DADSTORM

Desenvolvimento de Aplicações Distribuídas
Project - 2016-17 (IST/DAD): MEIC-A / MEIC-T / METI

September 27, 2016

Abstract

The DAD project aims at implementing a simplified (and therefore far from complete) implementation of a fault-tolerant real-time distributed stream processing system.

1 Introduction

The goal of this project is to design and implement **DADSTORM**, a simplified implementation of a fault-tolerant real-time distributed stream processing system. A stream processing application is composed of a set of operators, each transforming a stream of tuples that is submitted to the operator's input and that is output by the operator in its processed form. The stream engine will support a limited set of operators, but will have to provide strong guarantees on the correctness of the distributed computation and on performance predictability in presence of failures.

Tuples are records composed of an arbitrary number of ordered fields, each one of them being, for simplicity, a string.

The project should be implemented using C# and .Net Remoting using Microsoft Visual Studio and the CLR runtime.

2 System Architecture and Specification of Distributed Computations

We consider a distributed system where operators may execute on different machines — although, for simplicity's sake, multiple operators may be deployed within the same physical machine. In **DADSTORM**, a distributed computation can be expressed as a direct acyclic graph, whose vertexes consist of operators that operate on a stream of input data and may emit, for each input tuple, a set of corresponding output tuples. Operators can consume input data either from files or be fed with the output data produced by other operators, which allows for expressing arbitrary computations, as long as the resulting topology/graph remains acyclic.

To simplify the project, we assume that there is a configuration file that describes the entire set of operators that take part in the distributed computation and on which machines they are deployed.

The configuration files is composed by lines having the following format:

operator_id INPUT_OPS *source_op_id*$_1$|*filepath*$_1$,...,*source_op_id*$_n$|*filepath*$_n$
REP_FACT *repl_factor* ROUTING *primary*|*hashing*|*random*
ADDRESS *URL*$_1$,...,*URL*$_{\text{REP\_FACT}}$
OPERATOR_SPEC *operator_type operator_param_1*,...,*operator_param_n*

where:

- The *operator_id* parameter is an integer identifying the operator.
- The INPUT_OPS parameter specifies a list of operator identifiers that provide input tuples for the current operator or paths to files from which the operator should read its input tuples.
- The REP_FACT parameter indicates how many replicas of this operator will be activated.
- The ROUTING policy parameter specifies how the input tuples consumed by the operator should be distributed among its replicas. It has three possible values whose semantics are described below.
- The ADDRESS parameter specifies a list of all the replicas URLs. These URLs are in the format `tcp://<machine-ip>:<port>/op`. The port can be freely chosen (among the ones available on the local machine) with the exception of port number 10000 and 10001, which are reserved for the PuppetMaster and for the Process Creation Service (PCS) — see Section 4.
- The OPERATOR_SPEC parameter indicates the name of the tuple transformation being performed at this operator (see below) and it is followed by a list of the input parameters required by the operator.

2.1 Tuple Routing

The **DADSTORM** system shall implement three different policies for tuple routing from one operator to another:

- *Primary*: tuples are output to the primary replica of the operator it is connected to downstream.
- *Random*: tuples are output to a random replica of the downstream operator.
- *Hashing*(*field_id*): tuples are forwarded to a replica of the downstream operator according to the value of a hashing function computed on the tuple's field with identifier *field_id*.

Note that in absence of replication, i.e., REP_FACT=1, and in absence of fault-tolerance mechanisms, i.e., for the checkpoint's submission, all routing policies essentially boil down to *primary*.

2.2 Operators

The **DADSTORM** system shall support the following set of operators:

- *UNIQ field_number*: emit the tuple again if field_number field is unique.
- *COUNT*: emit the number of seen tuples.
- *DUP*: emit the tuple as is in input.
- *FILTER field_number, condition, value*: emit the input tuple if field_number is larger(">"), smaller("<") or equal("=") than value.

