Supplementary Materials:

Blogging Website and Hadoop Scenario Experiments, Supporting Chinese Remainder Theorem Use Cases Proofs, Configuration Guidelines, and Openstack Filter Scheduler Configuration

Alexandru G. Bardas[1]*, Sathya C. Sundaramurthy[2]†, Xinming Ou[3], Scott A. DeLoach[4]
[1]*University of Kansas,* [2]*DataVisor,* [3]*University of South Florida,* [4]*Kansas State University*

This document includes additional materials for [1].

1 Performance Testing

Additional scenarios mentioned in Section 3 from [1]:

1.1 Blogging Website

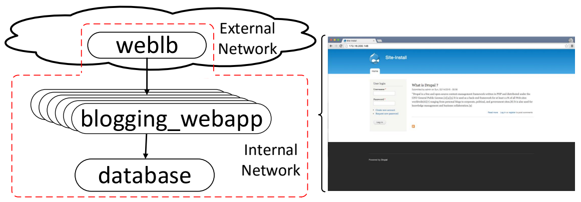

Figure 1: Blogging website blueprint. `blogging_webapp` is implemented by a homogenous cluster of Drupal instances.

A basic blueprint of the blogging website is pictured in Figure 1; arrows indicate dependencies between instances implementing the defined roles: `weblb`, `blogging_webapp`, and `database`. We used a simple but common design that includes a load-balancer (Varnish or Nginx), a cluster of web applications (three instances running Drupal [3]), and a MySQL database. Based on the scenario the only component that runs in a high-availability cluster format of active instances is the web application (`blogging_webapp`). Thus, we chose the `blogging_webapp` to be the component that is changed during this experiment (Table 1). All the measurements are averages of 20 experiment runs. Once we collected the baseline measurements, we started replacing instances while http-perf was running and collecting metrics (Table 1).

The *Baseline* column captures the system's metrics without performing any replacement actions; specifically the benchmarking tool (http-perf) was performing read operations on the database. One hundred and fifty (150) connections was the maximum number of connections our test scenario was able to handle. A higher number of connections would overwhelm our test scenario and would cause failed requests under baseline conditions. Next we ran the replacement operations under the same http-perf load to see if and how much the replacement process delays the requests' processing (Table 1). It is worth noting that the difference between the baseline and the replacement results is statistically non-significant (when performing the T-test), and that, overall, there were no or very few HTTP error responses during the replacement operations. The error responses (dropped requests) are generated when dependent instances (`weblb` in this case) reload the updated configuration and establish connections with the new `blogging_webapp` instances. All this while still processing the incoming http-perf requests.

1.2 Hadoop Scenario

We deployed and managed a Hadoop setup (CDH5 version of HDFS [4] and YARN [5]) using Cloudera Manager [2] on top of MTD CBITS. Our Hadoop deployment had one master (`cloudera_master`) and three compute nodes (`cloudera_compute_node`) (see Figure 2). The `dns_server` and the `cloudera_manager` are instances utilized to administer the actual Hadoop deployment. Compute nodes are mainly worker nodes that conduct MapReduce jobs, while the master node orchestrates the job execution and ensures that compute nodes are processing different parts of the data. Using the *Random Text Writer* job we have generated a total of 15 GB of data. Next we used Hibench to launch the default *Sort* job on the generated data and establish a baseline measurement (Table 2). Since a Cloudera Hadoop deployment supports only one active master at a time, we have focused our evaluation on replacing the compute nodes: one compute node and the whole compute node cluster.

We observed the following behavior: If the job started

Aggregated results from **20** experiment runs Each experiment run: **150,000** requests sent using **150** concurrent connections								
	Response time (sec)		Total time		Server Processing Rate (req/sec)		HTTP Error Resp.	
	Avg.	stdev	Avg.	stdev	Avg.	stdev	Avg.	stdev
Baseline	0.787	0.040	13min 8sec	40 sec	191.012	10.369	0	0
Replacing one webapp	0.793	0.047	13min 14sec	47 sec	189.667	11.195	0.25	0.79
Replacing webapp cluster	0.797	0.057	13min 17sec	58 sec	189.046	12.235	13.25	37.57

Table 1: Drupal blogging deployment – performance overhead of carrying out **one** replacement operation: replacing one `blogging_webapp` instance and replacing the whole `blogging_webapp` cluster.

