Using Simulation to Explore Distributed Key-Value Stores for Exascale System Services

Ke Wang
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, USA
kwang22@hawk.iit.edu

Abhishek Kulkarni
Department of Computer Science
Indiana University
Bloomington, IN, USA
adkulkar@cs.indiana.edu

Xiaobing Zhou
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, USA
xzhou40@hawk.iit.edu

Michael Lang
Ultra-Scale Research Center (USRC)
Los Alamos National Laboratory
Los Alamos, NM, USA
mlang@lanl.gov

Ioan Raicu
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, USA
iraicu@cs.iit.edu

ABSTRACT

Most of HPC services are still designed around a centralized paradigm and hence are susceptible to scaling issues. P2P services have proved themselves at scale for wide-area internet workloads. Distributed key-value stores (KVS) are widely used as a building block for these services, but are not prevalent in HPC services. In this paper, we simulate KVS for various service architectures and examine the design trade-offs as applied to HPC workloads to support exascale systems. Via simulation, we demonstrate how failure, replication, and consistency models affect performance at scale. Finally, we emphasize the general use of KVS to HPC services by feeding real workloads to the simulator.

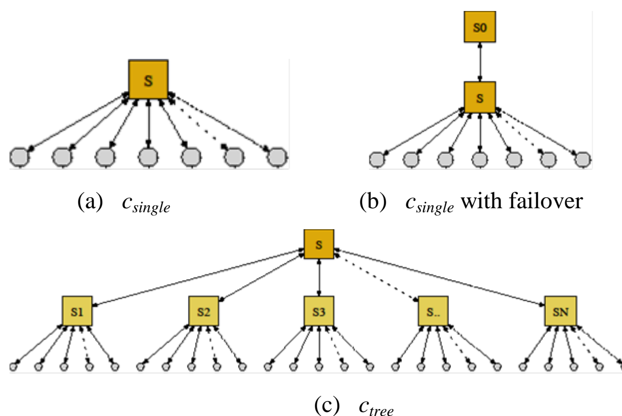
1. Introduction

Leadership-class systems have been managed using manual approaches under a single management domain. Many HPC services are designed around a centralized server hence suffer from scalability problems. Such concerns suggest a move toward scalable distributed system designs. The specific goal is to evaluate the different distributed key-value store (KVS) designs for exascale system services, such as job launch, I/O forwarding, and monitoring. These services all need to operate on large volumes of data in a consistent, resilient and efficient manner at extreme scales. These requirements are consistent with those of large-scale distributed data centers, such as Amazon, Facebook and LinkedIn, in which, NoSQL data stores – Distributed Key-Value Stores (KVS), in particular – have been used successfully. We assert that by taking the particular needs of HPC into account, we can use KVS for HPC services to help resolve many of our consistency, scalability and robustness concerns. We have used KVS to implement several real systems, such as a many task computing execution fabric, MATRIX [1][2][3] where KVS is used for task submission, dependency, and progress information; and the fusion distributed file system, FusionFS [4], where the KVS is used in tracking metadata.

2. A Taxonomy for KVS

We developed a four-dimensional taxonomy to classify and specify these requirements. By combing these four dimensions, we can define service architectures. The four components are: **Data Model** defines how a service distributes and manages its data, such as centralized and distributed manners with partitions; **Network Model** dictates the interconnection topology of a service's components, such as structured and unstructured overlay; **Recovery Model** specifies how a service deals with component failures (fail-over, checkpoint-restart and roll-forward); **Consistency Model** pertains to how rapidly data modifications propagate across the servers. Depending on the data model, a service can implement strong or eventual consistency.

Architectures from the taxonomy are depicted in Figure 1 and Figure 2. c_{tree} is a service architecture with centralized data model and tree-based overlay network. d_{fc} has distributed data model with fully-connected overlay network whereas d_{chord} is distributed data model and has a Chord overlay network [5].



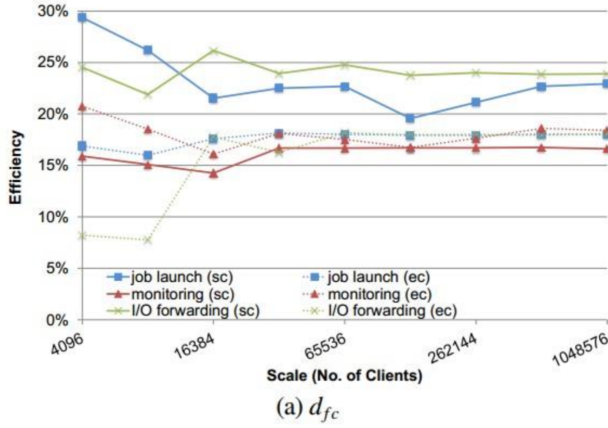
(a) c_{single} (b) c_{single} with failover

(c) c_{tree}

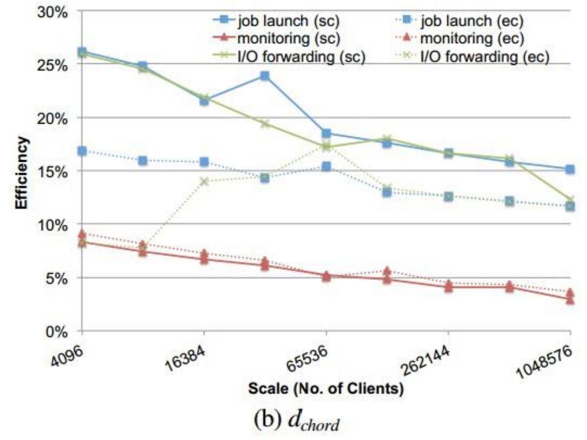
Figure 1: Centralized service architectures

3. Simulating Key-Value Stores

We simulate KVS with major components identified by the taxonomy. Each simulation consists of millions of clients that connect to thousands of shared servers. The workload for the KVS simulation is a stream of PUTs and GETs. At this point, each client connects to a server and sends synchronous blocking requests as specified by a workload file. Servers are modeled by two queues: a communication queue for sending and receiving messages and a processing queue for handling incoming requests that can be satisfied locally. Requests not handled locally are forwarded to another server. The two queues are processed concurrently, and the requests within one queue are processed sequentially.