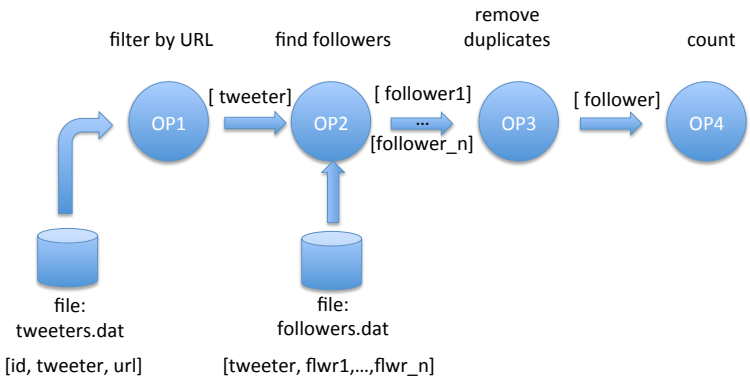

Figure 1: Logical view of the distributed computation for the reach of a URL in Tweeter.

- *CUSTOM dll ,class, method*: this command send each tuple it receives, in the form of a list of strings, to a custom *method* of a *class* within a specific class library, *dll* and outputs the tuples returned by the *method*.

2.3 Example

Below we report an example configuration file, which can be used to express the computation of the *reach* of a URL on Twitter. The reach of a URL is the number of unique people exposed to a URL, say "www.tecnico.ulisboa.pt", on Twitter. To compute reach, it is necessary to:

1. Get the people who tweeted about some URL. This is done by reading the file *tweeters.dat* containing tuples [id, tweeter, url] and by emitting, for each tuple in input, tuples of the format [tweeter] if url is equal to "www.tecnico.ulisboa.pt"
2. Find their followers. For each tweeter in the [tweeter] input stream search in the file *followers.dat* the set of followers of this tweeter and emit the set of corresponding [follower] tuples.
3. Unique the set of followers, by removing duplicates which occur if someone follows multiple people who tweeted the same URL.
4. Count the total number of followers emitted by the operator in the previous stage.

The diagram in Figure 1 illustrates the logical organization of the set of operators in a linear topology, highlighting the flows of data among them and from external data sources, i.e., files.

The file below specifies the corresponding distributed computation and how it is physically mapped to a set of 7 machines. This is also illustrated in Figure 2, which also depicts tuples are meant to be routed among the various replicas of each operator, according to the specifications provided in the example configuration file.

OP1 INPUT_OPS *tweeters.data*
REP_FACT *2* ROUTING *hashing*(1)
ADDRESS `tcp://1.2.3.4:11000/op`, `tcp://1.2.3.5:11000/op`
OPERATOR_SPEC *FILTER 3*,"=","www.tecnico.ulisboa.pt"

OP2 INPUT_OPS *OP1*

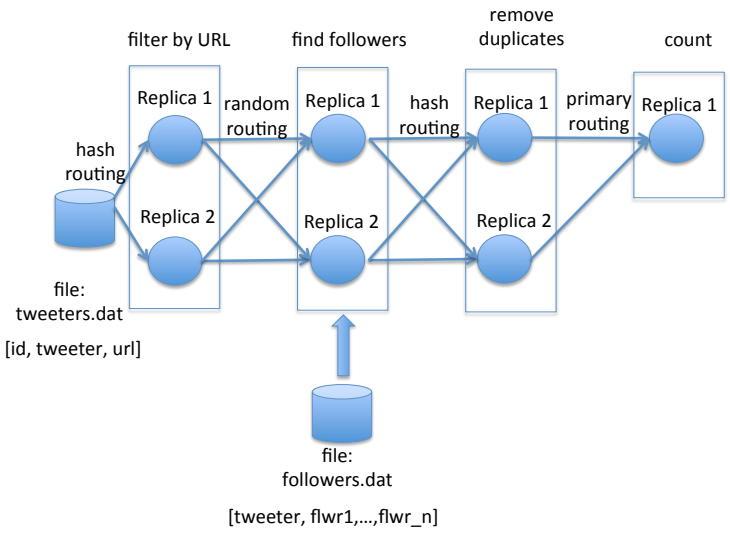

Figure 2: Physical mapping of the distributed computation for the reach of a URL in Tweeter according to the specification included in the example configuration file.

