Quantifying state staleness in a distributed resource management control plane

Harshit Gupta, Salini Mishra, Luca Verdejo Estvez
Georgia Institute of Technology
Atlanta, GA

***Abstract*—In this project, we present scheme for coordinated management of multiple clusters via state synchronization. We adopt mathematical approach for the same and take into account various factors like overhead for state synchronization, degree of staleness for clusters and overall time taken for the complete process.**

***Index Terms*—cluster management, resource allocation, quorums, quorum replication, controller instances, staleness**

I. Introduction

Clusters have been the center of attraction in the field of computing for years now. With increasing demands on the computing infrastructure, the budget for computing, storage and networking is quite low. Clusters born from off-the-shelf technology is a great alternative to the expensive high performance computing. But the growing requirements and demands identified that the current cluster technology lacked in scalability, fault tolerance and maintenance overhead.

Usually, clusters are integrated into bigger networks instead of using as standalones. They are upgraded constantly or installed incrementally, consequentially resulting in heterogenous hardware, software and application tools. This makes it difficult to manage them in a consistent manner. Where they provide outstanding cost/performance tradeoff, their effective orchestration is still a focal point from research perspective. This brings us to the requirement of efficient cluster management.

Cluster management systems like Borg [1], Omega [2] and Kubernetes [3] maintain cluster state that consists of information about what resources have been allocated for what tasks. When serving a resource deployment request, the resource schedulers read the cluster state, determine the best set of resources to allocate for the given request, and accordingly update the cluster state. In order to achieve scalability, there are often multiple scheduler threads performing resource allocation - all operating on the same cluster state. The aforementioned systems maintain up-to-date information consistent by appropriate concurrency control, for instance, Borg uses Paxos-based replication while Omega uses optimistic concurrency control.

This approach works in an environment where the cost of synchronizing cluster state is insignificant, for instance in data centers with fast interconnections. However, recent trends towards edge computing are pushing computing resources out of data centers closer to the network edge. The need for faster resource management necessitates moving cluster management logic closer to resources themselves, which naturally leads to the geo-distribution of the control planes. In such a setting, maintaining consistent cluster state can be prohibitive due to relatively slow connection between the control plane instances.

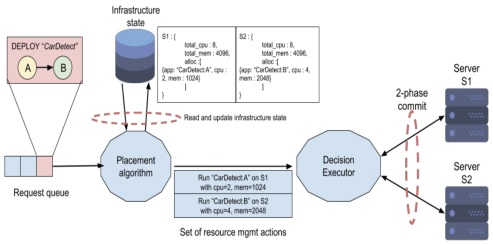

Fig. 1. System architecture

Such a distributed control plane requires novel methods to ensure as low staleness of cluster state as possible, while not degrading the performance. Most cluster management systems have :

- Server, that receives job requests and provides central access point to clients, users and schedulers
- Scheduler, that implements access policies, requirements, and determines efficient execution order
- Node manager, reports status of nodes to server and controls node access.

Usually, to make scheduling decisions, the scheduler has to interact with server and retrieve the status information. The server may also interact with the node managers to retrieve more detailed status information, not available initially with the server.

Our system assumes that an instance of the control plane (henceforth called controller) is responsible for servicing requests for application deployments/migrations/resource allocation updates. Such requests are sent by end-users of applications.

Placement Algorithm: In order to service a request, a controller determines which servers might be ideal for hosting tasks specified by that request. This decision is made by the placement algorithm that operates on the controllers view of the infrastructure (controllers state). The placement algorithm selects the best set of servers for that request and updates the controllers state with its changes for subsequent transactions to read. For the sake of simplicity, we assume that requests are processed serially by a given controller. The output of the

placement algorithm is a set of resource management actions that need to be performed on the chosen set of server, which we henceforth call a transaction.

Decision Executor: In order to execute the set of resource management actions (transaction) output by the placement algorithm, the Decision Executor employs a 2-phase commit protocol with the involved servers. This ensures that the transaction succeeds or fails atomically.

