The flaw of our current algorithms is that it is theoretically possible for many MUSIC nodes that are attempting *put*s to starve out a MUSIC node executing the while loop at lines 14-15. If experiments show that this is a problem in practice, it can be ameliorated by requiring a random wait between successive attempts to *put*.

6. Implementation and Results

6.1 Implementation

We implement a version of MUSIC based on the design in section 4 with our *dataStore* built on Cassandra and our *lockStore* built on Zookeeper. To obtain the necessary abstractions from the *lockStore* we modified the lock recipe of Zookeeper. We implemented the MUSIC algorithms in Java 1.8 and deployed it as web service running on the Apache Tomcat web server, accessible to the client through a REST API. Our implementation of MUSIC, that is currently being deployed in production, can be run with as many replicas as needed.

6.2 Micro-benchmarks

In this section, we evaluate the performance of MUSIC using micro-benchmark experiments. To evaluate MUSIC AP operations, we compare the MUSIC *put* and *get* operations with the Cassandra *put* and *get* operations, where reads and writes are performed on a single replica. To evaluate the CP operations, we first implement an operation called *CP Put*, in which we create and acquire a lock to a key, perform a *criticalPut* on that key, and finally release the lock. We compare this *CP Put* with an analogous operation in Zookeeper in which we replace the *criticalPut*, with a Zookeeper write to the key (specifically, the node created in the Zookeeper file system for this key). To illustrate the advantage of MUSIC over a system like Zookeeper, that only allows CP operations for writes, we compare the MUSIC AP operations with Zookeeper *gets* and *puts*.

Experiment setup: Our experimental setup consists of a 3 replica MUSIC cluster deployed across two Openstack [16] sites, where each replica has 2 VCPUs and 4 GB of RAM. We emulate users issuing a fixed number of requests to each of these systems using JMeter [17]. We vary the number of the simultaneous users by increasing the number of threads to saturate the systems and measure the median throughput achieved in each of the three systems over the duration of the experiment. We repeat our throughput measurements multiple times with each system and alternate uniformly between all three systems to mitigate the impact of any external performance variability on our results.

Figure 11 shows a bar graph comparing the performance of Cassandra and MUSIC *get/put* AP operations. The bars are grouped along the X-Axis based on the operation (*get/put*) and the bars (and error bars) correspond to the average (and standard deviation) in the median throughput observed across multiple experiments with both MUSIC and Cassandra. From the figure, we can see that the MUSIC AP operations performs comparably with Cassandra get and put, with a median throughput of nearly 1700 ops/sec while supporting as many as 2000 parallel requests.

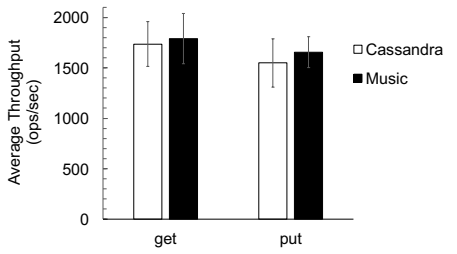

Figure 11: Comparing the performance of *get* and *put* operations of MUSIC and Cassandra, to show that MUSIC performs comparably to Cassandra despite the additional overhead of ensuring that the MUSIC *puts* never overwrite a locked key.

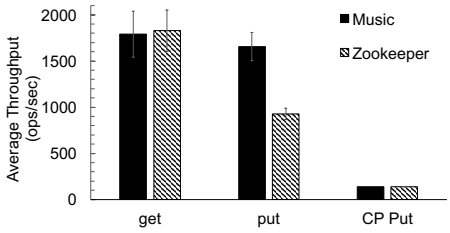

Figure 12: Comparing the performance of MUSIC and Zookeeper. While MUSIC *CP put* is comparable to Zookeeper *CP Put*, MUSIC *put* is nearly twice as fast as the Zookeeper *put*.

Figure 12 shows a similar bar graph comparing the performance of it get, *put* and *CP Put* operations across MUSIC and Zookeeper. Clearly, the *get* operation is comparable across both the systems, since a *get* happens locally at a single node for both systems. However, MUSIC *put* is significantly faster than Zookeeper *puts*, which incurs the Paxos latency for all writes. On the other hand, MUSIC *puts* can happen locally within a node, thereby being faster than Zookeeper writes. Finally, we also see that the performance of both systems are comparable for *CP Puts*, which shows that MUSIC performs similar to Zookeper even in the most adversarial scenario, when locks are acquired and released between every put operation.

6.3 Multi-site Application Scheduler

Our multi-site application scheduler (based on the example in section 3.1) experiment evaluates the impact of the choice of consistency on the performance and efficiency of distributed scheduling.

6.3.1 Experiment setup

We use our in-house holistic application scheduler for OpenStack [16] as the local scheduler at each cluster. The exper-