DADSTORM

Márcio Santos, Diogo Ferreira, Francisco Santos
76338, 79018, 79719
Group 24
Instituto Superior Técnico
Av. Rovisco Pais, 1, 1049-001 Lisboa, Portugal
marcio.santos@ist.utl.pt, diogo.lopes.ferreira@ist.utl.pt, franciscopolaco@ist.utl.pt

Abstract

Stream processing as become popular, so it is normal that there is a lot of technologies out there dealing with it, like Apache Spark. The scope of our work is to design and implement a simplified distributed fault-tolerant stream processing system, for education purposes only. In our system, the programmer can, very easily, write a configuration file and some custom functions, to give his program the correct semantic while running on top of our platform.

1. Introduction

Stream processing consist in given a sequence of data (tuples in our case) and apply a set of operations to it. When we take this to a distributed level other problems, like crashes, can appear, so it is required to ensure the correctness of the computation by using algorithms that apply fault-tolerant semantics.

Some concepts that are needed to understand our work are explain in the subsections. To facilitate the apprehension we will apply a metaphor, in which we will pretend that we have a factory, where we receive a raw material, work on that and produce something new.

1.1 Replica

A **Replica** or as we called them in the **Design** section, a **Slave**, is the most basic unit of our system. It is manly responsible to execute an operations over a set of tuples. They also are used to ensure replication and fault-tolerant properties. In our metaphor, this concept are mapped as assembly lines, where every line does the same, and if some stops, the others assure that the work is done.

1.2 Operator

An **Operator** is a set of **Replicas** that do the same operation. This set is responsible to tolerate faults within it self. In our metaphor, we mapped this concept, as the factory it self, where it receives tuples, process them and produce new tuples.

2. Design

One of the key aspects of every solution is its design, especially in distributed systems. So, having a good design was a top priority for us. Our design provides a highly flexible and scalable environment, which adapts to every need.

First, we will discuss the components of the system, their roles and modules. Then we will dive into the interactions between them, present the most relevant architecture and finish with a brief overview.

2.1. Components

Our system is divided in four main components, which are the Puppet Master, Process Creation Service, Slave and Common Types.

2.1.1 Roles

Puppet Master: reads the configuration file, processes it and sends the needed commands to the Process Creation Service of each machine. After that, it will only collect logs from all replicas and serve as a command prompt. It basically is a graphical interface where we can easily bootstrap, monitor and manipulate the system.

Process Creation Service (PCS): responsible for ordering the creation of the slaves, i.e. the replicas and for feeding the first tuples into the first operator. In some sense, the process creation service is just a simple parser.

Slave: in charge of the actual processing of the tuples. The processing happens in three, totally decoupled, phases: importing, processing and routing.

Common Types: a shared library between the different components. Which, defines the interfaces and specialized objects for data transfers.

2.1.2 Modules

In this section we will approach which modules composed the system.

The Puppet Master has the following modules:

Logger: logs the events, received from the slaves, by writing it to the screen and to a file.

Operator & Proxies: the operator is a composition of proxies. Each proxy is a remote object of a slave, which we use to send commands, asynchronously, after their creation.

Configuration File Processor: reads the configuration file and parses it into chunks of information which we send to the PCS.

The Slave has the following modules:

Importer: source of the tuples. We can distinguish three importing types: Input, File and Operator imports. The difference between Input and File imports is that the first is only used for the first operator of the chain, since its data comes from the PCS. This decision emerged because of the need to be sure that all the tuples were imported in the presence of a random importing policy.

The Operator import represents an operator that receives its input from a upstream operator.

Processor: given a tuple applies the domain logic. There are two types of processors: the stateless and stateful. The stateless, are composed by the Dup, Filter and Custom. The stateful, have the Uniq and Count. Its logic is described in the project description.

Routing: routes the processed tuple. There are four different types of routing: Primary, Random, Hashing and Output. The first three obey the domain logic. The last one came from the necessity of outputting the resulting tuples, in the last operator, into a file.

Note that the PCS and Common Types are not composed of modules, due to their simplicity.

2.2. Interactions

The flow begins with the Puppet Master when he loads the configuration file. After loading it, the Puppet Master orders the PCS to create the operators and their respective Slaves. When the PCS finishes, the Puppet Master starts sending commands to the Slaves. The Slaves, according to the required characteristics of the system, do their work in a pipe-and-filter manner, notifying the Puppet Master Logger when they finish work or when they are asked to.

