

**École polytechnique de Louvain**

# **Observing the detailed behaviour of large distributed applications in real time using $\Delta$ QSD**

Author: **Francesco NIERI**

Supervisor: **Peter VAN ROY**

Readers: **Tom BARBETTE, Peer STRITZINGER, Neil DAVIES**

Academic year 2024–2025

Master [120] in Computer Science

# Abstract

It is difficult to study the detailed behaviour of large distributed systems while they are running. What happens when there is an overload? Modern software development practices successfully fail to adequately consider essential quality requirements or even to consider properly whether a system can actually meet its intended outcomes. Current observability tools do not meet observability requirements when it comes to detecting problems early enough in running systems.

To tackle these problems, PNSol Ltd. developed the  $\Delta$ QSD paradigm a novel metrics-based and quality-centric paradigm that uses formalised outcome diagrams to explore the performance consequences of design decisions, as a performance blueprint of the system. PNSol Ltd. analyses the behaviour of existing systems using  $\Delta$ QSD, but the technology only works a posteriori, there is no way yet to analyse a system's behaviour in real time.

To advance the usage of  $\Delta$ QSD in distributed projects, we developed the  $\Delta$ Q oscilloscope, a dashboard to observe a running Erlang system in real time. The oscilloscope works by communicating to a stub which is attached to an Erlang system, the "system under test", which sends outcome instances to the oscilloscope.

In order to use the  $\Delta$ Q Oscilloscope, the first step is to determine the system's outcome diagram. The system's outcome diagram is a directed graph that shows the causal relationships between the system's internal operations (called outcomes) and its overall inputs and outputs. Each outcome corresponds to a system component. It has a start event and stop event, when the component is called and when it returns.

For each outcome of interest, an observation point (probe) is attached to measure the delay of that outcome. As many probes are put into the system as are needed to observe the desired system behaviours. The probes are implemented on top of OpenTelemetry tracing ability. Each probe sends a time series of data to the  $\Delta$ Q Oscilloscope, which performs statistical computations on all the time series and displays the results in real time.

# Acknowledgments

This thesis is the culmination of my studies, I would like to thank the people who made this possible and who supported me through the years.

My family, especially **my mom, my dad, my brother** and **my sister**, for their help which was a crucial shoulder I could lay on while writing this thesis and most importantly throughout these five years.

My **friends**, to those who have travelled to see the thesis presentation and to those who through the years have been my escape from the studies.

**A. M.**, for the projects and the moments we shared these four years together in uni.

**My dad** and **Maurizio**, who nurtured the passion for coding in me.

Lastly, **Peter Van Roy**, for his relentless interest, support and weekly and constant supervision that made this thesis possible.

# AI disclaimer

AI was used to help writing the dashboard interface and to help with the Erlang side.  
The written master thesis was written entirely without the aid of AI.

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Context . . . . .	2
1.2	Problem . . . . .	3
1.3	Previous work . . . . .	3
1.4	Contributions . . . . .	4
1.5	Roadmap . . . . .	4
<b>2</b>	<b>Background</b>	<b>6</b>
2.1	Observability . . . . .	6
2.1.1	erlang:trace . . . . .	6
2.1.2	OpenTelemetry . . . . .	7
2.2	An overview of $\Delta$ QSD . . . . .	9
2.2.1	Outcome . . . . .	9
2.2.2	Quality attenuation ( $\Delta$ Q) . . . . .	10
2.2.3	Failure semantics . . . . .	10
2.2.4	Partial ordering . . . . .	11
2.2.5	Timeliness . . . . .	11
2.2.6	QTA, required $\Delta$ Q . . . . .	11
2.2.7	Outcome diagram . . . . .	12
2.2.8	Independence hypothesis . . . . .	14
<b>3</b>	<b>Design</b>	<b>16</b>
3.1	Scenario . . . . .	16
3.2	Extending the notion of failure . . . . .	17
3.3	Time series . . . . .	17
3.4	Probes . . . . .	18
3.4.1	Inserting probes in the oscilloscope . . . . .	19
3.4.2	Probes: from spans to outcome instances . . . . .	19
3.5	Triggers . . . . .	20
3.5.1	Snapshot . . . . .	20
<b>4</b>	<b>Oscilloscope: High level implementation</b>	<b>21</b>
4.1	$\Delta$ Q implementation . . . . .	21
4.1.1	Internal representation of a $\Delta$ Q . . . . .	21
4.1.2	dMax . . . . .	22
4.1.3	QTA . . . . .	23

4.1.4	Operations . . . . .	23
4.1.5	Confidence bounds . . . . .	24
4.1.6	Rebinning . . . . .	25
4.2	$\Delta Q$ display . . . . .	26
4.3	Outcome diagram . . . . .	27
4.3.1	Probes . . . . .	27
4.3.2	Operators, outcome expressions . . . . .	28
4.3.3	Limitations . . . . .	29
4.4	Dashboard . . . . .	29
4.4.1	Sidebar . . . . .	29
4.4.2	Stub controls . . . . .	30
<b>5</b>	<b>Oscilloscope: Low Level Implementation</b>	<b>31</b>
5.1	Wrapper . . . . .	31
5.1.1	API . . . . .	31
5.1.2	Handling outcome instances . . . . .	33
5.1.3	TCP connection . . . . .	34
5.2	Parser . . . . .	34
5.2.1	ANTLR . . . . .	34
5.2.2	Grammar . . . . .	35
5.3	Oscilloscope GUI . . . . .	36
<b>6</b>	<b>Application on synthetic programs</b>	<b>37</b>
6.1	M/M/1/K queue . . . . .	37
<b>7</b>	<b>Performance study</b>	<b>39</b>
7.1	Convolution performance . . . . .	39
7.2	Stub performance . . . . .	40
<b>A</b>	<b>User manual</b>	<b>43</b>
A.1	Sidebar: Probes settings . . . . .	43
A.1.1	Setting a QTA . . . . .	43
A.1.2	Setting the parameters of a probe . . . . .	43
A.2	Sidebar: Outcome diagram and plots . . . . .	45
A.2.1	Creating the system . . . . .	45
A.2.2	Saving the system definition . . . . .	45
A.2.3	Loading the system definition . . . . .	45
A.2.4	Managing the plots . . . . .	46
A.2.5	Removing a plot . . . . .	46
A.3	Triggers . . . . .	46



# Chapter 1

## Introduction

### 1.1 Context

$\Delta$ QSD is an industrial-strength approach for large-scale system design that can predict performance and feasibility early on in the design process. Developed over 30 years by a small group of people around Predictable Network Solutions Ltd, the paradigm has been applied in various industrial-scale problems with huge success and large savings in costs. [1] This project will develop a practical tool, the  **$\Delta$ Q Oscilloscope**, for the Erlang developer community.

The Erlang language and Erlang/OTP platform are widely used to develop distributed applications that must perform reliably under high load. The tool will provide useful information for these applications both for understanding their behaviour, for diagnosing performance issues, and for optimizing performance over their lifetime.

The  $\Delta$ Q Oscilloscope will perform statistical computations to show real time graphs about the performance of system components. With the oscilloscope prototype we will present in this paper, we are aiming to show that the  $\Delta$ QSD paradigm is not only a theoretical paradigm, but it can be employed in a tool to diagnose large distribute systems.

Modern software development practices successfully fail to adequately consider essential quality requirements or even to consider properly whether a system can actually meet its intended outcomes, particularly when deployed at scale, the  $\Delta$ QSD paradigm addresses this problem!

The oscilloscope targets large distributed applications handling many independent tasks where performance and reliability are important.

$\Delta$ QSD has important properties which make its application to distributed projects interesting, it supports:

- A compositional approach that considers performance and failure as first-class citizens.
- Stochastic approach to capture uncertainty throughout the design approach.



- Performance and feasibility can be predicted at high system load for partially defined systems

While the paradigm has been successfully applied in a **posteriori** analysis, there is no way yet to analyse a distributed system which is running in real time! This is where the  $\Delta Q$  oscilloscope comes in.

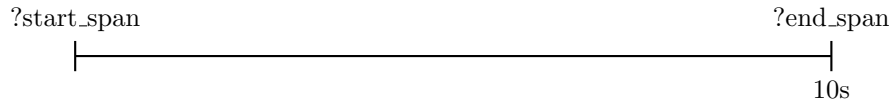
## 1.2 Problem

A legitimate question would ask why one would need an additional tool to observe their system, observability tools are already plenty and provide useful insights into an application's behaviour.

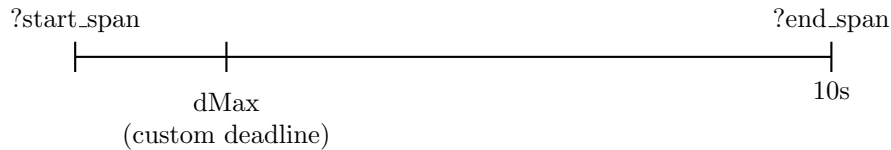
While they may seem adequate enough to provide a global oversight of applications, they fail to diagnose real time problems like overload, dependent behaviour early enough and in a quick manner.

The problem we are trying to tackle can be described by the following situation:

Imagine an Erlang application instrumented with OpenTelemetry, suddenly, the application starts slowing down, and the execution of a function takes 10 seconds, now, between its start and its end, the user instrumenting the application sees nothing in his dashboard.



This is a big problem! One would like to know right away if something is wrong with their application. This is where the  $\Delta QSD$  paradigm and the  $\Delta Q$  oscilloscope come in handy.



By extending  $\Delta QSD$  notion of early failure one can now right away when the system is overloaded and showing problems, one can know right away when something is wrong, as soon as the maximum delay is hit, this avoids waiting, in this case, 10 seconds to know that something is wrong with your application.

## 1.3 Previous work

The  $\Delta QSD$  paradigm has been formalised across different papers [2] [3] and was brought to the attention of engineers via tutorials [1] and to students at Université Catholique de Louvain [4].

A Jupyter notebook workbench has been made available on GitHub [5], it shows real time  $\Delta Q$  graphs for typical outcome diagrams but is not adequate to be scaled to real time systems, it is meant as an interactive tool to show how the  $\Delta QSD$  paradigm can be applied to real life examples.

Observability tools such as Erlang tracing [6] and OpenTelemetry [7] lack the notions of failure as defined in  $\Delta QSD$ , which allows detecting performance problems early on, we base our program on OpenTelemetry to incorporate already existing notions of causality and observability to augment their capabilities and make them suitable to work with the  $\Delta QSD$  paradigm.

## 1.4 Contributions

There are a few contributions that make the master thesis and thus, the oscilloscope, possible:

- A graphical interface to display  $\Delta Q$  plots for outcomes.
- An Erlang OpenTelemetry wrapper to give OpenTelemetry spans a notion of failure and to communicate with the oscilloscope.
- An implementation of a syntax, derived from the original algebraic syntax to create outcome diagrams.
- The implementation of  $\Delta QSD$  concepts from theory to practice, allowing outcomes and probes to be displayed and analysed on the oscilloscope.
- An efficient convolution algorithm based on the FFTW3 library.
- A system of triggers to catch rare events when system behaviour fails to meet quality requirements, giving a snapshot of the system, giving the user insights about their system's behaviour.
- Synthetic applications to test the effectiveness of  $\Delta QSD$  on diagnosing systems and their feasibility.

These contributions can show that the  $\Delta QSD$  has its practical applications and is not limited to a theoretical view of system design

## 1.5 Roadmap

The following thesis will give the reader everything that is needed to use the Oscilloscope and exploit it to its full potential.

We divided the thesis in multiple chapters, below is the roadmap of the content:

- The background chapter gives the reader an introduction to the tools we leverage in our program, namely, OpenTelemetry and an extensive background into the theoretical foundations of  $\Delta QSD$ , which are the basis of the oscilloscope and are fundamental to understand how to correctly use and analyse the output given by the oscilloscope.

