École polytechnique de Louvain

The $\triangle$Q Oscilloscope: Real-Time Observation of Large Erlang Applications using $\triangle$QSD

Author: **Francesco Nieri**
Supervisor: **Peter Van Roy**
Readers: **Neil Davies, Tom Barbette, Peer Stritzinger**
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. Many important questions are hard to answer. What happens when there is an overload? How can we feel something is wrong with the system early enough?

The purpose of the thesis is to create the ΔQ oscilloscope, a real time graphical dashboard that gives insights into a running Erlang system. It can be used to study the behaviour of running systems and explore tradeoffs in system design. It is based on the principles of ΔQSD.

Furthermore, we have developed an interface, called ΔQ adapter, named `dqsd_otel`. It allows for the running system to communicate with the oscilloscope to receive real time insights about the execution of the former. The adapter includes the OpenTelemetry framework to extend current observability tools functionalities.

The oscilloscope performs statistical computations on the time series data it receives and displays the results in real time, thanks to the ΔQSD paradigm. We provide a set of triggers to capture rare events, like an oscilloscope would, and give a snapshot of the system under observation as if it was frozen in time. An implementation of a textual syntax allows the creation of outcome diagrams which give an "observational view" of the system. Furthermore, the implementation of efficient algorithms for complex operations, such as convolution, allows for the computations to be done rapidly on precise representations of components.

We introduce the work by giving a summary of ΔQSD concepts, which have been extended to allow the instrumentation of applications written in Erlang. We also provide a summary of the observability tools for Erlang, namely, Opentelemetry. Subsequently, we explain the user level concepts which are essential to understand how the oscilloscope works and understand what is displayed on the screen, delving later on into the mathematical foundations of the concepts. Lastly, we provide synthetic applications which prove the soundness of ΔQSD and show how the oscilloscope is able to detect problems in a running system, diagnose it and explore design tradeoffs.

Acknowledgments

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

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

My **friends**, to those who have come from Italy and have taken time out of their lives to listen the thesis presentation, and to those who through the years have been there for me.

A-M., for the moments we shared these four years together in uni.

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

Peer Stritzinger, **Stritzinger GmbH** and the **EEF Observability Working Group** (Bryan Naegele and Tristan Sloughter) for their help in the EEF Slack, which helped me understand OpenTelemetry and gave me the send_after intuition.

Neil Davies for taking the time to proofread my thesis.

The **PNSol Ltd.** team for their extensive groundwork on **ΔQSD** and its dissemination, which made this thesis possible.

Lastly, **Peter Van Roy**, for his year-long relentless interest, support and weekly and constant supervision which made sure the project would come to fruition.

AI disclaimer

AI was employed to help with the graphical dashboard in C++ and the triggers, in positioning the elements, refactoring the code so the widgets would properly interact together, helping understand the FFT algorithm and refactoring the server when communication errors occurred. For the dashboard, 25% of ELOC are **refactored** by AI, they are the constructors of the widgets which nicely place the widgets on screen. The ANTLR CMake was provided by ChatGPT. In total, of around 6000 ELOC, around 10 to 15% has been done or refactored by AI, this is mostly composed of the server and dashboard/trigger code. Comments were generated by ChatGPT and reviewed so they would reflect actual code.

In Erlang, it was used to provide documentation and help with TCP communication exceptions. To give an estimate, around 20% of 350 ELOC are done or refactored by AI and they mostly relate to TCP communication and errors handling. Comments were generated and restructured to present the tools nicely.

The **written master thesis** was written **entirely** without the aid of AI.

Contents

Chapter 1

Introduction

1.1 Context

ΔQSD 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]. Moreover, it is the basis of Broadband forum's TR452 standard series, used in instrumenting data networks [2].

ΔQSD 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 [1].

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 ΔQSD paradigm addresses this problem! [3]

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 with ΔQSD! This is where the **ΔQ oscilloscope** comes in.

1.2 Objective

This project will develop a practical tool, the **ΔQ 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 [4]. The tool will provide useful information for these applications both for understanding their behaviour, for diagnosing performance issues, and for optimising performance over their lifetime. [5]

The ΔQ 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 ΔQSD paradigm is not only a theoretical paradigm, but it can be employed in a real-time tool to diagnose distributed systems. Its application can then be further extended to large systems once the oscilloscope is refined.

The oscilloscope targets large distributed applications handling many independent tasks where performance and reliability are important. [1]

1.3 Previous work

The ΔQSD paradigm has been formalised across different papers [3] [6] and was brought to the attention of engineers via tutorials [1] and to students at Université Catholique de Louvain. [7]

A Jupyter notebook workbench has been made available on GitHub [8], it shows real time ΔQ 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 ΔQSD paradigm can be applied to real life examples.

Observability tools such as Erlang tracing [9] and OpenTelemetry [10] lack the notions of failure as defined in ΔQSD, 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 ΔQSD paradigm.

1.4 Contributions

We make the following principal contributions in the master thesis:

- The ΔQ oscilloscope, from design to implementation, to plot real-time ΔQ graphs. The oscilloscope contributions include:
  - A graphical interface in Qt.
  - The underlying implementation of ΔQSD concepts, including a textual syntax to create oucome diagrams derived from the original algebraic syntax.
  - Efficient convolution algorithms.
  - A system of triggers to catch rare events when system behaviour fails to meet quality requirements.
- The ΔQ adapter to communicate from the Erlang application to the oscilloscope.

- The evaluation of the effectiveness of the oscilloscope on synthetic applications.
- The evaluation of the efficiency of the basic operations regarding the oscilloscope, convolution, graphing and the adapter overhead.

These contributions can show that the ΔQSD paradigm can be translated from a posteriori analysis to real-time observation of running system. Furthermore, it reinforces the validity of the paradigm.

1.5 Roadmap

This thesis gives the reader everything that is needed to use the oscilloscope and exploit it to its full potential.

We divided the thesis in multiple chapters:

- Chapter 2 gives the reader a background of the theoretical foundations of ΔQSD, which are the basis of the oscilloscope and are fundamental to understand what is shown in the oscilloscope. Secondly, an introduction to OpenTelemetry, the framework our Erlang adapter is built on top of. Lastly, we provide what we believe are the current limitations of the observability tool and how we plan to tackle them.
- Chapter 3 first provides the "measurement concepts". These concepts serve as an introduction to understand the following chapters and as a bridge from OpenTelemetry to the oscilloscope. We then delve on how the different parts of our design interact together and how to correctly apply the concepts we introduced. Lastly, after having introduced the oscilloscope, we explain abstract concepts implemented in it, like triggers and sliding windows.
- Chapter 4 & 5 present the oscilloscope. First providing "user level concepts" of how ΔQSD is used and what the user should expect visually from the dashboard. Chapter 4 also provides a complete explanation on how to write outcome diagrams and what the different sections on the dashboard do. Secondly, a more low level explanation, which goes into more technical details of the parts that compose the oscilloscope and the mathematical explanations of ΔQSD concepts explained in the previous chapter.
- Chapter 6 provides synthetic applications which have been tested with the oscilloscope that demonstrate the usefulness of the oscilloscope in a distributed setting. In Chapter 7 we perform evaluations of the performance of the different parts we have developed to understand the overhead that are present.

Chapter 8 provides future possibilities which can be explored to improve the application. In the appendix, we provide a user manual to help users use the oscilloscope, along with C++ and Erlang source code of the oscilloscope and the adapter. The oscilloscope (`https://github.com/fnieri/DeltaQOscilloscope`) and adapter(`https://github.com/fnieri/dqsd_otel`) can be found on GitHub as open source projects.

Chapter 2

Background

This chapter aims to provide firstly a complete background of the concepts key to understanding the ΔQSD paradigm.

Secondly, we provide a comprehensive background into the observability solutions that have been explored for the oscilloscope, delving deeper into OpenTelemetry and its macros.

We finish by explaining what we believe are the current limitations of OpenTelemetry and explaining where the paradigm and the oscilloscope comes in.

2.1 An overview of ΔQSD

ΔQSD is "a metrics-based, quality-centric paradigm that uses formalised outcome diagrams to explore the performance consequences of design decisions". [6]

The key concepts that give the paradigm its name are **quality attenuation (ΔQ)** and **outcome diagrams**. [1]

The dependency and causality properties of a system can be captured by outcome diagrams, while the probability distribution representation can precisely model a system's behaviour. [6]

The following sections are a summary of multiple articles and presentations formalising the paradigm.

2.1.1 Outcome

To build outcome diagrams, we need to first introduce outcomes.

Outcomes represent system behaviours that can start at some point in time and ***may*** be observed to complete at some later time. [2] The result produced by performing a system's task is mapped to an outcome. [6]

The particularity of outcomes is that they can represent multiple levels of granularity. Suppose an outcome is beyond the current system's control (e.g. a database/cloud

request), is non-atomic (can be broken down in multiple sub-outcomes). These outcomes can be represented as black boxes (you can observe their start and end, but do not know what is being executed). As the system gets refined, these outcomes can then be refined to model a single outcome or multiple outcomes, if needed.
Even though these outcomes are defined as “black boxes”, they still have timeliness constraints like any other outcome. [6]

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. [3]

Outcome instance An outcome instance is the result of an execution of an outcome given a starting event e_{in} and an end event e_{out}. [3]

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

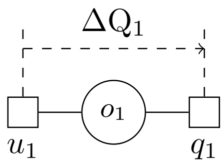


Figure 2.1: The outcome (circle) and the starting set (left) and terminating set (right) of events. [6]

2.1.2 Quality attenuation (ΔQ)

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]

ΔQ 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” (ΔQ) and can depend on many factors (geographical, physical . . .). Its distribution may be modelled by a random variable.

As ΔQ 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 finite or bounded delay < 1.

ΔQ(x) is the probability that an outcome O occurs in time $t \leq x$. The ***intangible mass*** $1 - \lim_{x\to\infty} \Delta Q(x)$ of a ΔQ will encode the probability of failure/timeout/exception occurring [6]. ΔQ is a CDF, sometimes we use its derivative, which is the probability density function (PDF).

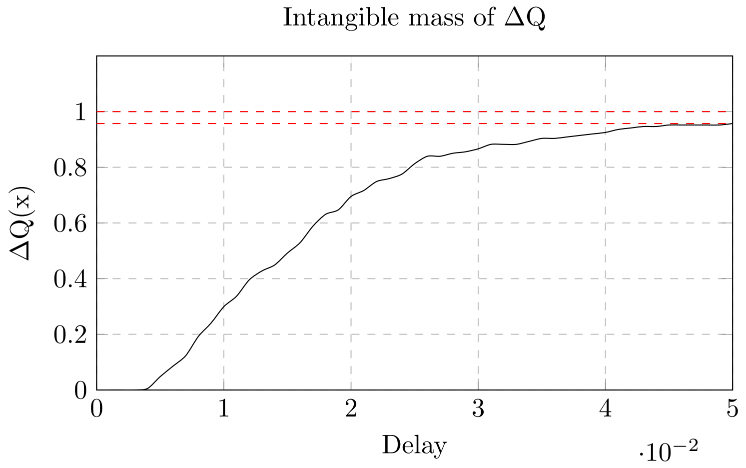


Figure 2.2: ΔQ CDF plot with intangible mass (red). The failure rate is about 5%

2.1.3 Failure semantics

In the CDF representation of a ΔQ, 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}". [3]

2.1.4 Partial ordering

A CDF of a ΔQ is *less than* the other if its CDF is everywhere to the left and above the other. It is represented mathematically as a partial order.

If two ΔQs intersect, they are not ordered. [1]

2.1.5 Timeliness

Timeliness is delivering results within required time bounds (sufficiently often). Timeliness is defined as "a relation between an observed ΔQ_{obs} and a required ΔQ_{req}". In ΔQSD, a system *satisfies timeliness* if $\Delta\mathrm{Q}_{obs} \leq \Delta\mathrm{Q}_{req}$. [3]

2.1.6 QTA, required ΔQ

The *Quantitative Timeliness Agreement* (QTA) maps objective measurements to the subjective perception of application performance. It specifies what the base system does and its limits. [2]

Slack There is performance *slack* when a ΔQ is strictly less than the requirement,

Hazard There is performance *hazard* when an observed ΔQ_{obs} intersects or is strictly greater than the required ΔQ_{req} ($\Delta Q_{obs} \nless \Delta Q_{req}$). [6]

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.

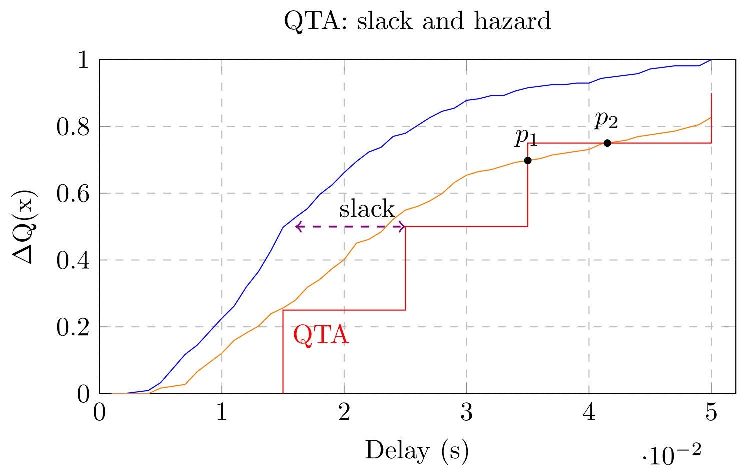


Figure 2.3: The system in blue is showing slack and satisfies the requirement. The system in orange is showing signs that it cannot handle the stress, it is not respecting the system requirements imposed by the QTA.

2.1.7 Outcome diagram

An outcome diagram is central to capture the causal relationships between the outcomes. It shows the causal connections between all the outcomes we are interested in, and it allows computing the ΔQ for the whole system [1]. It maps a system's behaviour as seen from outside to concrete outcomes [3]. There are four different operators that represent the relationships between outcomes.

Sequential composition

If we assume two outcomes O_A, O_B where the end event of O_A is the start event of O_B, then we say the two outcomes are sequentially composed. The total delay ΔQ_{AB} is given by the convolution of the PDFs of O_A and O_B ($O_A \circledast O_B$).

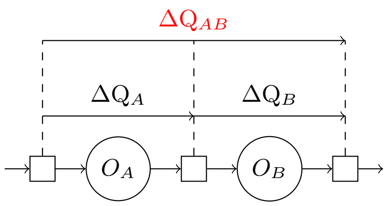


Figure 2.4: Sequential composition of O_A and O_B.

Where convolution ($\circledast$) between two PDF is:

$$PDF_A \circledast PDF_B(t) = PDF_{AB}(t) = \int\limits_0^t PDF_A(\delta) \cdot PDF_B(t-\delta) d\delta \tag{2.1}$$

Thus, ΔQ_{AB}:

$$\Delta Q_{AB} = \int_0^t PDF_A \circledast PDF_B d\delta \tag{2.2}$$

Convolution is the only operation which is based on the PDFs, the following operations are based on the CDF of the ΔQs (hence the use of the ΔQ notation).

