A Real-Time Reactive Platform for Data Integration and Event Stream Processing

ALEXANDRE TAMBORRINO
A Real-Time Reactive platform for Data Integration and Event Stream Processing

ALEXANDRE TAMBORRINO

Master’s Thesis at Zengularity, examined by KTH ICT
Supervisor: Antoine Michel
Examiner: Vladimir Vlassov

TRITA xxx yyyy-nn
Abstract

This thesis presents a Real-time Reactive platform for Data Integration and Event Stream Processing. The Data Integration part is composed of data pullers that incrementally pull data changes from REST data sources and propagates them as streams of immutable events across the system according to the Event-Sourcing principle. The Stream Processing part is a Tree-like structure of event-sourced stream processors where a processor can react in various ways to events sent by its parent and send derived sub-streams of events to child processors. A processor use case is maintaining a pre-computed view on aggregated data, which allows to define low read latency business dashboards that are updated in real-time. The platform follows the Reactive architecture principles to maximize performance and minimize resource consumption using an asynchronous non-blocking architecture with an adaptive push-pull stream processing model with automatic back-pressure. Moreover, the platform uses functional programming abstractions for simple and composable asynchronous programming. Performance tests have been performed on a prototype application, which validates the architecture model by showing expected performance patterns concerning event latency between the top of the processing tree and the leaves, and expected fault-tolerance behaviours with acceptable recovery times.
## Contents

### List of Figures

### List of listings

1 **Introduction**
   1.1 Context and system overview .................................. 1
   1.2 Related work .................................................. 5
   1.3 Contributions .................................................. 6

2 **Requirements** .................................................. 7
   2.1 Functional requirements ....................................... 7
   2.1.1 Data Integration ............................................. 7
   2.1.2 Journal and Stream Processing ............................... 8
   2.2 Non-functional requirements ................................... 10
   2.2.1 Data Integration ............................................. 10
   2.2.2 Journal and Stream Processing ............................... 11

3 **Study of functional programming abstractions for concurrency and asynchronicity** 13
   3.1 Monadic Futures .................................................. 13
   3.1.1 The problems of blocking IO with threads .................. 13
   3.1.2 The problems of asynchronous non-blocking IO with callbacks 14
   3.1.3 Monadic Futures for composable asynchronous non-blocking programming .................................................. 15
   3.1.4 Promises ....................................................... 18
   3.2 Iteratees ......................................................... 18
   3.3 Actor model ....................................................... 22
   3.3.1 The actor model in Akka ...................................... 22
   3.3.2 Mixing Actors with Futures ................................... 23

4 **Architecture and implementation of the Data Integration part** 25
   4.1 Architecture ..................................................... 25
   4.1.1 Puller actor system .......................................... 25
   4.1.2 Incremental pull jobs .................................... 27
4.2 Implementation
4.2.1 Puller actor system
4.2.2 Example of an incremental pull job
4.2.3 Distribution

5 Architecture and implementation of the Journal and Stream Processing part
5.1 Architecture
5.1.1 Naive push-only solutions
5.1.2 Pull-based streaming with event-sourced processors
5.1.3 Fault-tolerant persistent processors with exactly-once side-effect semantic
5.1.4 Adaptive push-pull streaming
5.1.5 Side-effect stream processors
5.1.6 Optimized ACK mechanism with back-pressure using Iteratees
5.1.7 Distributed processors with TCP-level back-pressure
5.1.8 The Journal: a special case of persistent processor
5.1.9 Use case business application
5.2 Implementation
5.2.1 Abstractions choice
5.2.2 PathId serialization and deserialization into MongoDB
5.2.3 Processors
5.2.4 Journal
5.2.5 Distribution
5.2.6 Example application

6 Performance evaluation
6.1 Latency and resource consumption
6.2 Fault-tolerance
6.3 Conclusion

7 Conclusion and future work
7.1 Conclusion
7.2 Future work
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Immutable datastore and the Event Sourcing principle</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Global architecture</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>A tree of stream processors</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>A stream processor</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>In-order insertion of a sub-stream in a stream</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Back-pressure mechanism with Enumerator and Iteratee using Futures</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Puller actor system</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>Incremental pull stream pipeline</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Pull-based streaming</td>
<td>40</td>
</tr>
<tr>
<td>5.2</td>
<td>Fault-tolerance in event-sourced pull-based processors</td>
<td>43</td>
</tr>
<tr>
<td>5.3</td>
<td>A generational tree of PathIds</td>
<td>44</td>
</tr>
<tr>
<td>5.4</td>
<td>State transition diagram for each child of a parent processor</td>
<td>46</td>
</tr>
<tr>
<td>5.5</td>
<td>The stream replay mechanism with side-effect processors</td>
<td>47</td>
</tr>
<tr>
<td>5.6</td>
<td>Distributed processors with TCP-level back-pressure</td>
<td>49</td>
</tr>
<tr>
<td>5.7</td>
<td>Use case: real-time low-latency dashboards on business data</td>
<td>51</td>
</tr>
<tr>
<td>5.8</td>
<td>PathId serialization ensuring ordering in MongoDB local journals</td>
<td>54</td>
</tr>
<tr>
<td>5.9</td>
<td>Processor implementation flow</td>
<td>56</td>
</tr>
<tr>
<td>6.1</td>
<td>Average latency between the Journal and a Dashboard while varying the Journal push rate</td>
<td>66</td>
</tr>
<tr>
<td>6.2</td>
<td>Maximum consumption rate of a dashboard while varying the Journal push rate</td>
<td>67</td>
</tr>
<tr>
<td>6.3</td>
<td>Latency of events between the Journal and a Dashboard with a push rate of 100 events per second</td>
<td>68</td>
</tr>
<tr>
<td>6.4</td>
<td>Latency of events between the Journal and a Dashboard with a push rate of 500 events per second</td>
<td>69</td>
</tr>
<tr>
<td>6.5</td>
<td>Latency of events between the Journal and a Dashboard with a push rate of 1000 events per second</td>
<td>70</td>
</tr>
<tr>
<td>6.6</td>
<td>Latency of events between the Journal and a Dashboard with variable push rates</td>
<td>71</td>
</tr>
</tbody>
</table>
List of listings

1. A callback in Javascript ........................................... 14
2. The "pyramid of doom" in Javascript .............................. 15
3. Performing two asynchronous operations concurrently in Javascript 16
4. Futures in Scala .................................................. 16
5. The Monad trait in Scala .......................................... 17
6. Future composition in Scala ....................................... 18
7. Promises in Scala .................................................. 19
8. A counter Iteratee ............................................... 19
9. A simple Enumerator ............................................... 19
10. Map and filter Enumeratees ..................................... 20
11. Stream composition .............................................. 20
12. Asynchronous non-blocking stream processing .................. 21
13. A counter actor ................................................... 23
14. Mixing Futures with Actors ..................................... 24
15. JobScheduler actor .............................................. 29
16. JobWorker actor .................................................. 30
17. Cron-style configuration to schedule jobs ........................ 31
18. Enumeratee that keeps only the most recent FinancialSoftwareEvent of each resource ........................................... 32
19. Enumeratee that re-order events in ascendant order ............... 32
20. Enumeratee that gets a resource according to its id ............... 32
21. ZEvent: a Journal event ........................................ 33
22. PerformSideEffects Iteratee ................................. 34
23. The whole stream processing pipeline from a data source to the Journal ...................................................... 34
24. Configuration file for master node - Akka Remoting ............. 35
25. Configuration file for worker node - Akka Remoting ............. 35
26. Enumerator that streams the last events of a data source ......... 37
27. PathId serialization and deserialization ......................... 55
28. Processor implementation ........................................ 58
29. Persistent processor and side-effect processor implementation . 59
30. Journal implementation ......................................... 61
31 Distributed processors: HTTP interface of a parent processor . . . . 62
32 Distributed processors: Remote source implementation of a child pro-
cessor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
33 Implementation of the FlatSnapshot side-effect processor . . . . . . 64
Chapter 1

Introduction

1.1 Context and system overview

Data is now at the center of organizations and is increasingly heterogeneous with an explosion of data sources that each exposes data in its own format that can be structured, semi-structured or non-structured. Another major trend is that data processing needs to be real-time, because business men no longer want to wait a whole day to have reports and alerts on their business data. Last but not least, the volume of data that enterprises need to analyze is constantly growing, which is commonly referred as 'Big Data'.

To meet these requirements, traditional Data Warehouse software start to be out-dated. They often propose to deal only with structured data in order to store it in a relational database. Moreover, they are often batch-oriented: the ETL mechanism (data extraction, transform and load) regularly happens once or twice per day, and there is no mechanism for real-time subscriptions on new data events (as highlighted by Jay Kreps, engineer at LinkedIn, in his article 'The Log: What every software engineer should know about real-time data’s unifying abstraction'). Furthermore, Data Integration, Data Storage and Data Reporting are often coupled into a single monolithic architecture.

Thus, a new kind of architecture for Data Integration and Data Processing is needed in order to meet these new requirements: real-time processing of potentially big volumes of unstructured data. This thesis presents an architecture that solves this problem using decoupled components that communicate with immutable events that represent data changes. Events flow across the platform enabling components to react to data changes in various ways. Real-time should be understood as soft real-time in comparison to batch modes that are more common in Big Data frame-

---

CHAPTER 1. INTRODUCTION

works. For example, event propagation across the platform should be measured in milliseconds or seconds, whereas batch jobs are often measured in hours. Moreover, in a real-time platform, the notion of Big Data is more related to the push rate of events than the size of an event itself. Thus, the platform should take care of possible performance problems in order to handle high push rates.

Each event represents the change (creation, update or deletion) made to a data resource at a particular time. Based on the Event-Sourcing principle\(^2\), events are stored in a Journal that is an ordered sequence of events. Then, the stream of events coming from the Journal can be processed by data consumers that can react to the change of data (see Figure 1.2 for the global architecture). An example of data consumer can be one that maintains a pre-computed view on the data that is updated upon each event, or one that pushes notifications to another service upon the reception of some kinds of events.

An example use case is when an organization uses different SaaS services for each of its teams. For instance, the sales team uses a SaaS software to process their sales pipeline, the project management team uses another SaaS software to manage the production teams, etc... Without a central data backbone, it is not possible to have a global view on the company data. The platform I present in this thesis can integrate these different SaaS softwares using their REST API, detect what changes have been made on the data, and emit the corresponding events. As a result, data consumers can use these events to update dashboards about the company data in real-time, mixing the data coming from different sources. A data consumer can also push a notification to SaaS service X when it receives an event from SaaS service Y, allowing real-time synchronization between heterogeneous services.

An advantage of Event Sourcing is that the whole history of the system is stored. Events are immutable changes made to the data and are always appended to the Journal (never deleted or modified). As a result, the system stores not only the current state of the data, but also all its previous states. This allows two interesting properties.

First, it is possible to query past states of the data. This can be very useful for various use cases where one is interested in the data history, for example a financial audit.

Moreover, storing all the data changes greatly improves the fault-tolerance of the system. As events are not deleted, it is always possible to come back in the past in the Journal, delete some delete events that were put by mistake, and replay the events after them to re-build the system in a right state. This is also referred as Human Fault-Tolerance\(^3\) in a mutable system, if an user accidentally delete a data entry, it is lost for ever. But in an immutable system, the deletion is just another

---


\(^3\)Nathan Marz. Human Fault-tolerance. Feb. 2013. URL: [https://www.youtube.com/watch?v=IpjrHueSBxg](https://www.youtube.com/watch?v=IpjrHueSBxg).
1.1. CONTEXT AND SYSTEM OVERVIEW

event added to the journal. Figure 1.1 illustrates the difference between a mutable system and an immutable event-sourced system.

![Mutable data system](image1)

**Mutable data system**

<table>
<thead>
<tr>
<th>Person</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>Stockholm</td>
</tr>
<tr>
<td>Alice</td>
<td>Paris</td>
</tr>
</tbody>
</table>

update(Bob, Madrid)

![Immutable data system](image2)

**Immutable data system**

<table>
<thead>
<tr>
<th>Person</th>
<th>Location</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>Stockholm</td>
<td>1396597161</td>
</tr>
<tr>
<td>Alice</td>
<td>Paris</td>
<td>1396590864</td>
</tr>
</tbody>
</table>

update(Bob, Madrid)

<table>
<thead>
<tr>
<th>Person</th>
<th>Company</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>Madrid</td>
<td>1398699987</td>
</tr>
<tr>
<td>Bob</td>
<td>Stockholm</td>
<td>1398597161</td>
</tr>
<tr>
<td>Alice</td>
<td>Paris</td>
<td>1398597161</td>
</tr>
</tbody>
</table>

Figure 1.1: Immutable datastore and the Event Sourcing principle

This kind of architecture is also known as CQRS\(^4\) (Command Query Responsibility Segregation). The core principle of CQRS is to decouple the write part and the read part of a system. The write part (Data Integration) only needs to push immutable events to the Journal in an append-only fashion, which is very efficient because there is no mutation of the data and no read-write contentions as in traditional databases. The read part is a set of denormalized pre-computed views that are optimized for low read latency (as the views are automatically re-computed when a new related event comes in). An obvious downside of such an architecture is that data is eventually consistent: when a data producer has received the acknowledgment from the Journal, there is no guarantee that data consumers has already processed the event and updated the data view.

This model also allows very easy distribution of the platform because it enables a message-oriented architecture where each component (data producer, journal, data consumers with data views) only exchanges messages (events) with each other (share-nothing architecture).

CHAPTER 1. INTRODUCTION

The platform is composed of three main parts:

- **Data Integration**, that must integrate several data sources in order to emit events (data changes) to the Journal.

- **Journal**, an abstraction for a sequence of immutable events. The Journal must expose methods to insert events, and expose methods to subscribe to the stream of events.

- **Stream processing**, where one can define a tree of data consumers (stream processors) that can react to events, maintain derived pre-computed views on the data, and emit new streams of events.

