DADSTORM Final Report

Instituto superior Técnico
Desenvolvimento de Aplicações Distribuídas
Lisbon, Portugal

78369 Catarina Belém
78958 Gonçalo Rodrigues
78572 Telma Correia

Abstract

Real-time data processing systems are more and more popular nowadays. DADSTORM is a simplified implementation of a fault-tolerant real-time distributed stream processing system. It provides a temporal stream model that allow users to express stream processing logic.

1. Introduction

There is a class of applications in which, a large amounts of data generated in external environments are pushed to servers for real time processing. These applications include sensor based monitoring, stock trading, web traffic processing and online social media such as Twitter and Facebook. The data generated by these applications can be seen as streams of events or tuples. Since large volume of data is coming to these systems, the information can't be processed in real time by the traditional centralized solutions. To facilitate such large scale real time data analytics it is required a class of applications called distributed stream processing systems (DSPS). In these systems, events are often processed in multiple stages that are organized into a directed acyclic graph (DAG), where a vertex corresponds to the continuous and often stateful computation in a stage and an edge indicates an event (or tuple) stream flowing downstream from the producing vertex to the consuming vertex.
The continuous, transient and latency-sensitive nature of stream computation makes it challenging to cope with failures and variations and makes stream applications hard to develop, debug and deploy. As a result applications might employ fault-tolerance mechanisms, such as checkpointing and replication, to mitigate the impact of a vertice failure in the system without having to rerun the application.
This paper presents the design and implementation of DADSTORM, a real-time distributed stream processing system. DADSTORM support not only a continuous stream computation model, but also the ability to allow user-defined functions to customize stream computation at each step.
DADSTORM is highly flexible allowing the user to configure the processing guarantees according to the application requirements, for instance, business-critical stream applications it's desirable a strong guarantee that each event is processed exactly once despite server failures and message losses. Failure recovery comes to the dependency between the upstream and downstream vertices and the dependency introduced by vertex computation states. An upstream vertex failure affects downstream vertices directly, while the recovery of a downstream vertex would depend on the output events from the upstream vertices. Failure recovery of a vertex would require rebuilding the state before the vertex can continue processing new events. To deal with failure recovery DADSTORM uses a checkpoint-based approach. Checkpointing intermediate state enables a vertex to replace the failed vertex and requires only partial recomputing from the last checkpoint. Albeit checkpointing enables the processing of the stream to continue, each checkpoint introduces an overhead proportional to the size of the memory state.
Our evaluation demonstrates a good scalability and capability of achieving low latency on simple applications while having a good throughput. Further tests on the system demonstrate little overhead resulting from checkpoints.
One key factor in DADSTORM's system is the capability of configuring the execution guarantees according the applications needs. It's flexibility allows users to get different performance properties. For instance if a user's main concern is the processing speed he might chose a weaker guarantee (at-most once), whereas an application responsible for processing bank transactions would require a more reliable and strong processing guarantee (exactly-once). The rest of the paper is organized as follows. Section 2 describes DADSTORM's stream model. Sections 3 and 4 defines DADSTORM's design and implementation, where it's discussed several design choices. Section 5 presents the evaluation results of DADSTORM. We discuss future work in Section 6 and conclude in Section 7.

2. Programming Model

In this section, we provide a high-level overview of the programming model, highlighting they key concepts including the data model.

Continuous tuple streams. In DADSTORM, data is represented as tuples. Each tuple is assigned a sequential unique ID and an ID of the issuer of that tuple. In addition, whenever the processing of a tuple produces multiple tuples, these tuples are assigned sequential 'sub IDs'. Thus, all the tuples in the system have a distinct ID which allows for a fully deterministic behaviour based on this identification.

DADSTORM not only supports a set of primitive operators including filters, duplicatons and counters, but also provides the user the ability to specify his own library of functions and execute them through the custom operator. For instance, imagine a bank application who runs thousands of transactions per day. Each transaction has a value associated and if that value is considerable high, the manager of the bank might want to be notified that a transaction with a high value is being processed. Thus, function could be specified and through the custom operator, the bank manager could be notified every time a higher valued-transaction occurred.

3. DADSTORM's Abstractions

The execution of a DADSTORM's can be modeled as a directed acyclic graph (DAG), where each vertex performs computation on input streams of tuples and produces output streams of tuples. In DADSTORM's terminology each vertex is an "Operator", which is an abstraction of the underlying system.

3.1 The Operator Abstraction

The operator abstraction conceals a fault-tolerant composite subsystem composed by several smaller modules called Replicas. Replicas are responsible for processing the stream of tuples.

