

TALLINN UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Software Science

Pavel Lavrešin 083834 IAPB

**APPLYING REACTIVE PROGRAMMING APPROACH  
IN IOT BASED RELIABLE CROSSROAD TRAFFIC  
MONITORING**

Bachelor's thesis

Supervisor: Martin Rebane  
MSc

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Tarkvarateaduse instituut

Pavel Lavrešin 083834 IAPB

**REAKTIIVSE PROGRAMMEERIMISE RAKENDAMINE  
IOT BAASIL TÖÖKINDLA  
RISTMIKUMONITOOINGU EHTAMISEKS**

Bakalaureusetöö

Juhendaja: Martin Rebane  
MSc

Tallinn 2018

## **Author's declaration of originality**

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Pavel Lavrešin

May 21, 2018

## **Abstract**

Crossroad traffic monitoring system is highly available and reliable IoT based crossroad traffic control system that leverages following set of technologies like IoT (Internet of Things), ML (Machine Learning) for and real-time video stream and time-series stream of sensors data. All these components and technologies combined together generate huge number of events, messages and workload for both local and cloud infrastructure adding also high network throughput.

Author is confident that the main constraint that needs to be addressed in such IoT based crossroad traffic monitoring is the efficiency of data processing pipeline and reliability of the mission critical services without having to sacrifice and high performance in distributed multi-threaded environment.

The aim of this work is to apply Reactive programming paradigms in the IoT based application, ensuring that built system remains reliable and high-performing.

This paper brings out two implementations based both on Actors and Reactive streams models as well as sheds some light on inefficiency of traditional OLTP / CRUD -like approach when it comes to designing an elastic and scalable solutions. Author describes how Actors and Reactive programming help to overcome limitations of traditional object-oriented programming models and allow to build concurrent, fault-tolerant and self-healing systems.

Finally, having implemented both Actors and Streams based solutions, author shares analysis and collected application metrics, proving that developing applications in a Reactive way is fully justified and highly recommend when building resilient, elastic and fault-tolerant systems.

This thesis is written in English and has 40 pages, consisting of 5 chapters with 19 figures and 3 tables.

## Annotatsioon

Ristmiku liikluse monitooringu rakendus ise ja süsteem tervikuna on loodud eesmärgiga olla töökindel ja kogu aega saadaval vaatamata suurele töökoormusele. Monitooringu süsteem sisaldab endas keerulist IoT baasil ehitatud riistvara ning ka töökindlat ja täpset tarkvara mis ette nähtud video töötlemiseks ja autode tuvastamiseks tänu masinõpe rakendamisele. Kõik need ülalmainitud komponendid koos genereerivad suurt mahtu andmetest ja töökormust nii lokaalsetele seadmetele, kuid ka pilveteenustele. Lisaks, nõuab selline süsteem ülitäpset ja ülikindlat juhtimist ja arendust et jääda kättesaadavaks ja reageeritavaks olles samuti ka vastupidav.

Autor leiab, et tuleb pöörata tähelepanu just sellele, kuidas on projekteeritud sellise IoT baasil ristmiku tarkvara ning kas selles tarkvaras on arvestatud ka sellega, mida pakkuvad modernsed ja jaotatud süsteemid.

Selle töö põhieesmärk on rakendada Reaktiiv programmeerimise põhimõtteid ja eeliseid IoT baasil olevale ristmiku monitooringule ning veenduda, et süsteem on töökindel ja kiire.

Selles töös uuritakse kaks erinevat Reaktiivset lahendust – üks on Actor mudeli põhinev lahendus ning teine on Reaktiiv stream-i lahendus. Lisaks, viitab autor sellele, kuivõrd ebaefektiivsed ning mitte väga töökindlad on olemasolevad CRUD / OLTP baasil tehtud lahendused, milleks on enamuse veebirakendusi. Autor kirjeldab, kuidas aitab Actor mudeli ja Reaktiiv stream-i põhjal tehtud lahendus lahendada skaleeruvuse muresid, olles samas ajal väga võimas ja töökindel suurte andmemahutude korral.

Töö tulemusena leiab autor, et Reaktiiv stream-i põhjal tehtud rakendus võrreldes Actor mudeliga palju efektiivsem andes rohkem võimalusi juba baaslahendusena ning olles väga paindlik konfigureerimisel. Lisaks, Reaktiiv stream-i lahendus on tarkvaraarendaja silmades tunduvalt kergemini hallatav ja toetatav edaspidi ning ei nõua madalal tasemel koodi kirjutamist nagu Actor mudel. Lisaks, autor analüüsib kogutud süsteemsed

mõõdikud, mis olid mõlema rakenduste töökäigust kogutud et tõestada põhieesmärki saavutamist.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 40 leheküljel, 5 peatükki, 19 joonist ja 3 tabelit.

## List of abbreviations and terms

<b>ACID</b>	ACID (Atomicity, Consistency, Isolation, Durability) of database transactions intended to guarantee validity even in the event of errors, power failures, etc. In the context of databases, a sequence of database operations that satisfies the ACID properties, and thus can be perceived as a single logical operation on the data, is called a transaction. [41]
<b>ADT</b>	Abstract data type is a mathematical model, where a data type is defined by its behavior
<b>Akka</b>	Akka framework a set of open-source libraries for designing scalable, resilient systems that span processor cores and networks [32]
<b>API</b>	API is a set of functions and procedures that allow the creation of applications which access the features or data of an operating system, application, or other service. [42]
<b>Back-pressure</b>	Back-pressure is an ability to notify producer (upstream) to slow-down due to consumer (downstream) being too slow. It allows components in your system to react resiliently (e.g. not consuming an unbounded amount of memory) and predictably, all in a non-blocking manner [40]
<b>CRUD</b>	CREATE, READ, UPDATE and DELETE operations (as an acronym CRUD) are the four basic functions of persistent storage.
<b>CQRS</b>	CQRS stands for Command Query Responsibility Segregation. It's a pattern that I first heard described by Greg Young. At its heart is the notion that you can use a different model to update information than the model you use to read information [43]
<b>DDD</b>	It is a development approach that deeply values the domain model and connects it to the implementation. DDD was coined and initially developed by Eric Evans. [44]
<b>DSL</b>	DSLs are small languages, focused on a particular aspect of a software system. You can't build a whole program with a DSL, but you often use multiple DSLs in a system mainly written in a general-purpose language. [45]

<b>FP</b>	Functional programming is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. [46]
<b>IoT</b>	Internet of Things - everyday objects and devices connected to the web and providing additional data and / or functionality
<b>MQTT</b>	Message Queue Telemetry Transport or MQTT is Machine 2 Machine pub sub platform. Publisher can publish anything on some channel and subscriber can subscribe to channel and listen to publisher [47]
<b>OOP</b>	Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. [48]
<b>OLTP</b>	OLTP (Online Transaction Processing) is characterized by a large number of short online transactions (INSERT, UPDATE, DELETE). The main emphasis for OLTP systems is put on very fast query processing, maintaining data integrity in multi-access environments and an effectiveness measured by number of transactions per second [49]
<b>PaaS</b>	Platform as a Service, often simply referred to as PaaS, is a category of cloud computing that provides a platform and environment to allow developers to build applications and services over the internet [50]
<b>REST</b>	REST, or Representational State Transfer, is an architectural style for providing standards between computer systems on the web, making it easier for systems to communicate with each other [51]
<b>SaaS</b>	Software as a service is a software licensing and delivery model in which software is licensed on a subscription basis and is centrally hosted [52]
<b>UI</b>	User interface of a product, where human and machine interaction occurs
<b>UX</b>	User experience perspective and satisfaction of product



## Table of content

1 Introduction	13
1.1 Common approach to software development	14
1.2 Reactive programming approach	17
2 Reactive paradigms in IoT	20
2.1 Crossroad traffic monitoring designed in CRUD way	20
2.2 Event Sourcing	22
2.3 Reactive programming paradigms and system design	27
2.3.1 Reactive manifesto	28
2.4 Actor model	29
2.5 Reactive streams	31
3 Overview of the proposed solution	33
3.1 Crossroad traffic operation problems	33
3.2 Proposed reactive solution	37
4 Implemented solution	39
4.1 Crossroad IoT devices	39
4.2 Vehicle recognition from video stream	41
4.3 Time-series metrics from IoT devices	42
4.4 Software product implementation	43
4.4.1 Platform	43
4.4.2 Programming language	44
4.4.3 Frameworks	44
4.4.4 Actor based implementation	45
4.4.5 Reactive streams-based implementation	47
4.4.5 Recorded system performance metrics	49
4.5 Analysis of results	50
5 Summary	52
References	54
Appendix 1 – Reactive-crossroad repository [38]	58
Appendix 2 - Time-series metrics for vehicle weight measurements	59
Appendix 3 - Time-series metrics for surface measurements	60
Appendix 4 - Time-series metrics for weather measurements	61
Appendix 5 - Defined commands according to DDD / CQRS model	62
Appendix 6 - Defined events according to DDD / CQRS model	63
Appendix 7 - Persistent Actor per every Crossroad	64

Appendix 8 - Worker Actor for vehicle recognition stream	65
Appendix 9 - Worker Actor for Time-series metrics	66
Appendix 10 - Handling of video recognition results with Akka streams	67
Appendix 10 (continued) – Handling of video recognition results with Akka streams	68
Appendix 11 – Actors vs Streams Threads utilization	69
Appendix 12 – Actors vs Stream Memory / Heap utilization	70
Appendix 13 – Actors vs Streams GC performance	71

## List of figures

Figure 1. High-level overview of OLTP-like application for crossroad monitoring.....	16
Figure 2. Simple CRUD implementation [4] .....	16
Figure 3. Interactions in crossroad monitoring system when built in CRUD way.....	20
Figure 4. State model and mutation in event-sourced systems .....	23
Figure 5. Crossroad monitoring system in Even Sourcing way .....	25
Figure 6. Eventual consistency and CQRS by Greg Young, MSDN, Microsoft [15]....	26
Figure 7. The Reactive Manifesto. <a href="https://www.reactivemanifesto.org/">https://www.reactivemanifesto.org/</a> [17].....	29
Figure 8. Actors communication with each other [19] .....	30
Figure 9. Example of Ask and Tell patterns in Actors .....	31
Figure 10. Publisher-subscriber like communication for Subscriber-Observable [20] ..	31
Figure 11. A Bird's Eye View to proposed IoT setup .....	36
Figure 12. Proposed implementation based on Event Sourcing, CQRS and DDD.....	37
Figure 13. Reactive Crossroad implementation stages .....	39
Figure 14. Visual Object Detection by YOLO algorithm [28].....	41
Figure 15. Actors-based implementation design .....	46
Figure 16. Akka-streams based implementation design .....	47
Figure 17. Akka streams-based back-pressure explained .....	48
Figure 18. JVM version .....	49
Figure 19. Threads utilization for Reactive Streams application .....	50

## List of tables

Table 1. Proposed hardware list for intelligent traffic monitoring.....	34
Table 2. Proposed software technologies used for intelligent traffic monitoring .....	35
Table 3. Proposed IoT devices for intelligent crossroad.....	40

# 1 Introduction

The IoT (*Internet of Things*) industry is relatively new but extremely growing and evolving industry that is surrounding us every day and transforming the way we do our routine daily tasks either at home or at work. It is estimated that there will be ca 200 billions of “smart” devices [\[1\]](#) around, connected to the web and capable to exchange and analyze collected data from various embedded devices and sensors.

Applying the Internet of Things technology in traffic management and regulation systems is a natural way towards better and more comfortable cities with cleaner urban environment if we want to make our life in the rapidly developing and growing cities more smarter and safer.

Unfortunately, putting all hardware like sensors, cameras and computers together won't be sufficient to build an intelligent system that we have described above.

Hardware is nothing without good software hence building a good and reliable IoT based system requires implementing software that can support IoT at large scale and being both resilient and responsive is rather non-trivial task and requires much more effort than traditional web-based product development most are familiar with. Moreover, it assumes that product architecture is designed in such way that it is meant to be used by millions, handle large volumes and in addition respond to user actions in few milliseconds.

The main goal of this Thesis is an attempt to apply Reactive programming approach and core paradigms in order to build a system that can scale, survive failures and remain coherent and reliable under significant load in the resources constrained environment. There are multiple paradigms available that make an application reactive compliant although this thesis work strictly focuses on Actors model [\[18\]](#) and Reactive Streams [\[6\]](#). As a foundation for both implementation, Akka framework [\[32\]](#) has been chosen due to wide range of available components under the hood and out of the box that can be put together to achieve a system, built in a reactive way.

There are few success criteria this Thesis is challenging to meet with a software product implementation. First of all, being reactive means utilizing common patterns as to be able to communicate within components in a message-driven way. Secondly, application should tolerate rapidly increased load and remain resilient and elastic under load. Lastly, the Reactive Streams based implementation should be most sophisticated solution available hence providing multiple features such as back-pressure, self-healing and parallelism literally for free at the same time utilizing system resources most carefully. The latter one should be justified and proven at least from JVM (Java Virtual Machine) resources utilization and backed by time-series metrics.

This Thesis provides repository and code excerpts that have been used to build a baseline for proposed IoT reactive solution as well as ad-hoc simulation. However most of the work author has carried has been done in the terms of the Proof of Concept and has not been battle-tested in production, following should be rather easily applicable to production-ready implementation if provided domain entities and reactive foundation is used.

Thesis contains of 4 chapters that shed light to existing software engineering issues encountered by using traditional CRUD / OLTP patterns as well as core Reactive programming paradigms, especially in the context of IoT devices load and scale. Truly, one can implement any product and keep optimizing it until the hardware limits are reached but here author is confident enough that the core values and performance gains should be taken from the Reactive platform and its paradigms with all the additional values provided.

## **1.1 Common approach to software development**

Traditionally, there are software engineering patterns and technologies known for decades already and they still remain popular and widely applied. When one is required to implement an abstract web application that should solve any of business goals that requires state management, an engineer would probably opt for having a monolith server-side application, single relational database and regular REST API (for fetching persisted data).

This approach described above is widely used, accepted and applicable for most web applications that is highly appreciated, sometimes even referred to as a cornerstone of software engineering. Following approach is often described as CRUD (*Create, read, update, delete*), which is a common set of verbs used to build a system which is intended to operate with data at any scale.

Being actually a very verbose acronym for describing a set of operations that are being done with data through different user flows, CRUD is actually much more than a way to build software - it is a requirement to have a strong consistency between all operations, which is critical for software to remain truthful and usable. CRUD applications are also often called as OLTP (*Online transaction processing*) systems [\[2\]](#).

The typical OLTP system stack has not changed for a while and can be characterized as:

- Single-threaded monolith server-side application, written in Java or PHP/Python/JS
- Relational data storage provided by popular RDBMS like MySQL and PostgreSQL, ACID-compliant to support strong consistency, highly normalized data structures
- Mixed usage of transactions, pessimistic and/or optimistic locking techniques to provide consistency

The OLTP stack is relatively standardized and well described and by following this stack prerequisites and best practices it should be quite trivial to build any complex system from scratch typically very fast. It does solve most business goals for instance in retail industry and finances however requires significant effort to build a proper monitoring to prevent any operational data loss that is persisted in RDBMS, because it is often the single source of truth for most OLTP application. In addition, OLTP applications are often very good for any kind of reporting and data mining needs. Combined with OLAP (*Online analytical processing*) systems, the main data storage is offloaded in favor of a new analytical database that is used for reporting and analysis hence allowing to retrieve data faster and perform more heavy-weight queries without affecting primary storage. Considering that it should be relatively straightforward to design any application according to CRUD best practices, the author has decided to model a high-level overview of how the crossroad monitoring could look like when designing it as an OLTP application.

