Using Simulation to Explore Distributed Key-Value Stores for Exascale System Services (SC13 Submission)

Ke Wang
Illinois Institute of Technology
kwang22@hawk.iit.edu

Abhishek Kulkarni
Indiana University
adkulkar@cs.indiana.edu

Xiaobing Zhou
Illinois Institute of Technology
xzhou40@hawk.iit.edu

Michael Lang
Los Alamos National Laboratory
mlang@lanl.gov

Ioan Raicu
Illinois Institute of Technology
iraicu@cs.iit.edu

Abstract: Owing to the extreme high rate of component failures at exascale, systemservices will need to be failure-resistant, adaptive and self-healing. A majority of HPC services are still designed around a centralized server paradigm and hence are susceptible to scaling issues and single-point-of- failure. Peer-to-peer services have proved themselves at scale for wide- area internet workloads. Distributed key-value stores (KVS) are widely used as a common building block for these services, but they are not prevalent in system services deployed on high performance computing systems. In this paper, we simulate a key-value store for various service architectures and examine the design trade-offs as applied to multiple HPC service workloads in support of an exascale class system. The simulator is validated against existing distributed KVS-based services. Via simulation, we demonstrate how the communication intensity of failure, replication, and consistency models affect performance at scale. Finally, real workloads obtained from three representative HPC services are fed into the simulator to emphasize the general use of KVS.

KVS and HPC services

Services: system booting, system monitoring, hardware or software configuration and management, I/O forwarding and run-time systems for programming models and communication libraries

KVS use cases: For resource management, KVS can be used to maintain necessary job and node status information. For monitoring, KVS can be used to maintain system activity logs. For I/O forwarding in distributed file systems, KVS can be used to maintain file metadata, including access authority and modification sequences. In job start-up, KVS can be used to disseminate configuration and initialization data amongst composite tool or application processes, this is under development for MRNet. Application developers from Sandia National Laboratory are targeting KVS to support local check-point restart . Additionally, we have used KVS to implement several real system, such as a many task computing execution fabric – MATRIXwhere KVS is used for task submission, dependency, and progress information; and the fusion distributed file system, FusionFS, where the KVS is used in tracking file-system metadata.

Simulation

1. Millions of clients , thousands of shared servers
2. each client does a stream of PUTs and GETs.

At simulation start, we model unsynchronized clients by having each

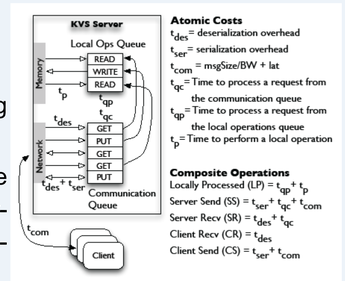
3. Servers are modeled by two queues: a communication queue and a processing queue The two queues are processed concurrently, however the requests within one queue are processed sequentially.
4. Data and Network Models: csing, ctree, dfc, dchod
5. Recovery Model: how a node recovers its state and how it

rejoins the system after a failure. The first replica of a failed server is notified by the external mechanism (EM) Then, it sends all the replicated data (including the data of the recovering server, and other servers.

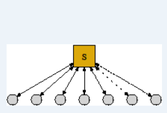
6. Consistency Model

Strong Consistency: Ensure only the primary server handles the “put” requests

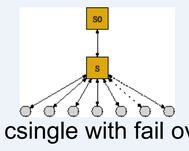
Eventual Consistency: Using Version clock, ensuring that W+R > N, in which R is the number of replicas that must participate in a successful “get” request, W is the number of replicas that must participate in a successful “put” request, and N is the number of replicas.



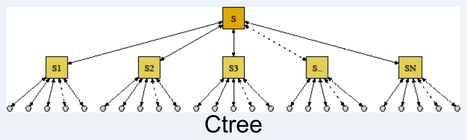
Architecture



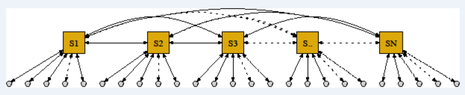
csingle



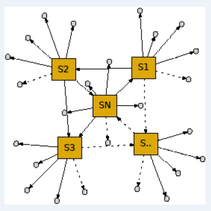
csingle with fail over



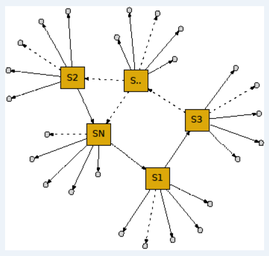
Ctree



dfc



dchord



drandom

KVS taxonomy

Components

Data Model: defines how a service distributes and manages its data.

