École polytechnique de Louvain

Observing the detailed behaviour of large distributed applications in real time using $\triangle$QSD

Author: **Francesco Nieri**
Supervisor: **Peter Van Roy**
Readers: **Tom Barbette, Peer Stritzinger, Neil Davies**
Academic year 2024–2025
Master [120] in Computer Science

Contents

Chapter 1

Background

1.1 Observability

Observability refers to the ability to understand the internal state by examining its output, in the context of a distributed system, being able to understand the internal state of the system examining its telemetry data. [CITE OTEL]
In the case of the Erlang programming language, we explain below two tools that can be used to observe an Erlang program.

1.1.1 erlang:trace

The Erlang programming language gives the users different ways to observe the behaviour of the code, one of those is the function `erlang:trace/3`. The erlang run-time system exposes several trace points that can be observe, observing the trace points allows users to be notified when they are triggered [CITE CITE CITE]. 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 1.1: erlang:trace\3 specification

Nevertheless, Erlang Tracing, according to our use case, has a major flaw: no notion of causality. If two messages a, b are sent and then received in disorder, the tracer has no default way of knowing which is which, this is a missing feature that is crucial for observing a program functioning and being able to connect an application to our oscilloscope. This is where the OpenTelemetry standard comes in.

1.1.2 OpenTelemetry

OpenTelemetry is an open-source, vendor agnostic observability framework and toolkit designed to generate, export and collect telemetry data, in particular traces, metrics and logs.
OpenTelemetry is available for a plethora of languages, including Erlang, although, as of writing this, only traces are available in Erlang.
OpenTelemetry provides a standard protocol, a single set of API and conventions and lets you own the generated data, allowing to switch between observability backends freely. The Erlang Ecosystem Foundation has a working group focused on evolving the tools related to observability.

Traces

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

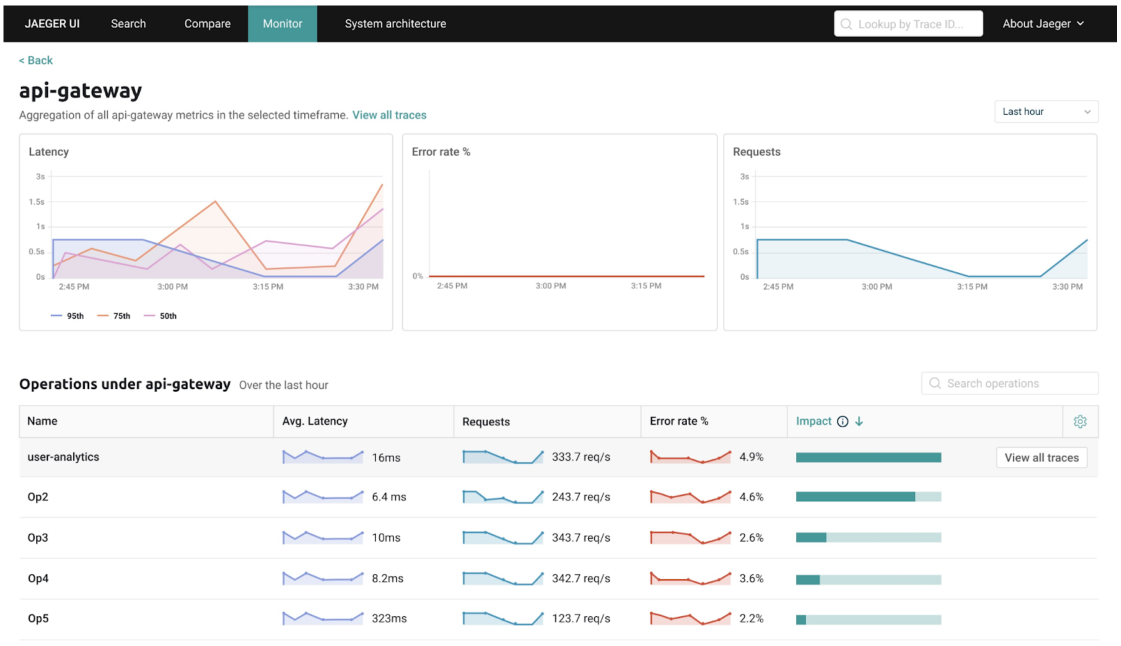
Span A span is a unit of work or operation. Spans can be correlated to each other and can be assembled into a trace. The notion of spans and traces allows us to follow the execution of a "request" and carry a context, allowing us to get the causal links of messages[cite cite].

```
{
  "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 1.2: Example of span with a parent, indicating a causal link between parent and children span

Exporters

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



api-gateway

Aggregation of all api-gateway metrics in the selected timeframe. View all traces

Operations under api-gateway Over the last hour

Name	Avg. Latency	Requests	Error rate %	Impact
user-analytics	16ms	333.7 req/s	4.9%	
Op2	6.4 ms	243.7 req/s	4.6%	
Op3	10ms	343.7 req/s	2.6%	
Op4	8.2ms	342.7 req/s	3.6%	
Op5	323ms	123.7 req/s	2.2%	

Figure 1.3: Jaeger interface example

Macros

OpenTelemetry provides macros to start, end and interact with spans.

?with_span `?with_span` creates active spans (explain what an active span is, no ref)

```
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`

1.2 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. Outcome diagrams capture observational properties of the system. The ΔQSD paradigm derives bounds on performance expressed as probability distribution, encompassing all possible executions of the system.
The paradigm was developed by PNSol Ltd. over 30 years ago and has been widely used in large industrial projects with great success and large savings.
The following sections are a summary of multiple articles and presentation formalizing the paradigm.