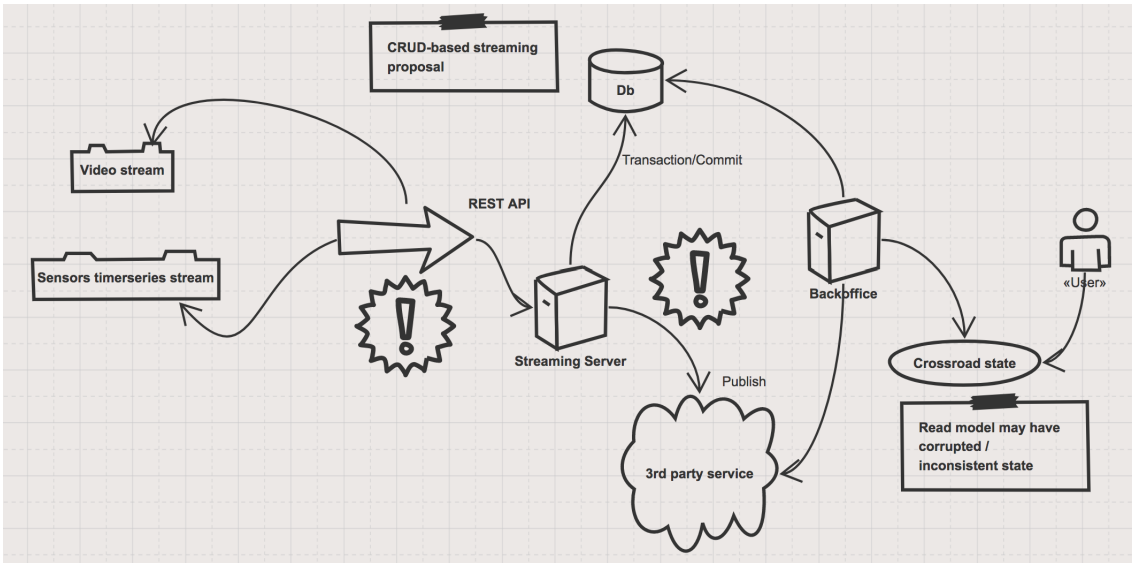


Figure 1. High-level overview of OLTP-like application for crossroad monitoring

OLTP applications are perfectly fine to use when one needs to persist data and perform lightweight computations in a monolith environment that has a single ACID-compliant data storage. This allows software engineers to focus on solving business problems rather than infrastructural issues and it is ensured that consistency is maintained by database in case of failure. It also works perfectly fine until that moment when engineers and business do not care much about previous state of application, and especially how did the application state actually evolve over time [3].

Once there is a demand to scale an existing application due to increased load and business needs, a common approach is to migrate from monolithic architecture to microservices / containers-based architecture like shown on the figure below.

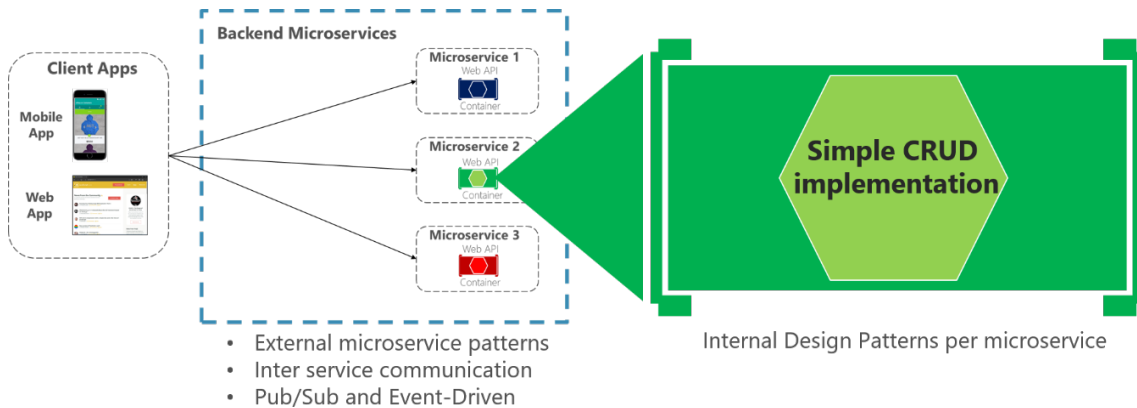


Figure 2. Simple CRUD implementation [4]



From architectural design perspective, it should be rather trivial to break down into loosely coupled microservices a proposed crossroad monitoring system however this will definitely lead to unpredicted failures and need to implement long-lasting transactions across microservices hence in result it will still be tightly coupled and database dependent.

The naive implementation of microservices based architecture for OLTP application might be as follows:

- Incoming data is consumed via blocking REST API
- A transaction is initiated, data fetched, transformed, nested transactions are made, data is written to database and eventually committed
- Response is returned to caller
- Not responsive, not scalable and not event-driven at all

Another significant flaw that is common to OLTP systems is a usage OOP (*Object-Oriented Programming*) paradigm. OOP is a first and foremost choice when it comes to developing an application these days.

Modelling the world and entities in the code using OOP is relatively easy and straightforward cause one can define objects as a composition of fields and methods. By invoking a method belonging to this object you should expect a synchronous response and hence all the operations in the OOP programming model are blocking by design.

Mainly, OOP paradigm is heavily praised for being understandable, reusable, testable and extensible but not efficient enough in modern distributed and scalable systems hence there is a need for a better approach.

## **1.2 Reactive programming approach**

In order to overcome limitations imposed by OOP programming model such as a need to interact with object and mutate state in blocking synchronous way, we shall have a look at Reactive Programming - functional event-driven programming model.

In today' asynchronous world we are getting across each other it is highly critical to maintain both speed of communication as well as data integrity without sacrificing one or another.

If we observe the constantly changing environment we live, we can spot that all the entities around us are always in motion. Whether these are stock exchanges, air traffic, vehicle traffic jams, public commutes or weather - we clearly see that things are constantly evolving, sometimes that we cannot even spot the current state willing to press a pause. Often, we are able to influence things happening around us in some controller manner, however many things just happen asynchronously at the same in the background forming a stream of events and generating new values in order to update current state. Such stream of events is grassroots of Reactive programming and adamantly defines main mantra of this approach: events are data and data are events.

Streams fit very well into the modern distributed and scalable environments. Hardware has evolved and changed drastically during recent years, where we have reached the limits of Moore's Law [5] and hence cannot double the speed of CPU units every other year. As a result, the free lunch is now over and applications do not improve automatically once new generation of CPU is released [5]. Instead, hardware makers keep adding multiple physical and virtual cores requiring engineers to adapt to completely different environment with multicore, cloud and containers-based architectures being de-facto top choice for most projects.

IoT essentially expect very high SLA while still having great throughput and availability which, in my opinion, cannot be achieved with classical OOP paradigm being fundamentally different.

Therefore, there have been lots of discussions and proposals how to handle increasing load from IoT devices and stream that data remaining in concurrent way none of these proposals has been better than Reactive Programming. Reactive programming isn't novel paradigm and has existed for decades already although gained popularity just recently when different reactive frameworks and extensions have appeared:

- Reactive Streams - an initiative, describing a standard way for asynchronous stream processing with non-blocking back pressure [6]
- ReactiveX - an API for asynchronous programming with observable streams [7]
- Akka Streams - a streaming interface of top of Akka actor systems, following Reactive Streams initiative [8]

Combined together with FP (functional programming), one can write coherent side-effect free code and enforces usage of immutable data structures that enable parallelism.

Moreover, algorithms used for filtering, persisting and modifying data can be parallelized as well ensuring great increase in speed while transforming data.

Sadly, engineers designing and implementing application alone with Reactive Programming in mind will not be successful at solving expected business goals because it provides mostly abstractions for data structures and operations without any hint how to build an application itself. Therefore, Reactive Programming has been recently enhanced with Event-Driven approach and combined together would allow us to build Reactive and Event-Driven application. Being event-driven means that system should:

- react to events - this is the cornerstone feature of event-driven system
- react to load - focus on overall system scalability rather than single user experience
- react to failure - being able to recover after inter

Such approach puts at the top of system design not only Reactive Programming concepts and model but also implies events being an essential foundation for reactive application. Events are immutable by design and these first of all act as notifications between various streams and flows, being passed from one to another carrying necessary payload and getting processed.

## 2 Reactive paradigms in IoT

The IoT based system is a distributed system by nature, which involves interaction of lots of independent services and devices in order to eventually deliver application current state to end-user, making it single coherent system. Such system has crucial expectations to software, expecting it to handle thousands of concurrent connections, tons of streamed data and provide real-time results to end-user within sub-second latencies. Designing intelligent crossroad monitoring system in an OLTP / CRUD like way would be feasible although very complicated in distributed environment, exposing several risks and likely causing failures. Greg Young, author of CQRS pattern has once said – “Oftentimes when writing software that will be cloud deployed you need to take on a whole slew of non-functional requirements that you don't really have...”. [9]

### 2.1 Crossroad traffic monitoring designed in CRUD way

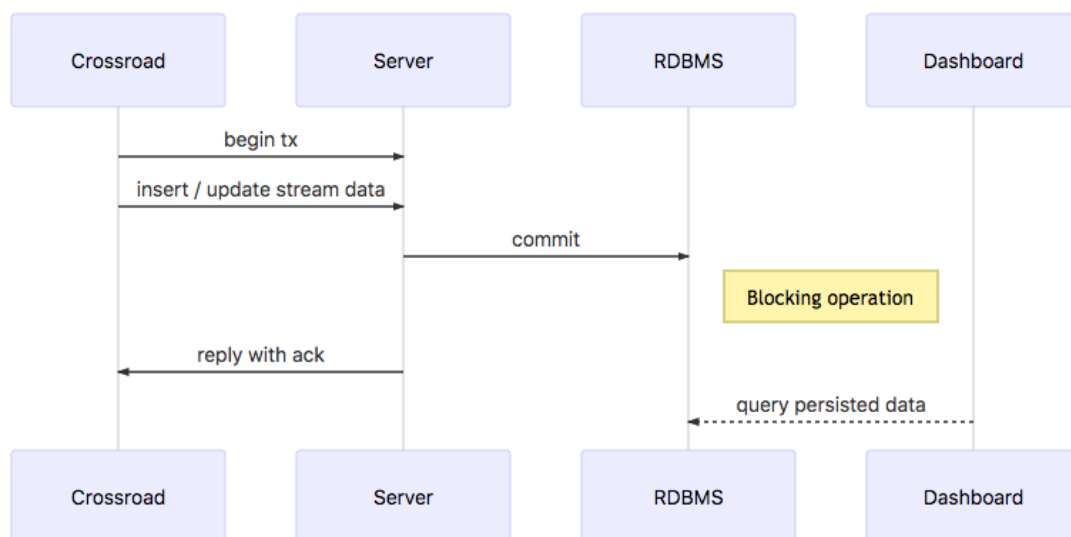


Figure 3. Interactions in crossroad monitoring system when built in CRUD way

Given the following proposed high-level design of crossroad monitoring system, we can easily spot several risks and failure points imposed by design and architecture of OLTP / CRUD applications. Trying to build and run this application in modern distributed

cloud environment, where every service would become a standalone micro-service, we shall ask ourselves following questions:

- What if something goes wrong among those tens microservices that have been deployed to cloud?
- What if crossroad API server goes down?
- What if database goes down?
- What if network goes down?
- What if orchestration services go down?

Combined with OLTP application inability to handle suddenly increased load and scale horizontally efficiently, one cannot expect smooth and reliable operation of such applications in modern SaaS (Software-as-A-Service) and PaaS (Platform as a Service) environments forcing software engineers to come up with crunches or workarounds without utilizing the power of cloud environments. The biggest known root cause for CRUD applications are failing in distributed environments, are issues with multiple threads accessing same shared instance and potential database integrity issues, also usually referred as shared mutable state is the root of all evil [\[10\]](#).

A common solution used to solve these issues mentioned above is an option to use some sort of locking mechanism - either optimistic or pessimistic locking.

Optimistic locking - is a locking scenario, when there is an assumption that conflicts due to concurrent reads of data are rare and hence concurrent edits are allowed until there is potential conflict that needs to be rolled back or failed [\[11\]](#). Pessimistic locking - is a much more stricter approach to maintain data integrity, which implies placing a lock on the database for small period of time assuming there will be a collision in the database. A significant drawback of pessimistic locking is a potential deadlock scenario due to data being locked and hence no other transactions and threads can access the same data [\[11\]](#). Software engineers often use locking approach as a silver-bullet in order to synchronize and serialize access to shared mutable state however forgetting about these drawbacks:

- Locks can severely limit concurrency as they require to halt threads and resume those once locks are released.

- From UX/UI perspective it looks rather badly and unacceptable when application is not responding for a while due to locks and prevents UI from being responsive enough.
- Eventually, the most harmful scenario that locks might badly introduce is a potential deadlock in application.

To sum up, on one hand we cannot guarantee that data integrity won't be violated and our shared mutable state for get corrupted without locks, on the other hand we sacrifice performance and accept potential deadlocks in favor of integrity. A possible solution to the problems listed above might be a two-phase commit solution. Two-phase commit is a solution used to verify that all the involved parties that are part of a single transaction have completed their work and hence it is safe to either commit or rollback in case of a failure [\[12\]](#). Unfortunately, two-phase commit comes at very high costs and eventually it just postpones that very moment when a failure may happen by reducing a window for a failure to cause a problem. Having that said, it seems that despite being so powerful and efficient, distributed systems cannot solve all the problems that OLTP system architecture actually introduces and hence applications crafted on top of this architecture are also renown for being unreliable and requiring lots of a consensus and locking mechanisms to remain consistent and reliable. A proposed solution would have been to build application using best of reactive programming, so that it could work in multi-threaded environment performing parallel computation in reactive and functional way.

Regretfully, the reactive programming itself cannot solve scalability and consistency problems although being very powerful programming paradigm. It requires that application foundation is built keeping immutability and parallel computations in mind, avoiding side-effects and performing state mutation in a clean and reliable way.

These fundamentals are Event Sourcing and Domain-Driven-Design.

## **2.2 Event Sourcing**

In general, event sourcing has been known and available for wide usage for decades already but unfortunately has not gained that much acknowledgement as well as

widespread usage among engineers and companies although being a great way to atomically update state and perform side-effects once events are persisted.

```
type State = Option[Aggregate]
// Command Handler
State => Command => F[List[Event]]
// Event Handler
State => Event => State
```

Figure 4. State model and mutation in event-sourced systems

Event sourcing intrinsic fundamentals are both simple and clever. The rule of thumb is to avoid storing current state while persisting all already occurred events in past tense to event store. Once these events are persisted, it shall be possible to compose current application state from events along with performing necessary side-effects [13]. Hence current state is a sum of the events applied. There are following rules that every event sourcing like application must obey:

- Persist into store.
- Append new events straight after existing ones.
- Never delete or rewrite these persisted events.

The grassroots of every event sourcing application are receiving set of commands, which conform to the intention of application user or caller. Commands do not imply yet any persistence though these can fail and be rejected by the system. The successful outcome of each command is either single or multiple events that shall be persisted, these are so-called immutable facts that have happened during application life cycle hence should be present in the store [13]. The store that is used in Event Sourcing is guaranteed to be the only source of truth. The latter one is insured by the following policy when reconstructing a state from the events store:

- We need to read all events sequentially starting from the very first event to reconstruct State.
- We have to persist events before continuing with effects.