2.3. Architecture

In the following section, we expose the qualities of our solution. Note that the diagrams presented are not complete attribute or operation wise for simplicity.

As we talked before, the importing, processing and routing are totally decoupled. In order to achieve that, we used a **Abstract Factory Pattern**, which allows us to create different specialized factories to fit our needs. So to create a Slave we just need to "ask" each factory for a concrete type, forming any combination of Slaves that we would like. This approach gives us **plenty of flexibility**, as shown in **fig. 1**. As an example, imagine that you want to create a Slave that imports from another operator, processes with uniq and routes for a file. You just have to tell the importing factory to give you a OpImport, the processing factory to give you a Uniq and finally the routing factory to give you a Output. Then you just need to create the Slave with those.

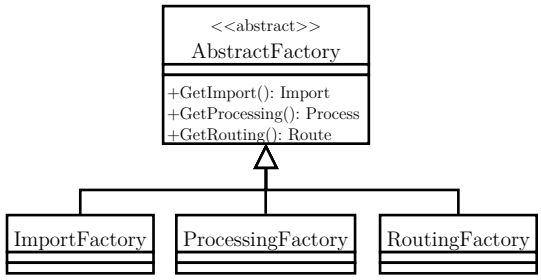

Figure 1: The different factories used to create the slaves.

Another important aspect of our design is how the Slave operates under different states. As we know, there are, currently, on the project the Froze and Unfrozen states. Depending on the state, the Slave should behave differently. To achieve such aim dynamically and in a scalable fashion, we used the **State Pattern fig. 4**. With this approach, we do not need to worry about the logic of the operations, since its behaviour is decided on runtime.

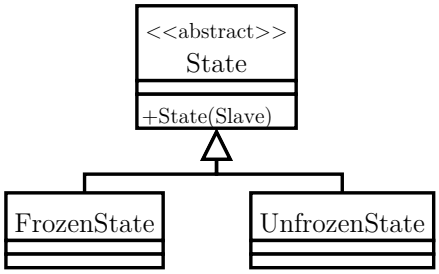

Figure 2: The different states.

To finish, is worth mentioning that in the Puppet Master, the Logger being a separated module, unburdens the Puppet Master of that "tedious job", allowing the PM to process the

commands given to him, i.e. without freezing the UI, which accomplishes the desired interactive model - command line.

2.4. Overview

At the beginning of this section we stated that our design provided "a highly flexible and scalable environment".

How do we achieve flexibility? Looking at the Architecture is easy to find this quality, the Slave creation example shows it clearly.

How do we achieve scalability[1] **?** Well, we can obviously see that adding anything new to the domain of the problem is trivial. Imagine that you want to add a Byzantine State, you simply inherit State and when the Puppet Master requires the Slave to switch the state there is no modification of code, besides creating a new state. This reasoning also, applies to Importing, Processing and Routing types.

3. Solution

This section is divided in three logical partitions, which obey the fault tolerance semantics: at-most-once, at-least-once and exactly-once. Notice that having at-most-once grants a basic solution to the problem and that the other semantics are just increments of the preceding one. Also, we will not be specific about all the operations on the domain, we will try to focus on the important things. The following glossary will help us to reach a common ground:

Dispatch: orders a Slave to process the tuple that he received as input.

Froze/Unfroze: changes the state to Froze/Unfroze respectively.

ReplicaUpdate: sends the processed tuples from the Slave to the Logger.

TuplePack: a data structure, present in Common Types, which carries the sender (Slave URL), a seq id (identifies the pack) and a list of tuples (content). We have an abusive terminology, since sometimes we say tuple and others TuplePack, but in this context they mean the same.

3.1. At-Most-Once

This semantic implies that we process a tuple $0..1$ times. Doing such endeavour, gives us the base solution to our problem.

The flow of information and roles between the PM and PCS were well explained in the previous section. So lets assume, that we are already in the first operator. This always imports from a file - in our case the PCS already distributed the tuples among its Slaves.

When, the PM calls start on a Operator, every Slave starts by calling the dispatch method. First it will try to Import according to its characteristics, it may get its input from the PCS (first operator), file or upstream operator. In the first two cases, if the file was not empty, they will import its content, create a TuplePack[2] for each of lines read, Process, Route and ReplicaUpdate. In the last case, we do nothing until a upstream operator calls dispatch on us.