1.2.1 Outcome

An outcome O is a specific system behaviour with a start event s and end event e, formally, what the system obtains by performing one of its tasks. One task corresponds to one outcome and viceversa. When an outcome is performed, it means that the task of an outcome is performed.
The distinct set of starting and terminating events is called the observables. Once an observable from the starting set occurs, there is no guarantee that a corresponding event in the terminating set will occur within the duration limit (required time to complete). An observable is *done* when it occurs during the time limit.
Outcome are represented as circles, with the starting and terminating set of events being represented by boxes.v

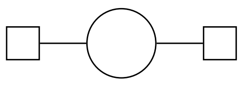


Figure 1.4: The outcome (circle) and the starting set (left) and terminating set (right) of events

1.2.2 Probes

While outcomes have distinct starting and terminating events, a probe can be "placed" anywhere in a system to observe the behavior of a part of the system. They share the set of starting events of the first outcome of the part of the system they observe and the terminating set of events of the last outcome of the system they observe. A probe can observe at least one outcome.

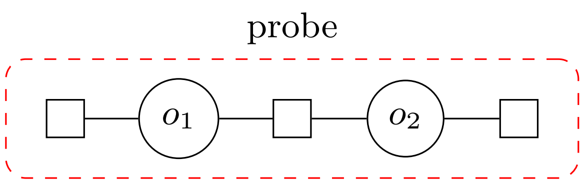


Figure 1.5: A probe that observes the sequential composition of o_1 and o_2, from the starting set of events of o_1 to the terminating set of o_2.

1.2.3 Quality attenuation (ΔQ)

In an ideal system, an outcome would deliver a desidered behaviour without error, failure, delay, but this is not the case. The quality of an outcome response "attenuated to the relative ideal" is called "quality attenuation" (ΔQ). Since it can depend on many factors (geographical, physical . . .), ΔQ is modeled as 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 delay < 1.
ΔQ(x) will be the probability that an outcome O will occur 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.

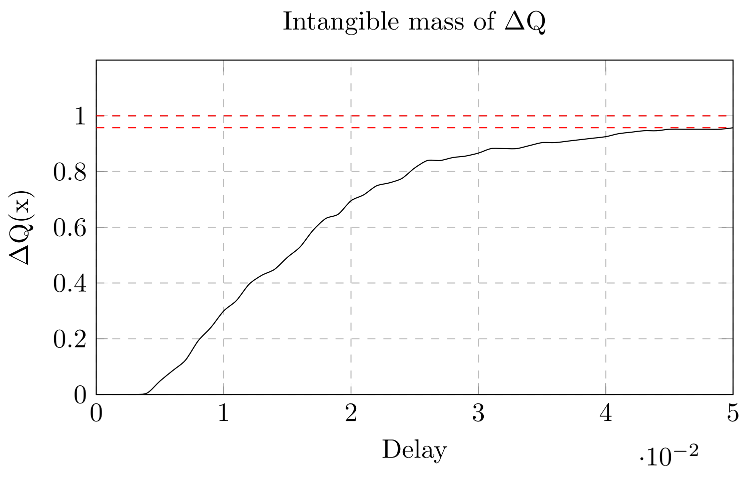


Figure 1.6: Intangible mass (red) of a ΔQ, the observable had a failure rate of about 5%

1.2.4 Timeliness

Timeliness is defined as a relation between an observed ΔQ_{obs} and a required ΔQ_{req}

1.2.5 QTA, required ΔQ

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

QTA example : Imagine a system where 50% of the executions should take < 5 ms, 75% of executions should take < 7.5 ms, all queries have a maximum delay of 10ms and 5% of executions can timeout, the QTA can be represented as a step function.

