to beat for any other configuration, despite it not being the best configuration. A solution to the lucky ticket problem is to evaluate both configurations when comparing them. This way, the impact of a lucky score is reduced significantly. Depending on the number of simulations run per evaluation, a configuration should be re-evaluated with every comparison, or only after it has been undefeated for a number of times. Algorithm 15 shows the pseudo-code for the Simulated Annealing algorithm implementation. Each configuration is simulated 10 times, and is simulated again for re-evaluation if it has survived 10 comparisons with other configurations since its last (re-)evaluation. These numbers were discussed with an expert on simulation at the Utrecht University [11].

Algorithm 15 The Simulated Annealing algorithm implementation.

```
Initialize $temperature = 240$ and $coolingRate = 0.95$
Generate a random Hystrix configuration $currentConfig$
Evaluate $currentConfig$ to get $currentResult$
Set $bestConfig = currentConfig$, and $bestResult = currentResult$
Set $iterationsSinceImprovement = 0$, $iterationsSinceEvaluation = 0$, and $iterationsAtTemp = 0$
while $temperature > 1$ AND $iterationsSinceImprovement < 80$ do
  Determine $parameterToChange = iterationsAtTemp/8$
  Generate $newConfig$ from neighbour function with $parameterToChange$
  Set $iterationsAtTemp = (iterationsAtTemp + 1)\%40$
  if $iterationsSinceEvaluation > 10$ then
    Re-evaluate $currentConfig$
    Set $iterationsSinceEvaluation = 0$
  end if
  Evaluate $newConfig$ to get $newResult$
  if $\frac{currentResult - newResult}{currentResult} > 0.01$ then
    Set $iterationsSinceImprovement = 0$
  else
    Increment iterationsSinceImprovement by 1
  end if
  Determine $acceptanceProbability$ as a function of $newResult$, $currentResult$, and $temperature$
  if $acceptanceProbability >$ random value then
    Accept $newConfig$ as new $currentConfig$, and $newResult$ as $currentResult$
    Set $iterationsSinceEvaluation = 0$
  else
    Increment $iterationsSinceEvaluation$ by 1
  end if
  if $newResult < bestResult$ then
    Accept $newConfig$ as $bestConfig$, and $newResult$ as $bestResult$
  end if
  if $iterationsAtTemp == 0$ then
    Cool down the system by setting $temperature = temperature * coolingRate$
  end if
end while
Write results to file
Output $bestConfig$
```

The Simulated Annealing heuristic determines the cost of each configuration via weights assigned to the different performance metrics. After discussing this with a domain expert at bol.com [10], the chosen cost function is showed in Algorithm 16. It shows that the cost consists of a combination of three performance metrics: the average processing time for requests that fail to process successfully, the false negative time, and the false positive time (see section 5.2

for definitions). The average processing time for errors is only relevant when it is higher than the average processing time for successfully processed requests. In which case, the difference between these two measures weighs 3000 times as heavy as the false negative time and false positive time. This number is based on the decision that an average error processing time of 3 milliseconds is the maximum value allowed. So at this point, this metric should weigh so heavily that optimizing other metrics at the cost of it are no longer worth it.

Algorithm 16 The cost function used by the simulated annealing heuristic.

```
Set $avgErrorCost = 0$.
if Avg. processing time for errors > avg. processing time for success then
    Set $avgErrorCost$ = (avg. processing time for errors - avg. processing time for success) * 3000.
end if
Return $cost = avgErrorCost + falseNegativeTime + falsePositiveTime$.
```

The simulated annealing algorithm has some parameters of its own, namely the initial temperature and the cooling rate. These parameters are usually chosen by some trial and error, but there are some general guidelines that should be followed. The cooling rate is a value with which the temperature gets multiplied each time the system cools down. As such, it should generally lie somewhere between 0.95 and 0.99. For this project a cooling rate of 0.95 was chosen.

The temperature affects the probability that a worse configuration is accepted. It should be high enough that at the start of the simulated annealing algorithm worse configurations have about a 50% chance of being accepted. This depends largely on the order of magnitude of the calculated costs for each configuration, as well as the order of magnitude of the difference between the costs of two configurations. These values can all also vary depending on the luck a configuration has during its evaluation. After experimenting with different settings for the experiments, an initial temperature of 240 is chosen. The temperature only decreases when the neighbour function has chosen different values for all parameters 8 times. With 5 parameters, this means that the temperature decreases after every 40 evaluations.

Finally, the algorithm starts with a random configuration. This configuration is generated by choosing a random value for each parameter in a range that is specified for that parameter. The specified ranges are: 0 to 10,000 milliseconds for the sleep period, 0 to 40 for volume threshold, 0% to 100% for error rate threshold, 1 to 20 for bulkhead size, and 0 to 3,000 milliseconds for timeout delay.

6 Input analysis

Probability distributions are used in order to incorporate some realistic randomness into the simulation. In this chapter an explanation is given why certain distributions are chosen for the different variables. An input analysis has been performed for the following variables:

- Request interarrival time
- Probability of service B processing error
- Service B request processing time

This chapter will describe the analyses performed, their results, and the distributions chosen.

6.1 Request interarrival time

A Poisson process can be used to model the time between two events, if there is a large population that each have a small chance to decide on their own to take an action. In textbooks, a commonly used example for this is a grocery store. In this example a large population of people in the neighbourhood all decide independently when they want to go to the grocery store. This example translates very easily to the situation at bol.com: a large population decides independently to visit the bol.com website. With this in mind, a Poisson process is chosen to model the interarrival times of requests to service A.

In reality, the traffic is affected by the time of day. To reflect this in the simulation, a pattern for the traffic scaling depending on the time of day has been researched. Figure 5 shows such a pattern for the period from 19-3-2018 to 23-3-2018.

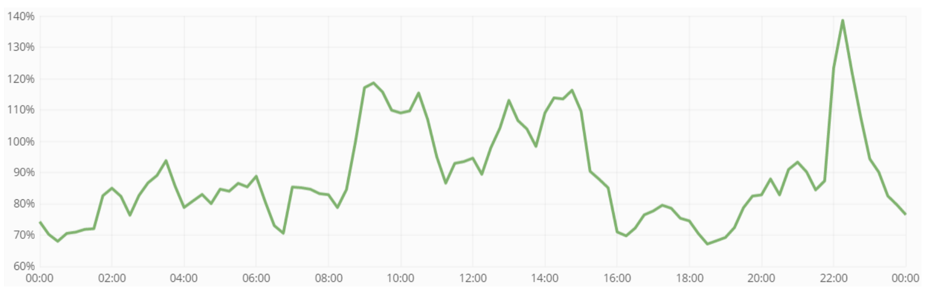

Figure 5: Traffic scaling pattern during the day for the period from 19-3-2018 to 23-3-2018. X-axis: time of day. Y-axis: scaling factor based on average traffic over 7 days.

This pattern is created from the average requests-per-second data in the system logs, specifically the average requests-per-seconds for the *product details* section of the webshop. The I2S service (modelled by service A in the simulation) is a service that is used in this part of the webshop. The system logs at bol.com keep track of average requests-per-second values for every couple of minutes. The following steps were used to create a time of day pattern from this data:

1. Define a method to determine scaling values.
   To create a scaling pattern, scaling values are needed. These values show the ratio of a certain datapoint to a base value. This base value is the average of all datapoints over the last 7 days. The scaling values are then created by expressing each datapoint as a percentage of this base value.

2. Determine scaling values for datapoints.
   By using this scaling calculation, a separate graph is created for each weekday in a specific week (Monday through Friday).

