DADSTORM
Distributed and Fault-Tolerant System for Tuple Stream Processing

Jorge Santos, Miguel Vera, José Semedo
Instituto Superior Tecnico
Computer Science Department
Av. Rovisco Pais, 1, 1049-001 Lisboa
{jorge.pessoa, miguel.vera, jose.semedo}@tecnico.ulisboa.pt

Abstract

Two of the biggest challenges on a distributed system are how to keep it properly working after a node failure and how to guarantee that it performs its tasks as intended despite those failures. We developed a distributed tuple processing system, DADSTORM, that can tolerate faults within a reasonable fault model, and guarantee that tuples are processed the intended number of times only. In the following paper we explain the reasoning behind it, and how the implemented algorithms work.

1. Introduction

Data streaming distributed system architectures are used throughout both research and production environments. On one hand, for research they provide a high degree of flexibility to quickly implement and test new algorithms. On the other hand, for production they need to provide scalability (small effort of adding servers to increase performance) and high availability without the need for human intervention. Our goal was to implement a simplified fault-tolerant real-time distributed stream processing system that has strong guarantees on the correctness of it's distributed computations (semantics) and the predictability of performance in case of failures (fault-tolerance).

We implemented an architecture based on tuple processing. Tuples are a data structure comprised of an arbitrary number of ordered fields (For simplicity all mentions of tuples will refer to tuples solely comprised of Strings) and a unique ID. Our tuple streaming system is comprised of a network of operators. Tuples are streamed throughout the network from a starting operator that might read them from a file to an ending operator that typically outputs to a file or produces a result. In this system a distributed computation can be seen as an acyclic graph, vertexes correspond to operators and links to data input and output. Each of these operators transforms the tuples they receive as input and may output them in a processed form. This processing varies accordingly to the type of Operator. In our system we only support a narrow list of different operators responsible for simple operations such as filtering, counting and duplicating tuples among others. We also allow the loading of custom dll's to support the inclusion of operators tailored for specific cases.

Since this is a distributed system, operators may execute on different machines. Furthermore operators might be divided into several replicas. All replicas of a certain operator perform the same processing operations, however they might also be deployed in different machines. These replicas distribute operator load and are the core of the system's fault tolerance capabilities. Finally, the fault model we assumed when designing the system implies that only one replica will crash concurrently, as such, some of our solutions might not be adequate for systems where other types of catastrophic faults might occur.

2. Solutions

How to create a system with scaling performance, guaranteed results and highly automatized fault tolerance is not a new problem. Before we arrived at our final solution we tested and discussed other solutions for fault tolerance and semantics. Some of these solutions and their main advantages and disadvantages are briefly discussed ahead.

2.1. Waiting for an ACK

One naive approach that is often mentioned when discussing semantics and result guarantees is waiting for some sort of confirmation from the following replica confirming that the tuple has been successfully delivered to it's destination. This way there is guarantee that tuples are delivered

and processed correctly.

Although it is correct to say that this provides guarantees that are uncommon in distributed systems, this approach is fundamentally incorrect. Implementing this into a data streaming distributed system would transform it into a synchronous system. This would go against the purpose of a distributed data streaming architecture. The first operator in a network would have to wait until the last operator to finish and for the message to be propagated back in order to know that the delivery and processing were executed correctly and that it could continue processing tuples.

2.2. Mirroring data throughout the network

Another naive approach to semantics and fault tolerance is mirroring data throughout the network. In this case it could be applied by mirroring and saving received tuples on all replicas of the operator that received the tuple. By having all information that goes through an operator, replicas can guarantee that no information is lost when a fault of one of the replicas occurs. Contrarily to the previous solution, this approach doesn't slow the processing of tuples by the operators, but it creates other problems with data.

Sending every received tuple to every other replica imposes a heavy communication load on the network infrastructure and creates the need for big storage capabilities on machines running the replicas if the tuples are of a considerable size. Despite the fact that this solution works, it is sub-optimal as keeping a copy of the tuple on every replica is unnecessary.