- The design chapter delves into how the parts of the system interact together.
- The implementation part is split in two. First, we provide a high level abstraction of how  $\Delta$ QSD is implemented in the oscilloscope. Secondly, a more low level explanation, which goes into more technical details of the parts that compose the oscilloscope.
- Lastly, we provide synthetic applications which have been tested with the oscilloscope and an evaluation of the performance of the oscilloscope.

We end by providing future possibilities which can be explored, and concepts which we believe ought to be implemented in observabilities tools. In the appendix, we provide a user manual to help users use the oscilloscope, along with Erlang and C++ source code of the oscilloscope and the wrapper.

# Chapter 2

## Background

### 2.1 Observability

Observability refers to the ability to understand the internal state by examining its output, in the context of a distributed system, being able to understand the internal state of the system examining its telemetry data. [8]

In the case of the Erlang programming language, we explain below two tools that can be used to observe an Erlang program.

#### 2.1.1 `erlang:trace`

The Erlang programming language gives the users different ways to observe the behaviour of the code, one of those is the function `erlang:trace/3`. The erlang run-time system exposes several trace points that can be observe, observing the trace points allows users to be notified when they are triggered [6]. One can observe function calls, messages being sent and received, process being spawned, garbage collecting ....

```
-spec trace(PidPortSpec, How, FlagList) -> integer()
  when
    PidPortSpec ::
      pid() |
      port() |
      all | processes | ports | existing | existing_processes |
      ↪ existing_ports | new |
      new_processes | new_ports,
    How :: boolean(),
    FlagList :: [trace_flag()].
```

Figure 2.1: `erlang:trace\3` specification

Nevertheless, Erlang Tracing, according to our use case, has a major flaw: no notion of causality. If two messages  $a, b$  are sent and then received in disorder, the tracer

has no default way of knowing which is which, this is a missing feature that is crucial for observing a program functioning and being able to connect an application to our oscilloscope. This is where the OpenTelemetry standard comes in.

## 2.1.2 OpenTelemetry

OpenTelemetry is an open-source, vendor agnostic observability framework and toolkit designed to generate, export and collect telemetry data, in particular traces, metrics and logs. [8]

OpenTelemetry is available for a plethora of languages, including Erlang, although, as of writing this, only traces are available in Erlang.

OpenTelemetry provides a standard protocol, a single set of API and conventions and lets you own the generated data, allowing to switch between observability backends freely.

The Erlang Ecosystem Foundation has a working group focused on evolving the tools related to observability.

### Traces

Traces are why we are basing our program on top of OpenTelemetry, traces follow the whole "path" of a request in an application, traces are comprised of one or more spans.

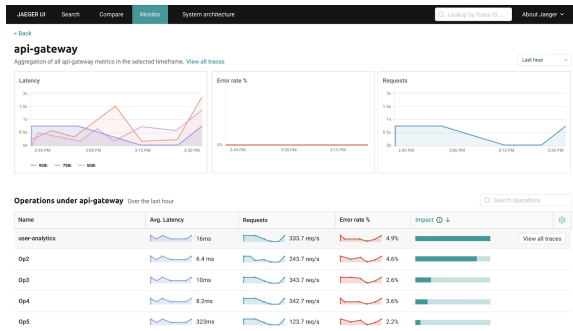
**Span** A span is a unit of work or operation. Spans can be correlated to each other and can be assembled into a trace. The notion of spans and traces allows us to follow the execution of a "request" and carry a context, allowing us to get the causal links of messages. [9]

```
{
  "name": "oscilloscope-span",
  "context": {
    "trace_id": "5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "5fb397be34d26b51"
  },
  "parent_id": "0515505510cb55c13",
  "start_time": "2022-04-29T18:52:58.114304Z",
  "end_time": "2022-04-29T22:52:58.114561Z",
  "attributes": {
    "http.route": "some_route"
  },
}
```

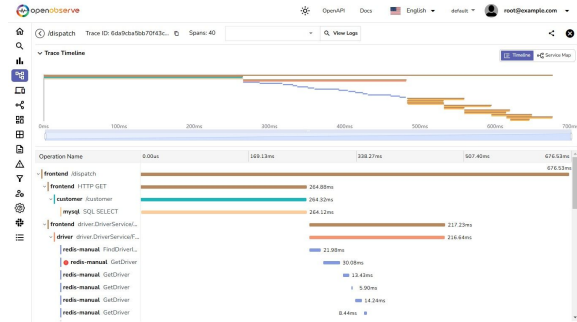
Figure 2.2: Example of span with a parent, indicating a causal link between parent and children span [9]

## Exporters

OpenTelemetry gives the possibility to export traces to backends such as Jaeger or Zipkin. A user can monitor their workflows, analyze dependencies, troubleshoot their programs by observing the flow of the requests in such backends. [10]



(a) Jaeger interface [11].



(b) Right: A span analysis on OpenObserve [12]

## Macros

OpenTelemetry provides macros to start, end and interact with spans in Erlang. [13]

**?with\_span** ?with\_span creates active spans. An active span is the span that is currently set in the execution context and is considered the "current" span for the ongoing operation or thread. [14]

```
parent_function() ->
```

```
    ?with_span(parent, #{}, fun child_function/0).
```

```
child_function() ->
```

```
    %% this is the same process, so the span parent set as the active
    %% span in the with_span call above will be the active span in
```

```
    ↪ this function
```

```
    ?with_span(child, #{},
```

```
        fun() ->
```

```
            %% do work here. when this function returns, child
            ↪ will complete.
```

```
        end).
```

**?start\_span** ?start\_span creates a span which isn't connected to a particular process, it does not set the span as the current active span.

```
SpanCtx = ?start_span(child),
```

```
Ctx = otel_ctx:get_current(),
```

```
proc_lib:spawn_link(fun() ->
```

```
    otel_ctx:attach(Ctx),
```

```
    ?set_current_span(SpanCtx),
```

```

    %% do work here
    ?end_span(SpanCtx)
end),

```

`?end_span` `?end_span` ends a span started with `?start_span`

## 2.2 An overview of $\Delta$ QSD

$\Delta$ QSD is a metrics-based, quality-centric paradigm that uses formalised outcome diagrams to explore the performance consequences of design decisions. [2]

Key concepts of  $\Delta$ QSD are **quality attenuation** ( $\Delta$ Q) and **outcome diagram**

Outcome diagrams capture observational properties of the system. The  $\Delta$ QSD paradigm derives bounds on performance expressed as probability distribution, encompassing all possible executions of the system.

The following sections are a summary of multiple articles and presentation formalizing the paradigm. [2] [1] [3] [15]

### 2.2.1 Outcome

Assume a component  $C$  which receives a message  $m_{in}$  and outputs a message  $m_{out}$  after a delay  $d$ . Over multiple executions, we will have observed multiple delays which can be represented as a cumulative definition where  $p$  percent of delays have delay  $\leq d$ . [3] Each outcome has a  $\Delta$ Q.

An outcome  $O$  is a specific system behaviour that can be observed to start at some point in time and **may** be observed to complete at some later time. [15] Formally, what the system obtains by performing one of its tasks. One task corresponds to one outcome and viceversa. When an outcome is performed, it means that the task of an outcome is performed.

**Observables** Each outcome has two starting sets of events: the starting sets and the ending sets. Such sets are called the *observables*. Once an event from the starting set occurs, there is no guarantee that a corresponding event in the terminating set will occur within the duration limit (required time to complete). An observable is *done* when it occurs during the time limit.

**Outcome instance** Given a starting event  $e_{in}$  and an end event  $e_{out}$ , an *outcome instance* is what the system gains within  $(e_{in}, e_{out})$ .

**Representation** Outcome are represented as circles, with the starting and terminating set of events being represented by boxes.

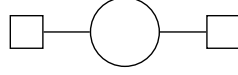


Figure 2.4: The outcome (circle) and the starting set (left) and terminating set (right) of events

### 2.2.2 Quality attenuation ( $\Delta Q$ )

$\Delta Q$  is a cumulative distribution function that defines both latency and failure probability between a start and end event [1]

In an ideal system, an outcome would deliver a desired behaviour without error, failure, delay, but this is not the case. The quality of an outcome response "attenuated to the relative ideal" (the cumulative distribution function) is called "quality attenuation" ( $\Delta Q$ ). Since it can depend on many factors (geographical, physical ...),  $\Delta Q$  is modeled as a random variable.

As  $\Delta Q$  captures deviation from ideal behavior, and incorporates delay, which is a continuous random variable, and failures/timeouts, which are discrete variables, it can be described mathematically as an *Improper Random Variable*, where the probability of a delay  $< 1$ . Combining latency and failure together makes it easy to examine the tradeoffs between them.

$\Delta Q(x)$  will be the probability that an outcome  $O$  will occur in time  $t \leq x$ .

The **intangible mass**  $1 - \lim_{x \rightarrow \infty} \Delta Q(x)$  of a  $\Delta Q$  will encode the probability of failure/timeout/exception occurring.

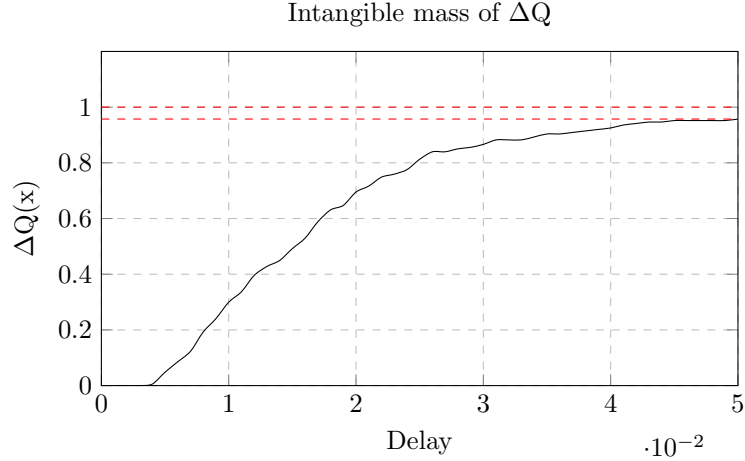


Figure 2.5: Intangible mass (red) of a  $\Delta Q$ , the observable had a failure rate of about 5%

### 2.2.3 Failure semantics

Quality attenuation models both delay and failure. In the CDF representation of a  $\Delta Q$ , there is an  $f$  percent probability that the delay is infinite, this is what failure models.



Concretely, it means that an input message  $m_{in}$  has no output message  $m_{out}$ .

Combining delay and failure in a single quantity is what makes  $\Delta QSD$  a great choice to explore feasibility in system design.

#### 2.2.4 Partial ordering

A CDF of a  $\Delta Q$  is *less than* the other if its CDF is everywhere to the left and above the other. Mathematically, it is a partial order.

If two  $\Delta Q$ s intersect, they are not ordered.

#### 2.2.5 Timeliness

Timeliness is defined as a relation between an observed  $\Delta Q_{obs}$  and a required  $\Delta Q_{req}$ . Timeliness is delivering results within required time bounds (sufficiently often).

A system *satisfies timeliness* if  $\Delta Q_{obs} \leq \Delta Q_{req}$ .

#### 2.2.6 QTA, required $\Delta Q$

The Quantitative Timeliness Agreement maps objective measurements to the subjective perception of application performance [cite]. It specifies what the base system does and its limits

**QTA example** : Imagine a system where 25% of the executions should take  $< 15$  ms, 50%  $< 25$  ms and 75%  $< 35$  ms, all queries have a maximum delay of 50ms and 5% of executions can timeout, the QTA can be represented as a step function.