After the start call, in a regular operator, depending on the Slave state Froze or Unfroze, we will add the job to a queue, sleep for some time[3] or we will start doing work. A regular operator follows the same work sequence as a first one.

The Route happens in a totally asynchronously manner, because of the semantic's weakness.

Whatever happens to the tuples in the downstream chain does not matter, since we already achieved the semantic and therefore a basic solution.

3.2. At-Least-Once

We need to guarantee that a tuple is processed $1..*$ times. To reach this semantic we expanded on the previous one, adding to the asynchronously Routing a Callback. When we route, we basically leave a thread that is responsible for the "faith" of that evocation and we continue with our normal operation. As soon as the call finishes, the thread wakes up to find its result. If we find any kind of problem[4] it means that it failed, so we try again until we tried every Slave on the downstream.

This guarantees that a tuple reaches at least one Slave in the downstream, therefore matching the requirements needed to this semantic.

3.3. Exactly-Once

We want to assure that a tuple only gets processed 1 time. Definitely the hardest of them all, because we need need to synchronize the state between the stateful processing Slaves, inside the same Operator. Race conditions will arise due to the nature of the system (parallel processing system). So, solving this conditions is key to match the requirements of the semantic.

To solve such competition for "resources", we need to restrict the access to shared resources (TuplePacks), avoid duplicates on the chain resulting from frozen Slaves becoming unfrozen and share state between the stateful replicas in the same operator.

[1] Scalability of code not performance.

[2] Temporary, because we always create a TuplePack before Route, but since Process has the same signature for a first and regular operator we need it as a dummy.

[3] Give the impression that we are experiencing some difficulties.

[4] Exception, in our case SocketException is a crash and SlowException is a Slave that is experiencing problems - frozen.

To solve the first problem we devised a simple solution, which only allows one Slave to do work at a given time - using locks. So, when the dispatch is called in a Slave, the first thing that he does is gather responses by *TryToPurpose* a input (TuplePack) to all of his siblings[5] . We can only advance to dispatch a tuple if every sibling agrees with the input, it can happen that a Slave has no responses, either because he has no siblings or all are dead/frozen, and in that case he can dispatch. If in that list of responses any of the siblings disagrees signifies that the sibling is doing a dispatching and he is not ready to respond until it finishes. Since the system is multi threaded, if one of our siblings informs that he already processed the tuple we can abandon that proposal.

After a successful proposal, we start dispatching the input. A Slave starts by asking his siblings if he can Process, *MayIProcess*, which basically asks in a tentative way with a random back-off, to every sibling if he has already seen the TuplePack. To break out of this cycle we need to fulfill one of the two conditions: a sibling has already seen the TuplePack (abandon the input) or the number of "accept" responses that we got are equal to the number of siblings minus the crashed ones (we can advance to process). As you probably figured it out, if we have a sibling frozen, it will try and try until he can contact it. Worth mentioning that this differs from the previous reasoning, since here we want to eliminate duplicates and in the one before we wanted to "cut off" race conditions.

In order to solve the last problem, we need to share state. Our solution was based on, making sure that after a tuple gets processed, pseudo-routed[6] and added to the list of seen tuples, was delivered to all the other siblings by, *DestributeTuple*. So, for each sibling alive, i.e. not crashed, we *AnnounceTuple* and make sure that he receives it. Once a sibling collects an announced tuple it checks if he have not seen it, if he did not he adds it to the list of seen tuples in order to eliminate future duplicates. In addition, a stateful Slave will also process it to synchronize his state with his siblings. This approach also helps with duplicated tuples.

4. Results

With this solution we acquired a system capable of doing almost everything proposed, but given the particularities of a failover distributed system we failed in some aspects - if we consider our perspective of a world where everything fails.

One of the biggest flaws of our project was assuming that only a Slave, from the same operator, is dispatching at the same time. We did not consider the case, where a Slave may acquire a positive response of his siblings and loose thread context. So imagine that, at the same time, two siblings try to propose an input. Considering a multi-threaded system, the first proposer after acquiring a positive response of all his siblings looses context, since he does not have a local lock to the dispatch we will agree with the second proposer. When the initial proposer thread recuperates, it will also acquire its local lock and start dispatching, resulting in a race condition if both of them have the same tuple.

