MHT: A Light-weight Scalable Zero-hop MPI Enabled Distributed Key-Value Store

Xiaobing Zhou[1], Tonglin Li[2], Ke Wang[4], Dongfang Zhao[2], Iman Sadooghi[2], Ioan Raicu[2,3]
[1]*Hortonworks,* [2]*Illinois Institute of Technology,* [3]*Argonne National Laboratory,* [4]*Intel*

***Abstract*—In this paper, we propose and implement a key-value store that supports MPI while allowing application access at any time without having to declaring in the same MPI communication world. This feature may significantly simplify the application design and allow programmers leverage the power of key-value store in an intuitive way. In our preliminary experiment results captured from a supercomputer at Los Alamos National Laboratory, our prototype shows linear scalability at up to 256 nodes.**

I. INTRODUCTION

Today's science increasingly relies on data driven paradigm and large scale simulations. From bioinformatics [1], seabed modeling [2] to micro electronic systems [3–5]; from cyberspace security risk assessment [6] to chemical catalyst simulation [7], scientific applications are calling for more computing power in larger scale systems. NoSQL databases, such as key-value stores, are known for their ease of use and excellent scalability and versatility [8–11] in clouds [12]. However supercomputers and HPC applications are not able to enjoy the benefits of distributed key-value stores due to their customized OS and communication stack. In most of supercomputers, MPI is the default if not the only communication protocol supported while most of key-value stores only use TCP/UDP. Further more, the MPI communication world is fixed and won't allow new processes to join during application run-time. This means that the number of both clients (applications) and servers of key-value store have to be clearly defined and fixed at the very beginning and thus not allow application to access the key-value store in a dynamic and flexible manner.

In this paper, we propose and implement a key-value store that supports MPI while allowing application access at any time without having to declaring in the same MPI communication world. This feature may significantly simplify the application design and allow programmers leverage the power of key-value store in an intuitive way. In our preliminary experiment results captured from a supercomputer at Los Alamos National Laboratory, our prototype shows linear scalability at up to 256 nodes.

The contributions of this paper are:

- Design and implementation of a prototype of MHT, a distributed key-value storage system that supports MPI, separates its own failure domain from that of applications, and allows application clients dynamically join the communication group and access data servers;
- Evaluation of MHT on a supercomputer at up to 256 nodes shows excellent performance and comparable scalability against the implementation on TCP.

II. DESIGN AND IMPLEMENTATION

A. Challenges and Design Considerations

In order to implement MPI process dynamic join and separated failure domain, it necessitates that MHT applications are built and launched independent of MHT server daemons running as MPI processes. Essentially, they belong to two independent MPI communicators, thus survive through failure of one or the other. However, typically, MPI application binary is launched and propagated to many MPI processes all at one time. In MHT case, many MHT server daemons are bootstrapped as MPI processes firstly and wait for processing incoming requests. Then it comes to dilemma that MHT applications fail to interact with the running MHT servers because they belong to two independent MPI communicators. There are a couple of options investigated in order to make MHT applications able to talk to running MHT servers.

1) MPI_Comm_spawn: MPI_Comm_spawn is a MPI 2 facility that is called to spawn new children on a brand new MPI communicator. New ranks within new communicator have dedicated MPI_COMM_WORLD different from that of parent running ranks. Although It is possible to create a new communicator that contains all parent running ranks, it is obligatory to have parent and children MPI processes launched together. In other words, by the means of MPI_Comm_spawn, MHT servers and MHT applications need to be single MPI launch, as a result, they belong to the same failure domain that makes whole system less resilient.

2) MPI_Comm_join: MPI_Comm_join is another MPI 2 routine called to connect two MPI processes by established socket. But there are a couple of constraints that make it less suitable for MHT use case. Firstly, it only works between two MPI processes that are connected by a socket, however, MHT and MHT applications are composed of many MPI processes which are 1 to N or N to N communication. Secondly, it requires quiescent socket in which case a read will not read any data that was written to the socket before the remote process returned from MPI_Comm_join. Quiescent socket is hard to be implemented. Finally, MPI_Comm_join is error prone if two endpoints of socket are based on different implementation-defined MPI communication universe.

3) MPI_Comm_connect and MPI_Comm_accept: In the MPI 2.0 and above versions, MP_Comm_accept and MPI_Comm_connect can be used to build client/server style MPI system, which yields separated failure domains, however, dynamic process management features are not universally available in many supercomputers, for example, Blue Gene/P and above, Cray, and so on.

Finally, we choose to implement portable mechanism that gains separated MPI failure domain: MHT and MHT application. With this work, it has been practical to build robust MHT applications on top of MHT MPI infrastructure and make them survive through failure of one or the other, although either MHT application or MHT MPI is not resilient to failure, which is due to MPI standard that does not address fault tolerance.

