SCALABLE HIGH PERFORMANCE STREAMING PROCESSING APPLICATION INSTRUMENTATION FRAMEWORK VIA IMPROVING DYNAMIC CONCURRENCY.

S. R. M. D. T. S. Wijesekara

179360M

Degree of Master of Science in Computer Science

Department of Computer Science and Engineering
University of Moratuwa.
Sri Lanka

May 2019

SCALABLE HIGH PERFORMANCE STREAMING PROCESSING APPLICATION INSTRUMENTATION FRAMEWORK VIA IMPROVING DYNAMIC CONCURRENCY.

S. R. M. D. T. S. Wijesekara

(179360M)

Thesis submitted in partial fulfillment of the requirements for the degree Master of Science, Specialized in Software Architecture

Department of Computer Science and Engineering

University of Moratuwa.

Sri Lanka

May 2019

DECLARATION

I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: ..... Date:.....

Name: S. R. M. D. T. S. Wijesekara

The supervisor/s should certify the thesis/dissertation with the following declaration.

I certify that the declaration above by the candidate is true to the best of my knowledge and that this report is acceptable for evaluation for the CS5999 PG Diploma Project.

Signature of the supervisor: ..... Date: .....

Name: Dr. Indika Perera.

ABSTRACT

The world has already moved to a highly technological stage and internet-based services plays a vital part of day to day life. Performance of those internet based services is a key factor of quality of the service and developers are forced to develop the best possible performant system. Usually gaining the best possible performance is hard due to low visibility and flexibility of the system in performance improvement phase.

This research is focusing on developing the framework ‘concor: A framework for high performance streaming applications, instrumentation in-built’ by combining the pre-placing instrumentation probes and data flow based architectures. The framework provides an API to form data flows, while providing in-built performance monitoring capabilities. Furthermore, the possibility of implementing a dynamic thread reconfiguration mechanism is also researched and included in the framework. Dynamic thread reconfiguration mechanism is used in simplifying the bottleneck isolation. Apart from this, dynamic thread configuration mechanism effectively lifts the initial concurrency design overhead from the developers and provides a new dimension of runtime performance tuning.

Keywords: Instrumentation, Concurrency framework, Dynamic concurrency, Runtime performance tuning, dynamically assigned thread pools. Bottleneck identification, data-flow architecture, event streaming.

ACKNOWLEDGEMENT

I would like to express my gratitude and appreciation to my project supervisor Dr. Indika Perera, for the knowledge, guidance, encouragement and suggestions throughout this research project, which was a drive force behind completion of this work on time.

I would like to thank my current company hSenid Mobile Solutions, especially to CEO, Mr. Dinesh Saparamadu, CTO; Mr. Harsha Sanjeewa , Senior Management Team, my architect, my project manager and colleagues for the endless support and words of encouragements throughout.

My heartfelt gratitude to my friends especially to Mr. Ryan Benjamin, Miss. Tirsha Melani, Ms. Nithila Shanmuganandan, and all other friends which was not mentioned here whose friendship, hospitality and support in the preparation and completion of this study.

Finally, I would like to extend my deepest gratitude to my parents for their never ending support and encouragement.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

LIST OF APPENDIX

LIST OF ABBREVIATIONS

Abbreviation	Description
API	Application programming Interface
TPS	Transactions per Second
CPU	Central Processing Unit
IO	Input/Output
GC	Garbage Collection
JVM	Java Virtual Machine
JMC	Java Mission Control
SEDA	Stage Event Driven Architecture
GUI	Graphical User Interface
PoC	Proof of Concept
DI	Dependency Injection
UI	User Interface
MO	Mobile Originated
AT	Application Terminated
DB	Database
async.	Asynchronous
JMX	Java Management Extensions
HTTP	Hypertext Transfer Protocol
JSON	Javascript Object Notation
USSD	Unstructured Supplementary Service Data
NDC	Nested Diagnostic Context
DB	Gigabyte
CEO	Chief Executive Officer

Chapter 1

INTRODUCTION

The term *internet-based service* was become popular with the wide adaptation of 'web2'. Initially those services initiated as small request serving applications and eventually these services started to take over most of the public use cases. The latency of an internet-based service is a key factor of the Quality of Service. The high latency creates a negative impression about the service; therefore, the authors of the system should provide the maximum contribution to reduce the latency of an internet based system. The *throughput* of a system also affects upon QoS. If the system is unable to provide the required throughput, it indirectly tends to either drop the requests or increase the latency of the requests. This effect will reduce the QoS of the system. Furthermore, if the throughput capability is low, the service providers forced to deploy more replications of the same system eventually increasing the operational cost. Therefore, the *latency* and the *throughput* can be considered as the key factors, which contributes to the performance requirement of the system. More about the throughput and latency are discussed in section 1.1.

Usually in the development process, developers tend to give the high priority for the correctness of the functionality, but forgets about the high performance requirement. This decision is partially correct as, according to the Pareto principle (80/20 rule), 20% of the code will be the most used functionality and investing on optimizing the other lesser-used parts may be an unnecessary cost. Naturally, the performance issues will be visible in the performance tests, which are conducted with the nearly developed systems. At that time, the cost of a change may be high. Therefore, it is vital to find the probable performance impacts early as possible. More about performance optimization process will be discussed in section 1.2.

1.1 **Instrumentation Properties**.

In the domain of high performance data streaming, the term *performance* refers to the ability of handling a high throughput with a lower latency. The latency and throughput can be described as follows.

Latency

In general, the term *latency* is referred to the delay between the requirement occurred and catered. In internet-based applications the latency is measured between the request and response. This includes request propagation delays, request waiting delay and service time.

This research mainly focused on the request waiting delay and service time. The delays outside the server are not considered within this research.

Throughput

The term *throughput* is referred to the number of requests served by a system for a defined time period. The internet-based applications are expected to serve a high number of clients at the same time. This results a higher throughput for a particular internet based service. The low throughput capabilities of an instance force the high number of replications and hence a high maintenance cost. Therefore, the increase of the throughput capability is a vital part of the performance optimization of a system.

There are several factors which impacts upon the performance of a java based high performance data streaming system.

1. Concurrency and Parallelism: At a given time, the number of requests which can be served at the system could be considered as the concurrency level of the system. This differs from the parallelism, as in parallelism, it refers the number of requests which executes at the same instant. The concurrency is highly depending on the thread count and the sum of queue sizes, and the parallelism depends on the number of processors.
2. Number of queues: The number of queues increases the throughput. Yet, since the requests are stacked in the queues, the response time of a request increases with the number of queue increases. This results the latency increase.
3. Number of threads and task affinity: While the throughput is increased with the number of threads, it results more context switching. This can be an overhead in terms of the CPU utilization. Furthermore, based on the given task, the thread can act as a CPU bound thread or an IO-bound thread. In CPU thread, the thread utilizes the CPU for computations, and in IO thread, the thread waits for an IO Operations to be completed. The effective maximum number of CPU threads is bounded by the number of processors. In the thread-per-request model, since the tasks are not specifically bounded to the threads, the same thread is forced to act as both the CPU and IO-bound. But in the stage based systems, there's a possibility of designing the systems with CPU bound stages and IO-bound stages separately. This allows the micro level control of the number of CPU bound threads. This is helpful on fine-tuning of CPU bound threads to reduce the context switches. Furthermore, this improves the thread affinity.
4. Synchronous and Asynchronous behavior: The synchronous and asynchronous behavior described in this section strictly relates to the behavior in the external

connectivity. Usually the synchronous external connectors tend to block the threads resulting thread scarcity. In some cases, if those connections aren't properly managed, this may result catastrophic failures of the system due to external delays. Therefore, it's always recommended to use asynchronous remote calls whereas possible. In terms of performance synchronous calls may result a low latency yet it limits the throughput due to the blocking threads.

1.1.1 Performance Optimization Process

As mentioned above, a nearly completed system is used in performance testing. In this stage the performance impacting, unnecessary features like debug and trace logs are disabled. Usually a dedicated setup is used for the performance testing and a separate firing tool is used to fire a controlled burst of requests to test the performance. The system is said to be under-performing if it shows signs of instability before the required throughput is reached. Latency of the system is evaluated separately by analyzing the logs. Irregular bursting sequences in the simulator side due to backpressure, increased GC events and increased number of errors are considered as the failures of the system. In a case where the system is identified as unstable, the troubleshooting process is conducted. Identifying the issues like leaks and bottlenecks is focused in this stage. The troubleshooting is discussed more in next subsection. After the bottleneck is identified, the necessary changes are done to the related component to gain the required performance. This requires another development/functional testing cycle and if the functional testing is successful, the system is re-deployed and the performance tests are conducted. This cycle will continue until either the system gains the required performance or the economical limit of the optimization process is reached the limit. If the performance is still not up to the expected level, replications are made in the development to cater the proper throughput.

1.1.2 Cost of finding the bottlenecks

In the development process, performance issues emerge during the performance test stage. In this stage, the components are nearly completed. Usually, the performance impacting features like debug logs are disabled in this stage, restricting a low visibility of application insight. If the performance test fails with system instability, this low visibility reduces the ability to point-out the root cause of the performance problem. In general, following indirect methods are used to troubleshoot a performance problem in java applications.

GC log analysis.

Most of the time, the system instability is caused by various kinds of leaks. Leaks tend to uncover over the time and if it is a memory leak, the JVM heap will be filled eventually. This will result a low memory state and the JVM is forced to run the garbage collection frequently to free up the memory. Therefore, increasing frequency of the GC events may indicate a memory leak.

Bottleneck analysis with logs.

As mentioned above, the trivial logs like debug and trace logs are disabled in the performance tests. However, the info level logs are kept enabled as a practice. If the time is also logged in the info logs, those logs can be analyzed to identify the sections with the highest latencies. Usually the components with increasing latencies tends to contain the bottlenecks. This analysis may provide some kind of indication to the bottlenecks. However, the effectiveness of this process completely depends on the early decisions of the logs of application.

Instrumentation and Profiling.

In order to make the JVM transparent, performance monitoring and instrumentation tools like JMC[1] (Java Mission Control) and jMeter[2] are introduced. These tools connect to the JVM in runtime and records the JVM statistics. The recorded row data contains the information such as object counts, method access, garbage collection patterns, thread dumps and many more other related stats. The above-mentioned tools provide a great insight about the JVM.

While these tools provide a good support on troubleshooting, the collected information are too generic and does not provide focused view on the bottlenecks to be improved. The developer need to be experienced enough and talented enough to identify the issues correctly and propose the solutions. If the diagnostics are incorrect, the developer may waste time on changing an unrelated area rather than fixing the real issue. Therefore, a mechanism to tryout and identify bottlenecks is invaluable. The common practice is to lay out the probes during the development and collect information in runtime. This method provides the visibility of the system for some extend, yet it also suffers from following drawbacks.

1. Implementing a probing mechanism is relatively complicated. The simplest probing mechanism would be logs.
2. The probing mechanism is not a functional requirement and tends to complicate the codebase. Therefore, developers reluctant to implement this kind of feature.

3. Additional probing mechanism in the hot zones may decrease the performance. Therefore, in most cases, implementing probing mechanism is considered as an unnecessary burden.
4. If the performance requirement is not critical, the application may run with the correct performance. In this case, adding probes may actually be unnecessary.