- We might have to save snapshots to avoid replaying events from the beginning and have an option to recover from certain state.
- We enforce consistency in case of concurrent access to mutable state.

Additional value of event sourcing architecture includes following:

- Accurate audit logging - cause each state update is caused by single or multiple events, there is no other chance that state could be somehow updated or corrupted externally hence we can rely on event sourcing when there is a need to have 100% accurate audit logging. Following is rarely possible within traditional OLTP applications and requires additional overhead imposing several risks and performance implications.
- Simple historical queries and previous state replay - thanks to events persisted in strict order each having its own sequence number, it is relatively cheap and straightforward to rewind state back to past one and perform historical queries of business entities.
- Production system troubleshooting - it is known that chasing an error in production systems might be exhausting and tricky but thanks to events persisted in Event Store, one can easily replay those events up to that moment when error occurred to observe what actually went wrong.

That being said, author would like to propose a simplified design of Event Sourcing based system for crossroad monitoring in comparison to CRUD applications. Following is a proposed system design according to above mentioned pattern:



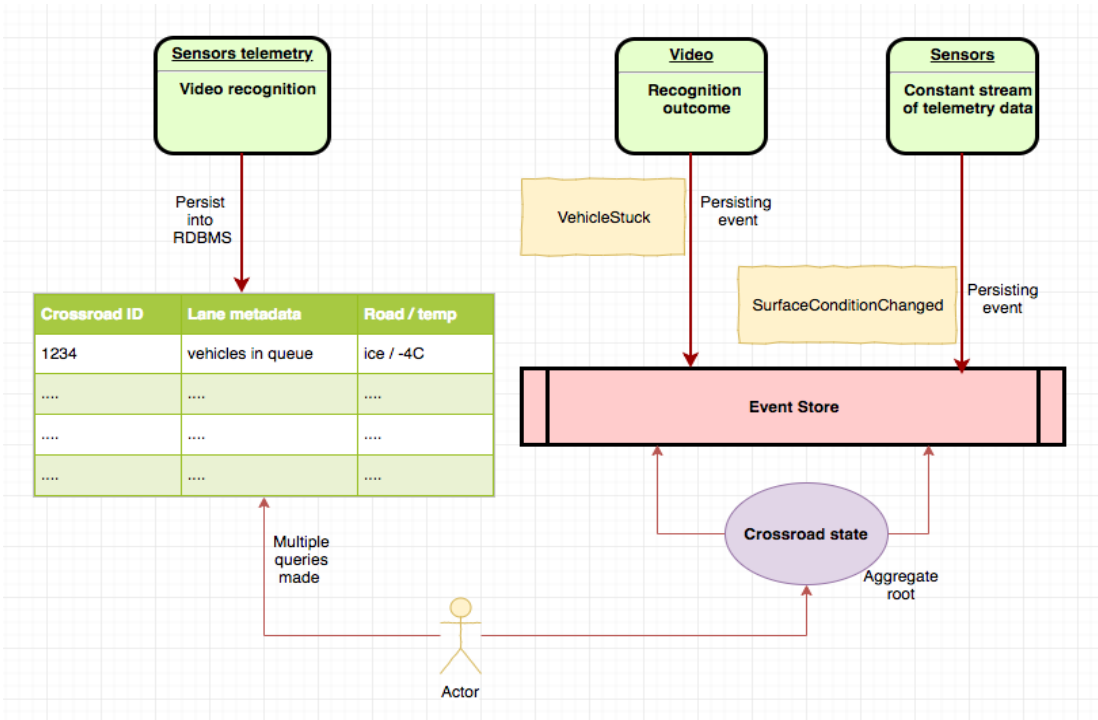


Figure 5. Crossroad monitoring system in Even Sourcing way

There are few things that have been added or changed in comparison to previous CRUD application design. Relational database as a single source of truth has been changed to Event Store [30], a database conceptually different when it comes to persistence as it is mostly meant for append only operations for persisting events that are in highly denormalized form hence the performance of Event Store storage is greatly improved. Every component that is collecting and providing crossroad related data is publishing events as a separate stream that is later persisted into Event Store hence thus processed concurrently without any data integrity or mutable state violation as it might have happened using traditional normalized model. Eventually, persisted data is available for end-user in an aggregated form thanks to materialized view, which is basically a result of subscription to different events streams and their composition to get eventual crossroad state at that moment of time. This is finally possible thanks to CQRS (Command / Query Responsibility Segregation Pattern) which is often used along with Event Sourcing due to multiple reasons [14].

Thanks to event sourcing we have scalable and append-only solution with all events persisted in a single store. Badly, all these events that are persisted as a result of processed commands are just small entities that don't form a high-level overview of the application, meaning that we cannot yet really on these entities persisted as they are just

literally small pieces of glass that needs to be glued together. In order to improve understanding of systems built with Event Sourcing in mind, there is a proposal to apply following architecture like Command Query Responsibility Segregation, which is a complementary addition to Event Sourcing.

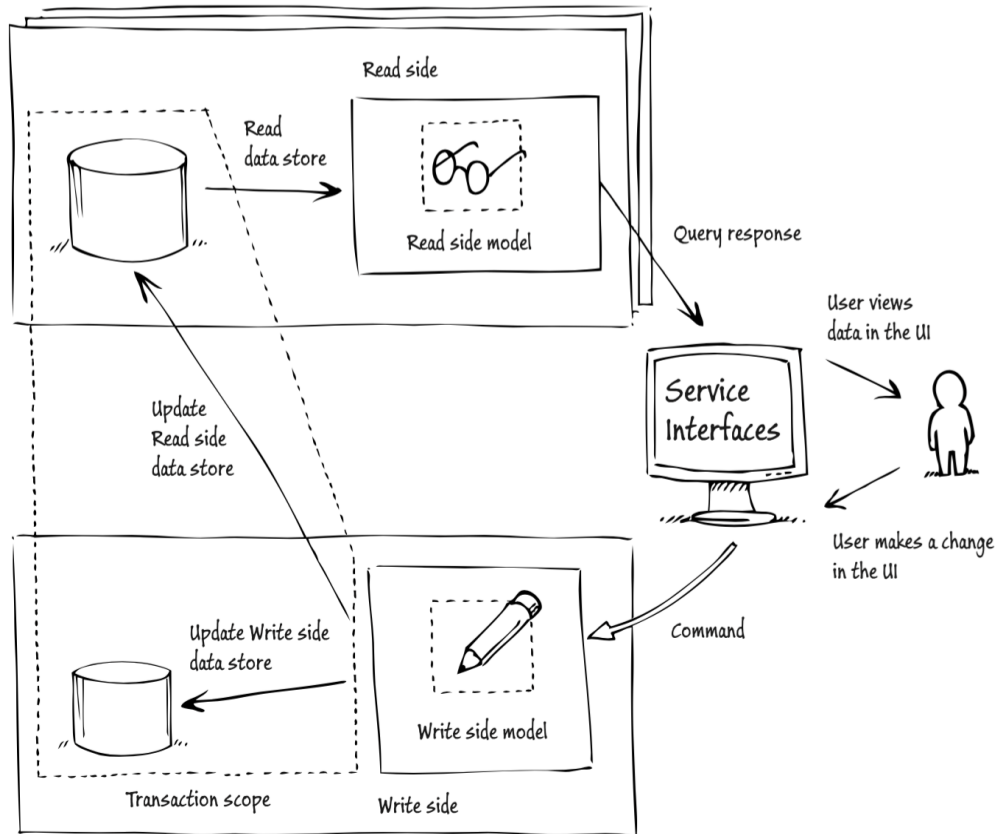


Figure 6. Eventual consistency and CQRS by Greg Young, MSDN, Microsoft [15]

CQRS stands for:

- Separation of models, meaning having different data structures and ADTs for reading and writing data.
- Command models meant for command processing, so called write side.
- Query models meant for data presentation, so called read side.

As with any other technology there are known disadvantages that engineers need either to accept or find workaround for. These are:

- Eventual consistency - the Event Sourcing pattern does neither propagate nor require usage of transactions for immutable data store, moreover with read and write sides being separated as shown in CQRS model.

- Increased complexity - the amount of work while designing a system using Event Sourcing architecture is huge and requires completely different approach and lots of supporting code to facilitate command and event handling.
- Duplicated messages - being message-driven architecture Event Sourcing does not guarantee that message won't be delivered multiple times in a row hence requiring engineers to implement deduplication logic or implement guarantee of at least once delivery solution.
- Events versioning - system may evolve over time, sometimes even changing dramatically hence there is a chance to experience backward compatibility issues and redundant or excessive events. As Event Sourcing enforces immutability from day one, one cannot simply remove excessive events and rebuild the application state, therefore requiring to implement logic to handle these old events [15].

Still, despite flaws described above author is confident that benefits heavily surpass those and the system will gain a lot from main benefit, which is handling high performance and streaming nature of crossroad traffic monitoring application.

Application domain itself is also a great fit for CQRS application as it can be represented with the amount of commands and events that shall mutate overall crossroad state and comply with CQRS read and writes sides. Thus, it will not help alone to design coherent, reliable and maintainable reactive systems as it is challenging task that throughout understanding of best practices available.

### **2.3 Reactive programming paradigms and system design**

Relying on extensive experience gained while developing CRUD-like applications, many engineers tend to think and design systems in imperative and stateful transactional way. Paradigms of Reactive programming require completely different mindset, forcing engineers to think in asynchronous and reactive way while working with data.

The Reactive Programming implies two fundamental paradigms – the data must be streamed in **asynchronous way** and be **immutable**. Once one has designed a system following reactive system principles it is natural to expect that system is resilient both for long failures as well as for the shorter ones. Hence always design for failure and never assume that components won't fail [16]. This Reactive System is supposed to

handle variety of failures such as short-term period of network traffic congestion, load on the storage node or loss of cluster member resulting in a loss of quorum. Reactive systems are generally better in terms of leveraging maintenance works hence there is a reduced risk of services unavailability. This essentially helps to develop and rapidly deliver any services updates to production having an option to rollback in case of any issue.

### **2.3.1 Reactive manifesto**

Taking a look at current developments in Software Engineering industry, one can spot a rising trend of reactive and asynchronous API-s developed in both enterprise, mobile or IoT fields. It is becoming more and more popular to design internal of mission critical software in reactive way, therefore it been quite natural to collect best practices and definitions what actually does coherent software mean. The Reactive Manifesto is an initiative lead by community and backed by initial authors of Reactive Streams specifications. Being a living document capable to evolve and change with the times, it has been the same for quite a long period of time and is ultimately supported by following four pillars [\[17\]](#):

- Being responsive - is about consistent responsive times, which is a pillar for great usability and utility from customers and integration perspectives. The system shall respond to requests in timely concise manner whenever it is possible delivering a reliable quality of service and making system eventually pleasant for end-user.
- Staying resilient - means that application shall embrace most failures and treat those as a common part of life cycle. Application should remain responsive in any case of severe failure and hence highly available for customers. Ideally, assuming that things might fail it is natural to implement a sort of self-healing machinery into the application itself, allowing either to recover from failure or isolate failing component.
- Rely on message-driven pass-through - means being foremost event-driven and passing messages asynchronously between components to achieve loosely coupling, isolation and location transparency. This is often called also as `ubiquitous` language with semantics that fits into distributed cluster behavior. Fundamentally, message-driven systems have great load management, elasticity and application flow control thanks to internal mailbox / queue like component

and by an ability to opt for back-pressure when needed. An asynchronous nature of such systems allows to consume incoming messages while being active and leads to great resources utilization without significant overheads.

- Retain being elastic - means being scalable on one hand by expanding according to system usage and utilizing resources as efficiently as possible but on the other hand it is crucial to stay responsive under load. Essentially it implies that system can and must scale without locks, contention points and bottleneck by sharding or replicating its internal components and distributing varying workload between those. Having that said, being elastic means also being cost-effective in the context of software and hardware utilization.



Figure 7. The Reactive Manifesto. <https://www.reactivemanifesto.org/> [17]

## 2.4 Actor model

Actors model has been available around for engineering since seventies and has been invented by Carl Hewitt [18] while being first of all successfully implemented in Erlang programming language. Foremost, an initial idea behind actor model was to provide a way how to handle efficiently parallel processing in a high-performance network in an

environment that might not be available all the time. A single actor can be described according to Carl Hewitt as a “fundamental unit of computation embodying processing, storage and communications” where “everything is an actor” and “one actor is no actor, they come in systems” [18]. Essentially, actors define loosely coupled senders and receivers built naturally for asynchronous communication.

However, current development of hardware, infrastructure and engineering competence has exceeded state of systems that used to be in place in seventies when actors first introduced, there are still challenges that cannot be solved now with frequently used (OOP) object-oriented programming approach and for sure cannot gain benefits from the actor model. Furthermore, actors-based systems being run in current modern multi-threaded and multi-CPU environments significantly outperform their counterparts and are recognized as highly efficient solutions for demanding architectures.

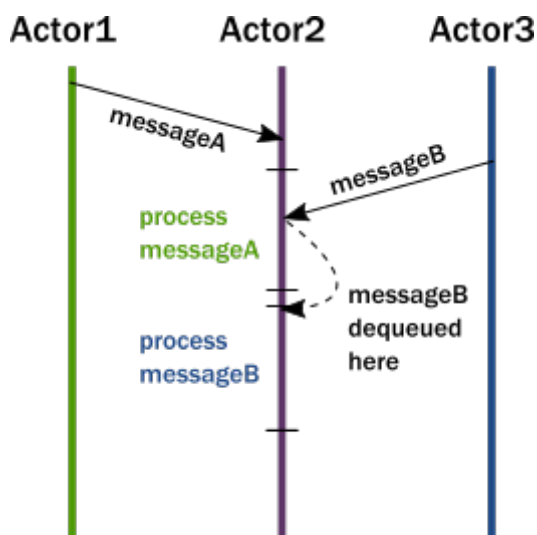


Figure 8. Actors communication with each other [19]

Unfortunately, Actors model is considered to be very low-level in terms of implementation and therefore comes with few caveats that one shall be aware of. These caveats are:

- High risk of running into out of memory issues and loss of critical data where producer's event stream is too fast for consumer, exceeding its capabilities. A solution is to implement back-pressure.
- Weak Type safety - actors are supposed to handle `Any` message and require lots of testing and caution during implementation.

- Code complexity - actors are low-level units and it becomes rather hard to maintain their internal state and mutations requiring a lot of effort and boilerplate code for debugging.
- Rather high learning curve and increased risk of mistakes in production systems due to lack of experience.

For what it is worth, the Actor model and intercommunication between actors reminds the interaction that between humans and the way we behave when we talk with each other, either we are asking or telling some valuable information, which is exactly corresponds to actors *ask* and *tell* patterns:

```
import akka.pattern.{ask, pipe}
import system.dispatcher

implicit val timeout = Timeout(5 seconds)

// tell pattern
// ! means "fire-and-forget", e.g. send a message asynchronously and return immediately
actor ! Message

// ask pattern
// ? sends a message asynchronously and returns a Future representing a possible reply
val reply: Future[Reply] = (actor ? Request).mapTo[Reply]
```

Figure 9. Example of Ask and Tell patterns in Actors

## 2.5 Reactive streams

A fundamental part of reactive world is a stream - often unbounded flow of events and values. An abstract stream does not enforce any strict rules neither on the data that is being emitted nor on the specifics of upstream and downstream flow, such as a number of subscribers for instance.