Nonetheless, this kind of evented architecture must be done with a lot of care concerning technical architecture. The platform needs to do lot of IO in order to push the stream of events from data sources to data consumers, and must parallelize a lot of operations. Moreover, it must ensure that the stream of events (producers) does not overwhelm the stream processors (consumers), i.e. if consumers process data slowly, producers must try to slow the push rate. The platform should also deal with possible failures of components and offer strong guarantees in these cases (like no message loss or duplication).
1.2. RELATED WORK

In order to fulfill those requirements, the platform will apply the principles of the 
Reactive Manifesto\(^5\) in order to guarantee that the platform is scalable, event-driven, 
resilient and responsive (the four Reactive Traits). An asynchronous non-blocking 
approach with a share-nothing architecture will be used to develop the platform 
in order to optimize resource consumption, decouple components to be able to dis-
tribute them easily, take easily advantage of parallelization and handle failures. The 
platform is developed using functional programming in the Scala programming lan-
guage\(^6\) in order to leverage functional programming abstractions to better handle 
asynchronous and stream-oriented code.

1.2 Related work

There exist several Big Data frameworks for real-time stream processing. Among 
them, Apache Kafka\(^7\) and Apache Storm\(^8\) have been thoroughly studied for this 
thesis.

Apache Kafka is a high-throughput distributed messaging system developed at 
LinkedIn. It uses a distributed publish-subscribe model where data producers can 
publish events to topics and data consumers can subscribe to topics. It is durable 
by persisting events on disk and data consumers can pull events with a guaranteed 
ordering. However, as it uses a publish-subscribe abstraction, it does not enable 
the user to clearly define stream processing flows (such as trees or DAGs) where 
components are both data consumers (receiving events from parent components) 
and data producers (sending events to their child components).

Apache Storm is a distributed and fault-tolerant real-time computation frame-
work developed at Twitter. It enables the user to define a DAG of stream processors 
that can receive events from their parent(s) and send derived events to their chil-
dren. However, messages are not persisted on disk, so there is no durability, which 
implies that slow processors are forced to keep past events in-memory if we want 
fast processors to move on to the next events without waiting for slow processors 
(more details will be given on these types of recurrent stream processing issues in 
chapter\(^5\)).

\(^6\)The Scala programming language. URL: \url{http://www.scala-lang.org/}
\(^7\)Neha Narkhede Jay Kreps and Jun Rao. “Kafka: a Distributed Messaging System for Log 
Processing”. In: NetDB 2011: 6th International Workshop on Networking Meets Databases (June 
netdb11papers/netdb11-final12.pdf}
\(^8\)Nathan Marz. Apache Storm: a distributed and fault-tolerant real-time computation framework. 
URL: \url{http://storm.incubator.apache.org}
As explained in more details in the thesis, our platform will take some architecture patterns of these two frameworks to achieve an original architecture with a list of properties that none of these frameworks fully provide on their own.

1.3 Contributions

The main contributions of this thesis are:

- Definition of the architecture of the Data Integration part and its implementation
- Definition of the architecture of the Event Stream Processing part and its implementation as a generic library
- Implementation of a business use case application using the generic library for event stream processing, as well as performance tests on this application
Chapter 2

Requirements

2.1 Functional requirements

This section details the following functional requirements:

- Incremental pull of data changes from various data sources’ REST APIs with data cleaning and validation.

- Insertion of data events in the Journal, ensuring no event duplication and no event loss even in cases of transient failures.

- Stream processing system composed of stream processors forming a tree structure. Each processor must ensure an exactly-once semantic for side-effects even in cases of transient failures. The stream processing system must also ensure no event duplication and no event loss even in case of transient failures, which implies a possibility for processors to replay the stream.

2.1.1 Data Integration

The Data Integration part of the platform needs to integrate several data sources in order to push data events into the Journal. Integration means that it must be able to detect the changes made to the data, and push events that can be either create event or update event or delete event. In the following, we call a data entry a resource. A resource is a keyed data defined by its id (for example /client/1 for a resource of type client of id 1). Each type of resource has a defined set of fields (for example a client will have a field name, address, ...).

More specifically, the platform needs to integrate several data sources that expose REST APIs. Such APIs expose information concerning business data such as new sales information, new financial information... Each time that a resource is modified in one of these data sources, the platform should detect this change, apply some data cleaning and transformation, create an event from it and push it into the Journal.
CHAPTER 2. REQUIREMENTS

The problem with most REST APIs is that they are not evented, i.e. they are pulled-based and not pushed-based. One must send an HTTP request to query new data each time they need to. There exists some techniques to stream data via HTTP 1.1 and the Chunked Transfer Encoding, but the REST APIs that the platform needs to integrate do not expose such stream interface. Thus, the architecture of this part needs to provide a way to perform incremental pull from data sources, and then transform it in a push stream of events towards the Journal. Moreover, the platform needs to make sure to insert the events in the same order that they happened in their data source.

2.1.2 Journal and Stream Processing

The Journal must provide a way for data producers to push one or several events that represent the creation, update or delete of a resource. Moreover, it must allow data consumers to subscribe to the stream of inserted events. Events must be immutable and are stored in a sequence that respects the insertion order. The stream of events pushed to the data consumers (stream processors) must be in the same order than the insertion order and with no event loss or duplication. Of course, the Journal must be persistent to be able to recover its data after a shutdown or a crash.

The Stream Processing part is the most complex part of the platform. This part should be a library that allows the user to define a tree a stream processors (see Figure 2.1), where the root of the tree is the Journal.

A stream processor receives events coming from its parent node. Upon the reception of an event, it can do one or several of these actions (see Figure 2.2):

- Creation of a sub-stream: the stream processor can transform a received event to a stream (several events), creating a sub-stream inside the global stream. The sub-stream must be inserted in-place in the stream: the whole sub-stream should be send in-order to the node’s children before processing the next incoming event. For example, in Figure 2.3 the processing of an input event 1 produces a sub-stream of out events 1-1, 1-2 and 1-3. Even if another input event 2 arrives, it should not be processed before the whole sub-stream 1-1, 1-2 and 1-3 has been produced and sent to the processor’s children. This function is called process.

- Side-effect with exactly-once semantics: The second action possible is to perform a side-effect upon each of the event of the sub-stream generated by the process method. This side-effect can for example consist in updating a database representing a derived view on the data. This method, called performSideEffect, must have an exactly-once semantic even in case of failures, so that the user can safely define non-idempotent side-effects.

2.1. FUNCTIONAL REQUIREMENTS

Figure 2.1: A tree of stream processors

Figure 2.2: A stream processor
2.2 Non-functional requirements

This section details the following non-functional requirements:

- Easy scale up and scale out with a share-nothing architecture.
- Decoupled processors that can consume the stream with heterogeneous processing speeds without affecting each other.
- Optimized resource consumption in the whole system with non-blocking IO.
- All the previous non-functional requirements should ensure a soft real-time property (as defined in the introduction). In a few words, for a realistic event push rate as in the business use case application, the end-to-end event processing latency should be measured in seconds (not minutes or hours).

2.2.1 Data Integration

The Data Integration part must be able to scale up easily as one of the goals of the platform is to potentially handle high push rates of events. Scale up means that the puller should automatically make the best possible use of all cores available on
2.2. NON-FUNCTIONAL REQUIREMENTS

A machine in order to parallelize the various pulls. The different parts of the puller should also be easily distributable in case of the load if too big for one machine to handle.

To prevent software and/or hardware faults that can happen in every kind of IT systems, the puller should also be fault-tolerant, i.e. if a component experiences a transient failure, the system should ensure that no event is duplicated or lost.

Moreover, the nature of the puller implies that it will spend the majority of its time doing IO to query different data sources. Those IOs can have various durations depending on the size of the data to pull, the latency and bandwidth of the data sources, etc. We want to optimize the use of resource (CPU, RAM) despite the fact that the platform is very IO-oriented. This enables to maximize the event push rate that a given machine can handle, and therefore minimize the cost of the infrastructure once the platform is in production. Chapter 3 will show how asynchronous non-blocking IO meets these expectations.

Another non-functional requirement is to have clean and composable code source despite its asynchronous nature. Asynchronous code can indeed lead to maintenance nightmare if the wrong abstractions are used. Chapter 3 will show that the use of functional programming solves these problems.

2.2.2 Journal and Stream Processing

The Journal and Stream Processing part requires complex non-functional requirements in order to optimize resource consumption and maximize performance.

A common problem with stream processing is to manage the flow rate. A producer can indeed produce events at a rate superior to the processing rate of a consumer. This problem is even more important when there is a tree-like structure of stream processors instead of a linear structure. Indeed, the platform should handle the fact that even if sibling processors in the tree have different processing speeds, they do not block each other based on the slowest sibling. In other words, sibling processors should be totally decoupled so that when a new event is sent from a parent to its children, the parent does not have to wait that its slowest children has finished to process the event in order to send the next event to them. This property guarantees that a long processing will not slow down other parallel slow processing (so that an event stream that goes only through fast processors can keep a low latency). This problem should be handled while minimizing RAM consumption in order to make the best use of the resources in the system so that a given resource configuration can handle a higher push rate of events.

Stream processors should also be easily distributable in order to deal with event flows that are too big for one machine to handle.
Chapter 3

Study of functional programming abstractions for concurrency and asynchronicity

The architecture of the platform is heavily based on functional programming concepts to handle concurrency and asynchronicity in an effective and composable way. The following section describes and compares these abstractions.

3.1 Monadic Futures

3.1.1 The problems of blocking IO with threads

To handle concurrency and IO, traditional languages use native threads and blocking IO. A thread is a unit of execution that is has its own stack on the underlining OS, and concurrency is achieved by switching threads on the machine cores. For example, with the blocking IO model, a thread that is waiting for IO is preempted by the OS. Traditionally threads have a high resource cost, both fixed cost (the default stack size for a thread is 1 MB on a 64 bit JVM), high context switching cost and high creation time. In case of Web-oriented application, a new thread is generally spawned for each new client, and if the Web application needs to call several backend services (that is usually the case in modern Service Oriented Architectures), this thread will be blocked, doing nothing but using stack space and causing context switching. Such a model has been proved to be inefficient for a large number of concurrent clients for Web applications that call various backend services and/or perform stream-oriented connections as highlighted by James Roper in its article "Scaling Scala vs Java". This is even more important when backend services can occasionally be slow / fail. In case of a blocking IO model, clients’ threads which request this failed service will wait for this service (until a timeout), causing a very

\[\text{http://jazzy.id.au/default/2012/11/02/scaling_scala_vs_java.html}\]
high number of threads in the server. This high number of threads prevents the other requests (calling another non-failed service) to be performed efficiently because the server spends a lot of its time doing context switching between blocked threads that are doing nothing. This is even worst if you have a maximum number of threads allowed in the server (that is usually the case in cloud platforms): new clients can not connect at all to your server because there is no thread to allow to them. Non-blocking IO servers are also known as evented servers. Mark McGranaghan highlights the difference between blocking IO and non-blocking IO in his article about Threaded vs Evented Servers. If we define \( c \) the CPU time that each request takes and \( w \) the total time of the request including waiting time calling external services, an evented server performs way better than a threaded server when the ratio \( w/c \) is high (so when a request spends a majority of its time waiting for external services).

**3.1.2 The problems of asynchronous non-blocking IO with callbacks**

In order to avoid the problems caused by blocking IO, one can use a non-blocking IO model: when a thread is doing an IO operation, it doesn’t wait until the IO is finished but rather provides a mechanism to notify the caller when the IO is finished. Meanwhile, the thread can be used for other tasks, like serving other web clients.

The problem is that this kind of asynchronous non-blocking programming can easily lead to hard code maintenance if no proper abstraction is used. The common way of many languages to deal with asynchronicity is to provide a callback mechanism (Javascript may be the language that uses them the most). A callback is way to perform an asynchronous operation by providing a function as a parameter of the function that does the asynchronous operation. The parameter function will be called back when the asynchronous operation is finished. An example of a GET HTTP request to a web service in Javascript is shown in Listing 1.

```javascript
performHttpGet("http://www.example.com", function(error, response) {
  if (!error) {
    console.log("Response status: " + response.status);
  }
});

Listing 1: A callback in Javascript
```

In Listing function(error, response) {...} is the user-defined function that is called back when the asynchronous GET request returns. We see that callbacks are only about side-effects: no value is returned by the performHttpGet function. This causes a serious lack of composability, popularly known as "callback

3.1. MONADIC FUTURES

Listing 2 shows how to perform several asynchronous operations sequentially with the callback model.

```javascript
action1(function(error, response1) {
  if (!error) {
    action2(function(error, response2) {
      if (!error) {
        action3(function(error, response3) {
          if (!error) {
            action4(function(error, response4) {
              // do a side-effect with response4
            });
          } else {
            return; // error occurred
          }
        });
      } else {
        return; // error occurred
      }
    });
  } else {
    return; // error occurred
  }
});
```

Listing 2: The "pyramid of doom" in Javascript

Such coding style is called "pyramid of doom" because the code invariably shifts to the right, and the intermediary steps can not be reused to compose them later with other operations.

Moreover, doing concurrent operations with the callback model is not easy too. We want to perform 2 asynchronous operations in parallel, and do something with the results. Listing 3 shows how to do such in standard Javascript.

The fact that the callback model is based on closures that performs side-effect prevents easy composability. What I mean by composability is the fact of defining independently various asynchronous operations, and then compose them (sequentially, in parallel) to obtain a composed result of these actions. Moreover, error handling must be done manually for each asynchronous operation. A Monadic Future is an abstraction coming from functional programming that solves these problems.

3.1.3 Monadic Futures for composable asynchronous non-blocking programming

A Future is a monadic abstraction that stands for a value that may be available in the future. Using Scala’s notation, a future is a type that is parametrized by the type of the value that will eventually be available. For example, `Future[Int]` is a type that represents an eventual integer. With futures, asynchronous functions return a `Future[ResponseType]` instead of taking a callback function as a parameter. Listing 4 shows simple future creations.