3. Aggregate scaling values to 15 minute intervals.

4. Average aggregated scaling values between weekdays.
   For each 15 minute interval of the day, an average scaling value is computed from the aggregated scaling values of each separate weekday for that interval.

The result of these steps is a time of day traffic scaling pattern for a specific week's weekdays (Monday through Friday), which can be used in the simulation to make the traffic more realistic.

6.2 Probability of service B processing error

The probability that service B fails to correctly process a request is modelled by a binary variable, since processing the request either fails or succeeds. To model this chance, a number is drawn from a uniform distribution and compared to a threshold value.

Fortunately, there are logs at bol.com that keep track of the amount of processing errors that happen when the SLI service is processing requests. This threshold can easily be determined by comparing the number of errors of a service to the total number of incoming requests.

Unfortunately, the logs at bol.com report errors on a lower granularity than desired, namely on a service level instead of endpoint level (A service can perform multiple tasks, an endpoint indicates which task the service is asked to perform). Instead of determining the error probability for the specific endpoint that the simulation models, the average error probability for all endpoints of this service is used. The average error percentage has been determined to be 8% for the SLI service.

6.3 Service B request processing time

This variable models the time it takes service B (the SLI service) to process a request. Originally, this variable was modelled as a gamma distribution with $\alpha = 2$, because this distribution is often used to model a time to complete task stochastic variable. However, after doing some research as to what is usually used in other studies to model the time to complete a computational task, a different distribution was chosen.

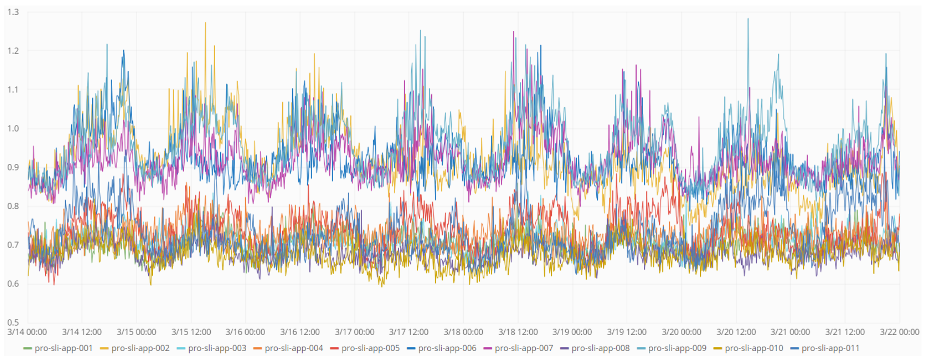

Figure 6: SLI service average request processing time per host, over the period from 14-3-2018 to 21-3-2018. Each colored line represents a separate host. X-axis: time. Y-axis: average request execution time in milliseconds.

Figure 6 shows the average request processing time for each host of the SLI service at a specific point in time (a host is an instance of the service running independently from eachother). The graph shows the data from a period of seven days (from 14-3-2018 to 21-3-2018), the values in the graph range from 0.6 milliseconds to 1.3 milliseconds average processing time per request. These values are reported directly by the SLI service itself and represent the time the SLI service needs on average to process a request from the moment it starts processing a request to the moment it finishes. Time spent waiting in the request queue is not incorporated in this metric.

The graph also shows small fluctuations in the measured values depending on the time of day. During the night, the request processing time is slightly lower than during the day. An explanation for this could be that during the night there is less traffic, which means more resources are available to process requests.

It is important to note that the values in the logs at bol.com are aggregate values. At bol.com the logs are created by StatsD [7], which works by aggregating data over a short period of time (10 seconds by default). This means that instead of all the raw data points, the logs are filled with averages over a short period. This in turn means that fitting a distribution becomes infeasible for this project.

Instead of attempting to fit a distribution with the average values in the logs, research to determine what type of distribution is most often used to model the processing time of computational tasks in other studies was done. Decker and Lesser [5] have done a study that involves software performing computational tasks. They mention that they have used an Exponential distribution to model the duration of the tasks that need to be performed.

Another study, performed by Cabrera et al. [4], finds that processing time data often has a long upper tail. This means that a very small part of all processing times are very high compared to the rest. These values are so high that they increase the average over all the data, despite the fact that there are so few of them. The authors of this study find that the LogNormal distribution fits best with data that has an upper tail. However, for data where the upper tail takes extreme values, an Exponential distribution fits better.

Finally, Kavulya et al.[14] have studied the completion times of computer jobs in a network and conclude that the LogNormal distribution fits best. They mention testing the Weibull and Exponential distributions as well, but unfortunately no results of these tests are shown.

From this research, it seems that the Exponential distribution and the LogNormal distribution are both contenders to model the service B request processing time. After discussing these findings with an expert at the Utrecht University [11], the Exponential distribution is chosen to model the service B execution time.

In order to sample from the Exponential distribution, an average needs to be calculated. As is visible in Figure 6, there is a division between two groups of hosts, where one group of hosts consistently needs more time to process a request than the other group. There are numerous possible explanations for this divide. The hosts could be running on a weaker machine, or on a machine that also runs other services that are very busy.

As a result of this division, two averages have been calculated: one for the group of slower hosts (consisting of hosts *pro-sli-app-002*, *pro-sli-app-006*, *pro-sli-app-007*, and *pro-sli-app-009*), and one for the faster group of hosts (consisting of hosts *pro-sli-app-001*, *pro-sli-app-003*, *pro-sli-app-004*, *pro-sli-app-005*, *pro-sli-app-008*, *pro-sli-app-010*, and *pro-sli-app-011*). Experimenting with these different averages can show what difference they make. The averages are calculated over all data points of a group of hosts over 7 days. For the period of 13-5-2018 to 19-5-2018, the calculated average request processing times are 0.6303 milliseconds for the faster group of host, and 0.8252 milliseconds for the slower group of hosts.

Note that in this analysis no difference is made between processing time of requests that fail or succeed. This is because, in the system logs at bol.com, the distinction between processing time of successful and failed requests is not made. The data for failed and successful request processing times are collected into the same metric, without any means of distinguishing them.

7 Experiments

This chapter gives an overview of the experiment setup and the results gathered. It is divided into sections for each iteration of the project. Each section about an iteration will go into detail about what has changed since the previous iteration, the current experiment setup, and the experiment results.

The input parameters (specified in the upcoming subsections) are based on the Inspire-to-Sell service (I2S, modelled by service A) at bol.com and its connection to the Sellingoffer-Integration-Service (SLI, modelled by service B). The I2S service is a relatively small service, with a number of connections to other services wrapped in Hystrix, one of which is a connection to the SLI service.

7.1 First iteration: Setup

This section describes the experiment setup and results for the first iteration of the project. This iteration focused on getting the simulation up and running.

The simulation is designed to automatically test different Hystrix configurations against the specified input parameters a variable number of times. This variable, specifying the amount of times each test case needs to be performed, is given as a command line argument to the simulation.

In addition to this, the simulation takes a number of additional command line arguments, specifying where to find the different Hystrix configurations, the input parameters, and where to write the output of each test case. The following subsections will describe what input parameters and Hystrix configurations were used for the experiments.

7.1.1 Experiment description

This subsection mentions the chosen input parameters and Hystrix configuration, and explain why these values were chosen.

Input parameters In order to sample from the chosen distributions, their means need to be determined. Note that some of these chosen parameters are not ideal, but have been chosen with an iterative improvement process in mind (as in, they can and will be improved upon in a later iteration of the project). Note that the values listed here are the values used in the first iteration of the project. Later on, some of these values have changed as a result of the input analysis. As a result, the values listed here may seem inconsistent with earlier chapters.