Due to those reasons, usually adding probes is not practiced during the development time. However, this aspect can be considered as a cross-cutting concern, and with a correct design this aspect can be abstracted out to a different framework. Extracting to a different framework is economical as it reuses the implementation to cate this specific aspect.

1.2 Data-flow architectures

Most of the high performance streaming applications have a clearly defined role, which can be catered with the event processing. Therefore, those applications usually model with data-flow network based streaming architectures[3] combining with client-server architectures.

In those data flow network architectures, a network of components is created with each component playing a specific role. There are so many researches following this architectural pattern. SEDA[4] architecture and Actor Systems[5] can be taken as examples for data-flow architecture pattern. More about data-flow based architectures and its implementations discussed within the section 2.2.

There are couple of properties can be identified from this data-flow architecture patterns.

- The flows are designed one-way and usually asynchronous. This property is exploited in the SEDA architecture to build up the stages.
- The network components are modularized and contains clearly defined entry and exit points. Usually the exit point would be the call to next component.

The first property clearly defines the data flows, which acts as the hot zones in runtime. And the second property provides the entry and exit points of the stages in this hot zones. Therefore, data-flow architecture provides a good basis on identifying the possible probe points. As a general rule, the probe points should be placed on the entry points of each flow component and in the exit point of the flow.

1.3 Dynamic Concurrency

In a java-based system, usually, the thread pools are coupled with queues to hold the requests until picked. Therefore, any bottleneck would result an increase of both queue size and busy

threads of the serving thread pool. This behavior can be exploited in identifying the bottlenecks, if the thread pool related data are available in the analysis. During the analysis, the active thread counts and queue sizes should be analyzed from the beginning of the flow and the last thread pool to indicate the abnormal behavior will contain the bottleneck. This exploitation can be used within the proposed framework by implementing a mechanism to adjust the thread model dynamically.

1.4 Type Safety and developer friendliness.

Type safety is a key feature of java programming language. Therefore, any framework written with java should provide the type-safety. Usually types are used to define the shapes of the object in a java program. In this research, those shapes are defined according to the runtime behavior of the flow computations. More about the types defined in this research is discussed in the section 3.

Usage of a framework usually increases with the developer friendliness. The framework should provide an API which is familiar to the developer community to reduce the metal mapping. One of the commonly used flow composing pattern is 'Function chaining'. In Function chaining pattern, a builder consists of higher order functions is used to compose the flow. Java stream API[6], Scala collection API[7] and RxJava[8] can be considered as the examples for the usage of function chaining patterns. This pattern loosely resembles the builder pattern[9]. For this research implementation, function chaining is used to build the flows. More about this is discussed in section 4.

1.5 Problem Statement

Performance optimization is a vital part of developing the high performance data streaming applications such as internet-based streaming applications, data processing applications, IOT related data processing applications, etc. These optimizations will improve the stability in a system and reduces the cost in long run. The conventional instrumentation tools are too generic in identifying the system bottlenecks. It is convenient to have a framework, which provides a good support for instrumentation while enforcing the developers to use stable mechanism when developing a high performance streaming system.

Additionally, in almost most all the applications, concurrency architecture is hardcoded to the application. Within the runtime, the concurrency model is always static. This rigidness can be changed by inventing ability to dynamically change the concurrency architecture using thread

pool placeholders. The runtime flexibility of the concurrency model can be used to identifying the bottlenecks easily. Furthermore, eventually this innovation open ups the 'thread pool reconfiguration' as another degree of performance tuning in current software development process.

1.6 Motivation

The performance of high performance data streaming applications is a high priority. But in current development methodologies, this aspect has been pushed back to the end of development process. Investing additional effort to develop an instrumentation mechanism, which might not be used at all, seems unnecessary. Yet this feature is extremely useful for high performance applications as it provides the proper insight of the application in the optimization phase. The norm of *not having instrumentation embedded* can be changed by introducing a framework, which provides the instrumentation alongside other benefits. Data-flow based architectures are a good candidate to introduce this feature due to following reasons. Data-flow based architecture is commonly used in high performance streaming applications; Data-flow architectures provide a structured basis on identifying the hot zones and possible probe points. 'Dynamic Concurrency' or 'dynamically assigned thread pools' is another concept which would mingle well with the performance troubleshooting. While high latency of a component hints a possible bottleneck, the effect on the thread pools from a bottleneck is more direct. In the high load situation, the bottleneck tends to make a backpressure on the thread pools resulting queue full situations and threads contention. Therefore, if the thread pool location can be changed, it will provide a great advantage on identifying and isolating the bottleneck.

1.7 Objectives

The main objective of the research is to provide the framework 'concor', a comprehensive java framework, which caters in-built instrumentation with dynamic concurrency. The concept behind this research; 'instrumentation in-built framework for high performance streaming applications', is a generic concern and should be adapted to the other programming languages such as *.Net* and *python* as well as the other frameworks like RxJava[8] and Spring-Webflux[10] in a different research. Following areas are covered within the research.

- Provide a type-safe and comprehensive API to develop flow components and assemble the flows. The API is built with the types based on the component runtime behavior analysis.
- Provide a mechanism to measure the performance stats on each component with the minimum performance cost. A sampling mechanism is used to reduce the runtime overhead, and the sampling can be switched off in production.
- Provide a UI tool, which can be used to visualize the flows and observe real time stats of each flow computation.
- Provide a mechanism to reconfigure the concurrency model by switching the thread pools within the flow. This ability provides a great flexibility on both bottleneck identification and runtime performance tuning.

1.8 Scope

As this research is mainly focusing on providing a proper mechanism to instrument and monitor the subjective system in runtime, the PoC implementation does not implement the advanced features such as multiple flow handling. The research will be limited to following scope.

- The research and evaluation will be conducted upon java implementations. The highest performing feature libraries will be used in the PoC implementation. Extending this concept to other platforms should be researched separately.
- Vanilla java implementation is used within the research and the extensions alongside other flow composition libraries should be researched separately.
- As the PoC, a single flow will be evaluated for each application. However, by changing the user interface, the ability to show multiple flows can be achieved.
- This Research is mainly focusing on implementing and evaluating data flows rather than data flow networks. The research on extending the concept to data flow networks should be conducted separately.

1.9 Structure of the thesis

The rest of the thesis is structured as follows. The chapter 2 is dedicated to the literature survey of the research. Chapter 3 discusses about the methodology and Chapter 4 discusses about the proof of concept implementation information. The evaluation of the implementation is discussed in the Chapter 5 and the conclusion is discussed in Chapter 6. The Appendixes contains the additional information about the PoC implementation.

Chapter 2
LITERATURE REVIEW

The literature review of this research is mainly two-folds as the study about instrumentation and the study about data flow architecture patterns.

2.1 Instrumentation and Profiling.

Instrumentation and profiling is a vital part of application performance optimization. Throughout the history there are several researches have been conducted upon the application instrumentation.

JSR 163: Java Platform Profiling Architecture[11] describes about the initiative of introducing the profiling APIs to the JVM. Almost all the tools related to the JVM profiling are using these APIs. However, current research is focusing on providing an architectural specific solution and therefore within this research these APIs will not be used. By moving this solution out of the JVM internals, the generalization of this mechanism in terms of the programming language/platform is achieved.

Other notable researches related to instrumentation are listed below.

The research in reference [12] is mainly focusing on developing a methodology which is capable of collecting individual component performance like method call start and end times and stack trace via manipulating the byte code. This research provides the capability to instrument arbitrary components in runtime and collect related performance data. While this method is powerful in instrumenting specific parts in the application, analyzing and comparing a flow for performance bottleneck will include guessing and checking. Furthermore, this is merely an instrumentation tool and does not provide runtime optimization capabilities. This is acceptable, as the specific research does not cover the runtime optimizations under its scope.

Reducing the performance footprint can be considered as an important aspect alongside the instrumentation framework development. The research in reference [13] is mainly focusing on reducing the performance overhead of collecting data only with the selected components. This is a good tactic to reduce the load while getting the valuable instrumentation data out of the system. Within this project, the JVM is modified ad a GUI tool is given for monitoring and configuration. As mentioned earlier research, this research does not provide the capability of directly finding bottlenecks of a system.

The research in [14] is a similar research, which uses on-the-fly bytecode instrumentation. The focus of this research is to provide a good framework with contracts, which enables user writing style and correctness improvements. Instrumentation is a part of this research and it does not focus on the performance optimization as a main objective.

There are two main approaches are commonly used in profiling an application as instrumentation and sampling. "A Portable Sampling-Based Profiler for Java Virtual Machines"[15] research provides a good insight about the both profiling mechanisms. Furthermore, this research develops a low overhead sampling based profiling technique. Within our research, we use the sampling as the primary data collection mechanism. While this research is again focusing on wide range of applications and does not focus on the features like topology based analysis.

There is another research [16] which is similar to 'concor' framework. This research provides the ability to visualize the insight of an application via extracting topology and sequence diagrams. It provides a good basis on instrumentation also. This research differs from the 'concor' framework in two ways. First, this research uses annotation mechanism to separate external aspects from core logic. The annotation mechanism provides the flexibility to the application itself, but may break the continuity of the monitored output. Furthermore, this may break with the conditions and loops. Secondly, 'concor' framework provides the dynamic concurrency abilities to the framework, which simplifies the bottleneck identification greatly. This would not be able to achieve with the flexibility provided by the 'KEIKER'.

Altogether, there are two main approaches used in instrumenting a java software. First, the bytecode is manipulated to inject instrumentation probes. This approach is too generic and may not provide the overall picture of the applications. Yet this method is widely used, as it is easy to adapt to existing applications. The second approach is to build the application considering instrumentation as a first class citizen. The research around this approach is rare due to the inability to adapt to existing applications and the rigidness provided by this mechanism itself. However, the authors decided to use this approach, as the rigidness is a necessary feature to use alongside the dynamic concurrency. Most of those researches tends to evaluate the effectiveness of the instrumentation through the overhead provided by the framework. The 'concor' framework effectiveness also measured using a similar approach. However, in 'concor' the authors decided to use the application performance breaking points instead of the

indirect measurements like CPU or Memory, as it is directly related to the practical application in terms of benchmarking.

2.2 Data-flow architectures

2.2.1 SEDA[4] architecture

As already discussed, the initial families of the concurrent server side applications were consist of thread-pre-request model, which seems simpler and reliable at the time. This model has been changed drastically with the famous SEDA architecture. SEDA architecture was initially proposed by Matt Welsh in 2000 as an alternative way of improving the performance in high performance internet-based applications. The idea was adapted for several other domains like distributed functions and SOA. *Figure 2.1* shows a simplified view of this architecture.

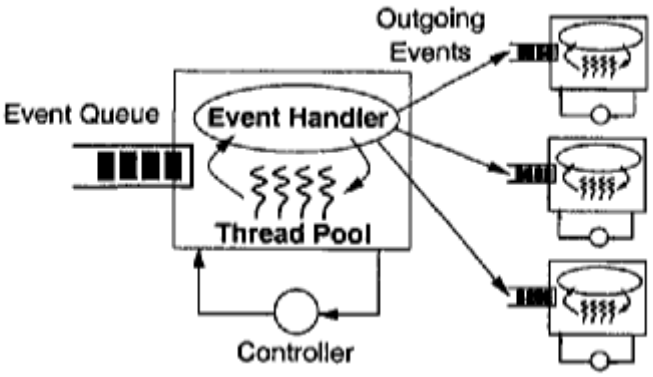

Figure 2.1: SEDA stage: source: SEDA: An Architecture for Well-Conditioned, Scalable Internet Services.