3. Overview

As we saw before, other solutions to this problem have several crippling disadvantages. Our analysis of other approaches allowed us to avoid some of the common pitfalls and get insight into factors like communication load imposed on the network infrastructure. Taking this into account we designed solutions we believe could be applied to real world scenarios. We tried to create an abstraction between our replica fault tolerance and our semantics algorithms, guaranteeing that even if they have to interact in certain operations, they can still be seen as separate.

Our Fault Tolerance algorithm can be divided into three separate phases: Fault detection, replacement and reinstating. Fault detection is handled solely by the replicas of a single operator. Replicas are organized into a ring architecture and are responsible for detecting faults on replicas placed next to them in this ring. Replica replacement is also handled by the replicas, when a replica fails, other replica will completely take over it's responsibilities. It signals the previous and following operators to ensure they no longer send data to the failed replica. Reinstating a replica is a simple process that returns the network to it's original configuration.

Our semantics algorithm has three different behaviors for the three types of semantics: at-most-once, at-least-once, exactly-once. The at-most-once doesn't require any special behavior as in this case the system always has the same behavior irrespective of failures. Tuples are identified by a Id that represents it's stream along the processing chain of operators, and serve to store information about those tuples in special data structures. At-least-once and exactly-once algorithms share a lot of their behavior and data structures that are explained in the corresponding section. The main difference between exactly-once and at-least-once is that the former must guarantee that tuples are never processed twice while also confirming the tuple has been processed at least once.

Further and more in-depth explanation of these algorithms follows in the next chapters.

4. Fault-Tolerance

To make sure that the system is always kept in a working state, it needs to be able to recover from faults within the fault model and recover to a working network configuration. Our fault tolerance algorithm provides the ability to tolerate a downed replica of any operator, as well as proceed to it's recuperation. The algorithm we used is possible due to the use of a simple, yet resourceful abstraction mechanism, which the semantics algorithm will also greatly benefit from aswell. Each of the replica mantains three sets of of dictionaries on which a replica is associated to a number (initally the replica number), except on the first and last operator's replicas. This is because each of the dictionaries holds the correspondence between a replica and the dictionary holder's view of it. One dictionary abstracts all of the colleague replicas of the holder (replicas of the same operator as itself), another distinct dictionary abstracts the replicas of the above operator, and again for the one below. So every replica of every operator has the ability to establish a connection with what they think is every replica above and below them, as well as their colleagues.

4.1. Fault Detection

Going into more detail on how a downed replica can be detected. Colleague replicas among themselves have the illusion of being in a ring like topology. We chose to consider the topology as a ring to simplify the used mechanisms since it perfectly fits the way replicas distribute on the type of network used in the system. Each replica periodically pings the following colleague like illustrated in Figure. 1. A replica can efficiently know what replica it precedes. It

knows it's own replica index i, therefore it knows the index of the replica it has to ping: $ping(i)$, by applying the following rule:

$$ping(i) = (i + 1)\ \%\ number\ of\ replicas$$

If the pinged replica is detected to be down, then the replica can just calculate the following replica's index using the very same rule, and contact it, asking for it to take over the downed replica's spot.

4.2. Take Over Procedure

The replica taking over goes through the 3 sets of replica holding abstractions aforementioned. Firstly it iterates over the replica holding abstractions corresponding to the operator above it, and for each one, it substitutes the place of the replica detected as downed with a representation of itself. It literally takes over it's spot, so every command that would be aimed at the downed replica, is routed to the one that took over it's functions. It then repeats the process for it's colleague replicas and the ones of the operator below it, as illustrated in Figure 2. where Replica 3 from the operator OPk takes over another failling replica reconfiguring the network.