- Request interarrival time:
  The amount of traffic is based on the performance tests used internally at bol.com. The I2S service at bol.com has performance load tests that send 165 requests per second for the duration of the test.

- Probability of service B processing error:
  From the logs at bol.com, the error probability of the SLI service is determined to lie at 8%.

- Service B request processing time:
  The logs at bol.com do report execution time on an endpoint level, but unfortunately do not make a difference between executions that are successful and those that fail. For now, the execution time of an endpoint is used for the process time of successful and failed requests alike. The average for this parameter is 10 milliseconds.

- Probability of network failure:
  The assumption that the network works flawlessly (see section 5.1) means that the probability of network failure is 0.

- Network latency:
  It is also assumed that the network works instantly (see section 5.1). Therefore, the network latency is also set to 0.

- HTTP driver time to error:
  The time it takes the HTTP driver to determine something went wrong and send back an error is set to the default value for the ecosystem at bol.com, which is 1000 milliseconds according to a domain expert at bol.com [10].

Hystrix configuration The I2S service uses the default Hystrix configuration, which has the following values (as specified in chapter 4):

- Sleep period:
  When Hystrix determines a circuit is unhealthy, it blocks all traffic for 5000 milliseconds before checking to see if it has recovered.

- Volume threshold:
  Hystrix considers the service health as a valid metric only when a minimum of 20 requests have been made in the last 10 seconds.

- Error rate threshold:
  A circuit is deemed unhealthy when the error rate exceeds 50%.

- Bulkhead size:
  Hystrix allows 10 concurrent requests to service B.

- Timeout delay:
  Hystrix interrupts the request after 1000 milliseconds have passed.

For now, this is the only Hystrix configuration that is used to run the experiments. In future iterations, different configurations can be used and their results can be compared.

Iteration	1	2	3 (Fast)	3 (Slow)	4
Success	13,115,550	8,367,271	15,431,571	15,431,042	15,275,281
Circuitbreaker	0	192	0	0	167,866
Bulkhead	0	5,160,852	0	0	0
Hystrix Timeout	0	0	0	0	0
HTTP Error	0	0	0	0	1,004
Service B Error	1,140,548	727,586	1,341,789	1,341,734	1,328,258
Avg. processing time	9.50 ms	55.90 ms	0.64 ms	0.85 ms	0.84 ms
Error rate	8.00%	41.31%	8.00%	8.00%	8.93%
False Negative Time	-	-	-	-	4,395 ms
False Positive Time	-	-	-	-	2,562 ms
Service B Downtime	-	-	-	-	900,000 ms

Table 2: Results of the experiments run during iterations 1 through 4.

7.1.2 Experiment results

Because in this first iteration of the project only a single Hystrix configuration was used, a comparison cannot be made. However, the resulting measures can be used to determine if the simulation behaves as expected. In Table 2 the averages of all performance metrics over 30 simulations are shown for each iteration. The first six rows (not counting the header) show the amount of processed requests, divided into categories corresponding to their results. The number of requests that were processed successfully can be found in the *Success* row. The amount of requests that were rejected by the Hystrix circuitbreaker or bulkhead is found in the *Circuitbreaker* and *Bulkhead* rows respectively. The number of requests that took so long that Hystrix decided to interrupt them is shown in the *Hystrix Timeout* row. The *HTTP Error* row shows how many requests were unable to reach service B. In the *Service B error* row, the number of requests that resulted in an error during processing by service B is shown. The next two rows (*Avg. processing time* and *Error %*) show the average processing time of all requests and percentage of requests that resulted in an error respectively. The last three rows are only used in the fourth iteration, as they denote the new performance metrics implemented in that iteration.

What the results of the first iteration show is that the simulation does indeed behave as expected. A total number of 14,256,098 requests passed through the system, which is very close to the expected amount for these input parameters (165 requests per second * (24 hours * 3600 seconds per hour) = 14,256,000 requests).

The fact that the only error that occurs is the Service B error can also be explained by the input parameters:

- The bulkhead starts rejecting requests when service A wants to send more than 10 concurrent requests to service B. However, due to the short processing time of service B (10ms), the bulkhead will likely never even have more than 2 concurrent requests in it (165 requests per second * 0.01 second per request = 1.65 concurrent requests).

- The HTTP error never occurs, because the input configuration has specified a flawless network, where no requests ever get lost and there is no latency.
- This in turn indicates why the timeout error also never occurs. All possible paths a request can take return their result (success or failure) in around 10 milliseconds. The timeout error only occurs when a request takes 1000 milliseconds to return a response.
- The service errors that do occur make up about 8% of all requests, which corresponds to how the input parameters are configured.
- Finally, the circuitbreaker only opens when the error rate of the last 10 seconds exceeds 50%. The chance of this happening is very low, because the only error that actually occurs every now and then is the service error. The 8% chance of a service error gets nowhere near the 50% errors needed to open the circuitbreaker.

This indicates that the model behaves as expected for these input parameters.

7.2 Second iteration

In this iteration the request queue logic for service B was implemented and underlying Hystrix logic was improved. The following subsections detail the experiments run in this iteration and their results.

7.2.1 Experiment description

No changes to the input parameters were made. The same parameters as those in the first iteration were used, and can be found in subsection 7.1.1. As such, only the experiment results will be described in this section.

7.2.2 Experiment results

The Hystrix circuitbreaker logic was a smaller change than the improvement on the request queue logic, as the queue logic actually changed the simulation model and the flow of requests through the system. Table 2 shows the averages of 30 simulations run with the aforementioned input parameters and Hystrix configuration on the updated simulation model in the column of iteration 2.

The major changes in these results are the large amount of bulkhead rejections that occur and the increased average processing time. Both of these changes can be explained by the new request queue logic. Before these changes, service B handled all incoming requests concurrently. However, with the new queueing logic, service B handles only one request at a time and the rest are inserted into a queue and wait there until they are processed. Naturally, this results in an increased processing time, which in turn causes the bulkhead to fill up and start rejecting requests.

Some of the smaller differences between the results of these first two iterations are:

Slightly reduced service B error count This can be explained by the fact that less requests are reaching service B in the second iteration. Of all the requests reaching service B, about 8% fail, which is in line with the input parameters.

Small amount of circuitbreaker errors This is an average over all 30 simulations. In the results of a single simulation there are either 0 or between 800 and 840 circuitbreaker errors. This number of around 800 errors can be explained. When the circuitbreaker opens, it will stay open for 5 seconds. During these 5 seconds about 825 requests will arrive (165 requests per second), which immediately get rejected. This means that in some of the simulation loops, the circuitbreaker opens once. The average error percentage of all simulations lies around 41%, which is close enough to the threshold of 50% that a small stroke of bad luck can push it over, opening the circuitbreaker as a result.

7.3 Third iteration

The focus of the third iteration was to improve the input parameters. An input analysis has been performed for the request interarrival time and the service B processing time. Chapter 6 describes the analyses performed and their outcomes. New values for the request interarrival time and the service B process time were determined as part of the improvement process.

7.3.1 Experiment description

In this subsection, the changes made to the input parameters are explained. Parameters that are not mentioned in this subsection are unchanged from previous iterations.

Request interarrival time In section 6.1 it is mentioned that the traffic scaling pattern is based on a base value from which the pattern can scale. In the same section, it also mentioned that the chosen base value is an average over 7 days. So for these experiments, the average requests per second over the 7 days was used, which is 200 requests per second.