The core idea of SEDA architecture can be summarized as follows. Application developers need to develop well-defined stages of the application flow. Those stages are bounded by a queue and consists of an event handler, a thread pool and a controller to control the scheduling requirements for the specific stage. The application data flows are created by arranging the stages in a sequential manner.

This architecture enables the application to achieve a higher concurrency with low number of threads, which eventually increases the throughput. SEDA architecture was massive step forward in high performance application development at the time and was referred in

developing one of the core java functions; non-blocking IO[17][18]. ‘Concor’ framework architecture is mainly based on the SEDA architecture.

There are couple of enhancement attempts to improve the concurrency and dynamic behavior of the SEDA architecture. One of the approaches was by having individual feedback loops to respond the throughput and queue size. “Adaptive Overload Control for Busy Internet Servers”[19] can be considered as one of those researches. This research is focused on optimizing the individual thread pool based on feedback on data flow. The optimizations proposed are localized to the individual stages.

SEDA architecture and above-mentioned optimizations addresses the concurrency and throughput problem in micro-level angle. Any stage, which creates a bottleneck, may generates a high latency in high load situations. This will eventually propagate to the start of the flow due to backpressure and reduces the application performance. Micro level feedback loops alone may looks as if it can handle this problem. But without having a global view it is really hard to get the most optimized solution. In order to address this problem, researches has been conducted with the focus of global view of the system.

The lack of global load distribution view of flows is the main reason to have such unbalanced stages. The authors found the research in [20] is a valuable research on solving this specific issue. The solution provided by this research tries to separate integration from business logic. In the research, main functional stages are written separately and the Clojure[21] language is used to integrate the stages. Clojure is a functional LISP running on top of JVM, which has the full interoperability with java. Therefore, this allows the stages to be written in java and integrate via Clojure language. Being a LISP, Clojure language inherently provides the tools to abstract out and visualize the integration architecture. While, this approach is a good combination of using two languages to solve the global concurrency and view problem, it suffers from the high language knowledge overhead. To use this system, the developers are expected to have the knowledge on Clojure and the additional knowledge introduces an additional cost for the projects. This may prevent wide industrial adaptation of this approach.

The same problem has been addressed differently in the research mentioned in [22]. This research introduces a language called FLIMP. In FLIMP, the program is expected to be written as a single threaded sequential program. Then the optimization compiler extracts the code computation chunks to form highly optimized concurrent stages. Having this new language, the platform provides the development simplicity.

The FLIMP language provided by the research is mainly focusing on the code analysis and parallelism. Therefore, the language should be developed further to cater other general purpose language features. Otherwise, the language itself is not usable in the industry. Furthermore, It does not talk about the integrity with other platforms like JVM. While the research provides one of the theoretically correct ways of solving the problem, the industrial adaptation of this research is very limited. However, the extensive research done on component in terms of the latency and caching is impressive and can be adapted for other similar researches.

2.2.2 Actor systems

Another alternative idea of modeling the concurrency in a stream application is having an actor system[23]. In an actor system, an actor hierarchy is created and those actors are assigned with specific tasks. The only communication mechanism between actors is message passing and this will effectively isolate the individual tasks. In the industry, actor systems are used in basically two different styles. As the name implies, it can be used in database like highly concurrent systems where each actor serves a particular client connection. Otherwise, the actor systems can be used to form an organization like models. In this model, a strict hierarchy is formed and tasks and management is handled by the workers and manager actors. However, in most cases, the reference to the next actor is kept within the actors, forming a sequential data flow like a factory pipeline. Actor systems promotes single threaded like behaviors. Wherever the concurrent behavior is needed, a manager actor is kept to accept the requests. Manager actors distribute the task among worker actors, which effectively emulates the concurrency.

The actor systems uses event loop based execution model[24]. In this model, one or more thread pools are used and each actor is scheduled to the thread pool alongside the actor state as the context. This allows creation of thousands of lightweight actors for a particular application using limited number of threads. While actor provides benefits like modeling simplicity and lightweight concurrency, usually actor systems on JVM like platforms suffers from lack of thread affinity.

An actor or a hierarchy focusing on a specific task can be considered as a stage in stage based architectures, except for the scheduling. Since actor systems consists of strictly defined boundaries between tasks, it is hard to implement fusion kind of operations on top of actor systems. Yet in stage-based systems, this is a possibility.

The research mentioned in [5] extensively talks about the idea behind the Scala implementation of actor system on top of JVM threads. This is a useful resource on understanding the theory behind actor systems. However, the actor system implementation in discussion was replaced with a new actor system called akka library[25].

2.2.3 Architecture comparison.

Deciding the correct architecture to be used in the 'concor' framework building is a crucial decision and in order to collect the information on the subject, the literature survey is extended to searching for architecture patterns comparision. The research mentioned in [26] is a research which gives an effort to evaluate and compare the above mentioned architectural styles. The researchers have used three different types of servers. µserver for event-driven approach, Knot for thread-per-connection and WatPipe for hybrid of events and threads. The results show that the µserver and Knot both performs well while WatPipe performance is degraded with regard to the response time. However, µserver shared non-blocking implementation and WatPipe performs well with regard to the throughput. This behavior is reasonable as the event driven architectures prone to keep requests within queues. However, this research is only focusing on the performance aspects like throughput and latency.

2.2.4 Stage analysis

Both the SEDA architecture and actor systems uses somewhat similar concept to the stages. Since in most cases the stages contains more than one functionality, it can be considered as a collection of small computation units. Those computation units contains different behaviors.

The research mentioned in [27] is a valuable resource on understanding the nature of the minimum atomic computations and the operations could be done on top of those components. It summarizes the other literature related to stream processing such as SEDA, and provides a good abstraction over the stream based application optimizations. The atomic computation units discussed in this research is similar to the operator in the mentioned research. The research is focusing on the operator interconnections arithmetic and optimizations. The target optimizations can be listed as follows.

- **Operator reordering** is changing the order of the operators and is not expected to be included in the proposed solution as it messes up with the architecture written by the application authors.

- **Redundancy Elimination** may not be supported by the framework design as a dynamic feature. The possible extension on top of the framework related to this optimization would be analyzed the data flow and propose the optimization as a suggestion.
- **Operator Separation** is expected as an input from the application authors by accepting the atomic computation units. The application authors are responsible for providing the atomic computation units, because the unit formation is highly application oriented and it requires extensive amount of analysis to dynamically separate the defined atomic computation units. However, this feature is emulated by redefining the virtual stage boundaries.
- **Fusion** is the ability to combine sequential operators to form combined operators. This is one of the key aspects of this research and this concept is used extensively to form virtual stages.
- **Fission** is the ability to run the same computation in parallel and in current contexts, it is achieved by the thread pools. This research is also expected to be utilize the same technique to achieve the fission ability. Instead of the static implementations, the ability to dynamically changing the degree of parallelism is achieved by gaining the ability to reconfigure thread pools.
- **Placement** is the ability to combine the operators in a single environment. Placement and fusion does not make any difference in this context.
- **Load Balancing** is the ability to balance the work among the parallel stages related to the current domain. This is implicitly provided as a particular thread executes a single computation at a time, and once the task is completed, the thread moves to get the next task.
- **State sharing** is the ability to share the state among the operators. This is a mandatory feature in most of the applications in current domain, and brings the complications on managing the whole structure of the application. An extensive research is to be done on handling this feature. For this research scope, the application authors are expected to bind the required information at once and use context to share the information.
- **Batching** is the ability to buffer the stream and process at once. This feature is may not be provided in the initial phase and may be implemented as an attribute of a computation. This to be served, the complete virtual stage should be capable of handling the batch of events. Using GPGPU for parallel computations can be considered as a good example of this nature.

- **Algorithm selection** is not a feature of the framework itself and should be implemented by the application authors if necessary.
- **Load Shedding** is the ability to drop events in emergencies. This is not a responsibility of the framework itself and if needed, should be provided by the application authors. However, the tools to identify current state is provided within the framework itself.

Within this research, Fission and Fusion can be emulated with the dynamic concurrency model.

"Pipelined fission for stream programs with dynamic selectivity and partitioned state"[28] research is one of the attempts, which tries to resolve the stage throughput problem by analyzing latencies and model the program into cost optimization problem. The problem is solved using a heuristic based approach to gain a reasonably correct solution within the time limit. This research is a good resource on understanding the computation modeling and may be used in solving the best stage configuration in current research. While this research provides the proper abstraction on selecting the concurrency model, it does not capture the individual computation behaviors such as critical sections or synchronous IO calls. Therefore, the methodology provided by the system should be modified according to the behaviors.

2.2.5 Queue implementations**.**

In almost all the stage-based architectures, queues are used to communicate between the stages. These queues may also refer as mailboxes in actor systems. The efficiency of the queue is a vital but unnoticed factor in most of the researches. Lmax organization has proposed a high performance queue implementation called Disruptor[29], which is the efficient and lowest overhead queue implementation at the time of this research.

Within disruptor implementation, it uses a ring buffer internally. There are couple of advanced optimizations implemented to provide the faster operations as listed below.

- It uses cached buckets to reduce object creation overhead in object passing through queue.
- The number of keys per ring buffer is advised to be a power of 2 (2^x) so that the read write head increment to be faster.
- It uses padding to the long keys so that the cache lines containing keys does not swing between the processors emulating ping-pong effect (Reduce false sharing effect).

This disruptor model is planned to be used in this research to gain the performance out of the application itself.

Chapter 3

METHODOLOGY

During our research, the author introduces an alternative approach of instrumentation in streaming applications with the collaboration of SEDA architecture. The instrumentation of each micro-component is achieved by providing constructs, which accepts a computation and generates a wrapper with the instrumentation abilities. Furthermore, this research will introduce dynamic staging and thread configuration. The dynamic staging capability provides a great help on isolating the bottlenecks.

Apart from this use case, this will provide following additional advantages.

- The developer can get a proper insight of the application architecture graphically.
- The threads and stages can be configured in the runtime. This will open up another degree of runtime optimizations.
- The framework provides a type-safe and structured mechanism for developers to develop individual components.
- The framework will effectively lift off the burden of concurrency design.

In order to achieve this requirement we propose following architecture.

3.1 Framework

The framework can be considered as the core of the system. It is supposed to accept the atomic computation units as the slots create flows using the computations and build the full fetched application with complete concurrency control. A high-level view of the architecture can be seen in *Figure 3.1*. The components of the framework can be described in sections 3.2 to 3.5.