Handling failed transactions: If either one of the actions in a transaction is not successful, the other successful actions need to be rolled back in order to ensure that the infrastructure is reverted back to the state before this transaction was executed. The controller updates its state with the information it receives from the servers involved in the transaction. The original request associated with that transaction is then re-submitted to the placement algorithm to generate another transaction.

Causes of transaction failures A transaction output by the placement algorithm fails to get executed on the infrastructure, fundamentally, because the infrastructure state which the placement algorithm read was not the same when the transaction actions were received by the servers involved. We enumerate the following reasons why such a transaction T would fail :

- **Server failure**: A server (involved in transaction T) might have failed after the last time it sent a message to the controller and before it receives actions for transaction T. In that case the controller had no way of knowing about this failure when it was executing the placement algorithm.
- **Autonomous resource allocation updates**: In order to support real-time applications and enable fast resource allocation updates, increasing/decreasing resources allocated to running applications on servers can be done autonomously by an agent running on the servers themselves. A controller may not be aware of these autonomous allocation updates when executing the placement algorithm, and hence may have an incorrect measure of free resources on servers.
- **Multiple concurrent controllers**: In order to make the control plane layer scalable, we envision an ensemble of loosely-coupled controller instances operating on shared infrastructure. The likelihood of state staleness due to concurrent controllers depends on the policy of state synchronization, which ranges from no sync at all (extreme 1) to one where each controller obtains consensus among all controllers before submitting a transaction for execution (extreme 2). Each of these policies have associated probability of transaction failure due to staleness and overheads of state synchronization.

II. Problem addressed

We take a mathematical approach to analyze various state synchronization techniques used by the aforementioned distributed control-plane. For the sake of tractability we break the problem down into two settings : single-controller and multicontroller settings. We identify the following high-level metrics (which hold for both settings) :

- **The kind of staleness guarantees does the placement algorithm expect ?** For the single controller case, the placement algorithm expects that if there have been new writes to the infrastructure state after its own last write, these new writes should be visible to it. This guarantee is called strictly monotonic reads.
- **Probability of providing aforementioned staleness guarantees.** Based on the policy used for state synchronization between servers and controller and across controllers. We begin our exploration with the single-controller case, wherein state is synced between the controller and a server only when the server is part of a transaction and gets involved in a 2-phase commit with the decision executor.
- **Overhead of state synchronization.** Depends on policy for state synchronization. State synchronization would be in the critical path of request processing, and thereby would affect end-to-end throughput.
- **Impact on high level request processing latency.** The overall request processing latency depends on both the probability of transaction failure due to stale state as well as overhead of state synchronization as follows:

$$T_{\text{e2e}} = T_{\text{placement}} + T_{\text{stateSync}} + P(trans.fail).T_{\text{e2e}}, \quad (1)$$

where we assume that $T_{\text{placement}}$ is constant.

III. Why is this problem important?

There is no standard for deployment of edge computing systems, hence there is no consensus about the flavor of control plane state management solutions will be ideal.

The analysis in this project will enable us to reason the pros and cons of a specific control plane design based on the properties like latency distribution of a given edge computing deployment.

IV. Single Controller Instance : 75% goal

We designed a preliminary approach, defined by the 75% goal of our solution, where one controller will run the previously mentioned placement algorithm, and various servers will act as resources to execute the transaction delivered from the controller.

Objective. Formulate the probability of transaction success.

Parameter	Description
N	Number of servers in infrastructure
w	Number of servers updated by a transaction
L	Distribution of controller-server latency
t_{ITT}	Mean inter-arrival time of transactions
λ_{auto}	Rate of autonomous updates on servers

We make the following assumptions when building the model

- The servers make changes to the resource allocations autonomously (without coordinating with the controller) following a Poisson process with a rate of γ_{auto}.
- Network latency between the controller instance and servers are drawn from independent and identical Gaussian distributions L with mean L_{mean} and standard deviation L_{stddev}.
- Requests arrive to the controller instance based on a Poisson process with mean inter-arrival time equal to t_{ITT}.
- When computing probability of success for transaction T_i we assume successful executions for all transactions before it, as we don't account for failures and retries yet in this model.