**Slack** When  $\Delta Q$  is strictly less than the requirement, we say there is performance *slack*.

**Hazard** When  $\Delta Q$  is strictly greater than the requirement, there is performance hazard.

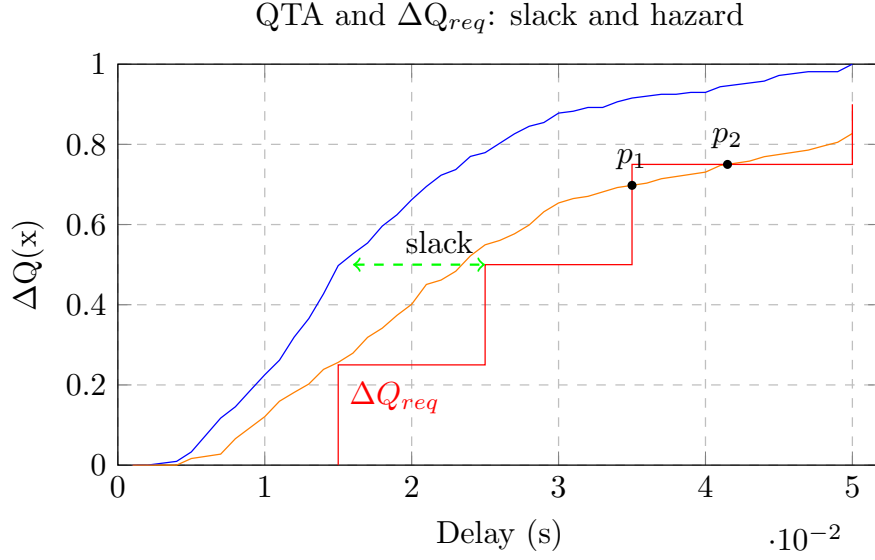


Figure 2.6: The system in blue is showing slack and satisfies the requirement, the system in orange is showing signs that it cannot handle the stress, it is not respecting the  $\Delta Q_{req}$ .

### 2.2.7 Outcome diagram

An outcome diagram captures the causal relationships between the outcomes, each outcome diagram can be presented algebraically with an outcome expression. It allows computing the  $\Delta Q$  for the whole system. The outcome diagram should capture the essential observational properties of a system. There are four different ways to represent the relationships between outcomes.

#### Sequential composition

If we assume two outcomes  $O_A, O_B$  where end event of  $O_A$  is the start event of  $O_B$ , the  $\Delta Q$  of  $O_A$  and  $O_B$  is given by the convolution of the PDFs of  $O_A$  and  $O_B$ . Where convolution ( $\otimes$ ) between two PDF is :

$$PDF_{AB}(t) = \int_0^t PDF_A(\delta) \cdot PDF_B(t - \delta) d\delta \quad (2.1)$$

and thus  $\Delta Q_{AB}$ :

$$\Delta Q_{AB} = \Delta Q_A \otimes \Delta Q_B \quad (2.2)$$

### First to finish

If we assume two independent outcomes  $O_A, O_B$  with the same start event, first-to-finish occurs when at least one end event occurs, it can be calculated as:

$$\begin{aligned}\Delta Q_{FTF(A,B)} &= Pr[d_A > t \wedge d_B > t] \\ &= Pr[d_A > t] \cdot Pr[d_B > t] = (1 - \Delta Q_A) \cdot (1 - \Delta Q_B) \\ \Delta Q_{FTF(A,B)} &= \Delta Q_A + \Delta Q_B - \Delta Q_A \cdot \Delta Q_B\end{aligned}\tag{2.3}$$

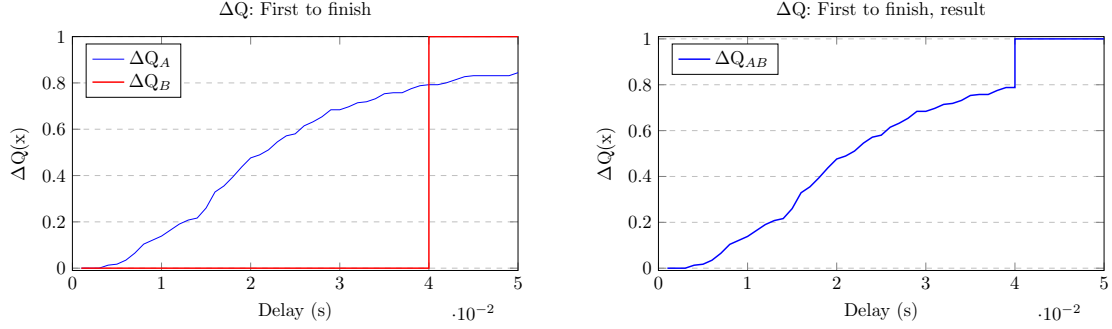


Figure 2.7: Left: Two  $\Delta Q$ s, A and B. Right: The result of the all to finish operator applied on A and B

### All to finish

If we assume two independent outcomes  $O_A, O_B$  with the same start event, all-to-finish occurs when both end events occur, it can be calculated as:

$$\begin{aligned}\Delta Q_{ATF(A,B)} &= Pr[d_A \leq t \wedge d_B \leq t] \\ &= Pr[d_A \leq t] \cdot Pr[d_B \leq t] = \Delta Q_A \cdot \Delta Q_B \\ \Delta Q_{ATF(A,B)} &= \Delta Q_A \cdot \Delta Q_B\end{aligned}\tag{2.4}$$

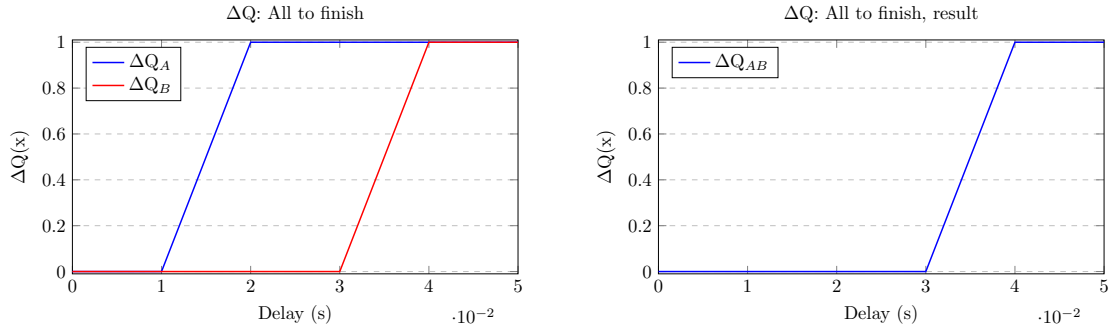


Figure 2.8: Left: Two  $\Delta Q$ s, A and B. Right: The result of the all to finish operator applied on A and B

### Probabilistic choice

If we assume two possible outcomes  $O_A$  and  $O_B$  and exactly one outcome is chosen during each occurrence of a start event and:

- $O_A$  happens with probability  $\frac{p}{p+q}$
- $O_B$  happens with probability  $\frac{q}{p+q}$

$$\Delta Q_{PC}(A, B) = \frac{p}{p+q} \Delta Q_A + \frac{q}{p+q} \Delta Q_B \quad (2.5)$$

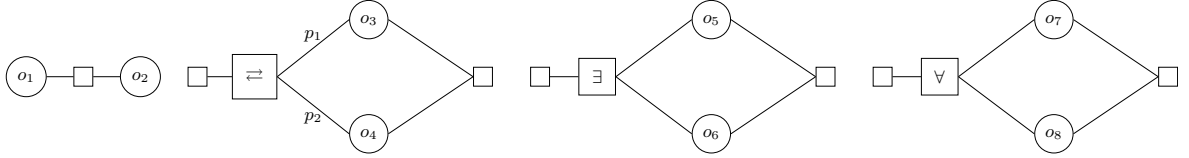


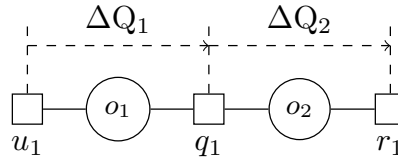
Figure 2.9: The possible relationships in an outcome diagram: Sequential composition, probabilistic choice, first-to-finish, all-to-finish

First-to-finish, All-to-finish and probabilistic-choice are calculated on the CDF of the  $\Delta Q$ .

These outcome expressions can be assembled together to create an outcome diagram, later on, we will see how one can put translate the graphical representation to outcome diagrams which can be used in the  $\Delta Q$  oscilloscope.

## 2.2.8 Independence hypothesis

Assume two sequentially composed outcomes  $o_1$ ,  $o_2$  running on the same processor. A probe  $p$  observing the execution from the start event of  $o_1$  to the end event of  $o_2$ .



At low load, the two components behavior will be independent, the system will behave linearly, the observed delay of the probe  $p$  will be equal to the convolution of  $o_1$ ,  $o_2$  ( $o_1 \otimes o_2$ ).

When load increases, the two components will start to show dependent behaviour due to the processor utilisation increasing, the observed  $\Delta Q$  will then deviate from what is calculated.

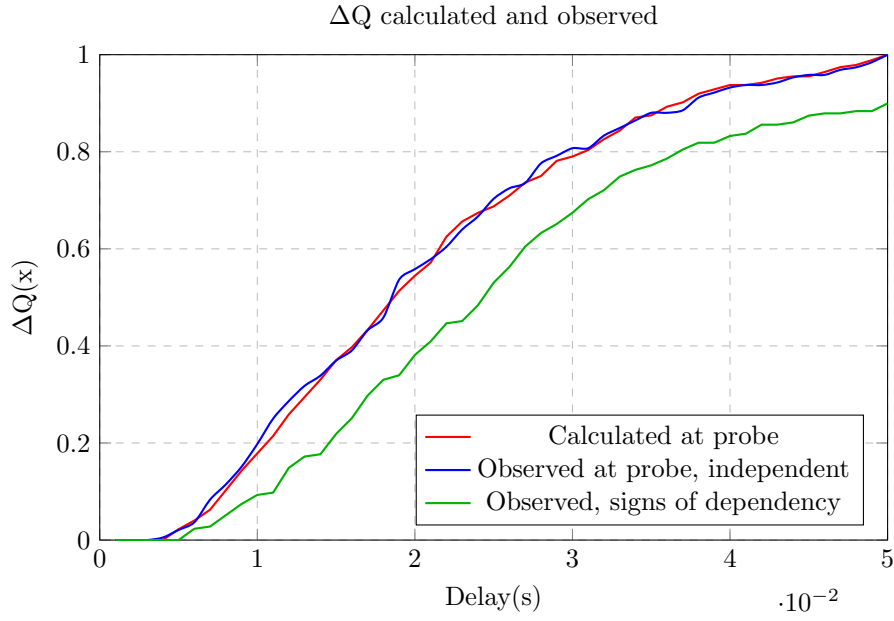


Figure 2.10: When the components are independent, what is observed (blue) and calculated (red) can be superposed, whilst when  $o_1$  and  $o_2$  show initial signs of dependency, what is observed (green) can be seen deviating from the calculated  $\Delta Q$ .

When the system is far from being overloaded, the effect is noticeable thanks to  $\Delta QSD$  even if the system is far from being overloaded. As the cliff edge of overload is approached, the nonlinearity will increase.

# Chapter 3

## Design

### 3.1 Scenario

We can define four parts of interest for our tool.

**System under test** The system under test is the Erlang system the engineer wishes to observe, it ideally is a system which already is instrumented with OpenTelemetry. The ideal system where  $\Delta QSD$  is more useful is a system that executes many independent instances of the same action.

**Stub/wrapper** The stub is the `otel_wrapper`, a wrapper that starts and ends OpenTelemetry spans, and start custom spans which are useful for the oscilloscope. Further handling of OpenTelemetry spans is delegated to the user, who may wish to do further operations with their spans.

The custom spans can be ended normally like OpenTelemetry spans or can timeout, given a custom timeout, and fail, according to user's definition of failure.

The stub is called from the system under test and communicates spans data to the oscilloscope via TCP.

The stub can receive messages from the oscilloscope, the messages are about updating observable's *dMax*.