3.2 Dispatcher

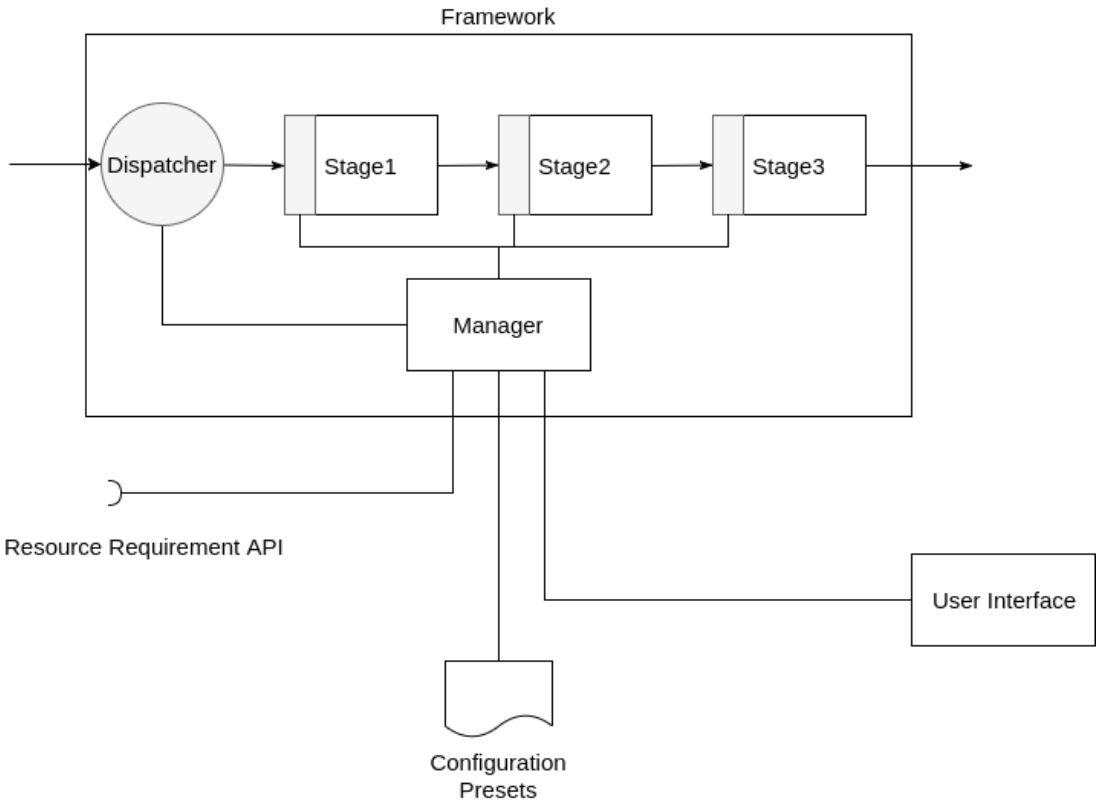

Figure 3.1: High-level architecture of 'concor' framework.

The dispatcher starts a message flow in the system. There can be more than one flows in the system, but multiple flows are not supposed to share the common computation units for the simplicity and correctness. In order to have branching, new flows with a dispatcher should be used.

In the operation, the dispatcher is supposed to accept events, wrap the events with the so-called contexts, and dispatch the wrapped events to the application flow. The context is useful in collecting the statistical information about the flow.

3.3 Virtual stages

All the atomic computations accepted by the framework will be wrapped with a computation wrapper. Those computation wrappers are capable of accepting incoming context, appended events, unwrap context and perform computation with the event. The computations are typed functions and are chained together to form the flows. Defining the flows is left for the application developer as the flow related information are application oriented. Constructs to bind the computations are provided by the framework. The computation wrapper can be considered as a generic Monad. A simplified view of a computation unit can be seen in *Figure 3.2*.

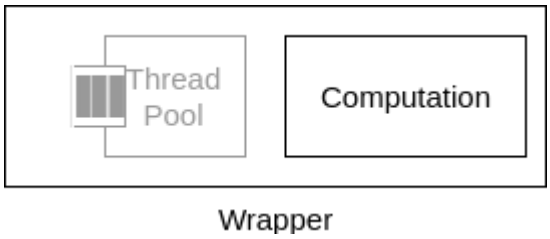

Figure 3.2: Computation Unit

Other than the above capabilities, the wrapper is capable of optionally having a thread pool, which denotes the start of a stage.

The virtual stages are formed dynamically by adding a thread pool configuration to a selected component wrapper. The considering stage is defined from the stage having the thread pool until the next stage starts or until the end of flow. Usually the dispatcher is supposed to start a thread pool and can be considered as a special type of starting component wrapper. *Figure 3.3* shows the formation of a virtual stage.

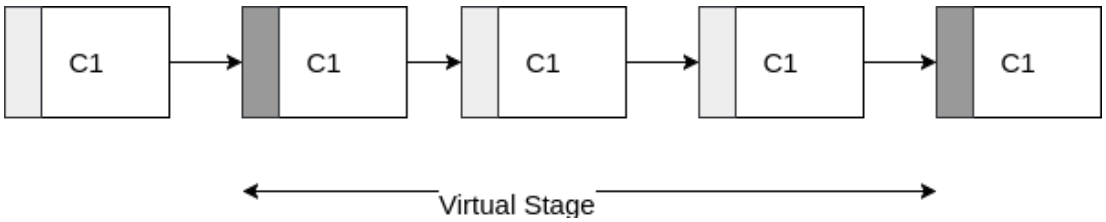

Figure 3.3: Virtual Stage

This design enables the ability to dynamically create, remove and reconfigure the stages by redefining the computation wrappers. Furthermore, this will allow the multi-dimensional (thread count vs stage size) control over the virtual stages to optimize on the latency and performance aspects. The exception handling is supposed to be conducted via a flow based

exception handler. The user should provide a special exception handling function. Internally, the context would act as an either monad, which carries the success response or an error at a time.

In terms of performance instrumentation, the virtual stages can be created in the runtime and isolate the stages, which has the most performance, degrade. Then the stages can be adjusted iterative until the specific low performance component is identified. It is repeatedly proven that, in a high load scenario, the last stage that has the highest number of active threads being the performance bottleneck.

3.4 Manager

The manager is supposed to control all the aspects of the concurrency control instrumentation and communication.

The manager communicates with all the dispatchers and all the wrappers to collect data about the current state of application. These performance stats will be transferred to the monitoring tool UI to be visualized.

3.5 User Interface.

User interface is a major part in the framework as it visualizes the topology of the system, current stats and concurrency configurations. Furthermore, it should provide the ability to change concurrency configurations on the fly. This UI is supposed to connect to the server, via a HTTP API to conduct its operations. For this PoC, advanced features such as dynamically connecting to a server is not be implemented.

3.6 Operation

In the boot time, the framework accepts the message flows and forms the comprehensive wrapper based flows including the dispatcher and wrapper chain. Then, if the system is in production and a preset is available, the system loads the preset and reconfigure the flows according to the preset. Otherwise, rule based dynamically reconfiguring mode is used in operation. These presets are used to save the application thread configuration.

During the operation, the events are pushed to the dispatchers. Dispatcher picks context objects and sends the wrapped event and context through the flow. In order to reduce the object creation overhead, a context pool is used to hold predefined contexts. This mechanism is useful in the

sampling aspect as well. The context is supposed to collect the performance stats like latency and queue sizes. Couple of predefined context objects will be enhanced with the stat collection capability and while the requests with specific context passed through, the stats are recorded in the system. This information will be used as the input for configurations changes.

In order to reduce the wastage, the sampling mechanism can be suspended in the production environment. This will reduce the performance overhead significantly. The sampling can be enabled in the runtime on demand.

Furthermore, this framework should be capable of validating the flows based on computation types. There are four major computation types identified so far.

- Critical sections: Critical sections are supposed to be run in a single thread due to the resource sharing nature of the computation. Session manager can be considered as an example of this kind.
- Synchronous IO blocking sections: In synchronous IO blocking sections, multiple threads are used to send the request, park and receive response. This type of computations consume more threads and a higher number of worker threads should be assigned to this kind of computations. Usually, in these sections, the performance is limited by the number of threads.
- Asynchronous IO Sections: In Asynchronous sections, the requests are sent asynchronously and another underlying library defined thread is used to receive the response. In terms of the application flow, this breaks the continuity of internal pipeline chain. Therefore, additional measures should be taken to mitigate this problem.
- Parallelizable computations: This type of computations can be considered as the currency of the system. These computations can act in single threaded or multi-threaded manner. Therefore, this can be used to trade between stages on load balancing.

This typed micro-component templates are important to maintain the structure of the application while preserving the type-safety.

3.7 **Constraints and problematic areas as a framework**.

Following areas are identified as the possible problematic areas of the system implementation.

- The system should be designed in such a way that the management and instrumentation overhead should be minimum.

- Maintaining the framework structure during asynchronous callbacks is problematic as part of the computation is executed in a different thread. A method of continuing the flow in asynchronous callback should be identified.
- Dealing with thread local variables like transaction is problematic as the thread can be changed during the flow. This should be handled by passing the thread local variables when events are passing the stage boundary.
- Still, there is no proper solution for possible reference leaks across stage boundaries. This should be mitigated by the application development practices.

Apart from above mentioned problematic areas, the proposed system is capable of delivering the concurrency automation aspects via 'concor' framework. The 'concor' framework is capable to co-exist with other major frameworks like spring DI[30]. However, integration with the frameworks such as RxJava, which are mainly focusing on flow handling, may be problematic.

3.8 Summery

Within this chapter, the concept behind the research and the PoC is extensively discussed. The possible implementation related conceptual information and operation are also discussed in this chapter. The problematic areas and challenges identified during the designing phase are also mentioned in the end of this chapter. Altogether, the Methodology chapter summarizes the conceptual and designing related information about the research and it builds a proper guide to follow within the research. In the next chapter; chapter 4, the PoC implementation related information are discussed.

Chapter 4

PROOF OF CONCEPT IMPLEMENTATION

This section describes the PoC implementation details of the instrumentation framework ‘concor’, including scope of the implementation, used technologies, test setup architecture and outcomes of the framework.

4.1 Scope of the implementation.

The complete test setup consists of the framework, UI tool, sample applications and the performance simulator.

- For this PoC java is selected as the main programming language. However, this concept can be extended to other platforms like *.Net* and *python*.
- The framework will focus on collecting data, transferring to the monitoring web UI and handling the reconfigurations. The framework is completely written from scratch and does not use any other stream processing framework like RxJava. Mixing between RxJava and concor framework may result unexpected outputs.
- UI tool consists of a standalone server and web module. It provides the graphical representation of the flows, real-time stats and re-configurability.
- Sample applications are built with a single flow, which consists of multiple steps. Ability to handle multiple flows is strictly an implementation concern and therefore this will not be evaluated in this PoC.
- A performance simulator is developed to send simple messages, controlling the TPS. Initially the simulator is started with a pre-configured TPS and gradually the TPS will increase stepwise.
- Throughput and latency are the main measurements provided by the tool. Other than that, the active thread count and queue sizes are also provided as the measurements.
- The tool provides the configurability of thread pools between the flow components, which can be used to isolate the bottlenecks.
- Although dynamically connecting the UI to a server is a part of the framework, the ability to initiate the connection on the fly is not considered in this POC. However, the connectivity can be configured in the UI server property files.

4.2 Framework.

For better adaptability, the framework is written using plain java. All the component constructs are built from scratch. There are five types of component constructs provided by the framework. All of those interfaces are marked as **@FunctionalInterface** and therefore lambdas can be used instead of java classes.