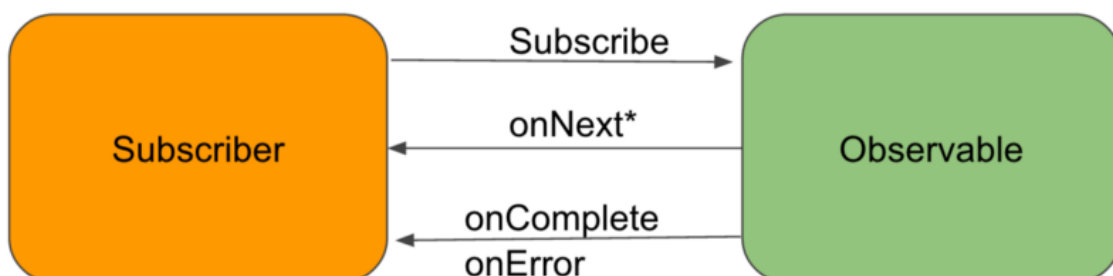


Figure 10. Publisher-subscriber like communication for Subscriber-Observable [\[20\]](#)

A regular stream implementation must have following methods defined in order to comply with generic stream definition:

- *onNext* - this shall transfer further every single item emitted by stream
- *onComplete* - this shall notify of an end of the stream, meaning that no further *onNext* methods will be called.
- *onError* - this shall propagate an exception further to the observer that might have happened in the stream.

The reactive streams itself cannot handle all the use-cases engineers might come up with hence there are couple of useful streams that complement default stream implementation.

- Flowable - with back-pressure
- Observable - no back-pressure

Remarkable advantage of Reactive Streams is a variety of building blocks consisting of streams themselves and different handy stream operators like map, flatMap, filter etc. that should satisfy any developer needs that may appear. In a few cases, I am strongly convinced that Reactive Streams could have been called even a Software 2.0 paradigm meaning a fundamental shift from OOP development model to a reactive-functional one [\[20\]](#). Comparing to classical software stack most of us are familiar with, Reactive Streams allows engineer to grasp streams and their operators and eventually glue together, requiring a minimal amount of code to be written for various side-effects.

A traditional OOP-driven development would have required developer to describe most of behavior and instructions manually and perform tons of performance and acceptance testing to prove the implementation being correct. In a contrast with actors, Reactive Streams do not require to develop that much of low-level machinery to support concurrency and back pressure from upstream to downstream making streams much more verbose and explicit from readability perspective. Regretfully, there are own disadvantages that one has to live with. For instance, although the high-level DSL that streams have is pretty verbose, it might take some time in order to find a bug in the code and sometimes streams might fail in cumbersome and non-intuitive way, or worse even just silently fail. Still, let's have a look at crossroad traffic monitoring being solved in a reactive way.



### **3 Overview of the proposed solution**

A typical crossroad is an essential city transport artery, connecting multiple roads and intersection and remaining useful for both vehicles and pedestrians. A modern crossroad can be compared to a human heart, which is full of processing pipelines and arteries (roads) that have to be reliable and efficient. Hence comparing crossroad monitoring reliability to human's heart, we would expect it not to work at fixed timing, not to suffer from traffic congestions and reduce risk of mistakes and faulty decisions in case of any. Having in mind that the amount of vehicles traffic is rather growing by the order of magnitude, crossroads have to adapt with increasing traffic flow and its fluctuation in real-time or near real-time manner under all possible scenarios.

#### **3.1 Crossroad traffic operation problems**

Traffic congestion is the main issue that is affecting highways, inner roads and what is most important crossroad as well. Current crossroad traffic management solutions are most fixed timing based and perhaps are adjusted only according to historical data of the traffic flow in one or another direction. Apparently, distributing crossroad traffic at fixed timing isn't scalable both in short-term and long-term perspective. Firstly, consider holidays and vacation period of time, when there is significantly lower amount of traffic due to parents and their children being away on holiday. Secondly, one more major issue that cannot be solved with fixed timing are traffic jams at the crossroad intersections due to cars being lined up along the lane in one direction and much less in other lanes. Finally, it is worth to mention quite likely occurrence of traffic accidents at the crossroad intersections caused by drivers ignoring blinking yellow or red light or just choosing a wrong speed to cross the crossroad.

Fortunately, thanks to latest development in the ICT (Information and Communication Technologies) field we have wide range both of hardware and software solutions combined that can be applied to improve current state of crossroad traffic monitoring

and eventually address most of the issues described above making our cities crossroads environment friendly and responsive.

Given the wide range of various IoT devices that have been tightly integrated into people every day's life helping to automate daily routine tasks and taking care of home automation, it is quite natural to blend IoT into the crossroad monitoring as well.

These days an intelligent crossroad monitoring built from scratch would definitely benefit from following hardware:

<b>Device type</b>	<b>Applied usage</b>	<b>Expected Outcome</b>
Video camera	Real-time image processing, lane queue analysis, accidents analysis	Vehicle count in the lanes, density monitoring
Sensors	Number of vehicles crossing intersection per / min, average speed of vehicles crossing	Total number of vehicles crossing, traffic density and average speed, real-time telemetry
Built-in road surface weight sensors	Vehicle types distribution (regular car, bus, truck)	Density of heavy-weight vehicles that may cause traffic delays and increase road wear
Weather sensors (external) and built-in surface sensors	Road temperature and upper layer condition analysis	Detailed report of road temperature which is highly vital during winter time

Table 1. Proposed hardware list for intelligent traffic monitoring

Usage of any of above mentioned hardware can make crossroad monitoring much more smart and intelligent though it is not that useful yet without proper software that can receive, process and make immediate decisions based on that data to make the monitoring system work and be successfully applied. Especially complicated is the fact that data and all available telemetry is being streamed from different sources hence requiring system to remain reliable even in case of any of sources is temporary suspended. Hence it is extremely important to implement reliable and performant software to handle loads of workload produced by IoT devices.

For instance, it would be essential to consider at least following emerging software technologies:

<b>Technology</b>	<b>Applied usage</b>	<b>Expected outcome</b>
Artificial Intelligence, Machine Learning, Deep Learning (YOLO object detection pipeline based on neural network <a href="#">[21]</a> )	Image or video stream real-time object detection with classifiers (car, bus, truck, bike)	Evaluation of real-time motion happening at crossroad with expected classifiers
Time-series database	Collection of telemetry streamed from crossroad sensors, various metrics	Telemetry raw data persisted in time-series db for further analysis and fast query
Real-time analytics and persisted data analytics	Decision-making evaluation, immediate feedback proposal, traffic accidents detection	Available in back-office for both manual and automatic decision making regarding crossroad state

Table 2. Proposed software technologies used for intelligent traffic monitoring

Additional prerequisites for expected crossroad traffic monitoring system would be an option to handle risk of traffic accidents and emergency situations which is a rather edge case for any monitoring system highly likely such behavior cannot be built up based on any amount of historical data. In many cases the easiest way would be to require manual intervention and hand-over of control to back-office operator to reduce risks of automatic control faults however there might however situation may change with further development of Neural Networks, especially convolutional neural networks [\[21\]](#). Overall, there is one more interesting proposal unfortunately left out of scope from this paper, which expects shipping vehicles with RFID tags installed. This would essentially allow recognize certain types of vehicles such as mission critical ones – ambulances, police and fire-extinguishers and help to switch control system to exceptional state that would guarantee pass-through crossroad within minimal amount of time. Unfortunately, such proposal requires effort both from car manufacturing industry and sensors suppliers and is comparably much more expensive that deployment of recognition-like software.

This paper mostly considers software part of smart crossroad monitoring system implementation and the hardware part is intentionally omitted here, assuming there is an agreed contract of time-series metrics being streamed to API's and additionally there is trained recognition model and pipeline for object detection in the video stream that is provided as is. Eventually, the proposed setup is described below.

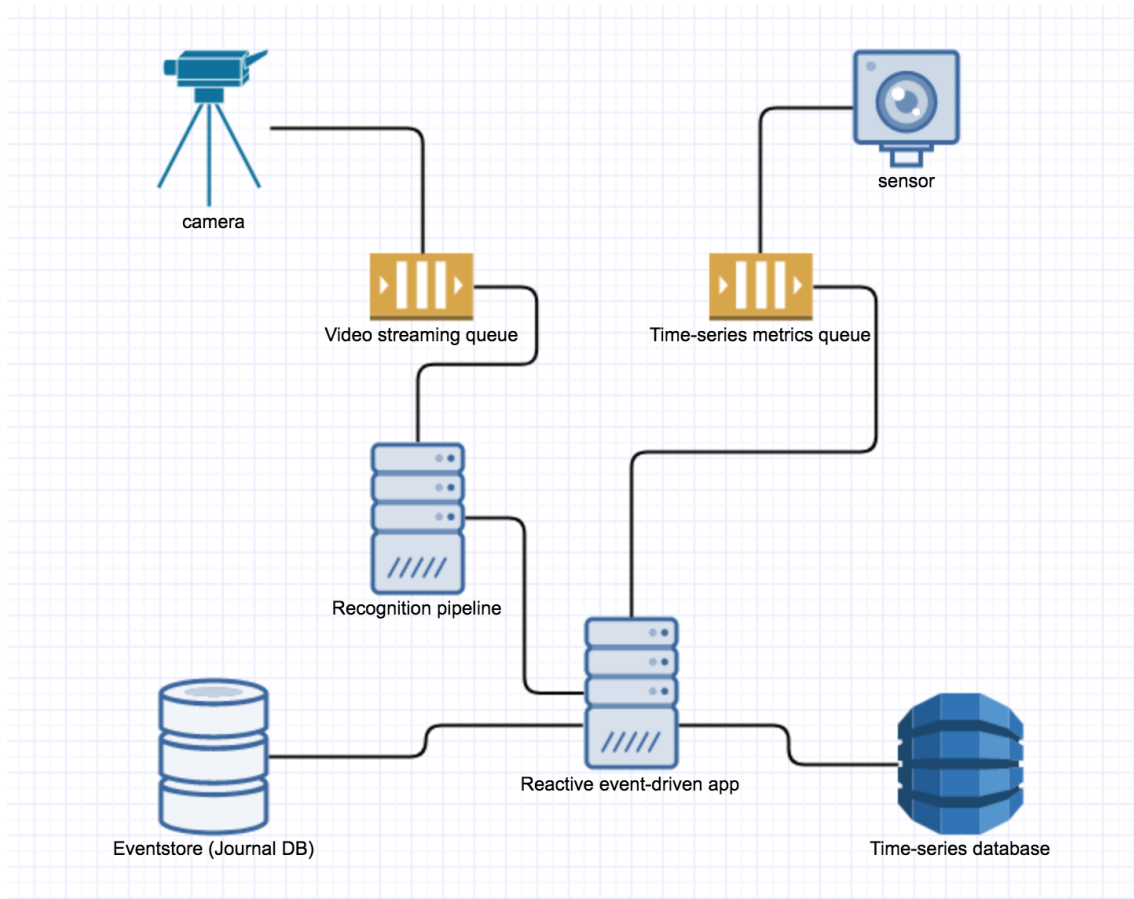


Figure 11. A Bird's Eye View to proposed IoT setup

Author would like to opt for the simplest possible solution that has both efficiency and lowest possible overhead. This does not pretend to be the only valid setup though but ideally it should minimal number of components needed to process both video stream, objection detection and time-series metrics from various sensors.

### 3.2 Proposed reactive solution

Given the crossroad traffic problems described above, in this Thesis author would like to try to propose following architecture inspired by Event Sourcing approach combined with CQRS and DDD core features and ideas.

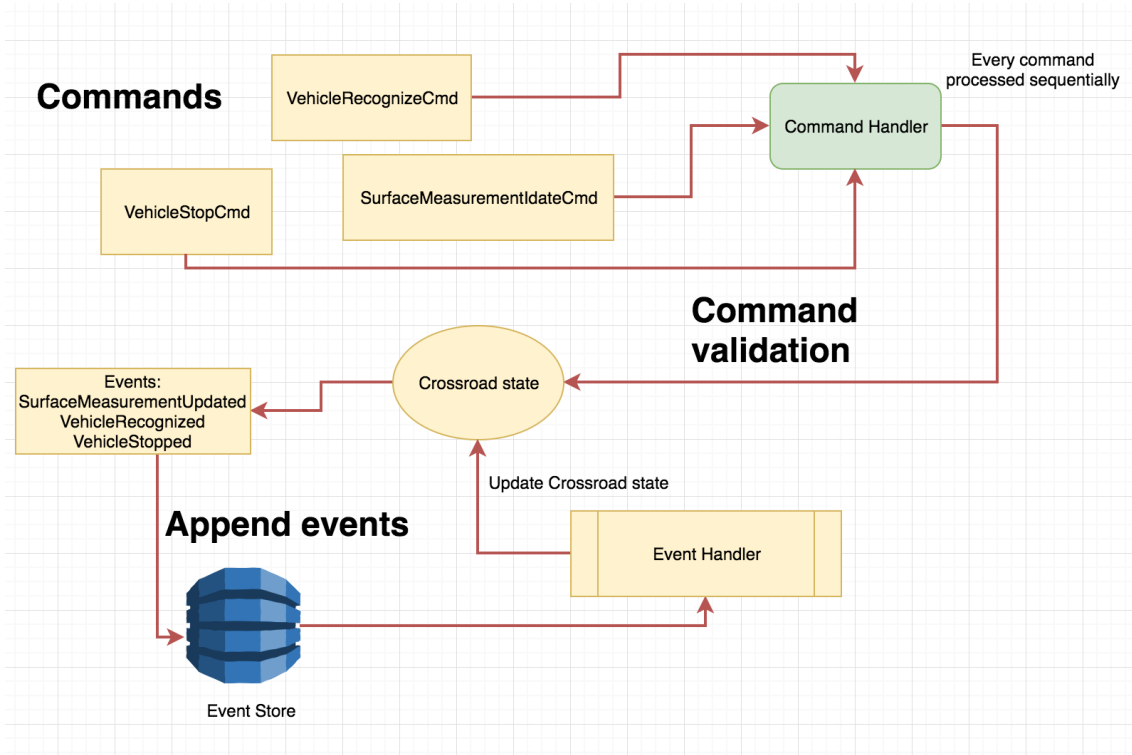


Figure 12. Proposed implementation based on Event Sourcing, CQRS and DDD

The domain model is intentionally simplified in the scope of Proof of Concept implementation during this Thesis work but should give a comprehensive view of the core ideas behind Domain Driven Design [22]. Therefore, Appendix 5 contains implemented commands that correspond to all the actions and requests coming from Crossroad IoT devices, whether this is video recognition pipeline or sensors streamed data. Every command defined in Appendix 5 represents an intention that has to be validated against current crossroad state and in case of success should produce single or multiple events depending on the expected result of command.

Appendix 6 contains system supported events that define are applicable to crossroad state for further state mutation as well as streamed to third party consumers if any. These events should carry all necessary payload to rebuild state from scratch or subscribe to updated regarding particular crossroad, the latter one could be a good task for Dashboard monitoring UI.

The implementation phase follows the proposed architecture and is described in the 4.3 - Software product implementation, which is deliberately split into two parts - Actors based implementation and Reactive streams-based implementation. The main goal is to provide two different implementations that follow Reactive programming paradigms and Reactive system prerequisites in a timely and efficient manner. To be more specific, chosen implementations should prove that application will be:

- Responsive - handle tons of requests and streamed data by IoT devices
- Resilient - remain available in case of failures, rely on persistence backed by Event Store
- Elastic - utilize given bounded system resources in an efficient manner
- Message-driven - communicate in a form of commands and events, utilize pub-sub principles in a distributed and decentralized environment like modern cloud / microservices architecture is

which should eventually result in a high-performing and fault-tolerant software for IoT based systems [\[24\]](#).