Service B request processing time In this iteration a different distribution was chosen for this parameter (see section 6.3), namely an exponential distribution. In section 6.3 two averages are determined. An average of 0.6303 milliseconds is used for the faster group of hosts. The slower group of hosts uses an average of 0.8252 milliseconds. Both scenarios have been experimented with.

Other than these parameters, no other changes were made to the input parameters.

7.3.2 Experiment results

Table 2 shows the results of the experiments with these new input parameters in the columns *3 (Fast)* and *3 (Slow)*.

The differences between the two scenarios are visible in the Success, Service B Error, and Avg. processing time columns. The biggest difference can be found in the Avg. processing time column, while the differences in the other columns are very small. Because the differences are so small, a two-sample t-Test assuming unequal variances is performed to confirm which difference are significant. The numbers in the Success and Service B Error columns are not statistically significantly different (t Stat 1.234 and 0.191 respectively). The numbers in the Avg. processing time column do have a significant difference (t Stat 3808.872). This shows that the difference between the fast and slow scenario only affects the average processing time. As such, in future experiments there is no need to test both scenarios. Instead, the slow scenario will be used in future experiments, since it is the worst-case scenario.

Comparing the results to those of the previous iteration shows a number of differences:

No more bulkhead errors In the second iteration, the results showed that the Hystrix bulkhead was frequently rejecting requests due to being full. Because the service B request processing time was changed (from 10 milliseconds per request to 0.7928 milliseconds per request on average, as per the input analysis in section 6.3) the bulkhead no longer fills up, which in turn means that no requests get rejected by it. During the input analysis it became apparent that the values used previously were inaccurate.

No more circuitbreaker errors Because the bulkhead no longer fills up and rejects incoming requests, the error percentage during simulation no longer reaches the threshold required to trip the circuitbreaker.

Lower error rate Fewer errors occur, which means the average error rate is lower.

Lower average processing time The changed input parameters for the service B request processing time means that requests are handled so quickly that the request queue for service B never gets a chance to fill up. This in turn means that requests are queued for a much smaller amount of time (if at all). Not having to wait in the request queue combined with the lower service B request processing times results in a much lower average processing time.

More requests passed through the system As a result of the traffic input analysis a time of day pattern and new request interarrival time parameters have been used in this experiment. This is reflected in the amount of requests that have passed through the system.

What these results show is that with the current input parameters, the services have no problems handling the traffic. Hystrix is only performing some graceful error handling in the form of fallback responses when service B returns an error.

7.4 Fourth iteration

The main focus of the fourth iteration was to expand the performance metrics and the simulation model.

The false positive time, false negative time, and service B downtime metrics were introduced as new performance metrics to evaluate the configuration used. These metrics are designed with the goal to minimize the time Hystrix needs to react to failing services in mind.

In order to better model a service failing, the *Service B breakdown* and *Service B recovery* events were introduced to the model. These events model an outside force breaking and restoring service B.

Finally, in this iteration the project was also restructured a bit so that simulations can be run in parallel.

7.4.1 Experiment description

With the introduction of the service B breakdown and recovery events, a scenario for this breakdown and recovery event needs to be defined. The scenario mimics a developer accidentally deploying bugged code, thereby breaking the service. This happens 2 hours into the day, and takes the developer 15 minutes to fix.

Other input parameters are unchanged from previous iterations.

7.4.2 Experiment results

Table 2 shows the results of the experiments run in this iteration for the various performance metrics in the column for iteration 4. This table also shows the performance metrics that have been used since the first iteration of the simulation. The results of the new performance metrics added in this iteration are shown in the last 3 rows of this table.

The values in Table 2 show that the simulation works as expected from the simulation model. Compared to previous iterations there are a few differences:

Circuitbreaker errors The Hystrix circuitbreaker actually gets some work once service B breaks down. This results in a number of circuitbreaker errors.

HTTP errors When service B breaks down, it will return a HTTP error on all incoming requests. Hystrix is quick to notice this and stops service A from sending requests to service B. This explains why there are so few HTTP errors.

Slightly higher error percentage A broken down service B results in errors, which is reflected in the slightly higher error percentage.

One thing that stands out from these results is that there are still no Hystrix timeout errors. According to the simulation model, when service B breaks down it should drop all requests that it is currently processing and that are in the request queue for service B. However, it turns out that due to the low processing time required per request, service B never has any requests waiting in the request queue, and is idle for most of the time. In short, there are no requests that can be dropped when service B breaks down, which explains why there are no Hystrix timeout errors.

The results for the new performance metrics in Table 2 show that Hystrix needs, on average, about 4.4 seconds to detect a service is failing. And that Hystrix notices, again on average, after about 2.6 seconds that a service has recovered. The service B downtime is a constant 900,000 milliseconds, which corresponds to the scenario detailed in subsection 7.4.1.

The fact that Hystrix needs 4.4 seconds to detect a failing service is to be expected when looking at the input parameters. Hystrix tracks the health of a circuit over the last 10 seconds, so by the time enough errors have passed through the system to exceed the threshold, about 5 seconds will have passed. Because service B already returns some errors before breaking down (the error probability is set at 8%), this means that it takes a little less than 5 seconds for the error rate to exceed the threshold.

The false positive time is an average over the results of 30 simulation loops. Table 3 shows the raw data. As can be seen, the actual values lie quite close to multiples of $5,000$, which is the number of milliseconds Hystrix is configured to sleep before trying a service again. This is because the service breakdown and service recovery events happen at the same time for every simulation. The small fluctuations are explained by the stochastic way the request interarrival time is determined. The larger fluctuations can be explained by the configured Hystrix sleep time of $5,000$ milliseconds. When Hystrix checks if a service has recovered and the check fails, Hystrix will automatically sleep $5,000$ milliseconds.

False Positive Time		
74.18	72.03	4,904.95
208.52	4,995.90	48.93
77.20	280.55	4,889.54
4,883.57	14.18	349.40
5,113.38	10.72	4,998.09
4,973.28	9,979.61	93.14
294.76	4,877.92	8.22
4,982.83	29.12	4,901.06
9,902.16	140.15	232.10
5,280.41	119.03	148.12

Table 3: Raw data of the False Positive time metric.

Finally, the new structure that allows the simulations to be executed in parallel decreases the time needed to run the 30 simulations. Before the simulation were running concurrently, each simulation took about 7 to 8 minutes, which resulted in a runtime of almost 4 hours for 30 simulations. With the current parallel specifications, the simulations take about 10 minutes each, but they run 3 at a time. This results in a total runtime of about 1 hour and 40 minutes for 30 simulations.

7.5 Fifth iteration: Optimization

The fifth iteration is the final iteration of this project. With the simulation model finished, the input analysis performed, and the performance metrics expanded, the simulation is ready to be used to evaluate and compare different Hystrix configurations. The goal is to find an optimal configuration for the specified input parameters. To this end, the Simulated Annealing heuristic described in section 5.6 was implemented.

When the simulated annealing heuristic was implemented, it became apparent that the evaluation of a configuration was taking far too long. In earlier experiments, a configuration was simulated 30 times. However, the Simulated Annealing heuristic simulates more than just one configuration. Each change to the configuration made by the neighbour function requires the new configuration to be simulated. Despite simulating the configurations only 10 times instead of 30, the Simulated Annealing heuristic was still taking far too long.

In addition to this, simulating a long period of time in which the system is healthy with only a few, relatively short, breakdowns has the effect of skewing the performance metrics towards the healthy status of the system. This makes it harder to determine what effect different configurations have on the handling of these breakdown moments.

In order to tackle both these issues, the length of each simulation was shortened. Naturally, shorter simulations result in a shorter runtime. But this also has the added benefit that the effects that different configurations have on how well Hystrix handles different breakdown scenarios are more visible in the performance metrics.