There is however a small detail that caused a somewhat meaningful problem. In the case of an operator having only two replicas, and one of them going down, the only remaining replica would substitute the representation of it's colleague with itself, and then effectively start pinging itself. In order to solve this problem each replica contains a list of the indexes of the replicas it is representing (original one, plus each index it has taken over). This allows a replica to know whether it would be attempting to connect to itself or not.

After a replica has fully taken over for another one, all other replicas have the illusion of the topology state remaining the same. This is because every method that implies connecting to another replica, obtains the connection through the abstraction layer previously described, so every work load is seamlessly diverted to the new replica. Only replicas within the same operator can detect downed replicas. If any other remoting method fails due to the target replica being down, it retries the operation after an active wait until successful, since the network will always reconfigure itself to a working state. We chose this implementation over the option of any remoting task possibly triggering a take over, due to the possibility of the retrying of the faulting method being faster than the take over process. Fixing this issue would over complicate things, and the simpler implementation as we have now revealed it self as the faster one.

4.3. Reinstate Procedure

The reinstating protocol is simply put, the reverse of the take over process. Once a replica is brought back it goes through the same process of accessing previous, colleague and following operator's replica abstraction holding structures in that respective order, and inserting a representation of itself in it's rightful spot. After this, all remote methods are once again routed correctly to the reinstated replica. The replica that had taken over for it also has it's index list updated, meaning that the index of the reinstated replica is removed from that list.

5. Semantics

One of the biggest concerns in distributed tuple processing is being able to a guarantee certain semantics associated with the processing of a tuple in the presence of system failures. This issue is inevitably linked to the way that the system does fault tolerance even if there is an effort to separate the issues. The algorithm used for at least once and exactly once delivery is very similar, as previously said. The approach used for at most once delivery that we discuss later consists of not using any kind of algorithm to recover from losses. We will first explain the general idea behind the more complicated algorithm, then the used data structures and the algorithm and finally how it used to assure that a tuple is processed at most once, at least once and exactly once.

5.1. Explanation

If the system must process a certain tuple at least once then there must a guarantee that in any kind of a node failure the tuples that weren't successfuly sent to a node in failure are sent either to another node or to the the original node if it recovers. There are however other scenarios in which a node can fail, For example after receiving a tuple but before sending it to the next operator. Keeping in mind that the system must obviously be ashynchronous in this confirmation, then the previous node can't easily know it has to resend the tuple, and when it doesn't, or how long it needs to keep the tuple. In this approach the tuples would need to be kept in every operator at least until the tuple was processed by every node, presenting scalability issues. That's why we opted to use an algorithm based in keeping the information of the tuples being processed and by which replica, as well as which replica originally contains the tuple. That way all the tuples that weren't sent can be processed and sent again if a new node takes over a dead node without the need to replicate tuples by all nodes of an operator.

5.2. Data Structures

To understand how the algorithm works we must first explain the used data structures: A tuple is identified by it's Tuple Id structure which represents a stream of a tuple along the processing chain. In each specific tuple there is an unique id that is usualy kept along all operators, unless there must an output of several tuples from the same one, efectively diverging the tuple stream. The remaining information kept in the Tuple Id refers to the operator and replica it came from.

In each node there is a delivery table (a simple Hash-based Map that stores each tuple by its unique id) and that keeps every tuple received in a replica until it can be disposed.

There is also a shared tuple table in each node that stores Tuple Records associated to a tuple id (in a simillar Hash Map). A tuple record is a small representation of a tuple containing its Tuple Id, as well as the replica emiting the tuple record. This way the tuple record contains only the necessary information for a node to re-process a tuple that wasn't properly processed due to failure. For information storing purposes the tuple records have a state (pending or purged) that is used to store tuple records of the tuples processed by a replica in a purged state instead of deleting them in order to properly know what tuples have already been processed by that replica.