We see in Listing 4 that futures can be used for non-blocking IO, but also as an abstraction for concurrency. In the example, the main thread executing the code
CHAPTER 3. STUDY OF FUNCTIONAL PROGRAMMING ABSTRACTIONS FOR CONCURRENCY AND ASYNCHRONICITY

```javascript
var results = [];

function doSomethingWithResults(results) {
  // final callback
}

action1(function(error, response) {
  results[0] = response;
  if (results.length == 2) {
    doSomethingWithResults(results);
  }
})

action2(function(error, response) {
  results[1] = response
  if (results.length == 2) {
    doSomethingWithResults(results);
  }
});
```

Listing 3: Performing two asynchronous operations concurrently in Javascript

```scala
val futureComputation: Future[Int] = future {
  // do long computation
}
```

Listing 4: Futures in Scala

does not block on both methods. The 'long computation' will be done in another thread as it is encapsulated by a `Future {`. Behind the scene, futures are multiplexed into a thread pool named ExecutionContext in Scala. ExecutionContexts can be passed to methods that return a future. This allows to decouple the `concurrency semantic` (which tasks should be run concurrently) from the `concurrency implementation` (an ExecutionContext can for example limit the number of threads it can use, etc.). Twitter's engineer and researcher Marius Eriksen highlights this idea in his "Your Server as a Function" paper³ where he states that the Future abstraction is a declarative data-oriented way of doing asynchronous programming.

Moreover, as the same author highlights in his article "Future aren’t ersatz threads"⁴ Futures "model the real world truthfully": a `Future[T]` can either result


3.1. MONADIC FUTURES

in a success with a value of type T, or with an error (Exception), which is inherently
the case with IO operations due to the unreliability of the network.

The term monadic comes from Monads, a key abstraction in typed functional
programming coming from the Haskell world. Thoroughly defining what is a monad
is out of the scope of this thesis, but in a few words a monad is a type that en-
capsulates another type in order to perform operations on it. Some operations are
mandatory to define a monad. Listing 5 defines the trait in Scala to define a monad,
coming from the book Functional Programming in Scala.

```
trait Monad[F[_]] extends Functor[F] {
  def unit[A](a: => A): F[A]
  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]

}
```

Listing 5: The Monad trait in Scala

unit allows to construct a monad that encapsulates a value of type A (equiva-
lent to future {}), map allows to apply a function to the encapsulated value, and
flatMap allows to apply a function to the encapsulated value that returns itself a
monad.

A Future is a monadic type, meaning that it extends the monad trait and im-
plements the unit and flatMap methods. These methods (and many more) allow
powerful compositions between different Futures instances. A Future is also an im-
mutable data structure with all the functional programming advantages related to
it (safe sharing between threads, ease of reasoning with referentially transparent
code, etc).

For example, map allows to transform the result of an asynchronous operation.
flatMap allows sequential and parallel composition. flat comes from flatten because
flatMap can transform a Future[Future[T]] to a Future[T]. Listing 6 illustrates these
compositions.

We see in Listing 6 that we avoid the 'pyramid of doom' effect for sequential
composition, and that concurrent composition is very simple and safe compared to
callback-based programming. Moreover, monad operations allows automatic error
propagation. In the sequential composition example, if for instance action2 failed,
the action3 and action4 will not be executed, and futureResult4 will be a failed
future with the Exception that action2 throwed. For more examples of future com-
positions, LinkedIn’s engineer Yevgeniy Brikman highlights the composability of

9781617290657.
Chapter 3. Study of Functional Programming Abstractions for Concurrency and Asynchronicity

*/
* Sequential composition of asynchronous operations returning Integers
*/
val future1: Future[Int] = action1()
val future2: Future[Int] = future1 flatMap (result1 => action2(result1))
val future4: Future[String] = future2
.flatMap(result2 => action3(result2))
.flatMap(result3 => action4(result3))
.map(result4 => "This is result4: " + result4)

_listing{6: Future composition in Scala}

Futures in his article "Play Framework: async I/O without the thread pool and callback hell".

In summary, a monadic future is an immutable abstraction for concurrency and asynchronicity that allows easy reasoning and composition. However, a future only model the fact that one value will be available in the future. Hence, it is not directly applicable to model asynchronous non-blocking streams.

3.1.4 Promises

A promise is an abstraction that can be seen as a Writable Future. One can create a Promise, and get a Future from it. Then, when the method promise.success(value) is called, the related Future is fulfilled asynchronously with this value. Listing 7 illustrates the use of Promises.

Promises can for example be used to let communicate a consumer and a producer as we will see in the Stream Processing architecture and implementation chapter.

3.2 Iteratees

To model streams that can be produced in an asynchronous non-blocking way, the Iteratee abstraction can be used. An Iteratee is an immutable data structure that allows incremental, safe, non-blocking and composable stream processing. One key

---

3.2. ITERATEES

```scala
def future2 = future1 map (value => value + 1) // future2 will eventually contain the value 2
// ...
promise1.success(1) // triggering future1 with the value 1
```

Listing 7: Promises in Scala

The feature of Iteratee is back-pressure that will be described later on. The Iteratee way of processing stream involves three abstractions: Enumerators, Enumeratees and Iteratees. The Iteratee library from Play Framework is used for the examples.

An Iteratee is a stream consumer and is represented by the type `Iteratee[E, A]`. An iteratee receive chunks of type E in order to produce a result of type A. The main method of an iteratee is a fold method (a common functional programming method) that passes around its current state and the next chunk to process. Listing 8 shows how to define an iteratee that sums the number of characters it receives.

```scala
def chunkCounter: Iteratee[String, Int] = Iteratee.fold { (chunk, nbBytesReceived) => nbBytesReceived + chunk.length }
```

Listing 8: A counter Iteratee

An Enumerator is a stream producer of type `Enumerator[E]`. Listing 9 shows how to create an enumerator that streams a collection.

```scala
def producer: Enumerator[String] = Enumerator.enumerate(List("foo", "bar", "foobar"))
```

Listing 9: A simple Enumerator

An Enumeratee is a stream transformer of type `Enumeratee[E, F]` transforming chunks of type E to chunks of type F. Listing 10 shows several enumeratee examples.

An interesting properties of Iteratees/Enumerators/Enumeratees is that they can be easily composed. Listing 11 shows how to run the data flow that returns a Future of the result.

During the stream processing, either the Enumerator can choose to end the stream (sending an EOF) or the Iteratee can choose that it has processed enough

---

7Sadek Drobi and Guillaume Bort. Handling data streams reactively. URL: [http://www.playframework.com/documentation/2.2.x/Iteratees](http://www.playframework.com/documentation/2.2.x/Iteratees)
CHAPTER 3. STUDY OF FUNCTIONAL PROGRAMMING ABSTRACTIONS FOR
CONCURRENCY AND ASYNCHRONICITY

val filter = Enumeratee.filter[String](chunk => chunk != "bar")
val mapper = Enumeratee.map[String](chunk => chunk + "!")

Listing 10: Map and filter Enumeratees

// An composed enumeratee that will perform filter and map to the stream
val filterAndMapper: Enumeratee[String, String] = filter compose mapper

// A composed enumerator which produces chunk that will be filtered and mapped
val modifiedProducer: Enumerator[String] = producer through filterAndMapper

// Please note that all the operations were lazy for now.
// Now we run the enumerator into the iteratee in order to process the flow
val futureResult: Future[Int] = modifiedProducer run chunkCounter
// the future result will be "foo!".length + "foobar!".length == 11

Listing 11: Stream composition

chunk to compute its final value and stop the stream processing by returning a Done
state to the Enumerator.

A very interesting feature is when we have to compose asynchronous operations
in order. Iteratees allows to define producer, transformer and consumer that return
Futures of their operations. Moreover, Play Framework’s Iteratee library provides
helpers that allow for example to fetch an HTTP stream in a non-blocking way
through an Enumerator. Listing 12 shows how to get a Http stream (for example a
stream of tweets), call an external web service to process the chunks, and insert the
processed chunks in a database with the position of this chunk in the stream. The
Iteratee design ensures that chunks will be process in the order of the producer,
even with asynchronous operations during the processing flow.

In the example of Listing 12, chunks are guaranteed to be in-order when they
are inserted in the database. The M after the map and fold methods stands for
Monad, that is in our case a Future.

Under the hood, an Enumerator is a kind of fold method that push chunks in an
Iteratee. An Iterate can be in Cont state meaning that it wants to consume more
chunks from an Enumerator, in Done state meaning that it does not want more input
to compute its result value, or in Error state. For each chunk, the Iteratee returns
to the Enumerator a Future of its state. When this future is redeemed, it means
that the Iteratee has finished processing the current chunk, so the Enumerator can
push the next chunk into it (which can be also done by returning a Future of the
chunk). Figure 3.1 illustrates this mechanism.

Thus we have asynchronous production and consumption with the consumer that
'tells' (via its future state) to the producer that it is ready to consume more chunks

20
3.2. ITERATEES

```scala
val asyncProducer: Enumeratee[String] = getHttpStream("http://example.com/stream")

val asyncTransformer: Enumeratee[String] = Enumeratee.mapM {
  Chunk =>
    val futureProcessedChunk: Future[String] = callWebService(Chunk)
    futureProcessedChunk
}

val asyncDatabaseSink = Iteratee.foldM[String, Int](0) { (processedChunk, n) =>
  val futureInserted = database.insert((processedChunk, n))
  futureInserted map (_.=> n + 1)
}

// Starting the processing
asyncProducer through asyncTransformer run asyncDatabaseSink
```

Listing 12: Asynchronous non-blocking stream processing

Figure 3.1: Back-pressure mechanism with Enumerator and Iteratee using Futures

(or not). This mechanism is known as **back-pressure** and allows the consumer to slow down the producer rate depending on its own processing speed.

With back-pressure, we can differentiate two kinds of producers: **cold sources** and **hot sources**. Cold sources are sources that produce chunks from a static durable collection, meaning that the consumer can process the stream at its own speed without the risk of losing events. On the contrary, hot sources are for example events coming from a network connection. If the consumer is not ready to consume the next chunk, a choice has to be done (drop the event, buffer it, ...). These problems will be studied more thoroughly further in the report.

Future and Iteratee compositions allow to model asynchronous processing of both single values and streams of values, but they can not be used to model state and arbitrary (non-linear) message passing. The Actor model is an abstraction that can be used to model your entire system to easily handle concurrency, distribution and fault-tolerance with thread-safe encapsulated states.
CHAPTER 3. STUDY OF FUNCTIONAL PROGRAMMING ABSTRACTIONS FOR
CONCURRENCY AND ASYNCHRONICITY

3.3 Actor model

First of all, it should be noted the Actor model is not a purely functional model as it uses some side-effects. It is generally said that it sits between functional and imperative. Nevertheless, we will study it in this part as it integrates very well with functional code and is part of Scala’s way of handling concurrency. The examples use the Akka framework that provides actor systems for the JVM in Scala.

3.3.1 The actor model in Akka

In imperative languages, synchronization of different concurrent operations is usually done by using locks (including synchronization blocks in Java). However, a lock is a very low-level primitive that can easily lead to problems like dead-lock, data inconsistency if not enough locks, slow performance if too many locks. Moreover, IO is generally done explicitly via sockets.

An actor system is a higher level abstraction to deal with concurrency, states and synchronization, and abstracts away sockets by providing a location transparent model via message passing. It enables simple fault-tolerance and distribution.

An actor is a lightweight concurrent entity which has very cheap creation cost (far cheaper than a thread). Basically, an actor receives messages from other actors and send messages to other actors. Upon the reception of a message, it can modify its internal state, send messages to other actors, and change its message handling behavior of the next messages. Each actor has one mailbox that corresponds to a FIFO queue of incoming messages. It is guaranteed that message processing is sequential and thread-safe inside a same actor, i.e. one should not have to worry about concurrency problems inside an actor when modifying its internal state. Listing 13 shows how to define a simple Actor that counts the number of messages it receives. Messages can be sent to him concurrently without worrying about synchronization problems.

Another important part of the actor model in Akka is the Tree-like hierarchical organization between actors, called Supervision Hierarchy. A parent that creates child actors is responsible for them, meaning that it should handle their possible failures (by stopping them, restarting them, etc).

Last but not least, the fact that actors communicates only via message passing (share-nothing architecture) allows location transparency, which enables easy distribution. In Akka, the fact that one or several actors are located on different machines is written in a configuration file, so one can write the exact same code for a program that executes locally or in a distributed fashion. Thus, an Actor system makes automatically the best use of a multi-core machine via parallelization.

---

8 Akka: Build powerful concurrent and distributed applications more easily. URL: [http://akka.io](http://akka.io)
3.3. ACTOR MODEL

```scala

```case object Message
```

class Counter extends Actor {
  var counter = 0

  def receive = {
    case Message =>
      counter = counter + 1
  }
}

val system = ActorSystem("MySystem")
val counter = system.actorOf(Props[Counter], name = "counter")

// Sending concurrently 100 messages to the actor
// (the send operation "!" do not block or wait for an ack)
(1 to 100).par foreach (_ => counter ! Message)
```

Listing 13: A counter actor

and the best use of distributed machines via easy distribution. Under the hood, a Scheduler is in charge of multiplexing actors into threads, like the ExecutionContext multiplexes Futures into threads.

### 3.3.2 Mixing Actors with Futures

Akka actors can be used with libraries that use Futures. However, one should pay attention to the mix between SchedulerContext and futures’ ExecutionContext. Akka ensures that all message processing are sequential and thread-safe (so that one does not have to worry about concurrency problems inside an actor), but if the result of a future tries to modify the internal state of an actor, Akka can not ensure thread-safety as it only controls the SchedulerContext that is used to process incoming messages. To avoid this, one must use the `pipeTo` pattern that transforms the result a Future in a message sent to the actor. Listing 14 illustrates this mechanism.