**Server** The server is responsible for receiving the messages containing the custom spans from the oscilloscope. The server forwards the spans to the oscilloscope.

**Oscilloscope** The oscilloscope receives the custom spans from the stub and creates samples from said spans.

The oscilloscope has a graphical interface which allows the user to create an outcome diagram of the system under test, display real time graphs which show detail about the execution of the system and allow the user to set custom timeouts for observables.

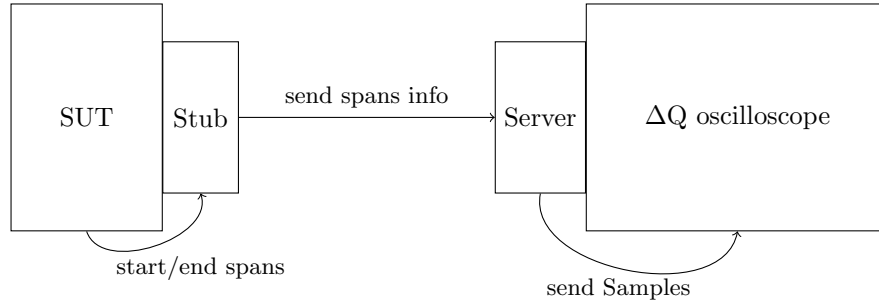


Figure 3.1: Global system diagram: the SUT calls the wrapper when starting, ending and failing spans. The spans are received by the server which processes them and sends them to the oscilloscope. The server communicates with the stub to update informations about the system under test ( $dMax$ )

## 3.2 Extending the notion of failure

Whilst previously we defined as "an input message  $m_{in}$  that has no output message  $m_{out}$ ", we extend this definition. If you recall the introduction 1.2, we introduced the notion of a maximum delay.

We extend the notion of failure to the following definition:

*"An input message  $m_{in}$  that has no output message  $m_{out}$  after  $x$  seconds"*

Where  $x$  is defined by the user. We can leverage this new definition to observe  $\Delta Q$ s in real time.

## 3.3 Time series

Consider an observable  $O$  with two distinct sets of events, the starting set of events  $s$  and ending set of event  $e$ , the time series of an observable can be defined by  $n$  outcome instances  $s_i$  with the following structure:

- The probe's name
- The start time  $t_s$
- The end time  $t_e$
- Its status
- Its elapsed time of execution

The sample has three possible statuses: **success**, **timeout**, **failure**, it can thus be broken down in the representations, based on its status:

- $(t_s, t_e)$ : This representation indicates that the execution was successful ( $t < dMax$ ), the nstance contains the start time and end time of the event.

- $(t_s, \mathcal{T})$ : This representation indicates that the execution has timed out ( $t > dMax$ ). The instance does not contain the end time, it is irrelevant as we know that its execution has taken more than  $dMax$ .
- $(t_s, \mathcal{F})$ : This representation indicates the execution has failed given a user defined requirement (i.e. a dropped message given buffer overload in a queue system). It must not be confused with a program failure (crash), if a program crashes during the execution of event  $e$ , it will timeout since the stub will not receive an end message. As was the case for the timeout, we do not need to know the end time as the sample encodes a failure.

The  $\Delta Q$  can then be modelled easily with  $n$  outcome instances by calculating its PDF and consequently the ECDF.

### 3.4 Probes

To observe the system under test, the resulting outcomes, the result of causal links and outcome expressions, we must put probes in the system.

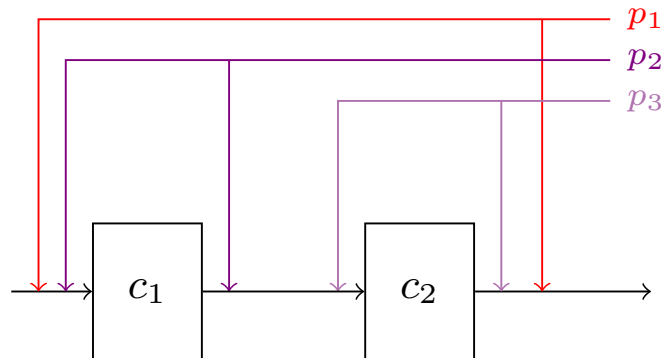
For each outcome of interest, a probe (observation point) is attached to measure the delay of the outcome, like one would in a true oscilloscope.

These probes allow to connect the system under test to the stub, which in turns sends outcome instances to the oscilloscope, which performs statistical computations on all the time series.

Consider the figure below, a probe is attached at every component to measure the delay of N outcome  $(p_2, p_3)$ , the stub will send the outcome instance data for each probe observing an outcome to the oscilloscope, which will measure their respective  $\Delta Q$ s from the time series data and display them (if chosen by the user).

Another probe ( $p_1$ ) is inserted at the beginning and end of the system to measure the global execution delay.

Thanks to this probe, the user can observe the  $\Delta Q$  "observed at  $p_1$ ", which is the  $\Delta Q$  which was calculated from the data received by inserting probe  $p_1$ . The  $\Delta Q$  "calculated at  $p_1$ " is the resulting  $\Delta Q$  from the convolution of the observed  $\Delta Q$ s at  $c_2$  and  $c_3$ .

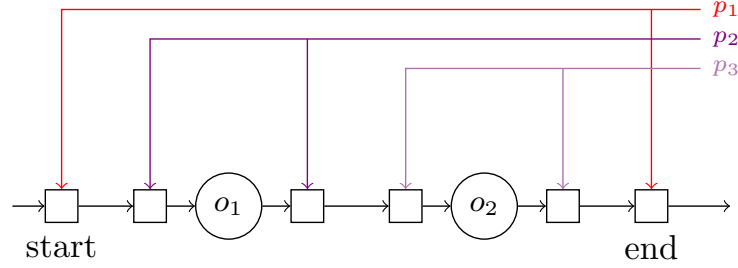




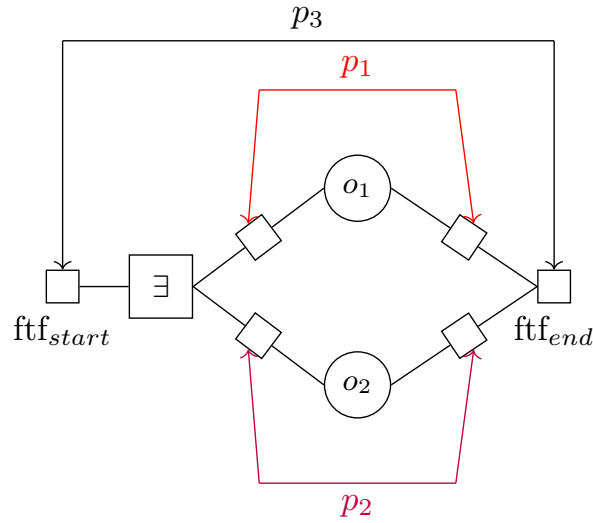
### 3.4.1 Inserting probes in the oscilloscope

Probes are automatically inserted in the oscilloscope when defining outcomes, part of the system the user wishes to observe and operators.

In the system below, which is equal to the one defined above, probes are automatically attached to outcomes  $o_1, o_2$ . The user who wants to observe the result of the sequential composition can insert probes at the start and end of the routine.



As for operators (outcome expression), probes are automatically attached to the components inside them and to the start event and end events of the operators.



The **observed**  $\Delta Q$  for the first-to-finish operator is the  $\Delta Q$  from the instances (**start**, **end**). The **calculated**  $\Delta Q$  is the  $\Delta Q$  which is the result of the first-to-finish operator being applied on  $o_1, o_2$

### 3.4.2 Probes: from spans to outcome instances

OpenTelemetry spans are useful to carry context, attributes and baggage in a program, the plethora of attributes they have is nevertheless too much for the oscilloscope.

To get the equivalent of spans for the oscilloscope, the stub needs to be called at the starting events of a probe to start an instance, and at the ending events to end the outcome instance and send the data to the oscilloscope.

```

% Start the outcome instance of worker_2
{WorkerCtx, WorkerPid} = otel_wrapper:start_span(<<"worker_2">>),

% Do work here ...

%End the outcome instance of worker_2
otel_wrapper:end_span(WorkerCtx, WorkerPid),

```

## 3.5 Triggers

The concept of triggers is key to the oscilloscope, much like an oscilloscope that has a trigger mechanism that fires when a signal of interest is recognized by the oscilloscope, the  $\Delta Q$  oscilloscope has a similar mechanism that can recognize when an observed  $\Delta Q$  violates certain conditions regarding required behaviour.

Each time an observed  $\Delta Q$  is calculated by the oscilloscope, it is checked against the requirements set by the user. If these requirements are not met, a trigger is fired and a snapshot of the system is saved to be shown to the user.

### 3.5.1 Snapshot

A snapshot of the system gives insights into the system before and after a trigger was fired. It gives the user a still of the system, as if it was frozen in time.

When an observed  $\Delta Q$  is calculated for each component being observed in the oscilloscope, they are stored away. In case the probe is observing multiple components, the  $\Delta Q$  which is the result of the outcome expressions is also stored.

If no trigger is fired, older observed (and calculated)  $\Delta Q$ s are removed. In the case that a trigger is fired, the oscilloscope keeps recording  $\Delta Q$ s without removing older ones, to allow the user to look at the state of the system before the trigger and after.

# Chapter 4

## Oscilloscope: High level implementation

### 4.1 $\Delta Q$ implementation

Originally,  $\Delta Q(x)$  denotes the probability that an outcome occurs in a time  $t \leq x$ , defining then the "intangible mass" of such IRV as  $1 - \lim_{x \rightarrow \infty} \Delta Q(x)$ . We then extend the original definition to fit real time constraints, needing to calculate  $\Delta Q$ s continuously.

For a given probe,  $\Delta Q(t_l, t_u, dMax)$  is the probability that the time of series with samples between time  $t_l < t_u$ , an outcome or probe occurs in time  $t \leq dMax$ .

#### 4.1.1 Internal representation of a $\Delta Q$

We provide a  $\Delta Q$  class to calculate the  $\Delta Q$  of a probe between a lower time bound  $t_l$  and an upper time bound  $t_u$ .

The  $\Delta Q$  can be calculated in various ways:

**Observed  $\Delta Q$**  The first way is by having  $n$  collected outcome instances between  $t_l$  and  $t_u$ , calculating its PDF and then calculating the *empirical cumulative distribution function* (ECDF) based on its PDF. This is called the **Observed  $\Delta Q$** .

**Calculated  $\Delta Q$**  A  $\Delta Q$  can also be calculated by performing operations which are the result of outcome expressions on two or more  $\Delta Q$ s, the notion of outcome instances is then lost between calculations, as the interest shifts towards calculating the resulting PDFs and ECDFs. This is called the **Calculated  $\Delta Q$** .

Below, is how both are calculated given  $n$  samples.

#### PDF

We approximate the PDF of the observed random variable  $\mathbf{X}$  via a histogram. We partition the values into  $N$  bins of equal width, this is required to ease future calculations.

Given  $[x_i, x_{i+1}]$  the interval of a bin  $i$ , where  $x_i = i\Delta x$ , and  $\hat{p}(x_i)$  the value of the PDF at bin  $i$ , for  $n$  bins:

$$\begin{cases} \hat{p}(i) = \frac{n_i}{n}, & \text{if } i \leq n \\ \hat{p}(i) = \hat{p}(n), & \text{if } i > n \end{cases} \quad (4.1)$$

With  $n_i$  the number of successful samples whose elapsed time is contained in the bin  $i$ ,  $n$  the total number of samples.

## ECDF

The value  $\hat{f}(x_i)$  of the ECDF at bin  $i$  with  $n$  bins can be calculated as:

$$\begin{cases} \hat{f}(i) = \sum_{j=1}^i \hat{p}(j), & \text{if } i \leq n \\ \hat{f}(i) = \hat{f}(x_n), & \text{if } i > n \end{cases} \quad (4.2)$$

### 4.1.2 dMax

The key concept of  $\Delta$ QSD is having a maximum delay after which we consider that the execution is failed, this is represented in an observable as  $dMax$ . The user defines, for each observable the maximum delay its execution can have.

