# Dynamic Reconfiguration and Resource Provisioning in Pi Clusters

Rohit Das
*Dept. of Electrical Engineering and Computer Science*
*Indian Institute of Technology Bhilai*
Raipur, India
rohitd@iitbhilai.ac.in

Dr. Subhajit Sidhanta
*Dept. of Electrical Engineering and Computer Science*
*Indian Institute of Technology Bhilai*
Raipur, India
subhajit@iitbhilai.ac.in

***Abstract*—In a resource-constrained computing setup, efficient allocation of resources is essential. Edge clusters consisting of small, single-board computers are cheaper, easily deployable and re-configurable. During unavailability of cloud services or intermittent network coverage, edge clusters can be helpful. To enable feasible usage of such clusters, algorithms to manage failure and overloading are essential. Most of the related works lack in the regard of using ARM devices for such clusters.**

***Index Terms*—edge computing, raspberry pi, distributed systems, resource-constrained, resource provisioning**

## I. Introduction

With the emergence of cloud computing, computational resources have become more flexible and easily available. Be it deep learning analytical models, online software subscriptions or data storage, cloud computing is the most popular choice. However, using cloud services require stable bandwidth, SLA's promising competitive low values of downtime and fast results. Where cloud fails, edge computing comes in. Using single-board computers to create an edge cluster, we can offload cloud services to the edge, speeding up data processing and lowering transmission latency. Such lightweight clusters need to have algorithms in place to manage node failures, overloading and reconfiguration and be in tandem with quality of cloud services.

Single-board computers like Raspberry Pi usually come with a quad-core processor and at most 4GB of RAM. Nowadays, in the advent of data analytics, the above-mentioned resources are too less for a single node to handle vast amounts of data. So, we are faced with the following problems:

- To enable fast processing of a huge in-stream of data, either the rate of data fed into the cluster must be regulated, or the data stream must be parallelized amongst multiple nodes.
- Some nodes will be overburdened with processes taking up a lot of resources, like applications involving deep-learning or neural networks, object-recognition from a live video feed, so efficient load-balancing algorithms are essential.
- In case of failure of a node, container running on the failed node must be migrated before-hand to a standby node, and the new node must be up and running with as less downtime as possible.

## II. Related Works

Live migration intends to transfer user control or a process from one node to another in a cluster in case of failures or crashes. While live migration is good and suitable for systems set up over LAN having bandwidths in the range of 1-40 Gbps, VM hand-off [1] focuses mainly on the edge, having a WAN connection of bandwidth 5 to 25 Mbps.

The process of VM hand-off [1] can be broken down into 3 parts: **delta encoding**, **deduplication** and **compression**. Each step of the optimizing process is a module in the VM Hand-off pipeline. The aim is to send only the difference between the modified data in the source VM and the base VM at the destination. The pipeline switches between algorithms to make the best use of both the CPU cores and the network bandwidth.

Redundancy live migration [2] has been shown to be a significant improvement over the pre-copy, post-copy or the hybrid live migration algorithm.

Service migration reduces service latency, and user applications are migrated to the closest MEC(Mobile Edge Clouds) only when it has benefits surpassing the negative effects of downtime. The migration is performed by sending files synchronously over the network. Meaning, only files which have changed significantly in comparison to the destination MEC are sent over. To optimize the amount of data sent out during migration, a three-layered approach [3] is used. **Instance layer** migration consists of migrating only application-specific information like the running state of the application. **Application layer** is involved when the entire application is not found at the destination MEC, and the same is cloned from the source. The topmost **Base layer** is only involved when the container or VM used itself is not present in the destination, and needs to be packaged and sent from the source.

Dynamic Provisioning of resources [4] for a resource-constrained cluster will be helpful to deal with situations like flash crowds, when a sudden amount of huge traffic overwhelms the cluster and fast enough to rearrange the topography of the cluster nodes to accommodate for the sudden surge. The cluster nodes can be tiered to enable handling of the computations and requests at multiple stages. Predictive and Reactive provisioning [5] can handle flash crowds or deviations from predicted long-term behaviour. Analytical models

to allocate servers to a tier based on the estimated workload and accounting for multiple requests that comprise a session and the stateful nature of session-based Internet applications will also help in handling surges.