The following chapters describe the architecture and implementation of the platform that makes heavy use of the functional programming abstractions that have been presented in this chapter.
case class Message(data: String)
case class Result(result: String)

// An actor that receive a Message, call an async method to process it, 
// store the result of the processing
class ProcessActor extends Actor {
  var storeResults = Seq.empty[String]

  def receive = {
    case Message(data: String) =>
      val futureResult = process(data)
      futureResult map { result =>
        Result(result)
      } pipeTo self

    case Result(result: String) =>
      storeResults = storeResults :+ result
  }
}

// Below is an example that is concurrently unsafe because the future 
// modify the internal state of the actor
class UnsafeActor extends Actor {
  var storeResults = Seq.empty[String]

  def receive = {
    case Message(data: String) =>
      val futureResult = process(data)
      futureResult foreach { result =>
        storeResults = storeResults ++ result
      }
  }
}

Listing 14: Mixing Futures with Actors
Chapter 4

Architecture and implementation of the Data Integration part

As stated in the Requirements chapter, the Data Integration part must incrementally pull various REST APIs (data sources) in parallel. For each resource type in each data source, it must create an event flow. This event flow runs through several data cleaning and data transformation steps that can be asynchronous. Despite the asynchronous nature, it should ensure that the event flow remains in-order. In the end, event flows are pushed into the Journal.

4.1 Architecture

4.1.1 Puller actor system

In order to schedule and perform periodic incremental pulls of data sources, an Akka Actor system is defined.

First, the system needs for each data source to receive a "top" message corresponding to the fact that a certain data source must be queried. We will use for this Akka Quartz\footnote{Quartz Extension and utilities for cron-style scheduling in Akka. URL: \url{https://github.com/typesafehub/akka-quartz-scheduler}} a cron-style scheduler that allows to define periodic sending of certain types of messages. An usual pull rate for a data source is every 5 seconds, in order to create a near-realtime stream.

These top messages will be received by a singleton actor named JobScheduler. The purpose of the JobScheduler is to launch a child actor for each job (a job corresponds to an incremental pull from a certain data source). Once the child actor has finished the incremental pull, it kills itself. Figure 4.1 illustrates this architecture.

The JobScheduler must handle the fact that the job message rate for a data source can be faster than the incremental pull of this data source (for example if
the data source has produced a lot of new data since the last pull, or if it experiences some network problems). If a pull is still running when a new job message arrives for a resource type of a data source, the JobScheduler should ignore the new pull message to avoid doing two or more pulls in parallel of the same resource and risking a wrong order of events. The JobScheduler can do this by assigning to the actor the name of the resource and data source when it spawns a new worker child. Then, when a new job message comes in, it checks if it has a child of this name, and only if such child doesn’t exist it spawns a new child.

The actor model also allows to deal with errors. In our case, we just want to ignore the failure of a child worker. The next top message for this data source will automatically launch a new child worker for this data source. Thus, the JobScheduler actor will have a special Supervision Strategy that just ignores the failure of its children.
4.1. ARCHITECTURE

4.1.2 Incremental pull jobs

When a job is launched, it must do an incremental pull on a particular resource of a particular data source via its REST API. For each of the data source that the platform must integrate, there exists a GET method that allows to get all the resource ids of a certain resource type that were updated in descendant order (most recent first). The GET response is paginated, meaning that ids are coming 50 by 50 for example (the API caller has to make several HTTP calls until it has all the ids it wants).

A pull job has to pull the event ids that were updated after the last incremental pull. To do this, we define a stream where the producer makes one or several HTTP calls to the paginated REST API to produce a stream of JSON containing the ids of the resources updated. The producer must stop pulling when the date of the current update is less than the last update event processed during the previous job. In order to persist for each job the date of the last event processed, we use a persistent storage that stores the timestamps of the last event processed of each resource type for each data source (the storage system is the NoSQL database MongoDB\(^2\), but it can be any other simple storage system that can store an association of job names with their timestamp).

Moreover, for each resource, we are only interested in keeping the latest update. Indeed, the REST API only gives us the type of the update (create, update, delete) with the id of the resource, so if we pull (in descendant order) a delete event before an update event for a particular resource, we only want to retain the delete event.

Then, the stream of events should be re-sorted in ascendant order, then for each event we must query the REST API to transform the resource id into the resource itself, then we must clean and validate the resulting JSON to transform it to a known data model, then insert the event into the Journal, and finally update MongoDB with the timestamp of this event. Figure 4.2 illustrates this pipeline in a simple schema.

We see in Figure 4.2 that the data stream pipeline must asynchronously process (calling external services) events while keeping ordering of messages, so Iteratees and Futures will be used to meet these requirements. Moreover, an effort is made to isolate side-effects at the end of the stream pipeline in order to enable easy reuse of intermediate blocks. Isolation of side effects for better code reuse and reasoning is one of the core principles of functional programming.

Such an architecture allows transparent concurrency and parallelism up to the number of cores. Each child actor is executed concurrently, and the asynchronous stream processing is using Iteratees that use Futures to allow transparent concurrency. If we give to the Iteratees/Futures the SchedulerContext as ExecutionContext, we share threads between actors and futures, which will create the best possible use of cores in the machine (the total number of threads roughly equals to the number of cores).

---

Moreover, in case the system needs to scale out, the Actor model also allows easy distribution. In this architecture, the JobScheduler can transparently spawn some worker children on other machines. The implementation part will detail this part more thoroughly.

4.2 Implementation

4.2.1 Puller actor system

The Akka framework is used to implement the puller actor system in Scala. The JobScheduler is the main/master actor that receives top messages related to a data source and a special kind of resource, and spawns a new worker child to accomplish this task only if it has not already a child doing this particular task. Listing 15 shows the code of the JobScheduler actor.
4.2. IMPLEMENTATION

```scala
class JobScheduler extends Actor {
  private val logger = LazyLogger.apply("services.JobScheduler")

  override val supervisorStrategy = stoppingStrategy

  def receive = {
    case jobType: JobMessage =>
      // ensure sequentiality of the iterations of a same job
      val isRunning = context.child(jobType.toString) map (_ => true) getOrElse false
      if (!isRunning) {
        logger.info("Launching child...")
        val worker = context.actorOf(Props[JobWorker], name = jobType.toString)
        worker ! jobType
      } else {
        logger.warn("Job " + jobType + " ignored because the previous iteration of the job is still running.")
      }
  }
}
```

Listing 15: JobScheduler actor

The `supervisorStrategy` is set to `stoppingStrategy` in order to ignore possible failures of children. Listing 16 shows the worker actor code.

The type `Job` is the type that must be implemented for an incremental pull stream job. `( => Future[Int])` means that the pull job must be a function that take no parameter and return a Future of Int. This Future of Int will be fulfilled when the pull job is finished with the number of Journal events that were created during this iteration of the incremental pull job. Upon the completion of the future, we map it to a `JobFinished` message that we pipe to `self` (the current actor). Upon the reception of this message, it knows that the job is finished, and so it kills itself (its parent actor JobScheduler will be automatically notified by its death). Please note that `context` is part of the actor internal state, so it is not safe to access it into the Future as we saw in section 3.3.2. That’s why we pipe the future to a message that will be sent to the actor.

The module Akka Quartz allows to define in a configuration file the periodicity of the top messages that will be sent to the JobScheduler actor. See the configuration file shown in Listing 17 for an example.

4.2.2 Example of an incremental pull job

In this section we will describe the implementation of an incremental pull job. We take for example a job that pulls every 5 seconds the resources of a certain type, called Credit Notes, that has been created, updated or deleted in a SaaS financial software (called FinancialSoftware for this report).
Iteratees and Futures are used to model asynchronous non-blocking stream processing. As we will see, the composableity of Iteratees allows a very clear modularization of the different processing components.

**Enumerator of Events coming from FinancialSoftware**

The first step is to create an enumerator (a producer) that pull events that happened to a certain resource type since the last pull. The REST API of FinancialSoftware is paginated by 50, meaning that a GET request on the last events gives 50 events and a link the next "page" containing the next 50 events in descendant order. The enumerator has to pull the REST API until it detects that the current event has a date inferior to the last update date stored in MongoDB.
4.2. IMPLEMENTATION

```scala
akka {
    quartz {
        schedules {

            DataSource1ResourceType1 {
                description = "Fire DataSource1ResourceType1 every 5 seconds"
                expression = "*/5 * * ? * *"
            }

            DataSource1ResourceType2 {
                description = "Fire DataSource1ResourceType2 every 2 seconds"
                expression = "*/2 * * ? * *"
            }

            DataSource2ResourceType1 {
                description = "Fire DataSource2ResourceType1 every 5 seconds"
                expression = "*/5 * * ? * *"
            }

        }
    }
}
```

Listing 17: Cron-style configuration to schedule jobs

We have to use an enumerator that repeatedly fetch pages from the FinancialSoftware REST API until it has streamed all the events since a date named `since`, and return a stream of `FinancialSoftwareEvent` containing each the id of a resource of certain type with its related event (create, update or delete) and its date. Listing 26 shows the code of such enumerator.

The method `retrieveUpdates` returns an `Enumerator[FinancialSoftwareEvent]`. The `&>` operator between an enumerator and an enumeratee is an alias for the `through` composition method explained in section 3.2.

Stream pipeline composition

From this producer of `FinancialSoftwareEvent`, we want to apply several operations to the stream processing pipeline as illustrated in Figure 4.2. First, we want to keep only the most recent event of each resource. Listing 18 shows how to define such an Enumeratee. It returns a `Map[String, FinancialSoftwareEvent]` where document id is the key and the last event (so the first in the descendant order stream) related to this resource is the value.

Then, we must create an enumeratee that transforms this Map in a stream of events in ascendant date order (see Listing 19).

Then, we must call again the data source's REST API to transform the documentId by the document (resource) itself (see Listing 20). Note the use of `Enumeratee.mapM` that allows sequential (in-order) composition of an asynchronous
CHAPTER 4. ARCHITECTURE AND IMPLEMENTATION OF THE DATA INTEGRATION PART

```scala
def groupByDocumentIdKeepingHeadEvent:
    Enumeratee[FinancialSoftwareEvent, Map[String, FinancialSoftwareEvent]] = {
      Enumeratee.grouped(Iteratee.fold(Map.empty[String, FinancialSoftwareEvent])) {
        (record, financialEvent) =>
        val id = financialEvent.documentId
        if (!record.contains(id))
          record + (id -> financialEvent)
        else record
      }
    }
```

Listing 18: Enumeratee that keeps only the most recent FinancialSoftwareEvent of each resource

```scala
val reorder: Enumeratee[Map[String, FinancialSoftwareEvent], FinancialSoftwareEvent] =
  Enumeratee.mapConcat { mapIdToEvent =>
    val ascendingSeqOfEvents = mapIdToEvent.toSeq.sortBy { case (id, event) => event.date }
    ascendingSeqOfEvents
  }
```

Listing 19: Enumeratee that re-order events in ascendant order

```scala
def getDocument(resourceType: String):
    Enumeratee[FinancialSoftwareEvent, (JsObject, String, DateTime)] =
    Enumeratee.mapM { event =>
      val id = event.documentId
      FinancialSoftware.getResource(resourceType + "/" + id) map { response =>
        (response \
          "response").as[JsObject], event.updateType, event.date
      }
    }
```

Listing 20: Enumeratee that gets a resource according to its id

Then, several other Enumeratees are created to clean and validate the data. Their implementation will not be shown in this report because the code is long and very business specific. We name this resultant enumeratee cleanAndValidate of type Enumeratee[(JsObject, String, DateTime), Command[ZEvent]], ZEvent being the type of the Journal events. Listing 21 shows the case class ZEvent which will be more detailed in the Journal and Stream Processing part. It contains the name of the resource (for example /resourceType1/id4), the user that inserted the event in the Journal, the insertion date, the type of event and the body (data) of the event.
4.2. IMPLEMENTATION

event in a JSON object.

The **Command** type is a functional type that allows to accumulate side-effects in order to execute them at the end of the pipeline. For example, in the data validation part, the detection of erroneous data may imply to send a message back to the data source (this type of side-effect is called Annotation). To enhance code re-usability and correctness according to functional programming, the Command type accumulates the different side-effects that must be executed at the end of the pipeline. The final Iteratee that does all the side-effects is named performSideEffects. It takes a stream of Command[ZEvent], sends the annotations to the data source, writes the ZEvent to the Journal and updates the MongoDB collection that stores the date of the last event processed. The type is Option[ZEvent] because sometimes if data validation fails we don’t even want to write an event in the Journal. An Option[T] is a functional type that represents the fact that a value of type T may exist (Some(value)) or not (None). Listing 22 illustrates the Command type and the performSideEffects Iteratee (the Iteratee counts the number of events it has sent to the Journal).

case class ZEvent(
  id: PathId,
  resource: String,
  user: String,
  date: DateTime,
  name: String,
  body: JsObject)

Listing 21: ZEvent: a Journal event

Thus, we have defined data producers (Enumerator), data transformers (Enumeratee) and data sinks (Iteratee). We now just have to connect them together. Composability and static typing allows to do so easily, safely and clearly (see Listing 23). The final type of the job method is Job, the alias type for () => Future[Int] that was defined in the Puller actor system implementation section.