First to finish (FTF)

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:

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

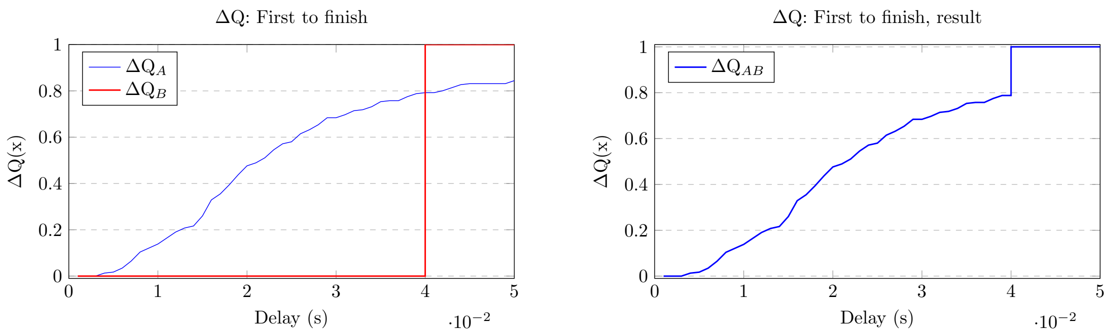


Figure 2.5: Left: $\Delta Q_{(A,B)}$. Right: $\text{FTF}_{(A,B)} = \Delta Q_{AB}$

All to finish (ATF)

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:

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

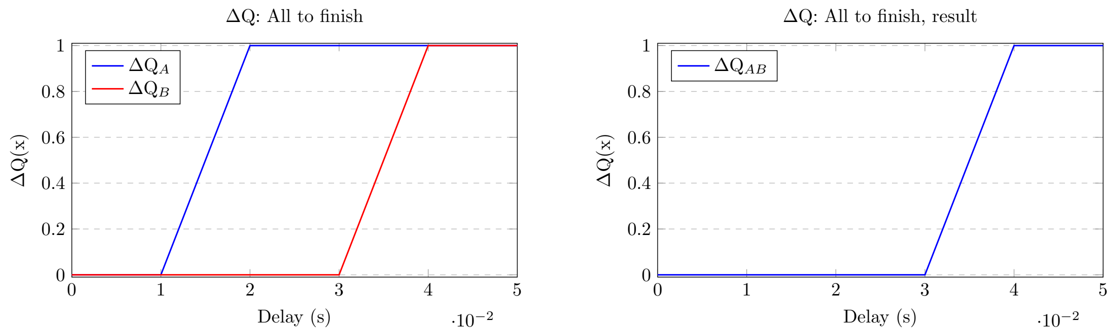


Figure 2.6: Left: $\Delta\mathrm{Q}_{(A,B)}$. Right: $\mathrm{ATF}_{(A,B)} = \Delta\mathrm{Q}_{AB}$

Probabilistic choice (PC)

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

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

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

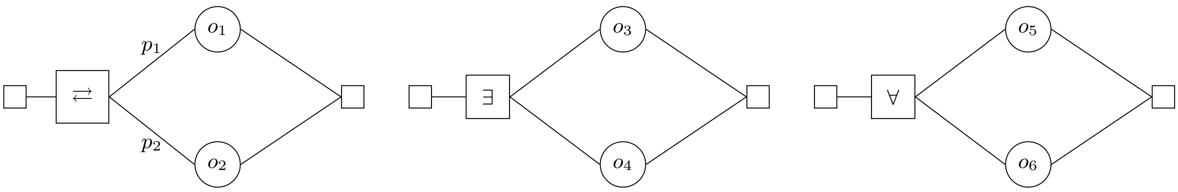


Figure 2.7: The possible operators in an outcome diagram: Probabilistic choice, first-to-finish, all-to-finish

First-to-finish, All-to-finish and probabilistic-choice are calculated on the CDF of the ΔQ of their components.

These operators can be assembled together to create an outcome diagram, later on, we will see how one can go from the graphical representation to outcome diagrams which can be used in the ΔQ oscilloscope.

2.1.8 Outcome diagrams refinement

An important feature of outcome diagrams is the ability to be able to design a system even with *"black boxes"*, before the complete details of it are known. [6]
An outcome diagram can be "unboxed" by refining the outcomes that compose it. We can adapt a situation described by the previously cited article, "Mind your Outcomes", to understand how refinement can allow the user to have a very precise representation of a system.

We first start with a black box, unnamed outcome with start event A and end event Z, somewhere in the system. The first refinement step would be giving the outcome a name.

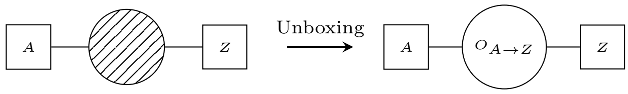


Figure 2.8: Refinement from black box to named outcome.

The system can be further refined by adding outcomes that represent tasks. For example, the engineer might believe that it will take two tasks to get from A to Z. We can then add another outcome, sequentially composed, to represent this situation.

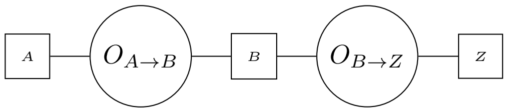


Figure 2.9: Further refinement from one task to two tasks.

We can also model the chance of executing two tasks as a probabilistic choice, where there is p_2 probability that the execution from A to Z will execute two tasks. The outcome diagram can be refined as a probabilistic choice.

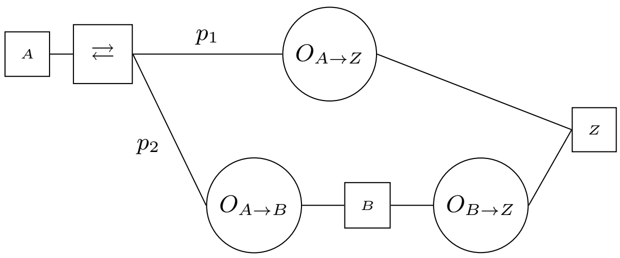


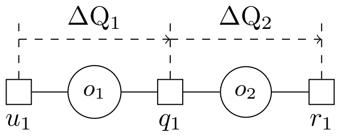
Figure 2.10: Refinement as probabilistic choice of executing either one or two tasks.

In essence, the refinement could model a very fine-grained representation of the system by further refining the system, to represent the possibility of executing n tasks. This demonstrates the power of outcome diagrams to represent system diagrams with high precision. They can help explore design decisions thanks to outcomes and operators.

2.1.9 Independence hypothesis

An important aspect of sequential composition is the assumption of outcomes having independent behaviour [5]. Let us explain the following assumption clearly.

Assume two sequentially composed outcomes o_1, o_2 running on the same processor. The overall delay of execution can be observed from the start event of o_1 (u_1) to the end event of o_2 (r_1).



At low load, the two components behavior will be independent, the system will behave **linearly**. According to the superposition principle, the overall delay will be the sum of the two delays, as will the overall processor utilisation. [11]

When load increases, the two components will start to show dependent behaviour due to the processor utilisation increasing. The ΔQ of the observed total delay will then deviate from the sum of the two delays ($o_1 \circledast o_2$).

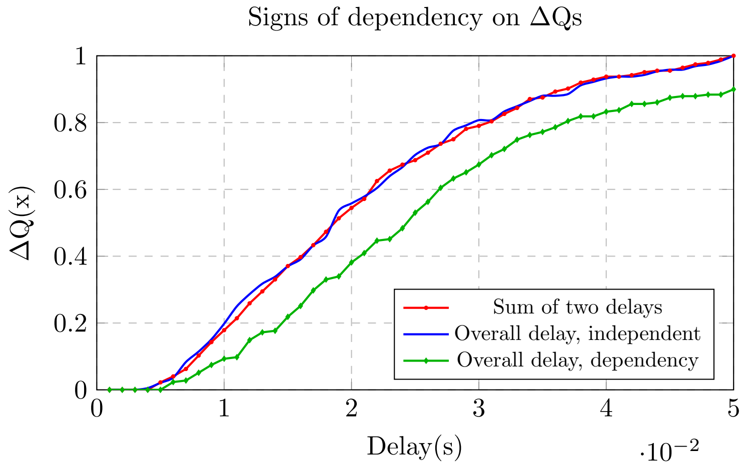


Figure 2.11: When the components are independent, the sum of the two delays (blue) and the overall delay (red) can be superposed. As o_1 and o_2 show initial signs of dependency, the overall delay (green) can be seen deviating from the sum of the two delays.

When the system is far from being overloaded, the effect is noticeable thanks to ΔQSD. As the cliff edge of overload is approached, the nonlinearity will increase [5]. These theoretical results can be observed in practice in the oscilloscope. We will explore such cases in the synthetic applications section.

2.2 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 by examining its telemetry data." [12]

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

2.2.1 erlang:trace

The Erlang programming language gives the users different ways to observe the behaviour of a system, one of those is the function `erlang:trace/3`. According to the documentation: "The Erlang run-time system exposes several trace points that can be observed, observing the trace points allows users to be notified when they are triggered" [9]. 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 2.12: erlang:trace/3 specification

Nevertheless, in Erlang Tracing there is no default way to follow a message and get its whole execution trace. 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 framework comes in.

2.2.2 OpenTelemetry

According to its website: "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. OpenTelemetry provides a standard protocol, a single set of API and conventions and lets the user own the generated data, allowing to switch between observability backends freely". [12]

OpenTelemetry is available for a plethora of languages [13], including Erlang, although, as of writing this, only traces are available in Erlang [14].

The Erlang Ecosystem Foundation has a working group focused on evolving the tools related to observability, including OpenTelemetry and the runtime observability monitoring tools [15].

Traces

Traces are why we are basing our program on top of OpenTelemetry, traces follow the whole path of a request in an application, and they are comprised of one or more spans. Traces can propagate to multiple services and record multiple paths in different microservices [16].

Span A span is a unit of work or operation. Multiple spans can be assembled into a trace and can be causally linked. The spans can have a hierarchy, where *root spans* represent a request from start to finish and a child span the requests that are completed inside the root span [16]. We will see in later sections how this can relate to what the oscilloscope does.

The notion of spans and traces allows us to follow the execution of a request and carry a context. Spans can be linked to mark causal relationships between multiple spans [17]. This relation can be represented in the oscilloscope via **probes**, we will present how spans relate to probes in following sections.

```json
{
  "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"
  },
}
```

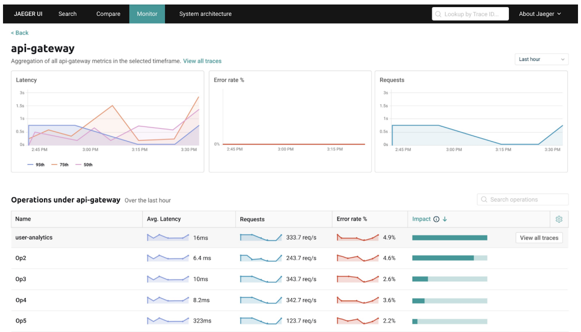
Figure 2.13: Example of span from the OpenTelemetry website. The span has a parent, indicating that child and parent spans are related and are both part of the same trace [17].

Monitoring OpenTelemetry spans

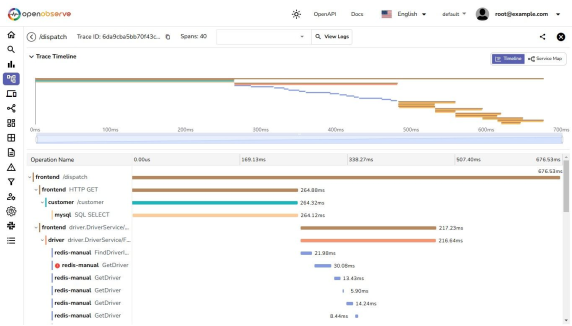
In OpenTelemetry, the user can export their traces export to backends and monitoring such as Jaeger, Zipkin, Datadog [18]. There, a user can analyse the traces to troubleshoot their programs by observing the flow of the requests[19]. These monitoring tools give extensive details about a running system, but may fail to capture essential timeliness requirements and performances issues early enough.

Our oscilloscope is a kind of monitoring tool, one that gives precise statistical insights about a running system. It is clear that the oscilloscope does not have the same capabilities as Datadog [20] might have, where you can observe cloud instances, instances cost, dependency graphs ... but the oscilloscope can nevertheless provide precise insights about dependency, overload thanks to the ΔQSD paradigm.

This is also the reason why we work alongside OpenTelemetry, the oscilloscope can be put next to a monitoring tool where one might export spans to. An engineer might consult the main monitoring tool to get the global picture of a running app, and the oscilloscope to give more precise insights to understand the system's behaviour.



(a) Jaeger interface. [21]



(b) A span analysis on OpenObserve. [22]

Span macros

OpenTelemetry provides macros to start, end and interact with spans in Erlang, the following code excerpts are taken from the OpenTelemetry instrumentation wiki. [14]