Setting a maximum delay for an observable is not a job that can be done one-off and blindly, it is something that is done with an underlying knowledge of the system inner-workings and must be thoroughly fine tuned during the execution of the system by observing the resulting distributions of the obtained  $\Delta$ Qs.

We define in our oscilloscope a formula to dynamically define a maximum delay:

$$dMax = \Delta_{tbase} * 2^n * N \quad (4.3)$$

Where:

- $\Delta_{tbase}$  represents the base width of a bin, equal to 1ms.
- $N$  the number of bins.

The user must choose:

- $n$ : We define a range  $[-10, 10]$  for  $n$ . This gives a good enough range to account for all possible delays an user wishes to have.
- $N$ : We define a range  $[1, 1000]$  for  $N$ . This is a good enough bound to allow for finer grained bins, or less precision if needed.

Some tradeoffs must though be acknowledged when setting these parameters, a higher number of bins corresponds to a higher number of calculations and space complexity, a lower  $dMax$  may correspond to more failures. These are all tradeoffs that must be considered by the system engineer and set accordingly.

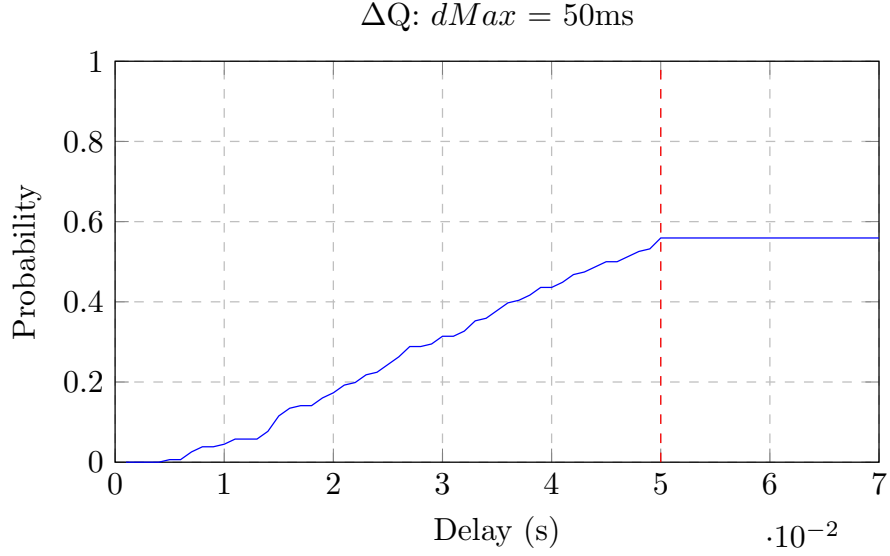


Figure 4.1:  $\Delta Q: dMax = 50ms$ , the CDF will stay constant when delay  $> dMax$

### 4.1.3 QTA

A simplified QTA is defined for probes. We define 4 points for the step function at 25, 50, 75 percentiles and the maximum amount of failures accepted for an observable. An observed  $\Delta Q$  will calculate that based on the samples collected.

### 4.1.4 Operations

In a previous section we talked about the possible operations that can be performed on and between  $\Delta Q$ s, the time complexity of FTF, ATF and PC is trivially  $\mathcal{O}(N)$  where  $N$  is the number of bins. As to convolution, the naïve way of calculating convolution has a time complexity of  $\mathcal{O}(N^2)$ , this quickly becomes a problem as soon as the user wants to have a more fine-grained understanding of a component. Below we present two ways to perform convolution.

#### Convolution

**Naïve convolution** Given two  $\Delta Q$  binned PDFs  $f$  and  $g$ , the result of the convolution  $f \otimes g$  is given by

$$(f \otimes g)[n] = \sum_{m=0}^N = f[m]g[n - m] \quad (4.4)$$

**Fast Fourier Transform Convolution** FFTW (Fastest Fourier Transform in the West) is a C subroutine library [16] for computing the discrete Fourier Transform in one or more dimensions, of arbitrary input size, and of both real and complex data. We use FFTW in our program to compute the convolution of  $\Delta Q$ s.

Whilst the previous algorithm is far too slow to handle a high number of bins, convolution leveraging Fast Fourier Transform (FFT) allows us to reduce the amount of calculations

to  $\mathcal{O}(n \log n)$ .

FFT and naïve convolution produce the same results in our program barring  $\varepsilon$  differences (around  $10^{-18}$ ) in bins whose result should be 0.

### Arithmetical operations

We can apply a set of arithmetical operations between  $\Delta Q$ s ECDFs, and on a  $\Delta Q$ .

**Scaling (multiplication)** A  $\Delta Q$  can be scaled w.r.t a constant  $0 \leq j \leq 1$ . It is equal to binwise multiplication on ECDF bins.

$$\hat{f}_r(i) = \hat{f}(i) \cdot j \quad (4.5)$$

**Operations between  $\Delta Q$ s** Addition, subtraction and multiplication can be done between two  $\Delta Q$  of equal bin width (but not forcibly of equal length) by calculating the operation between the two ECDFs of the  $\Delta Q$ s:

$$\Delta Q_{AB}(i) = \hat{f}_A(i)[\cdot, +, -] \hat{f}_B(i) \quad (4.6)$$

### 4.1.5 Confidence bounds

To observe the stationarity of a system we must observe a window of  $\Delta Q$ s of an observable and calculate confidence bounds over said windows. We present here the formulae required to give such bounds with 95% confidence level. For a bin  $i$ , its mean over a window is:

$$\mu_i = \frac{1}{n_i} \sum_{j=1}^{n_i} x_{ij} \quad (4.7)$$

Where  $x_{ij}$  is a bin's  $i$  value for an ECDF  $j$ . Its variance:

$$\sigma_i^2 = \frac{1}{n_i} \sum_{j=1}^{n_i} x_{ij}^2 - \mu_i^2 \quad (4.8)$$

The confidence intervals  $CI_i$  for a bin  $i$  can then be calculated as:

$$CI_i = \mu_i \pm z_{\alpha/2} \cdot \frac{\sigma_i}{\sqrt{n_i}} \quad (4.9)$$

The bounds can be updated dynamically by inserting or removing a  $\Delta Q$ , this allows us to consider a small window of execution rather than observing the whole execution.

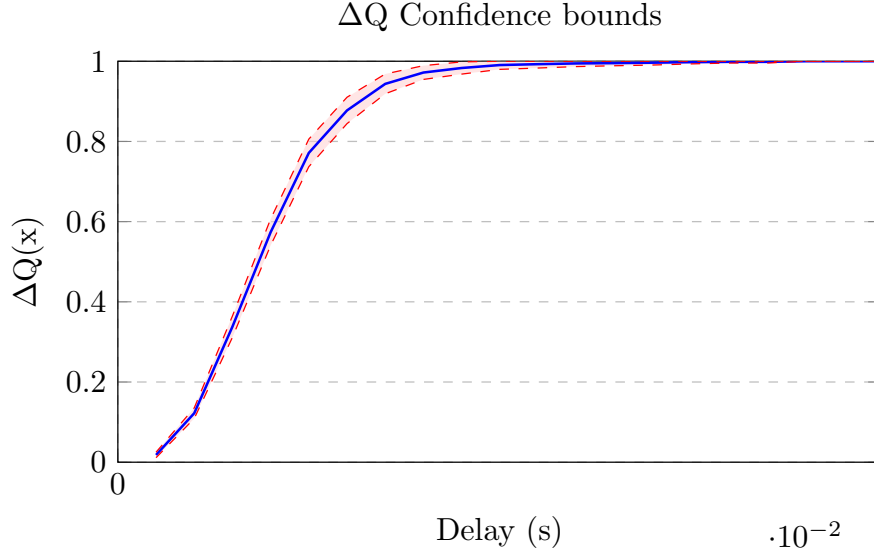


Figure 4.2: Upper and lower bounds of the mean of multiple  $\Delta Q$ s. In a system that behaves linearly, the bounds will be close to the mean, once the overload is approaching, or a system is showing behaviour that diverges from a linear one, the bounds will be larger.

#### 4.1.6 Rebinning

Rebinning refers to the aggregation of multiple bins of a bin width  $i$  to another bin width  $j$ . Operations between  $\Delta Q$ s can be done on  $\Delta Q$ s that have the same bin width, this is why it is fundamental that all observables have a common  $\Delta_{tbase}$ . This allows for fast rebinning to a common bin width.

Given two  $\Delta Q$ s  $\Delta Q_i$ ,  $\Delta Q_j$ :

$$\Delta_{Tij} = \max \{ \Delta_{Ti}, \Delta_{Tj} \}$$

and the PDF of the rebinned  $\Delta Q$  at bin  $b$ , from the original PDF of  $n$  bins, where  $k = \frac{\Delta_{Ti}}{\Delta_{Tj}}$ :

$$p'_b = \sum_{n=b \cdot k}^{b+1 \cdot k - 1} p_n, \quad b = 0, 1, \dots, \lceil \frac{N}{k} \rceil \quad (4.10)$$

We perform rebinning to a higher bin width for a simple reason, while this leads to loss of information for the bin with the lowest bin width, rebinning to a lower bin width would imply inventing new values for the  $\Delta Q$  with the highest bin width.