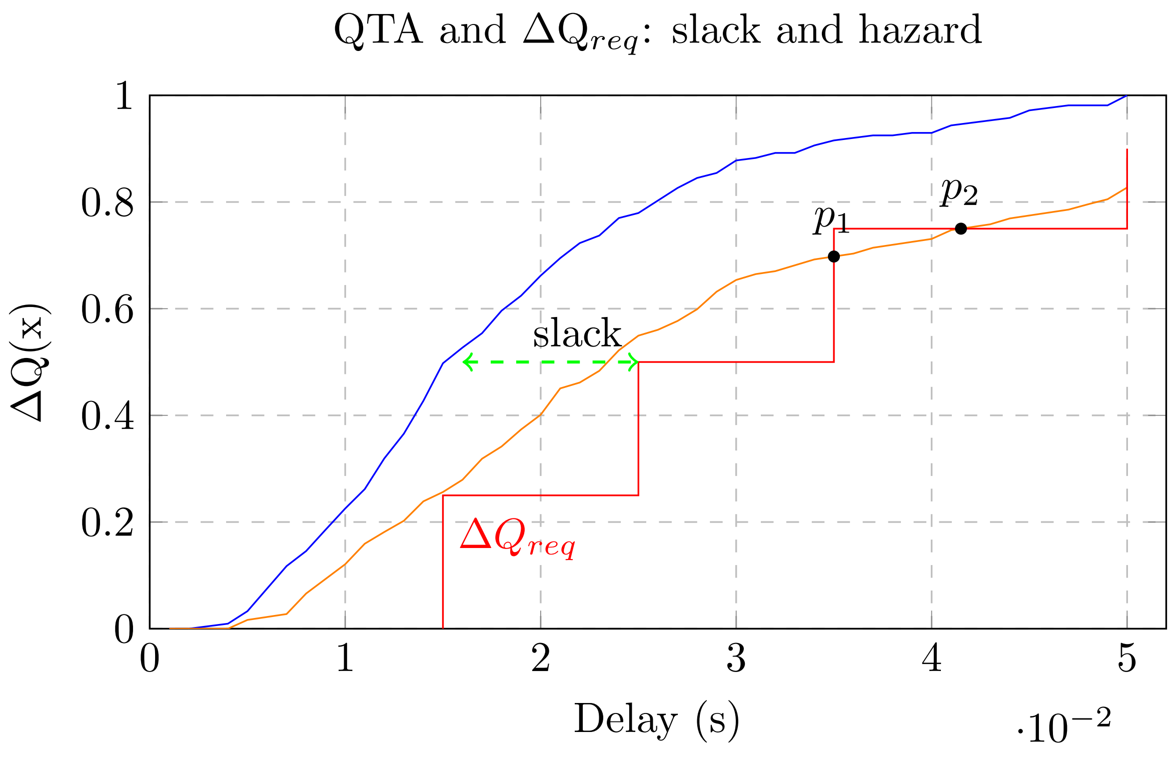


Figure 1.7: The system in blue is showing slack and satisfies the requirement, the system in orange is showing signs that it cannot handle the stress, it is not respecting the ΔQ_{req}

1.2.6 Outcome diagram

An outcome diagram captures the causal relationships between the outcomes, each outcome diagram can be presented algebraically with an outcome expression, there are four different ways to represent the relationships between outcomes.

Sequential composition

If we assume two outcomes O_A, O_B where end event of O_A is the start event of O_B, the probability distribution of O_A and O_B is given by the convolution of the probability distributions (PDF) of O_A and O_B. Where convolution ($\circledast$) between two PDF is :

$$PDF_{AB}(t) = \int_0^t PDF_A(\delta) \cdot PDF_B(t-\delta) d\delta \tag{1.1}$$

and thus ΔQ_{AB}:

$$\Delta Q_{AB} = \Delta Q_A \circledast \Delta Q_B \tag{1.2}$$

First to finish

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

$$\Delta Q_{FTF(A,B)} = \Delta Q_A + \Delta Q_B - \Delta Q_A \cdot \Delta Q_B \tag{1.3}$$

All to finish

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

$$\Delta Q_{ATF(A,B)} = \Delta Q_A \cdot \Delta Q_B \tag{1.4}$$

Probabilistic choice

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 $\dfrac{q}{p+q}$

$$\Delta Q_{PC}(A,B) = \frac{p}{p+q}\Delta Q_A + \frac{q}{p+q}\Delta Q_B \tag{1.5}$$

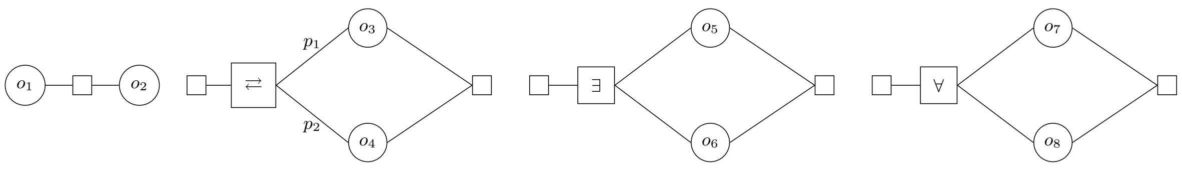
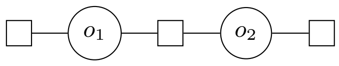


Figure 1.8: The possible relationships in an outcome diagram: Sequential composition, probabilistic choice, first-to-finish, all-to-finish

1.2.7 Independence hypothesis

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



At low load, the two components behavior will be independent, the system will behave linearly, the observed delay of the probe p will be equal to the convolution of o_1, o_2 ($o_1 \circledast o_2$).
When load increases, the two components will start to show dependent behaviour due to the processor utilisation increasing, the observed ΔQ will then deviate from what is calculated.

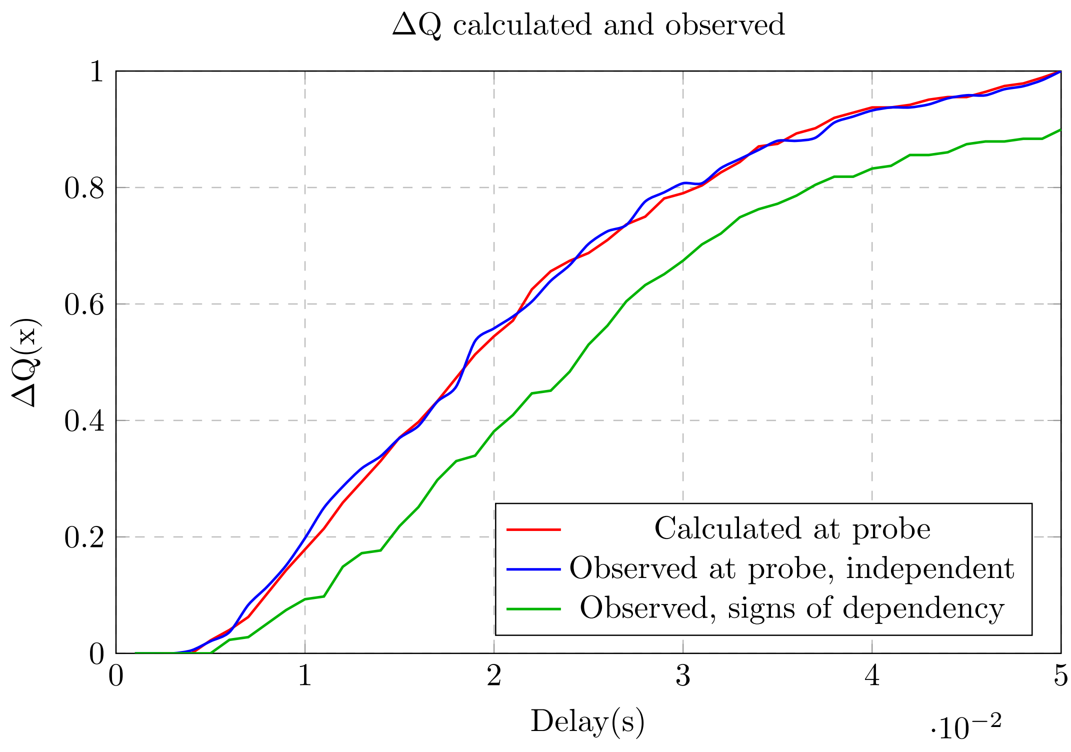


Figure 1.9: When the components are independent, what is observed (blue) and calculated (red) can be superposed, whilst when o_1 and o_2 show initial signs of dependency, what is observed (green) can be seen deviating from the calculated ΔQ.