Determinism. For an operator with its given input stream of tuples, independently of which Replica processes the tuples, the produced tuples of the operator will always be the same. Determinism is accomplished by ensuring that the order in which the tuples are taken to be processed by Replicas is deterministic, we explain how this is anchieved in Section 4; and by also ensuring that the processing logic is deterministic, which means that for a specific tuple the result of processing that tuple is always the same.

4. Basic Architecture

DADSTORM is a real-time distributed stream processing system which supports both batch and interactive processing. DADSTORM's core design and implementation is fully distributed and is complemented with a centralized component, the PuppetMaster, which is reponsible for creating a physical distributed system according to the user's specification. In order to enable the PuppetMaster to successfully create physical vertices, it's required that each physical machine integrated in the system has the Process Creation Service (PCS) component up and running (waiting for requests). This centralized strategy facilitates the development, deployment and debugging of the system. Unlike other approaches such as StreamScope[1], DADSTORM does not restrain to a single component the tasks of (1) monitoring processing and tracking snapshots; (2) providing fault tolerance by detecting failures and initiating recovery actions. Instead, each operator is responsible for monitoring processing and tracking snapshots within its own Replicas.

4.1. Merge Operator

In order to guarantee input determinism, we designed a merge operator which merges different input streams in a deterministic manner. This operator waits for a tuple from each input stream, and then chooses the one with the lowest ID removing it from the input buffer. In case of a tie, it chooses based on the input stream ID. The figure 1 represents a simulation of the merging of 3 input streams. This can have some problems if an input stream doesn't send any tuple for a long time. In order to avoid indefinite waiting, every some seconds all operators send a "flush" tuple, which carries the ID of the patest outputted tuple in that operator, to theirs downstream operators. Upon receiving this "flush" tuple, downstream operators are aware that from that moment on sent tuples will have higher IDs than the ID in the "flush" tuple.

Assuming that the functions specified by the user for the custom operator are deterministic, we're guaranteed that all the operators are deterministic. By processing them sequentially and outputting them one by one, we also assure that the output is a stream of tuples ordered by their ID. Thus, it's possible to conclude that the system is fully deterministic as all tuples are received in ascending order of their ID, or in case of a tie, by the order of the input streams defined by the user's specification.

4.2. Tuple Routing

As mentioned in Section 3.1, DADSTORM introduced the concept of an Operator which has its own processing modules called Replicas. DADSTORM provides the user

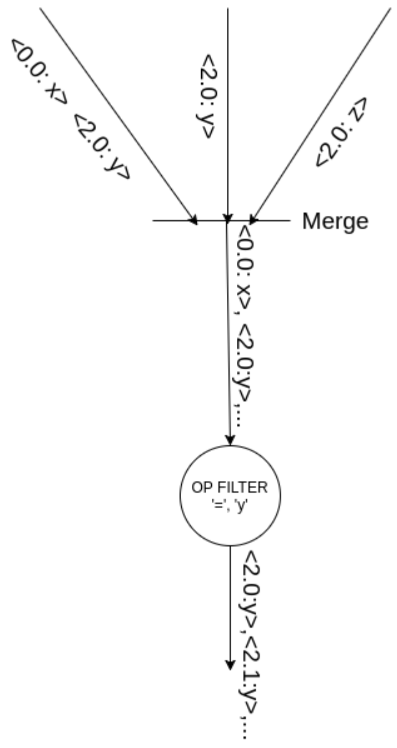


Figure 1. Merge operator

with the flexibility to specify different policies concerning to tuple routing: Primary, Hashing and Random.

Primary routing the upstream operator sends tuples to the first replica of the operator that it's connected to.

Hashing routing the upstream operator sends tuples to a replica that is selected by the value of a hashing function computed on the specified tuple's field.

Random routing the upstream operator sends tuples to an arbitrary replica of the operator that it's connected to. A problem of consistency between different replicas arises when one of them crashes. In order to solve the inconsistency issue, it was used the same seed to generate the same random sequence in each replica of the same operator.

4.3. Fault Tolerance

In a distributed stream processing system it is desirable that the distributed computation, the data availability and the performance remain predictable and correct. Thus, in order to provide these properties, DADSTORM provides different implementations of processing semantics and implements a checkpoint-based approach to deal with failure recoveries. Our fault model assumes that faults are crash-only and that the system is synchronous.

4.3.1 Tuple Processing Semantics

DADSTORM implements three different guarantees regarding the semantics of each input tuple in presence of possible failures:

At-Most-Once semantic a tuple is processed at most once, which means that if a tuple is lost (possibly due to failures in the operator) it might not be processed. To ensure this semantic, the upstream operator sends each tuple asynchronously, as the upstream operator does not require an acknowledgment from the downstream operator to proceed with its execution.

At-Least-Once semantic guarantees that all tuples are processed at least once. To ensure that tuples are processed even in the presence of failures in the operator, the upstream operator keeps sending the same tuple until an Ackownledge is returned from the downstream operator.