B. Archiecture and Design

This work proposed a broker architecture, see also fig.1. In essence, MHT is a variant of ZHT that is customized to work over MPI. It inherits ring topology of ZHT. All MHT servers are bootstrapped as MPI processes and assigned to proper ranks. MHT broker is also running as MPI process being allocated dedicated rank, and launched alone with many MPI processes of MHT servers by the same mpiexec in order to share the same communicator. MHT application usually runs as standalone general process or MPI process, in both cases, MHT application calls MHT client that provides key/value API to send requests to MHT broker through IPC (inter-process communication) facilities as part of Linux/Unix kernel, such as message queue. MHT broker forwards requests to pluggable MHT Routing module by library call. MHT Routing then sends requests over MPI protocol to destination MHT server. Default MHT Routing algorithm is consistent hash. Weighted hash like CRUSH is alternate routing algorithm to accommodate node heterogeneity, minimize unnecessary data movement between MHT servers, and distributes data to proper MHT servers to enforce separation of replicas across failure domains.

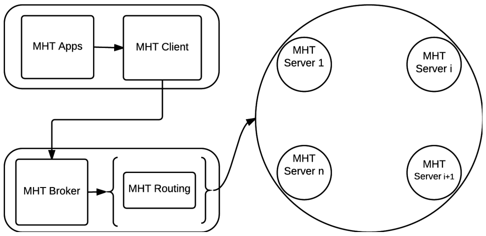


Fig. 1: MHT architecture.

From process perspective, MHT application and MHT client are in one regular process or MPI process. MHT broker and MHT Routing belongs to another MPI process. MHT Server runs within its own MPI process. Only the process for MHT application and MHT client could be either regular process or MPI process, otherwise, the others must be MPI process since MHT Broker and MHT Server need to be bootstrapped as MPI processes by a single `mpiexec` in order to share the same MPI_COMM_WORLD.

C. Deployment Contract

In terms of deployment, MHT application and MHT client, along with MHT broker and MHT Routing are deployed to one node. MHT Server could be in another node. Specially, every node needs deploying one MHT Broker which is the only one MPI entry point in that node to a network of MHT Servers. To ensure MHT Broker and MHT Servers are allocated to proper MPI ranks that can be used by MHT Routing module to determine correct destination MHT Server for any specific request, the order of MPI Broker and MHT Servers assumed by mpiexec really matters. Here are good cases in point.

1) Deploy MHT in Pseudo Distributed Mode: The command below will launch four MTH Servers (zht-mpiserver as their binary) as four MPI processes, and one MHT Broker (zht-mpibroker as its binary) as one MPI process in a single node. The zht-mpiserver must precede zht-mpibroker because that is the only way that the binary zht-mpiserver will be propagated to four MPI processes with MPI rank range of 0 to 3. The binary zht-mpibroker will be loaded into a dedicated MPI process with rank 4. MPI Routing module reads MHT Server nodes membership from neighbor.mpi.conf which contains MHT Server addresses. The number of nodes in neighbor.mpi.conf corresponds to initial MPI rank range associated with MHT Servers, i.e., there must be 4 MHT Server nodes with addresses configured in neighbor.mpi.conf, mapping to MPI rank 0 to 3. While determining destination MHT Server, MPI Routing module only considers rank 0 to 3 by simply ignoring rank of MHT Broker, i.e,. MPI rank 4, since MHT Broker is not qualified as part of MHT Server membership.

```
    mpiexec -np 4 ./zht-mpiserver -z zht.conf -n
neighbor.mpi.conf : ./zht-mpibroker -z zht.conf -n
neighbor.mpi.conf
```

2) Deploy MHT in Cluster Mode: For simplicity, the following command will start *n_proc* MHT Servers in the format of *n_proc* MPI processes on the nodes configured in neighbor.mpi, on each of which one MHT Broker is also launched as one MPI process. The MPI rank of 0 to *n_proc*-1 are the membership consulted by MHT Routing module.

```
    mpiexec -f neighbor.mpi.conf -np n_proc ./zht-mpiserver
-z zht.conf -n neighbor.mpi.conf : ./zht-mpibroker -z
zht.conf -n neighbor.mpi.conf
```

D. Implementation

In order to support multiple communication protocols, it is necessary to design protocol abstraction. MHT adopts proxy and stub structure. Basically, proxy is a set of classes called by and hosted in client process, and stub is one that is hosted within server process. See fig.2 for the proxy stub class hierarchy.

ProtoProxy is designed to provide send, receive, and sendrecv functions. ProtoStub offers receive, send and recvsend ones, simply put, sendrecv and recvsend are the combined functions of send/receive. New communication protocols are easily to be implemented into this abstraction by simply extending the corresponding ProtoProxy and ProtoStub. For example, TCPProxy and TCPStub are used for support TCP as communication mechanism.