4.2.3 Distribution

This architecture can be easily distributed thanks to actor systems’ location transparency. Actually, the above code doesn’t need any changes to run it in a distributed environment. For example, we want the worker children that pulls DataSource1 to be executed on a remote machine different than the master machine where the JobScheduler actor runs. We can use the Akka Remote module for this use case. It allows via a configuration file to configure the JobScheduler actor to create some of its children in a remote machine rather than locally. The configuration file shown in Listing 24 should be put in the master node and tells the JobScheduler to create the

3 Akka Remoting, URL: [http://doc.akka.io/docs/akka/2.2.4/scala/remoting.html](http://doc.akka.io/docs/akka/2.2.4/scala/remoting.html)
CHAPTER 4. ARCHITECTURE AND IMPLEMENTATION OF THE DATA INTEGRATION PART

```scala
case class Command[A](date: DateTime, maybeEvent: Option[A], annotations: List[Annotation])

def performSideEffects: Iteratee[Command[ZEvent], Int] =
  Iteratee.foldM(0) {
    case (nbEvents, Command(eventDate, maybeEvent, annotations)) =>
      annotations foreach { annotation => // send to data source
        annotation.annotate()
    }
    maybeEvent match {
      case Some(event) =>
        for {
          _ <- RefJournal.write(event)
          _ <- setLastUpdate(jobKey, eventDate) // set last update date
        } yield nbEvents + 1
      case None =>
        setLastUpdate(jobKey, eventDate) map (_ => nbEvents)
    }
  }

Listing 22: PerformSideEffects Iteratee

object FinancialSoftware {
  def job: Job = {
    val resourceType = "exampleResource"

    val futureNbEventsInserted: Future[Int] =
      getLastUpdate(resourceType) flatMap { lastEventDate =>
        retrieveUpdates(resourceType, lastEventDate) &>
        groupByDocumentIdKeepingHeadEvent &>
        reorder &>
        getDocument(resourceType) &>
        cleanAndValidate|>>> // |>>> is an alias for "run"
        performSideEffects
      }
  }
}

Listing 23: The whole stream processing pipeline from a data source to the Journal child actor of names DataSource1ResourceType1 and DataSource1ResourceType2 on the remote machine of address 127.0.0.1:2553.

On the worker machine, an actor system of name "remote" should be launched with the configuration file shown in Listing 25. Moreover, one should ensure that the JVM classloader on the worker machine has a JAR containing the class JobWorker.

Thus, we transform our system to a multi-core local system to a distributed system without having to change the code. This can be very useful for systems that first have enough resources with one machine, but after a while need to be run on several machines for performance.
4.2. IMPLEMENTATION

### Akka master node

```scala
# Akka master node
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
    deployment {
      /master/SellsyCreditnotesJob {
        remote = "akka.tcp://remote@127.0.0.1:2553"
      }
    }
  }
}
remote {
  enabled-transports = ["akka.remote.netty.tcp"]
  netty.tcp {
    hostname = "127.0.0.1"
    port = 2552
  }
}
```

Listing 24: Configuration file for master node - Akka Remoting

### Akka worker node

```scala
# Akka worker node
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = 2553
    }
  }
}
```

Listing 25: Configuration file for worker node - Akka Remoting

However, one problem with the approach of Akka Remoting is that it is 'end-point to end-point' oriented, meaning that the jobs for data source X or Y are statically mapped to one machine. For elastic and adaptive scalability, it would be more interesting to give several worker machine addresses to Akka, and then let it determine by itself on which machine it is better to launch the new child worker according to the current resource availability of the machines (CPU, RAM, ...).

This approach in currently under development in Akka and is named Akka Cluster\(^4\). In Akka cluster, distribution is cluster centric instead of end-to-end centric,

---

meaning that failed nodes are automatically removed from the cluster, and new node
can be added at runtime. Moreover, it will allow automatic actor tree partitioning
on the cluster, which means that a given actor child will be automatically created
on the machine that is the more available at the current time.
### 4.2. IMPLEMENTATION

```scala
case class Page(nb: Int, totalNbPages: Int, docs: Seq[JsObject])
case class FinancialSoftwareEvent(documentId: String, updateType: String, date: DateTime)

object FinancialSoftware {
  private val dateFormat = DateTimeFormat.forPattern("yyyy-MM-dd HH:mm:ss")
  val apiUrl = "..."
  val authentificationParams = "...

  def retrieveUpdates(resourceType: String, since: DateTime): Enumerator[FinancialSoftwareEvent] = {
      val url = apiUrl + resourceType + authentificationParams + "&page=" + nb
      WS.url(url).get map { response =>
        val json = Json.parse(response.body)
        val nbPage = (json \ "pagenum").as[Int]
        val totalNbPages = (json \ "nbpages").as[Int]
        val docs = (json \ "results").as[Seq[JsObject]]
        Page(nbPage, totalNbPages, docs)
    }
  
    val producer: Enumerator[Seq[JsObject]] =
    Enumerator.unfoldM[Option[Int], Seq[JsObject]](Some(1)) {
      case Some(nextPageNb) =>
        getPage(nextPageNb).map { nextPage =>
          if (nextPage.nb < nextPage.totalNbPages) {
            Some((Option(nextPage.nb + 1), nextPage.docs))
          } else {
            // last page
            Some(None, nextPage.docs)
          }
        }
      case None => Future.successful(None)
    }

    val flattenedProducer: Enumerator[JsObject] =
    producer &> Enumeratee.mapConcat(identity)