		Average map time	Average shuffle time	Average merge time	Average reduce time	Elapsed time	State
Baseline		9sec	20sec	8sec	43sec	5min 38sec	SUCCEEDED
Replacing one compute instance	After job started	22sec	1min 2sec	3sec	32sec	8min 28sec	SUCCEEDED
	Before job started	15sec	19sec	4sec	34sec	5min 34sec	SUCCEEDED
Replacing compute cluster	After job started	53sec	1min 36sec	2sec	41sec	13min 21sec	SUCCEEDED
	Before job started	10sec	22sec	5sec	55sec	6min 3sec	SUCCEEDED

Table 2: Hadoop Deployment – **Sort** job on **15 GB** of data.

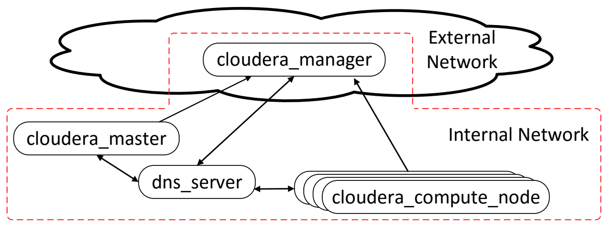

Figure 2: Cloudera Hadoop Deployment (CDH5). `cloudera_compute_node` is the only service implemented by a high-availability cluster of active instances.

before the new (fresh) compute instance was added to the compute cluster and the old (to-be-replaced) one removed, then the old instance was assigned to participate in processing the job. When the old instance is removed, the data to be processed on it will be relocated to other nodes by `cloudera_master`. Thus when performing the replacement after the job has started, the "map time" and "shuffle time" (see Table 2) were significantly impacted. As a result the Sort job needed a longer time to finish. Once the old instance was removed, the fresh instance did not affect the completion of the current job. In case the replacement completed before the job started, there was no compelling difference between the baseline measurements and replacing one compute instance. We have noticed a similar behavior when replacing all the compute instances (the whole compute cluster). As shown in Table 2, a whole compute cluster replacement after the job started would highly impact the completion time of a running Sort job compared to the baseline measurements. However, if the replacement completed before the job started, the difference was not significant.

MTD CBITS triggered the changes in the Hadoop deployment through Cloudera in an organized way, by notifying the Cloudera manager about the changes it should perform. The manager can be configured to act instantly (similar to the experiments presented in Table 2) or it can wait for certain conditions to be fulfilled (e.g., certain jobs should complete). Therefore, the performance overhead of proactively replacing compute nodes can be negligible or non-existent if it is performed before a job starts. Otherwise the performance impact can be contained by trying to perform the changes when lower-priority jobs are running or during off-peak times.

2 Chinese Remainder Theorem

Case 3 – Section 4.2 "Adaptation Points Placement"

Assuming $A, B \in \{X, Y_1, \ldots, Y_{l-1}\}$ AND $A \neq B$ then

$$t_{min} = \mathtt{min}(start\_time_{T_r(A)}, start\_time_{T_r(B)})$$
$$t_A = start\_time_{T_r(A)} - t_{min}$$
$$t_B = start\_time_{T_r(B)} - t_{min}$$

$$\begin{cases} T_r(A) \text{ and } T_r(B) \text{ are NOT pairwise coprime} & (1) \\ t_A \equiv t_B \mod \gcd(T_r(A), T(B)) \text{ is FALSE} & (2) \end{cases}$$