?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. [23]

```
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`

2.3 Current observability problems

A legitimate question to pose would be why one would need an additional tool to observe their system, monitoring tools are already plenty and provide useful insights into an application's behaviour. While they may seem adequate 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 instead of the usual 1 second. Between its start and its end, the user instrumenting the application sees nothing in their dashboard.

This is a big problem! One would like to know right away if something is wrong with their application. This is where the ΔQSD paradigm and the ΔQ oscilloscope come in handy.
Using ΔQSD, we can set a maximum delay ($dMax$) to be notified quickly that there is a problem. This allows for them to be detected right away in the oscilloscope.

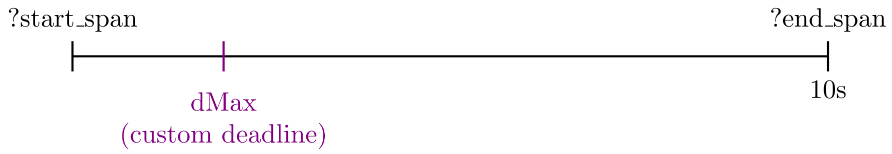


Figure 2.15: Execution of a long span in OpenTelemetry. Normally, the user will be notified after 10 seconds that the function has ended. The $dMax$ deadline allows knowing that the span has taken too long.

2.3.1 Handling of long spans

OpenTelemetry presents a bigger problem, what happens when there are long-running spans? Worse, what happens when spans are not actually terminated?

OpenTelemetry limits the length of its spans, moreover, those who are not terminated are lost and not exported. Why? They are the ones that tell more about a program execution. If the span is the parent/root span, its effect could trickle down to child spans. We can quickly see how this can become problematic, all the information about an execution of your program ... lost. Moreover, a span could not be terminated for trivial reasons: refreshing a tab, network failures, crashes ... [13]. Hazel Weakly, the author of the cited article, states that there are a few solutions that can be implemented: having shorter spans, carrying data in child spans, saving spans in a log to track spans which were not ended to manually set an end time; why the need to circumvent limitations when observing a system?

We believe that the adapter we provide can be a great start to improve observability requirements surrounding OpenTelemetry. Data about spans will always be carried to the oscilloscope, whether the span is long or non-terminated.

Chapter 3

Design

This chapter aims to first extend the concepts of ΔQSD, giving more insights into how the systems need to be instrumented to correctly work together, and how the different parts need to be integrated to interact together.

- We first provide concepts of probes, we extend the ΔQSD notion of failure and describe how time series will work in our oscilloscope, this part is crucial to understand how the measurements are done in real time.
- We then split the design of the oscilloscope in two. First explaining the Erlang side, where the system to be tested is. Secondly, we explain the C++ side. Both chapters explain how probes can be inserted and made to work together.
- Lastly, we provide high level concepts of triggers and execution windows, the key elements of the oscilloscope.

Before delving deeper into the chapter, we present the global system design diagram.

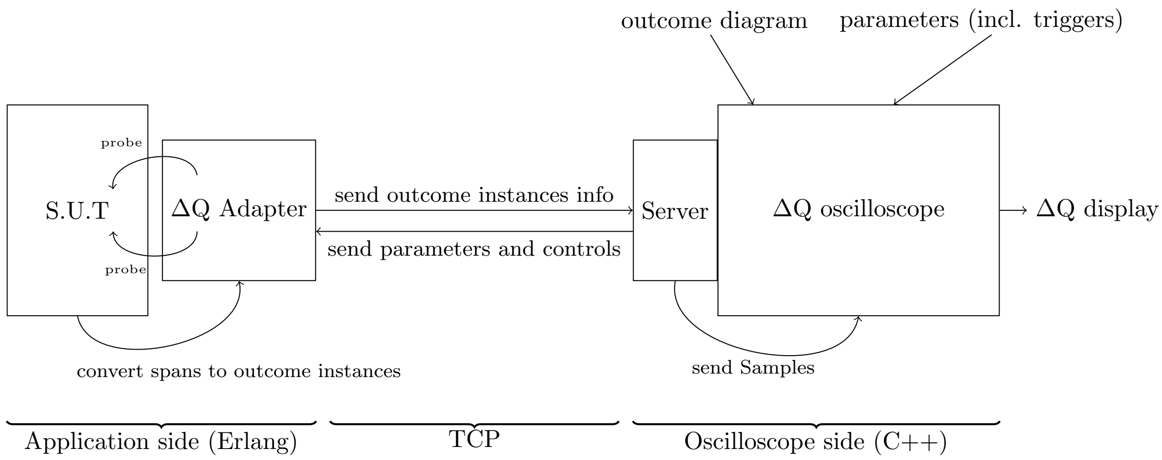


Figure 3.1: Global system design diagram. The two sides communicate via TCP sockets to share information about outcome instances and probe parameters.

We can recognise two distinct parts:

- The application side, where the Erlang system under test is. Consequently, it's where the ΔQ adapter will be, which performs the translation of spans to outcome instances.
- The oscilloscope side, where the ΔQ oscilloscope receives information from the adapter attached to the system under test to display graphs, define outcome diagrams and set parameters for probes.

3.1 Measurement concepts

3.1.1 Probes

A system instrumented with OpenTelemetry has spans and traces to observe the execution of an operation [17]. The same level of observability must be assured in the oscilloscope, this is why we provide the concept of probes, which, like spans, follow an execution from start to end.

To observe a system, we must put probes in it. 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 [5].

Consider the figure below, a probe is attached at every component to measure their ΔQs (c_2, c_3), 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 ΔQ *"observed at p_1"*, which is the ΔQ which was calculated from the data received by inserting probe p_1. The ΔQ *"calculated at p_1"* is the resulting ΔQ from the convolution of the observed ΔQs at c_2 and c_3.
Probe p_1 is the equivalent of a "root/parent span" which observe the whole execution of c_1, c_2, while p_2 and p_3 are child spans which represent single instances of execution.

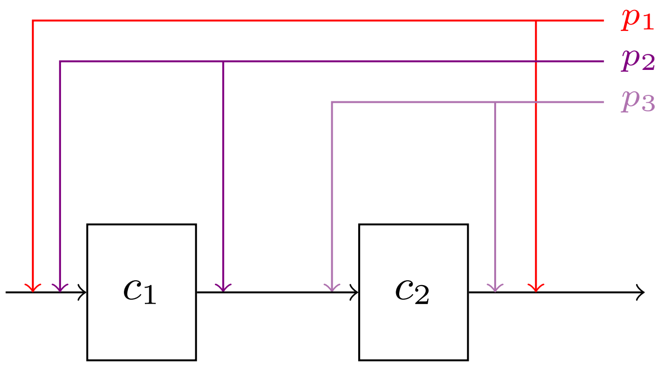


Figure 3.2: Probes inserted in a component diagram. In an application instrumented with OpenTelemetry, p_1 could be considered the root span, c_1 and c_2 its children spans sharing a causal link.

3.1.2 Extending failure

Recall the definition of failure: *"an input message m_{in} that has no output message m_{out}"* [3]. In the previous section 2.15, we also introduced the notion of a maximum delay.

By extending the notion of failure to include $dMax$, we can know right away when execution is straying away from engineer defined behaviour, avoiding having to wait until the execution is done. In ΔQSD, an execution may as well take 10 or 15 seconds, but if the delay of execution is $> dMax$, we consider that **failed** right away, we do not need to know the total execution time, the execution has already taken too much! Moreover, the full span will be exported regardless to monitoring tools which were set up by the user.

The user can observe both real time information with ΔQSD notion of failure on the ΔQ oscilloscope, and observe those spans in their monitoring tools if they wish.

The notion of failure is extended to the following definition:

"An input message m_{in} that has no output message m_{out} after $dMax$"

We can leverage this new definition to observe the system and the ΔQs in real time.

3.1.3 Time series of outcome instances

Consider a probe p with two distinct sets of events, the starting set of events s and ending set of event e. The outcome instance of a message $m_s \to m_e$ contains:

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

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

- (t_s, t_e): This representation indicates that the execution was successful (t $<$ $dMax$).
- $(t_s, \mathcal{T})$: This representation indicates that the execution has timed out (t $>$ $dMax$). The end time is equal to $t_s + \text{timeout}$
- $(t_s, \mathcal{F})$: 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 time out since the adapter will not receive an end message.

The **time series** of a probe is the sequence of n outcome instances. The collected elapsed times of execution (delay) from the outcome instances can be represented as a

CDF, which is a ΔQ!

What can be considered a failed execution? Imagine a queue with a buffer: the buffer queue being full and dropping incoming messages can be modeled as a failure.

More generally, the choice of what is considered a failed execution is left up to the user who is handling the spans and is program-dependent. Exceptions or errors can be kinds of failure.

On another note, the way of handling “errored” spans in OpenTelemetry can differ from user to user [24], so the adapter will not handle ending and setting statuses for “failed” spans.
In any case, by the new definition of failure, **timed out and failed will both be considered as a failure** in the calculation of a ΔQ. The distinction in an outcome instance is there for future refinements of the oscilloscope, where more statistics can be displayed about a ΔQ.

3.2 Application side

3.2.1 System under test

The system under test **(S.U.T)** is the Erlang system the engineer wishes to observe, it ideally is a system which already is instrumented with OpenTelemetry. The ideal system where ΔQSD is more useful is a system that executes many independent instances of the same action [1].

3.2.2 ΔQ Adapter

The ΔQ adapter is the `dqsd_otel` Erlang application [25], it starts and ends OpenTelemetry spans and translates them to outcome instances which are useful for the oscilloscope. This can be done thanks to probes being attached to the system under test, like an oscilloscope would! The outcome instances end normally like OpenTelemetry spans or, additionally, can timeout, given a custom timeout ($dMax$), and fail, *according to user’s definition of failure.*

Handling of OpenTelemetry spans which goes beyond starting and ending them is delegated to the user, who may wish to do further operations with their spans. The adapter is called from the system under test and communicates outcome instances data to the oscilloscope via TCP.

The adapter can receive messages from the oscilloscope, the messages are about updating probe’s $dMax$ or starting and stopping the sending of data to the oscilloscope.

3.2.3 Inserting probes in Erlang - 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. [16]

To get the equivalent of spans for the oscilloscope, the adapter needs to be called at the starting events of a probe to start an instance of a probe, and at the ending events to end the outcome instance. The name given with `"start_span"` is the name of the probe. Further details about the implementation of the adapter are explained in the

```
% Start the outcome instance of probe. The call to dqsd_otel starts an
↪  OpenTelemetry span, as it contains a call to ?start_span(Name)
{ProbeCtx, ProbePid} = dqsd_otel:start_span(<<"probe">>),

% Start and fail span directly
{WorkerCtx, WorkerPid} = dqsd_otel:start_span(<<"worker_1">>),
dqsd_otel:fail_span(WorkerPid),
%Here, you would need to end the span manually with ?end_span

