Network Partitioning Solution Analysis

Babbar, Abhimanyu

890729-7751

babbar@kth.se

May 2, 2015

1 Introduction

Structured overlay networks form basis for the majority of the distrubuted systems. They are popular for there ability to stabalize and handle churns in the network. In case the system is long lived, there is a chance of network partitioning happening. For the structured overlay networks given there generic nature, the case of network partitioning is another case of churn in the system. The case of cleaning up of the nodes is handled just like churn management.

2 Problem

The problem of healing or merging of the network partitions is really hard because of the fact that the nodes discover the new nodes as some nodes joining the system and therefore integrate them in there overlays. This causes the nodes in the different network partitions to merge seamlessly. But in cases where the nodes carry some information with them, we need to disseminate the information that they bring to other nodes in our network partition. This makes the process of network merge really hard.

The network partitioning issue becomes more complex in case of sweep. The system in case of network partitioning, divides the nodes in two separate partitions and then these nodes elect new leaders and work normally. In case the partition is long lived, the nodes would evolve separately from each other. When the partition heals, it becomes really difficult to synchronize the nodes in terms of the data being added to them during the network partitioning phase.

3 Solution Analysis

In order to deal with the issue of network partition we take help of the concept of Strong Eventual Consistency (SEC). The problem caused by network partitioning was that the nodes inside the different partitions can add the entries with the same identifier, so when the partition would heal, the nodes would contain different entries with the same id which is not correct in terms of consistency.

The other issue that arises is the absence of knowledge in terms of common history. The case of detection of the healing of network partition is really hard because as stated earlier the nodes discover the nodes from the other side of healing partition as new nodes joining the system and seamlessly integrate them. Therefore we need a mechanism for storing the history in terms of the evolution of system. Now, when the nodes merge, they will look into the history and see the other nodes having a common point and then diverging. This discovery would help with the detection of the network partition and inform the system to take appropriate steps to merge the information contained in other nodes.

Before we describe the actual solution, we would like to present some terminologies used in the solution.

3.1 Terminologies:

This section present the terminologies used in the solution. The explanation of the terminologies is regarding **Peer to Peer Search Engine**.

1. **Epoch**: The epoch can be described as a simple counter which helps the system always move forward as it evolves with time.
2. **Epoch Update**: This is a packet which contains information regarding the current epoch, leader in that epoch. The epoch update is closed with the entries that were added in that epoch.
3. **Leader Container**: Metadata representing the leader along with the entries added by the leader. It is contained inside the epoch update.

3.2 Solution Overview

The section deals with the solution used for detecting and healing the network partitioning. It is mainly divided in different parts which are described as follows:

3.2.1 Index Entry Addition Mechanism

In order to achieve Strong Eventual Consistency (SEC), the index entry addition mechanism needs to be updated to a conflict free replicated data type. The index entry is not identified by the entry number but by a sequence of parameters which allows us to create dense spaces for seamless merging. The entry is uniquely identified by **epoch, leaderid, entry number**. So this means that the entries in same epoch in two different network partitions can be identified uniquely as the leader will be in either one of the partitions and therefore a total order can be established.

3.2.2 Epoch Update History

In order to keep track of the event of the healing of the network partitioning, we need to store the Epoch Updates as it will be used to compare the histories of two nodes and determine the case of network partitioning. A node whenever it becomes a leader it pushes a landing entry commit in the leader group which creates a mark in the continously evolving timeline of the entry commits for the nodes.

3.2.3 Partition Aware Gradient

The partition aware gradient is a wrapper over the gradient component which will filter the samples that the gradient receives from the croupier, gradient exchanges between themselves and also send above to the application.

3.3 Solution Description

In order to better understand the soution, we will go through the normal functioning of the system and the procedure that needs to be followed by the nodes in order to make sure that the merging process of network partitioning goes smoothly.

3.3.1 Epoch Updates

1. When a node becomes a leader, it adds a **Landing Entry** to the leader group nodes. In addition to this it adds an epoch update to the leader group nodes which conveys information regarding the epoch , leaderid and empty entries added for now.

2. When the leader dies, the next leader follows the same procedure but with a slight modification. It calculates the entries added by the previous leader and then closes the previous epoch update with the entries calculated before publishing it's own epoch update to the nodes. This procedure helps to expand the history by appending new epoch updates to the already added.

3. The pull mechanism can be divided into Index Pull Mechanism and Control Pull Mechanism. The node pulls the epoch updates, leader for current round as part of the Control Pull Mechanism. The node doesn't start pulling indexes unless it has epoch update information. Upon receiving the epoch updates it goes through the epoch updates incrementally and then starts pulling index entries using the entry pull mechanism.

4. As the nodes pull the epoch updates through the control pull mechanism, the nodes in other partition will have a different epoch history and therefore a node needs to be careful regarding the nodes that it pulls from. Control Pull needs to be done from the nodes that either at the same level in terms of partitioning depth and same partition or ahead of the node but in the same partition that the node would be when it comes to there level.

5. Every node has a current epoch information **<epoch, leader, entries>** which informs about the current epoch that the node is pulling from nodes above him.

6. An important step that needs to be carried out is to always have nodes in the gradient who's update history is ahead i.e my update history is a subset of their's update history. For this, whole of the history need not be transferred just my current epoch update.

7. Now a nodes need to send the current epoch update when exchanging its descriptor with other nodes. This will help other nodes to make an informed decision to take decisions.

3.3.2 Partition Aware Gradient

1. Ideally the gradient wants to shuffle with the node that is closest to him and above him in the utility.

2. In case of network partitioning handling, the gradient shuffling needs to be refined. Now the node will not keep samples that are below him and who's epoch update is not a subset of it's own epoch history. These nodes are potential cases of network partitioning.

3. So the filtering will happen in stages, firstly I filter the nodes that are at the same level and not in my partition. In addition to this, allow the nodes that are ahead in the partitioning depth but if reached at that level, will lie in there partition. This is done, so that a node doesn't pull epoch updates from a wrong node.

4. Even of we have a gradient sample, the different pull mechanism will filter the sample according to there needs. For the control pull mechanism, you always pull from the nodes that are higher than you in the epoch update history or the leader directly if you see him in the gradient. The pulling node decides who to pull from and they should be nodes who's history is an **extension** of the node's history.

5. Once you have the epoch update, based on the information contained inside the update, the index pull mechanism starts asking for the entries from the nodes above them. The node swiches to the pulling the next epoch update only onces it closes the current epoch update.

4 Special Cases

1. If nodes need to pull the whole epoch update in order to move to the next update, how would the node move forward in case the nodes are above him in terms of partitioning depth and have removed the entries that the lower node is looking for ?

   The node will pull this information as part of epoch update and once it sees the partitioning update contained inside the epoch update, the nodes automatically mark the remaining epoch packets as completed, which it would be removing in case it reaches at that point. This will make sure that the node is not stuck and keeps moving forward.

2. Explain the process of network partitioning merge, how the entries will be transferred and nodes fetch the data from the above nodes ?

   During the gradient exchange, every node descriptor also contains the information regarding the current epoch update and therefore when a node sees some nodes that are either below or above it and the hostories have diverged then it triggers an event to the application with the set of nodes saying that there seems to be a network partition merging back in and therefore allow the application to process the update. The application in turn can try to locate the leader of its own partition and inform it about a potential network partition merge.

3. What about very short lived partition, where the epoch history difference might be very less ?

4. When would we need to collapse the history ?

5 References