Network Model: dictates the interconnection topology of a service’s components.

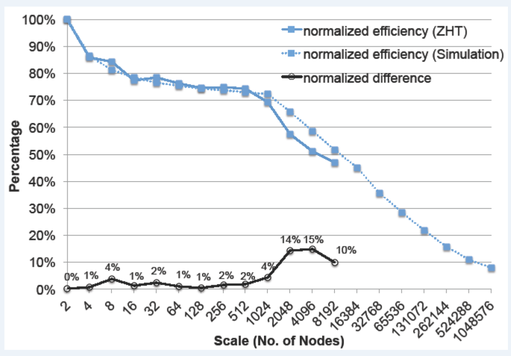
Recovery Model: specifies how a service deals with component failures.

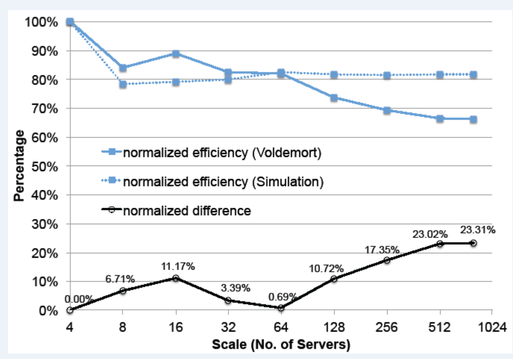
Consistency Model: pertains to how rapidly data modifications propagate across the components of a system or, in other words, how coherent and consistent different views of the same data objects are.

KVS examples

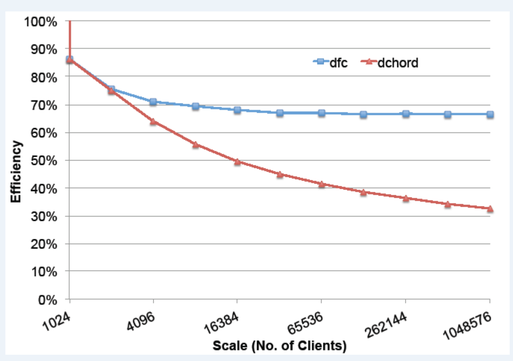
Service	Description	Data Model	Network Model	Failure Model	Consistency
Voldemort	KVS	Distributed	DFC	N-way Replication	Eventual
Cassandra	KVS	Distributed	DFC	N-way Replication	Both
D1HT	KVS	Distributed	DFC	N-way Replication	String
Pastry	KVS	Distributed	DCHORD	N-way Replication	String
ZHT	KVS	Distributed	DFC	Replicas	weak

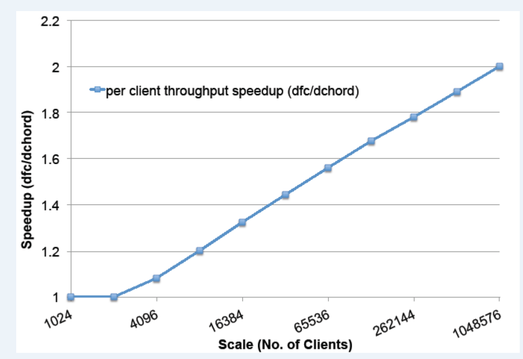
Validation



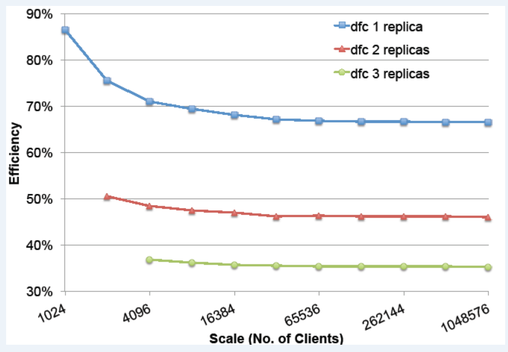


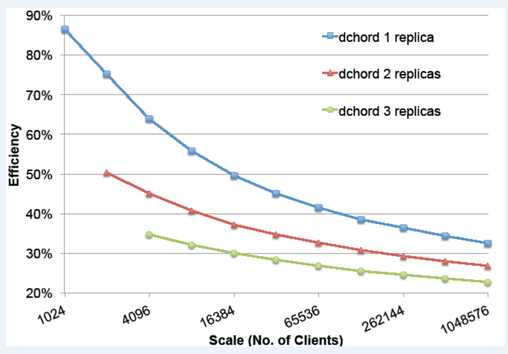
dfc vs dchord (basic case)



