Deciding Backup Location Methods for Distributed Stream Processing System

Naoki Iijima, Koichiro Amemiya, Jun Ogawa, Hidenobu Miyoshi
ICT Systems Laboratories FUJITSU LABORATORIES LTD., Kawasaki, JAPAN
e-mail: {iijima-naoki, amemiya.kouichi, ogawa.jun, h-miyoshi}@fujitsu.com

Abstract—**A large amount of stream data are generated from some devices such as sensors and cameras. These stream data should be timely processed for real-time applications to satisfy the data latency requirements. To process a large amount of data in a short time, utilizing stream processing on edge/fog computing is a promising technology. In the stream processing system, a snapshot of processes and replications of the stream data are stored on another server, and when server fault or load spike of server occurs, the process is continued by using the stored snapshots and replicated data. Therefore, with edge computing environment, which has low bandwidth resource, process recovery takes a long time due to the transferring of restored data. In this paper, we propose a stream processing system architecture to decide servers to store snapshots and replication data and redeploy processes by considering the load of each server and the network bandwidth. We also propose a semi-optimal algorithm that reduces the computational cost by appropriately sorting servers and tasks according to the network bandwidth and server load. The algorithm can find a solution over 1000 times faster than the Coin or Branch and Cut (CBC) solver.**

Keywords-distributed system; stream processing; fault recovery

I. INTRODUCTION

There is a demand to get a large amount of data from devices such as sensors and cameras in the field. The large amount of data must be processed, and a notification or control signal must be returned to the devices in a timely manner. For example, monitoring by cameras, which needs instant decision making, requires processing the collected data and periodically outputting results with low latency [1]. Mobility services such as Dynamic map service, that provides a current state of traffic, require to process a large amount of data collected from many cars with low latency [2]. These services need a stream processing which processes data stream continuously and outputs the results periodically. Stream processing utilizing edge/fog computing is a promising technology to achieve the requirements of the real-time application [3]. The edge computing can deal with a large amount of data with low latency by processing data sequentially at edge servers that are located close to the device.

In real-time applications, a long-time outage of service can impair the quality of service and lead to monetary losses. For example, let us consider the camera monitoring system in a factory that monitors workers and Automated guided vehicles (AGV) walking in poor-visibility areas so that they do not collide with each other. If the system stops for any reason, the risk of collision increases. Thus, for fast recovery, stream processing stores a snapshot and replicated stream data to other servers [4]. The conventional approach selects the servers where the recovered process will be deployed based on the load of the servers. However, it leaves selecting where the snapshot and replication data are stored in the distributed data storage system. The system decides where to store data regardless of where the process will be redeployed; thus, the distance between the task-redeployed server and the data-stored server may become longer. The redeployed process cannot restart until transferring the snapshot and replication data ends. This could cause a long-time outage of application due to network delay between a server storing snapshot and replication data, and the server redeploying process. Especially, the edge computing environment cannot often prepare enough bandwidth. Therefore, we propose a stream processing system architecture with a location controller that controls the location of the snapshot data and replication data. The controller regularly collects loads of servers and networks and decides which server to store snapshots and replication data and which server to redeploy each process. By making the distance between the data-stored server and the task-redeployed server close, the controller reduces the transferring delay. Additionally, we propose a semi-optimal algorithm that quickly decides the semi-optimal combination of processes and servers even when the number of processes or servers is large since deciding optimal combination of processes and servers is a combinational optimization problem which takes long time to solve [5].

The contribution of this paper is two-fold: First, we propose a stream processing system architecture that can choose which server to store process snapshots and replication data, and which server to redeploy each process. Second, we propose an algorithm to quickly decide the combination of processes and servers. We also compare the CPU time derived from the proposed algorithm and the Coin or Branch and Cut (CBC) solver [6], which is a well-known open-source combinational optimization solver.