Window of vulnerability : Consider a transaction T_i's command arriving at server S_j at time t. The controller computed T_i based on its view of S_j which was the actual state of S_j when it last communicated with the controller at time t'. Hence T_i is calculated based on the state of S_j at t'. If there were any autonomous changes after t', the transaction T_i would conflict them them and thus fail. Hence, the success of T_i on S_j depends on the non-existence of autonomous updates in the time interval $[t', t]$. We define this time interval as the window of vulnerability for transaction T_i on server S_j. This concept is explained in Figure 2 with a sequence diagram. Assuming that the last transaction updating S_j was T_i', the time interval between the send of acknowledgement from S_j for T_i' and arrival of commit request on S_j for T_i is window of vulnerability here.

Given that the latency between controller and a server is sampled from a distribution L, we can define the window of vulnerability using the following equations. Note that all times are with respect to wall-clock time.

$$t(send(ACK), T_i') = t(T_i') + L[0] + L[1] + L[2] \quad (2)$$

$$t(arrive(COMMIT\_REQ), T_i) = t(T_i) + L[4] \quad (3)$$

$$t(T_i) = t(T_i') + t_{ITT} \quad (4)$$

Based on the aforementioned equations, we have the window of vulnerability as

$$t(arrive(COMMIT\_REQ), T_i) - t(send(ACK), T_i') = t_{ITT} - L[0] - \quad (5)$$

Transaction T_i would fail if one of the servers it touches has conflicting updates due to autonomous resource allocations.

$$\mathcal{P}(T_i \text{ succeeds}) = \prod_{i=1}^{w} \mathcal{P}(S_j \text{ does not conflict with } T_i) \quad (6)$$

Let us assume that the last transaction that included S_j in its write set was T_{i-k}. Hence, the window of vulnerability for S_j would be $k \cdot t_{ITT} - L[0] - L[1] - L[2] + L[4]$. Hence, the probability of a conflicting autonomous change occurring on S_j is as follows.

$$\mathcal{P}(S_j \text{ conflicts with } T_i | T_{i-k}) = 1 - e^{k \cdot t_{ITT} - L[0] - L[1] - L[2] + L[4]} \quad (7)$$

The absolute probability of S_j conflicting with T_i is as follows.

$$\mathcal{P}(S_j \text{ conflicts w/ } T_i) = \sum_{k=1}^{\infty} \mathcal{P}(S_j \text{ conflicts w/ } T_i | T_{i-k}) \cdot \mathcal{P}(T_{i-k} \text{ last upd} \quad (8)$$

$$\mathcal{P}(T_{i-k} \text{ last updated } S_j) = \left(\frac{\binom{N-1}{W}}{\binom{N}{W}} \right)^{k-1} \cdot \left(1 - \frac{\binom{N-1}{W}}{\binom{N}{W}} \right) \quad (9)$$

A. Behaviour of success probability

We use Monte-Carlo simulation to determine the variation of $\mathcal{P}(T_i \text{success})$. The only random variable in the equation is the network latency between controller and servers , which is sampled from a distribution. The variation of success probability with respect to various parameters is shown in following figures.