## 4 Implemented solution

Despite a range of available frameworks, programming languages and best practices, there is no carved in stone technology-wise choice that would fulfill all the requirements. This particular Thesis is an attempt to play around different available approaches described above and determine which one of them is a good fit to achieve coherent, reliable and fault-tolerant system for the crossroad traffic monitoring based on video streams and available sensors. During the work on this Thesis, different implementation stages have been proposed and analyzed but author has decided to proceed with Actors and Reactive streams-based implementation and eventually craft a dashboard for real-time monitoring as well.

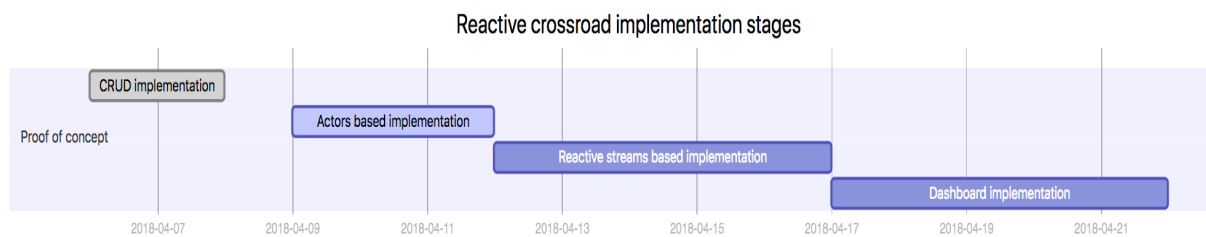


Figure 13. Reactive Crossroad implementation stages

### 4.1 Crossroad IoT devices

In this Thesis scope the actual implementation and setup has been intentionally omitted in favor of readymade solutions that are available on market. In addition, main focus of this work is to provide justify the reactive way of implementing software side to support the vast amount of data that is being collected by IoT devices through various protocols and APIs and prove the proposed architecture to be reliable and efficient. Though, author finds it useful to advice what could be the possible setup for IoT devices for given problem statement using affordable devices available on the market and also describe briefly their features, protocols and inputs / outputs.

<b>Device</b>	<b>Features</b>	<b>Protocol</b>
CCD camera, one of Imprex products	Wide range of operating temperatures, High resolution and low noise	CameraLink interface, high speed interface for real-time video
Raspberry PI 3 <a href="#">[27]</a>	64bit quad-core processor, Bluetooth protocol support, wireless LAN, core of local crossroad monitoring setup	Wide range of protocols for I/O
Surface sensors	Road condition (dry, wet, ice, snow etc.), water film height, relative humidity, surface temperature	Bluetooth or CAN-Bus if connected to Raspberry PI module. MQTT if direct pub-sub streaming to server

Table 3. Proposed IoT devices for intelligent crossroad

The proposed list of hardware should be sufficient for Proof of Concept stage of intelligent crossroad traffic monitoring system, considering there is a wide range of inputs and outputs that can be used by the controlling software. Given this setup, the main role of collecting and transforming the data is assigned to Raspberry PI. Namely, Raspberry PI software should do the following:

- Collect raw metrics data from installed sensors and transform those to human readable metrics format, which will be proposed later. The sensors raw data should be captured through Bluetooth and / or CAN-Bus
- Capture and process video / images stream in order to extract significant events from provided video such as vehicle collisions or incidents, lane queues and slowdowns as well as vehicle count / pass through crossroad.

Transforming captured raw data to human and / or machine-readable format requires not only non-trivial resources but also a custom optimized software, that could do the work considering bounded resources constraint enforced by Raspberry PI. Hence, author would like to expand these topics and describe what kind of available software and algorithms can be used for gathering time-series metrics and performing video recognition from stream.



## 4.2 Vehicle recognition from video stream

The suggested proof of concept setup of live video stream from IoT devices consists of following options:

- Live stream is from IoT devices is streamed via RTSP (Real Time Streaming Protocol) network protocol mostly for debugging purposes, having a backup options to persist video streams to local disk for next 72h in case there is a need to playback those or rewind to some particular moment
- In addition, live stream data is processed for visual object detection locally, to be more specific machine learning algorithms are involved to categorize certain details extracted from captured data and classify those further according to level of interest. Further, the sensor data that has passed ML stage is transmitted to the cloud subsequently fast and efficiently. In a long-term future it should be possible to rely only on extracted data from sensors once ML algorithms are improved, leaving the raw video available on-demand only from IoT

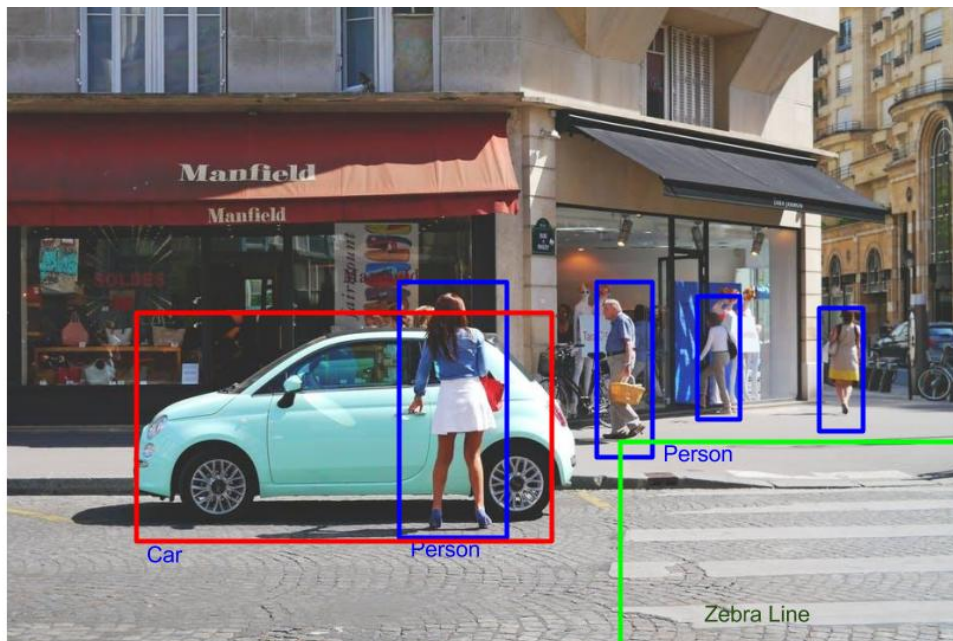


Figure 14. Visual Object Detection by YOLO algorithm [28]

When it is mostly no overhead or additional implementation required to provide live stream via RTSP protocol further to crossroad controlling system, though there is a significant amount of work that is required to perform video detection using Visual Object Detection [28] techniques in order to recognize, identify, localize and classify

objects detected in the video stream. To be more specific, an abstract video recognition pipeline has to perform following tasks.

- Classification - is an attempt to analyze an image and predict the object in this image.
- Localization - bounded location of a given classified object within an image
- Object Detection - combined location of an object plus classified object itself
- Image segmentation - precise location of a classified object in a segment

The following tasks simulate the human way to recognize objects from images and these four tasks are grassroots of Visual Object Detection model called YOLO (*You Only Look Once*) [21], which has great speed, accuracy and recognition speed being also very popular and simple for end-user. The main constraint here is the Raspberry PI limited resources as the intention is to perform recognition locally while streaming only detected and classified events and object further to the cloud hence the expectation is to get YOLO model running on the in the resources bounded IoT devices.

Author would like to outline that implementation of Video Object Detection pipeline wasn't considered as a goal of this Thesis work though author has made a background research and identified that YOLO neural network algorithm is a perfect match for robust and high-performing recognition pipeline in resources constrained environments, in addition being accurate enough as well. Thus, author assumes that all the video recognition related work is done by one of YOLO implementation, for instance this one published in GitHub under GPL V3.0 license [36] by Junsheng Fu [37].

Author assumes that any video recognition pipeline should be comprehensive enough to provide event-driven stream of raw data which is eventually streamed through near real-time API to server.

### **4.3 Time-series metrics from IoT devices**

Being low-level devices, most sensors provide only proprietary APIs which are not suitable for proposed IoT devices setup. Fortunately, there is a wide range of sensors available on the market that either support MQTT (Message Queue Telemetry Transport) protocol which has been standardized a while ago or can be connected to Raspberry PI module through Bluetooth / Can-Bus. The latter one will allow to use any

TCP based transport such as HTTP (Hypertext Transport Protocol) or preferably WebSocket to meet near real-time communication requirements. In addition, it is essential to pack metrics into compact payload such as ProtoBuf or JSON to reduce the amount of data transmitted through network and increase the throughput as well. Author has chosen to define payload for three types of metrics that are being collected by IoT devices and streamed to server. Firstly, surface measurements are provided by built-in road sensors and proposed payload is described in Appendix 3 with conforming implementation of domain entity. Secondly, weight measurements are provided by built-in road sensors and proposed payload is described in Appendix 2 with conforming implementation of domain entity. Last but not least, there are weather related sensors installed both in road surface and externally, proposed payload is described in Appendix 4 with conforming implementation of domain entity.

#### **4.4 Software product implementation**

Martin Fowler [\[29\]](#) has made crisp definition of software architecture – “Software architecture is those decision which are both important and hard to change. The importance of software architecture impacts either success or failure, the latter one might be unnecessarily expensive”. Following Martin Fowler [\[29\]](#) proposal, the software product architecture should be designed keeping in mind fundamental requirements that are being put by IoT based intelligent traffic monitoring and Reactive System definition hence proposed implementation should satisfy following needs:

- Be resilient
- Be responsive
- Be fault-tolerant
- Be message-driven

##### **4.4.1 Platform**

Reactive platform implemented according to Event Sourcing and CQRS patterns has the following components:

- Event Store - the primary and only single source of truth that has been praised for immutability and performance. Author has chosen to rely on Open-source and functional database EventStore [\[30\]](#)

- Time-series data - is streamed to open-source time-series database InfluxDB which is coupled together with great open-source analytical platform - Grafana [\[31\]](#)
- Data ingestion / Streaming - IoT metrics and data are streamed through WebSocket to server in JSON payload

#### **4.4.2 Programming language**

Primary language used to implement the solution defined in this thesis work is Scala [\[55\]](#), a hybrid object-functional programming language. Scala is strongly typed, immutable-first and JVM [\[56\]](#) based language which perfectly fits into coherent distributed applications architecture. Under the hood of reactive crossroad implementation there is a lot of concurrent connection handling and amount of data, being streamed from producers to consumers, which is akin to most Internet of Things solutions. Furthermore, it is rather crucial to be able to handle with real-time results within relatively low latencies and that is exactly the case where vast majority of frameworks and solutions built on top of Scala might outperform other JVM counterparts and definitely all other event-loop single-threaded languages like JavaScript etc.

#### **4.4.3 Frameworks**

The choice of frameworks is implied by the chosen Scala programming language. Obviously, it is clear that just relying on the core Scala language features it would have taken ages to get this solution ready and therefore it is a good idea to choose suitable frameworks to adhere requirements of the reactive crossroad and constraints we have set in the problem statement. A modern yet reliable choice to build a resilient, reliable and message driven system is Akka. Akka is a comprehensive toolkit for building distributed and resilient message driven applications on JVM. The framework initially served a goal to implement the Actor Model on top of the JVM and eventually became a `de-facto` option for elastic and decentralized applications. Being resilient by the design, Akka allows us to build such systems that are able to self-heal and recover from failure, while still being responsive in the meantime [\[24\]](#). It is a vital property of reactive systems that is considered to be essential for the IoT solution of reactive crossroad monitoring. The IoT domain which is the scope of this Thesis work is a great use case for Akka framework in general and Actors system in particular thanks to built-

in solutions like routing, sharding, pub-sub and cluster support. Having all that said, Akka remains very high performing even on a single machine offering an asynchronous non-blocking stream processing with back pressure out of the box.

In order to support statements proving that Akka is the best toolkit for building highly concurrent, distributed and resilient applications, the Akka team has conducted performance testing claiming that it can handle up to 50 million msg /sec on a single machine. Besides great performance, Akka does not require any significant amount of memory for operation as Actors are very lightweight units and it is rather inexpensive to create actors in the application. Akka team claims that one can have up to 2.5 million actors in the system for just 1GB of heap memory making Akka far beyond other frameworks in terms of efficiency and performance [32]. Therefore, first of all author would like to start with Actors based implementation.

#### **4.4.4 Actor based implementation**

In this Thesis author has used two types of Actors provided by the Akka framework - common stateful Actors [33], which have state persisted in-memory tied to life cycle of JVM and persistent Actors [33] capable to persist internal state to external journal and eventually recover once JVM has been restarted. Following Akka architecture principles, every crossroad needs to be an actor with own unique persistence id allowing to maintain its own internal state that can be changed only by persisted event. The simplified version of persistent actor is available in the Appendix 7 - CrossroadPersistenActor.scala [38]. The main workload however is done by the worker actors, described in the Appendix 8. These small units are initial entry points for every type of requests that are received by application and in order to scale these better are distinguished by type of requests - a separate worker for video recognition results - Appendix 8 and another one for time-series metrics - Appendix 9.

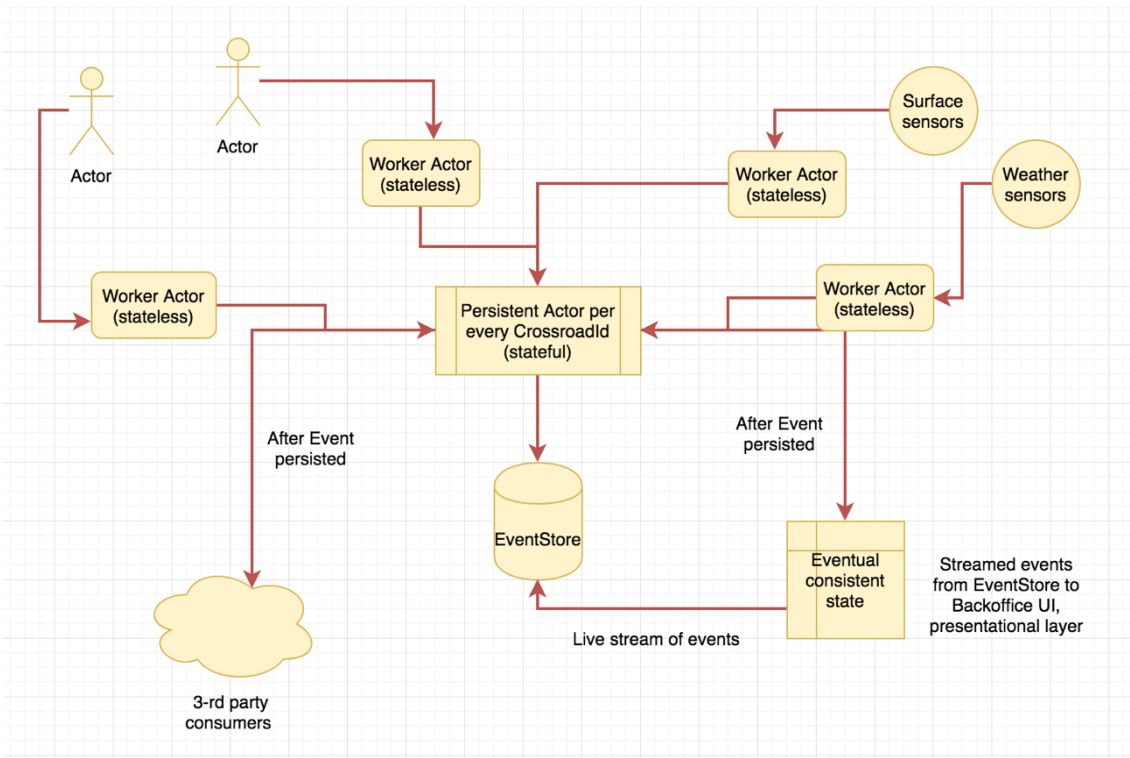


Figure 15. Actors-based implementation design

Given the goals defined in a Problem statement that need to be solved with Reactive solution, an actors-based implementation does address most of the given prerequisites. To be more specific, a **fault-tolerant** requirement is fulfilled by usage of persistent actor implementation. **Responsiveness** and **elasticity** is backed by worker actors that keep receiving streamed data and basically do single-responsibility tasks such as either persisting metrics or transforming raw request data to persistent actor commands hence these worker actors serve as gatekeepers to offload the persistent actors. Finally, message-driven throughput is supported by the intrinsic property of actors to communicate via publisher-subscriber pattern and publish any side-effects requiring actions through the EventStream. [25, 26]

Having implemented first solution, author finds that despite application being able to match expected reactive system requirements there are few crucial flaws that may lead to unexpected issues in production. These are unpredictable resources handling in case of increased load from IoT devices, which will result in application running out of memory due to missing back-pressure in actors and also there is significant risk of sending unhandled commands and requests to actors due to their weak type-safety. Author also finds that a requirement to implement that much of verbose and low-level

code for getting Actors implementation done is a serious and time-consuming issue that needs to be address.