The rest of this paper is organized as follows: Section II shows related works and explain the problem of the conventional process recovery method. Section III describes the proposed stream processing system architecture and its problem about the computational cost to decide the allocation of processes and server. Section V shows how the proposed algorithm deals with deciding allocation. Section

978-1-7281-7283-5/20/$31.00 ©2020 IEEE
DOI 10.1109/ICICT50521.2020.00047

VI discusses the result of the proposed algorithm compared with the CBC solver. Finally, section VII concludes the paper.

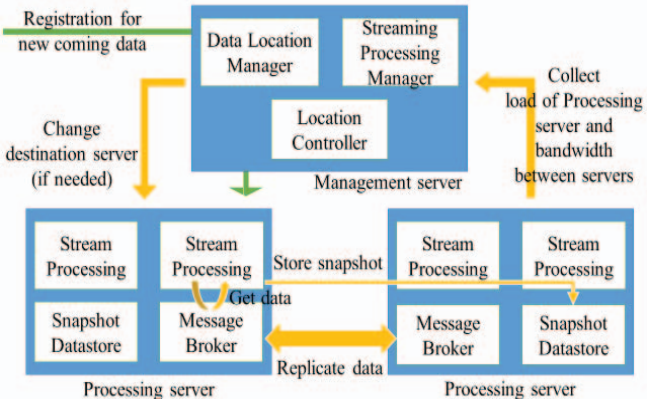

Figure 1. System architecture.

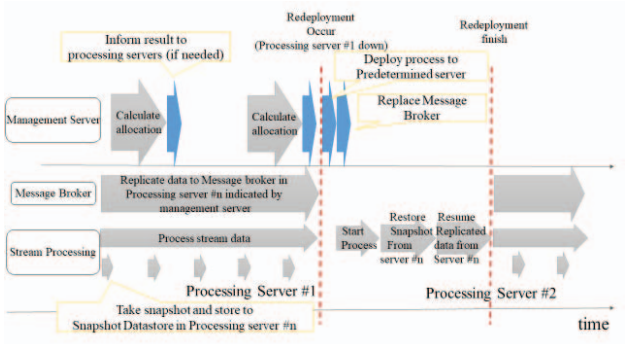

Figure 2. The time series of stream processing system.

II. Related Work

There are many studies about fault tolerance in stream processing. For example, Flink [4] has a checkpointing mechanism for the process recovery. It regularly stores a task's snapshot consisting of intermediate data of processing. The snapshot has all the information to restart the process. Flink stores the snapshot to a file system such as the Hadoop distributed file system (HDFS) [7]. When a server failure occurs, Flink redeploys a task on the server to the other available servers. Then, the task/process restores the failed process from the snapshot in the file system and re-processes data generated after taking a snapshot. Flink decides to which server the task should be deployed based on only the information of the server load. When there is plenty of network bandwidth in the system, Flink can complete the process recovery in a short time, however, the process recovery takes much of time when there is little network resource.

Zaharia et al. [8] proposes discretized streams that treat a streaming computation as a series of deterministic batch computations on small time intervals implemented on Spark Streaming. That realizes efficient fault recovery by treating a stream processing as a series of short batch jobs and regularly storing the snapshot to other servers. The shorter the interval, the shorter the process recovery time. But this approach also does not consider the relationship between the snapshot storage server and the processing server, the process recovery can be delayed in low-bandwidth environment.

Akidau et al. [9] proposes a stream processing framework to process steam data in low latency. This framework reduces the recovering time by adjusting the interval of taking snapshots by stream data size. It is difficult to work this method in low-bandwidth environment because higher frequency of taking snapshot
requires much bandwidth.