REP_FACT *2* ROUTING *random*
ADDRESS `tcp://1.2.3.6:11000/op`, `tcp://1.2.3.6:11001/op`
OPERATOR_SPEC *CUSTOM* "mylib.dll","QueryFollowersFile","getFollowers"

OP3 INPUT_OPS *OP2*
REP_FACT *2* ROUTING *hashing*(1)
ADDRESS `tcp://1.2.3.8:11000/op`, `tcp://1.2.3.9:11000/op`
OPERATOR_SPEC *UNIQ 1*

OP4 INPUT_OPS *OP3*
REP_FACT *1* ROUTING *primary*
ADDRESS `tcp://1.2.3.10:11000/op`
OPERATOR_SPEC *COUNT*

3 Fault-Tolerant Capabilities

In a distributed stream processing system, such as the one described so far, it is of paramount importance that the occurrence of failures does not compromise the correctness of the distributed computation. If such system is used for real-time purposes, it is also desirable that performance remains predictable despite the occurrence of failures, e.g., by minimizing the amount of data that has to be reprocessed if a subset of machines in the system fail and by enhancing data availability via replication techniques.

In case of failures, the system should reconfigure automatically, without disrupting the processing of the stream of tuples. For instance, if a downstream operator crashes, the upstream operator should automatically reconfigure its routing strategy so that he propagates

its results only to available replicas of the downstream operator, and no tuples should be lost.

The students are free to use the replication technique they feel more appropriate to solve the problem at hand. Depending on the selected fault-tolerance strategy, the students shall state how many faulty replicas, *f*, can be tolerated out of the total number of available ones, *replication_factor*, as well as what are the assumptions on the synchrony model (e.g., synchronous, partially synchronous, asynchronous) of the system.

3.1 Tuple Processing Semantics

The system should support three different guarantees regarding the semantics of processing of each input tuple in presence of possible failures:

- **At most once**: Tuples may or be not processed, i.e., some tuples may be lost in presence of operator failures. E.g., all the tuples sent towards an operator downstream may be lost if this is unavailable.
- **At least once**: Tuples are guaranteed to be processed at least once, i.e., no tuple in input is missed. Yet, it cannot be excluded that an operator processes twice a tuple, e.g., once before crashing and a second time upon recovering.
- **Exactly once**: Tuples are guaranteed to be processed exactly once. This also ensures exactly once semantics not only for the update of the internal state of the operators involved in the processing, but also for the execution of any non-idempotent action (e.g., interaction with external devices like DBMSs or storage systems) issued by the operators during the processing of a tuple.

The tuple processing semantics for the entire computation are specified in the configuration file in a line with the following syntax:

`Semantics` *at-most-once*|*at-least-once*|*exactly-once*

Depending on the type of operator, and on the requested execution semantics, replicas of an operator may have to agree on/decide:

- Order of input events.
- Whether an input event has already been processed.
- How long it is necessary to buffer output events.

The goal is to exploit the benefits of replication while avoiding having to reprocess large number of events in case of faults of nodes.

4 PuppetMaster

To simplify project testing, all nodes will also connect to a centralised *PuppetMaster*. The role of the PuppetMaster process is to provide a single console from where it is possible to control experiments. Each physical machine used in the system will also execute a process, called PCS (Process Creation Service), which the PuppetMaster can use to launch operator processes on remote machines. For simplicity, the activation of the PuppetMasters and of the process creation service will be performed manually. It is the PuppetMaster that reads the system configuration file and starts all the relevant processes. The process creation services on each machine should expose a service at an URL on port 10000 for requesting the creation of node replicas on the local machine. This service can be used by the PuppetMaster to create new replicas. For simplicity, we assume that the PuppetMaster know the URLs of

the entire set of process creation services. This information can be provided, for instance, via configuration file or command line. The PuppetMaster can send the following commands to the stream processing nodes:

- `Start` *operator_id*. Tells the *operator_id* operator to start processing tuples.
- `Interval` *operator_id x_ms*. The operator should sleep x milliseconds between two consecutive events.
- `Status`: This command makes all nodes in the system print their current status. The status command should present brief information about the state of the system (who is present, which nodes are presumed failed, etc...). Status information can be printed on each nodes' console and does not need to be centralized at the PuppetMaster.

Additionally, the PuppetMaster may also send to the replicas debugging commands:

- `Crash` *processname*. This command is used to force a process to crash.
- `Freeze` *processname*. This command is used to simulate a delay in the process. After receiving a freeze, the process continues receiving messages but stops processing them until the PuppetMaster "unfreezes" it.
- `Unfreeze` *processname*. This command is used to put a process back to normal operation. Pending messages that were received while the process was frozen, should be processed when this command is received.