- *Simple task*: The thread safe tasks are considered as simple tasks in this domain. This can be run parallel. A sample of simple task is shown in **APPENDIX A: Sample simple task implementation**.

- *Single threaded task*: The tasks, which consists of synchronous blocks, are considered as the single threaded tasks. The framework makes sure that these components will not run in parallel, unless the framework is misused. A sample single threaded task is shown in **APPENDIX B: Sample single threaded task implementation**.

- *Synchronous remote task*: Synchronous tasks tends to block the threads until the remote call or DB query being complete. Configuring such a component to run in single thread will result in a huge blockage as the threads are parked until the result is received. Therefore, an additional validation is placed in the framework to prevent the synchronous tasks being configured with the single threaded thread pools. **APPENDIX C: Sample Synchronous remote task implementation** shows a sample implementation of a synchronous remote task.

- *Asynchronous remote task*: Asynchronous tasks are there to fix the flow continuation break. Async task provides an additional parameter called ‘continue’ and should be used upon async task completion. This pattern is referred as the ‘continuation passing style’ in functional programming. A sample implementation of Asynchronous remote task can be found in **APPENDIX D: Sample Synchronous remote task implementation**.

- *Catch Task*: All the errors thrown to the framework will be collected within the flow context. To handle the errors, a special component type is provided as catch tasks. The flow works similar to the either monad in functional programming. Upon any exception, the specific request context will keep the result and exception and next steps will be skipped until the catch block is available. Catch block can be used to recover

from the exceptions. **APPENDIX E: Sample Catch task implementation** shows an implementation of a catch task.

- Apart from those components, another mini flow component is added for the completeness called side effect. Side effects will listen to the incoming messages, yet it does not provide any output except for the incoming message modification.

All of those components can be bound together with a flow builder. Flow builder will create a flow and binds to the flow manager for management aspects. **APPENDIX F: Sample Flow composition** shows a sample flow construction.

Requests can be injected to the flow using the 'apply' method in the flow. Then an implicit context is assigned to the request and dispatched to the flow itself. There are two types of contexts as mock contexts and sampling contexts. Sampling contexts tends to collect data while traversing in the flow. In order to reduce the instrumentation overhead, sampling contexts are dispatched in periodic manner.

All the data collected in the sampling context are reorganized and stored in memory for the visualizations. At a sampling burst, there will be three requests and the average values of those requests are taken as the result. Results will be available to query from the monitoring tools via a JMX API[31].

The concurrency reconfiguration aspect is achieved by implementing a slot mechanism in front of each component. A so-called *join* can be configured in these slots. Those joins consist of a disruptor ring buffer, and a thread poll of the choice. This mechanism automatically provides queues to split the stages. Currently, Single threaded thread pools, multi-threaded thread pools and cached thread pools are available in the system.

The data collection, transfer to the monitoring tool and response to the thread configuration changes are available through the manager component. Furthermore, upon starting the manager, it will collect the schema information about the flows to map the topology. This information is sent to the web module upon web module connect.

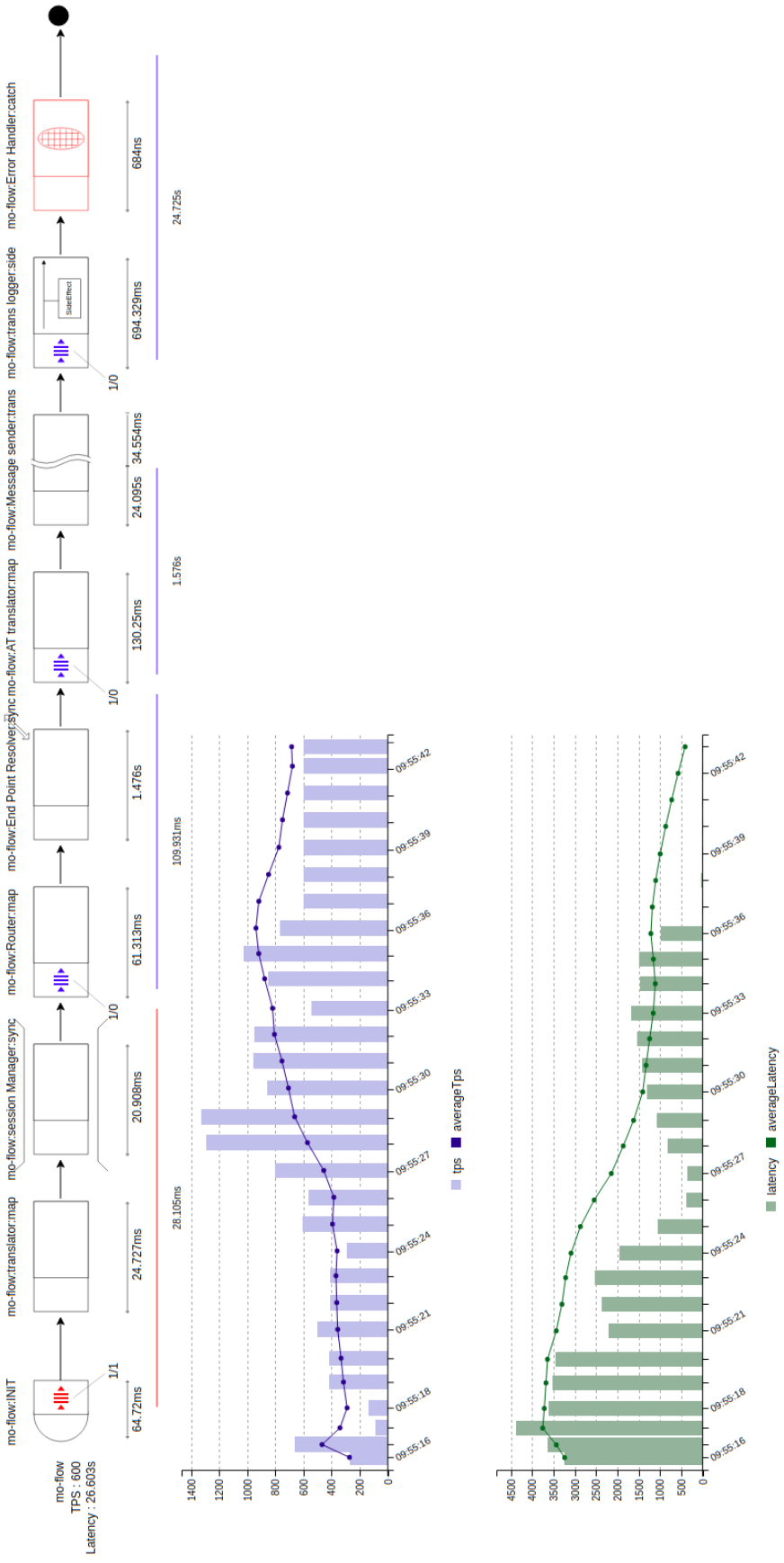

Figure 4.1: A runtime view of a flow in ‘concor’ UI.

4.3 Performance monitoring tool.

Performance monitoring tool is consists of a standalone server and a web module. Standalone server acts as the gateway between UI and the system. The server communicates with the system using *java management extensions* and with the UI using HTTP/JSON API. The server is developed using *spring-webflux*.[10]

The web module is built using ReactJS[32] and has the capability to pull the schema and data and sending the requests to do the runtime changes. A sample flow with the other stats can be seen in the *Figure 4.1*.

In the UI flow view, each flow component has its name and the real time latency. The icon in the component shows the type of the component and the block in left of a component indicates the slot for the join. For the configured joins, the number of active threads and queue size are shown. The simulated stage latencies are shown below of the flow.

The two graphs below the flow indicate the flow specific throughput and latency.

A new thread pool can be configured with the popup shown when clicked upon the slot as in *Figure 4.2*.

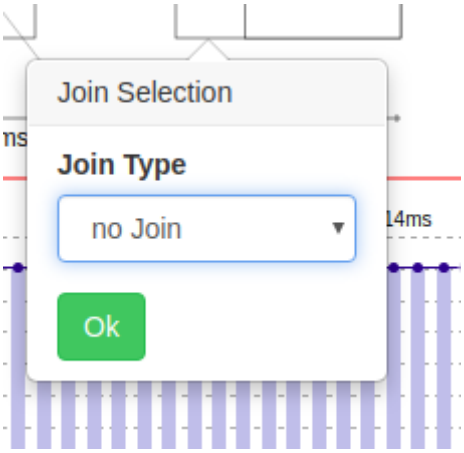

Figure 4.2: Runtime thread model update view.

4.4 Sample application: Messaging MO Flow

The MO-flow sample application is built to visualize an MO flow. The term MO and AT are used in telecommunication messaging domain. In this sample, we consider a messaging flow from Mobile to Application. Following stages are considered in developing the flow. This flow loosely resembles a use case of mobile messaging gateway.

- Mo Translator: Simple task, which converts the incoming message to internal format.
- Session manager : A single threaded task which provides the session resolving capability.
- Mo Router: A Simple task, which provides the route resolving capability.
- Mo Endpoint resolver: A Synchronous task, which provides the load balancing and endpoint resolving capability. This component queries the MySQL[33] database per request basis.
- AT Translator: A simple task, which provides the translation capability from internal format to external format.
- Message Sender: An asynchronous task, which sends the message to the user.
- Trans Logger: MO Trans logger is a side effect, which listen and log the message status.
- Error handler: An error-handling component to log the errors.

This sample app is designed to be a balanced use case to verify the framework capabilities.

4.5 Sample Application 2: USSD server.

USSD[34] server application is built to visualize a static USSD flow. The term USSD (Unconstructed Supplementary Service Data) is strictly used in telecommunication industry. The USSD services are used in providing various supplementary services within the telecommunication network, and currently this has been extended to serve various third party services also as the value added services. The USSD sessions are interactive and real-time. Usually, in terms of the server, this service is modeled as question and answer cycles.

In a USSD serving application, the application accepts USSD requests and returns a new response with the next step instructions. This can be modeled as a single flow. This flow consists of following components.

- USSD-In-Translator: will translates the incoming message to the internal format.

- Session Manager: This element will resolves the session based on the mobile number. Usually this is modeled as a single threaded component.
- Menu Structure Resolver: resolves the menu structure for newly created sessions..
- Menu Resolver: This block is used to select the menus based on the user input. If a menu is not present, a default text will be sent.
- Message Sender: This component is used to send the reply back to the requested party. This is modeled as an asynchronous remote connector.
- Logging and Error Logging: The errors and outgoing messages are logged at the end of the flow.

USSD flow sample application does not have a DB connectivity and therefore this is mainly bound by either the CPU usage or external connectivity.

4.6 Sample application 3: Mobile money tracker

Mobile money tracker is a simple application, which is built to track the transactions and hold the current account values in a DB. Evaluating the 'concor' framework for an IO bound flow is the main objective of this sample. This contains following items.

- Translator: This translator element will transform incoming plain message to the internal format. This is completely a CPU bound process.
- TrxManager's reader: This element is mainly focusing on DB reads. This is an IO bound process and modeled with synchronous remote component.
- TrxCalcluator: The result value calculation is done on this element. This is a CPU bound process.
- TrxManager's updater: This element should update the DB with calculated values. This is an IO bound process with synchronous remote component.
- Logger and Error Handler: This part is mainly written for the logging purposes. This is an IO bound process. Yet this does not give a high overhead in the application compared to the TrxManager.