III. Streaming Processing System Architecture

Fig. 1 illustrates the proposed stream processing system architecture that is featured by the location controller, which decides the location to store snapshots and replicated stream data for every process. We explain each component below:

Message Broker collects data from sensors or cameras located in the field and sends it to stream processing in the same server. Message Broker also replicates the data and sends it to another message broker for process recovery. The location of each data is managed by Data Location Manager. Stream Processing processes the stream data obtained from Message Broker, takes a snapshot, and stores the snapshot to Snapshot Datastore, which is working on another processing server. Stream Processing Manager manages the location and status of each Stream Processing function. Moreover, it manages the location of Stream Processing snapshots. Location Controller collects the load information from each server and network. It also calculates which server to store the replication data and snapshot and which server to redeploy each process. By making the distance between the data-stored server and the task-redeployed server close, the controller reduces the transferring delay. The detailed algorithm will be described in Section V If the result is different from the previous result, the management server informs the difference of each processing server through Data location Manager and Stream Processing Manager to change the destination where to store snapshots and replication data.

Fig. 2 illustrates the sequence of the proposed system. The Location Controller regularly calculates which server to store snapshots and replication data and which server to redeploy each process because the server load and network bandwidth may often change depending on the dynamic field in which people move continuously. If the calculation result differs from the previous one, the management server informs the difference of the processing server. Then Stream Processing and Message Broker in the processing Server change the destination server of data storage. When a server failure occurs, Stream Processing Manager deploys the process to the predetermined processing server. Then, the process can restore snapshots and replication data and resume the processing.

IV. Problem Discription

Deciding the appropriate combination of processes and servers requires to solve a combinational optimization problem.

Let us assume that $T = \{\tau_1, \tau_2, \cdots \tau_m\}$ is a set of tasks (which indicates computing processes) and $S = \{s_1, s_2, \cdots s_m\}$ is a set of servers. The decision about which servers to store a task's backup data and redeploy the task is the subset, L, of direct product T × S × S s.t. for $\forall \tau \in T$ and $\forall s \in S$; τ appears exactly once in L. We define three functions:

$$s_w: T \to S, s_d: T \to S \text{ and } s_t: T \to S, \quad (1)$$

where each converts to server whose tasks are to work, backup data stored, or set to redeploy, respectively

Task τ can be characterized by (c_τ, l_τ, d_τ^{store}, d_τ^{sum}), where c_τ represents the amount of CPU resources that τ requires, l_τ represents the requirement of latency, which is low if τ needs to be processed quickly, d_τ^{store} represents the data size consumed to store the task backup data (which indicates the snapshot and replicated stream data), and d_τ^{sum} is the amount of data transferred when task redeployment occurs. Server s has c_s, which represents s's CPU resource. When the task stores its backup data to another server, network bandwidth is consumed and is denoted as $E = \{e_1, e_2, \cdots, e_l\}$. For example, if τ working on s_k stores backup to $s_{k'}$ through eh and e_h',

$$e_h = e_h - c_t, \; e_h' = e_h' - c_t \quad (2)$$

We model the problem as follows:

$$\text{Minimize } TL(T,S,L) = \sum_{t \in T} b(\tau)/Z(\tau), \quad (3)$$

Subject to

$$\forall e \in E, \sum_{\tau \in T \setminus T(s)} E_1(\tau, e) d(\tau) + \sum_{\tau \in T(s)} E_2(\tau, e) Z(a) \le U(e), \quad (4)$$

$$\sum_{s \in S \setminus S(\tau)} x(r,s) = 1, \sum_{s \in S \setminus S(\tau)} y(\tau, s) = 1, \quad (5)$$

where $b(\tau)$ represents the size of backup data, $Z(a)$ represents the available bandwidth for transferring the backup data,$d(a)$ represents the amount of data transferred per second, and $U(e)$ represents the bandwidth of the network edge e. Moreover,$x(\tau, s)$ and $y(\tau, s)$ are 1 if task τ's backup data is stored to server s and redeployed to server s; otherwise, 0. $E_1(\tau, e)$ and $E_2(\tau, e)$ are also 1 if τ utilize e for storing backup data and transferring backup data to redeploy task; otherwise, 0.

V. Proposed Algorithm

A regular calculation to adapt to environmental changes is difficult due to the computation time; thus, we propose an algorithm that reduces the calculation time.