#### 4.4.5 Reactive streams-based implementation

Luckily, the proposed Reactive Streams implementation should be able to solve most (if not even all) the above-mentioned issues. Following is proposed design of streams based implementation, which when compared to has retained crucial components like Event Store [30] and Persistent Actor [33] per every crossroad id.

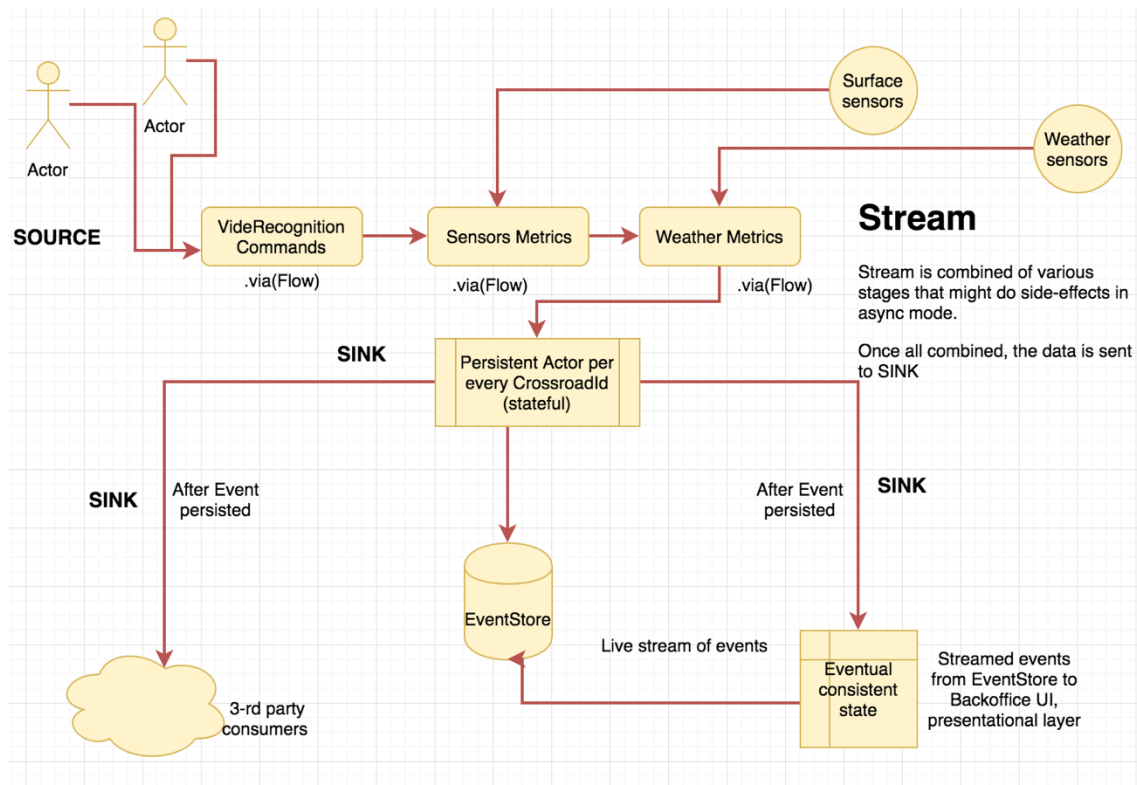


Figure 16. Akka-streams based implementation design

Due to author's choice to use Akka streams implementation of reactive streams, which is slightly different from baseline implementation, there is a need to explain what Akka streams equivalents for such streams core concepts like Observable [34] and Operators [35]. **Source** is the Akka streams way of consuming streamed data with exactly one output. **Flow** is the Akka streams way to allow data to flow through a function, possible map-reduce function or any other equivalent. Flow has exactly one input and one output, making it possible to apply transformations. **Sink** is the final stage of a stream in terms of Akka streams terminology, it might do any I/O operation and complete the stream [26]. Dealing with flaws introduced in the actors-based implementation, the first top priority thing to solve is the back-pressure issue of an IoT streaming application that

might lead the whole system to crash and fail expectations to meet reliable and resilient system. Schematically, the implementation of a stream with back-pressure is shown on figure:

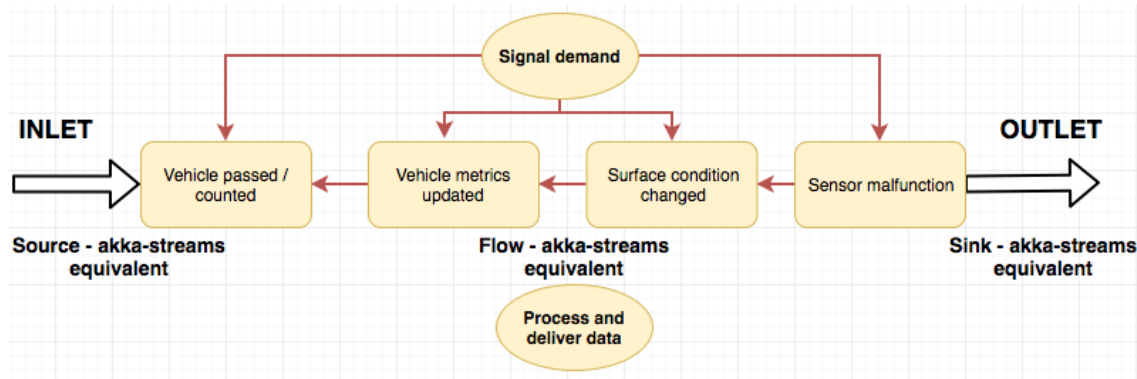


Figure 17. Akka streams-based back-pressure explained

Author has implemented Video recognition results handling with Akka streams in the Appendix 10, which is basically one single class written in Scala which does the following [38]:

- Subscribes indefinitely to incoming requests from video stream recognition pipeline (note `def videoRecognitionPipelineStarted` function) which is basically a single Source of data. This source is also limited to 100 elements in buffer and once this threshold is reached, back-pressure will be applied.`
- Performs time-series metrics persistence per every incoming request (note usage of `val persistMetrics`)` via **Flow** that does not apply any transformations yet.
- Transforms every incoming request received into valid persistent Actor command (note usage of `val transformToCommands`)` via **Flow**
- Dispatches transformed collection of commands to persistent Actor asynchronously (note usage of `val dispatchCommands`)` via **Sink**
- In case of non-fatal failure during stream processing such as sensor malfunction, Reactive Streams offer great control over recovery procedures known as Streams Error Handling [54]. Streams offer wide range of recovery options and Akka will take care of recovery, restart with retry attempt and resume with back off hence supporting Reactive Manifesto core principle like being **fault-tolerant**.

Thanks to this implementation and Akka streams framework, author was able to solve processing of IoT data in a reactive way with minimal amount of code in a timely and



an efficient manner comparing to Actors based implementation. Moreover, a flow-control or back-pressure comes literally for free without any additional low-level machinery. Finally, it is very convenient and easy to process all the received data applying transformations and performing non-blocking side-effects which will not hinder progress of the application [24].

#### 4.4.5 Recorded system performance metrics

Author has come up with following simulation setup for both implementation that was later conducted and all the JVM (Java Virtual Machine) performance metrics were recorded. The test environment was configured on a regular MacBook Pro Mid 2015 laptop with following JVM version and settings.

```
~ > java -version
java version "1.8.0_172"
Java(TM) SE Runtime Environment (build 1.8.0_172-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.172-b11, mixed mode)

~ > echo $JAVA_OPTS
-Xms256m -Xmx2048m
```

Figure 18. JVM version

Having the wide range of data sources that could be streamed to servers from IoT devices, author has decided the scope of simulation with just a reasonable amount of payload with video recognition results that could be turned to commands and events in CQRS context, described in Appendix 1 provided Gitlab repository. The simulation scenario contained of following steps:

- Define maximum number of 200 000 unique crossroads for simulation purposes
- Create every crossroad by sending ‘C.Create’ command to every crossroad with default configuration of sensors, lanes etc.
- Start sending CQRS compliant commands and produce events for persistence in Event Store [30]
- Collect metrics in Graphite [53], especially JVM metrics for further analysis.

A sample report is added below with recorded threads activity for Reactive Streams based application.

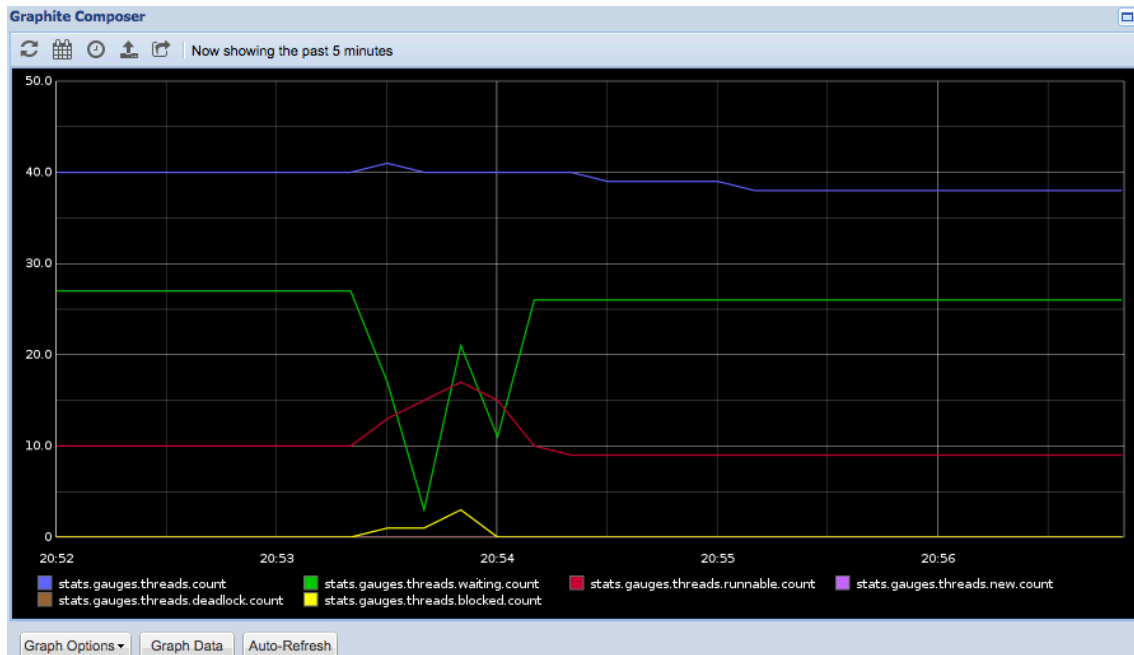


Figure 19. Threads utilization for Reactive Streams application

The following graph above has **Y-axis** with thread counts numbers for running, blocked and news threads in the context of JVM where application is being run now. One can note that thread-pool size is capped at 40 threads. The **X-axis** has just a timeframe, when particular measurements were collected, total timeframe in the scope is just 5 minutes. One can notice the spike right just after fall on the graph which is a consequence of Java GC (*Garbage Collector*) work. Author has attached a comparison for both implementation in the Appendix 11. Additionally, Appendix 12 has comparison of memory utilization for both applications where-as Appendix 13 contains comparison of GC stress.

## 4.5 Analysis of results

Given the results of the performed simulation, Actors based implementation has proven throughout the application and simulation lifecycle that actors are very efficient for managing and encapsulating mutable state of single crossroad, arranging fault-tolerance and distributing workload when it comes to a cluster setup. Unfortunately, designing actors-based application required essentially to become familiar with low-level API Akka actors are built on top of and hence much more effort from the engineering perspective. Moreover, actors do not provide back-pressure support out of the box making it possible to run out of allocated resources in resources constrained environments.

Reactive streams implementation based on Akka streams turned out to be great and much more convenient from engineering perspective. To be more specific, reactive streams API is built with high-level semantics and flexibility in mind comparing to actors' API. In addition, streams are utilizing bounded system resources constraints much more carefully and have built-in asynchronous back pressure out of the box.

The overall performance of Reactive streams-based application has impressed author due to its smooth and predictable resource usages, which results in **less CPU context switches, reduced memory heap usage and less stress to Java Garbage Collector**.

Comparing all the effort required to craft both implementations and pros and cons analyzed above, one can treat reactive streams approach as scalable, reliable and robust toolkit that fits extremely well into modern powerful and distributed systems. [[25](#), [26](#)]

## 5 Summary

The goal of this thesis was to investigate whether Reactive programming is applicable to the crossroad traffic monitoring solution based on IoT devices both that imposes system resources constraints and offers challenges such as handling tons of data and hundreds of events streams in durable and distributed environments. Whether all this done in Reactive way would have performed better and efficiently with a fault-tolerant behavior than traditional pattern of developing systems using CRUD / OLTP models characterized by large number of short-living transactions and backed up by relational ACID-compliant databases. Author has claimed that common methods and practices described above come with a peculiar trade-off such as either having a relatively fast processing speed and suffering from data integrity or having a strong consistency and data normalization while operating at very low processing speed.

As a result, author has designed the crossroad monitoring solution in a proof of concept, applying Reactive programming. This has required to define all domain entities in DDD (*Domain Driven Design*) form, thus also defining all the commands and events that need to support CQRS pattern and message-driven nature of application. Following has allowed to support core Reactive manifesto principle as to build message and event-driven system. Further, the most significant Reactive programming paradigms such as being reliable and resilient was proven by the implementation of two different solutions – one Actors based and another one Reactive streams based on top of Akka [\[32\]](#) foundation. Additionally, author has justified precedence of Reactive streams based implementation in terms of system resource utilization, lack of low-level machinery and infrastructural code and benefits of elasticity that was provided out of the box, such as back-pressure and parallelism hence it is valid to claim that Reactive streams were found to be a great fit and outperformed competitors.

It is worth to mention that Reactive Streams learning curve exists and requires some time to get it up and running. It has also taken some time to understand what actually happens under the hood, required to learn particular building blocks and DSL and

eventually author had to analyze few culprits in the implementation in the application runtime when it started to fail silently.