4.7 Performance Simulator.

The performance simulator is designed in a way that it initially starts with a predefined TPS and increases the TPS in every 10 seconds by 10. The issued throughput is logged in a separate log. This log is used to observe the point of instability.

4.8 **Wiremock**[35] **Instance.**

A Wiremock instance is started to emulate the remote server. This Wiremock server is optimized to cater the best possible performance.

4.9 **Summery**

This chapter discusses about the implementation related details extensively. The implementation scope, the components, which are build and the tools used in evaluating the system are described within this chapter. The collected information within the tests and the evaluation are discussed in the next chapter, Chapter 5: Evaluation.

Chapter 5

EVALUATION

This section discusses the usability, performance impact and the pros and cons of the ‘concor’ 'framework. Then the results of the two experiments conducted upon the framework sample applications are discussed.

5.1 Usability

One of the main objective in this research is to propose an easy to use framework providing the dynamic concurrency capabilities. By means of the usability, the developer community should be able to easily adapt to the framework without additional mental mapping. Therefore, commonly used techniques and patterns are used in the implementation. As the flow composing mechanism, function chaining is used. This is a widely adapted pattern in many other use cases like java stream API, Scala collection API and RxJava. Furthermore, by defining the component types based on the behavior enforces the developer to think in clear functional aspects rather than jumbling up the code to get the functionality done. This cleanness provides a maintainable clear code.

Even though the ‘concor’ framework provides those pros, following features and safety mechanisms are lacking in the framework.

- Reference leak: A developer can misuse this system to leak some references from one stage to another, which might lead to concurrency issues, if the leaked object is not thread safe.
- Single threaded behavior is enforced flow basis: A user can use same single threaded component either in different flows or in different locations in the same flow. This might lead to concurrency problems. This problem can addressed in a future work by enforcing single thread access via a locking mechanism.
- Thread local value passing is problematic: Since the objects are passed within dynamically allocated thread pools, the thread local values like MySQL connections and NDC may not be available throughout the flow. This research should be extended to resolve these issues.

Apart from the developer friendliness, this framework provides a clear view of the system flows, in a graphical manner. For a particular new joiner to the project, this feature is helpful as it provides the accurate insight and topology of the application.

5.2 Performance

5.2.1 Experiment 1: Runtime performance overhead.

The ‘concor’ framework provides the runtime instrumentation capabilities. Usually the additionally injected code for instrumentation results in performance decrease. In order to measure the performance drop, following experiment is conducted.

This research covers the analysis of the types of micro-level components and it concluded that there are four major types as single threaded, multi-threaded CPU bound asynchronous remote connectivity and synchronous remote connectivity. Any application, which uses this architecture, is forced to categorize each component to one of above-mentioned categories. The sample flow to be tested is selected as a fair mix of those components.

In the real world usage, the designer can choose between three options.

1. Application assembles without ‘concor’ framework.
2. Application assembles with ‘concor’ framework yet the sampling disabled.
3. Application assembles with ‘concor’ framework with the runtime sampling.

In order to measure the performance of the system for these options, a controlled experiment is designed as follows.

- The simulator initial TPS is set to an initial value and increase the TPS by 10 for each 10 seconds.
- Sampling overhead is kept as 3 sampling messages per 128 messages.
- For the configuration setup without the framework, exact same thread model like in the diagram is configured.
- Before each run, all the components were restarted to remove the effect of previous run.
- The experiment was conducted on a machine with following statistics.
    - CPU: Intel core i7 6th gen.
    - Memory 16 GB

5.2.1.1 System behavior during performance test

The system behavior during a performance test can be described as follows.

- In the initial stage, the throughput in the firing tool shows some fluctuations due to the empty queue filling and other system processing behaviors. Within couple of seconds, the system stabilizes with the expected throughput. In this state, the throughput is smooth across the time. The throughput and latency effect on the system initiation is shown in *Figure 5.1*.

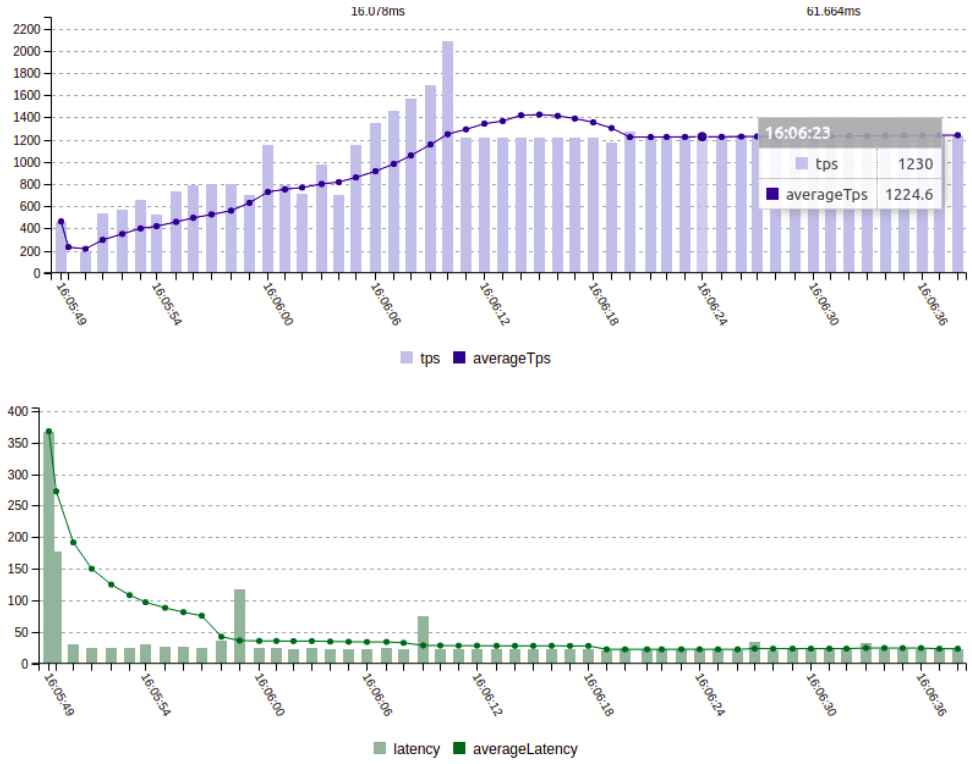

Figure 5.1: Throughput and latency profile in a beginning of a performance test.

- When the throughput increases, the system saturates and starts to show throughput pulses due to the backpressure. In this state also, the system is stable, yet this glitches indicates that the system is reaching instability. *Figure 5.2* shows a view of such a through glitch.

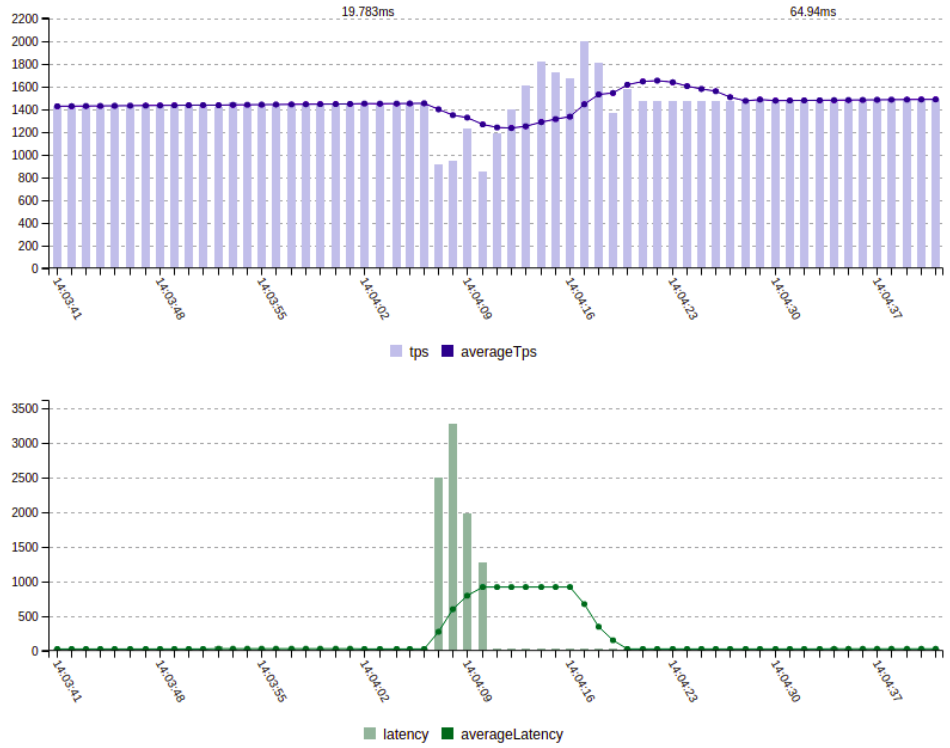

Figure 5.2: Dynamic impulse in throughput and latency profile at a near saturation situation.

- At some point, the system shows a high number of fluctuations in the throughput and system reaches to a point without recovery. At this time, the system is considered as saturated. *Figure 5.3* shows a saturated situation of a failed performance test.

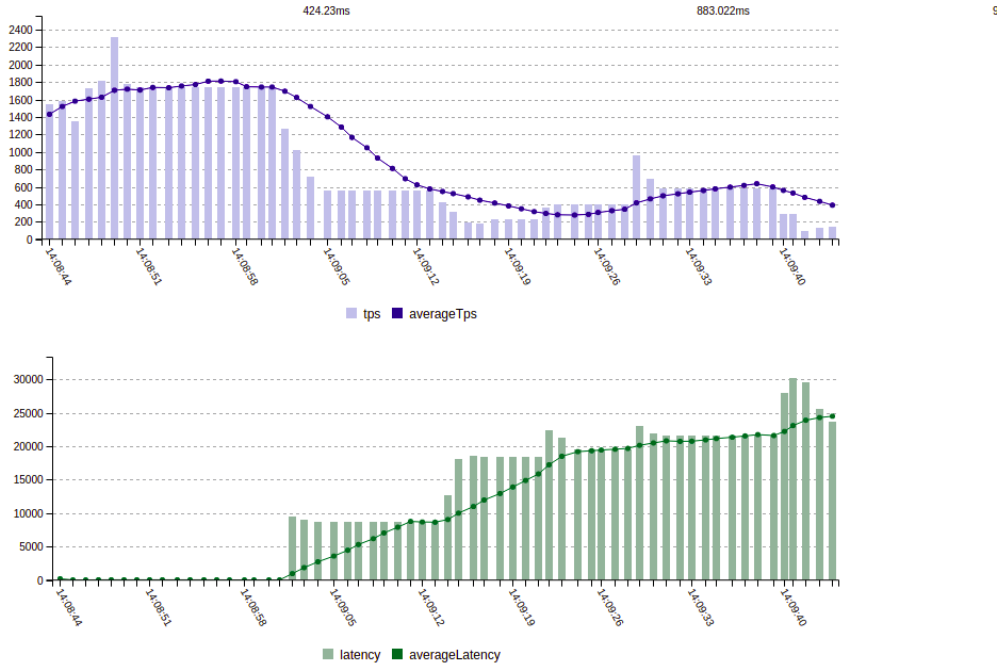

Figure 5.3: Throughput and latency profile of a performance failure.