The algorithm intends to allocate tasks to one of the servers so that a server that stores backup data and a server that redeploys task is as the same as possible. However, a server storing the backup data and a server redeploying a task are not necessarily the same due to bandwidth constraints. In this case, we try to allocate tasks to servers so that the bandwidth between a server that restores backup and a server that redeploys task is as high as possible. It needs to satisfy the condition that the server that redeploys a task can execute the task and the bandwidth between the server in which the task is working and the server that stores backup data are enough to transfer backup data. The algorithm is shown in Algorithm 1. In summary, this algorithm can determine to allocate tasks to servers that are close to the best solution by appropriately sorting tasks and servers.

Algorithm 1 Combination decision algorithm

$S = \{s_1, s_2, \cdots, s_n\}$
$T = \{\tau_1, \tau_2, \cdots, \tau_m\}$
$B = S \times S = \{s_{1,1}, s_{1,2}, \cdots, s_{n,n}\}$
$S' = \{\}$
Sort T with l_τ in ascending order
for τ in T **do**
  for s in S **do**
    $S = \{\}$
    if for each e between $s_w(\tau)$ and s, $U(e) > d_\tau^{store}$ **then**
      $S' = S' \cup s$
    end if
    Sort B with the number of edge between $s_w(\tau)$ and s in ascending order
    for $s_{k,k}$ in B' **do**
      if CheckCPU(τ,s_k) **then**
        for each edge between $s_w(\tau)$ and s, $U(e) = U(e) - d_\tau^{store}$
        $T = T \setminus \tau$
      end if
    end for
  end for
end for
if $T \neq \{\}$ **then**
  for τ in T **do**
    Sort B with minimum $U(e)$ between $s_w(\tau)$ and s^{state} in decending order
    for $s_{i,j}$ in B **do**
      if CheckCPU(τ,s_k) **then**
        for each edge between $s_w(\tau)$ and s_i, $U(e) = U(e) - d_\tau^{store}$
        $T = T \setminus \tau$
      end if
    end for
  end for
end if

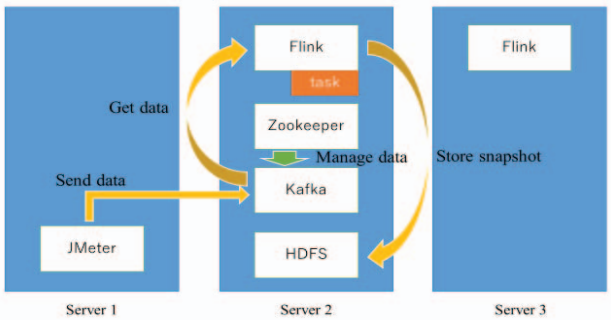

Figure 3. Stream processing system architecture.

In this paper, we assume that only one server is broken at a time. Thus, the server can accommodate more tasks than the server can execute if the tasks are working on different servers. Thus, task-redeployed servers can accommodate more tasks than their capacity if those tasks are working on different servers. CheckCPU checks if s can deploy τ by confirming if the tasks working on $s_w(\tau)$ are already allocated to s and if the sum of their c does not exceed c_s.

VI. Evaluation

A. Task Redeploying Time

1) Experimental setting: We evaluate how data transferring time affects task redeployment to another server. We prepared three servers; one server sends stream data, and two servers process the stream data and store backup data, and then measure the data transferring time for redeployment.

In this experiment, we build the system using JMeter [10], Flink, Kafka [11], Zookeeper [12], and HDFS working on Kubernetes cluster. The system is shown in Fig. 3. JMeter on Server 1 sends messages to Kafka on Server 2. Then, Flink on Server 2 gets stream data and processes it. Flink also stores snapshots to HDFS on Server 2. When the task redeployment occurs, the task gets a snapshot from HDFS. Then, the task gets the stream data generated from when the snapshot was taken to when the task was redeployed (about 20 MB).