Nevertheless, opting for a reactive system built either on top of either actors or reactive streams or actors and streams combined has been confirmed to be more natural choice for crossroad traffic monitoring. Whenever there is a requirement to build a coherent, consistent and highly available system one should consider actors and reactive streams before making an eventual decision. Author is confident that the modern crossroad real-time monitoring system combined with Reactive Programming model is a key to success in reducing traffic congestion, increased reliability and reduced fuel consumption.

## References

- [1] Intel.com, “A guide to the Internet of Things”, 2017, [Online], Available: <https://www.intel.com/content/www/us/en/internet-of-things/infographics/guide-to-iot.html>
- [2] Microsoft.com, “Online Transaction Processing (OLTP) – Technical Reference Guide”, [Online]. Available: [https://technet.microsoft.com/en-us/library/hh393556\(v=sql.110\).aspx](https://technet.microsoft.com/en-us/library/hh393556(v=sql.110).aspx)
- [3] MSDN Microsoft, “Cutting edge – Rewrite a CRUD system with events and CQRS”. 2016. [Online]. Available: <https://msdn.microsoft.com/en-us/magazine/mt790196.aspx>
- [4] Microsoft.com, “Creating a simple data-driven CRUD microservice”. 2017. [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/multi-container-microservice-net-applications/data-driven-crud-microservice>
- [5] G.E. Moore, „Moore’s Law”, 1970. [Online]. Available: <http://www.moorelaw.org/>
- [6] Reactive-streams.org, “Reactive Streams”, [Online]. Available: <http://www.reactive-streams.org/>
- [7] Reactivex.io, “ReactiveX”, [Online]. Available: <http://reactivex.io/>
- [8] Akka.io, “Akka streams”, [Online]. Available: <https://doc.akka.io/docs/akka/2.5/stream/index.html>
- [9] Greg Young, “CQRS Advisors Mail List”, 2014. [Online]. Available: <https://msdn.microsoft.com/en-us/library/jj591577.aspx>
- [10] H. Eichenhard, “Why shared mutable state is the root of all evil”, 2013. [Online]. Available: <http://henrikeichenhardt.blogspot.com/2013/06/why-shared-mutable-state-is-root-of-all.html>
- [11] D. Pinto, “Optimistic or pessimistic locking – Which one should you pick”, 2014. [Online]. Available: <https://blog.couchbase.com/optimistic-or-pessimistic-locking-which-one-should-you-pick/>
- [12] MSDN Microsoft, “Two-Phase Commit”, [Online]. Available: [https://msdn.microsoft.com/en-us/library/aa754091\(v=bts.10\).aspx](https://msdn.microsoft.com/en-us/library/aa754091(v=bts.10).aspx)
- [13] Microsoft.com, “A CQRS and ES Deep Dive”, 2014. [Online]. Available: [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/jj591577\(v=pandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/jj591577(v=pandp.10))
- [14] Microsoft.com, “Command and Query Responsibility Segregation (CQRS) pattern”, 2017. [Online]. Available: <https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs>

- [15] G. Young, “CQRS Journey”, 2014. [Online]. Available: [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/jj554200\(v=pandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/jj554200(v=pandp.10))
- [16] J. Boner, V. Klang, “Reactive programming versus Reactive Systems”. [Online]. Available: <https://www.lightbend.com/reactive-programming-versus-reactive-systems>
- [17] J. Boner, D. Farley, R. Kuhn, M. Thompson, “The Reactive Manifesto”, 2014. [Online]. Available: <http://www.reactivemanifesto.org/>
- [18] Wikipedia, “The Actor Model”, 1973. [Online]. Available: [https://en.wikipedia.org/wiki/Actor\\_model](https://en.wikipedia.org/wiki/Actor_model)
- [19] Akka.io, “How the Actor Model meets the needs of modern, distributed systems”. [Online]. Available: <https://doc.akka.io/docs/akka/2.5.5/scala/guide/actors-intro.html>
- [20] RxJava, “RxJava Anatomy: What is RxJava, how it is designed and how it works”, 2017. [Online]. Available: <https://blog.mindorks.com/rxjava-anatomy-what-is-rxjava-how-rxjava-is-designed-and-how-rxjava-works-d357b3aca586>
- [21] YOLO, “Real-time object Detection”, [Online]. Available: <https://pjreddie.com/darknet/yolo/>
- [22] M. Fowler, “CQRS. Command Query Responsibility Segregation”, 2011. [Online]. Available: <http://martinfowler>
- [23] H. McKee, O. White, “How Akka works: Akka A to Z, An illustrated White Paper”, 2018. [Online]. Available: <https://www.lightbend.com/blog/how-akka-works-akka-a-to-z-illustrated-white-paper>
- [24] D. Gosh, “Functional and Reactive domain modelling”, 2017.
- [25] V. Vernon, “Reactive Message Patterns with the Actor Model”, 2016.
- [26] H. McKee, “Designing Reactive Systems”, 2016.
- [27] RaspberryPI, “Documentation”. [Online]. Available: <https://www.raspberrypi.org/>
- [28] CV-tricks, “Zero to Hero: Guide to object detection using Deep Learning”, [Online]. Available: <http://cv-tricks.com/object-detection/faster-r-cnn-yolo-ssd/>
- [29] M. Fowler, “Making Architecture Matter”, 2015. [Online]. Available: <https://www.youtube.com/watch?v=DngAZyWMGR0>
- [30] EvenStore, “Open-source functional database”. [Online]. Available: <https://eventstore.org/>
- [31] InfluxDB, “The modern engine for Metrics and Events”. [Online]. Available: <https://www.influxdata.com/>
- [32] Akka.io, “The Main Page”. [Online]. Available: <https://akka.io/>

- [33] Akka.io, “Actors”. [Online]. Available: <https://doc.akka.io/docs/akka/2.5/actors.html>
- [34] Observable, “Observable”. [Online]. Available: <http://reactivex.io/documentation/observable.html>
- [35] Operators, “Operators”. [Online]. Available: <http://reactivex.io/documentation/operators.html>
- [36] GPL V3, “License”. [Online]. Available: <https://www.gnu.org/licenses/gpl-3.0.en.html>
- [37] J. Fu, “Vehicle Detection Pipeline based on YOLO neural network algorithm”, 2017. [Online]. Available: <https://github.com/JunshengFu/vehicle-detection>
- [38] Reactive-crossroad, Repository, 2018 [Online]. Available: <https://gitlab.com/plavreshin/reactive-crossroad/tree/master>
- [40] S. Kapadia, “Akka streams Backpressure”, 2016. [Online]. Available: <https://chariotsolutions.com/blog/post/simply-explained-akka-streams-backpressure/>
- [41] Wikipedia, “ACID”, [Online]. Available: <https://en.wikipedia.org/wiki/ACID>
- [42] Wikipedia, “API”, [Online]. Available: [https://en.wikipedia.org/wiki/Application\\_programming\\_interface](https://en.wikipedia.org/wiki/Application_programming_interface)
- [43] MSDN Microsoft, “CQRS Journey”, [Online]. Available: [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/jj554200\(v=pandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/jj554200(v=pandp.10))
- [44] M. Fowler, “DDD”, 2006. [Online]. Available: <https://martinfowler.com/bliki/UbiquitousLanguage.html>
- [45] Wikipedia, “DSL”, [Online]. Available: [https://en.wikipedia.org/wiki/Domain-specific\\_language](https://en.wikipedia.org/wiki/Domain-specific_language)
- [46] Wikipedia, “FP”, [Online]. Available: [https://en.wikipedia.org/wiki/Functional\\_programming](https://en.wikipedia.org/wiki/Functional_programming)
- [47] Wikipedia, “MQTT”, [Online]. Available: <https://en.wikipedia.org/wiki/MQTT>
- [48] Wikipedia, “OOP”, [Online]. Available: [https://en.wikipedia.org/wiki/Object-oriented\\_programming](https://en.wikipedia.org/wiki/Object-oriented_programming)
- [49] Wikipedia, “OLTP”, [Online]. Available: [https://en.wikipedia.org/wiki/Online\\_transaction\\_processing](https://en.wikipedia.org/wiki/Online_transaction_processing)
- [50] Wikipedia, “PaaS”, [Online]. Available: [https://en.wikipedia.org/wiki/Platform\\_as\\_a\\_service](https://en.wikipedia.org/wiki/Platform_as_a_service)




- [51] Wikipedia, “REST”, [Online]. Available: [https://en.wikipedia.org/wiki/Representational\\_state\\_transfer](https://en.wikipedia.org/wiki/Representational_state_transfer)
- [52] Wikipedia, “SaaS”, [Online]. Available: [https://en.wikipedia.org/wiki/Software\\_as\\_a\\_service](https://en.wikipedia.org/wiki/Software_as_a_service)
- [53] Graphite, “Graphite”, [Online]. Available: <https://graphiteapp.org/>
- [54] Akka.io docs, “Error Handling”. [Online]. Available: <https://doc.akka.io/docs/akka/2.5/stream/stream-error.html>
- [55] Scala, “Programming Language”. [Online]. Available: <https://www.scala-lang.org/>
- [56] JVM, “JVM”. [Online]. Available: [https://en.wikipedia.org/wiki/Java\\_virtual\\_machine](https://en.wikipedia.org/wiki/Java_virtual_machine)

## Appendix 1 – Reactive-crossroad repository [\[38\]](#)

Pavel Lavreshin > reactive-crossroad > Repository

master reactive-crossroad / +

History Find file Web IDE

 **Proceed with streams and actors**  
Pavel Lavreshin authored less than a minute ago d4e8843d

Name	Last commit	Last update
actors/src/main	Proceed with streams and actors	less than a minute ago
assets	Initialize all modules and dashboard	3 days ago
dashboard	Initialize all modules and dashboard	3 days ago
domain/src/main/scala/domain	Proceed with streams and actors	less than a minute ago
gatling	Initialize all modules and dashboard	3 days ago
http/src/main	Proceed with streams and actors	less than a minute ago
project	Proceed with streams and actors	less than a minute ago
simulation/src	Initialize all modules and dashboard	3 days ago
src/test	Initialize all modules and dashboard	3 days ago
streams/src/main/scala/streams	Proceed with streams and actors	less than a minute ago
.gitignore	Proceed with streams and actors	less than a minute ago
.scalafmt.conf	Fresh project, created with sbt-fresh	2 months ago
.travis.yml	Initialize all modules and dashboard	3 days ago
LICENSE	Fresh project, created with sbt-fresh	2 months ago
NOTICE	Fresh project, created with sbt-fresh	2 months ago
README.md	Fresh project, created with sbt-fresh	2 months ago
build.sbt	Proceed with streams and actors	less than a minute ago
version.sbt	Initialize all modules and dashboard	3 days ago

## Appendix 2 - Time-series metrics for vehicle weight measurements

```
package domain.measurements

import java.time.Instant

import domain.vehicle.VehicleType

/** Following is a sample json for weight measurements
 * {
 * "weight" : 1550.0,
 * "vehicleType" : "RegularCar",
 * "measuredAt" : "2018-05-20T17:12:44.367Z"
 * }
 */

// Below is an domain entity conforming above defined json

case class WeightMeasurement(
  weight: Double,
  vehicleType: VehicleType,
  measuredAt: Instant,
)

object WeightMeasurement {
  import io.circe._
  import io.circe.generic.semiauto._
  import io.circe.java8.time.encodeInstant
  import io.circe.java8.time.decodeInstant

  implicit val encoder: Encoder[WeightMeasurement] = deriveEncoder[WeightMeasurement]
  implicit val decoder: Decoder[WeightMeasurement] = deriveDecoder[WeightMeasurement]
}
```

## Appendix 3 - Time-series metrics for surface measurements

```
package domain.measurements

import java.time.Instant

import domain.measurements.SurfaceMeasurement.SurfaceLayerCondition

/**
 * {
 *   "surfaceLayerCondition" : "Wet",
 *   "waterFilmHeight" : 0.05,
 *   "measuredAt" : "2018-05-20T17:32:01.138Z"
 * }
 */

// Below is an domain entity conforming above defined json payload
case class SurfaceMeasurement(
  surfaceLayerCondition: SurfaceLayerCondition,
  waterFilmHeight: Double,
  measuredAt: Instant)

object SurfaceMeasurement {
  sealed trait SurfaceLayerCondition
  object SurfaceLayerCondition {
    case object Dry extends SurfaceLayerCondition
    case object Wet extends SurfaceLayerCondition
    case object Ice extends SurfaceLayerCondition
  }

  import io.circe._
  import io.circe.generic.semiauto._
  import io.circe.java8.time.encodeInstant
  import io.circe.java8.time.decodeInstant
  import io.circe.generic.extras.semiauto.deriveEnumerationEncoder
  import io.circe.generic.extras.semiauto.deriveEnumerationDecoder

  implicit val surfaceLayerDecoder: Decoder[SurfaceLayerCondition] = deriveEnumerationDecoder[SurfaceLayerCondition]
  implicit val surfaceLayerEncoder: Encoder[SurfaceLayerCondition] = deriveEnumerationEncoder[SurfaceLayerCondition]

  implicit val encoder: Encoder[SurfaceMeasurement] = deriveEncoder[SurfaceMeasurement]
  implicit val decoder: Decoder[SurfaceMeasurement] = deriveDecoder[SurfaceMeasurement]
}
```

## Appendix 4 - Time-series metrics for weather measurements

```
package domain.measurements

import java.time.Instant

/**
 * {
 * "surfaceTemperature" : 4.0,
 * "externalTemperature" : 6.0,
 * "windSpeed" : 10.0,
 * "humidityLevel" : 30.0,
 * "measuredAt" : "2018-05-20T17:44:34.935Z"
 * }
 */

// Below is an domain entity conforming above defined json
case class WeatherMeasurement(
  surfaceTemperature: Double,
  externalTemperature: Double,
  windSpeed: Double,
  humidityLevel: Double,
  measuredAt: Instant)

object WeatherMeasurement {
  import io.circe._
  import io.circe.generic.semiauto._
  import io.circe.java8.time.encodeInstant
  import io.circe.java8.time.decodeInstant

  implicit val encoder: Encoder[WeatherMeasurement] = deriveEncoder[WeatherMeasurement]
  implicit val decoder: Decoder[WeatherMeasurement] = deriveDecoder[WeatherMeasurement]
}
```

## Appendix 5 - Defined commands according to DDD / CQRS model

```
package domain

sealed trait Command

object Command {
  import io.circe.generic.semiauto._
  import io.circe.java8.time.encodeInstant
  import io.circe.java8.time.decodeInstant

  case class Create(config: CrossroadConfig) extends Command
  case class EnableSensor(sensorId: SensorId) extends Command
  case object StartMaintenance extends Command
  case object StopMaintenance

  case class VehicleStop(laneId: LaneId, createdAt: Instant) extends Command
  case class VehicleCrash(
    laneId: LaneId,
    createdAt: Instant,
    vehicleType: VehicleType) extends Command
  case class VehicleLaneQueue(
    laneId: LaneId,
    createdAt: Instant,
    vehicleType: VehicleType) extends Command

  case class SurfaceConditionUpdate(
    surfaceLayerCondition: SurfaceLayerCondition,
    measuredAt: Instant
  ) extends Command

  case class WeatherUpdate(
    surfaceTemperature: Double,
    externalTemperature: Double,
    humidityLevel: Double,
    measuredAt: Instant,
  ) extends Command

  implicit val encoder: Encoder[Command] = deriveEncoder[Command]
  implicit val decoder: Decoder[Command] = deriveDecoder[Command]
}
```