Within the experiment, if the fluctuations does not stabilize within 15 seconds, it is considered as a failure, and the last stable throughput is considered as the breaking point throughput value.

For each experiment case, nine attempts were tried out as three for each without framework, without sampling and with sampling. The average values ware calculated using *Equation 1*.

$$TPS_{avg} = \frac{\sum TPS_{attempt}}{attempt\ count}$$

Equation 1: Average TPS Calculation

5.2.1.2 Case 1: Starting TPS: 200, with bottleneck in messaging flow.

The first experiment run conducted upon messaging flow and the collected results are shown in *Table 5-1*.

Table 5-1: Results of messaging flow performance test with starting 200 TPS

	Without framework (TPS)	**Without sampling (TPS)**	**With Sampling (TPS)**
Attempt 1	400	390	390
Attempt 2	390	390	380
Attempt 3	390	390	390
Average	393.33	390	386.66

In this experiment, an abnormal blockage was appeared in the 'end point resolver' component. This was visible as high latency in the specific block and abnormally high active threads and queue sizes in catering join.

5.2.1.3 Case 2: Starting TPS: 300, with bottleneck in messaging flow

In order to identify the bottleneck root cause another experiment was conducted with starting point as 300 and increment as 10 messages per 10 seconds. The results yielded form this experiment is listed in *Table 5-2*.

Table 5-2: Results of messaging flow performance test with starting 300 TPS

	Without framework (TPS)	**Without sampling (TPS)**	**With Sampling (TPS)**
Attempt 1	450	450	450
Attempt 2	440	460	460
Attempt 3	450	450	450
Average	446.66	453.33	453.33

An important observation of this experiment result was the approximate equality of request counts. That request counts were calculated as using the *Equation 2*.

Considering the linear increment of the throughput,

$$Number\ of\ requests\ = \frac{time\ taken\ *\ (Starting\ TPS\ +\ Ending\ TPS)}{2}$$

Equation 2: Number of requests calculation

The calculation yielded the results in *Table 5-3*.

Table 5-3: Object count comparison

	Case 1: Start with 200 TPS	**Case 2: Start with 200 TPS**
Time taken to reach the instability	190s	150s
Starting TPS	200	300
Breaking point TPS	390	450
Number of requests	56050	56280

Therefore, $Object\ Count_{Case\ 1}\ \approx\ Object\ Count_{Case\ 2}$

This observation hints that there is a leak present in the system. With further analysis of the code, it is identified that DB connection pooling was not properly configured. With this bottleneck removal, the performance of the system improved greatly.

5.2.1.4 Case 3: Starting TPS: 800, without bottleneck in messaging flow

The same flow was tested against starting TPS 800 to benchmark the system. The yielded results on this experiment are listed in *Table 5-4*.

Table 5-4: Results of messaging flow performance test after bottleneck removal

	Without framework (TPS)	**Without sampling (TPS)**	**With Sampling (TPS)**
Attempt 1	1650	1650	1640
Attempt 2	1650	1650	1640
Attempt 3	1660	1640	1640
Average	1653.33	1646.66	1640

The breaking point is reached due to a bottleneck at asynchronous remote connectivity in 'message sender'. The performance of the specific component is acceptable according to used libraries and third party service. Therefore, this concluded the limit of optimization.

5.2.1.5 Case 4: USSD application performance.

The USSD application is also tested against the same simulator according to the same procedure. The experiment started with 1500 TPS. *Table 5-5* shows the results of this experiment.

Table 5-5: Results of USSD application performance test.

	Without framework (TPS)	**Without sampling (TPS)**	**With Sampling (TPS)**
Attempt 1	2130	2120	2090
Attempt 2	2110	2110	2070
Attempt 3	2120	2120	2090
Average	2120	2116.66	2083.33

Within the USSD flow test runs, the system reached the performance limit with a bottleneck in 'Message Sender'. This did not show any thread blockage and hence, this concluded that the external system is contributing to the bottleneck.

5.2.1.6 Case 5: Mobile-money-tracker application performance.

The mobile-money-tracker sample application is also tested against the same simulator. *Table 5-6* shows the results yielded from the test runs.

Table 5-6: Results of Mobile Money Tracker application performance test

	Without framework (TPS)	**Without sampling (TPS)**	**With Sampling (TPS)**
Attempt 1	2130	2100	2090
Attempt 2	2120	2110	2090
Attempt 3	2130	2100	2080
Average	2126.66	2103.33	2083.33

In this experiment, the system reached the instability due to 'Out of memory' error and bottleneck was visible in the Synchronous remote components. This concluded that either the bottleneck is within the database or the operating system limits the performance.

5.2.1.7 Analysis

The results of all three cases above are analyzed based on following criterion. The system without using the framework is taken as the baseline.

The percentage of the performance deviation of the other cases is calculated according to the *Equation 3.*

$$performance\ difference\ \% = \frac{(limiting\ TPS_x\ - limiting\ TPS_{base})}{limiting\ TPS_{base}} \times 100$$

Equation 3: Performance overhead percentage calculation

The error margin relative to limiting TPS of base is calculated using the *Equation 4*.

$$Error\ Margin\ \% = \frac{step\ size}{limiting\ TPS_{base}} \times 100$$

Equation 4: Error margin calculation

Complete result analysis and calculation of the experiment is listed in *Table 5-7*.

Table 5-7: Final calculation of the results and comparison to the error margin.

Case	**Error margin**	**Without sampling throughput deviation**	**With sampling throughput deviation**
Case 1	2.54%	0.84%	1.693%
Case 2	2.20%	1.47%	1.47%
Case 3	0.604%	0.402%	0.806%
Case 4	0.471%	0.157%	1.729%
Case 5	0.470%	0.626%	2.039%

The final results in the Table 5-7 shows that the performance impact of using 'concor' framework is significantly lower and, in most cases this value is even within error margin. Compared to the impact of a bottleneck, this performance drop is insignificant. Furthermore, in the real world scenarios, the systems are not expected to run in full potential and a safe margin is kept for the system to be stabilized.

5.2.2 Experiment 2: Overhead of thread pool switching

Since runtime reconfiguration of thread pool is also a part of the research, another experiment is designed to observe the impact on a thread pool addition and deletion, as follows.

In runtime, each of the components are configured to a stage and each stage is served with a specific thread model. When a thread pool is configured in runtime, there might be a performance impact in throughput due to two reasons.

- If the thread model changes from Multi-threaded to Single threaded, the system starts to act upon new thread model resulting performance decrease.
- Due to new queue, slight changes may occur in the throughput.

The experiment is designed to run in a constant TPS. In order to isolate the second case, in the experiment a thread pool is added as similar to the existing thread pool of the stage. For the deletion the same thread pool will be removed. Then the TPS in simulator side will be observed for the changes for 5 seconds.

The results in Table 5-8 were collected in the experiment with multi-threaded thread pool. A 1000 TPS constant throughput is maintained throughout the experiment.

Table 5-8: Throughput impact on thread pool addition and removal

	operation	**t s**	**(t + 1) s**	**(t + 2) s**	**(t + 3) s**	**(t + 4) s**
Attempt 1	**Add**	1000	1000	1001	999	1000
	Delete	996	1004	1000	1000	1000
Attempt 2	**Add**	1000	999	1001	1000	1000
	Delete	999	1001	1000	1000	1000
Attempt 3	**Add**	1000	1000	1000	1000	1000
	Delete	1000	1000	1000	1000	1000

Results in *Table 5-8* shows that the addition of a thread pool does not show any TPS impact, yet deletion of a thread pool may show a performance drop in the immediate second. Since this change is significantly low, this does not affect upon the overall performance.

5.3 Analysis

The results of the tests shows that the framework is capable of exposing the internal structure and realtime performance information in relatively low cost. The performance cost due to the instrumentation is relatively low and the features provided to disable instrumentation on production also provides the performance overhead production. However, in the real life scenarios, the systems are not expected to run on full potential. The performance difference introduced by the framework, in most cases, is not significant enough to alter the estimated performance measures.

Furthermore, the dynamic changes introduced within the research, especially the dynamic concurrency, are also tested against the load and the result shows lesser impact on the performance on those changes. In most cases, the value provided by the framework outweighs the performance penalty.

5.4 Summery

This chapter was mainly focusing on testing and evaluating the PoC implementation related to research. The evaluation was conducted on two aspects as follows. The framework should operate within the acceptable performance overhead limits and the dynamic changes should not introduce additional significant performance impacts to the system. The results shows that the framework is within the required bounds.

Chapter 6

CONCLUSION AND FUTURE WORKS

6.1 Conclusion

There are multiple objectives of this research. First one being the instrumentation, the output of this research, ‘concor’ framework, abstracts out the instrumentation probing point placement mechanism. The framework was built using the SEDA architecture as the base. This architecture provides properly defined stage separation mechanism and, these boundaries of the components were selected as the most suitable probing points.

The stat collection mechanism is designed using sampling as the main data collection mechanism. For a configured number of request, the sampling contexts were sent to collect the data. This result the low overhead data collection compared to the probing of each message. A Web UI based tool is also provided with the framework, so that not only the flow stats itself but also the internal topology of the application is also can be visualized graphically.

Apart from the instrumentation, the ability to reconfigure the system in runtime is also provided with the framework. This is useful in designing the system as the designer have the room to correct the concurrency problems even in runtime. Furthermore, this effectively prevents rocky developers making concurrency related mistakes within the application.

While this framework provides those features, there is a risk that this additional features may reduce the performance of the application altogether. In order to prove that this framework does not affect the runtime performance, two experiments were designed. In first experiment, a sample application build with ‘concor’ framework is tested until reaching the unstable point against increasing TPS. This experiment results showed that the specific features does not provide a significant additional burden to the production runtime.

In the second experiment, runtime thread configuration is tested against stable TPS. This experiment showed that the reconfiguration does not effect on the run time significantly.

This concluded that ‘concor’ framework does provide a solution for the runtime reconfiguration and instrumentation at the same time. Yet they are other problems need to be solved within this research like the possible *reference leak, Thread local values* and *Enforcing behaviors across the platform.* These aspects are discussed in the future works section.

6.2 Future Works

As mentioned above, following problems are still exists within the application.

A careless developer may expose an internal mutable object from one stage and modify that in another stage, which may cause concurrency issues. This is a common issue in current development practices, yet the static nature of the thread pools mitigates this issue for some extend. A proper guideline or a proper solution should be researched to prevent this fault being occur.

A careless developer may reuse the same single threaded component in different places making the specific single threaded component being accessed from multiple threads effectively. This will raise concurrency issues. A proper mechanism should be investigated to prevent the single threaded components being reused in multiple places.

The thread local value transfer between stages is not yet implemented. This is a mandatory feature for the use cases such as MDC and MySQL transactions. A proper low overhead solution should be investigated to solve the thread local value transfer between stages.

This framework is designed as a proof of concept using plain java. This concept can be used in other environments like .Net and python also. Research should be conducted on implementing this mechanism in other platforms and the related problems.