## III. Cluster Provisioning and Reconfiguration

Using single-board chips with low resources make the cluster prone to overloading and failures. Reconfiguring the network in case of node failures, and optimal provisioning of nodes for more intensive computations, as well as scaling down when not needed, is essential for the edge cluster.

Linux Containers (LXC) will be running on each node in the cluster. VMs will not be suitable for single-chip boards that we will be using as VMs take up more resources than containers, and hence more bandwidth will be needed to exchange data among nodes.

There will be two parts to the algorithm: failure handling and overload management. The entire cluster will be segregated into two parts - master cluster, and worker cluster. The master cluster will consist of nodes which can be assigned as pseudo-masters, or super peers. The master node will be handling the worker cluster, consisting of worker nodes. Experiments have shown that having a master node with at least 4 GB RAM prevent the node from crashing due to overload.

### A. Node Failure Handling

Any node may fail due to random crashes, overloading or environmental factors. Each worker node will have an associated standby node to fall back on in case of node failure. Frequent delta migration of data encapsulated by containers among nodes will prevent data loss and mitigate downtime.

The master cluster, containing the master nodes, as well as the worker cluster, containing the worker nodes, will be having a replication factor of $n + k$, $n$ being the cluster size of each type. Each worker node will be allocated a standy node from its respective cluster. Container information like current memory and disk state shall be sent over after every interval of $T$ seconds.

For the first time, entire container information will be sent over to the standby node. In consecutive intervals, only the delta difference in states of active and standby nodes will be sent over. That way, information sent over the network and overall bandwidth usage can be minimized.

Whena standby node does not receive any information after $T + t$ seconds, $t$ being the failure tolerance interval, it will inform the active master, take over from the failed active node and mark itself as an active node. The master will allocate a standby node from the pool and the new node will pick up from where the failed node left off.

For a master node, two nodes will be on standby, and will receive delta differences from the active master. When the active master node fails, a bully leader election will decide the new master node, which will then broadcast its own MAC to the worker cluster.

**Algorithm 1** Node Activation on Worker Node

```
1:  function ACTIVATE_NODE(standby_node_MAC)
2:      // Checks if active node is down
3:      if time.now() > last_seen + T + t then
4:          send msg(MAC,active_node) to curr_master
5:          // Standby node takes over
6:          restart container from last_state_recvd
7:          send container data to standby node
8:          while true do
9:              send delta_diff(curr_lxc_state, last_state)
10:             last_state ← curr_lxc_state
11:             last_seen ← time.now()
12:             sleep (T)
13:         end while
14:     end if
15: end function
```

**Algorithm 2** Node Activation on Master Node

```
1:  function ACTIVATE_NODE(standby_node_MAC)
2:      // Marks respective active node as down
3:      active_node ← hash_list [standby_node_MAC]
4:      hash_list [active_node].state = down
5:      for node in hash_list do
6:          if hash_list [node] == NULL AND
7:  hash_list [node].state != down then
8:              standby_node_MAC.activate_node (node)
9:              exit
10:         end if
11:     end for
12: end function
```

A list of available nodes with their hostname and MAC will be maintained at every node. It will also contain important information on the nodes like their status (dead or alive), if they are an active or standby node, their corresponding standby or active node, etc. When a worker node fails, only the active master can change its own record. It can broadcast the new node after every $T'$ interval. When a master node fails, the new master node will broadcast itself and the changes in the record, thus reducing bandwidth usage.

### B. Node Provisioning through Reconfiguration

Monitoring the nodes in the cluster can prevent failure due to overloading. Any under-utilized nodes from the cluster can be removed to optimize resource usage. Using YourKit, memory and disk usage can be monitored on each node.

When resource usage is above a certain threshold, the active node can request the active master to add more nodes to the cluster to distribute the jobs and prevent overloading. If two free nodes are found in the pool, the master assigns one as the

active node, and the other as a standby node, and adds them both to the cluster.