$(1) \Rightarrow \gcd(T_r(A), T(B)) = z,\ z \neq 1$

In other words

$T_r(A) = z \times s_1$, s_1 is an Integer

$T_r(B) = z \times s_2$, s_1 is an Integer

$(2) \Rightarrow t_A \neq t_B$

If m would exist then

Assuming $T_r(A)$ starts before $T_r(B)$, then

$$t_{min} = start\_time_{T_r(A)}$$
$$t_A = 0$$
$$t_B = start\_time_{T_r(B)} - start\_time_{T_r(A)}$$

Furthermore

$$\begin{cases} m = t_B + q \times T_r(B) & ,q \texttt{ is an Integer} \\ m = p \times T_r(A) & ,p \texttt{ is an Integer} \end{cases}$$

$$\begin{aligned}
&\Rightarrow p \times T_r(A) = t_B + q \times T_r(B) \\
&\Leftrightarrow t_B = q \times T_r(B) - p \times T_r(A) \\
&\Leftrightarrow t_B = q \times z \times s_2 - p \times z \times s_1 \\
&\Leftrightarrow t_B = z \times (q \times s_2 - p \times s_1) \\
&\Rightarrow t_B \texttt{ is divisible by } z \\
&\Rightarrow t_B \texttt{ is divisible by } \gcd(T_r(A), T_r(B)) \\
&\Rightarrow 0 \equiv t_B \mod \gcd(T_r(A), T_r(B)) \texttt{ is TRUE} \\
&\Rightarrow t_A \equiv t_B \mod \gcd(T_r(A), T_r(B)) \texttt{ is TRUE}
\end{aligned}$$

which CONDRADICTS the initial assumption (2), therefore m does **not** exist.

If $T_r(B)$ starts before $T_r(A)$ then the proof is symmetric to the one presented above.

Case 4 – Section 4.2 "Adaptation Points Placement"

Assuming $A, B, C, D \in \{X, Y_1, ..., Y_{l-1}\}$

$$\begin{cases} T_r(A), T_r(B), T_r(C), T_r(D) \texttt{ NOT pairwise coprime} & (3) \\ t_A \equiv t_B \mod \gcd(T_r(A), T(B)) \texttt{ is FALSE} & (4) \\ t_C \equiv t_D \mod \gcd(T_r(C), T(D)) \texttt{ is TRUE} & (5) \end{cases}$$

If m **would exist then** based on the proof above:

$$\begin{cases} t_A \equiv t_B \mod \gcd(T_r(A), T(B)) \texttt{ is TRUE} \\ t_C \equiv t_D \mod \gcd(T_r(C), T(D)) \texttt{ is TRUE} \end{cases}$$

Contradicts the initial assumption (4), therefore m does **not** exist.

3 Configuration Guidelines

Every environment has a certain budget – restrictions regarding the number and the frequency of adaptation points. The frequency and the number of adaptation points depend directly on the size of the T_r values. Moreover, the lower bound of the T_r values rely upon the resources of the environment (*ch* and *d*) and also the performance penalty the environment can afford. On the other hand, the objective of controlling the number of various window sizes is mainly influenced by the starting times and the user-introduced delay. The lower bounds for the adaptation intervals are imposed by the environment's capabilities. One of the primary challenges is to find a balance between the budget (number and frequency of adaptation points) and the main objective (controlling the number of various window sizes). Once the lower bounds or the preferred values for the adaptation intervals are known, one can determine the parameter values (starting times and user introduced-delay) along with the adaptation order by generating all potentially-interesting options.