## Appendix 6 - Defined events according to DDD / CQRS model

```
package domain

sealed trait Event {
  def timestamp: Instant
}

object Event {
  case class Created(crossroadConfig: CrossroadConfig, timestamp: Instant) extends Event
  case class SensorEnabled(sensorId: SensorId, timestamp: Instant) extends Event
  case class MaintenanceStarted(timestamp: Instant) extends Event
  case class MaintenanceStopped(timestamp: Instant) extends Event

  case class VehicleStopped(
    laneId: LaneId,
    createdAt: Instant,
    timestamp: Instant) extends Event
  case class VehicleCrashed(
    laneId: LaneId,
    incidentId: String,
    vehicleType: VehicleType,
    createdAt: Instant,
    timestamp: Instant) extends Event
  case class VehicleLaneQueued(
    laneId: LaneId,
    incidentId: String,
    vehicleType: VehicleType,
    createdAt: Instant,
    timestamp: Instant) extends Event

  case class SurfaceConditionUpdated(
    surfaceLayerCondition: SurfaceLayerCondition,
    measuredAt: Instant,
    timestamp: Instant,
  ) extends Event

  case class WeatherUpdated(
    surfaceTemperature: Double,
    externalTemperature: Double,
    humidityLevel: Double,
    measuredAt: Instant,
    timestamp: Instant,
  ) extends Event
}
```

## Appendix 7 - Persistent Actor per every Crossroad

```
class CrossroadPersistentActor extends PersistentActor with LazyLogging {
  protected val crossroadId: String = self.path.name
  protected val persistentType: String = self.path.parent.parent.name
  override val persistenceId: String = s"$persistentType-$crossroadId"
  protected val snapshotInterval: Int = 100

  private var stateOpt: Option[CrossroadState] = None
  def state: CrossroadState = stateOpt.getOrElse(sys.error("State is still empty"))

  def updateState(event: Event): Unit = {
    stateOpt = for (s ← stateOpt) yield s.update(event)
  }

  override def receiveRecover: Receive = {
    case e: Event ⇒ updateState(e)
    case SnapshotOffer(_, s: CrossroadState) ⇒ stateOpt = Some(s)
  }

  override def receiveCommand: Receive = {
    case c: Command ⇒ onCommand(c) match {
      case Right(e) ⇒ persisted(e) { event ⇒
        logger.info(s"Persisted nr: $lastSequenceNr event: $event")
        updateState(event)
        context.system.eventStream.publish(event)
        if (lastSequenceNr % snapshotInterval == 0 && lastSequenceNr ≠ 0) {
          saveSnapshot(state)
        }
      }
      case Left(f) ⇒
    }
  }

  def onCommand(cmd: Command): Either[String, Event] = {
    cmd match {
      case c: Create ⇒ onCreate(c)
      case c: EnableSensor ⇒ onEnabledSensor(c)
      case c: VehicleStop ⇒ onIncident(c)
      case c: VehicleCrash ⇒ onIncident(c)
      case c: VehicleLaneQueue ⇒ onTrafficJam(c)
      case c: SurfaceConditionUpdate ⇒ onConditionChanged(c)
      case c: WeatherUpdate ⇒ onConditionChanged(c)
      case StartMaintenance ⇒ onInternalCommand(c)
      case StopMaintenance ⇒ onInternalCommand(c)
    }
  }
}
```



## Appendix 8 - Worker Actor for vehicle recognition stream

```
class VideoRecognitionWorker(vehicleMetrics: VehicleMetrics)
  extends Actor
    with ActorLogging {

  override def receive: Receive = idle

  def idle: Receive = {
    case Start =>
      log.info("Listening for videoRecognition results")
      vehicleMetrics.status.inc()
      context become receiving
  }

  def onIncident(x: VehicleIncident): Unit = {
    vehicleMetrics.incidents.inc()
    crossRoadActor() ! Command.VehicleCrash(
      x.crossroadId,
      x.incidentPayload.laneId,
      createdAt = x.createdAt,
      vehicleType = x.vehicleType)
  }

  def onVehiclePassed(x: VehiclePassed): Unit = {
    vehicleMetrics.passed.inc()
  }

  def receiving: Receive = {
    case x: VehiclePassed => onVehiclePassed(x)
    case x: VehicleIncident => onIncident(x)
    case Pause =>
      vehicleMetrics.status.dec()
      context become idle
  }

  private def crossRoadActor: () => ActorRef = {
    ClusterExtension(context.system).crossroadPersistentActor
  }
}

object VideoRecognitionWorker {
  type IncidentDetails
  def props(vehicleMetrics: VehicleMetrics): Props = {
    Props(new VideoRecognitionWorker(vehicleMetrics))
  }
  sealed trait VideoRecognitionWorkerMsg
  case class VehiclePassed(
    crossroadId: CrossroadId,
    vehicleType: VehicleType)
}
```

## Appendix 9 - Worker Actor for Time-series metrics

```
class MetricsWorker(sensorMetrics: SensorMetrics)
  extends Actor
    with ActorLogging {

  override def receive: Receive = idle

  def idle: Receive = {
    case Start =>
      log.info("Listening for metrics results")
      sensorMetrics.status.inc()
      context become receiving
  }

  def differenceAboveThreshold(x: TemperatureMeasurement): Boolean = {
    CrossroadState.get(x.crossroadId)
      .exists(c => c.sensorsData.isSignificantDiff(x.surfaceTemp, x.externalTemp))
  }

  def receiving: Receive = {
    case x: TemperatureMeasurement =>
      sensorMetrics.surfaceTemperature.inc(x.surfaceTemp)
      sensorMetrics.externalTemperature.inc(x.externalTemp)
      if (differenceAboveThreshold(x)) {
        log.warning("Detected significant change in temperature conditions: ", x)
        crossRoadActor() ! Command.WeatherUpdate(
          crossroadId = x.crossroadId,
          surfaceTemperature = x.surfaceTemp,
          externalTemperature = x.externalTemp,
          humidityLevel = 0.0,
          measuredAt = x.measuredAt)
      }
    case x: SurfaceConditionMeasurement =>
      crossRoadActor() ! Command.SurfaceConditionUpdate(
        crossroadId = x.crossroadId,
        surfaceLayerCondition = x.surfaceLayerCondition,
        measuredAt = x.measuredAt)
    case Pause =>
      sensorMetrics.status.dec()
      context become idle
  }

  private def crossRoadActor: () => ActorRef = {
    ClusterExtension(context.system).crossroadPersistentActor
  }
}

object MetricsWorker {
  def props(sensorMetrics: SensorMetrics): Props = {
```

## Appendix 10 - Handling of video recognition results with Akka streams

```
class VideoRecognitionTransformations(
  vehicleMetrics: VehicleMetrics)(
  implicit system: ActorSystem,
  materializer: Materializer)
  extends LazyLogging {

  import system.dispatcher

  def run(): KillSwitch = {
    val killSwitch = KillSwitches.shared("VideoRecognitionResults")
    val (ref, done) = processVideoRecognitionResults(killSwitch).run()

    done.onFailure {
      case ex =>
        killSwitch.shutdown()
        materializer.scheduleOnce(10.minutes, () => VideoRecognitionTransformations.this.run())
    }

    killSwitch
  }

  private def processVideoRecognitionResults(
    killSwitch: SharedKillSwitch): RunnableGraph[(ActorRef, Future[Done])] = {
    // back-pressure strategy applied once there is > 100 elements in buffer
    def videoRecognitionPipelineStarted: Source[VideoRecognitionWorkerMsg, ActorRef] = {
      Source.actorRef[VideoRecognitionWorkerMsg](100, OverflowStrategy.backpressure)
    }
    videoRecognitionPipelineStarted
      .via(killSwitch.flow)
      .via(persistMetrics)
      .via(transformToCommands)
      .alsoTo(dispatchCommands)
      .toMat(Sink.foreach(println))(Keep.both)
  }

  private val persistMetrics: Flow[VideoRecognitionWorkerMsg, VideoRecognitionWorkerMsg, NotUsed] = {
    Flow[VideoRecognitionWorkerMsg].map { msg =>
      msg match {
        case x: VehiclePassed => vehicleMetrics.passed.inc()
        case x: VehicleIncident => vehicleMetrics.incidents.inc()
        case Start => vehicleMetrics.status.inc()
        case Pause => vehicleMetrics.status.dec()
      }
    }
  }
}
```

## Appendix 10 (continued) – Handling of video recognition results with Akka streams

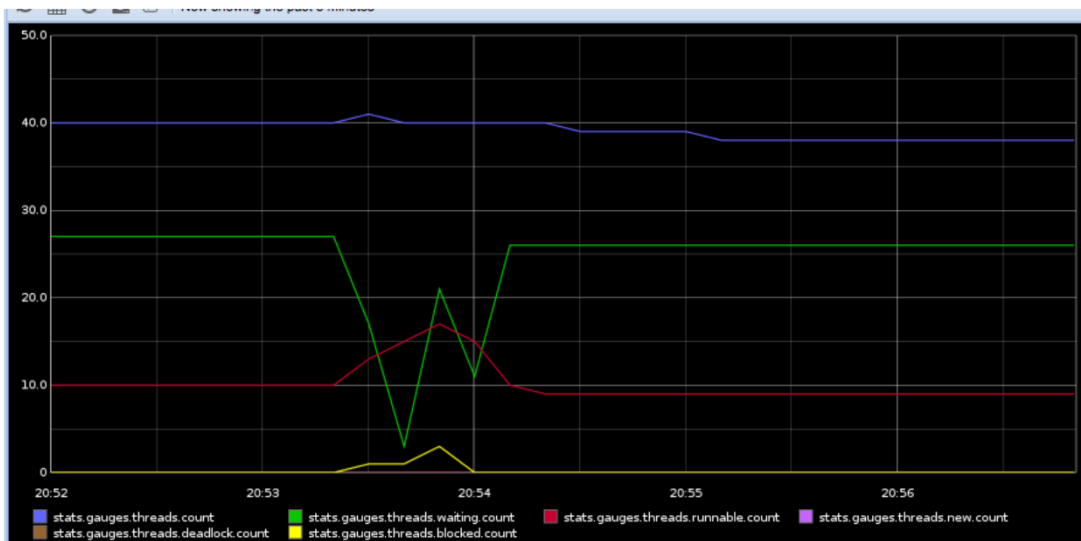
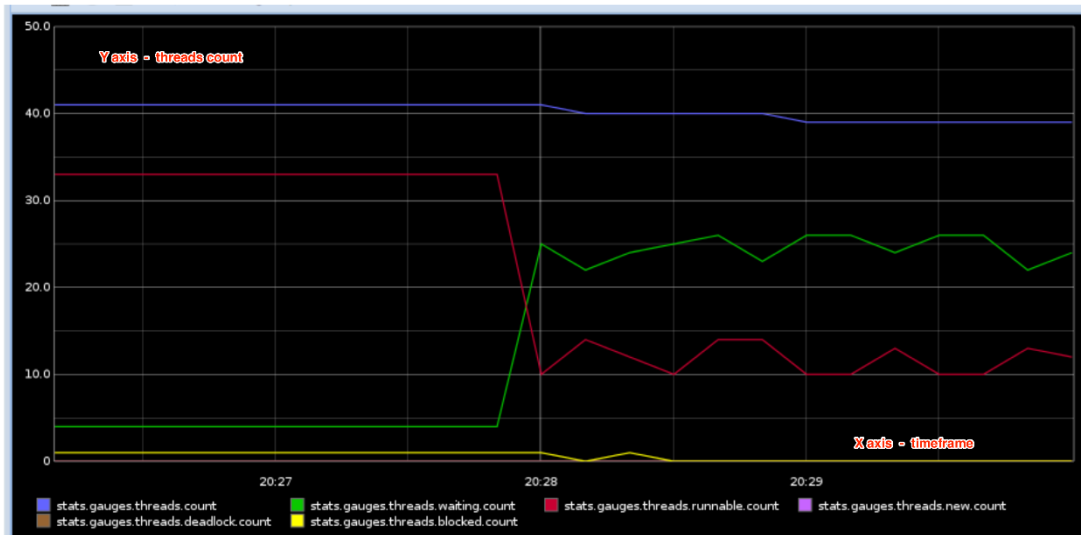
```
private val transformToCommands: Flow[VideoRecognitionWorkerMsg, List[Command], NotUsed] = {
  Flow[VideoRecognitionWorkerMsg].map {
    case x: VehiclePassed ⇒ List.empty
    case x: VehicleIncident ⇒ List(Command.VehicleCrash(
      x.crossroadId,
      x.incidentPayload.laneId,
      createdAt = x.createdAt,
      vehicleType = x.vehicleType))
    case Start ⇒ List.empty
    case Pause ⇒ List.empty
  }
}

private val superVisionDecider: Supervision.Decider = { ex ⇒
  logger.error("Error during command dispatch", ex)
  ex match {
    case NonFatal(_) ⇒ Supervision.Resume
    case _ ⇒ Supervision.Stop
  }
}

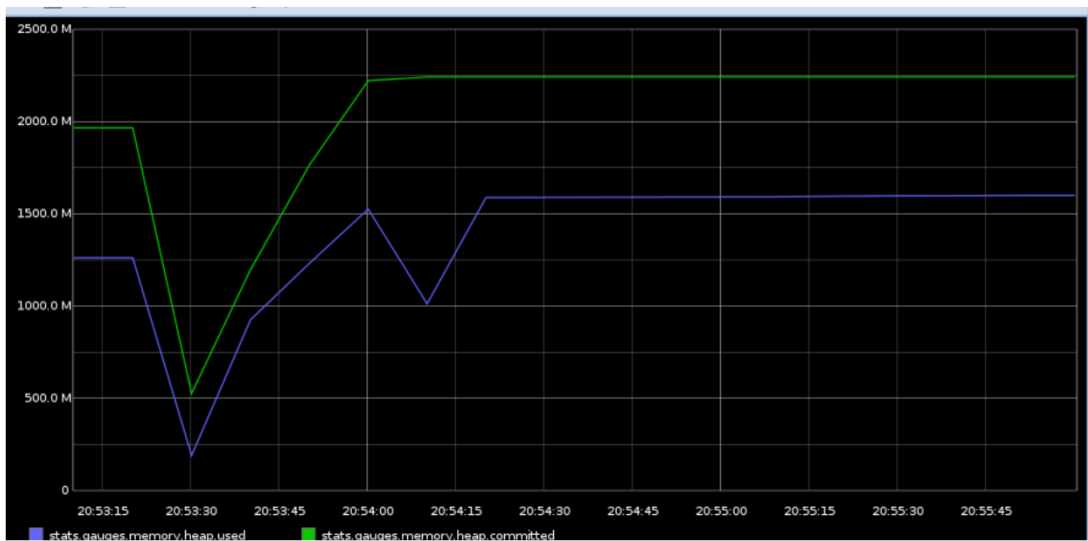
private val dispatchCommands: Sink[List[Command], NotUsed] = {
  Flow[List[Command]].mapAsync(parallelism = 1) { commands ⇒
    Future.sequence(commands.map { cmd ⇒
      crossRoadActor() ! cmd
      Future.successful(cmd)
    })
  }
  .withAttributes(ActorAttributes.supervisionStrategy(superVisionDecider))
  .to(Sink.ignore)
}

private def crossRoadActor: () ⇒ ActorRef = {
  ClusterExtension(system).crossroadPersistentActor
}
}
```

## Appendix 11 – Actors vs Streams Threads utilization



## Appendix 12 – Actors vs Stream Memory / Heap utilization



## Appendix 13 – Actors vs Streams GC performance