7.5.1 Experiment description

The experiments in this iteration consist of running the simulated annealing heuristic, which uses the simulation as a sub-routine to evaluate different configurations.

In order to reduce the runtime of the algorithm, the simulation length and amount of simulations per configuration were reduced. These experiments were run with 10 simulations per configuration evaluation, and each simulation simulating 5 minutes. Since this means that the entire day is no longer being simulated, the simulation start time is also moved from 00:00 to 22:00. This means that the simulation now simulates a time period from 22:00-22:05, which is an interesting time period for bol.com, since it coincides with the evening traffic peak.

During this period two breakdowns are scheduled. The first breakdown happens at 22:00:15, lasts for 45 seconds, and slows down all request processing by 500 milliseconds. The second breakdown happens at 22:01:30, lasts for 90 seconds, and slows down all request processing by 2500 milliseconds. This leaves the simulation 15 seconds to warm up before the first breakdown, 30 seconds between the two breakdowns to recover normal operation, and 2 minutes after the last breakdown to recover normal operation until the simulation is terminated.

In total, the entire Simulated Annealing was run 5 times, each time starting with a different (random) start configuration. All other parameters are equal between runs.

7.5.2 Experiment results

The results of the experiment are shown in the following figures. Each Hystrix configuration parameter (sleep period, volume threshold, error rate threshold, bulkhead size, and timeout delay) has its own graph. In these graphs, each marker represents an evaluation of a certain Hystrix configuration. The Y-axis represents the cost of a configuration, calculated by the cost function defined in Algorithm 16. The X-axis shows the different values that the parameter takes in the evaluations.

Note that markers in a vertical line indicate different evaluations with the same variable for the specified parameter. Differing costs for the same parameter value can be explained by either the randomness of the simulation, or the fact that other parameters of the configuration may have changed. As such, looking at averages or trends is not as useful with results like this.

Instead, the minimal cost values for specific parameter values are more interesting to look at. The minimal cost values attainable for different values of the configuration parameters can give an indication in how these parameters limit how well the configuration can possibly perform. Since the algorithm will automatically accept configurations that perform better, values resulting in these better configurations will get evaluated more often. This explains why the density around the found optimal values is higher than elsewhere in the graph.

Figures 7 through 11 show the results of the first run of the Simulated Annealing algorithm. Due to the randomness in choosing the start configuration, as well as the randomness in the simulation process, results from other runs might look different from these results. As such, the best configuration found might differ as well between runs.

These figures can show how the Simulated Annealing algorithm reached the best found solution. From this we can determine that the Simulated Annealing algorithm works as expected, since the figures show that a range of values for each parameter is tested, and that values that result in lower costs are evaluated more often.

Figure 7 shows the evaluation results of different values for the sleep period parameter. Looking at the minimal cost values for each sleep period value, the value resulting the lowest cost is found in a valley at a sleep period parameter value of 3,306 milliseconds, surrounded by numerous other local optima.

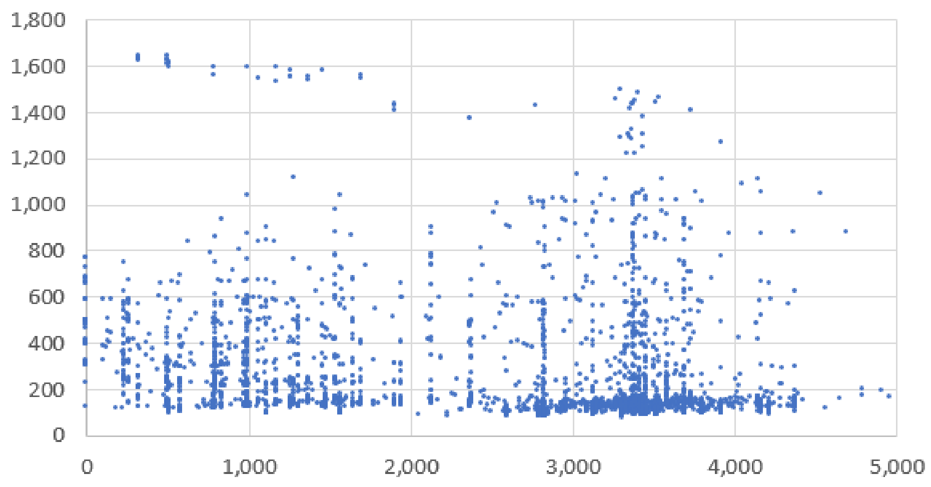

Figure 7: Cost plotted against Hystrix sleep period values. X-axis: sleep period in ms. Y-axis: cost.

In Figure 8 the costs are plotted against different volume threshold values. The minimal costs evaluated for the different values seems quite steady, up to a threshold value of 8. Values higher than 8 seem to perform (slightly) worse. It is unclear if this threshold value of 8, above which costs rise, is specific to this run of the experiments.

The next figure (Figure 9) shows the costs plotted against different error rate thresholds. A more noticeable increase in minimal costs with increasing threshold values is visible here. Exceedingly low values result in higher costs as well, indicating that lowering this value beyond a certain point is not beneficial. In this run, the lowest cost is found at an error rate threshold of 40.19%.

Figure 10 shows the costs plotted against different bulkhead size values. For values lower than 10, the minimal costs are all more or less similar. After this the minimal costs appear to rise and higher values are evaluated less often. At a bulkhead size of 3, the lowest cost is found in this run.

Finally, in Figure 11 the costs are plotted against the different timeout delay values. Again a number of different local optima are found. The best cost is found at a timeout delay of 609 milliseconds.

These graphs do indeed show that the Simulated Annealing algorithm is escaping local optima and evaluating promising configuration parameter values more often. The algorithm works as expected.

Table 4 shows the best configurations found during the different runs of the Simulated Annealing algorithm. Since each run starts with a different start configuration, the range of values evaluated for each parameter can differ between runs as well. This can explain why the best configurations found during the different runs differ from each other. In the fourth run, the configuration that scores the lowest cost overall has been found. Conclusions drawn from these

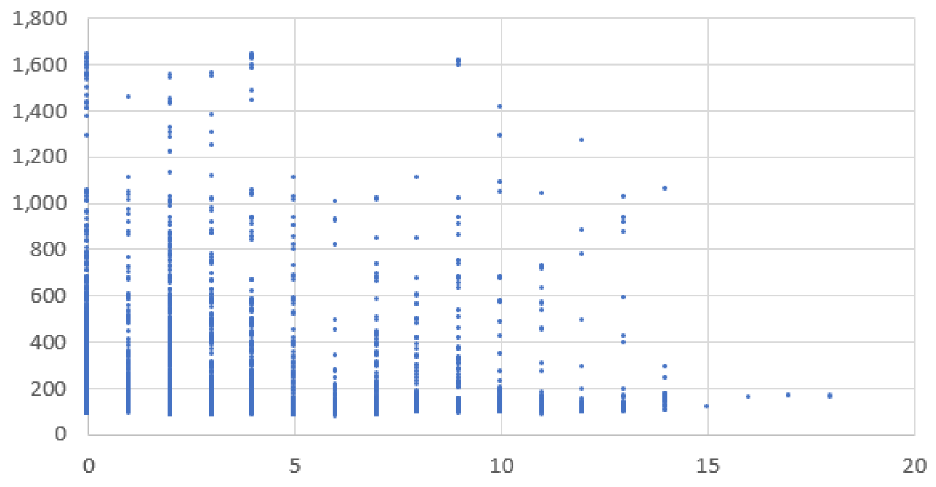

Figure 8: Cost plotted against Hystrix volume threshold values. X-axis: volume threshold value. Y-axis: cost.

results will be discussed in chapter 8.