%Example of with_span, the call to OpenTelemetry ?with_span is inside
↪  the adapter function, the function fun() -> ok end is executed
↪  inside dqsd_otel.
dqsd_otel:with_span(<<"worker_2">>, fun() -> ok end),
%End the outcome instance of probe. This ends the OpenTelemetry span
↪  aswell. If the outcome instance has already timed out (the time
↪  from start_span to end_span > dMax), the oscilloscope receives no
↪  message where the status is successful. Otherwise, this sends a
↪  message with startTime, endTime, the name "probe" and success
↪  status.
dqsd_otel:end_span(ProbeCtx, ProbePid),
```

Figure 3.3: Example usage of the adapter

following chapters.

3.3 Oscilloscope side

3.3.1 Server

The server is responsible for receiving the messages containing the outcome instances from the adapter. The server forwards the instances to the oscilloscope.

3.3.2 ΔQ Oscilloscope

The oscilloscope is a C++ graphical application which implements a dashboard to observe ΔQs of probes inserted in the system under test [26]. It receives the instances corresponding to probes from the server and adds them to the time series of the probes whose instance is being received. 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 parameters for probes. It can also display snapshots of the system as if it was frozen in time.

3.3.3 Inserting probes in the oscilloscope

Probes are automatically inserted in the oscilloscope when creating outcome diagrams. They are inserted on the outcomes observables, operators observables and to the sub-outcome diagrams observables (probes that observe the causal links of multiple outcomes/operators), we will see later on how they can be defined and how an outcome diagram can be created.
The names that are given to outcome, operators and sub-outcome diagrams are the names of the probes that observe them. Giving these probes a name allows the oscilloscope to match the outcome instances to the probes' time series.

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.

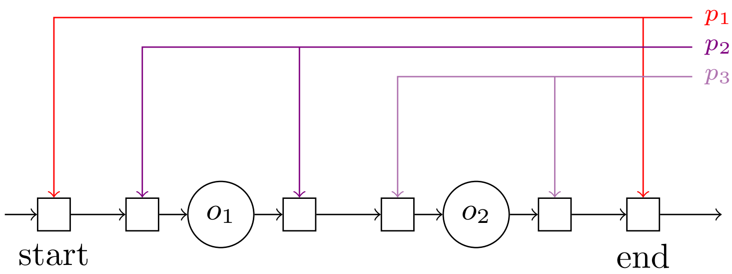


Figure 3.4: Probes inserted in the outcome diagram of the previous component diagram in Fig. 3.2.

The **observables** are an abstract representation of events. Consider the previous code snippet Fig. 3.3: the *start* event of "probe" and worker$_1$'s start event are subsequent instructions. The probe's start event is practically the same as worker$_1$'s start event, indeed, they could be overlapped in the graph above. We nevertheless show the distinction to show that probe and worker$_1$ need to be started differently in Erlang as the information they carry is about two distinct instances. Furthermore, this difference is remarked in the definition of outcome diagrams, for which we provide a syntax in the following chapter.

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

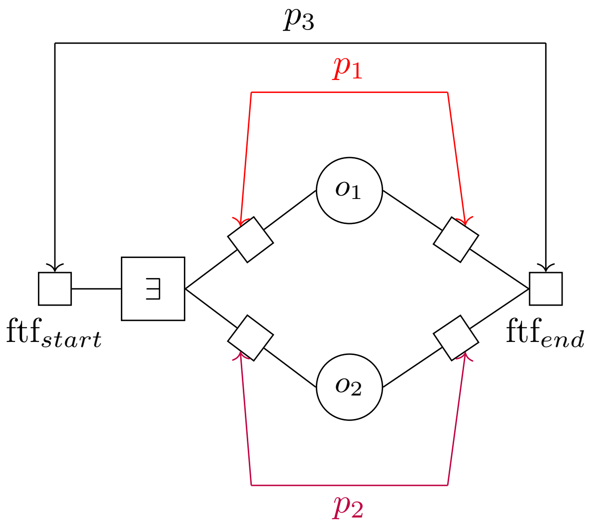


Figure 3.5: Probes inserted into an operator.

The **observed ΔQ** for the first-to-finish operator is the ΔQ for the observables (**start**, **end**). The **calculated ΔQ** is the ΔQ which is the result of the first-to-finish operator being applied on o_1, o_2.

3.4 Triggers

Much like an oscilloscope that has a trigger mechanism to capture periodic signals or investigate a transient event [27], the ΔQ *oscilloscope* has a similar mechanism that can recognise when an observed ΔQ violates certain conditions regarding required behaviour and record snapshots of the system.

Each time an observed ΔQ is calculated, 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.

3.4.1 Snapshot

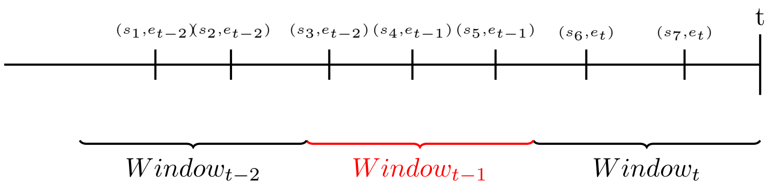
A snapshot of the system gives insights into the system before and after a trigger was fired. It gives the user a still of the system, as if it was frozen in time. All the ΔQs which are calculated during the system's execution are stored away. Then, if no trigger is fired, older ΔQs are removed. Otherwise, the oscilloscope keeps recording ΔQs without removing older ones, to allow the user to look at the state of the system before and after the trigger.

3.5 Sliding execution windows

There are two important windows that we consider in our oscilloscope, the *sampling window* and the *polling window*.

3.5.1 Sampling window

Suppose we are at time t, the observed (and calculated, if applicable) ΔQs at time t we will display are the ΔQs obtained from the outcome instances who ended within a sampling window in the **window of time** $(t-1)_l$ **-** $(t-1)_u$, with $t-1$ equal to $t-x$, and x the sampling rate. The sampling rate is how often ΔQs are calculated.
This is to account for various overheads that need to be taken into consideration. They could be network overhead, the adapter overhead, C++ latency ... Imagine multiple outcome instances that are ended at a time slightly lower but close to t, and due to the overheads the messages arrive at a time slightly higher but close to t, the outcome instance would not be taken into consideration for the calculation of a ΔQ.



The sampling window then advances every x seconds, setting the new window:

$$\text{From: } (t-1)_l,\ (t-1)_u \xrightarrow{t+1} t_l, t_u.$$
$$\text{Where: } t_l = (t-1)_u \text{ and } t_u = (t-1)_u + x$$

The ΔQs which are observed and calculated in a sampling window are not precise, this is why we need to introduce the polling window.

3.5.2 Polling window (Observing multiple ΔQs over a time interval)

The polling window is the window of ΔQs which are stored to keep a snapshot of the system over time and over which confidence bounds are calculated. The polling window serves to improve the precision of the ΔQ measurement.

Suppose we are at time $t = 0$, the polling window will have 0 ΔQs. As the sampling window advances, more ΔQs are sampled, which in turn are added to the snapshot and to the confidence bounds.

The limit of ΔQs for a polling window (subsequently snapshots and confidence bounds) is 30 ΔQs. At $t = 31$, the older ΔQs will be removed from the polling window and in turn from the snapshots and confidence bounds. Newer sampled ΔQs will be added, keeping the limit of ΔQs in a polling window to 30.

Chapter 4

Oscilloscope: User level concepts

The following chapter gives insights on the user level concepts of ΔQSD in the oscilloscope. They are the concepts needed by the user to understand how the oscilloscope works.

- We first provide insights into how ΔQSD was implemented in the oscilloscope, the parameters that define a probe's ΔQ, its representation and what can be done with ΔQs. We show how probe's ΔQ(s) will be shown in the oscilloscope.
- We then provide a language to write outcome diagrams based on an already existing syntax.
- Lastly, we explain the different controls present on the oscilloscope dashboard.

4.1 ΔQSD concepts

We provide in this section the concepts needed to understand what is displayed on the oscilloscope.

4.1.1 Representation of a ΔQ

We provide a class to calculate the ΔQ of a probe between a lower time bound t_l and an upper time bound t_u. It can be calculated in two ways:

Observed ΔQ The first way is by having n collected outcome instances between t_l and t_u, calculating its *probability density function* (PDF) and then calculating the *empirical cumulative distribution function* (CDF) based on its PDF. This is called the **Observed ΔQ**.

Calculated ΔQ A ΔQ can also be calculated by performing operations on two or more observed ΔQs (convolution, operators operations), the notion of outcome instances is then lost between calculations, as the interest shifts towards calculating the resulting PDFs and CDFs. This is called the **Calculated ΔQ**. A simple outcome can **not** have a "calculated ΔQ", we can only observe the delay from its observables.

If you recall Section 3.3.3, the probes p_2 and p_3 observe simple outcomes, they can only display the observed ΔQs of o_2, o_3. The probe p_1 instead observes the sequential composition of said outcomes. We can display its "observed ΔQ" from the execution from *start* to *end* and the "calculated ΔQ" as the convolution of the observed ΔQs of o_1, o_2

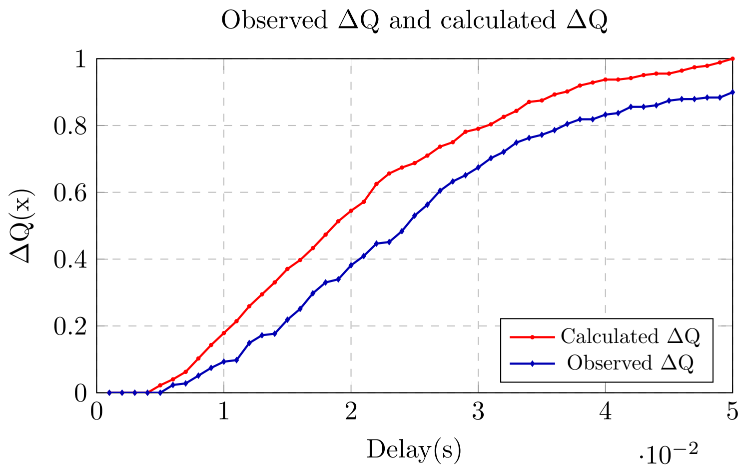


Figure 4.1: (Red, circle, above): Calculated ΔQ. (Blue, diamond, below): Observed ΔQ

4.1.2 dMax = Δt · N

The key concept of ΔQSD is having a maximum delay after which we consider that the execution is timed out. This is represented in the oscilloscope as $dMax$. Understanding this equation is key to correctly using the oscilloscope and exploring tradeoffs

Setting a maximum delay for a probe 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 ΔQs.

Let us explain the following equation:

$$dMax = \Delta t \cdot N \tag{4.1}$$

- $dMax$: The maximum delay, it represents the maximum delay that an outcome instance of a probe can have. The execution is considered "timed out" (failure) after $dMax$.
- Δt: The resolution of a ΔQ. It is the bin width of a bin in a probe's ΔQ.
- N: The precision of a ΔQ. It is the number of bins in a probe's ΔQ.

It can be informally described as a "two out of three" equation. If the user wants higher precision but the same $dMax$, the resolution must change, and so on for every parameter.

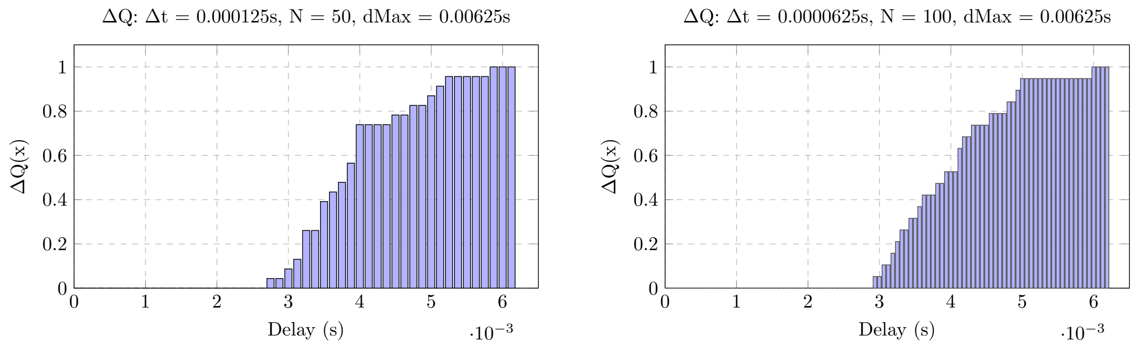


Figure 4.2: Left: Sample ΔQ representation as a histogram with higher resolution but lower precision.
Right: Sample ΔQ representation as a histogram with lower resolution but higher precision.
Both ΔQs have the same $dMax$, but the amount of precise information they provide is far different.

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. The user must set these parameters carefully during execution.

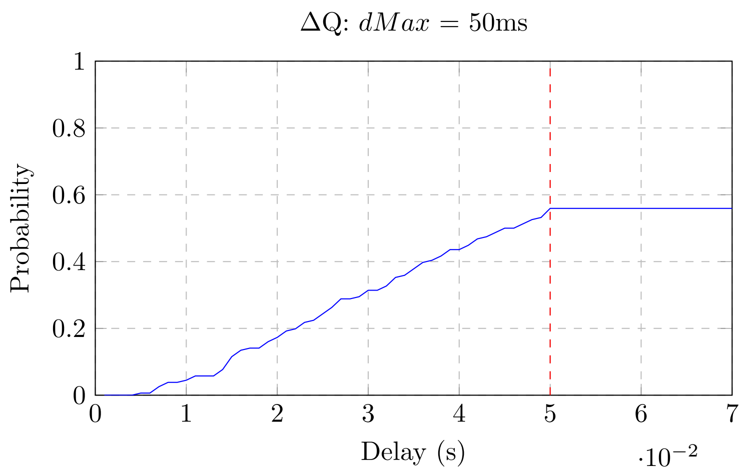


Figure 4.3: ΔQ: $dMax$ = 50ms, the ΔQ will stay constant when delay > $dMax$.

dMax limitation

$dMax$ can **not** be lower than 1 millisecond and will be rounded to the **nearest** integer in the adapter, this is a limitation of Erlang `send_after` function which only accepts integers and milliseconds values. For example, if on the oscilloscope the $dMax$ is equal to 1.56ms, the adpater will fail spans after 2 ms.

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, it is equal to the QTA we have shown in Figure Section 2.1.6. The user can set the QTAs they want for the probes they have inserted, but the delays at the percentiles must be $< dMax$.

4.1.4 Confidence bounds

To observe the stationarity of a probe we must observe its ΔQs over a polling window and calculate confidence bounds over said ΔQs. A single ΔQ may fluctuate, moreover, as it is an CDF, it is not as precise as a CDF. This is why we include the mean and confidence bounds of ΔQs in the plot, which give a probability range over which the true CDF of the ΔQ should fall. [28]

Confidence bounds are given for *observed* and *calculated* ΔQs.
We first calculate a mean of the ΔQs in the polling window, this gives an idea of how the probe has been behaving during the polling window. Given this mean, we can calculate its confidence bounds.

The bounds are updated dynamically by inserting or removing a ΔQ. Every time a new ΔQ is calculated, the oldest ΔQ in a window is removed if #ΔQs(polling window) > limit. The new ΔQ is added to the calculation of the mean and confidence bounds as it is calculated.
This allows us to consider a small window of execution rather than observing the execution since the start for the bounds, this can help in observing stationarity of the system, where less sampled ΔQs can help observe short term behaviour.
With a big window of ΔQs, temporary overload may not greatly affect the mean and bounds, while, if we consider the current size of the polling window (30 ΔQs), a few ΔQs which deviate from stationary behaviour have a greater impact on the bounds and mean.

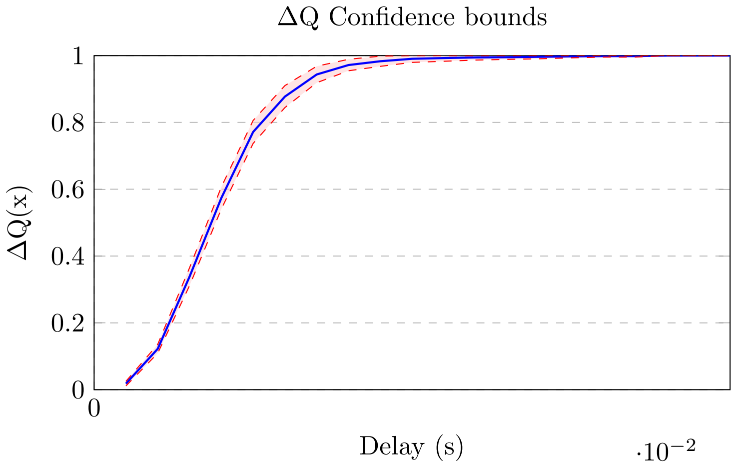


Figure 4.4: Upper and lower bounds (dashed, red) of the mean (blue) of multiple ΔQs. In a system that behaves linearly, the bounds will be close to the mean, once the overload is approaching, or a system is showing behaviour that diverges from a linear one, the bounds will appear larger.

4.2 ΔQ display

Now that we have introduced the required concepts, we can put everything together to be plotted. In summary, a probe's displayed graph must contain the following functions:

- The observed ΔQ of the sampling window, with the mean and confidence bounds of the polling window ΔQs.
- If applicable, the calculated ΔQ from the causally linked components observed in a probe, with the mean and confidence bounds of the polling window ΔQs.
- Its QTA (if defined).

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

In the photo below we can observe the multiple elements as they are displayed in real time in the oscilloscope.

- (1): The mean of the polling window observed ΔQs (yellow) with the confidence bounds of the mean. Upper bound (dark green) and lower bound (light green). The observed ΔQ of the sampling window can be observed going out of the confidence bounds at delay 0.00125 s. The ΔQ in a sampling window as is less precise than the mean and confidence bounds calculated in the polling window.
- Arrow, red, above: The mean of the calculated ΔQs (ochre) with the confidence bounds of the mean. Upper bound (purple) and lower bound (magenta). The calculated ΔQ of the sampling window is inside its confidence bounds.
- (3): The **QTA**.

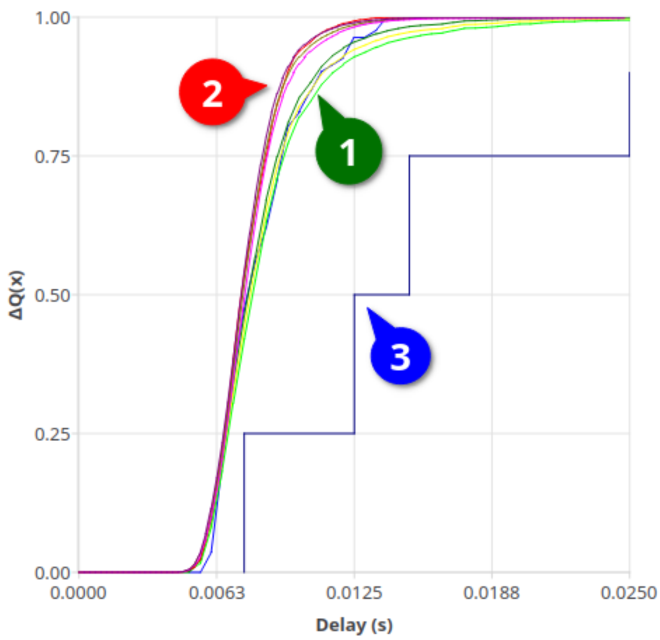


Figure 4.5: ΔQs, confidence bounds, means and QTA for a probe observing the causal link of multiple components.

4.3 Outcome diagram

An abstract syntax for constructing outcome diagrams has already been defined in a previous paper [3], nevertheless, the oscilloscope 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 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
```

4.3.2 Sub-outcome diagrams

Multiple sub-outcome diagrams can be created for multiple parts of the system. They can then be linked together to form the global system outcome diagram. Sub-outcome diagrams can observe one or multiple components. Recall Section 3.3.3, we defined a probe which observes the sequential composition of o_1, o_2. The probe (sub-outcome diagram) p_1 can be defined as:

```
p_1 = o_1 -> o_2;
```

A probe is attached at the begin and end of p_1, it will observe the whole system and the calculated ΔQ will be the convolution of o_1, o_2.

The lines defining these diagrams must be semicolon terminated. Outcomes and operators cannot be defined on their own, they must be observed in a sub-outcome diagram.

Sub-outcome diagrams can be reused in other diagrams by adding `s:` (sub-outcome diagram) before they are used.

```
p_3 = s:p_1 -> s:p_2;
```

This allows for easy composition and reuse of different parts of the system, allowing for independent refining of diagrams.

4.3.3 Outcomes

To attach a probe to an outcome observables, it is enough to declare an outcome with its name inside a diagram.

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

4.3.4 Operators

First-to-finish, all-to-finish and probabilistic choice operators must contain at least two components.

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 probabilites p_i, $0 < p_i < 1$, $\sum_{i=0}^{n} p_i = 1$

4.3.5 Limitations

Our system has a few limitations compared to the theoretical applications of ΔQ, namely, no cycles are allowed in the definition of outcome diagrams.

```
p_1 = s:p_2;
p_2 = s:p_1;
```

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

4.3.6 Outcome diagram example

We provide a sample example of an outcome diagram definition. We also provide its resulting outcome diagram with probes inserted.

```
two_hops = o2 -> o3;
total = p:pc[0.9, 0.1](o1, s:two_hops);
```

Figure 4.6: Sample textual definition of an outcome diagram.

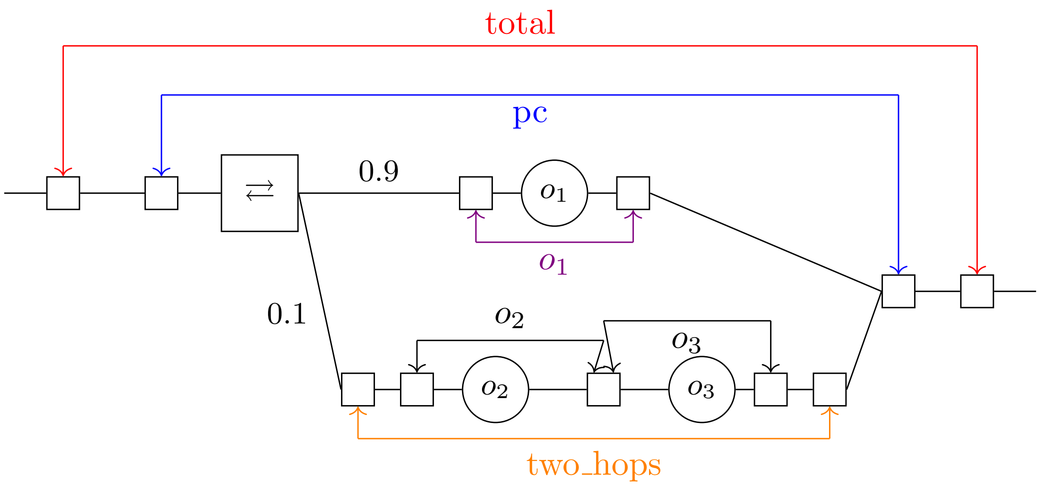


Figure 4.7: Resulting outcome diagram for definition 4.6.