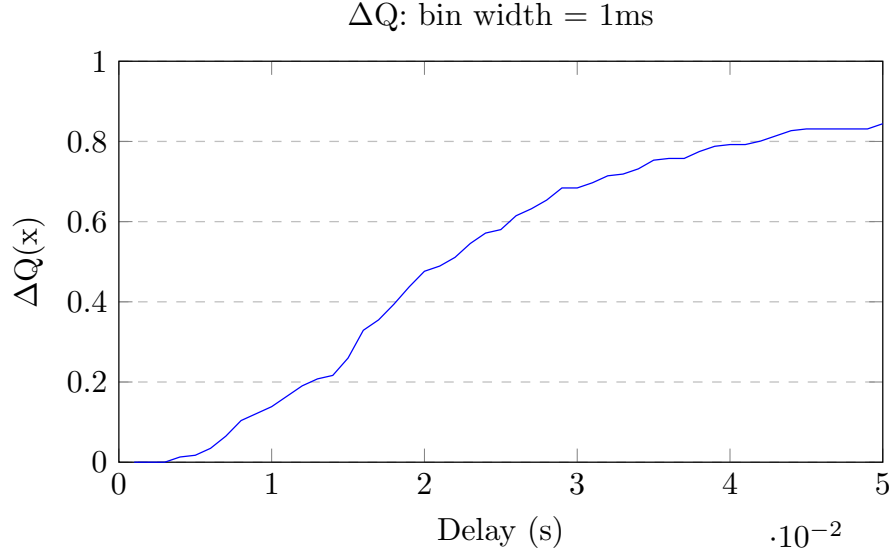


Figure 4.3: Sample  $\Delta Q$  with 1ms bins

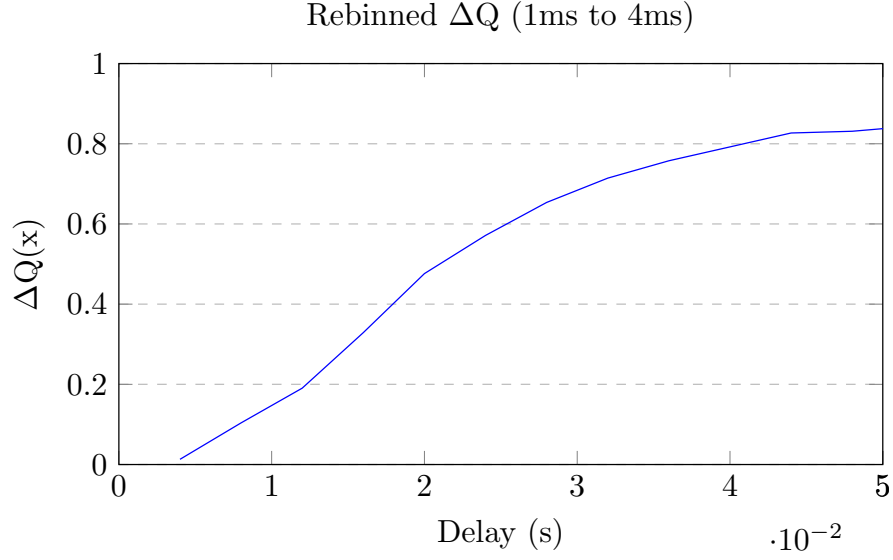


Figure 4.4: Previous  $\Delta Q$  after rebinning to 4ms bins

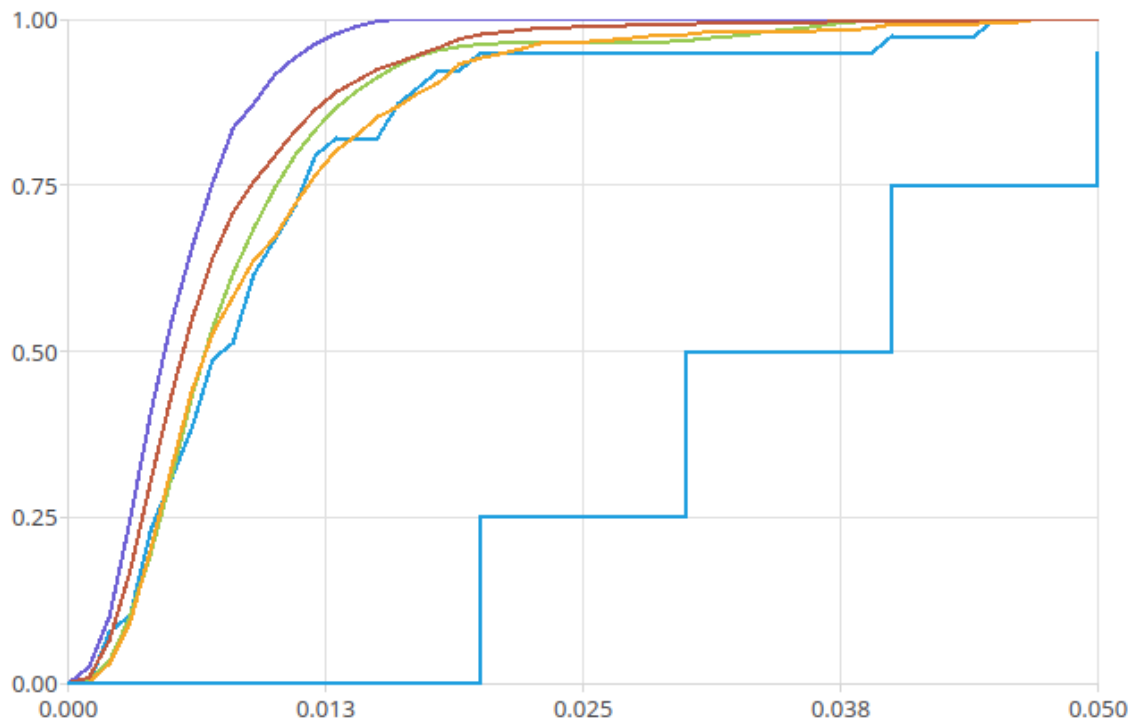
## 4.2 $\Delta Q$ display

An observable's displayed graph must contain the following functions:

- The mean and confidence bounds of a window of previous  $\Delta Q$ s
- The observed  $\Delta Q(t_l, t_u, dMax)$
- For a probe, the calculated  $\Delta Q$  from its components.
- Its QTA



This allows for the user to observe if a  $\Delta Q$  has deviated from normal execution, analyse its stationarity, nonlinearity and observe its execution.



## 4.3 Outcome diagram

An abstract syntax for outcome expressions and consequently outcome diagrams has already been defined in a previous paper [3], nevertheless, the oscilloscope provides additional features not included in the original syntax and, moreover, needs a textual way to define an outcome diagram.

We define thus a grammar to create an outcome diagram in our oscilloscope, our grammar is a textual interpretation of the abstract syntax.

### 4.3.1 Probes

To attach probes in the oscilloscope, the user must define outcomes and probes that observe outcome expressions.

#### Outcome

In our system an outcome is defined with its name

```
... = outcomeName;
```

### Probes containing outcome expressions

A probe can contain one component or a sequence of causally linked components. The user can define as many probes that contain outcome expressions as they want, they have to be declared as follows:

```
probe = component [-> component2];  
probe2 = newComponent -> anotherComponent;
```

These probes can be reused in other probes or in the system by adding "s:" (subsystem) before they are used.

```
probe3 = s:probe -> s:probe2;
```

### 4.3.2 Operators, outcome expressions

To build a system, we must define the relations between outcomes and outcome expressions, below is how they can be defined.

First-to-finish, all-to-finish and probabilistic choice must contain at least two components, this is because the operations to calculate the *calculated*  $\Delta Q$  rely on using the CDF of the components that define the operator.

#### Causal link

A causal link between two components can be defined by a right arrow from `component_i` to `component_j`

```
component_i -> component_j
```

#### All-to-finish operator

An all-to-finish operator needs to be defined as follows:

```
a:name(component1, component2...)
```

#### First-to-finish operator

A first-to-finish operator needs to be defined as follows.

```
f:name(component1, component2...)
```

#### Probabilistic choice operator

A probabilistic choice operator needs to be defined as follows:

```
p:name[probability_1, probability_2, ... probability_i](component_1,  
↪ component_2, ..., component_i)
```

In addition to being comma separated, the number of probabilities inside the brackets must match the number of components inside the parentheses. For  $n$  probabilities  $p_i$ ,  $0 < p_i < 1$ ,  $\sum_{i=0}^n p_i = 1$

### 4.3.3 Limitations

Our system has a few limitations compared to the theoretical applications of  $\Delta Q$ , namely, no cycles are allowed in the definition of a system.

```
probe = s:probe_2;  
probe_2 = s:probe;
```

The above example is not allowed and will raise an error when defined.

## 4.4 Dashboard

The dashboard is devised of multiple sections where the user can interact with the oscilloscope, create the system, observe the behaviour of its components, set triggers.

### 4.4.1 Sidebar

The sidebar has multiple tabs, we explain here the responsibility of each one.

#### System/Plots tab

**System creation** In this tab the user can create its system using the grammar defined before, he can save the text he used to define the system or load it, the system is saved to a file with any extension, we nevertheless define an extension to save the system to, the extension `.dq`. If the definition of the input is wrong, he will be warned with a pop up giving the error the parser generator encountered in the creation of a system.

**Adding a plot** Once the system is defined, the user can choose the probes he wants to plot. They can select multiple probes per plot and display multiple plots on the oscilloscope window.

#### Parameters tab

In this tab, the user can define parameters for the probes they have defined.

**Set a QTA** The user is given the choice to set a QTA for a given observable, they have 4 fields where they can fill in which correspond to the percentiles and the maximum amount of failures allowed, they can change this dynamically during execution.

**dMax, bins** The user has a slider which goes from -10 to 10, where they can set the parameters we explained previously,  $n$ , the exponent of  $\Delta_{tbase} \cdot 2^n$  and the bins  $N$ . When these informations are saved by the user, the new  $dMax$  is transmitted to the stub and saved for the selected observable.

#### Triggers tab

In the triggers tab the user can set triggers and observe the snapshots of the system.

**Set triggers** The user can set which triggers to fire for the probes they desire, they are given checkboxes to decide which ones to set as active or not (by default, the triggers are deactivated).

**Fired triggers** Once a trigger is fired, the system start a timer, during which all probes start recording the observed  $\Delta Q$ s (and the calculated ones if applicable) without discarding older ones. Once the timer expires, the snapshot is saved for the user in the triggers tab. In the dashboard, it indicates when the trigger was fired (timestamp) and the name of the probe which fired it.

#### 4.4.2 Stub controls

Below the sidebar, two buttons are present, these buttons communicate to the stub.

The **start stub** button sends a message to the stub, telling it to start sending spans. The **stop stub** button stops it.

# Chapter 5

## Oscilloscope: Low Level Implementation

### 5.1 Wrapper

The wrapper, called `otel_wrapper` is a rebar3 application built to replace OpenTelemetry calls and create custom spans, it is designed to be paired with the oscilloscope to observe an erlang application.

#### 5.1.1 API

The wrapper functions to be used by the user are made to replace OpenTelemetry calls to macros as for `?start_span` and `?with_span` and `?end_span`. This is to make the wrapper less of an encumbrance for the user.

Moreover, the wrapper will always start OpenTelemetry spans but only start custom spans if the stub has been activated. The wrapper can be activated by: WIP

`start_span/1, start_span/2`

```
start_span/1: -spec start_span(binary()) -> {opentelemetry:span_ctx(),  
  ↳ pid() | ignore}.
```

```
start_span/2: -spec start_span(binary(), map()) ->  
  ↳ {opentelemetry:span_ctx(), pid() | ignore}.
```

**Parameters:**

- Name: Binary name of the observable
- Attributes: The OpenTelemetry span attributes (Only for `start_span/2`)

`start_span` incorporates OpenTelemetry `?start_span(Name)` macro.

**Return:** The function returns either:

- {SpanCtx, span\_process\_PID} if the wrapper is active and the observable's *dMax* has been set
- {SpanCtx, ignore} if one of the two previous conditions was not respected.

With SpanCtx being the context of the span created by OpenTelemetry.

### **with\_span/1, with\_span/2**

```
with_span/1: -spec with_span(binary(), fun(() -> any())) -> any().
with_span/2: -spec with_span(binary(), map(), fun(() -> any())) ->
  ↳ any().
```

#### **Parameters:**

- Name: Binary name of the observable
- Fun: Zero-arity function representing the code of block that should run inside the ?with\_span macro
- Attributes: The OpenTelemetry span attributes (Only for with\_span/3)

with\_span incorporates OpenTelemetry with\_span macro.

**Return:** with\_span returns what Fun returns (any()).

### **end\_span**

```
-spec end_span(opentelemetry:span_ctx(), pid() | ignore) -> ok |
  ↳ term().
```

#### **Parameters:**

- SpanCtx: The context of the span returned by start\_span.
- Pid: span\_process\_PID || ignore

As is the case for start\_span, end\_span incorporates an OpenTelemetry macro, in this case ?end\_span(Ctx).

### **fail\_span**

```
-spec fail_span(pid() | ignore) -> ok | term().
```

#### **Parameter:**

- Pid: ignore || span\_process\_PID

fail\_span does not incorporate any OpenTelemetry macro, it is let up to the user to decide how to handle failures in execution.

## `span_process`

`span_process` is the process, spawned by `start_span`, responsible for handling the `end_span`, `fail_span`, `timeout` messages.

Upon being spawned, the process starts a timer with time equal to the `dMax` set by an user for the observable being observed, thanks to `erlang:send_after`. when the timer runs out, it sends a `timeout` message to the process. The process can receive three kinds of messages:

- `{end_span, end_time}`: This will send a custom span to the oscilloscope with the start and end time of the execution of the observable.
- `fail_span`: This will send a custom span to the oscilloscope indicating that an execution of an observable has failed.
- `timeout`: If the program hasn't ended the span before `dMax`, the timer will send a `timeout` message and it will send a custom span to the oscilloscope indicating that an execution of an observable has taken  $> dMax$ .