5.3. Algorithm

The ordered steps represented in Figure 3. show most of the necessary procedures executed every time a tuple is received, to keep the data structures synchronized in order to allow a proper recovery when a node fails. When a node receives a tuple (1) it delivers the confirmation of receiving it to the previous node (2). The next thing it must do is insert it into it's delivery table (3) and it's shared tuple record table (4). The node then synchronizes every table with the other replicas shared tables (5) and finally processes the tuple, sending it afterwards (6). After the tuple is sent and confirmation is received (7) the node will issue the purging of the tuple record of every other shared table and purge its own shared table of the same tuple record (8). Finally it will warn the replica which the tuple originated from that it can finally delete it from its delivery table (8) since it was already received by the forward node. This information is used whenever a node takes over another failling node. To recover from a dead node, it scans it's shared table looking for tuple records of the dead node and for each record it finds, it requests the tuple to the replica that the tuple originated from. Finally that tuple is processed and sent to the following node, by the same procedures explained before, as if a new tuple was received.

Note that this approach only ensures the semantics of the delivery in case of a single horizontal failure (when two sequential replicas on the path of processing a tuple fail). It could also support more failures if necessary by extending the number of nodes where a tuple is kept before it is delivered.

5.4. At-Most-Once

To assure that a tuple is processed at most once, a system must only send any tuple once, indepent of failure. Since a configuration for this system is dependent on being an acyclic graph, the only guarantee we need to provide is that there isn't any kind of mechanism to resend tuples in case of a takeover of a node that crashed by another node. As such the implementation of this strategy relies on not sharing any kind of information about tuples already processed, which is achieved by not executing any kind of the procedures explained before related to the applied algorithm (that isn't really applied in this case of semantics). This way, a certain tuple from an input is only sent once in the forward direction of the distributed network, and in case of a node failing all the tuples that it had processed but not sent yet won't be processed at all. The same applies to a replica that sends a tuple to the next replica, which in case of not getting a response, will just drop the tuple.

5.5. At-Least-Once

The algorithm discussed for semantics accomplishes at least once delivery in a system without further changes, since it guarantees that even in case of failure each tuple that wasn't surely delivered will get sent again. The only corrections necessary are when the tuples can't be sent if already sent.

5.6. Exactly-Once

Finally to guarantee that tuples are delivered exactly once, and taking the algorithm descbibed as a starting point, there is the need to guarantee that a tuple is never processed twice on the same operator. The first thing to remember is that each tuple has a unique Id that identifies it's path along the processing chain. Based on that, each received tuple on a node is checked by it's id to guarantee that it has never been processed, by comparing it's unique id to every tuple record stored in the shared tuple record table (hence the possible states for a certain tuple). The tuple is accepted with success in case it was never inserted in the table, assuring the replica that the tuple was never processed and sent to next operator before, which together with the guarantees provided by the algorithm assure exactly once delivery of tuples.

6. Evaluation

In order to proper understand the effectiveness of the developed solution it is necessary to understand the performance implications of the used algorithm and how it compares with other possible solutions. Firstly we will analyze the used solution from a theoretical standpoint and compare it with the previously mentioned naive solution. Then we will test a few measures and compare them to the theoretical results in order to provide some confirmation.

6.1. Analysis

To simplify the process of creating a model that fits the developed system we will consider a few assumptions that hopefully won't distort the obtained results by much:

- The considered system is only comprised of simple operators where one tuple can only originate another tuple, without stream divergences;
- The system can be simplified by some measures where R is the average replication factor of each operator, O is the number of operators and T is the average size of a tuple being sent;
- Tuple Records have have a constant size of 64 bits for the unique id plus 32 bits to identify the operator and replica and 1 bit for the state. This value is represented by $TR \approx 128\ bits$.

In this case the system consists of an acyclic (forward) graph where we can attribute a weight to each transition depending on what measure we're analyzing.