One of the biggest limitation of this framework is the inability to coexist with the other java data flow based frameworks like RxJava. Research should be conducted to analyze and propose better ways of integrating this concept in those frameworks.

REFERENCES

[1] "Java Mission Control." [Online]. Available: https://www.oracle.com/technetwork/java/javaseproducts/mission-control/index.html. [Accessed: 16-May-2019].

[2] "Apache JMeter™." [Online]. Available: https://jmeter.apache.org/. [Accessed: 16-May-2019].

[3] M. Shaw and P. Clements, "A field guide to boxology: Preliminary classification of architectural styles for software systems," in *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, 1997, pp. 6–13.

[4] M. Welsh, D. Culler, and E. Brewer, "SEDA: an architecture for well-conditioned, scalable internet services," in *ACM SIGOPS Operating Systems Review*, 2001, vol. 35, no. 5, pp. 230–243.

[5] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theor. Comput. Sci.*, vol. 410, no. 2–3, pp. 202–220, 2009.

[6] "JEP 107: Bulk Data Operations for Collections." [Online]. Available: http://openjdk.java.net/jeps/107. [Accessed: 18-May-2019].

[7] "Scala 2.8 Collections API." [Online]. Available: https://www.scala-lang.org/docu/files/collections-api/collections.html. [Accessed: 18-May-2019].

[8] "ReactiveX, 'ReactiveX/RxJava,' GitHub." [Online]. Available: https://github.com/ReactiveX/RxJava. [Accessed: 18-May-2019].

[9] H. Gomaa, "Catalog of Software Architectural Patterns," in *Software Modeling and Design*, pp. 495–520.

[10] "Web on Reactive Stack." [Online]. Available: https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html. [Accessed: 19-May-2019].

[11] "The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 163." [Online]. Available: https://www.jcp.org/en/jsr/detail?id=163. [Accessed: 19-May-2019].

[12] M. Gagliardi and Y. Sun, "Conditional dynamic instrumentation of software in a specified transaction context." Google Patents, 2013.

[13] M. Dmitriev, "Design of JFluid: A profiling technology and tool based on dynamic bytecode instrumentation," 2003.

[14] P. Abercrombie and M. Karaorman, “jContractor: Bytecode instrumentation techniques for implementing design by contract in Java,” *Electron. Notes Theor. Comput. Sci.*, vol. 70, no. 4, pp. 55–79, 2002.

[15] J. Whaley, “A portable sampling-based profiler for Java virtual machines,” in *Proceedings of the ACM 2000 conference on Java Grande*, 2000, pp. 78–87.

[16] M. Rohr *et al.*, “Kieker: Continuous monitoring and on demand visualization of Java software behavior,” 2008.

[17] “‘JSR 51: New I/O APIs for the JavaTM Platform,’ The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 51.” [Online]. Available: https://www.jcp.org/en/jsr/detail?id=51. [Accessed: 16-May-2019].

[18] “‘JSR 203: More New I/O APIs for the JavaTM Platform (“NIO.2”),’ The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 203.” [Online]. Available: https://www.jcp.org/en/jsr/detail?id=203. [Accessed: 16-May-2019].

[19] M. Welsh and D. E. Culler, “Adaptive Overload Control for Busy Internet Servers.,” in *USENIX Symposium on Internet Technologies and Systems*, 2003, p. 4.

[20] S. Ertel, C. Fetzer, and P. Felber, “Ohua: Implicit Dataflow Programming for Concurrent Systems,” in *Proceedings of the Principles and Practices of Programming on The Java Platform*, 2015, pp. 51–64.

[21] “Clojure.” [Online]. Available: https://clojure.org/. [Accessed: 16-May-2019].

[22] E. Kiciman, B. Livshits, M. Musuvathi, and K. C. Webb, “Fluxo: a system for internet service programming by non-expert developers,” in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 107–118.

[23] G. A. Agha, “Formal methods for Actor systems: A progress report,” in *Proceedings of the IFIP TC6/WG6. 1 5th International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols-FORTE’92*, 1993, pp. 217–228.

[24] T. Van Cutsem, S. Mostinckx, and W. De Meuter, “Linguistic symbiosis between event loop actors and threads,” *Comput. Lang. Syst. Struct.*, vol. 35, no. 1, pp. 80–98, 2009.

[25] “‘Akka: build concurrent, distributed, and resilient message-driven applications for Java and Scala | Akka,.’” [Online]. Available: https://akka.io/. [Accessed: 16-May-2019].

[26] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton, “Comparing the performance of web server architectures,” in *ACM SIGOPS Operating Systems Review*, 2007, vol. 41, no. 3, pp. 231–243.

[27] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, “A catalog of stream

processing optimizations," *ACM Comput. Surv.*, vol. 46, no. 4, p. 46, 2014.

[28] B. Gedik, H. G. Özsema, and Ö. Öztürk, "Pipelined fission for stream programs with dynamic selectivity and partitioned state," *J. Parallel Distrib. Comput.*, vol. 96, pp. 106–120, 2016.

[29] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart, "Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads," *Tech. Pap. LMAX, May*, p. 206, 2011.

[30] "'spring.io,' Spring." [Online]. Available: https://spring.io/. [Accessed: 19-May-2019].

[31] "'Java Management Extensions (JMX) Technology,' Java Management Extensions (JMX)." [Online]. Available: https://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html. [Accessed: 21-May-2019].

[32] "'React – A JavaScript library for building user interfaces,' – A JavaScript library for building user interfaces." [Online]. Available: https://reactjs.org/. [Accessed: 21-May-2019].

[33] "MySQL." [Online]. Available: https://www.mysql.com/. [Accessed: 21-May-2019].

[34] "'ETSI TS 124 390 V11.0.0 : Unstructured Supplementary Service Data (USSD) specification.,' ETSI TS 124 390 V11.0.0 : Unstructured Supplementary Service Data (USSD) specification." [Online]. Available: https://www.etsi.org/deliver/etsi_ts/124300_124399/124390/11.00.00_60/ts_124390v110000p.pdf. [Accessed: 21-May-2019].

[35] "'WireMock,' WireMock." [Online]. Available: http://wiremock.org/. [Accessed: 21-May-2019].

APPENDIX A: Sample simple task implementation

```
public class MOInTranslator implements SimpleTask<String, MOMessage> {

        private static final Logger logger = LoggerFactory.getLogger(MOInTranslator.class);

        private static final Gson GSON = new Gson();

        @Override
        public MOMessage apply(String s) throws Throwable {
        logger.debug("Translating message [{}]", s);
        Msg msg = GSON.fromJson(s, Msg.class);
        return new MOMessage(msg.msisdn, msg.shortCode, msg.message);
        }
}
```

APPENDIX B: Sample single threaded task implementation

```
public class MOSessionManagerWrapper implements SingleThreadedTask<MOMessage, MOMessage> {

        private static final Logger logger = LoggerFactory.getLogger(MOSessionManagerWrapper.class);

        private SessionManagerI sessionManagerI;

        public MOSessionManagerWrapper(SessionManagerI sessionManagerI) {
        this.sessionManagerI = sessionManagerI;
        }

        @Override
        public MOMessage apply(MOMessage moMessage) throws Throwable {

        logger.debug("Resolving session for [{}]", moMessage.getFrom());

        moMessage.setSession(sessionManagerI.getSession(moMessage.getFrom()));

        logger.debug("Session has been successfully resolved");
        return moMessage;
        }

}
```

APPENDIX C: Sample Synchronous remote task implementation

```
public class EndPointResolver implements SynchronizedRemoteTask<MOMessage, MOMessage> {

	private static final Logger logger = LoggerFactory.getLogger(EndPointResolver.class);

	private Map<String, Optional<RemoteInfo>> remoteInfoCache = new ConcurrentHashMap<>();

	@PostConstruct
	public void init() {
	Executors.newSingleThreadScheduledExecutor()
	.scheduleAtFixedRate(() -> remoteInfoCache.clear(), 1, 1, TimeUnit.SECONDS);
	}

	@Autowired
	private ServerConfigRepository serverConfigRepository;

	@Override
	public MOMessage apply(MOMessage moMessage) throws Throwable {

	logger.debug("Resolving destination information");

	Optional<RemoteInfo> remoteInfo =
serverConfigRepository.findByRemoteId(moMessage.getSession().getApplication());

	remoteInfo.ifPresent(moMessage.getSession()::setApplicationInfo);

	logger.debug("Destination information resolved");
	return moMessage;
	}
}
```

APPENDIX D: Sample Synchronous remote task implementation

```
public class ATMessageSender implements TransitionTask<MOMessage, MOMessage> {

	private static final Logger logger = LoggerFactory.getLogger(ATMessageSender.class);

	private OkHttpClient client;

	private AtomicReference<DurationData> durationData = new AtomicReference<>(new DurationData());

	@PostConstruct
	public void init() {
	client = new OkHttpClient();
	client.dispatcher().setMaxRequestsPerHost(400);

	Executors.newSingleThreadScheduledExecutor()
	.scheduleAtFixedRate(
			() -> logger.info("Connection Count [{}]| average time [{} ms]",
client.connectionPool().connectionCount(), durationData.getAndSet(new DurationData()).avg())
			, 1, 1, TimeUnit.SECONDS);
	}


	@Override
	public void apply(MOMessage moMessage, Continuation<MOMessage> continuation) throws Throwable
{

	logger.debug("Sending message to remote server");

	Request request = new
Request.Builder().url(moMessage.getSession().getRemoteInfo().getServerInfo().get(0).getUrl())
	.post(RequestBody.create(MediaType.get("text/plain"), moMessage.getResponse()))
	.build();

	logger.debug("Message preparation completed");

	long before = System.currentTimeMillis();

	client.newCall(request).enqueue(new Callback() {
	@Override
	public void onFailure(Call call, IOException e) {
	logger.error("An error occurred while sending the message");
	durationData.updateAndGet(data -> data.addDuration(System.currentTimeMillis() - before));
	continuation.onError(() -> e);
	}

	@Override
	public void onResponse(Call call, Response response) throws IOException {
	logger.debug("Message successfully delivered");
	durationData.updateAndGet(data -> data.addDuration(System.currentTimeMillis() - before));
	continuation.continuing(() -> moMessage);
	if (response.body() != null) {
	response.body().close();
	}
	}
	});

	logger.debug("Message sent to the remote server");
	}
}
```

APPENDIX E: Sample Catch task implementation

```
public class Catching implements CatchTask<MOMessage> {

        private static final Logger logger = LoggerFactory.getLogger(Catching.class);

        @Override
        public MOMessage onError(Throwable e) throws Throwable {
        logger.error("An error occurred while executing the request", e);
        return null;
        }
}
```

APPENDIX F: Sample Flow composition

```
Flow<String> flow = Flows.<String>create("mo-flow")
        .map(new MOInTranslator(), "translator")
        .mapSingleThreaded(moSessionManagerWrapper, "session Manager")
        .map(moRouter, "Router")
        .mapSynchronizedRemote(endPointResolver, "End Point Resolver")
        .map(new ATTranslator(), "AT translator")
        .bind(atMessageSender, "Message sender")
        .forEach(new MOTransLogger(), "trans logger")
        .catching(new Catching(), "Error Handler")
        .build();
```