    flattenedProducer &>
    Enumeratee.map { jsObject =>
      val id = (jsObject \ "relatedid").as[String]
      val date = (jsObject \ "date").as[DateTime]
      val eventType = (jsObject \ "type").as[String]
      FinancialSoftwareEvent(id, eventType, date)
    } &>
    Enumeratee.takeWhile { event =>
      event.date.compareTo(since) > 0
    }
  }
}
```

Listing 26: Enumerator that streams the last events of a data source
Chapter 5

Architecture and implementation of the Journal and Stream Processing part

5.1 Architecture

As presented in the Requirements chapter, the Journal needs to store the events as an append-only list. The Journal should also provide a way for stream processors to subscribe to the real-time stream of events. Stream processors can subscribe to themselves in a Tree-like structure (see Figure 2.1), and upon the reception of an event a processor can create a substream of events and perform side-effects.

5.1.1 Naive push-only solutions

An important problem that will solve the architecture is the fact that the Journal can have an event stream rate that is superior to its subscribers. More generally, any parent node (Journal or processor) can have an output stream rate that is superior to the event processing time of one or several of its children. Several simple push solutions can be applied, but none of them were applicable for our system.

First, any child can just have an in-memory buffer that stores the incoming events not yet processed. However, an in-memory buffer has obvious limitations like causing an OutOfMemory exception if the child can not handle the flow rate and start queuing a lot of events. Moreover, as said in the non-functional requirements part, one goal is to limit the RAM consumption of the platform. Thus, this solution is not applicable.

Another solution can be for a parent node to wait that all its children have processed the current event to send the next one via an ACK mechanism. An obvious issue of this approach is that the slowest child of a parent will slow down the event stream rate for all its siblings. This is clearly not acceptable for a scalable system with loose coupling between components. Moreover, such a solution implies that the failure of a child stops the stream for its siblings, which is clearly not a
fault-tolerant approach.

As a reminder, no message loss is a requirement for the platform, so dropping the events if the stream rate is too fast is not an option.

### 5.1.2 Pull-based streaming with event-sourced processors

One suitable solution is to use a pull-model instead of a push-model. For each received event, a processor processes it to produce a substream and stores the events of this substream in a local journal as a side-effect. Thus, each processor maintains its own event journal, so each processor is *event-sourced*. This allows each child to maintain a cursor on the journal of its parent pointing on the next event to pull. Thus, children can have totally different pulling and processing speeds, they are not coupled to each other. This approach of pull-based stream systems with decoupled multi-consumers using cursors comes from Apache Kafka. Figure 5.1 illustrates the notion of pull-based processors.

![Figure 5.1: Pull-based streaming](image)

A common problem with pull-based system is the polling part. How can children know when the next event is ready to be pulled? The naive way to do this is to

---

check every X seconds / milliseconds if the parent has a new event in its journal. This can waste a lot of resources when a parent has no new event for a while. The solution brought by Kafka is to perform long-polling: when the child’s cursor goes on the next event, either it pulls this next event if it exists or it blocks until a new event comes in to pull it. Thus, children do not have to pull periodically to know whether there is a new event.

However, the term "blocks" does not really get along well with our reactive non-blocking architecture. We therefore have to find a way to implement long-polling without blocking threads. To do that, we will use the Promise abstraction described in section 3.1.4. Mixing Futures, Iteratees and Promises, we will be able to implement an asynchronous non-blocking long-polling system (more details in the Implementation section 5.2).

Both the Journal and the local journals of processors are persistent using MongoDB. MongoDB is a document-oriented NoSQL database that stores BSON documents (a binary representation of JSON). The format of stored event is a BSON-serialized version of ZEvents (see Listing 21). It contains an id, the name of the resource (for example /resourceType1/id4), the user that inserted the event in the Journal, the insertion date, the type of event and the body (data) of the event in a JSON object. To model a journal, we just use a MongoDB collection where we only insert new documents (events). To keep the insertion order of events, an id of type PathId is serialized into the document. MongoDB provides a built-in id generation mechanism that keeps the insertion order, but the fact of ensuring message ordering of substream across processors implies to create a more sophisticated id generation mechanism. This will be explained thoroughly in section 5.1.3.

Thus, each event produced by a processor goes into its MongoDB local journal, and children pull events (one by one or by bulk) according to their cursor position in their parent local journal. If one or several of them are "up-to-date" with the last event of their parent, a long-polling mechanism allows to prevent them to waste resources periodically pulling their parent.

However, this mechanism can be improved. For example, if a parent processor knows that one of its children is "up-to-date", it can directly send him the next event when it has just been created without passing by the persistent storage (in order to improve efficiency). This approach is described in section 5.1.4.

5.1.3 Fault-tolerant persistent processors with exactly-once side-effect semantic

Of course, a persistent storage on MongoDB allows fault-tolerance in case of a processor crashes and restarts. When a processor restarts, it checks in MongoDB what was the id of the latest output event it has successfully processed before crashing, and asks its parent for the next event after this id (it replays the stream from where

---

CHAPTER 5. ARCHITECTURE AND IMPLEMENTATION OF THE JOURNAL
AND STREAM PROCESSING PART

it crashed). However, if a processor crashes during the processing of an event, how to know if it has successfully and entirely processed this event? What does it mean to process 'entirely' an incoming event?

First, we differentiate the process method and the performAtomicSideEffect method in a processor. As stated in the Requirement chapter, the process method takes one event and produces a substream of events from it. Its signature is

\[
\text{process(event: I)}: \text{Enumerator}[O]
\]

where I is the type of input events and O the type of out events. As we saw in the previous parts, an Enumerator is a functional abstraction for a non-blocking stream producer. The function implementing the process interface should be pure, i.e. should not have any side-effect. More precisely, it can have side-effects, but the call semantic of this function is at-least-once for each event, meaning that the same side-effect can be called several times. Thus, it is ok to have idempotent side-effects. However, for side-effects that are non-idempotent and thus requires exactly-once semantic, one can use the performAtomicSideEffect method whose signature is

\[
\text{performAtomicSideEffect(event: O)}: \text{Future[Unit]}
\]

The function is called for each output event of the created sub-stream and is guaranteed to be called exactly-once for each output event even in cases of processor failures. The Future[Unit] return type is a Future returning nothing (i.e. a side-effect). The difference of returning Future[Unit] instead of simply returning Unit is that a side-effect can be asynchronous but returning a Future of nothing allows us to known when this side-effect is finished. This enables to ensure sequentiality of side-effects (the side-effect of 'out event 1' in the substream will be finished when the side-effect of 'out event 2' begins).

With these two functions, fault-tolerance is handled in a clear manner. process is an user-defined function that is the logic of the processor: how to derive one input event to a substream of out events. As a pure function, it is not a problem if it is called several times for the same input. performAtomicSideEffect is then used on each out event to store it in a local journal. This operation must me atomic. Thus, if a processor crashes, when it recovers it just takes the last out event id "LastID" from its local journal and asks its parent for the event that generated the substream containing this out event. The parent re-sends this event which is put again in the process method (at-least-once call semantic). This process method re-creates the substream, and a filter is put on the substream to take only the out events that are after the original "LastID" output event. This filtered substream is then given to the performAtomicSideEffect method that goes on storing events in the local journal as usual. Figure 5.2 illustrates the fault-tolerance of event-sourced pull-based processors. In this example, we suppose that Processor 2 fails just after the insertion in its local journal of event 5. When it recovers, it checks its local journal what was the latest event inserted, deduces from it the generator input event (event C in our case), and asks its parent to replay the stream from event C. The parent sends event C which is put again in the process method to generate the substream event 5, event 6 and event 7. Then, between the process and the performSideEffect
5.1. ARCHITECTURE

operations, a filter is applied on the substream by Processor 2 to filter the output events that had already been inserted (in our case, event 5). And then the stream processing goes on with the insertion of event 6 and event 7.

Figure 5.2: Fault-tolerance in event-sourced pull-based processors

Thus, we have ensured no message loss, no message duplication and guaranteed message ordering in cases of processor crashes. It is easy to see why the "insert in journal" operation in performAtomicSideEffect needs to be atomic: if it is not (say it has 2 steps), if the processor fails between step 1 and step 2, we are in a position where we don’t know if the current out event has been inserted or not where the processor recovers.

This atomicity leads to a potential problem: if we already use the performAtomicSideEffect for inserting output events in the journal, there is no room left for an arbitrary side-effect that can be defined by the user of the library. This is why we define two kinds of processors: persistent processors where the performAtomicSideEffect is already implemented to insert output events in a local journal (the type of processor described in this section), and side-effect processors that allows to define an arbitrary atomic side-effect. Side-effect processors will be described in details in section 5.1.5.

Thus, for a persistent processor, the fact that an input event has been "entirely" processed means that all of the out events in the substream it has generated are inserted in the local journal.

One key operation that has not been explained is how to know which input event has generated a particular output event generated in a substream. This is a
complex task for which the notion of Path Ids has been created.

Auto generation of Tree-like Ids: Path Ids

The generation of events across the tree of processors can be seen itself as a Tree-like structure. For example, when event 1 comes from the Journal and enters in the first processor, event 1 can generate a substream of events (say event 1-1, event 1-2 and event 1-3). Then, a child processor that generates a 2-event substream will generate event 1-1-1, event 1-1-2, event 1-2-1, event 1-2-2, event 1-3-1 and event 1-3-2 (see Figure 5.3).

We see that the generation of events has a tree-shape, and each event can be characterized by a path in this tree (for example, 1-2-1). The idea of PathId is to auto-generate these path ids when events pass through processors so that it is possible to re-climb the tree given a particular event.

Concerning fault-tolerance, a child processor that recovers after a crash can check its last PathId, remove the last node 'LastNode' of it and send this path id to its parent. Then, the parent sends him the event that the child re-processes (re-creating the substream), and thanks to the number 'LastNode' of the last node the
child can know from where in the substream it crashed. The idea of id generation to
be able to retrieve from a child event the parent event comes from the "Guaranteeing
message processing" system of Apache Storm.\(^3\)

The implementation part will explain in more details how these PathIds are
serialized and deserialized in MongoDB.

The ability of re-climbing the generational tree is also very useful for side-effect
stream processors that are described in the next section.

5.1.4 Adaptive push-pull streaming

An important optimization can still be made to this model. For now, from a pro-
cessor point of view, receiving input events and sending output events are totally
decoupled tasks that are asynchronous between each others. All input events are
processed and the resultant substream is written to the local journal, and then
children consume events from this journal. If the children were late in the stream,
this is the only solution, but if the children were up-to-date in the stream, it would
be more performant to just send the processed out events directly to the children
(without having them pulling from the local journal of its parent, which involves
more IO calls to the local journal).

To do this, a processor maintains a state for each of its children. The state can
be 
 UpToDate, WaitingForDownStreamAck or Late.

 UpToDate means that the parent knows that its child is up-to-date with the
stream, so the out event(s) generated by the next input event that it will receive
can directly be sent to its child after being inserted in the local journal. Thus, the
child does not have to lookup in the journal to get the event (this saves one IO
round-trip to the local journal).

WaitingForDownStreamAck is an intermediate step where the parent knows that
it has sent an output event to its child, and is waiting for a child ACK meaning that
it has processed the event. Thus, if a new input event comes in when the parent
maintains this state for its child, it means that the child is now Late in the stream,
so the parent moves the child’s state to Late. If an ACK message comes from the
child, the parent moves the child’s state to UpToDate.

Late means that the child is late in the stream, so there is no point sending him
the processed out events when a new input event arrives (or it will receive the events
out of order). In this case, the child will replay the stream at its rate. However, if
it manages to catch-up with the real-time stream, its state will move to UpToDate.

Figure 5.4 illustrates this as a state transition diagram.

Thus, a processor adapts its stream strategy in real-time between push and pull
to optimize performance and latency.

\(^3\)Nathan Marz. Guaranteeing message processing in Apache Storm. URL: \url{http://storm.incubator.apache.org/documentation/Guaranteeing-message-processing.html}
5.1.5 Side-effect stream processors

Persistent event-sourced processors use their performAtomicSideEffect method to insert the generated out events in their local journal. We saw that a persistent processor can only do one atomic side-effect per out event in order to guarantee exactly-once side-effect semantic even in cases of failure. Moreover, a persistent processor uses a lot of disk space as it stores every output event in its local journal.

To deal with these problems, we define another type of processors: side-effect processors. Side-effect processors do not have a local journal. Instead, when one of its children is slow and ask for past events, it must ask to its parent to replay these events. However, for children that are up-to-date with the stream, it sends directly the output events to them.

Concerning the replay mechanism for slow children, this is a recursive call until the nearest parent processor that is persistent which is the tree root (Journal) in the worst case. Moreover, each intermediate side-effect processor in this recursive call must take care of sending only the minimal amount of messages in substreams. Indeed, as we re-climb the generational PathId tree, each event sent by a parent is put in the process method of its child that generates the substream. We must just send events of this substream from a certain event because all events generated from previous events of the substream have already been processed by the original side-effect processor asking for a stream replay. Figure 5.4 illustrates this mechanism.
In this example, a side-effect processor has two children: Slow Processor and Fast Processor. Fast processor has a processing time that is quicker than its parent, therefore the latter will always have an up-to-date state for its children, so it will be in push-mode for this children. However, Slow processor has a slower processing rate than its parent. Therefore, after having push event 1-1, side-effect processor will be in pull-mode for this child (Late state). When slow processor asks for a replay of the stream since event 1-2 (included), its parent side-effect processor has no choice but to ask itself to its parent persistent processor (because it has no local journal itself). The notion of Path Ids is again used here in order to deduce from event 1-2 that the generator event was event 1. Therefore, side-effect processor asks for a replay of the stream from event 1 to persistent processor, and this one sends him event 1 from its local journal. Then, side-effect processor has to apply the same filter mechanism as the recover process of crashed processor in order to send only the minimum number of events needed (in the example it filters event 1-1 and sends to slow processor the rest of the substream).

Concerning the performAtomicSideEffect method, the contract for the side-effect is that it has to be atomic if one wants to have the exactly-once side-effect semantic, and moreover the side-effect of an out event should store its PathId somewhere to be able to retrieve it when it recovers from a crash (to keep the fault-tolerance
property). Note that this is not the same than inserting the event in a journal as in persistent processors: here we only need to store the latest PathId, not the entire sequence of events. Upon a recover, a method `getLastProcessedEventId()` of signature `getLastProcessedEventId(): Future[PathId]` is able to retrieve the last PathId processed to initiate the recover mechanism explained in the previous section.

In the end, side-effect processors allow to define an arbitrary side-effect with exactly-once semantic, allow to save disk space because it does not maintain a local journal, but it implies that event stream replays take longer because a side-effect processor cannot replay the stream itself, it has to ask to its parent. However, if a processor is known to produce events slower than its children process them, a side-effect processor is a good trade-off because its "push-mode" is as fast as a persistent processor for up-to-date children.

5.1.6 Optimized ACK mechanism with back-pressure using Iteratees

Concerning the ACK mechanism from a child to its parent, a naive way to do it would be for the child to send an application-level ACK message for each event received and processed. We will take a more performant approach by leveraging the built-in back-pressure mechanism of Iteratees.

A processor is composed of one Iteratee in input (stream consumer) and several Enumerators (stream producers) in output (one per child). As explained in section 3.2, the back-pressure mechanism allows a stream producer to automatically slow down its producing rate if the consumer is slow. More precisely, for each child, the 'producing' code of it's parent Enumerator will be called to generate the next event only when the consuming Iteratee has totally consumed the current event (like an ACK mechanism). In our case, the Enumerator has the choice to get a stream from the local journal or its parent (Late case), or to directly send the latest output event (UpToDate case). The Iteratee is the child that receives an event and processes it to create a substream. Thus, we see that we don’t need to create an application-level ACK as the mechanism of notifying the producer when the consumer has finished processing an event is already built-in in the Iteratee concept. But how does this work in a distributed setting?

5.1.7 Distributed processors with TCP-level back-pressure

The pipeline composed by an Enumerator (source) and an Iteratee (sink) can be distributed via an HTTP chunked stream. An HTTP chunked stream is a unidirectional stream protocol that allows the producer (server) to maintain a connection with a client to send a possibly infinite number of chunks. An enumerator can be transformed very easily into a HTTP stream producer, and an Iteratee can be easily transformed in a HTTP stream client (more details in the Implementation part). Thus, processors can be distributed via HTTP streams on top of TCP.
5.1. ARCHITECTURE

By plugging Enumerators / Iteratees via an HTTP stream, one very interesting property is that the back-pressure mechanism remains by leveraging TCP’s flow control mechanism. In a few words, if a client does not consume its receive TCP buffer, it will send an ACK message to the sender with a sliding window of size 0, meaning that it does not want more data for now, so the sender stops sending data (this is known as TCP’s sliding window flow control protocol).

Thus, Play Framework’s Iteratee library leverages this mechanism the propagate back-pressure in a distributed setting. Therefore we have a TCP-level ACK and back-pressure mechanism, which is of course very performant compared to application level ACK messages. Figure 5.6 illustrates this concept.

![Figure 5.6: Distributed processors with TCP-level back-pressure](image)

5.1.8 The Journal: a special case of persistent processor

The Journal (root of the processing tree) is a particular processor as its source is not a replayable processor. Instead, it has multiple sources: all the pullers from the Data Integration part that push events concurrently in the Journal. Moreover, its `process` method is the identity function, as the events inserted are the same as the output events.

Concerning the ordering of the events, a particular abstraction from Play Framework Iteratee library is used, called Broadcast Channel, which allows to linearize into a single pipeline events that are pushed from different sources (more details in the Implementation part).

Concerning back-pressure, it depends on the data source if the producer can be slowed down or not. To handle different cases, the Journal has an interface to push events that allows optional back-pressure. Its signature is `write(event: E):`
**Future[E]**. E is the type of the event. The method returns a Future[E] that will be fulfilled when the event has been inserted in the journal. The event returned in the future in the same event but with a PathId added when it has been inserted in the MongoDB-based journal. Thus, this Future is like an ACK that the Journal had successfully stored the event. If another event is written once this Future has been fulfilled, it is ensured that this new event will be ordered after the previous event. The future also propagates back-pressure, because if a producer waits for this Future before sending a new event, it means that it has waited that the Journal had time to process it. Producers that can not wait for the Future to be fulfilled can call the write method several times ignoring the Futures returned. However, in this case, there is no guarantee that the event has been correctly inserted, and if all producers act like that the Journal may be overwhelmed. Thus, if possible, it is better to throttle the insertion rate using the Future returned by the write method.

### 5.1.9 Use case business application

Using this generic stream processing library, an example application has also been implemented to meet functional business requirements. The aim of this application is to maintain real-time dashboards on business data (sales, production, finance, ...). Data read latency must be really low, so creating pre-computed dashboards that update themselves upon the reception of certain kinds of events is an interesting approach. Moreover, upon the reception of some events, the platform should react by updating other external services.

![Figure 5.7 shows the tree of processors used.](image)

**Figure 5.7** shows the tree of processors used.

- **Snapshot** is a persistent processor that maintains the history of all resources (it contains all states of each resource). It takes journal events in input and return the current state of a resource in output. Its persistent nature allows other services to query its local journal to know past states of resources.

- **Pusher** is a side-effect processor responsible for updating other services upon the reception of particular events. Its side-effect is used to do an in-place update of the last PathId it has seen.

- **FlatSnapshot** is a side-effect processor that uses its side-effect to maintain a MongoDB view on the last state of each resource that is not deleted. Its process method is the identity function on the current state of a resource.

- **Dashboards** are side-effect processors that use their side-effect to maintain pre-computed aggregation views on various resources. They take the new current state of a resource as input, and returns a substream of new updated aggregation lines as output.
5.1. ARCHITECTURE

Figure 5.7: Use case: real-time low-latency dashboards on business data
CHAPTER 5. ARCHITECTURE AND IMPLEMENTATION OF THE JOURNAL AND STREAM PROCESSING PART

5.2 Implementation

5.2.1 Abstractions choice

As stated in the Architecture part, a stream processor is composed of one Iteratee in input and N Enumerators in output (one per child). Distribution is done using HTTP streaming on top of TCP. Custom processors on top of Iteratees have been selected over actors for several reasons.

First, actors does not handle back-pressure in a built-in way. But back-pressure is very important for our system in order to optimize resource consumption. Back-pressure is even used from a parent’s local journal to its children using the reactive MongoDB driver ReactiveMongo\(^4\) that exposes methods returning Enumerators. It is really convenient to have only one composable abstraction to compose streams with back-pressure from MongoDB or from other processors.

Moreover, actors do not have a simple mechanism to sequentially compose asynchronous operations. When an actor processes a message and call an asynchronous function (which returns a Future), it handle the next message in its mailbox meanwhile. An actor cannot "block in a non-blocking way" over asynchronous operations as Iteratees can. There exists a solution using the Stash trait\(^5\) that allows to put in local memory the messages that we want to process after the current event has been asynchronously processed, but it is not fault-tolerant if the actor fails and not efficient as messages are swapped between the mailbox and the actor local memory (and not compatible with back-pressure).

In the end, Iteratees, Futures and Promises are better abstractions than actors to handle this problem.

5.2.2 PathId serialization and deserialization into MongoDB

The PathId Scala class is composed of the MongoDB id of the root event inserted in the Journal, plus a Vector of Int that represents the Path in the generational event tree. A MongoDB id is an id created by the Scala MongoDB driver that is a 24-char hexadecimal string made with the current time plus a local incremental counter in order to guarantee that each document id is unique and that the id of events are strictly ascendant to retain the order of insertion.

By default, MongoDB uses the _id field of a document to store its unique id. Moreover, all MongoDB’s collections have a default index on this field. Thus, for simplicity and efficiency, we will serialize PathId to a hexadecimal string to put as a value of this _id field. This serialization has to maintain the order of events in a local journal. Indeed, ReactiveMongo’s stream capabilities from MongoDB allows to get from MongoDB a stream of all the documents of a collection since a particular id in ascendant order according to ids. Therefore, this id has to be ascendant for

\(^4\)Stephane Godbillon and contributors. Reactive Mongo: Asynchronous and Non-Blocking Scala Driver for MongoDB. URL: http://reactivemongo.org/

\(^5\)Akka 2.2: Simpler use of Stash. URL: http://letitcrash.com/post/54507231889/2-2-spotlight-simpler-use-of-stash
5.2. IMPLEMENTATION

all events of a local journal (to define the order, MongoDB does a simple String comparison from left to right).

The simplest way to do this is to first create the hexadecimal string as the root event 24-char id, and then append each integer of the path id as a padded string of 8 chars (so 8 bytes). The padding allows easy deserialization.

With this serialization, events that have the same height in the tree have ids with the same number of chars (so in particular, all event ids generated by a processor have the same size). Moreover, locally in each local journal of processor, all events generated from a root event 01 created before root event 02 will have an id smaller than all events generated from root event 02 (because they have the same size, and the root event id 01 has a higher id than 02 which are at the beginning of the hexadecimal string id). Furthermore, sibling nodes in the event tree have an incremental number according to their creation order, so order is ensured in a sub-stream. Figure 5.8 illustrates this serialization and ordering mechanism. With this model, for each input event, a processor can generate sub-events with an additional 4-byte part at the end of the PathId, so for each event a processor can generate at maximum 2 power 32 events (around 4 billion events). This limit is considered to be sufficient for the use cases of the platform.

Listing 27 shows the code of Path Id and its serialization and deserialization.

5.2.3 Processors

A stream processor is composed of one Iteratee in input and several Enumerators in output (one per child). In order to link the Iteratee with the Enumerators, we use the Promise abstraction to be able to fulfill manually a Future, and Scala STM\(^6\) to handle concurrent accesses on shared state (more details below).

When the Iteratee receives an input event, it processes it to create a substream that is flattened in-place into the main stream using the Enumeratee.mapFlatten helper. Moreover, the Enumeratee updatePathId is responsible of updating the PathId of each sub-event with the new level in the event tree.

Then, each sub-event goes through the effector method that is an Enumeratee executing asynchronously and sequentially the performAtomicSideEffect method (which is the insertion in the local journal for persistent processors).

Last, sub-events go in the downstreamTrigger Iteratee that updates the state of each child according to the state transition diagram Figure 5.4. Moreover, each child that is UpToDate had previously registered a Promise to trigger in the consumersTrigger Map. This promise is linked to a Future that is returned by the Enumerator of a child when this one is up to date. Thus, when we call promise.trySuccess(Some(outEvent)), the Future pushed in the Enumerator is fulfilled with the new sub-event (and so it will be sent to the child).

In the Enumerator corresponding to a child (the createOutStream method),

\(^6\)A lightweight STM for Scala. URL: [http://nbronson.github.io/scala-stm/](http://nbronson.github.io/scala-stm/)
when the code is called a new time (corresponding the ACK that previous events has been processed by the child in local mode, or that the events are at least in the TCP send buffer in distributed mode), we check the state of the child. If it is UpToDate or WaitingForDownStreamAck, we register a promise to be called when a new event will come in, and we put the Future linked to the promise into the Enumerator. This mechanism allows non-blocking long-polling. If it is Late, we call the since method that retrieves the past events from the sinceId parameter. If the processor is a persistent stream processor, it will directly take the past event streams from its MongoDB local journal and returns an Enumerator of it using the ReactiveMongo reactive driver. If the processor is a side-effect stream processor, it will ask its parent for the past event streams, removing the last node of the path id and then filtering the substream via an offset in order to remove the already processed sub-events (as explained in the architecture part). Figure 5.9 illustrates this mechanism.
5.2. IMPLEMENTATION