There are other flaws, but in systems like these is hard to tell which exist, since each run of the system is different from one another. Of course that, the previous flaw will affect the system and in the right conditions will lead to a incoherence. It is also worth mentioning that, our approach also lacks performance, since we stall for every decision.

5. Implementation Notes

Our solution was developed in C#. We used remoting to establish the communication between the PM-PCS and PM-Operators. The asynchronous routing was achieved with a delegate with a callback. The synchronous access to shared resources inside the same slave, was accomplished with regular lock primitives.

6 Evaluation

We setup some configuration files in order to test our system. Each test has a specific focus, for instance we evaluate if a stream reach the end even when a replica crashes.

All the following tests were ran in the three possible semantics, however we focus more on the at-least-once semantic, since is not trivial to implement and on the exactly-once semantic, because is the most difficult to implement. Note that at-most-once semantic does not even assure that a tuple will reach the end, or that the operators have their state synchronized, so it will not be shown in detail like the others.

Also, before reaching to the critical point, we assured that all tested had tuples that would make the operator do something instead of a trivial case, where it would do nothing.

6.1 Environment Setup

Since we want people to be able to reproduce these tests, we present our environment setup. Every test was ran locally, since we did not had the resources to have various Windows machines in the same network.

Our environment was:

- Windows 10 Pro x64
- Intel Core i7-6700HQ @ 2.60GHz

[5]The other Slaves of the same Operator.

[6]Since we promote a asynchronously routing with callback.

- 16 GB DDR4 RAM
- Microsoft Visual Studio Enterprise 2015
- Microsoft .NET Framework 4.5.2

6.2 State Synchronization

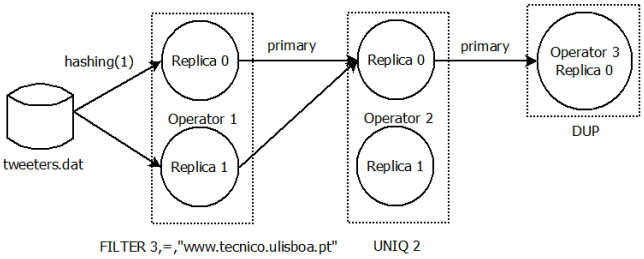

Figure 3: State Synchronization Test

In this test, we hoped to verify if the state was being synchronized in the second operator since we froze replica 0 and some 7 seconds after we unfreeze it.

This test does not have any semantic per se, however the objective was to check if there was any tuple in the output with same user.

6.2.1 at-least-once semantic

All the tuples reach the end, sometimes in duplicated. This happens because we send the tuples to replica 0 of operator 2, but it is frozen. Since operator 1 does not receive an *ACK*, it resends to other replica. Later, when replica 0 is unfrozen, it will process everything, which will result in duplicated tuples.

6.2.2 exactly-once semantic

Here the output will be slower, as it is expected, since siblings are asked if a tuple was seen. Nevertheless, even with the frozen period outputs what should output if no failure had occur.

6.3 Example in the project description

In this test, we wanted to check if we were able to keep the semantics with more operators and by crashing a replica. This test should output the number of user that had contact with a post related with *"www.tecnico.ulisboa.pt"*.

6.3.1 at-least-once semantic

Again all the tuples reach the end, sometimes in duplicated.

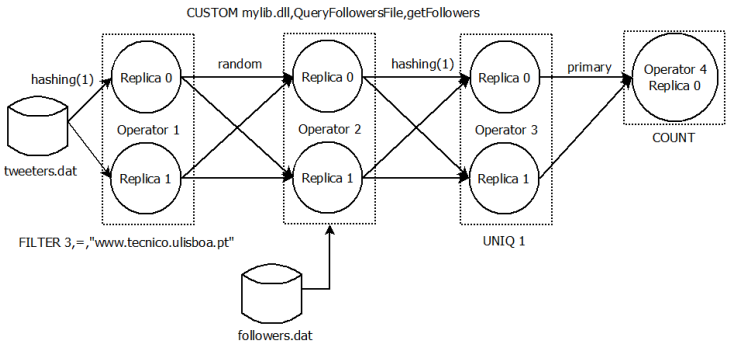

Figure 4: Crash Test

6.3.2 exactly-once semantic

Every tuple reach the end, as if crashes or freezes had not occur.

7 Conclusions

Our work provides a framework for distributed stream processing that any programmer can use to solve his needs. He can take advantage of its simplicity and guaranties to build his applications on top of Dadstorm.