Exactly Once Semantic guarantees that all tuples are processed exactly once. This semantic is achieved by using a similar approach to At-Least-Once: in addition to the synchronous dispatch of tuples in every upstream operators, each downstream operator upon receiving tuples filters the duplicate ones. To filter duplicates, each Operator keeps the last tuple received from each input stream. When a new tuple is received, the operator checks whether the ID is smaller than the last one he received and ignores it if so. This is guaranteed to work because the tuples are being sent in a strictly ascending order as explained in Section 4.1.

4.3.2 Checkpoint-based Recovery

Before discussing our checkpoint policy, it's important to stress that DADSTORM is fault-tolerant to deterministic failures within the same operator, that is if the whole operator fails, meaning that every replica within that operator has failed, the stream processing flow in the system is interrupted and the application has to be re run. DADSTORM uses the concept of checkpoints to provide effective failure recovery. While a not so frequent checkpointing approach results in higher latency and higher recovery time, checkpointing too frequently degrades the average case performance, due to the added overhead. After some tests considering the memory used to cache tuples, the number of messages on the network and the size of the state, it was defined a period of 10 seconds between checkpoints. This value optimizes both the checkpoint overhead and the overhead of recovering crashed replicas. Each replica maintain the state of every other replica, to allow the capability to recover in case of failure. In order to avoid having several replicas attempting to recover the failed replica, DADSTORM has stored, for each operator, each replica's ID in a circular list. Whenever there's a failure, the failed replica will automatically be recovered by the next alive replica in the list. The recovery process will create a new replica's instance, which will load the state of the failed replica and if needed will ask upstream operators to resend some tuples so that it can process them. After ensuring that a consistent and coherent state is reached, upstream operator routing strategies are updated. Although, the recovered replica might send already sent tuples, this constitutes no problem because all duplicates are filtered based on the tuple's ID. Since both input and processing are deterministic, we're certain that the recovered replica's final state is coherent with the state of the replica when it failed. DADSTORM supports the simultaneous failure of replicas in different operators, without compromising the whole system. If a failed replica asks for resending of tuples of another failed replica, the upstream one will resend them after recovering and the process will eventually complete. If two out of three replicas fail in the same operator, they will both be recovered by the remaining one. Thus, our failure model assumes that f failures are supported within an operator as long as it has f+1 replicas.

DADSTORM implements perfect failure detector (PFD) modules in each replica, providing them with the ability to detect failures and therefore each replica might act quicker upon a failure. The fact that this failure detection module is not centralized means that the PFD is always available.

Garbage collection

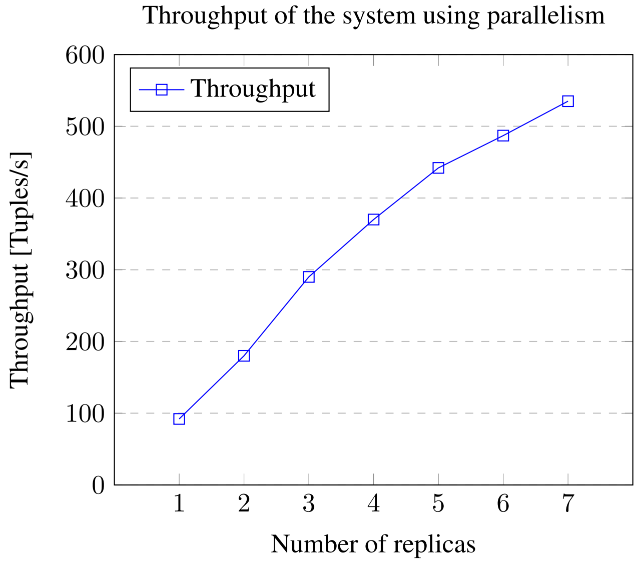
The ever increasing cache of tuples stored by each replica is a problem when the system is kept alive for long periods of time. To solve this, DADSTORM implements a garbage collection mechanism. Every time a replica propagates the state to all other replicas, it tells its input streams that it no longer needs tuples processed prior to the saved state. When the input replica receives this message, it saves it on a list. It then checks in its list if all output replicas also no longer need those tuples, and deletes them if that's the case.

5. Evaluation

To evaluate DADSTORM, we carried out a few experiments, specifically to measure the failure impact and scalability of the system. In order to simulate a real application, we created a CUSTOM operator that slept for 10ms and then returned the input it received.

5.1 Scalability

To evaluate scalability, we built a simple network of 3 operators where the middle operator had a varying number of replicas. As expected the results resemble a linear relation between degree of parallelism and throughput, as there is no synchronicity between replicas, and so they work completely in parallel. For this measurements we used no delay in the input or output operators and used a 10ms delay in the middle operator (which had the replicas).