We suppose three cases of experiments. The first case is redeploying the task to Server 2, so that the backup stored server and the task-redeployed server are the same. This means that this case does not require communication between servers. The second case is such that task redeployment is assigned to Server 3 and the bandwidth between Servers 2 and 3 is 100 Mbps. In the third case, task redeployment is assigned to Server 3, and the bandwidth between Servers 2 and 3 is only 10 Mbps.

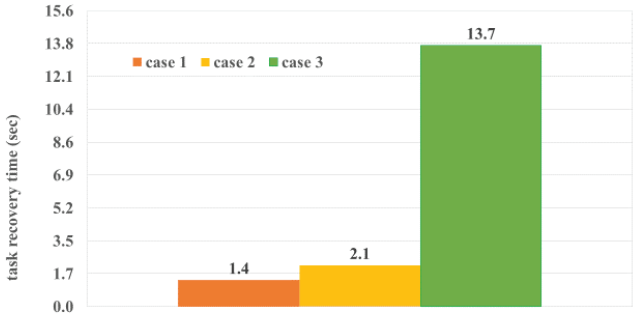

Figure 4. Task recovering time.

2) Experimental result: Fig. 4 shows the transferring time from when the task is deleted to when the redeployed task finishes processing the data generated from when the last snapshot was taken to when the former task was deleted. As shown in Fig. 4, the redeployment to Server 2, which stores backup data, results in a faster task recovery than the redeployment to Server 3, which does not store backup data. Moreover, the results of cases 2 and 3 indicate that appropriate decision about which server to store backup data and redeploy task makes the recovery time short if two servers are different. It is clarified that deciding where to store backup and where to redeploy processes is significant.

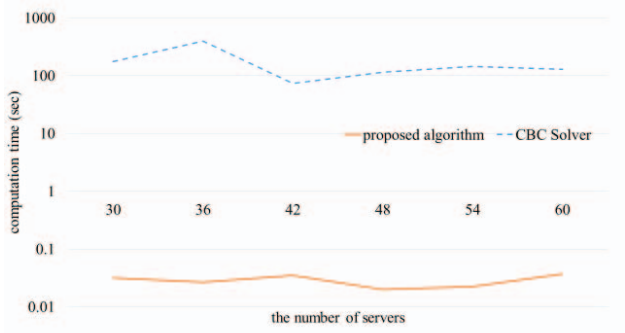

Figure 5. Calculation time when the number of tasks is varied from 10 to 120 by 10 while the number of servers is fixed to 30.

B. Computation Time of Proposed Algorithm

We investigated the computation time of the solution derived by the algorithm proposed in Section V by comparing it with the solution derived from the CBC solver. The computation time indicated as calculate allocation in Fig. 2.

We conduct two types of experiments. In the first experiment, the number of servers is fixed to 30 while the number of tasks is varied from 30 to 120 by 10. In the second experiment, the number of tasks is fixed to 30 while the number of servers is varied from 30 to 60 by 6. We use the time that the CBC solver derives the initial solution, which is not always optimal. The data plots shown below are the averages of 10 independent trials in the same machine.

1) Dependence on the number of tasks: Fig. 5 shows how the computational time changed according to the number of tasks. When there were 30 tasks, the calculation time of the CBC solver is about 20 seconds. However, when there are more than 70 tasks, the solver cannot complete calculation even in a whole day. On the other hand, the proposed algorithm can calculate in less than 1 second even when the number of tasks is 120. When the number of tasks is more than 30, the proposed algorithm can calculate over 1000 times faster than CBC Solver.

2) Dependence on the number of servers: Fig. 6 shows how computational time changes according to the number of servers. In this case, the calculation time of the proposed algorithm is almost the same when the number of servers changes. On the other hand, the calculation time of CBC Solver does not show an exponential increase like when the number of tasks is changed and does not change much.

C. Discussion

When there are many tasks while the number of servers is fixed, the CBC solver takes a long time to calculate. This

is because there are many combinations; however, there are few combinations to be resolved since computational resources and network resources are occupied by increased tasks.