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

Chapter 2

Design

2.1 Scenario

We can define four parts of interest for our tool.

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

Stub/wrapper The stub is the `otel_wrapper`, a wrapper that starts and ends OpenTelemetry spans, and start custom spans which are useful for the oscilloscope. Further handling of OpenTelemetry spans is delegated to the user, who may wish to do further operations with their spans. The custom spans can be ended normally like OpenTelemetry spans or can timeout, given a custom timeout, and fail, according to user's definition of failure.
The stub is called from the system under test and communicates spans data to the oscilloscope via TCP.
The stub can receive messages from the oscilloscope, the messages are about updating observable's $dMax$.

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

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

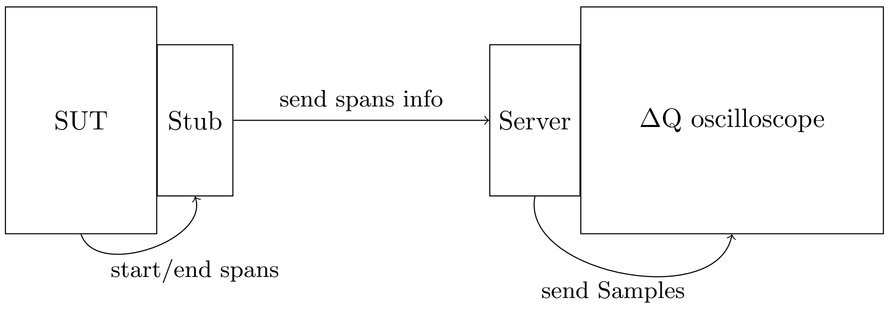


Figure 2.1: Global system diagram: the SUT calls the wrapper when starting, ending and failing spans. The spans are received by the server which processes them and sends them to the oscilloscope. The server communicates with the stub to update informations about the system under test ($dMax$)

Chapter 3

Implementation

3.1 Wrapper

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

3.1.1 API

The wrapper functions to be used by the user are made to replace OpenTelemetry calls to macros as for `?start_span` and `?with_span` and `?end_span`. This is to make the wrapper less of an encumbrance for the user.
Moreover, the wrapper will always start OpenTelemetry spans but only start custom spans if the stub has been activated. The wrapper can be activated by: WIP

start_span/1, start_span/2

```
start_span/1: -spec start_span(binary()) -> {opentelemetry:span_ctx(),
↪  pid() | ignore}.
start_span/2: -spec start_span(binary(), map()) ->
↪ {opentelemetry:span_ctx(), pid() | ignore}.
```

Parameters:

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

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

Return: The function returns either:

- `{SpanCtx, span_process_PID}` if the wrapper is active and the observable's $dMax$ has been set
- `{SpanCtx, ignore}` if one of the two previous conditions was not respected.

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

with_span/1, with_span/2

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

Parameters:

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

`with_span` incorporates OpenTelemetry `with_span` macro.

Return: `with_span` returns what `Fun` returns (`any()`).

end_span

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

Parameters:

- SpanCtx: The context of the span returned by `start_span`.
- Pid: `span_process_PID` || `ignore`

As is the case for `start_span`, `end_span` incorporates an OpenTelemetry macro, in this case `?end_span(Ctx)`.

fail_span

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

Parameter:

- Pid: `ignore` || `span_process_PID`

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

span_process

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

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

The process is able to receive one and only message, if the execution times out and subsequently the span is ended, the oscilloscope will not be notified as the process is defunct. This is assured by Erlang documentation:
If the message signal was sent using a process alias that is no longer active, the message signal will be dropped.

3.1.2 Handling sample spans

To create custom spans we must obtain three important informations:

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

They are all supplied upon starting a span with `otel_wrapper:[start_span, with_span]`. The custom span is created only if two conditions are met: the wrapper has set as active and the user set a timeout for the name of the observable, the functions will spawn a `span_process` process, passing along all the necessary informations.
Once the span is subsequently ended/timed out/failed, the function `send_span` creates a message carrying all the informations and sends it to the C++ server. The formatting of the messages is the following:

```
n:Observed name, b: Start time (beginning), e: End time (if end_span wa
```

3.2 Parser

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

3.2.1 ANTLR

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

3.2.2 Grammar

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

```
grammar DQGrammar;

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

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

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

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

component
    : behaviorComponent
    | probeComponent
    | outcome
;

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

probeComponent
    : PROBE_ID ':' IDENTIFIER
;
```