5.2 Failure Recovery

To perform this experiment, we created a network of nodes detailed in figure 2 using random routing.

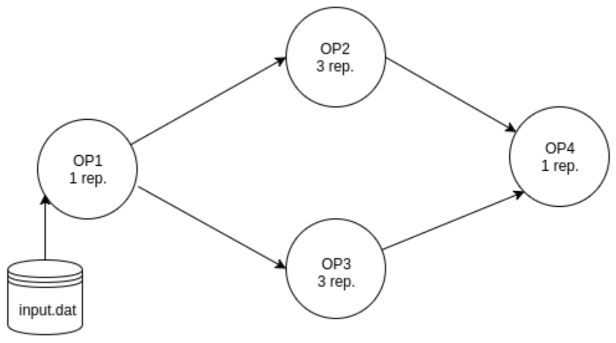
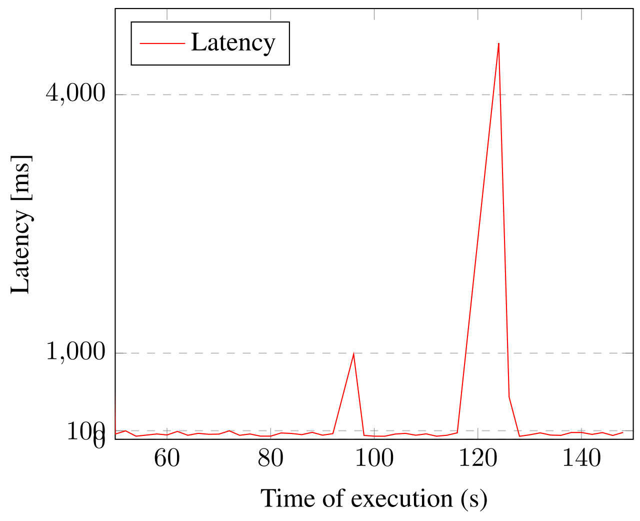


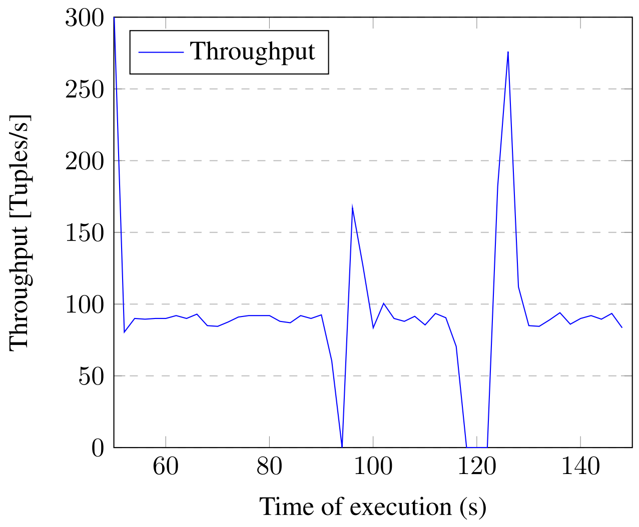
Figure 2. Configuration used in evaluation

OP1, OP2 and OP3 were using the Custom Operation that takes 10ms to process a tuple. The PFD assumed the network had no delays bigger than 1 second (local machine). We then crashed a replica of OP2 and a replica of OP3, 30 seconds apart. In both graphs we can notice that the impact of a failure is quite large. This is mainly because the merge operator waits for a tuple from each input stream, and when one of these stops sending tuples, it just keeps waiting. We can also notice big throughput spikes after the failures. This is because while a node was failed, others kept sending tuples and the output node just saved them in a buffer. Upon recovery of the failed nodes, the buffered tuples can be sent rather quickly. This diminishes the real impact of the failure.

Latency of the system when nodes fail



Throughput of the system when nodes fail



When failures aren't happening, the overhead in performance of state propagation is minimal. However, the memory required might be large as a result of the periodical checkpoint executed by the system.

6. Future Work

DADSTORM is a far from complete implementation of a fault-tolerant real-time distributed stream processing system. In the future we would like to relax the fault model to see if we don't need a synchronous system and also provide solutions to improve throughput when there are node failures.

7. Conclusions

DADSTORM takes a principled approach to distributed fault-tolerant stream computation. In spite of being a simplified version of a stream processing system, DADSTORM required the implementation of several algorithms and the application of several key concepts in order to produce a coherent and consistent system.

References

[1] W. Lin, H. Fan, Q. Zhengping, J. Xu, J. Zhou, and S. Yang. Streamscope: Continuous reliable distributed processing of big data streams. pages 1–16, March 2016.