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/Handle plots tab

System creation In this tab the user can create its system with the outcome diagram grammar. They can save the outcome diagram text or load it, the outcome diagram definition 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, they 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 they wants to plot. They can select multiple probes per plot and display multiple plots on the oscilloscope window.

Sampling rate The user can choose the sampling rate of the system: How often ΔQs are calculated and displayed in the oscilloscope.

Editing a plot By clicking onto a plot that is being shown, the user can choose to add or remove probes to and from it, thanks to the widget in the lower right corner. Multiple probes can be selected to either be removed or added.

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 probe, 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 can set the parameters we explained previously, Δt and N. When this information is saved by the user, the new $dMax$ is transmitted to the adapter 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 oscilloscope starts a timer, during which all probes start recording the observed ΔQs (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.

Connection controls

Erlang controls The user can set the IP and the port where the ΔQ adapter is listening from. Two additional buttons communicate with the adapter by sending messages, they can start and stop the adapter's sending of outcome instances.

C++ server controls The user can set the IP and the port for the oscilloscope's server.

4.4.2 Plots window

To the left, the main window shows the plots of the probes being updated in real time.

4.5 Triggers

There are two available triggers which can be selected by the user, the triggers are evaluated on the **observed ΔQ**.

4.5.1 Load

A trigger on an observed ΔQ can be fired if the amount of outcome instances received for a probe in a sampling window is greater than what the user defines:

$$\#\text{instances}_{probe}(\Delta\text{Q}) > \text{maxAllowedInstances}_{probe}$$

4.5.2 QTA

A trigger on an observed ΔQ can be fired if:

$$\Delta\text{Q}_{obs} \not< \text{observableQTA}$$

Chapter 5

Oscilloscope: implementation

The following chapter gives a more technical description of the oscilloscope.

- We provide a more in-depth look at the ΔQSD concepts introduced in the previous chapter.
- We then explain how the ΔQ adapter works, its API and the underlying mechanism that let us export outcome instances to the oscilloscope.
- Next we give a technical explanation of the parser generator we used to parse the outcome diagram syntax.
- Lastly, we briefly talk about the dashboard graphical framework.

5.1 ΔQSD implementation

A probe's ΔQ can be represented internally by a PDF and displayed as an CDF. Here is how both can be calculated given n outcome instances.

5.1.1 Histogram representation

The ΔQ representation is one of a histogram for its PDF and a cumulative histogram for its CDF.

PDF

We approximate the PDF of the observed ΔQ via a histogram. We partition the values into N bins of equal width, Given $[x_i, x_{i+1}]$ the interval of a bin i, where $x_i = i\Delta t$, and $\hat{p}(x_i)$ the value of the PDF at bin i, for n bins:

$$\begin{cases} \hat{p}(i) = \dfrac{s_i}{n}, \text{if } i \leq n \\ \hat{p}(i) = 0, \text{if } i > n \end{cases} \tag{5.1}$$

Where s_i the number of successful outcome instances whose elapsed time is contained in the bin i, n the total number of instances. [29]

CDF

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

$$\begin{cases} \hat{f}(i) = \sum_{j=1}^{i} \hat{p}(j), & \text{if } i \leq n \\ \hat{f}(i) = \hat{f}(n), & \text{if } i > n \end{cases} \tag{5.2}$$

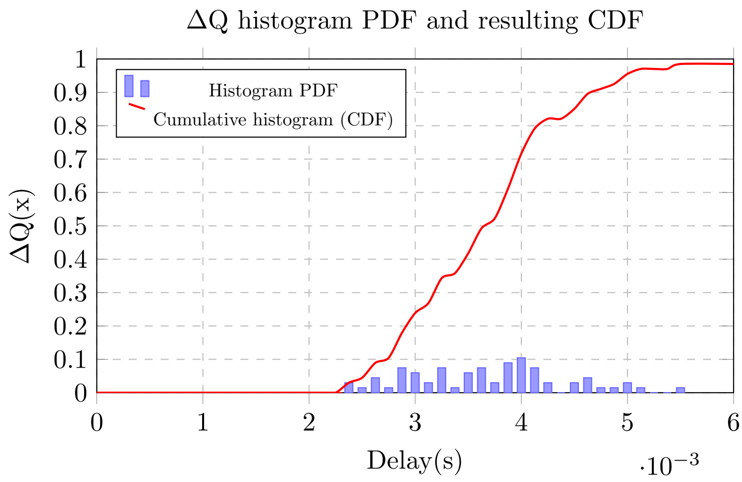


Figure 5.1: Blue bins: PDF of a sample ΔQ. Red: Resulting CDF of ΔQ PDF, the CDF is what is displayed on the dashboard.

5.1.2 dMax

We introduced $dMax$ in the previous chapters, we provide here the full equation that allows $dMax$ to be calculated:

$$dMax = \Delta t_{base} * 2^n * N \tag{5.3}$$

Where:

- Δt_{base} represents the base width of a bin, equal to 1ms.
- n the exponent that is set by the user in the dashboard. It is limited to [-10, 10].
- N the number of bins.

We chose 1 ms in combination with 2^n as it allows us to go from very fine bin widths ($\approx$ 1 μ) to large bin widths ($\approx$ 1 s)

5.1.3 Convolution

We present the two solution to perform convolutions we explored during the implementation.

Naïve convolution

Given two ΔQ binned PDFs f and g with equal bin widths, the result of the convolution $f \circledast g$ is given by [30]:

$$(f \circledast g)[n] = \sum_{m=0}^{N} = f[m]g[n-m] \tag{5.4}$$

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. Moreover, the oscilloscope started presenting noticeable lag and frame skipping. This is why we decided to explore Fast Fourier Transform convolution.

Fast Fourier Transform Convolution

FFTW (Fastest Fourier Transform in the West) is a C subroutine library [31] 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 ΔQs. We adapt our script from an already existing one found on GitHub. [32]

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 $\mathcal{O}(N\log N)$. This is why the naïve convolution algorithm is not used. We will analyse the time gains in a later chapter.

FFT and naïve convolution produce the same results in our program, barring ε differences (around 10^{-18}) in bins whose result should be 0. This is most likely due to rounding error.

FFTs algorithms are plenty, the choice of the one to use is left up to the subroutine via the parameter `FFTW_ESTIMATE` [33].

5.1.4 Arithmetical operations

The FTF, ATF and PC operators on ΔQs use a simple set of arithmetical operations to calculate a ΔQ. The time complexity of FTF, ATF and PC is trivially $\mathcal{O}(N)$ where N is the number of bins.

Scaling (multiplication) A ΔQ can be scaled w.r.t. a constant $0 \leq j \leq 1$. It is equal to binwise multiplication on CDF bins. It is used for the probabilistic choice operator.

$$\hat{f}_r(i) = \hat{f}(i) \cdot j \tag{5.5}$$

Operations between ΔQs Addition, subtraction and multiplication can be done between two ΔQ of equal bin width (but not forcibly of equal length) by calculating the operation between the two CDFs of the ΔQs:

$$\Delta\mathrm{Q}_{AB}(i) = \hat{f}_A(i)[\cdot, +, -]\hat{f}_B(i) \tag{5.6}$$

They are used for all operators.

5.1.5 Confidence bounds

Here is how we calculate the mean and lower/upper confidence for the ΔQs CDF at bin $i \quad \forall i < N$. [29]

For x_{ij} the value of an CDF j at bin i, the mean of all CDFs for the bin over a window is:

$$\mu_i = \frac{1}{n_i} \sum_{j=1}^{n_i} x_{ij} \tag{5.7}$$

Its variance:

$$\sigma_i^2 = \frac{1}{n_i} \sum_{j=1}^{n_i} x_{ij}^2 - \mu_i^2 \tag{5.8}$$

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

$$CI_i = \mu_i \pm \frac{\sigma_i}{\sqrt{n_i}} \tag{5.9}$$

5.1.6 Rebinning

Rebinning refers to the aggregation of multiple bins of a bin width i to another bin width j. Previous operations between ΔQs must be done on ΔQs that have the same bin width. This is why it is fundamental that all probes have a common Δt_{base} and why we have a 2^n factor to calculate the total bin width.

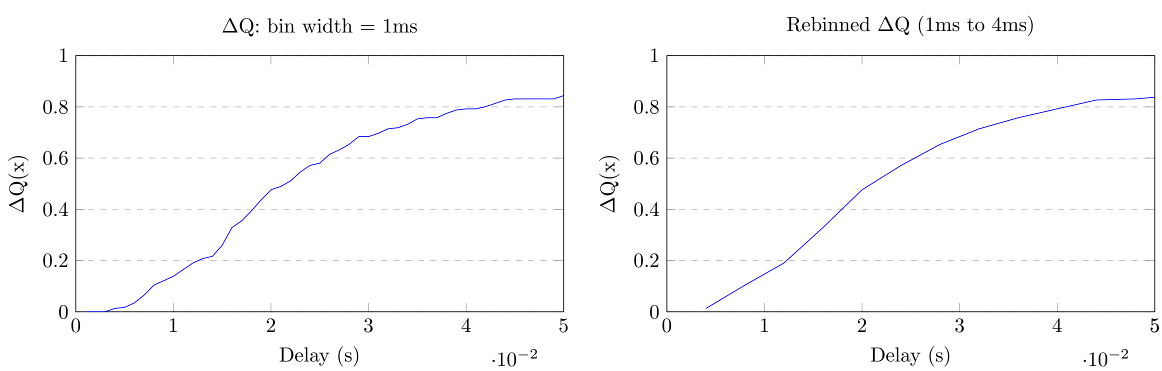
Given two ΔQs ΔQ_i, ΔQ_j, the common bin width Δt_{ij} is:

$$\Delta t_{ij} = \max \{\Delta_{Ti}, \Delta_{Tj}\}$$

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

$$p'_b = \sum_{n=b\cdot k}^{b+1\cdot k-1} p_n, \quad b = 0, 1, \ldots \lceil \frac{N}{k} \rceil \tag{5.10}$$

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



(a) Sample ΔQ with 1ms bins (b) ΔQ on the left after rebinning to 4ms bins

5.2 Adapter

The adapter, called `dqsd_otel` is a rebar3 [34] application built to replace OpenTelemetry calls and create outcome instances. It is designed to be paired with the oscilloscope to observe an Erlang application.

5.2.1 API

The adapter 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 adapter less of an encumbrance for the user.

Moreover, the adapter will always start OpenTelemetry spans but only start outcome instances if the adapter has been activated. The adapter can be activated by the oscilloscope by pressing the *"start adapter"* button and can be stopped via the *"stop adapter"* button.

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: Name of the probe in Erlang binary representation. For example, if the name of the probe is "probe", the binary representation would be constructed with "«"probe"»" [35]. We will refer to this as "binary name" from now on.
- 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 adapter is active and the probe'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/2, with_span/3

```
with_span/2: -spec with_span(binary(), fun(() -> any())) -> any().
with_span/3: -spec with_span(binary(), fun(() -> any()), map()) ->
↪  any().
```

Parameters:

- Name: Binary name of the probe.

- 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/2

```
-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/1

```
-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 probe 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 an outcome instance to the oscilloscope with the start and end time of the execution of the probe and success status.
- `{fail_span, end_time}`: This will send an outcome status to the oscilloscope indicating that an execution of a probe has failed.

- `{timeout, end_time(StartTime +` $dMax$`)}`: If the program hasn't ended the span before $dMax$, the timer will send a `timeout` message and it will send an outcome instance to the oscilloscope indicating that an execution of a probe has timed out.

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.* [36]

5.2.2 Handling outcome instances

To create outcome instances of a probe we must obtain three important informations:

- Its name.
- The time when the span was started.
- Its $dMax$.

They start time and end time are supplied upon starting/ending a span by calling this function:

```
StartTime/EndTime = erlang:system_time(nanosecond).
```

The name is given when starting a span and the $dMax$ is stored in a dictionary in the adapter.

The outcome instance is created only if two conditions are met: the adapter has been set as active and the user set a timeout for the probe, 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:probe name,b:Start time (beginning),e:End time (end or
↪  deadline),s:The status
```

5.2.3 TCP connection

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

TCP server

The TCP server is responsible for receiving commands from the oscilloscope. It can be run by setting its IP and port via:

```
-spec start_server(string() | binary() | tuple(), integer()) -> ok |
↪  {error, Reason}
```

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

- `start_stub`: This command sets the adapter as active, it can now send outcome instances to the oscilloscope if the probe's $dMax$s are defined.
- `stop_stub`: This commands sets the adapter as inactive, it will no longer send outcome instances to the oscilloscope.
- `set_timeout;probeName;timeout`: This command indicates to the adapter to set the $dMax =$ timeout for a probe, a limit of the adapter 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 adapter to send the spans to the oscilloscope. The client connects over TCP to the oscilloscope by connecting to the oscilloscope server's address and opens a socket where it can send the outcome instances.

```
-spec try_connect(string() | binary(), integer()) -> ok.
```

5.3 Parser

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

5.3.1 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 [37].

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. [38]

5.3.2 Grammar

ANTLR provides a yacc-like metalanguage [38] to write grammars. Below, is the grammar for our system:

```
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;
```

```
start: definition* system? EOF;

definition: IDENTIFIER '=' 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 [39], 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 parser generator thinks that an operator or a probe will be next.

5.4 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. [40] 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.

The interface is composed of a main window, where widgets can be attached to it easily. Everything that can be seen is customisable widgets. This allows for easy reusability, modification and removal without great refactoring due in other parts of the system.

Chapter 6

Application on synthetic programs

This section aims to provide an example of how the oscilloscope could be used to instrument an application, in this case, a synthetic one. We explain how the ΔQSD paradigm can be applied to explore tradeoffs in design and to gain more insights into a running system.

6.1 System with sequential composition

We model a first system with two sequentially composed component. We choose two model the two components as M/M/1/K queues.

Why M/M/1/K queues? An average component in a distributed system can be modeled as an M/M/1/K, due to the exponential inter-arrival rate of messages λ, the exponential distribution of the execution delay μ, the buffer size of messages K of a component and the failure rate f. [1]

Let us first provide a refresher about M/M/1/K queues:

- λ: The arrival rate.
- s: The service time, is the time it takes to serve a message.
- μ: The service **rate** and E[s] $= \dfrac{1}{\mu}$
- Offered load: $\rho = \dfrac{\lambda}{\mu}$

We will control λ to show its effects on the offered load. The offered load can tell much about the system:

- At low load ($\rho < 0.8$) the failure will tend to 0, the system is behaving correctly and the ΔQ will show that, the delay will tend to 1.
- Once ρ is approaching high load ($\rho > 0.8$) we can observe the failure increasing quickly, but we can observe the system starting to get bad after $\rho > 0.5$! [1]

6.1.1 System composition

The system has two components `worker_1`, `worker_2`. Each individual component is composed of a queue of size $K = 1000$ and a worker process.