**Algorithm 3** Additional Node Allocation

```
1: function ALLOCATE_NODE(active_node_MAC,stats)
2:     // Checks if stats are above threshold
3:     if stats.memory > Mem_thresh. OR stats.disk_usage
   > Disk_thresh. then
4:         active_node = NULL
5:         standby_node = NULL
6:         for node in hash_list do
7:             if hash_list [node] == NULL AND
8: hash_list [node].state != down then
9:                 if active_node == NULL then
10:                    active_node = node
11:                else if standby_node == NULL AND node
   != active_node then
12:                    standby_node = node
13:                else
14:                    exit
15:                end if
16:            end if
17:        end for
18:    end if
19: end function
```

If only one free node is found, the node will be assigned as an active node and added to the cluster. An existing standby node may be reassigned to this active node. If no free node is found, one of the standby nodes will be assigned as an active node, and a standby node will reassigned from the cluster.

Similarly, when resource usage falls below the lower threshold, nodes can be unassigned to scale-down the cluster. Important info on the node will be broadcast to all workers before taking it out. (How to scale down?)

## IV. Experimental Setup and Implementation

The experimental setup consists of creating a cluster using Raspberry Pi 3 Model B+ single-board computers. Each Pi consists of a 1.4GHz 64-bit quad-core processor, 1GB LPDDR2 SDRAM and 32GB micro-SD cards. Each node is running Raspbian OS based on Linux. The networking is done using wireless LAN, and each node has its own micro-USB charger. A file with the list of MACs of all nodes in the cluster will be maintained in each node. That way, we can easily find the IPs of each node when the cluster starts up.

Apache™ Hadoop® and Apache Spark™ are installed in each node to distribute jobs across the cluster. One node is chosen as the name node, which will be responsible for starting HDFS and Yarn across the cluster.

Python libraries like Tensorflow, Keras and Elephas help to distribute Deep Learning tasks across a Spark Cluster. However, since we are starting the cluster from the name node, due to low resources, the jobs start to lag.

## References


[1] K. Ha, Y. Abe, T. Eiszler, Z. Chen, W. Hu, B. Amos, R. Upadhyaya, P. Pillai, and M. Satyanarayanan, "You Can Teach Elephants to Dance: Agile VM Handoff for Edge Computing," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC 17, (New York, NY, USA), Association for Computing Machinery, 2017. https://doi.org/10.1145/3132211.3134453.

[2] K. Govindaraj and A. Artemenko, "Container Live Migration for Latency Critical Industrial Applications on Edge Computing," in *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, pp. 83–90, 2018.

[3] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis, "Live Service Migration in Mobile Edge Clouds," *IEEE Wireless Communications*, vol. 25, no. 1, pp. 140–147, 2018.

[4] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic Provisioning of Multi-tier Internet Applications," in *Second International Conference on Autonomic Computing (ICAC'05)*, pp. 217–228, 2005.

[5] S. Daniel and M. Kwon, "Prediction-based Virtual Instance Migration for Balanced Workload in the Cloud Datacenters," 2011. http://scholarworks.rit.edu/article/985.

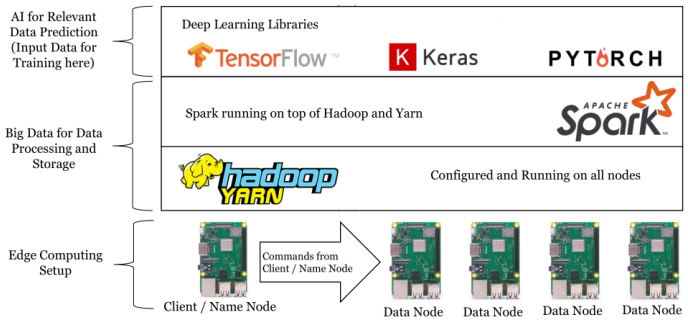


Fig. 1. Edge Cluster Setup using Raspberry Pi