On the other hand, when the number of servers increases while the number of tasks is fixed, the computational time of the CBC solver does not increase because the number of combinations to be resolved increases as the number of combinations increase.

The proposed method can finish calculation in less than 1 second in the both cases when the number of server increases while the number of tasks is fixed and when the number of tasks increases while the number of servers is fixed This is because the proposed method appropriately sorts tasks and servers. This sorting can exclude combinations that cannot clearly be resolved; thus, the number of candidates does not increase even if the number of tasks increases.

As mentioned above, we have verified that the proposed method is especially superior to the CBC solver when the number of tasks increases while the number of servers is fixed.

VII. Conclusion

Stream processing is a promising technology for real-time applications that need to continuously return responses within milliseconds or seconds. Especially in the distributed computing environment with a small bandwidth, the relationship between where to store backup and where to deploy processes is a crucial factor for fast recovery.

Therefore, to reduce the recovery time, we propose a distributed system architecture that can decide which server to store a snapshot of a process and replication data and which server to redeploy processes. We also propose a fast algorithm to appropriately decide the combination of server and process to meet the application's latency requirement. We experimentally showed that the proposed algorithm can calculate over 1000 times faster than the CBC solver when the number of processes is large. This means that the algorithm can allocate the combinations of servers and processes in a short time following the dynamic change of the field.

For future study, we plan to implement the location controller and to experiment in the actual environment. We will also consider the interval of taking snapshots because it affects the throughput for recovery.

Iijima and Amemiya conducted the research; Iijima did experiment and analyzed the data; Iijima wrote the paper; Miyoshi and Ogawa gave advices about the research.

References

[1] Ning Chen, Yu Chen, Yang You, Haibin Ling, Pengpeng Liang, and Roger Zimmermann. "Dynamic Urban Surveillance Video Stream Processing using Fog Computing." Proceedings - 2016 IEEE 2nd International Conference on Multimedia Big Data, BigMM 2016, pages 105–112, 2016, doi: 10.1109/BigMM.2016.53

[2] Hisatoshi Yamaoka et al. "Dracena: A Real-Time IoT Service Platform Based on Flexible Composition of Data Streams." Proceedings of the 2019 IEEE/SICE International Symposium on System Integration, SII 2019, pages 596–601, 2019, doi: 10.1109/SII.2019.8700465

[3] Marcos Dias de Assunc¸˜ao, Alexandre da Silva Veith, and Rajkumar Buyya. "Distributed Data Stream Processing and Edge Computing: A Survey on Resource Elasticity and Future Directions. " Journal of Network and Computer Applications, 2018, doi: 10.1016/j.jnca.2017.12.001

[4] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache Flink: Stream and Batch Processing in a Single Engine." Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36:28–38, 2015

[5] Atashbar, N. Z., & Rahimi, F. (2012). "Optimization of Educational Systems Using Knapsack Problem." International Journal of Machine Learning and Computing, 2(5), 552.

[6] Coin-or branch-and-cut mip solver. https://projects.coin-or.org/Cbc. (Accessed on 10/01/2019).

[7] Konstantin et al. "The Hadoop Distributed File System." In MSST, volume 10, pages 1–10, 2010, doi: 10.1109/MSST.2010.5496972

[8] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. "Discretized Streams : An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters." Presented as part of the, 2012.

[9] Slava Chernyak et al. "MillWheel : Fault-Tolerant Stream Processing at Internet Scale." Proceedings of the VLDB Endowment, 6(11), 1033-1044. doi: 10.14778/2536222.2536229

[10] Apache Jmeter. https://jmeter.apache.org/. (Accessed on 10/01/2019).

[11] Jun and others Kreps, Jay and Narkhede, Neha and Rao. "Kafka: A Distributed Messaging System for Log Processing." Proceedings of the NetDB, 2011.

[12] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." USENIX Annual Technical Conference, 8:11–11, 2010.