The system sends n messages per second following a Poisson distribution to `worker_1`'s queue. The queue notifies its worker, which then does N loops, which are defined upon start, of fictional work. Once done, worker_1 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 as worker_1. When a worker completes its work, it notifies its queue, freeing one message from its buffer size and allowing the next message to be processed.

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

A probe named "*probe*" is defined, which observes the execution from when the message is sent to `worker_1` up until `worker_2` is done.

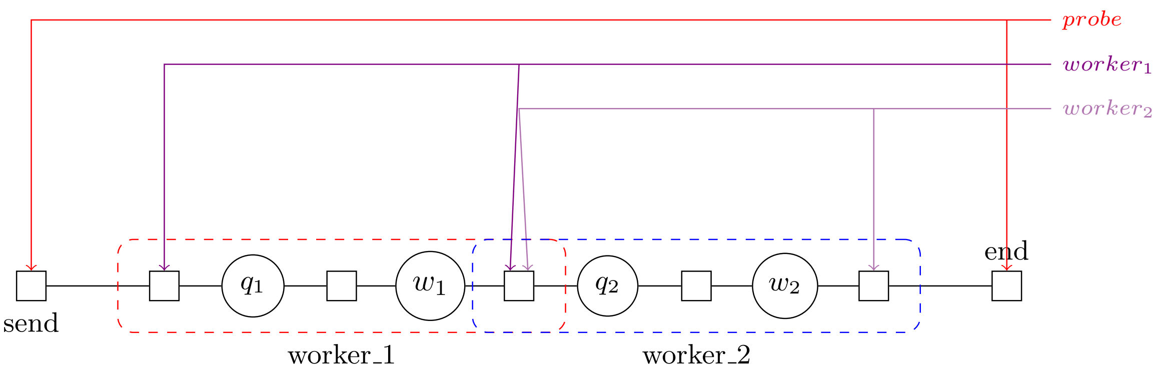


Figure 6.1: Outcome diagram of the M/M/1/K queue with the colored lines representing the probes that were inserted.

6.1.2 Determining parameters dynamically

We stated previously that determining parameters is something that must be done with an underlying knowledge of the system. The oscilloscope can provide knowledge of the system, here is an example of worker_1 and worker_2 as observed in the oscilloscope.

The engineer supposes that the workers executions should take a maximum of 4 ms to complete, but doesn't actually know how long the executions should take. The engineer, after having set the required parameters observes in the following graph in the oscilloscope **??**.

The oscilloscope shows the engineer that their assumptions do not correspond to the actual system ΔQ, the user can then modify the parameters to observe the actual system's behaviour. By setting $dMax$ to 8 ms, they can observe the worker's ΔQs failure approaching 0.

On the other hand, the engineer's assumption could have been what they truly expected from the system, in this case, the oscilloscope tells him that the system is not behaving as expected.

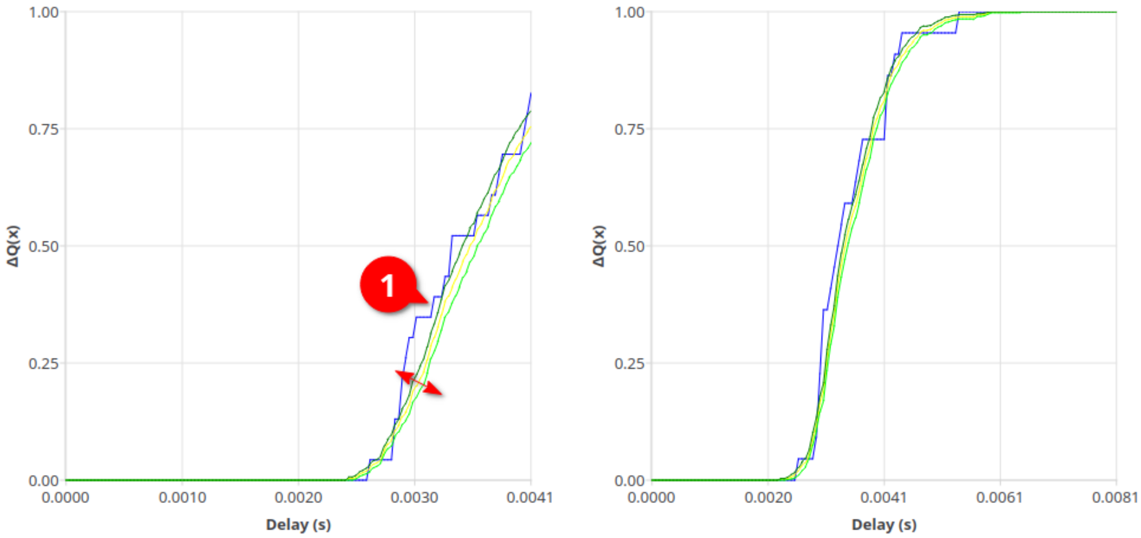


Figure 6.2: Left: worker_1 ΔQ with $dMax = 4ms$. (Green, between the arrow): Mean and confidence bounds. (1, blue): Observed ΔQ. The worker failure tends to 0.25.
Right: worker_1 ΔQ with $dMax$ = 8ms. The worker failure now tends to 0.
In both graphs we can observe how the observed ΔQ of the sampling window is not precise. The step function representation of it fluctuates. The mean and confidence bounds provide a more precise representation of the ΔQs of worker_1 over a polling window.

Low Load

Let's first observe the system at low load, we will send 50 messages per second to observe the system under test to get key properties. We can observe that the average worker's execution takes ≈ 30ms. We then have $\mu_{worker} = \dfrac{1}{0.0033} \approx 300$ req/s. Thus, $\rho = \dfrac{50}{322} = 0.16$, we are in nice grounds!

At low load, we can observe in the oscilloscope the probe **observed ΔQ** and **calculated ΔQ** confidence bounds overlap.

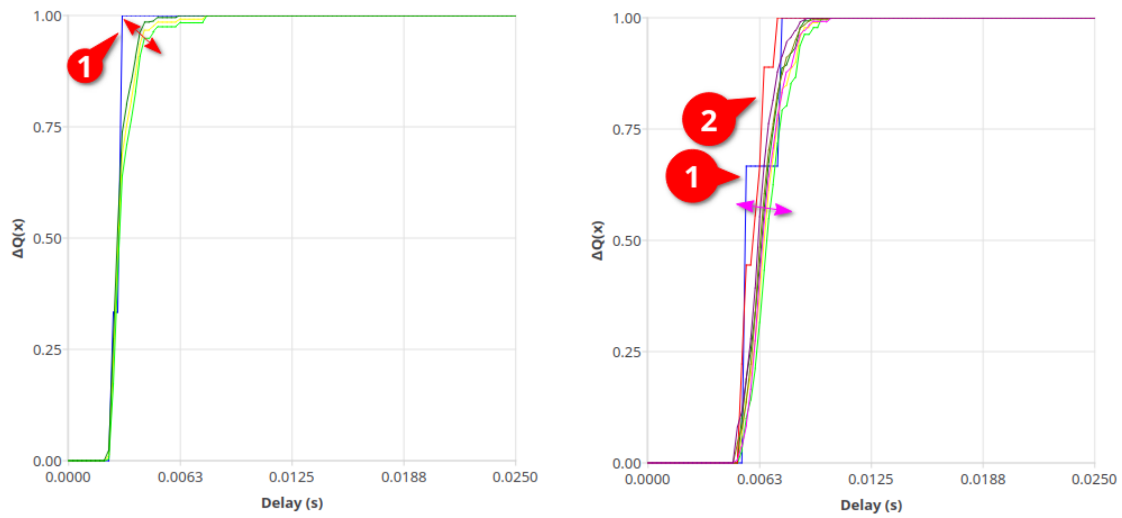


Figure 6.3: Graphs for $\lambda = 50$. Left: worker_1 ΔQs. (1, Blue): The observed ΔQ. (Green, between arrows): The confidence bounds.
Right: probe ΔQs. (1, blue): The observed ΔQ. (2, red): The calculated ΔQ. (Magenta, between arrows): the observed and calculated ΔQs confidence bounds overlapping.

Early signs of overload

At load $= 0.5$ the system should start showing bad behaviour. Let us observe what happens when $\lambda = 150 \rightarrow \rho = 0.5$.

Recall 2.11, we can start to observe early signs of dependency! At load 0.5 the calculated ΔQ is deviating from the observed one. This is a sign that the performance is degrading. At the 50th percentile the deviation is minimal, around 0.4 ms. Nevertheless, the precision of the paradigm allows even for these minimal deviations to appear on the graphs, being able to recognise early signs of overload approaching.

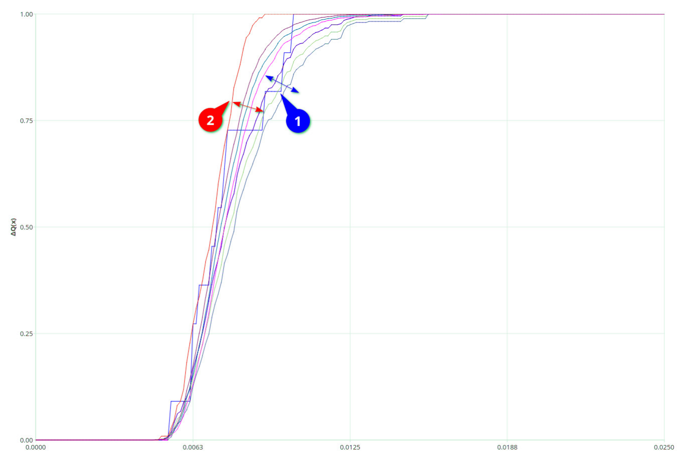


Figure 6.4: probe ΔQs with $\lambda = 150$. (1, blue): The observed ΔQ. (2, red) the calculated ΔQ. Arrow, above, blue: Observed ΔQ confidence bounds. Arrow, below, red: Calculated ΔQ confidence bounds.

High load

Performance degradation at 0.5 offered load is already remarkable, the ΔQs even if the difference is seemingly minimal. We can go even further and observe the system under high load situations. We set $\lambda = 200 \rightarrow \rho = 0.83$, just above the high load threshold.

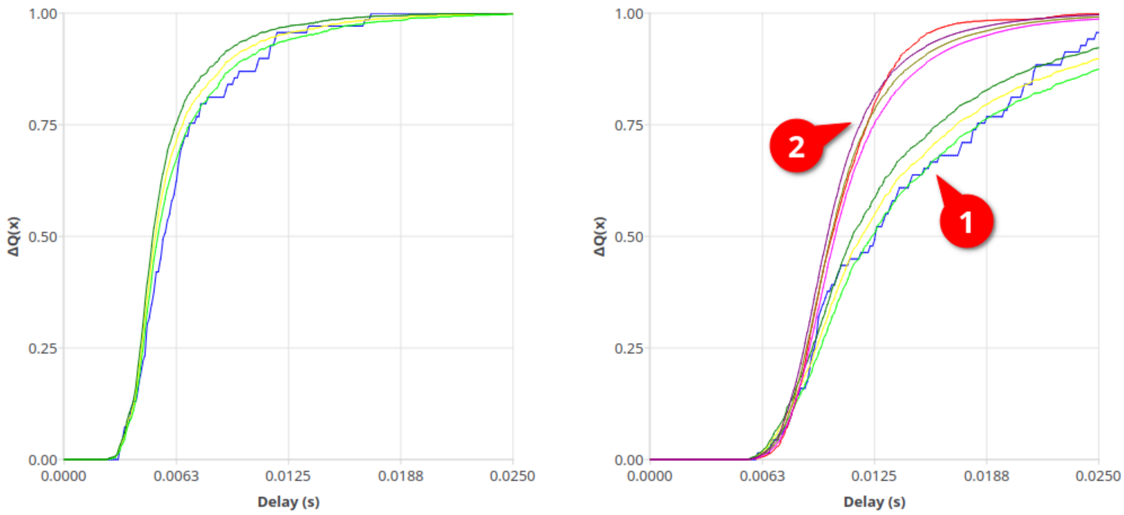
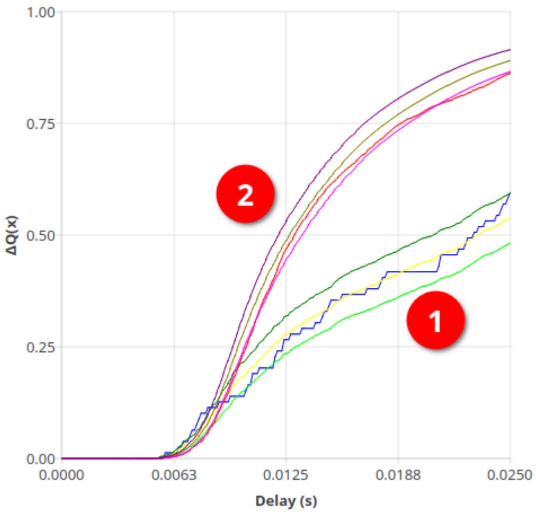


Figure 6.5: Graphs for $\lambda = 250$. Left: worker_1 ΔQ with confidence bounds. Right: probe ΔQ, in blue (1), the observed ΔQ with its confidence bounds, in red (2) the calculated ΔQ with its confidence bounds.

This is what we expected previously, and confirms what is expected by queueing theory, ΔQ is capable of observing the basic observation requirements and capable of recognising dependency. While what is expected by the execution of the queue (observed ΔQ) is a nice normally distributed CDF with little to no failure. What we can actually observe is a degraded performance in both workers and the probe observed execution.

The workers CDF has completely degraded, with the average request taking almost double the time as under normal queueing conditions.

Further degradation can be observed by increasing $\lambda = 300, 350 \rightarrow \rho \geq 1$.



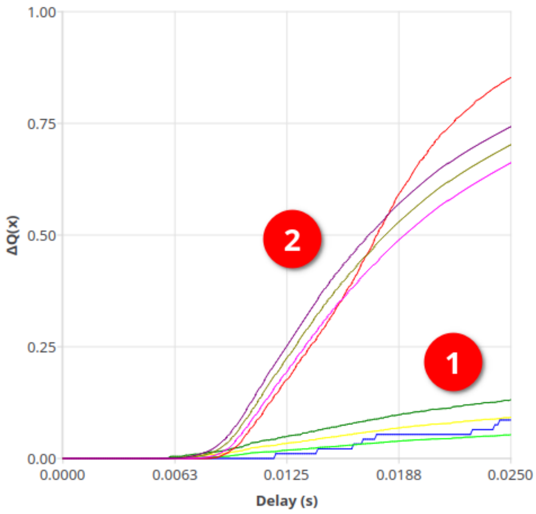


Figure 6.6: Left: (1) probe calculated ΔQ and confidence bounds. (2) probe observed ΔQ and confidence bounds at $\lambda = 300$. Right: probe ΔQs at $\lambda = 350$

The system degrading clear, the ΔQs show how almost all messages are being dropped or take $> dMax$. Let us look at triggers and how they can be useful to diagnose such cases.