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

outcome: IDENTIFIER;
```

Limitations

A previous version was implemented in Lark[cite], a python parsing toolkit. The python version was quickly discarded due to a more complicated integration between Python and C++. Lark provided Earley(SPPF) strategy which allowed for ambiguities to be resolved, which is not possible in ANTLR.
For example the following system definition presents a few errors:

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

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

3.3 Oscilloscope

Our oscilloscope graphical interface has been built using the QT framework for C++. Qt is a cross-platform application development framework for creating graphical user interfaces.
We chose Qt as we believe that it is the most documented and practical library for GUI development in C++, using Qt allows us to create usable interfaces quickly, while being able to easily pair the backend code of C++ to the frontend.

Chapter 4

Oscilloscope

4.1 ΔQ implementation

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

For a given observable, ΔQ(t_l, t_u, $dMax$) is the probability that the time of series with samples between time $t_l < t_u$, an observable occurs in time t ≤ dMax.

4.1.1 Internal representation of a ΔQ

We provide a ΔQ class to calculate the ΔQ of an observable between a lower time bound t_l and an upper time bound t_u. The ΔQ can be calculated in various ways:
The first way is by having n collected samples between t_l and t_u and calculating its PDF and then calculating the *empirical cumulative distribution function* (ECDF) based on its PDF. A ΔQ can also be calculated by performing operations on two or more ΔQs, the notion of samples is then lost between calculations, as the interest shifts towards the calculate PDFs and ECDFs. Below, is how both are calculated given n samples.

PDF

We approximate the PDF of the observed random variable **X** via a histogram. We partition the values into N bins of equal width, this is required to ease future calculations. Given $[x_i, x_{i+1}]$ the interval of a bin i, where $x_i = i\Delta x$, and $\hat{p}(x_i)$ the value of the PDF at bin i, for n bins:

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

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

ECDF

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

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

4.1.2 dMax

The key concept of ΔQSD is having a maximum delay after which we consider that the execution is failed, this is represented in an observable as $dMax$. The user defines, for each observable the maximum delay its execution can have.
Setting a maximum delay for an observable is not a job that can be done one-off and blindly, it is something that is done with an underlying knowledge of the system inner-workings and must be thoroughly fine tuned during the execution of the system by observing the resulting distributions of the obtained ΔQs.

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

$$dMax = \Delta_{tbase} * 2^n * N \tag{4.3}$$

Where:

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

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

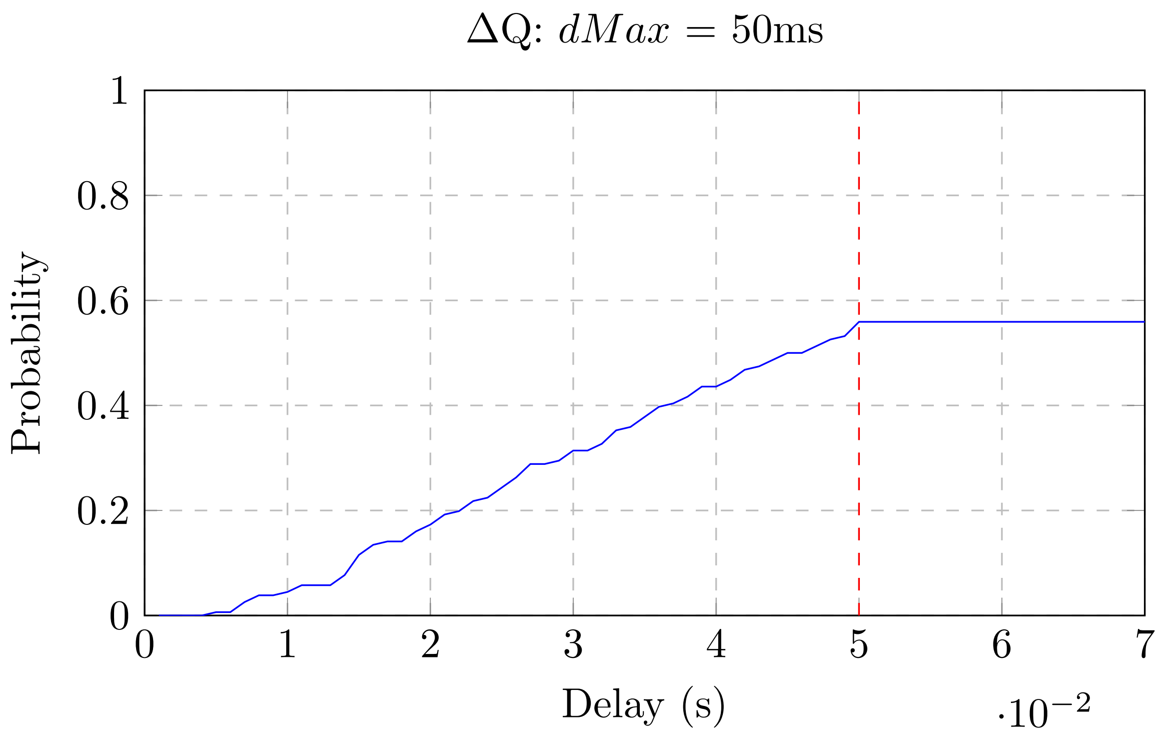


Figure 4.1: ΔQ: $dMax$ = 50ms, the CDF will stay constant when delay > $dMax$

4.1.3 QTA

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

4.1.4 Operations

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

Convolution

Naïve convolution Given two ΔQ binned PDFs f and g, the result of the convolution $f \circledast g$ is given by

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

Fast Fourier Transform Convolution FFTW (Fastest Fourier Transform in the West) is a C subroutine library [cite site] 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. Whilst the previous algorithm is far too slow to handle a high number of bins, convolution leveraging Fast Fourier Transform (FFT) allows us to reduce the amount of calculations to $\mathcal{O}(n\log n)$. FFT and naïve convolution produce the same results in our program barring ε differences (around 10^{-18}) in bins whose result should be 0.

Arithmetical operations

We can apply a set of arithmetical operations between ΔQs ECDFs, and on a ΔQ.

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