Based on Section 4, we are proposing a few guidelines to consider when choosing the parameters:

$\texttt{min\_Tr} = \texttt{min}(T_r(X), T_r(Y_1), ..., T_r(Y_{l-1}))$

- If $t_X, t_{Y_1}, ..., t_{Y_{l-1}} \le \texttt{min\_Tr}$ then:
  1. *max attack window* $\le \texttt{min\_Tr}$
  2. the range of window sizes $\in$ *mod* $\texttt{min\_Tr}$
- Parameters that fall under Case 3 ensure that one adaptation point translates to one interruption.
- Case 3 parameters with the same T_r values for all nodes usually result in a more reduced range of various windows sizes. The sizes of the windows depend on the difference between the starting times of the adaptation intervals.
- Interchanging nodes starting times may also impact the window size distribution.
- The maximum time an attacker may spend on a target node in case of a successful attack: $T_{\texttt{target}}(X) =$ *max attack window* - $T_a(X)$

4 OpenStack Filter Scheduler

Section 5 "Discussion and Limitations"

The latest versions of Openstack use the Filter Scheduler [6], which supports filtering and weighting to enable informed decisions on where a new instance should be created. Depending on the employed filters and the weighting approach, the schedulers' complexity ranges from simple (basic) to very complex. Fig. 3 illustrates a sample decision process when using the Filter Scheduler.

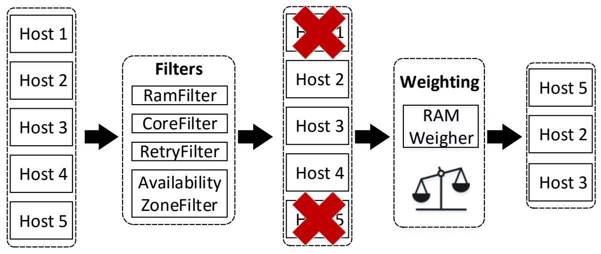

Figure 3: OpenStack sample Filter Scheduler [6]

The scheduler uses various filters to find the potential destination hosts and then orders these hosts based on a weighting scheme. For example, the RamFilter only passes hosts that have sufficient RAM for the new instance. On the other hand, the RAMWeigher is used to sort the valid hosts based on their available RAM.

In our deployment, we used Openstack (Icehouse distribution) with the default scheduler options.

```
/etc/nova/nova.conf:
 scheduler_default_filters=RetryFilter,
  AvailabilityZoneFilter, RamFilter,
  ComputeFilter,ComputeCapabilitiesFilter,
  ImagePropertiesFilter,ServerGroupAntiAffinityFilter,
  ServerGroupAffinityFilter
  ram_weight_multiplier=1.0
```

Figure 4 illustrates the distributions of the `webapp` instances on the 13 physical compute hosts when initially deployed, and after two whole-cluster webapp replacements.

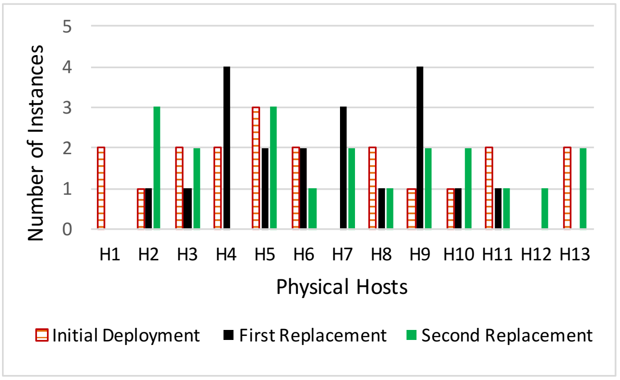

Figure 4: The distribution of 20 webapp instances across 13 physical hosts (initial deployment and two whole-webapp-cluster replacements)

References

[1] A. G. Bardas, S. C. Sundaramurthy, X. Ou, and S. A. DeLoach. MTD CBITS: Moving Target Defense for Cloud-Based IT Systems. In *ESORICS*, 2017.

[2] Cloudera Manager – accessed 8/2017. http://bit.ly/1KrT5F8.

[3] Drupal – accessed 8/2017. https://www.drupal.org/.

[4] Hadoop HDFS – accessed 8/2017. http://bit.ly/1bEdmeT.

[5] Hadoop YARN – accessed 8/2017. http://bit.ly/1IpM3BU.

[6] OS Filter Scheduler – accessed 8/2017. http://bit.ly/1ETQkLq.