The PuppetMaster should have a simple console where an human operator may type the commands above, when running experiments with the system. Also, to further automate testing, the PuppetMaster can also read a sequence of such commands from a *script* file. A script file can have an additional command that controls the behaviour of the PuppetMaster itself:

- `Wait` *x_ms*. This command instructs the pupper master to sleep for x milliseconds before reading and executing the following command in the script file.

For instance, the following sequence in a script file will force broker *broker0* to freeze to $100ms$:

`Freeze` *operator_url*
`Wait` *100*
`Unfreeze` *operator_url*

All PuppetMaster commands should be executed asynchronously except for the `Wait` command. The PuppetMaster should produce a time ordered log of all events it triggers or observes. All events triggered by the PuppetMaster should appear in the log with the same syntax as in the script files. The operators should notify the PuppetMaster of every tuple they output.

This should appear in the PuppetMaster log as:
`tuple` *replica_URL*, $< list - of - tuple - fields >$

There are two levels of logging, light and full. The logging level to be used by the **DADSTORM** system is defined in the configuration file, in a line with the format:

`LoggingLevel` *full|light*

If the line is missing, the system should default to light.

In the full logging mode, all tuple emissions should be included in the log. In the light logging mode, tuple emission by the operators need not be included in the log and therefore the operators need not notify the PuppetMaster of those events.

The port 10001 is reserved for the PuppetMaster and can be used to expose a service that gathers incoming events from the various operators for logging purposes.

5 Final report

Students should prepare a final report describing the developed solution (max. 6 pages). In this report, students should follow the typical approach of a technical paper, first describing the problem they are going to solve, the proposed solutions, and the relative advantages of each solution. However, please avoid including in the report any information included in this project description. The report should include an explanation of the algorithms used and justifications for the design decisions. The project's final report should also include some qualitative and quantitative evaluation of the implementation. The quantitative evaluation should focus on the following metrics:

- Routing of events.
- Fault-tolerance features.
- Other optimisations.

This should motivate a brief discussion on the overall quality of the protocols developed. The final reports should be written using LaTeX. A template of the paper format will be provided to the students.

6 Checkpoint and Final Submission

The evaluation process includes an intermediate optional checkpoint and a mandatory final submission. For the checkpoint the students may submit a preliminary implementation of the project; if they do so, they may gain a bonus in the final grade. The goal of the checkpoint is to control the evolution of the implementation effort. Given that students are expected to perform an experimental evaluation of the prototype, it is desirable that they have a working version by the checkpoint time. In contrast to the final evaluation, in the checkpoint only the functionality of the project will be evaluated and not the quality of the solution.

For the checkpoint, students should implement the entire base system, excluding the fault-tolerance mechanisms and the semantics other than "at-most-once". After the checkpoint, the students will have time to perform the experimental evaluation and to fix any bugs detected during the checkpoint. The final submission should include the source code (in electronic format) and the associated report (max. 6 pages). The project *must* run in the Lab's PCs for the final demonstration.

7 Relevant Dates

TODO: CONFIRM THE DATES

- November 11^{th} - Electronic submission of the checkpoint code;
- November 14^{th} to November 18^{th} - Checkpoint evaluation;

- December 9^{th} - Electronic submission of the final code.
- December 12^{th} - Electronic submission of the final report.

8 Grading

A perfect project without any of the fault-tolerant features will receive 15 points out of 20. The fault-tolerant features are worth 5 additional points for a total of 20 points.

The project grading will depend on a discussion at the end of the semester where all members of the groups must be present and where individual grades will be determined. That grade will depend on, besides the quality of the project, the individual performance in the discussion and the lecturer's evaluation.

The project grade (45% of the course's grade) is the *best* of the following two:

- Final_Project_Grade
- 85% of the Final_Project_Grade + 15% of Checkpoint_Grade

9 Cooperation among Groups

Students must not, *in any case*, see the code of other groups or provide their code to other groups. If copies of code are detected, both groups will fail the course.

10 "Época especial"

Students being evaluated on "Época especial" will be required to do a different project and an exam. The project will be announced on January 27th, 2017, must be delivered February 2nd, and will be discussed on February 3rd, 2017.