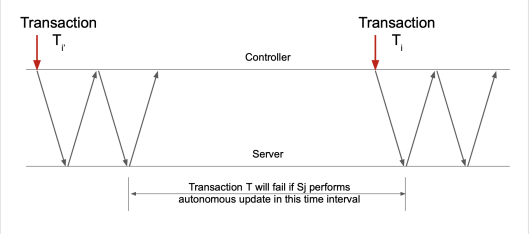

Fig. 2. Quantifying the window of conflict

Fig. 3. Variation of transaction success probability - (1)

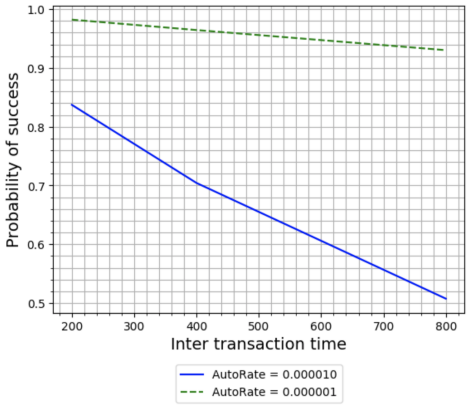

Fig. 4. Variation of transaction success probability - (2)

B. Issues with model

- Does not take into account failed transactions
  - As it does not support retries of transactions
- Cannot quantify the end-to-end transaction execution time

C. End-to-end simulation design

In order to observe the behaviour of transaction failure on queuing delays and end-to-end transaction execution times, we implement a more complex simulator which keeps track of the window of vulnerability of each server and handles transaction failures by retrying them and queues up any other transactions that arrive in the system before they are executed.

Assumptions :

- Transaction arrival follows a Poisson process (with mean inter-arrival time t_{ITT})
- Autonomous resource allocation updates follow Poisson process with rate λ_{auto}
- There is a single transaction executing at a time
  - Transaction finishes only after the two phases of a successful 2PC execution
  - Subsequent transactions have to queue and wait
- If the 2PC execution fails, the transaction has to retry

We show that the results obtained from the model described earlier match the results from the end-to-end simulator. Of course, for fair comparison, we don't retry failed transactions in the simulator and the inter-arrival time of transactions is significantly higher than the execution time of a transaction so that there is no queueing.

N	w	t_{ITT}	λ_{auto}	L_{mean}	Model	Simulator
100	4	200	0.00001	40	0.825	0.841
100	4	400	0.00001	40	0.685	0.709
100	4	800	0.00001	40	0.483	0.496
100	4	1600	0.00001	40	0.259	0.248
100	4	3200	0.00001	40	0.093	0.058
100	8	200	0.00001	40	0.826	0.832
100	8	400	0.00001	40	0.681	0.690
100	8	800	0.00001	40	0.468	0.475
100	8	1600	0.00001	40	0.232	0.215
100	8	3200	0.00001	40	0.066	0.051

V. Next steps

A. 100% goal

- Formulate the probability of staleness/transaction failure for multi-controller setting
- Implement a Monte Carlo simulation (based on PBS source code) and generate behavior of the staleness probability with respect to the independent variables like controller-server latency, inter-controller latency.

B. 125% goal

- Implementation of single and multi controller designs for a Docker cluster
- Infrastructure emulation to mimic edge-computing setting
- Correlate with results from the models built earlier

It is to be noted that all aspects of this simulated environment (e.g. communication latencies) will be drawn from probability distributions so that they match the assumptions we make in our mathematical analysis.

References

[1] Verma, Abhishek, et al. "Large-scale cluster management at Google with Borg." Proceedings of the Tenth European Conference on Computer Systems. ACM, 2015.

[2] Schwarzkopf, Malte, et al. "Omega: flexible, scalable schedulers for large compute clusters." (2013).

[3] Rensin, David K. "Kubernetes-scheduling the future at cloud scale."(2015).

[4] Satyanarayanan, Mahadev. "The emergence of edge computing." Computer 50.1 (2017): 30-39.

[5] Jyothi, Sangeetha Abdu, et al. "Patronus: Controlling Thousands of Geo-Distributed Micro Data Centers." Under submission (preprint available at http://abdujyo2. web. engr. illinois. edu/papers/Patronus. pdf).

[6] Bailis, Peter, et al. "Probabilistically bounded staleness for practical partial quorums." Proceedings of the VLDB Endowment 5.8 (2012): 776-787.