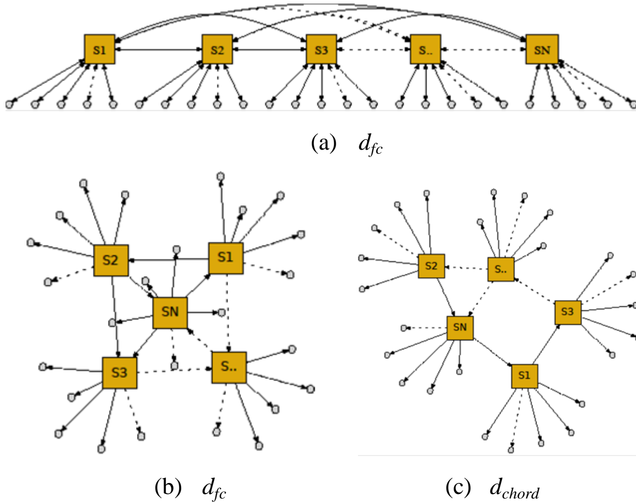


(a) d_{fc}



(b) d_{chord}

Figure 4: d_{fc} and d_{chord} with different workloads



(a) d_{fc}

(b) d_{fc} (c) d_{chord}

Figure 2: Distributed service architectures

The cost parameters of KVS simulation design are shown in Figure 3. The time to resolve a query locally (t_{LR}) and the time to resolve a remote query (t_{RR}) is given by t_{LR} = CS + SR + LP + SS + CR. For d_{fc}: $t_{RR} = t_{LR}$ +2 (SS + SR); for d_{chord}: $t_{RR} = t_{LR} + 2k$ (SS + SR), where k is the number of hops to find the predecessor.

4. Performance Evaluation

We evaluate our simulation with various workloads from real HPC services, such as monitoring, job launch, and I/O forwarding. Each client submits 10 requests, the number of replicas is 3, the failure/recover rate is 5 events/min, and we explore both strong consistency (sc) and eventual consistency (ec) models. The results for both d_{fc} and d_{chord} are shown in Figure 4. We see that for job launch and I/O forwarding workloads, eventual consistency performs worse than strong consistency. This is because these two workloads have almost uniform random distribution for both request type and the key. For the monitoring workload, eventual consistency outperforms strong consistency because all the requests are Put type. Another fact is that the efficiency of the monitoring workload is the lowest because the key space is not uniformly generated, which leads to poor load balancing.

5. CONCLUSION AND FUTURE WORK

The conclusions we draw are as follows: when the client requests dominate the communication, d_{fc} actually scales very well under moderate failures (MTTF) with different replication and consistency models, while d_{chord} scales moderately with less expensive overhead under failure events. Strong consistency is more suitable for running read-intensive applications, while eventual consistency is preferable for applications that require high availability and fast response times. Future work includes extending the simulator to cover more of the taxonomy. Additionally, we will use the simulator to model other system services and validate these at small scale, and then simulate at much larger scales.

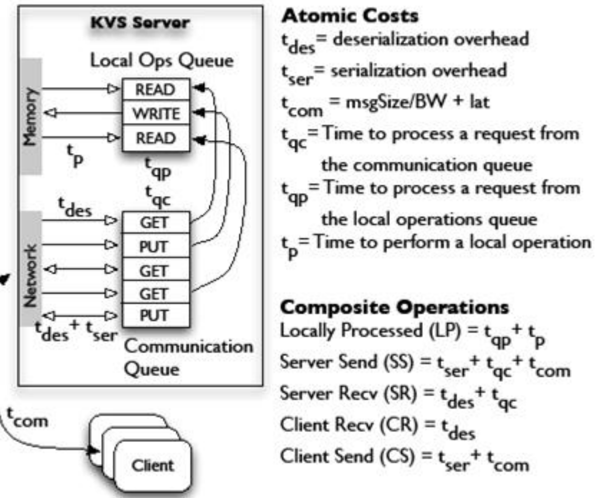


Figure 3: Cost parameters of KVS simulation design

6. REFERENCES

[1] K. Wang et. al. Simmatrix:Simulator for many-task computing execution fabric at exascale. ACM HPC 2013.

[2] I. Raicu et. al. Many-Task Computing for Grids and Supercomputers. 1st IEEE MTAGS workshop 2008.

[3] K. Wang et. al. MATRIX: MAny-Task computing execution fabRIc at eXascale. 2013. Available from http: //datasys.cs.iit.edu/projects/MATRIX/index.html.

[4] D. Zhao and Ioan Raicu. Distributedfile systems for exascale computing. In Doctoral Showcase, SC'12, November 2012.

[5] I. Stoica et. al. Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput, August 2001.