The process is able to receive one and only message, if the execution times out and subsequently the span is ended, the oscilloscope will not be notified as the process is defunct. This is assured by Erlang documentation:

*If the message signal was sent using a process alias that is no longer active, the message signal will be dropped.*

## 5.1.2 Handling outcome instances

To create custom spans of the outcome instances we must obtain three important informations:

- The name of the observed outcome or probe
- The time when the span was started
- The `dMax` of the observed

They are all supplied upon starting a span with `otel_wrapper:[start_span, with_span]` by calling

```
StartTime = erlang:system_time(nanosecond),
```

The custom span is created only if two conditions are met: the wrapper has been set as active and the user set a timeout for the name of the observable, the functions will spawn a `span_process` process, passing along all the necessary informations.

Once the span is subsequently ended/timed out/failed, the function `send_span` creates a message carrying all the informations and sends it to the C++ server. The formatting of the messages is the following:

```
n:Observed name, b: Start time (beginning), e: End time (if end_span  
↪ was received), s: The status
```

### 5.1.3 TCP connection

The wrapper is composed of two `gen_server` which handle communication to and from the oscilloscope. This `gen_server` behaviour allows the wrapper to send spans asynchronously to the oscilloscope.

#### TCP server

The TCP server listens by default on localhost at port 8081, the user can define a port and ip to listen at.

The oscilloscope can send commands to the stub, these commands are:

- `start_stub`: This command sets the stub as active, it can now send spans to the oscilloscope if the items are defined.
- `stop_stub`: This commands sets the stub as inactive, it will no longer send spans to the oscilloscope
- `set_timeout;probeName;timeout`: This command indicates to the stub to set the *dMax* for a probe to timeOut, a limit of the stub is that `erlang:send_after` does not accept floats as timeouts, so the timeout will be rounded to the nearest integer

#### TCP client

The TCP client allows the stub to send the spans to the oscilloscope, by default the oscilloscope is on localhost:8080, but that can be changed by the user.

The client connects over TCP to the oscilloscope and opens a socket where it can send the spans.

## 5.2 Parser

To parse the system, we use the C++ ANTLR4 (ANother Tool for Language Recognition) library.

### 5.2.1 ANTLR

ANTLR is a parser generator for reading, processing, executing or translating structured text files. ANTLR generates a parser that can build and walk parse trees [17].

ANTLR is just one of the many parsers generators available in C++ (`flex/bison`, `lex`, `yacc`), although it presents certain limitations, its generated code is simpler to handle and less convoluted with respect to the other possibilities.

ANTLR uses Adaptive LL(\*) (*ALL(\*)*) parser, namely, it will move grammar analysis to parse-time, without the use of static grammar analysis. [18]



## 5.2.2 Grammar

ANTLR provides a yacc-like metalanguage to write grammars. Below, is the grammar for our system: [cite]

```
grammar DQGrammar;

PROBE_ID: 's';
BEHAVIOR_TYPE: 'f' | 'a' | 'p';
NUMBER: [0-9]+('.'[0-9]+)?;
IDENTIFIER: [a-zA-Z_][a-zA-Z0-9_]*;
WS: [ \t\r\n]+ -> skip;

// Parser Rules
start: definition* system? EOF;

definition: IDENTIFIER '=' component_chain ' ';
system: 'system' '=' component_chain ' ';

component_chain
    : component ('->' component)*
    ;

component
    : behaviorComponent
    | probeComponent
    | outcome
    ;

behaviorComponent
    : BEHAVIOR_TYPE ':' IDENTIFIER ('[' probability_list ']')? '('
      ↪ component_list ')'
    ;

probeComponent
    : PROBE_ID ':' IDENTIFIER
    ;

probability_list: NUMBER (',' NUMBER)+;
component_list: component_chain (',' component_chain)+;

outcome: IDENTIFIER;
```

## Limitations

A previous version was implemented in Lark[cite], a python parsing toolkit. The python version was quickly discarded due to a more complicated integration between Python and C++. Lark provided Earley(SPPF) strategy which allowed for ambiguities to be

resolved, which is not possible in ANTLR.

For example the following system definition presents a few errors:

```
probe = s -> a -> f -> p;
```

While Lark could correctly guess that everything inside was an outcome, ANTLR expects ":" after "s, a, f" and "p", thus, one can not name an outcome by these characters, as the parsers generator thinks that an operator or a probe will be next.

## 5.3 Oscilloscope GUI

Our oscilloscope graphical interface has been built using the Qt framework for C++. Qt is a cross-platform application development framework for creating graphical user interfaces.

We chose Qt as we believe that it is the most documented and practical library for GUI development in C++, using Qt allows us to create usable interfaces quickly, while being able to easily pair the backend code of C++ to the frontend.

# Chapter 6

## Application on synthetic programs

### 6.1 M/M/1/K queue

We devised a simple Erlang system that can act as a simple M/M/1/K.

The system has two components `worker_1`, `worker_2`, the components are made of a buffer queue of size  $K$  and a worker process.

The system sends  $n$  messages per second following a Poisson distribution to `worker_1`'s queue, the queue then reduces its available buffer size.

The buffer notifies its worker, which then does  $N$  loops, which are defined upon start, of fictional work. The worker then passes a message to `worker_2`'s queue, which has another queue of same size, who passes the message to `worker_2`'s worker, which does the same amount of loops. When a worker completes its work, it notifies the queue, freeing one "message" from its buffer size.

If the queue's buffer is overloaded, it will drop the incoming message and consider the execution a failure.

A probe  $p$  is defined, which observes the execution from when the first message up until `worker_2` is done.

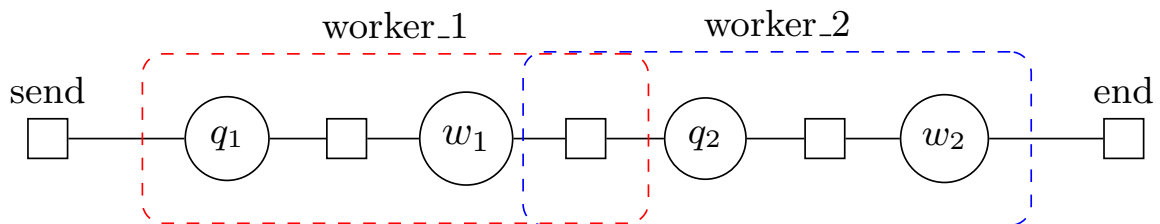


Figure 6.1: Outcome diagram of the M/M/1/K queue

At low load, we can observe how worker\_1 and worker\_2 mean  $\Delta Q$ s overlap and the same can be said for

# Chapter 7

## Performance study

### 7.1 Convolution performance

We implemented two versions of the convolution algorithm as described before, the naïve version and the FFT version. We compared their performance when performing convolution on two  $\Delta Q$ s of equal bins.

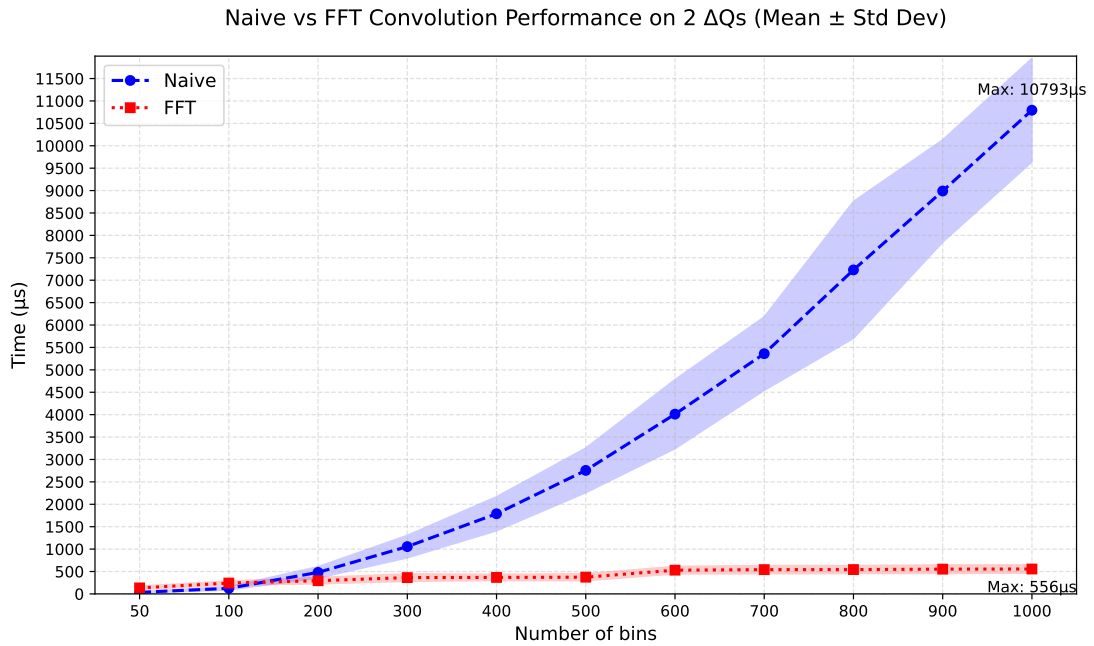


Figure 7.1: Performance comparison of two convolution algorithms

As expected, the naïve version has a time complexity of  $\mathcal{O}(n^2)$  and quickly scales with the number of bins, this is clearly inefficient, as a more precise  $\Delta Q$  will result in a much slower program.

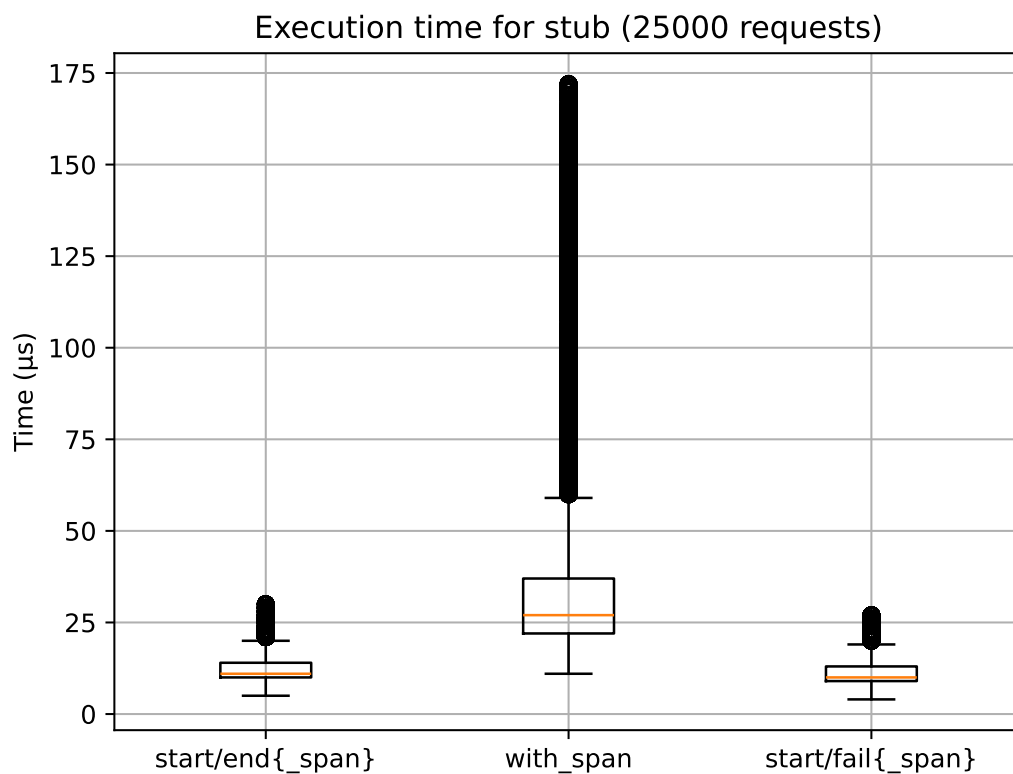
As for the FFT algorithm, it is slightly slower when the number of bins is lower than 100. This is due to the FFTW3 routine having slightly higher overhead.