Let's start by considering the amount of data transfered per tuple sent: For each tuple received by a replica it sends a tuple record to every other replica, so $(R-1)\times TR$. There is also a purge message that contains only a unique id, which we will aproximate to the Tuple Record (TR) size. Similarly the confirmation to the previous contains the id. All things considered the total amount of data for each tuple received per replica for all the operators is

$$(R-1+1+1)\times TR\times O \Rightarrow O(R\times TR\times O)$$
$$TR\,is\,constant\ \Rightarrow O(R\times O)$$

Comparatively a naive solution that receives a tuple on a replica and replicates it through all the replicas (removing the necessity for forward and backwards confirmation) would require

$$(R-1)\times T\times O \Rightarrow O(R\times T\times O)$$

The difference of both approaches being that our proposed solution scales with the size of a tuple record, TR which is a constant value, while the naive approach will scale with the tuple size as well T. For increasing tuple sizes the difference in data that needs to be exchanged between replicas for each tuple is noticeably smaller as we expect to confirm in the pratical results. In this sense, we can start to understand the performance advantages presented by our solution when $T \gg TR$.

This is a simple approximation of asymptotic data costs for exchanging tuples, but serves to understand the fundamental difference between the considered solutions. When sending a tuple all the considered messages must be sent first and their reception confirmed synchronously as explain in the algorithm section. Since the time a message takes to be sent also deeply impacts the performance of the system and scales with the size of the message we can infer that our approach would also scale better than the naive approach in terms of time taken for a tuple to be processed on a certain node, since much less data has to be sent to other replicas in the case where $T \gg TR$.

Table 1. Total messages sent with different number of replicas per operator

Two	Four
400	2400

6.2. Testing

Unfortunately there wasn't an implementation of the naive approach against which we could compare the obtained results of testing our system. Nevertheless we will try to distinguish the impact of the implemented algorithms by analyzing a few measures.

The system is configured in a very simple way for most of the tests, using only two operators with the same number of replicas, which will then allow us to extrapolate some conclusions about the system.

On table 1 are the results of using different number of replicas for each operator, and how the number of messages exchanged scales with it. Note that the impact on replicating tuples instead of records would be much bigger. On table 2 we show the time difference between using at-most once and exactly-once semantics with two and four replicas. First row shows the results for small tuples, and the second row for tuples with 10x times size. While we can't compare the execution times, we can at least infer the performance implications of using the algorithm.

Table 2. Time difference between at most once and exactly once

Two	Four
0.6	3.3
1.4	4.2

7. Conclusion

Prior to analyzing the used algorithms, we have to acknowledge that the solutions aren't perfect but are a step forward from the naive approaches that might be picked in this situation. Most of the differences are actually unintelligible when processing small datasets, in which each tuple is small.

For example in this case there might even be an advantage of replicating tuples in each node after receiving from the operator, considering their sizes might be smaller than the tuple records and there are less steps involved in the algorithm. But on the other hand, if we try to scale the tuple size even a litle bit, then the tuple-replicating approach would scale much worse, resulting in huge amounts of exchanged data between replicas of the same operator before they can even process the tuples. On the other hand, replicating only tuple records scales in a constant way with tuple size (doesn't increase). One thing to note is that there is no perfect solution that applies to every kind of system in a scallable and eficient way. A much better compromise is to develop the system according to whatever are its needs of scalability in a real world deployment.

As far as fault detection goes, we think that other topologies can be adopted besides our ring-like one. A possibly good heuristic could be how far replicas are from each other, so the whole operator ping cycle (all replicas get pinged) is made as fast as possible. With different topologies different methods of picking what replica takes over are necessary. If the topology is a tree-like, then choosing a candidate becomes much harder. Some other things can be taken into account when picking a take over candidate, for example, work load metrics can be shared with each "are you alive" in order to pick which replica would take the extra load better. Another even more complex possible solution is to have more than one replica assigned to taking over the downed replica's work load. Once again it all depends on the real life application scenario.

To conclude we think that our developed solution fits the requirements of the project but isn't a perfect solution, as such a thing doesn't exist on the distributed system's world.