Triggers

By observing the system under test in high load cases, we can set the load trigger by setting the sampling window to 1 second and trigger when outcome instances $\gtrapprox 150$. We can also set a trigger based on observation of the running system.

QTA trigger By observing the system, we create a QTA for the probe with: 25% = 0.0075 s, 50% = 0.0125 s, 75% = 0.015s and minimum intangible mass = 0.9.

By setting the trigger to fire for $\Delta Q_{obs} < QTA$. We captured a handful of snapshots. Here, $\lambda = 150$.

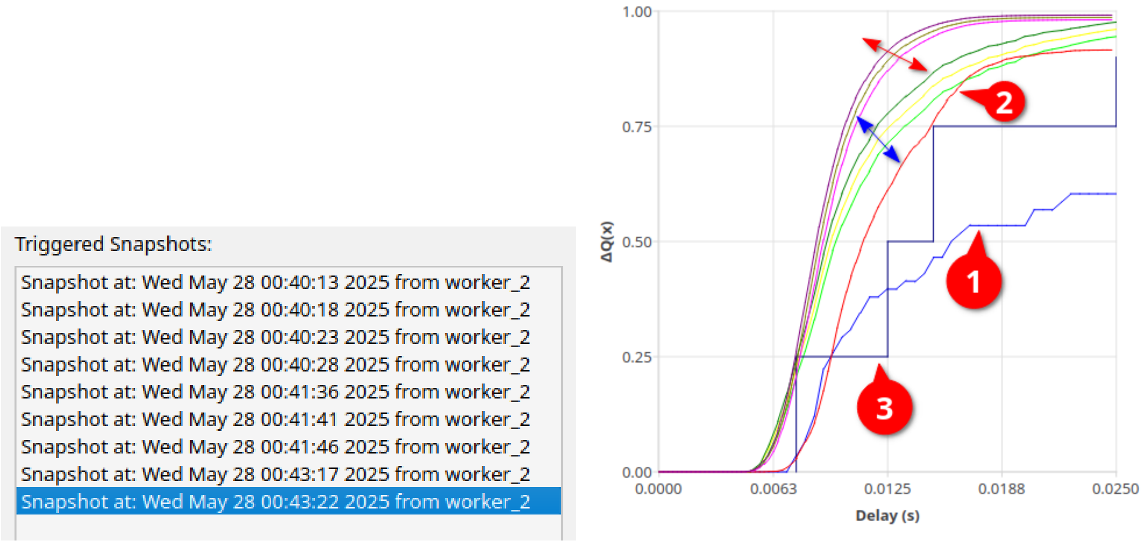


Figure 6.7: Left: Snapshots for fired triggers.
Right: (1) Observed ΔQ violating QTA (3). The confidence bounds for the observed ΔQ (blue, arrow, below) shows the system deteriorating. The calculated ΔQ (2) is behaving worse than its confidence bounds (red, arrow, above).

QTA triggers can help to detect overhead even before high load becomes evident!

Instances trigger By knowing the inner details of the system, setting a QTA on the number of instances can be useful. Here is an example of a fired trigger on the number of instances.

Even though the QTA requirement isn't being violated, the number of instances fires a trigger, where the user can observe that the system is showing early signs of overload.

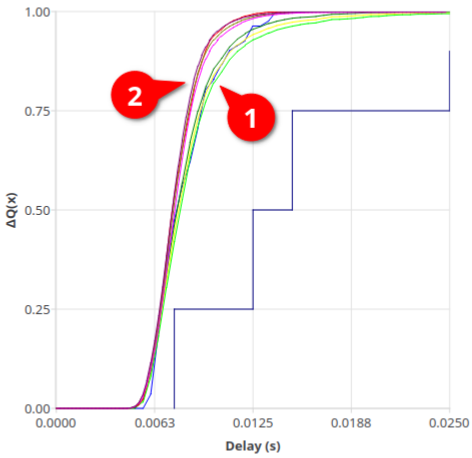


Figure 6.8: Snapshot for a fired trigger observing the load of a probe. The trigger fires for observed load > 150 in a sampling window of 1 second. Even if the QTA requirement (step function) are not being violated, the system is showing early signs of non-linear behaviour.
With knowledge about the system inner workings, smart triggers can be set on load to detect non-linear behaviour.

6.2 Detecting slower workers in operators

6.2.1 First to finish application

Next, we provide a synthetic application modeling an application that can be modeled by a first to finish operator

Why first to finish? Recall the previous FTF graph Fig. 2.5. Assume a send request to "the cloud" that waits for a response or a timeout, it is modeled by a FTF operator. [1]

Using the wrong operator

What happens if the wrong operator is chosen to represent the causal relationships between the outcomes? What if the user believes that the system diagram is the one we presented before Fig. 6.1? The result on the oscilloscope will clearly show that something is wrong!

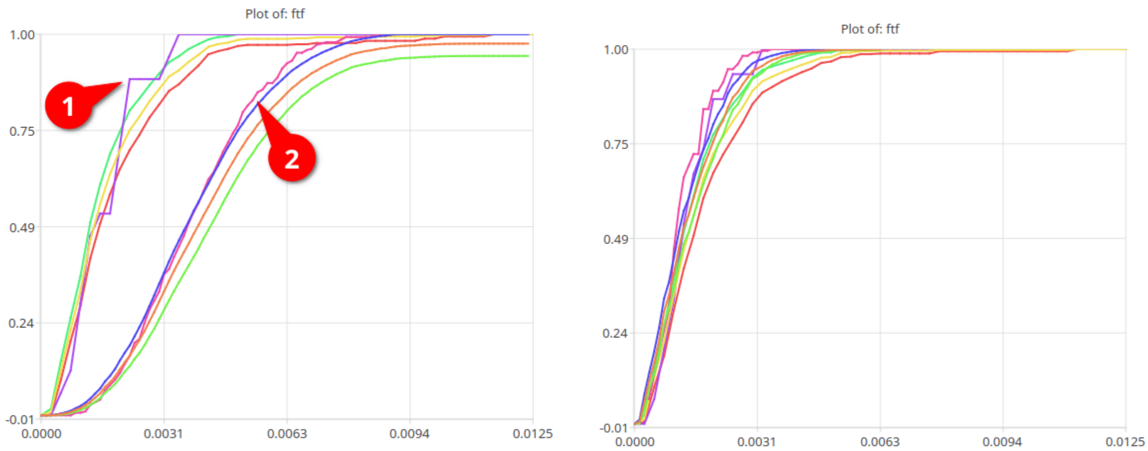


Figure 6.9: *(Left)* FTF plot **with wrong outcome diagram definition** as shown in the oscilloscope. (1) Observed ΔQ. (2) Calculated ΔQ.
(Right) FTF plot **with correct outcome diagram definition** as shown in the oscilloscope. Observed ΔQ and calculated ΔQ overlapping.

On the left, we can observe how the **calculated ΔQ** (2) is clearly greater than the **observed ΔQ** (1). A difference this drastic tells us that the proposed outcome diagram does not correctly represent the actual system. On the right, if no dependencies are present and the correct operator is chosen, the two graphs will overlap.

Introducing a slower component

Let us introduce a slower worker into the system, we introduce an artificial delay into worker_2 (about 20ms). If the oscilloscope works correctly, the paradigm operations are sound and no dependencies are present in the system, we should not see any difference in the observed and calculated ΔQs of the FTF operator.

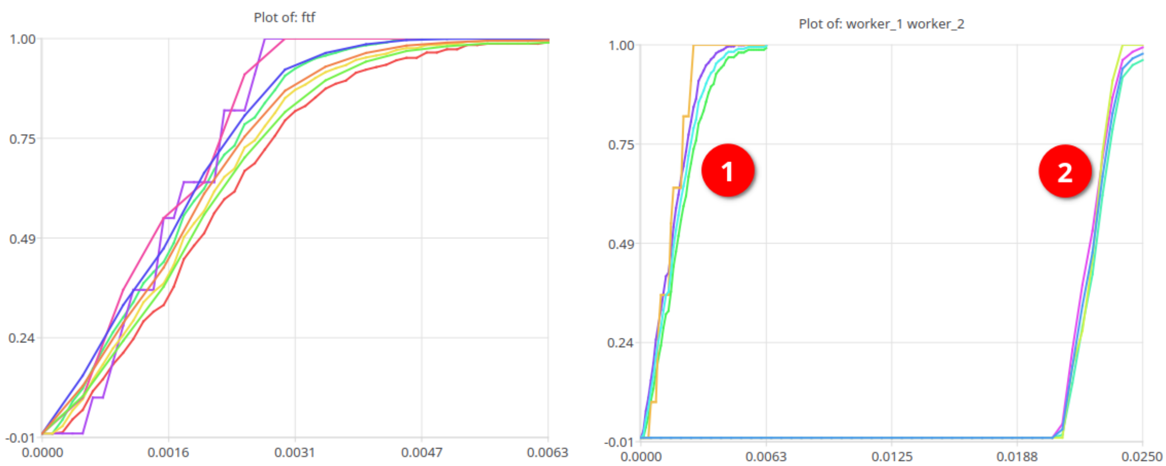


Figure 6.10: *(Left)* FTF plot of worker_1 and worker_2, observed and calculated ΔQ overlapping.
(Right) worker_1 (1) and worker_2 (2) ΔQs.
The FTF plot correctly displays how worker_2 does not have an effect on the ftf plot.

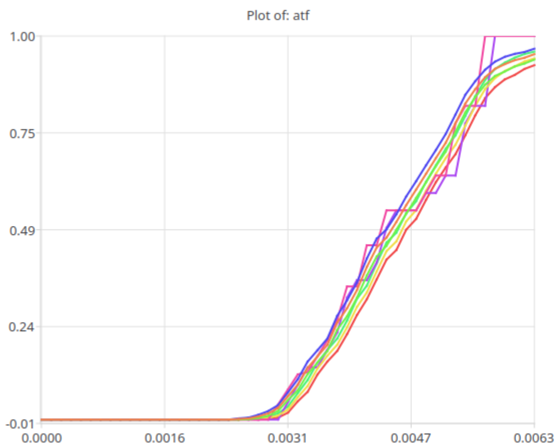
6.2.2 All to finish application

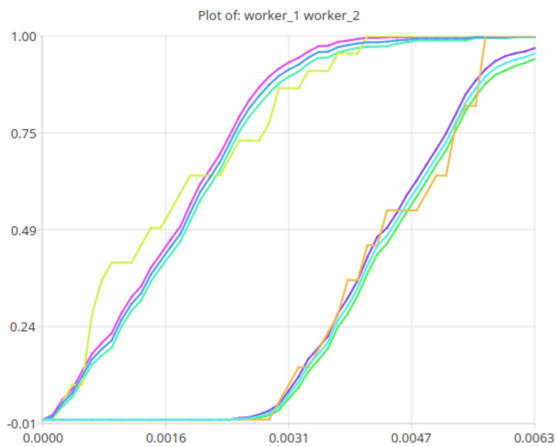
We can extend the previous application to an all-to-finish operator, this operator can for instance parallel work, a task that requires a lot of computation and can be done in separate pieces by separate workers. [1]

Introducing a slower component

Like we did for the FTF operator, let's introduce a slower work into the mix. We introduce a slight delay to show how even a few milliseconds can be noticeable right away by a keen eye (or by triggers, which avoids having to look constantly at the graphs). The delay is a 2ms sleep on worker_2.

The difference in the worker's ΔQ can be noticed with $\Delta Q_{w2} > \Delta Q_{w1}$. The difference can then be observed in the all-to-finish plot, where the operator's ΔQs (both observed and calculated) can be overlaid on top of worker_2 ΔQ, showing once again that the ΔQSD algebraic foundation is sound.





These plots show the usefulness of ΔQSD, the system can be decomposed to understand which part of the system is showing hazards, furthermore, the causal relationships can be observed to determine the behaviour of a part down to the single component.

Chapter 7

Performance study

This chapter evaluates the components and operations we introduced in previous sections, analysing their performances.

- We first evaluate the convolutions algorithms we previously introduced. We stated that naïve convolution would have $\mathcal{O}(n^2)$ time complexity, while FFT convolution would have $\mathcal{O}(n \log n)$ complexity. We will evalute to see if what we observe corresponds to theory.
- We then evaluate the ΔQ adapter API performances, to see the overhead it introduces into a system.
- Lastly, we want to evaluate the QT framework plotting performances, we believe it is the weakest link in the oscilloscope and thus want to evaluate the QtCharts class when plotting ΔQs.

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 ΔQs of equal bins. In theory, we should observe the naïve version delay quickly grow, while the FFT version have a log-linear growth.

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 ΔQ 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. Moreover, if we look closer at the FFT graph, we can observe slight increases after we surpass powers of 2 (e.g at $600 > 512$, $300 > 256 \ldots$). This is because the algorithm is based on ΔQs which are zero-padded to the nearest power of 2, this is to make the calculation more accurate.

While we limit the number of bins to 1000 right now, this limit could potentially be

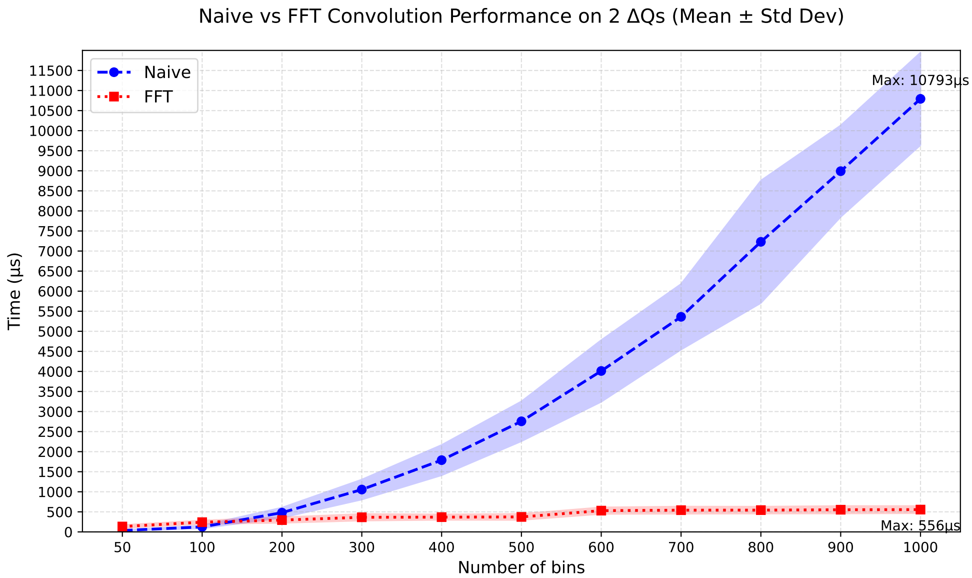


Figure 7.1: Performance comparison of two convolution algorithms

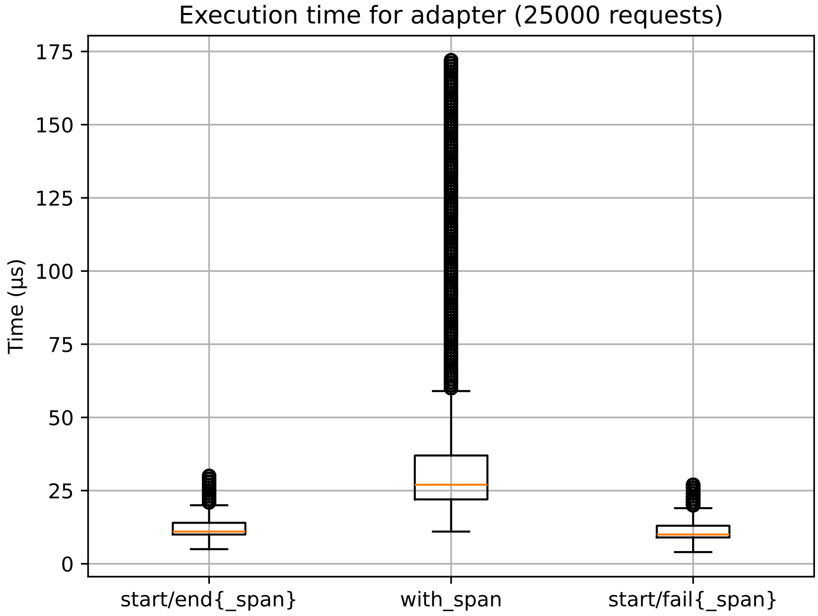
increased as the convolution algorithm is very efficient (0.5 ms for 1024 bins).