Run	Sleep Period	Volume Threshold	Error rate Threshold	Bulkhead Size	Timeout Delay	Cost
1	3,306 ms	6	40.19%	3	609 ms	76.62
2	4,027 ms	11	21.38%	2	2,948 ms	45.88
3	2,876 ms	26	25.65%	1	1,335 ms	48.23
4	3,170 ms	24	18.73%	8	441 ms	32.54
5	257 ms	0	18.44%	1	4,371 ms	57.75

Table 4: Configuration parameter values and costs of the best configurations found during each run of the Simulated Annealing algorithm.

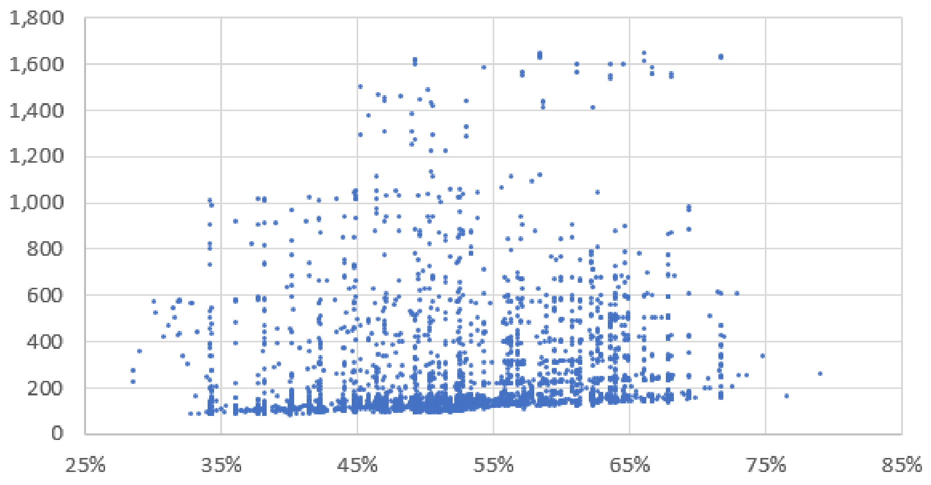

Figure 9: Cost plotted against Hystrix error rate threshold values. X-axis: error rate threshold. Y-axis: cost.

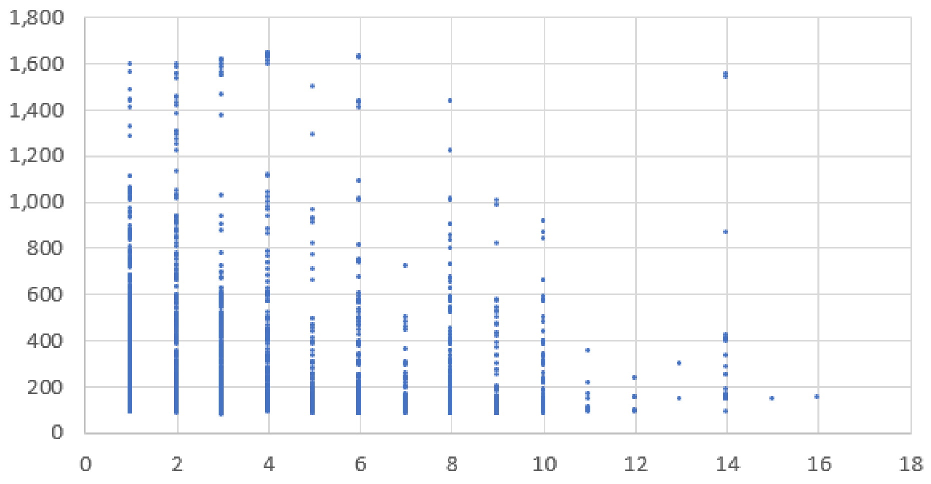

Figure 10: Cost plotted against Hystrix bulkhead size values. X-axis: bulkhead size. Y-axis: cost.

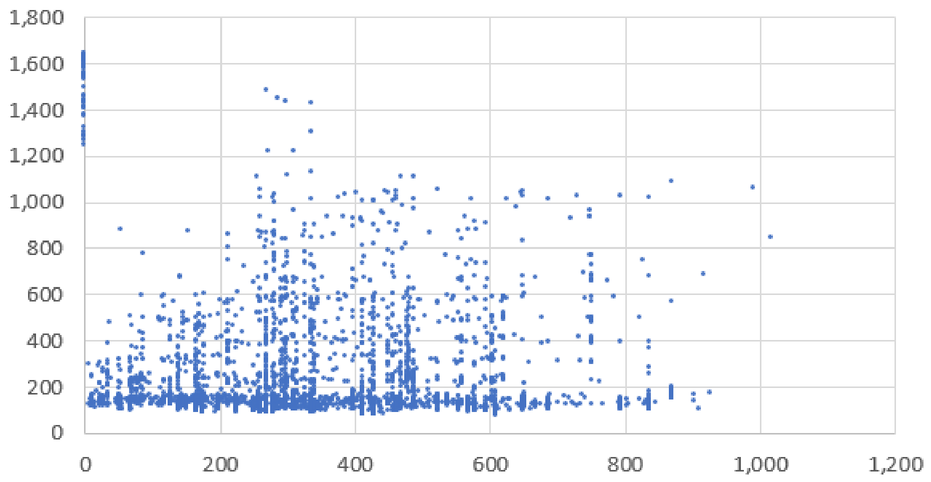

Figure 11: Cost plotted against Hystrix timeout delay values. X-axis: timeout delay in ms. Y-axis: cost.

8 Conclusion

In this chapter, the conclusions drawn from the experiments performed in each iteration are described. Similarly to chapter 7, this chapter is split into sections for each iteration of the project. From the conclusions reached in these sections, answers to the research questions are drawn. Finally, an advice for bol.com is given.

8.1 Iterations

Conclusions from the results of each iteration will be discussed here.

8.1.1 First iteration: Setup

The results of the first iteration (see section 7.1) allow us to conclude that the simulation works as expected from the simulation model and input parameters, as far as the successful and service error paths go. For these configurations, the other paths (of other errors) also function as expected. However, to ensure that all paths do indeed function as expected (under different input configurations as well), some other configurations, specifically designed to test this functionality should be run and their outputs analyzed.

Analyzing the results of this first iteration has shown that there is still work to be done on improving the simulation.

8.1.2 Second iteration

The findings of the second iteration reveal that the improvements made in this iteration perform as expected. However, it does highlight an issue with the service B processing time input parameters. The input parameter is based on the execution time of the entire Hystrix command in the real system, instead of the actual processing time the SLI service (service B) needs for a single request. This includes time spent waiting in the request queue. Because this total value is used to determine the processing time (excluding time spent waiting in the queue) in the simulation, the total processing time of each request in the simulation is higher than in the real system.

8.1.3 Third iteration

The outcomes of the experiments in the third iteration display that the implementation of the chosen input parameters operates as expected. The difference between the fast group of hosts and the slow group of hosts only affects the average processing time of requests. In future iterations the slow group of hosts should be used to model the service B execution time, as it is the worst-case scenario.

It also shows that with these input parameters, the services have no problem handling the traffic directed to them. Hystrix seems to not be doing much either, only handling some errors from service B now and then. It would be interesting to see what happens when service B breaks down. In order to perform such experiments, improvements to the simulation model and the performance measures should be made.