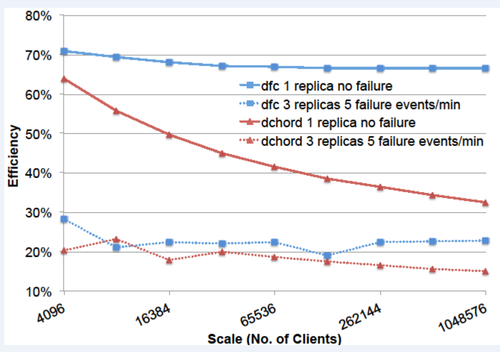


Replication Overhead

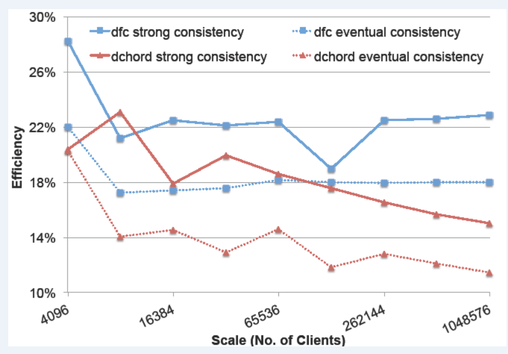




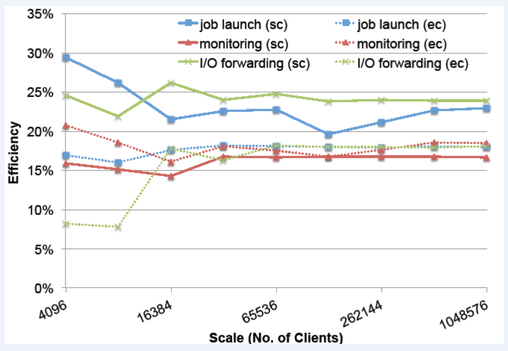
Replication and Failure Overhead

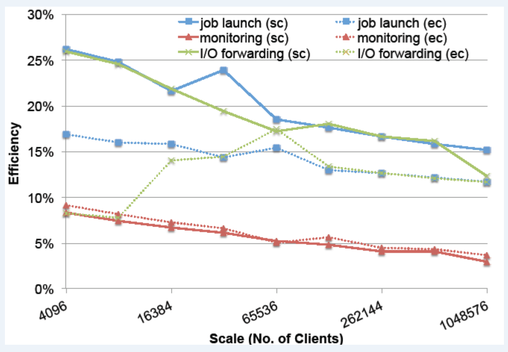


Consistency Overhead



Real Application Workload





Conclusion

when the client requests dominate the communication (up to billions at exascale), dfc actually scales very well under moderate MTTF, with different replication and consistency models, though it is relatively expensive to do a broadcast to update everyone’s membership list when a failure happens; while dchord scales moderately with less expensive overhead under failure events. When the communication is dominated by server messages, (due to fail/recover, replication or consistency) rather than client request messages, then dchord would have an advantage. Different consistency models (strong and eventual) have different application domains, 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

extending the simulator to cover more of the taxonomy, adding network models, and recovery models such as log based replay. Additionally using the simulator to model other system services and validate these at small scale then simulate at much larger scales. This work is guiding the development of a building block library that can be then used to compose distributed resilient system services for large-scale systems. We are currently developing a distributed job launch service using SLURMand ZHT. Other service building block implementations will be developed to support csingle, ctree, and dchordwith various properties from the taxonomy.

References:

[1] I. Stoica, R. Morris, D. Karger,M. Frans Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput. Commun. Rev., 31(4):149–160, August 2001. ISSN: 0146-4833.

[2] I. Diane, I. Niang, and B. Gueye. A Hierarchical DHT for Fault Tolerant Management in P2P-SIP Networks. In Proceedings of the 2010 IEEE 16th , ICPADS ’10, pages 788–793,Washington, DC, USA, 2010.

[3] I. Raicu, I. T. Foster, P. Beckman. Making a case for distributed file systems at exascale. In Proceedings of the 2011 workshop Large-scale system and application performance, LSAP’11 pages 11–18, San Jose, California, USA 2011

[4] K.Wang, A. Rajendranl, and I. Raicu. "MATRIX: Many-task computing execution fabric at exascale". 2013. Available from http: //datasys.cs.iit.edu/projects/MATRIX/index.html