7.2 ΔQ adapter performance

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

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

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

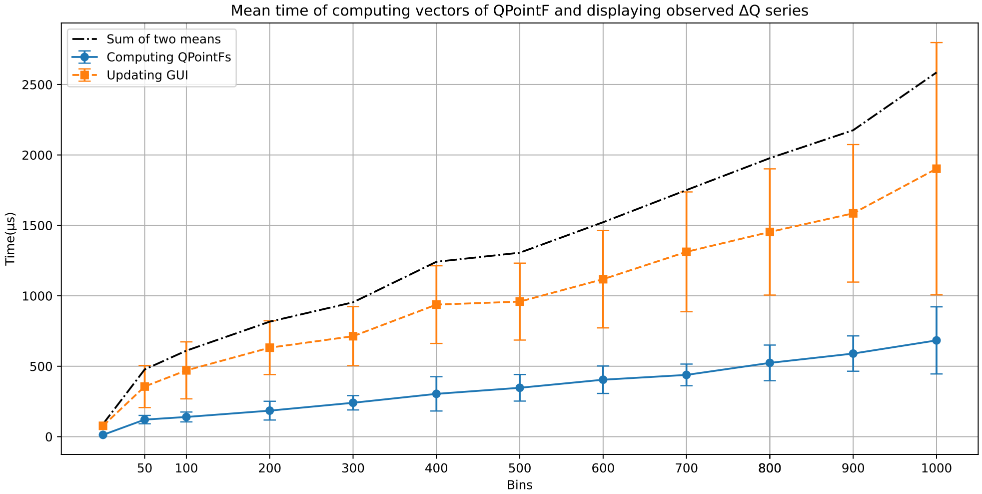


The overhead is minimal, around 10 microseconds on average to start and end/fail a span. The same cannot be said about with span, the increased overhead is nevertheless due to a function needing to be called inside it for it to record a span.

7.3 GUI plotting performance

We evaluated the performance of the GUI plotting routine for an observed ΔQ, the mean and confidence bounds of the polling window of observed ΔQs. The procedure is the same for an observed and calculated ΔQ, so we would need to double the results we have to obtain the total time for the plot The routine first prepares the ΔQs, creating vectors of QPointF (a Qt class representing a point for a QtChart), representing all the x and y values of the ΔQs CDF. The vectors are created for the lower bound, the upper bound, the mean of the window of ΔQs and the observed ΔQ.

Then, once the vectors are prepared, Qt replaces the old points with the new points for every series being plotted.



The result scales up to 2 ms for 1000 bins. We believe that these performances are the choke point of the oscilloscope. If we were to plot the calculated ΔQ and its confidence bounds, the time increase would be twofold. If the sampling rate was 100ms, some frames would probably be skipped if the number of bins = 1000. The results may nevertheless be explained by the specifications of the PC where we ran the tests, namely by the CPU and the GPU (Appendix **??**).

Chapter 8

Conclusions and future work

As we introduced the thesis, its background and current problems we think exist in observability tools, we set out a clear goal: design a graphical dashboard, the **ΔQ oscilloscope**, to observe running Erlang applications and plot the system's behaviour in real time thanks to the ΔQSD paradigm. While we can not say that we are fully finished with the development of the oscilloscope, we can clearly say that a **working prototype** that reflects the theoretical findings of the paradigm and fulfills the initial goals was created.
This was successfully achieved thanks to multiple important implementations which make it fast, reliable and moreover capable of accurately detecting deviations from required behaviour.

The ΔQ adapter, named `dqsd_otel`, an Erlang application which is able to work alongside OpenTelemetry to add the notion of failure according to ΔQSD to spans. The adapter can communicate data about outcome instances from a running system directly to the oscilloscope and can directly receive commands from the oscilloscope. The `gen_server` behaviour allows for this to be done fast and asynchronously.

The ΔQ oscilloscope, a fully fledged Qt dashboard application that is able to observe running systems and provide real time plotting capabilities to the user. Moreover, it provides full control over to the user of the outcome diagrams, the parameters of probes, their QTAs, the triggers they want to set for a given probe, a system of snapshots which allows observing the system as if it was frozen in time. In fine, the FFT convolution algorithms allows us to scale down from $\mathcal{O}(n^2)$ complexity to $\mathcal{O}(n \log n)$, bringing the time to provide precise insights significantly down.

The synthetic applications further prove the oscilloscope's usefulness in detecting early signs of overload and dependent behaviour. This reinforces the solid theoretical basis of ΔQSD, which we remind has already been applied in many industrial projects.

Many crucial features are still missing from the dashboard, and it could require less code modifications in the Erlang side. The next important step of the oscilloscope is its trial in a true distributed application.

8.1 Future improvements

We believe the oscilloscope and the Erlang application can be drastically improved, the size of the project and its intended goal is too big to be encompassed in a single master thesis. We list here some improvements which could be made to both the oscilloscope and the adapter.

8.1.1 Oscilloscope improvements

- The oscilloscope could be turned into a **web app**, we feel that a C++ oscilloscope is a good prototype and proof of concept, but its usability would be greater in a browser context. It would be great as a plugin for already existing observability platforms like Grafana.
- A wider selection of **triggers**, as of writing this thesis, only the QTA trigger and load are available, this is a limitation due to time constraints. Nevertheless, triggers can be easily implemented in the available codebase.
- **Better communication between stub - server - oscilloscope**. The current way of sending outcome instances may be a limiting factor under high load, if hundred of thousands of spans were to be sent, the current way the server and oscilloscope are tied together may throttle communications. TCP socket connections could quickly become the chokepoint which makes the oscilloscope temporarily unusable.

  Future improvements on the server side could implement epoll system server calls to make the server more efficient; **Detaching server from client**, as of right now, the oscilloscope and the server are tied together, using ZeroMQ to assure real time server-client communications could be an interesting solution to explore.
- **Improve real time graphs**. The class QtCharts does not perform correctly with high frequencies update. Moreover, since we are plotting multiple series (from a minimum 4 to a maximum of 9) per probe, which allows up to 1000 bins per probe, the performance quickly degrades with more probes being displayed. A better graphing class for Qt could definitely improve the experience.
- **Saving probe parameters**: As of writing this thesis, there is no way to save the parameters one may have set.
- **Deconvolution**: An important aspect of ΔQSD, which was not introduced in this paper is deconvolution. It is used to check for infeasability in system desing. Since convolution has already been implemented, this could be integrated using the FFTW3 library.
- **Exporting graphs**: The graphs can only be observed in the oscilloscope and have no way to be exported to other programs via standard formats.
- **Many more**: This oscilloscope is just a start, if we were to list everything we may want to add, it would take many pages. What we provide is a sufficient enough basis to provide possibilities to observe a running system and understand

the power of ΔQSD in analysing its behaviour.

8.1.2 Adapter improvements

- As suggested by Bryan Naegele, a member of the observability group of Erlang, the adapter, instead of working on top of OpenTelemetry, could be directly included inside the context of a span by using the ctx library [41], which provides deadlines for contexts, propagating the value in otel_ctx, making it available to the OpenTelemetry span processor. Leveraging `erlang:send_after` as we already do, we could create outcome instances with telemetry events to handle successful executions and timeouts. The span processor will then be responsible for creating outcome instances, without creating the need for custom functions in the adapter, like we have now.

8.1.3 Real applications

A flaw of the oscilloscope and adapter is that they have not been tested on real applications, while their usefulness has been proven on synthetic applications, the lack of real life applications is an evident weakness.

8.1.4 Licensing limitations

Lastly, a notable limitation is created by **Qt**, namely, QtCharts. The usage of Qt does not allow us to release our project under BSD/MIT licenses, but rather a GPLv3 one (we cannot release it under LGPL due to QtCharts). [42]

Bibliography

[1] Peter Van Roy and Seyed Hossein Haeri. *The ΔQSD 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] 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`.

[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] Erlang/OTP. *What is Erlang.* Accessed: (26/05/2025). 2025. URL: `https://www.erlang.org/faq/introduction.html`.

[5] Francesco Nieri. *Master thesis presentation poster UCLouvain - ΔQ oscilloscope.* 2024.

[6] Seyed H. Haeri et al. “Mind Your Outcomes: The ΔQSD 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`.

[7] Peter Van Roy. *LINFO2345 lessons on ΔQSD.* Accessed: (19/05/2025). UCLouvain, 2023. URL: `https://www.youtube.com/watch?v=tF7fbU9Gce8`.

[8] Neil J. Davies and Peter W. Thompson. *ΔQSD workbench - GitHub.* Accessed: (19/05/2025). 2022. URL: `https://github.com/DeltaQ-SD/dqsd-workbench`.

[9] Erlang programming language. *Erlang tracing.* Accessed: (19/05/2025). 2024. URL: `https://www.erlang.org/doc/apps/erts/tracing.html`.

[10] OpenTelemetry Authors. *OpenTelemetry in Erlang/Elixir.* Accessed: (19/05/2025). 2025. URL: `https://opentelemetry.io/docs/languages/erlang/`.

[11] Isaak D. Mayergoyz and W. Lawson (Auth.) *Basic Electric Circuit Theory. A One-Semester Text.* pg. 116. Academic Press, 1996. ISBN: 9780080572284; 0080572286; 9780124808652; 0124808654.

[12] OpenTelemetry Authors. *What is OpenTelemetry?* Accessed: (19/05/2025). 2025. URL: `https://opentelemetry.io/docs/what-is-opentelemetry/`.

[13] Hazel Weakly. *OpenTelemetry Challenges: Handling Long-Running Spans.* Accessed: (21/05/2025). 2024. URL: `https://thenewstack.io/opentelemetry-challenges-handling-long-running-spans/`.

[14] OpenTelemetry Authors. *Instrumentation for OpenTelemetry Erlang/Elixir*. Accessed: (19/05/2025). 2025. URL: https://opentelemetry.io/docs/languages/erlang/instrumentation/.

[15] Erlang Ecosystem Foundation. *Observability Working Group*. Accessed: (28/05/2025). 2025. URL: https://erlef.org/wg/observability.

[16] OpenTelemetry Authors. *Observability primer - Distributed traces*. Accessed: (28/05/2025). 2024. URL: https://opentelemetry.io/docs/concepts/observability-primer/.

[17] OpenTelemetry Authors. *OpenTelemetry - Traces*. Accessed: (19/05/2025). 2025. URL: https://opentelemetry.io/docs/concepts/signals/traces/.

[18] OpenTelemetry Authors. *Exporters*. Accessed: (28/05/2025). 2025. URL: https://opentelemetry.io/docs/languages/erlang/exporters/.

[19] The Jaeger Authors. *Jaeger*. Accessed: (19/05/2025). 2025. URL: https://www.jaegertracing.io/.

[20] Datadog. *Datadog - Product*. Accessed: (28/05/2025). 2025. URL: https://www.datadoghq.com/product/.

[21] 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/.

[22] 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/.

[23] OpenTelemetry Authors. *Active spans, C++ Instrumentation*. Accessed: (19/05/2025). 2025. URL: https://opentelemetry.io/docs/languages/cpp/instrumentation/.

[24] OpenTelemetry authors. *Error handling*. Accessed: (29/05/2025). 2022. URL: https://opentelemetry.io/docs/specs/otel/error-handling/.

[25] Francesco Nieri. *dqsd_otel*. Accessed: (28/05/2025). 2025. URL: https://github.com/fnieri/dqsd_otel.

[26] Francesco Nieri. *ΔQOscilloscope*. Accessed: (28/05/2025). URL: https://github.com/fnieri/DeltaQOscilloscope.

[27] KeySight. *What is an Oscilloscope Trigger?* Accessed: (23/05/2025). 2022. URL: https://www.keysight.com/used/id/en/knowledge/glossary/oscilloscopes/what-is-an-oscilloscope-trigger.

[28] Alex Tsun. *Statistical inference - Confidence intervals*. Accessed: (29/05/2025). 2020. URL: https://courses.cs.washington.edu/courses/cse312/20su/files/student_drive/8.1.pdf.

[29] FM Dekking et al. *A modern introduction to probability theory and statistics*. Vol. Springer texts in statistics. Springer, 2005. ISBN: 1-85233-896-2.

[30] Steven B. Damelin and Willard Miller Jr. *The Mathematics of Signal Processing*. USA: Cambridge University Press, 2012. ISBN: 1107601045.

[31] FFTW3. *Fastest Fourier Transform in The West*. Accessed: (19/05/2025). 2025. URL: https://www.fftw.org/.

[32] Jeremy Fix. *FFTConvolution*. Accessed: (21/05/2025). 2013. URL: https://github.com/jeremyfix/FFTConvolution/blob/master/Convolution/src/convolution_fftw.h.

[33] *Planning-rigor flags*. Accessed: (23/05/2025). URL: https://www.fftw.org/doc/Planner-Flags.html.

[34] Rebar3. *Rebar3 Basic Usage*. Accessed: (25/05/2025). 2025. URL: https://rebar3.org/docs/basic_usage/.

[35] Erlang. *Binary module*. Accessed: (30/05/2025). URL: https://www.erlang.org/docs/24/man/binary.

[36] Erlang System Documentation. *Signals/Sending signals*. Accessed: (24/05/2025). 2025. URL: https://www.erlang.org/doc/system/ref_man_processes.

[37] ANTLR. *What is ANTLR4?* Accessed: (19/05/2025). 2025. URL: https://www.antlr.org/.

[38] 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.

[39] lark-parser. *Lark - A parsing toolkit for Python*. Accessed: (24/05/2025). 2025. URL: https://github.com/lark-parser/lark.

[40] Wikipedia. *Qt (software) Wikipedia, The Free Encyclopedia*. Accessed: (24/05/2025). 2025. URL: https://en.wikipedia.org/wiki/Qt_(software).

[41] Tristan Sloughter. *ctx*. Accessed: (21/05/2025). 2023. URL: https://github.com/tsloughter/ctx.

[42] The Qt Company. *Add-ons available under Commercial Licenses, or GNU General Public License v3*. Accessed: (23/05/2025). 2025. URL: https://doc.qt.io/qt-5/qtmodules.html.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**