8.1.4 Fourth iteration

The fourth iteration's results show that the implementation of the simulation model expansion and the new performance metrics work as expected. It is interesting to see how fast Hystrix reacts to service failure and recovery. The model improvements allow the testing of scenarios that are the most common use-case for Hystrix at bol.com (service breakdowns). And with the new performance metrics an actual comparison between different Hystrix configurations can be made.

8.1.5 Fifth iteration: Optimization

Looking at the results of the fifth iteration, they indicate that the simulated annealing algorithm works as expected. The graphs in subsection 7.5.2 show that a range of configurations has been evaluated. Parameter values with lower costs have been evaluated more often, which shows the algorithm attempting to find an optimal configuration near parameter values that perform better. The results also show outliers being tested, but usually being discarded quite quickly due to high costs.

From the results, conclusions about how the configuration parameters affect the effectiveness of Hystrix can be drawn. Looking at the minimal cost for each value of the different parameters, we conclude that:

Sleep period The sleep period has numerous local optima. Over all 5 runs, the run with a sleep period of 3,170 milliseconds scored the lowest cost. This suggests that the optimal sleep period value for this Hystrix circuit can be found around this value.

Volume threshold Looking at Figure 8, the minimum cost stays mostly the same over different values of this parameter. In Table 4 the volume threshold value of the best found configuration varies wildly. This suggests that the value of this parameter is largely inconsequential to the effectiveness of Hystrix.

Error rate threshold The results in Figure 9 and Table 4 show an increase in cost with increasing threshold values, indicating that smaller threshold values result in a more effective Hystrix circuit. There also seems to a limit to how small this value can become before the costs start to rise again. From the results in Table 4, the best value for this parameter is 18.73%.

Bulkhead size Figure 10 shows that this parameter has very little effect the minimum cost of a configuration. Results in Table 4 seem to favor smaller values, however the best result is achieved with a bulkhead size of 8.

Timeout delay Again, a number of local optima are found during the experiments. This is reflected in the results shown in Table 4, where the Timeout delay values for the best configurations found also vary quite a bit.

From this, we can conclude that the Volume threshold and Bulkhead size parameters are of little interest, and could be largely ignored when tuning the Hystrix circuit. The other configuration parameters (Sleep period, Error rate threshold, and Timeout delay) all do seem to affect the cost of a configuration. The results show that lower values for the Error rate threshold are beneficial, up to a point. The Sleep period and Timeout delay parameters both have numerous local optima, and thus are most difficult to tune optimally. Further experimentation could help provide additional insight into how these parameters affect the effectiveness of resilience framework configurations.

It should be noted that the results of this project, and by extension the conclusions reached from these results, are specific to the services simulated in the experiments, namely the I2s and SLI services. Future work could run more experiments with different setups to determine how Hystrix configurations influence the results in different scenarios.

8.2 Research question

In order to provide a meaningful answer to the main research question, the sub-questions will be answered first. After this, the main research question will be answered with the answers to the sub-questions in mind.

How can a microservice software ecosystem that uses a resilience framework be modelled as a discrete event simulation model? The model proposed in this project uses the state to model the connected services and the resilience framework used in this connection between the services. Events are used to model requests and responses being sent to and from these services, as well as outside forces influencing the simulated system (such as service breakdowns). Realistic randomness is introduced to the simulation by using probability distributions for stochastic variables (such as request interarrival time). A number of assumptions and abstractions are made to scope the model.

How can different resilience configurations be evaluated and compared? Performance metrics that measure how fast the resilience framework is able to respond to changes in the system are proposed in this work. This response time is measured by two metrics: the false negative time and the false positive time. The false negative time measures the time during which the framework incorrectly determines that the connection is healthy, and thus takes no action. The false positive time measures the time during which the framework incorrectly determines the connection is unhealthy, and thus takes unnecessary action.

In collaboration with domain experts at bol.com [10], a cost function was designed to evaluate the results of a simulation to a single number, which can then be compared to other evaluations. This cost function uses the false negative time and false positive time performance measures in addition to the average response time for failed and successful responses. Priority can be given to more important metrics by weighing their values against each other.

What different resilience configurations are interesting to compare? Instead of attempting to determine a set of interesting configurations (and thus limiting the project to those configurations), an optimization algorithm was implemented that is able to create new configurations as it sees fit. The Simulated Annealing algorithm is able to create "neighbour"-configurations, which differ slightly from their original. Uninteresting configurations do get created and evaluated, but get rejected if they do not perform well enough. Some configuration parameters are of less importance to the effectiveness of the resilience framework (namely, the volume threshold and bulkhead size parameters). Removing these parameters from the set of configuration parameters that the optimization algorithm is allowed to change could reduce the runtime of the algorithm, or allow the algorithm to look at more values for the other (relevant) parameters.

Do the results of the experiments offer valuable insights into tuning resilience framework configurations? The results show that there are configuration parameters that seemingly do not affect the effectiveness of resilience framework much. These parameters (the Volume threshold and Bulkhead size) can be ignored when tuning resilience framework configurations.

The other parameters (Sleep period, Error rate threshold, and Timeout delay) do appear to have an impact on the effectiveness of the resilience framework. The Error rate threshold seemingly allows for a lower cost of the configuration at lower values. When tuning this parameter, attempting to tune it as low as possible seems beneficial. However, this threshold value has a threshold of its own, beyond which the costs seem to rise again. Tuning should be done careful not to cross this threshold value.

The Sleep period and Timeout delay parameters seem to have multiple locally optimal values. Further experimentation might provide additional insight into the effect these parameters have on the effectiveness of resilience frameworks.

With the answers to these sub-questions in mind, the main research question will be answered. The main research question is as follows:

"How can discrete event simulation be used to support the decision-making process of tuning resilience framework configurations in a microservice software ecosystem?"

By modelling the microservice software ecosystem in a discrete event simulation model, it can be used to help identify effects of resilience configuration parameters on the effectiveness of the resilience framework. It can help identify the kind of relations that exist between parameter values and effectiveness of the resilience framework.

By combining the simulation with an optimization algorithm and adapting the input parameters to specific cases, it can be used to help tune specific Hystrix circuits by finding bounds for parameters within which the resilience framework operates most effectively. This will return an optimal configuration given a specific case. However, due to the nature of optimization, this configuration is highly specific to the case it is optimized for, and thus should be used more as a guideline when tuning the configurations of the real system.

8.3 Advice to bol.com

The results show that the Volume threshold and Bulkhead size parameters are largely inconsequential to the effectiveness of a Hystrix circuit. As such, it is recommended not to spend too much time on tuning these parameters.

Lower values for the Error rate threshold result in a more effective Hystrix circuit, however there is a limit to this. Values lower than a certain point result in a less effective Hystrix circuit. During tuning, this parameter should be tuned to lower values, while taking care not to choose a value that is too low such that it would reduce the effectiveness of the Hystrix circuit. Further experimentation could indicate where the optimal value for this parameter lies for different Hystrix circuits.

The Sleep period and Timeout delay parameters have numerous locally optimal values, which makes tuning these parameters a lot harder. Additional experiments could show where the optimal values for these parameters lie for different services and scenarios.

In order to use the work done in this project, it would be nice to create an interface that allows engineers to alter the experiment setup to their specific situation. In addition to this, attempting to find a way to reduce the runtime of the Simulated Annealing algorithm could greatly increase the practicality of this work (for example by running it on a more powerful machine, or in the cloud).

9 Discussion

This study shows that a combination of a simulation model and an optimization algorithm can be used to help tune resilience framework configurations in microservice software ecosystems. Valuable insights such as relations between configuration parameters and the effectiveness of the resilience framework are found.