```scala
case class PathId(rootEvent: String, path: Vector[Int])

object PathId {
  import reactivemongo.bson.utils.Converters
  import java.nio.ByteBuffer

  def apply(rootEvent: String): PathId = PathId(rootEvent, Vector.empty)

  def serialize(id: PathId): String = {
    val array = id.path
    val byteBuffer = ByteBuffer.allocate(array.length * 4)
    val intBuffer = byteBuffer.asIntBuffer
    intBuffer.put(array.toArray)
    byteBuffer.flip()

    id.rootEvent + Converters.hex2Str(byteBuffer.array())
  }

  def deserialize(str: String): PathId = {
    val (idStr, pathStr) = str.splitAt(12*2)
    val byteBuffer = ByteBuffer.allocate(4)
    val arrayBytes = Converters.str2Hex(pathStr)
    val path = arrayBytes.grouped(4).toVector map { bytes =>
      byteBuffer.put(bytes)
      byteBuffer.flip()
      val int = byteBuffer.getInt
      byteBuffer.clear()
      int
    }

    PathId(idStr, path)
  }

  val min = PathId("000000000000000000000000")
}
```

Listing 27: PathId serialization and deserialization

In order to share in a thread-safe way the Maps of Promises and States between the input Iteratee and the output Enumerators (so several concurrent threads), we use Scala STM (Software Transactional Memory)\[7\]. In a few words, a STM is an optimistic approach to concurrency that let all operations on a shared data structure to be done in parallel. An operation must be committed when it is finished. If there was another commit from another operation during this operation, the operation is aborted (rollback) and tried again. Using java.util.concurrent.ConcurrentHashMap is roughly equivalent for our use case.

\[7\] A lightweight STM for Scala. URL: [http://nbronson.github.io/scala-stm/](http://nbronson.github.io/scala-stm/)
Listing 28 shows the code of the generic stream processor. Listing 29 shows the code of persistent stream processor and side-effect stream processor.
5.2. IMPLEMENTATION

// State for each consumer
sealed trait State
  case object UpToDate extends State
  case object WaitingForDownStreamAck extends State
  case object Late extends State

trait Source[E] {
  def createOutStream(sinceId: PathId): Enumerator[E]
  def since(id: PathId, included: Boolean = false): Enumerator[E]
}

trait Evented[A] {
  def getId(a: A): PathId
  def updateId(a: A, id: PathId): A
  def serialize(a: A): JsObject
}

abstract class StreamProcessor[I, O](implicit ec: ExecutionContext, ei: Evented[I], eo: Evented[O]) extends Source[O] {

import scala.concurrent.stm._

  def since(id: PathId, included: Boolean = false): Enumerator[O]
  def process(event: I): Enumerator[O]
  def performAtomicSideEffect(event: O): Future[Unit]
  def getLastProcessedEventId(): Future[PathId]

  /* Internals */

  val consumersTrigger = Ref(Map.empty[String, Promise[Option[O]])
  val consumersState = Ref(Map.empty[String, State])

  def processor: Enumeratee[I, O] = {
    Enumeratee.mapFlatten[I](inEvent => process(inEvent) &> updatePathId)
  }

  def effector: Enumeratee[O, O] = {
    Enumeratee.mapM[O](outEvent => performAtomicSideEffect(outEvent) map (_, => outEvent))
  }

  def downstreamTrigger: Iteratee[O, Unit] =
    Iteratee.foreach[O](outEvent =>
      consumersState.single transform { prev =>
        prev mapValues {
          case UpToDate => WaitingForDownStreamAck
          case WaitingForDownStreamAck => Late
          case Late => Late
        }
      }
    )

  val promisesToTrigger = consumersTrigger.single.swap(Map.empty)
}
promisesToTrigger foreach { case (_, promise) =>
  promise.trySuccess(Some(outEvent))
}

def inStreamSink: Iteratee[I, Unit] = {
  processor >> effector &>> downstreamTrigger
}

def createOutStream(sinceId: PathId): Enumerator[O] = {
  val consumerId = java.util.UUID.randomUUID().toString
  consumersState.single transform (prev => prev + (consumerId -> Late))

  StreamProcessorHelper.unfoldPathId(sinceId) { currentId =>
    val longPollingTrigger = promise[Option[O]]
    consumersTrigger.single.getAndTransform(prev => prev + (consumerId -> longPollingTrigger))

    consumersState.single.get.apply(consumerId) match {
      case WaitingForDownStreamAck | UpToDate =>
        // we are up to date with upstream, no need to pull from source
        longPollingTrigger.future map { maybeEvent =>
          maybeEvent map (event => Some(currentId, Enumerator(event))) getOrElse Some(currentId, Enumerator.empty[O])
        }
      case Late =>
        StreamProcessorHelper.headOption(since(currentId)) flatMap {
          case None. _ =>
            // we don’t have anything more to poll
            longPollingTrigger.future map { maybeEvent =>
              maybeEvent map (event => Some(currentId, Enumerator(event))) getOrElse Some(currentId, Enumerator.empty[O])
            }
          case Some(event), remainingEnum =>
            // ignore promise
            consumersTrigger.single.getAndTransform(prev => prev - consumerId)
            // send next events
            Future.successful(Some(currentId, Enumerator(event) andThen remainingEnum))
        }
    }
  }
}

def updatePathId: Enumeratee[O, O] = StreamProcessorHelper.mapWithCounter { (outEvent, n) =>
  val oldId = eo.getId(outEvent)
  val newId = oldId.copy(path = oldId.path :+ n)
  eo.updateId(outEvent, newId)
}

Listing 28: Processor implementation

58
5.2. IMPLEMENTATION

```scala
abstract class StreamProcessorWithReplayableSource[I, O] {
  (implicit ec: ExecutionContext, ei: Evented[I], eo: Evented[O]) extends StreamProcessor[I, O] {
    val source: Source[I]
    def realtime(sinceId: PathId): Unit = {
      // restart the processor in case of failure
      def loopRestart(futIt: Future[Iteratee[I, Unit]]): Future[Iteratee[I, Unit]] = {
        futIt flatMap (_ =>
          getLastProcessedEventId() flatMap { lastId =>
            loopRestart(source.createOutStream(lastId) |>> inStreamSink)
          }
      }
      loopRestart(source.createOutStream(sinceId) |>> inStreamSink)
    }
    def start(): Future[Unit] = {
      for {
        sinceId <- getLastProcessedEventId()
      } yield realtime(sinceId)
    }
  }
}

abstract class SideEffectStreamProcessor[I, O] {
  (implicit ec: ExecutionContext, ei: Evented[I], eo: Evented[O]) extends StreamProcessorWithReplayableSource[I, O] {
    def since(id: PathId, included: Boolean = false): Enumerator[O] = {
      val ancestors = id.path.dropRight(1)
      val offset =
        if (id == PathId.min) 0
        else id.path.lastOption.getOrElse(0) + (if (included) 0 else 1)
      source.since(PathId(id.rootEvent, ancestors), included = true) &>
      processor &>
      Enumeratee.drop[O](offset)
    }
  }
}

abstract class PersistentStreamProcessor[I, O] {
  (implicit ec: ExecutionContext, ei: Evented[I], eo: Evented[O]) extends StreamProcessorWithReplayableSource[I, O] {
    def collection: JSONCollection
    def performAtomicSideEffect(event: O): Future[Unit] =
      collection.insert(eo.serialize(event)) map (_ => ())
  }
}