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

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 ECDFs of the ΔQs:

$$\Delta\mathrm{Q}_{AB}(i) = \hat{f}_A(i)[\cdot, +, -]\hat{f}_B(i) \tag{4.6}$$

4.1.5 Confidence bounds

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

$$\mu_i = \frac{1}{n_i}\sum_{j=1}^{n_i} x_{ij} \tag{4.7}$$

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

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

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

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

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

4.1.6 Rebinning

Rebinning refers to the aggregation of multiple bins of a bin width i to another bin width j.
Operations between ΔQs can be done on ΔQs that have the same bin width, this is why it is fundamental that all observables have a common Δ_{tbase}. This allows for fast rebinning to a common bin width.
Given two ΔQs ΔQ_i, ΔQ_j:

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

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

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

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

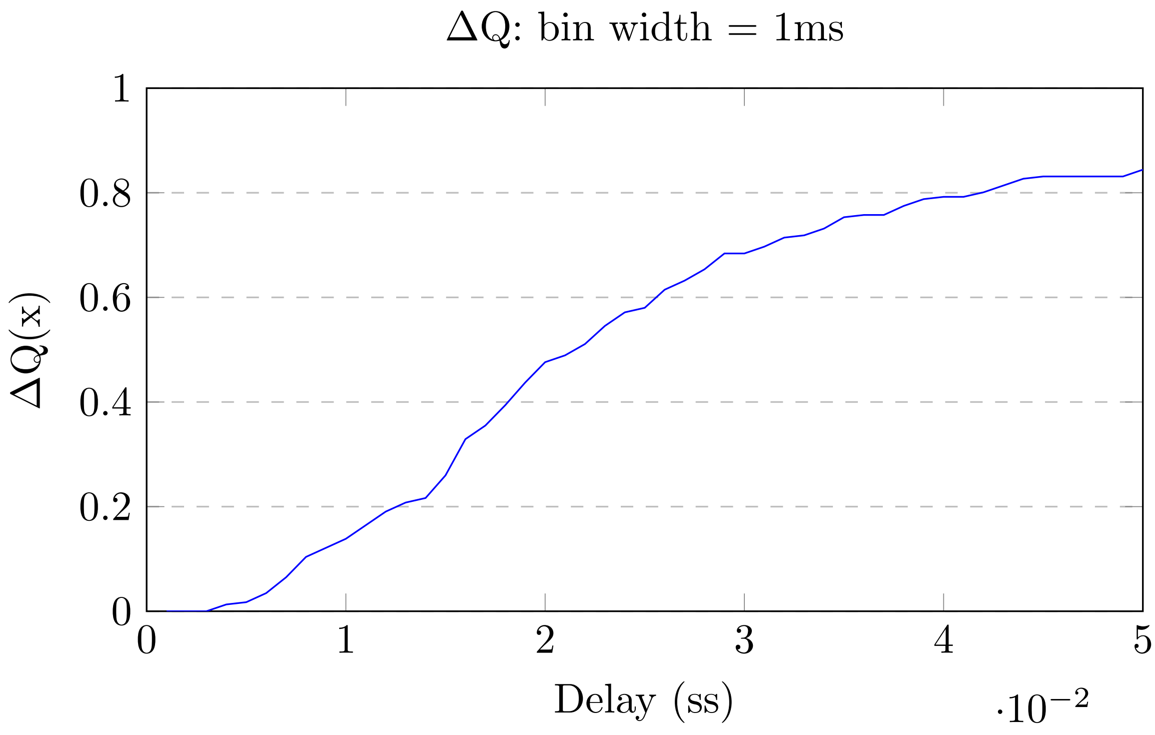


Figure 4.2: Sample ΔQ with 1ms bins

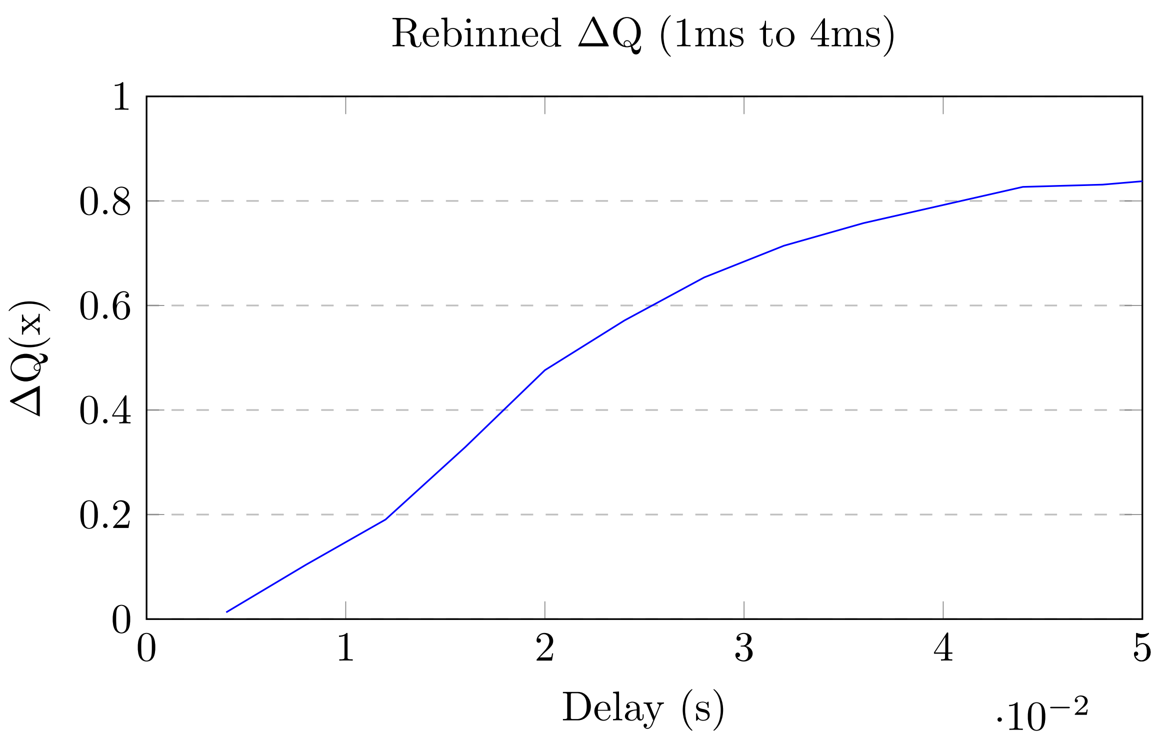


Figure 4.3: Previous ΔQ after rebinning to 4ms bins

4.2 ΔQ display

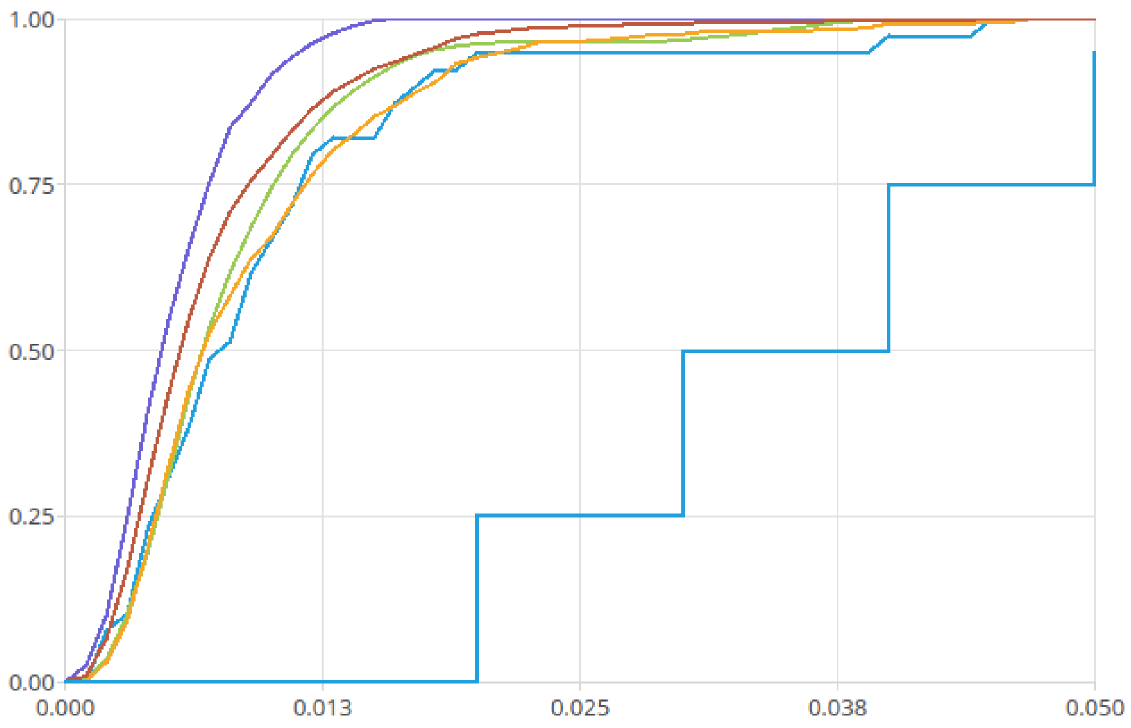
An observable's displayed graph must contain the following functions:

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

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

4.3 Outcome diagram

An abstract syntax for outcome expressions has already been defined in a previous paper, nevertheless, the oscilloscope provides additional features not included in the original syntax and, moreover, needs a textual way to define an outcome diagram.
We define thus a grammar to create an outcome diagram in our oscilloscope, our grammar is a textual interpretation of the abstract syntax.



4.3.1 Observables

Below is a way to define the observables in a system.

Outcome

In the definition of the system or of a probe, an outcome is defined with its name