The caveat here is that the optimal configurations found by this combination of techniques should be used as a guideline, and not an absolute truth. The reason for this is that the optimization algorithm optimizes against very specific circumstances. The optimization algorithm optimizes to the specific input parameters; cost function (which evaluates the simulation results), breakdown scenario (specifies the breakdowns that occur during simulation), traffic (determines request interarrival times), and simulated services (defining service properties such as error probability). Changes in any of these settings can affect the results of the optimization algorithm. Iterating over these settings and running more experiments with different settings might offer additional valuable insights. Running additional experiments with different experiment settings could also provide more insight in the relation of the sleep period and timeout delay parameters to the cost.

Furthermore, a simulation is always an abstraction of the real world system. This abstraction needs to walk a fine line between pragmatically abstracting unimportant details and taking care not to abstract too much details such that the system becomes unrealistic. On top of this, in some cases desired data was not available in the system logs at bol.com. In these cases, the desired data was substituted by data that came close.

Finally, there is an issue discovered in the event handlers. Some of these handlers (namely the *Request received A*, *Request processing start B*, and *Request processed B* event handlers) seem to look into the future. The results of these predictions are used to determine which event should be scheduled next, at which time it should be scheduled, and to make some changes to the State. The State changes are purely administrative changes that help reduce the memory required to run the simulation. Because of this, the effect this has on the simulation results is estimated to be very small. However, looking into the future is not a good practice in simulation and should be avoided.

9.1 Future work

A number of possibilities for future work have been identified at the end of this project. Mentioned earlier, running additional experiments with different experiment settings can provide more insight in how the resilience framework reacts to different circumstances. Interesting changes that can be made to experiment settings include: peak season traffic, simulating different services, alternative breakdown scenarios, and redefining the cost function.

Furthermore, the simulation model could be improved upon by simulating traffic of other services (not service A) to service B. This would be a less abstract and more realistic simulation of how service B becomes latent due to increased traffic from other services.

Another way to improve the simulation model is to add additional services that service A depends on. When one of the connected services breaks down, the bulkhead should prevent requests to that broken service from blocking requests to other, healthier services. This improvement to the simulation could give additional insights in the bulkhead size configuration parameter and the effects it has on the effectiveness of resilience frameworks.

Next, in an effort to adhere to the best practices of simulation, the event handlers that look into the future to schedule new events should be changed to not do so. Looking into the future is not a good practice in simulation and should be avoided. Despite this, the effects of changing this are expected to be negligible.

When taking another look at the simulation code, there is a specific issue that should be investigated further. When the *Request start processing B* event handler schedules a new *Request processed B* event, it retrieves the required time to process the request from the input parameters. The input parameters in turn sample this value from a normal distribution. In some very rare cases, the sample method of this distribution returns a *NaN* (Not a Number) value. This is then returned to the event handler, who attempts to schedule an event at time *NaN*. The simulation continues to run, but because all events are ordered on the time they are scheduled, this event is never processed. This disrupts the simulation in such a way that the results become unreliable and should no longer be used in the evaluation. Despite this, the results are used in the evaluation, which causes the optimization algorithm to optimize for the rare occurrence that this bug occurs. It will create such a configuration that, should the bug occur, the configuration is able to exploit it to maximum efficiency, resulting in an unbeatable cost. The end result of this is that the algorithm is optimizing for a bug instead of a realistic scenario, making the experiment results useless.

The symptoms of this bug have been discovered, and a workaround has been created to prevent this bug from disrupting the simulation run. This way, the simulation results are still usable and the optimization algorithm is not able to exploit the bug to get an unbeatable cost. However, the root cause for this bug is still unknown and should be investigated further.

For practical reason, the runtime of the Simulated Annealing algorithm could use another look. The runtime of the experiments vary wildly. Some runs took 3.5 hours, some took 16 hours, one even took 19 hours. It is unclear what causes these varying runtimes. Perhaps the machine on which the experiments are run is also performing updates or entering some low power mode during the night. Whatever the reason, these runtimes are quite impractical. Possible solutions could be to run the experiments on a more powerful machine, or even running it in the cloud.

Finally, it would be valuable for bol.com to create a tool that uses the work done in this project to attempt to find an optimal configuration for different services and scenarios. A graphical user interface that allows engineers at bol.com to setup their own experiments would be a practical continuation of the work done in this project for bol.com.

References

[1] Binkert, N. L., Hallnor, E. G., and Reinhardt, S. K. Network-oriented full-system simulation using m5. In Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW) (2003), pp. 36–43.

[2] Boucharas, V., Jansen, S., and Brinkkemper, S. Formalizing software ecosystem modeling. In Proceedings of the 1st international workshop on Open component ecosystems (2009), ACM, pp. 41–50.

[3] Breslau, L., Estrin, D., Fall, K., Floyd, S., Heidemann, J., Helmy, A., Huang, P., McCanne, S., Varadhan, K., Xu, Y., et al. Advances in network simulation. Computer 33, 5 (2000), 59–67.

[4] Cabrera, J. B., Gosar, J., Lee, W., and Mehra, R. K. On the statistical distribution of processing times in network intrusion detection. In Decision and Control, 2004. CDC. 43rd IEEE Conference on (2004), vol. 1, IEEE, pp. 75–80.

[5] Decker, K., and Lesser, V. Quantitative modeling of complex computational task environments. In AAAI (1993), pp. 217–224.

[6] Düllmann, T. F., and van Hoorn, A. Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches. In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion (2017), ACM, pp. 171–172.

[7] Etsy. StatsD. `https://github.com/etsy/statsd`, 2018. [Online; accessed 7-May-2018].

[8] Google. Google Istio. `https://istio.io/`, 2018. [Online; accessed 1-March-2018].

[9] Hardavellas, N., Somogyi, S., Wenisch, T. F., Wunderlich, R. E., Chen, S., Kim, J., Falsafi, B., Hoe, J. C., and Nowatzyk, A. G. Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. ACM SIGMETRICS Performance Evaluation Review 31, 4 (2004), 31–34.

[10] Hartmann, L. Private communication with carst tankink.

[11] Hartmann, L. Private communication with marjan van den akker.

[12] Heinrich, R., van Hoorn, A., Knoche, H., Li, F., Lwakatare, L. E., Pahl, C., Schulte, S., and Wettinger, J. Performance engineering for microservices: research challenges and directions. In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion (2017), ACM, pp. 223–226.

[13] Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M. K., and Sekar, V. Gremlin: systematic resilience testing of microservices. In Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on (2016), IEEE, pp. 57–66.

[14] Kavulya, S., Tan, J., Gandhi, R., and Narasimhan, P. An analysis of traces from a production mapreduce cluster. In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (2010), IEEE Computer Society, pp. 94–103.

[15] Kellner, M. I., Madachy, R. J., and Raffo, D. M. Software process simulation modeling: why? what? how? Journal of Systems and Software 46, 2-3 (1999), 91–105.

[16] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. Optimization by simulated annealing. science 220, 4598 (1983), 671–680.

[17] Netflix. Netflix Hystrix. `https://github.com/Netflix/Hystrix`, 2018. [Online; accessed 1-March-2018].

[18] Schaeffer-Filho, A., Smith, P., and Mauthe, A. Policy-driven network simulation: a resilience case study. In Proceedings of the 2011 ACM Symposium on Applied Computing (2011), ACM, pp. 492–497.

[19] Sterbenz, J. P., Çetinkaya, E. K., Hameed, M. A., Jabbar, A., Qian, S., and Rohrer, J. P. Evaluation of network resilience, survivability, and disruption tolerance: analysis, topology generation, simulation, and experimentation. Telecommunication systems 52, 2 (2013), 705–736.