Listing 29: Persistent processor and side-effect processor implementation
```
CHAPTER 5. ARCHITECTURE AND IMPLEMENTATION OF THE JOURNAL
AND STREAM PROCESSING PART

5.2.4 Journal

The Journal extends the StreamProcessor class but it differs from other processors by its way of handling input. Indeed, the Journal has multiple pushers (that pull from various data sources) which push events concurrently. To be sure to insert events with an increasing ordered id, we use Concurrent.broadcast that provides a channel to push events. Pushed events will go into an unique sequential pipeline that first creates a PathId for the event, and then inserts this event with its PathId to MongoDB (this is done by the journalSink Iteratee). Listing 30 shows the code of the Journal.

5.2.5 Distribution

Implementing point-to-point distribution is very easy thanks to Play Framework’s helpers to transform an Enumerator into a HTTP Stream and a HTTP Stream into an Iteratee.

Basically, a parent processor can expose HTTP end-points for 2 functions: createOutStream that push the infinite stream, and since that allows child side-effect processors to ask for replay of past events. createOutStream can be mapped to an URL like /stream?pathId=<PATH_ID> where PathId is the start point of the stream (non-included). since can be mapped to /since?pathId=<PATH_ID>&included=<BOOLEAN> where PathId is the start point of the stream and included a boolean stating if we want the event corresponding to PathId to be replayed. Contrary to createOutStream, the since HTTP Stream is not infinite: it finishes when there is no more event to replay (no long-polling mechanism). Listing 31 shows the implementation of such a HTTP interface.

Concerning the child processor that has a remote parent, it must declare a remote source of type Source that can be transparently plugged into the StreamProcessor class. Listing 32 shows the implementation of a remote source. Composability of Enumerators / Iteratees makes the distribution very easy (almost location transparent). Moreover, as explained in the architecture part, this code maintains back-pressure with TCP-level ACK, which is very efficient.

5.2.6 Example application

The example application described in the architecture part has been implemented. The major part of the code is business specific and therefore will not be part of the report, but as an example Listing 33 shows the implementation of the FlatSnapshot side-effect processor.
5.2. IMPLEMENTATION

abstract class Journal[E](implicit ec: ExecutionContext, eo: Evented[E])
  extends StreamProcessor[E, E] {

  def collection: JSONCollection

  val (enum, channel) = Concurrent.broadcast[(E, Promise[E])]

  protected def journalSink: Iteratee[(E, Promise[E]), Unit] = {
    Enumeratee.mapM[(E, Promise[E])] { case (event, p) =>
      // generate new event id to ensure that event are inserted in ascending order
      val bsonid = BSONObjectID.generate
      val value = bsonid.value
      value(7) = Byte.MinValue // remove thread id to ensure sequentiality of events
      value(8) = Byte.MinValue
      val eventToInsert = eo.updateId(event, PathId(BSONObjectID(value).stringify))
      collection.insert(eo.serialize(eventToInsert)) map {
        _ =>
        p.trySuccess(eventToInsert)
      }
    } &>> downstreamTrigger
  }

  def start(): Future[Unit] = {
    enum |>> journalSink
    Future.successful()
  }

  def write(event: E): Future[E] = {
    val p = promise[E]
    channel.push(event, p)
    p.future
  }

  def writeSequentially(events: Seq[E]): Future[Option[E]] =
    Enumerator.enumerate(events) |>>> Iteratee.foldM(Option.empty[E]) { (
      _,
      event
    ) =>
      write(event) map {
        Some(_)
      }
    }

  def process(event: E): Enumerator[E] = Enumerator(event)
  override def inStreamSink = Iteratee.ignore // use journalSink instead
  override def updatePathId = Enumeratee.map(identity) // 1 to 1 event generation relationship

Listing 30: Journal implementation
object ProcessorHTTPInterface extends Controller {
  def stream(sinceId: String) = Action {
    val pathId = PathId.deserialize(sinceId)
    val enum = processor.createOutStream(pathId) &> Enumeratee.map(Json.toJson(_))
    Ok.chunked(enum)
  }

  def since(sinceId: String, included: Boolean) = Action {
    val pathId = PathId.deserialize(sinceId)
    val enum = processor.since(pathId, included) &> Enumeratee.map(Json.toJson(_))
    Ok.chunked(enum andThen Enumerator.eof)
  }
}

Listing 31: Distributed processors: HTTP interface of a parent processor
5.2. IMPLEMENTATION

class RemoteParentProcessor extends Source[ZEvent] {
    val remoteParentUrl = "http://localhost:9001"

    def createOutStream(sinceId: PathId): Enumerator[ZEvent] = {
        val serializedId = PathId.serialize(sinceId)
        val url = s"$remoteParentUrl/stream?sinceId=$serializedId"
        getHttpStream(url)
    }

    def since(sinceId: PathId, included: Boolean = false) = {
        val serializedId = PathId.serialize(sinceId)
        val url = s"$remoteParentUrl/since?sinceId=$serializedId&included=$included"
        getHttpStream(url)
    }

    private def getHttpStream(url: String): Enumerator[ZEvent] = {
        val (it, enum) = Concurrent.joined[Array[Byte]]
        WS.url(url).get(_ => it).flatMap(_.run)
        enum &> Enumeratee.mapInput {
            case Input.El(chunk) =>
                val tryJs = Try(Json.parse(chunk))
                tryJs match {
                    case Success(js) =>
                        js.validate[ZEvent] map { ev =>
                            Input.El(ev)
                        } getOrElse {
                            Input.Empty
                    }
                    case Failure(err) =>
                        logger.error("Failure during json parsing", err)
                        Input.Empty
                }
            case Input.Empty => Input.Empty
            case Input.EOF => Input.EOF
        }
    }

Listing 32: Distributed processors: Remote source implementation of a child processor

63
object FlatSnapshot extends SideEffectStreamProcessor[ZEvent, ZEvent] with FlatSnapshotQuery {
  def collection = ReactiveMongoPlugin.db.collection[JSONCollection]("flat_snapshots")

  val logger: LazyLogger = LazyLogger.apply("api.FlatSnapshot")

  val source = Snapshot

  def getLastProcessedEventId(): Future[PathId] = {
    collection.find(Json.obj(), Json.obj(
      "_lastUpdateByEvent" -> 1)).sort(Json.obj("_lastUpdateByEvent" -> -1)).one[JsObject].map(_.flatMap { json =>
        (json ".lastUpdateByEvent").asOpt[PathId]
      ).getOrElse(PathId.min))
  }

  def process(event: ZEvent): Enumerator[ZEvent] = Enumerator(event)

  def performAtomicSideEffect(event: ZEvent): Future[Unit] = {
    val snapshot = ZSnapshot(event)
    if (!((event.body ".archived").asOpt[Boolean].isEmpty) {
      FlatSnapshot.collection.remove(Json.obj("_id" -> snapshot.id)) map (_ => ()
    } else {
      FlatSnapshot.collection.save(ZSnapshot(event)) map (_ => ()
    }
  }
}

Listing 33: Implementation of the FlatSnapshot side-effect processor
Chapter 6

Performance evaluation

One of the goals of the performance evaluation is to validate the fact that the business application implemented with the generic stream processing library (see section 5.1.9) correctly reacts to an increasing push rate of events inserted in the Journal. From an initial push rate to a certain threshold, we expect the end-to-end latency\(^1\) to be constant as the dashboards have a processing speed higher than the push rate. However, after this threshold, we can expect the dashboards to start being late in the stream, so the end-to-end latency should increase. Moreover, we want to validate the fact that after a period of high push rate (implying that the dashboards are late in the stream), a period of low push rate enables the dashboards to reduce their lateness in the stream in order to catch up again with the real-time stream.

Furthermore, resource consumption should not exponentially increase when the push rate increases as we have used several techniques to optimize resource consumption in the system.

Concerning fault-tolerance of processors, we want to verify the fact that processors that have a persistent parent have a better recovery time than processors that have a side-effect parent (which implies to re-climb another level in the tree for the replay mechanism).

Last but not least, the end-to-end latency of events should be acceptable (i.e. maximum several seconds) for a push rate that corresponds to a real-world push rate for the business application (around 100 events per second).

Performance tests have been performed on the Journal and Stream Processing part of the business application on a local machine with a 2.2 GHz processor of 8 cores (so a parallelization factor of 8) with a SSD. The JVM is configured with a maximum Heap size of 2 GB. In order to perform those tests, an external Scala program has been developed to insert arbitrary events (of 1 Ko each) in the Journal with the possibility of varying the push rate.

\(^1\) The duration between the moment when an event is inserted in the Journal and the moment when the derived sub-events have updated the related Dashboard
CHAPTER 6. PERFORMANCE EVALUATION

6.1 Latency and resource consumption

First, we measure the end-to-end latency between the time when an event is inserted and the time when the resulting dashboard update(s) have been entirely performed. Figure 6.2 shows the plot of the average end-to-end latency between the Journal and a Dashboard when we increase the push rate of events inserted in the Journal. We notice that before a threshold of roughly 200 events inserted in the Journal per second, the latency between the Journal and a Dashboard is constant at 4 ms. This means that the processors lower in the tree structure (snapshot, flatSnapshot and the dashboards) can handle the push rate of the Journal and are not late in the stream (push-mode). However, after 200 events per second, the latency becomes linear with the push rate. This means that dashboards start to be late in the stream, and are forced to replay events at their rate because their processing time is too slow. Thus, the resultant plot is either constant (up-to-date child processors) or linear (late child processors), which is an expected and good result for scalability.

Another interesting way to look at these results is to compute the consumption rate of a dashboard, i.e. the inverse of the end-to-end latency. Figure ?? shows the plot computed with the data of Figure 6.2 by inverting the latency. We see that when the dashboards are up-to-date with the real-time stream (push rate from 0
6.1. LATENCY AND RESOURCE CONSUMPTION

Figure 6.2: Maximum consumption rate of a dashboard while varying the Journal push rate

![Graph showing maximum consumption rate vs push rate]

Figure 6.2: Maximum consumption rate of a dashboard while varying the Journal push rate

to 200), the maximum consumption rate is 250 events per second. However, this value decreases rapidly with a push rate higher than 200 events per second, which means that dashboards are late in the stream and don’t manage to reduce their lateness.

To better visualize the fact that low-level processors in the tree (dashboards) can catch up (or not) with the real-time stream speed, Figure 6.3, Figure 6.4 and Figure 6.5 show the latency of each event in sequential order for several push rates. We see that in Figure 6.3 (100 events / s), almost all events have a latency of 4 ms from the beginning of the test. The peaks of latency that happen sometimes are due to garbage collection. However we notice a higher and slightly larger peak at the beginning of the plot (corresponding to the first events). This is due to the initial warm-up of the JVM and the processors.

On Figure 6.4 (500 events / s), we see an initial large latency peak, and then the latency goes back to the normal 4 ms with a few latency peaks due to garbage collection. We notice that the initial latency peak is now larger than in the 100 events / s case. This is because the system needs more time to warm-up for a stream
at 500 events / s rather than 100 events / s, but then the dashboards manage to catch up with the real-time stream (as their processing speed is still lower than the push rate), and the latency goes back to normal (4 ms) after this warm-up time. However, in Figure 6.5 (1000 events / s), we notice that the latency goes on increasing. This is because dashboards do not manage to catch up with the real-time stream that is too fast compared to their processing speed, so they keep accumulating lateness compared to the real-time stream. One solution could be to distribute the dashboards on several machines to allow more resource to them. Moreover, in a real-world scenario, one can hope that moments with 1000 events / s are interleaved with moments where the push rate is slower so that the processors have time to reduce their lateness and eventually catch up with the real-time stream. This scenario is simulated in Figure 6.6: event 0 to 10000 are pushed with a push rate of 1000 events / s, and just after event 10000 to 20000 are pushed with a push rate of 100 events / s. We see that the processors manage to catch up with the real-time stream around event 12500 and then the latency remains constant at 4 ms.

Furthermore, during these performance tests, the resource consumption (JVM Heap space, number of threads) has been profiled. Figure 6.7 and Figure 6.8 show the JVM Heap space used and the number of threads used while increasing the event
6.2. FAULT-TOLERANCE

Figure 6.4: Latency of events between the Journal and a Dashboard with a push rate of 500 events per second

Concerning the number of threads used, we notice that this number is constant (76) independently of the push rate, which is expected with our non-blocking IO model to avoid too many thread creations and context switching. Concerning the JVM heap space consumption, we notice that the value increases in a less than linear fashion with the push rate (the plot has a logarithmic shape). These two plots validate the fact that the platform uses an amount of resources that does not increase too much with the load (optimization of resource consumption).

6.2 Fault-tolerance

In this part, we define a performance test to measure the recovery time of a processor that recovers from a crash and must replay 1000 events that happened when it was crashed. Figure 6.9 shows the resulting bar chart. Using the tree structure of the business use case application presented in section 5.1.9, the Snapshot processor is first killed and then restarted. Its parent is the Journal, a persistent processor, so it has only one level to climb in the tree to replay the stream. Then, FlatSnapshot is killed. It is at level 2 in the tree, but its parent is a persistent processor (Snapshot),
so it has also only one level in the tree to climb to replay the stream. As a result, its replay time is roughly the same than Snapshot (the processing time of these two processors is equivalent). Last but not least, a Dashboard is killed. As its parent is a side-effect processor (FlatSnapshot), it has to climb 2 levels to replay the stream (until reaching the Snapshot persistent processor). Moreover, the processing time of a dashboard is slightly superior than other processors. As a result of these two factors (side-effect parent and slightly longer processing time), we see that Dashboard takes more time to replay the 1000 events that it missed, which is expected according to our model.

### 6.3 Conclusion

To conclude, the performance tests meet the goals with expected latency patterns (the end-to-end event latency is constant until a push rate threshold, and then it increases linearly with the push rate because the dashboards start to be late in the stream, but they can reduce their lateness if the push rate decreases after a
6.3. CONCLUSION

Figure 6.6: Latency of events between the Journal and a Dashboard with variable push rates

period of high push rate). Moreover, resource consumption does not increase too much with the push rate. Furthermore, the fault-tolerance behavior of processors follows the expected patterns with a lower recovery time for processor that have a persistent parent. Last but not least, for a push rate of 100 events per seconds (which corresponds to the common push rate for the business application), the end-to-end latency is constant at 4 ms on a local machine, which is a very good result for our use case.
CHAPTER 6. PERFORMANCE EVALUATION

Figure 6.7: JVM heap space consumption while varying the Journal push rate

Figure 6.8: Number of threads used while varying the Journal push rate
Figure 6.9: Recovery time of processors for a replay of 1000 events
Chapter 7

Conclusion and future work

7.1 Conclusion

The platform meets the requirements specified in chapter 2. Concerning functional requirements, the Data Integration part defines an architecture that performs incremental pulls of data changes from various data sources’ REST APIs and inserts these data changes as events into the Journal. The Stream Processing part defines a generic architecture with a Tree-like structure of stream processors that can react to events and perform side-effects with an exactly-once semantic. In the whole platform, fault-tolerance mechanisms ensure no event loss and no event duplication even in cases of transient failures of pullers and/or processors.

Concerning non-functional requirements, the Data Integration part uses an actor system to provide easy scale up and scale out of data pullers. The Stream Processing part uses an adaptive push-pull model with back-pressure to allow decoupled stream processing while optimizing resource consumption. The tree of processors guarantees in-order sequential asynchronous stream processing with fault-tolerance: the temporary failure of a processor is guaranteed without message loss or duplication by allowing a processor to replay the stream from its parent.

To conclude, this thesis has presented a Reactive platform for Data Integration and Event Stream Processing. Events and Event-Sourcing are the core concepts of the platform to enable real-time propagation of data changes across the system, from external data sources to data aggregation dashboards. Data sources are pulled in a parallel and incremental way via their REST API to create a real-time flow of events that are inserted in an event-sourced Journal. Saving not only the current state of the data but also all the data changes that led to this current state allows to be able to query and replay the history of the data, which can be useful for business purposes but also for technical purposes such as fault-tolerance or stream processing with subscribers that have heterogeneous processing speeds. From the events stored in the Journal, a tree of stream processors can be defined to subscribe to the data changes and react in various ways. An example use case of a processor
is maintaining a specific data aggregation dashboard that is updated incrementally upon the reception of certain types of events in real-time. Strong technical guarantees are ensured by the platform as in-order sequential asynchronous processing with exactly-once side-effect semantic even in case of failures. Under the hood, a complex adaptive push-pull model with back-pressure is used to maximize the performance of the system and minimize the amount of resource (CPU, RAM) used. Functional programming abstractions have been used for maintainable and composable asynchronous programming. Performance tests have been performed on a use case business application, which validates the architecture model by showing expected performance patterns concerning event latency between the Journal (top of the processing tree) and the Dashboards (leaves of the processing tree), and expected fault-tolerance behaviors with acceptable recovery times.

7.2 Future work

Concerning Future Works that can be made on the platform, one problem is that the distributed mode is for now point-to-point oriented, i.e. we need to manually configure which component (data puller, processor) is located on which machine. To obtain elastic scalability, it would be interesting to use a cluster oriented approach with a cluster manager layer that automatically puts components on machines according to the current resource availability of each machine.

Moreover, it would be interesting to allow customizable parallelization of event processing inside a processor via partitions as in Apache Kafka. A partition is a sequence of events that needs to be process sequentially. The current stream processing system can be thought to have only one partition. However, by defining multiple partitions, events that do not need to be processed sequentially could be processed in parallel inside a stream processor.

Another interesting future work is extending the tree structure of stream processors to a DAG structure. Such extension would of course complicate the replay mechanism of streams as a processor could have several parents.

Last but not least, even if data storage systems become larger and cheaper, there is a time when the Journal and the local journals will grow too much. An event compaction mechanism could be implemented to deal with this issue. It could be naive by just removing events that are X day old, but it could also keep track of keyed data (resource) to compact events in a smart way (for example, if several updates have happened on a same resource during one day, it could merge these updates to create only one event).
Bibliography


BIBLIOGRAPHY