```
probe = outcomeName;
```

Probes

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

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

Probes will not be parsed after the system has been defined, in the case below, an error will be thrown.

```
probe = ...;
probe2 = ...;
system = ...;
probe3 = ...;
```

Proes can be reused in other probes or in the system by adding a s: before they are used.

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

4.3.2 Operators

To build a system, we must define some operators, below is how they can be defined. About first-to-finish, all-to-finish and probabilistic choice, they must contain at least two components, this is because the operations to calculate the DeltaQ of these operators rely on using the CDF of the components that define the operator.

Causal link

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

```
component_i -> component_j
```

All-to-finish operator

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

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

First-to-finish operator

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

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

Probabilistic choice operator

A probabilistic choice operator needs to be defined as follows:

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

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

4.3.3 Limitations

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

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

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

4.3.4 Time series

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

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

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

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

The stub will send a standard message with the following format:

```
"n:name,s:start_time,e:end_time,s:status"
The status can be "ok", "fa", "to"
```

The C++ receiving server will then calculate the elapsed time if the execution was successful.

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 two tabs, in one tab the user can interact with the system, in the other tab the user can place triggers on the observabes and observe the fired ones.

System tab

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

Adding a plot The user can decide to display the plot of an observable or not, he's given the opportunity to add them to the display window (decide how).

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

dMax, bins The user has a slider which goes from -10 to 10, where he can set the exponent of Δ_T and another field where he can set the number of bins. When these informations are saved by the user, the new $dMax$ is transmitted to the stub and saved for the selected observable.

Trigger tab

In the trigger tab the user can set triggers and observe those that have been fired.

Set triggers The user can set which triggers to fire for a certain observable, he is given checkboxes to decide which ones to set as active or not (by default, the triggers are deactivated).

Fired triggers WIP.

4.5 Triggers

Like an oscilloscope that has a trigger mechanism that fires when a signal of interest is recognized by the oscilloscope, our ΔQ oscilloscope has a similar mechanism of triggers that are fired when an observed ΔQ violates certain conditions. Let us define what these triggers are.

4.5.1 Load

A trigger on an observed ΔQ can be fired if

$$\mathrm{nSamples}(\Delta\mathrm{Q}(t_l, t_u, dMax)) > \mathrm{maxAllowedSamples}$$

4.5.2 QTA

There are two possible [can change] triggers that can be fired based on the observable defined ΔQ's QTA.

Percentiles

A trigger can be fired if:

$$\Delta\mathrm{Q}_{obs}[\text{percentile}] < \text{observableQTA}_{req}[\text{percentile}] \quad \forall \text{ percentile } \{0.25, 0.5, 0.72\}$$

Failure

A trigger can be fired on the percentage of failed samples for $\Delta\mathrm{Q}(t_l, t_u, dMax)$ if:

$$\text{success}(\Delta\mathrm{Q}_{obs}) < \text{success}(\text{observableQTA}_{req})$$

4.5.3 Time series snapshots

When a trigger is fired, the oscilloscope will capture ...

Chapter 5

Application on synthethic programs

5.1 M/M/1/K queue

We devised a simple Erlang system that can act as a simple M/M/1/K. The system has two components `worker_1`, `worker_2`, the components are made of a buffer queue of size K and a worker process. The system sends n messages per second following a Poisson distribution to `worker_1`'s queue, the queue then reduces its available buffer size.
The buffer notifies its worker, which then does N loops, which are defined upon start, of fictional work. The worker then passes a message to `worker_2`'s queue, which has another queue of same size, who passes the message to `worker_2`'s worker, which does the same amount of loops. When a worker completes its work, it notifies the queue, freeing one "message" from its buffer size.
If the queue's buffer is overloaded, it will drop the incoming message and consider the execution a failure.
A probe p is defined, which observes the execution from when the first message up until `worker_2` is done.

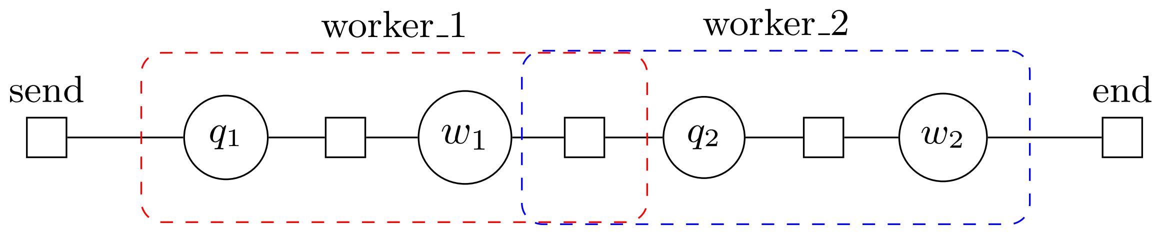


Figure 5.1: Outcome diagram of the M/M/1/K queue

5.2 Cache

Chapter 6

Performance study

6.1 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.

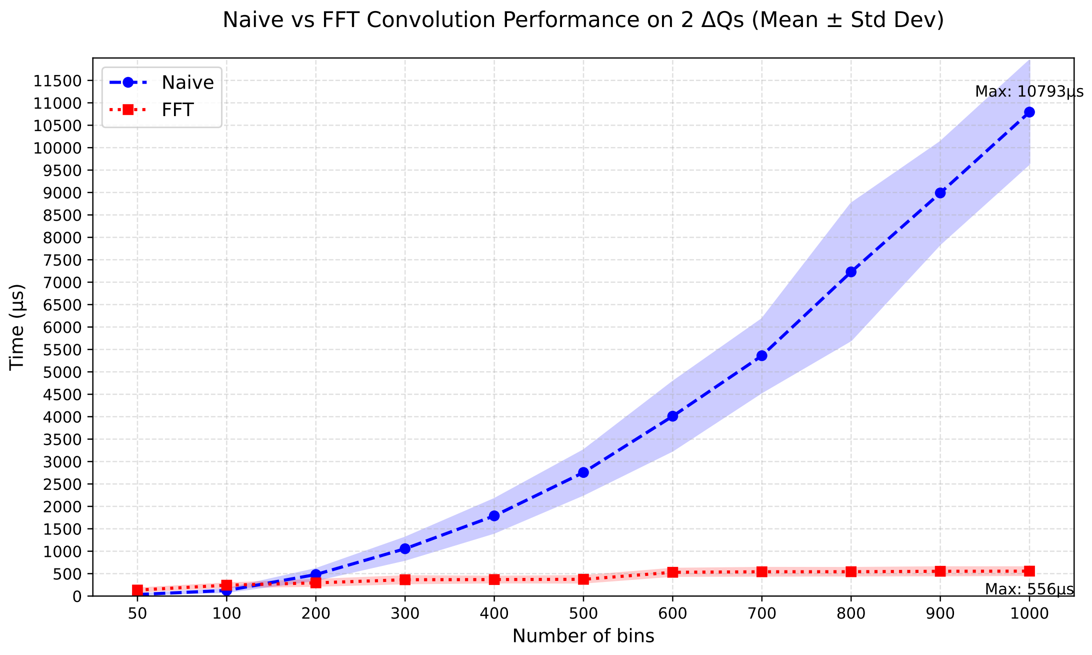


Figure 6.1: Performance comparison of two convolution algorithms

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

6.2 Server performance

6.3 Stub performance