## 7.2 Stub performance

We evaluated the performance of the stub to measure its impact in a normal execution, namely we tested the following calls which represent a normal usage of the wrapper.

- `start_span`  $\rightarrow$  `end_span`.
- `with_span` with the following function: `fun()`  $\rightarrow$  `ok`.
- `start_span`  $\rightarrow$  `fail_span`.

We ran the simulation for 25000 subsequent iterations, these are the results.



# Bibliography

- [1] Peter Van Roy and Seyed Hossein Haeri. *The  $\Delta QSD$  Paradigm: Designing Systems with Predictable Performance at High Load. Full-day tutorial*. 15th ACM/SPEC International Conference on Performance Engineering. 2024. URL: <https://webperso.info.ucl.ac.be/~pvr/ICPE-2024-deltaQSD-full.pdf>.
- [2] Seyed H. Haeri et al. “Mind Your Outcomes: The  $\Delta QSD$  Paradigm for Quality-Centric Systems Development and Its Application to a Blockchain Case Study”. In: *Comput.* 11.3 (2022), p. 45. DOI: 10.3390/COMPUTERS11030045. URL: <https://doi.org/10.3390/computers11030045>.
- [3] Seyed Hossein Haeri et al. “Algebraic Reasoning About Timeliness”. In: *Proceedings 16th Interaction and Concurrency Experience, ICE 2023, Lisbon, Portugal, 19th June 2023*. Ed. by Clément Aubert et al. Vol. 383. EPTCS. 2023, pp. 35–54. DOI: 10.4204/EPTCS.383.3. URL: <https://doi.org/10.4204/EPTCS.383.3>.
- [4] Peter Van Roy. *LINFO2345 lessons on  $\Delta QSD$* . Accessed: (19/05/2025). UCLouvain, 2023. URL: <https://www.youtube.com/watch?v=tF7fbU9Gce8>.
- [5] Davies N. and Thompson P.  *$\Delta QSD$  workbench - GitHub*. Accessed: (19/05/2025). 2022. URL: <https://github.com/DeltaQ-SD/dqsd-workbench>.
- [6] Erlang programming language. *Erlang tracing*. Accessed: (19/05/2025). 2024. URL: <https://www.erlang.org/doc/apps/erts/tracing.html>.
- [7] OpenTelemetry. *OpenTelemetry in Erlang/Elixir*. Accessed: (19/05/2025). 2025. URL: <https://opentelemetry.io/docs/languages/erlang/>.
- [8] OpenTelemetry. *What is OpenTelemetry?* Accessed: (19/05/2025). 2025. URL: <https://opentelemetry.io/docs/what-is-opentelemetry/>.
- [9] OpenTelemetry. *OpenTelemetry - Traces*. Accessed: (19/05/2025). 2025. URL: <https://opentelemetry.io/docs/concepts/signals/traces/>.
- [10] The Jaeger Authors. *Jaeger*. Accessed: (19/05/2025). 2025. URL: <https://www.jaegertracing.io/>.
- [11] Dotan Horovits. *From Distributed Tracing to APM: Taking OpenTelemetry & Jaeger Up a Level*. Accessed: (19/05/2025). 2021. URL: <https://logz.io/blog/monitoring-microservices-opentelemetry-jaeger/>.
- [12] Sampath Siva Kumar Boddeti. *Tracing Made Easy: A Beginner’s Guide to Jaeger and Distributed Systems*. Accessed: (19/05/2025). 2024. URL: <https://openobserve.ai/blog/tracing-made-easy-a-beginners-guide-to-jaeger-and-distributed-systems/>.
- [13] OpenTelemetry. *Instrumentation for OpenTelemetry Erlang/Elixir*. Accessed: (19/05/2025). 2025. URL: <https://opentelemetry.io/docs/languages/erlang/instrumentation/>.

- [14] OpenTelemetry. *Active spans, C++ Instrumentation*. Accessed: (19/05/2025). 2025. URL: <https://opentelemetry.io/docs/languages/cpp/instrumentation/>.
- [15] Peter Thompson and Rudy Hernandez. *Quality Attenuation Measurement Architecture and Requirements*. Tech. rep. MSU-CSE-06-2. Sept. 2020. URL: <https://www.broadband-forum.org/pdfs/tr-452.1-1-0-0.pdf>.
- [16] FFTW3. *Fastest Fourier Transform in The West*. Accessed: (19/05/2025). 2025. URL: <https://www.fftw.org/>.
- [17] ANTLR. *What is ANTLR4?* Accessed: (19/05/2025). 2025. URL: <https://www.antlr.org/>.
- [18] Terence Parr and Kathleen Fisher. “LL(\*): the foundation of the ANTLR parser generator”. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 425–436. DOI: 10.1145/1993498.1993548. URL: <https://doi.org/10.1145/1993498.1993548>.



# Appendix A

## User manual

### A.1 Sidebar: Probes settings

Now that you have create your outcome diagram, you can modify the probes settings to set the  $dMax$  you want and the QTA you desire.

#### A.1.1 Setting a QTA

You can set a QTA in the "**Set a QTA**" section, there you are presented with the possibility to:

- Select the probe for which you want to set the QTA.
- Set the QTA at the three percentiles (25%, 50%, 75%), the text to the left indicates which percentile is which. You need to set the delay in seconds.
- The minimum amount of successful events you can allow, which is bigger than 75%.

Of course, for the delay of the QTA at three percentiles, the delay at the percentile must be higher or equal than the delay at the previous percentile and higher than 0.

By pressing "*Save QTA settings*" you will save the QTA for the defined probe.

#### A.1.2 Setting the parameters of a probe

You can set the parameters for a probe in its section. To the left you can select the probe you want to set the parameter for. We provide a slider (which goes from -10 to 10) for the  $n$  parameters, to the right of the slider, you can select the number of bins for the probe. The maximum delay calculated will be shown below.

Once you press "**Save delay**", a message to the erlang wrapper will be sent, which will set the maximum delay you have set in the wrapper.

Sidebar

Probes settings

Triggers

Set QTA for a probe:

Probe:

probe

25th Percentile (s):

0.1

50th Percentile (s):

0.25

75th Percentile (s):

0.5

Max. allowed failure (0-1):

0.95

Save QTA Settings

Set the parameters for a probe.

probe

1000

Max delay is: 62.50 ms

Save delay

Figure A.1: Probes settings tab. Above: QTA settings. Below: Probe parameters settings.

## A.2 Sidebar: Outcome diagram and plots

### A.2.1 Creating the system

You need to provide a textual description of your outcome diagram following the syntax which was defined previously. You can create your system by clicking on the **Create or edit system** button. If the parser successfully parsed the text, your system will be created and you can start setting the parameters for your probes.

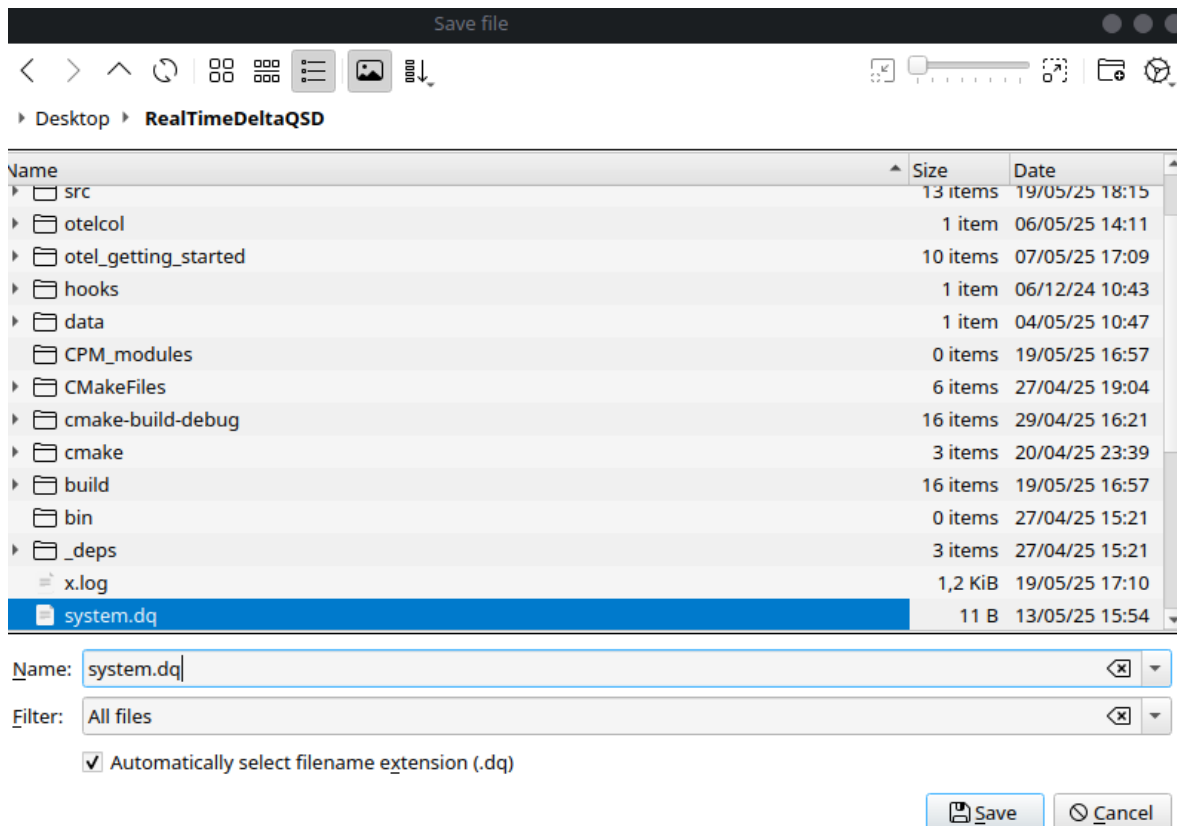
If the parser does not correctly parse the text, it will show a popup indicating the line where the error was produced and what it was expecting.

### A.2.2 Saving the system definition

The button **Save system to** gives you the possibility to save the textual definition of the outcome diagram to a file, you can save the file to any extension, but preferably the file will have the **.dq** extension.

### A.2.3 Loading the system definition

The button **Load system to** gives you the possibility to load the textual definition of the outcome diagram you may have previously saved to a file, as for saving, you can load any file extension.



### A.2.4 Managing the plots

Once you have your system defined you can start adding plots of the  $\Delta Q$ s of the probes you inserted in your system.

#### Adding a plot

Multiple probes can be added at once in your plot, you can select the probes you want to add to a new plot by selecting them in the "**Add a new plot**" section. Then by clicking the "**Add plot**" button, the selected probes will be added to the plot.

#### Editing a plot

You can remove the probes you have added to the plot by first clicking to it, this sets the plot as the **selected plot**. Once you have clicked the plot, a section will pop up beneath the rest of the controls on the sidebar.

In the section there are two subsections, one which shows the selected components which form the plot (those you have added previously), and the available components. You can select the probes you want to remove in the "**selected components**" zone, by clicking "**Remove selected components**" the components will be removed from the plot. Inversely, in the "**Available components**" section, you can select the plots you want to add, and by clicking "**Add selected components**" you can add the selected components to the selected plot.

### A.2.5 Removing a plot

By left-clicking the plot, a popup appears, you can click "**Remove plot**", this removes it from the selected plot.

## A.3 Triggers

In the triggers section you can define which triggers to apply for a given probe, once selected, they will be automatically activated.

Once a trigger is triggered, the oscilloscope will keep recording the system for a few seconds. Once stopped, under the "snapshots" section, you can click the snapshots to view them in a separate section, there, you can observe the  $\Delta Q$ s of all the probes before and after the trigger was triggered. You can discard the snapshot by left clicking on it and clicking "*Deleted snapshot*".

UNIVERSITÉ CATHOLIQUE DE LOUVAIN  
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | [www.uclouvain.be/epl](http://www.uclouvain.be/epl)