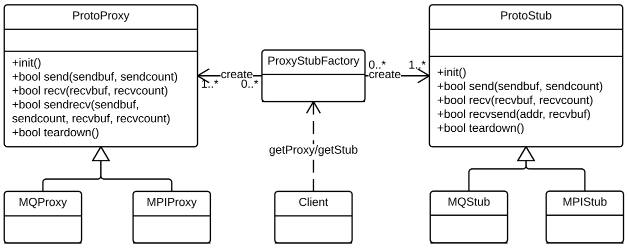


Fig. 2: Structure of protocol abstraction.

E. Runtime Sequence

Referring to fig.3, to put/get/delete/append data, MHT applications do library call to MHT client key/value API, which in process invokes MQProxy::sendrecv to send requests to through IPC (inter-process communication) and waits for responses from MHT Broker. Within MPI process of MHT Broker, MQStub::recvsend is long running to serve requests from MQProxy over IPC. After getting address of correct destination MHT Server by consulting MHT Routing module, MQStub simply does library call to MPIProxy::sendrecv that sends requests to through MPI protocol and waits for responses from MHT Server. Within MPI process of MHT Server, MPIStub::recvsend is long running to serve incoming requests from MPIProxy over MPI protocol, and then returns responses along with backward communication link.

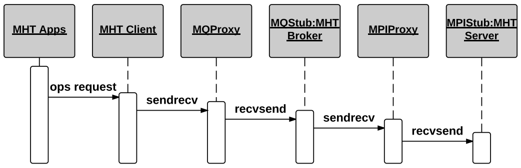


Fig. 3: MTH request/response sequence.

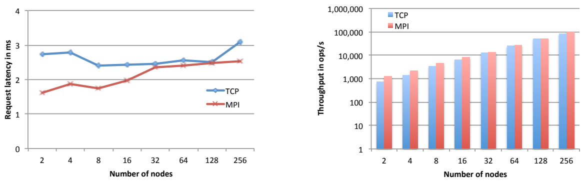
III. EXPERIMENTAL RESULTS

A. Experiment setup

We conduct the evaluation on Kodiak supercomputer, a Parallel Reconfigurable Observational Environment (PROBE)[13] at Los Alamos National Laboratory, it has 1024 nodes, and each node has two 64-bit AMD Opteron processors at 2.6GHz and 8GB memory. In all experiments, requests are sent from clients in tight loops. Like Facebook [14] and MICA's [15] workloads, we focus on small requests with fixed key (10 bytes) and value length (20 bytes), 95% `get` and 5% `put`.

B. Results

In fig.4(a) we can see that at up to 256 nodes, MHT has a bit lower latency than the ZHT with TCP, especially on smaller scales. Similarly on throughput, MHT also shows slightly better performance than ZHT with TCP. It's worth to note that Kodiak's MPI is running over TCP and goes through all TCP's network protocol stacks. On some larger supercomputers such as IBM BlueGene series, MPI is implemented on hardware level. We would expect even better performance from MHT on those platforms.



(a) Latency (b) Aggregated throughput

Fig. 4: MHT performance with TCP v.s MPI

IV. RELATED WORK

Content-MPI (C-MPI)[16], is the only key-value store project that we are aware of supporting MPI. It is built on MPI functionality, and offers a scalable data store that is fault tolerant. C-MPI does not support dynamic MPI process join. ZHT [17–21], the parent project of this work, is a zero-hop distributed hash table, and has been tuned for the requirements of high-end computing systems. ZHT has been used in multiple distributed systems, namely file system [22], job scheduler [23–25], distributed message queue [26], graph processing system [27] and many others. But the published version of ZHT doesn't support MPI.

V. CONCLUSIONS

In this paper, we present a prototype of a distributed key-value store that supports MPI while allowing application access at any time without having to declaring in the same MPI communication world. This feature may significantly simplify the application design and allow programmers leverage the power of distributed key-value store in an intuitive way. The preliminary results shows close-to-linear scalability at up to 256 nodes.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation grant NSF-1054974. This work used Kodiak supercomputer, a Parallel Reconfigurable Observational Environment (PROBE) [13] deployed at Los Alamos National Laboratory (LANL).

REFERENCES

[1] Yi Wang, Gagan Agrawal, Gulcin Ozer, and Kun Huang. Removing sequential bottlenecks in analysis of next-generation sequencing data. In *IPDPSW, 2014 IEEE International*. IEEE, 2014.

[2] Tianyun Su, Zhihan Lv, Shan Gao, Xiaolong Li, and Haibin Lv. 3d seabed: 3d modeling and visualization platform for the seabed. In *Multimedia and Expo Workshops (ICMEW), 2014 IEEE International Conference*.

[3] Fan Shi, Xiang Zhang, Qian Li, and Changyu Shen. Notice of retraction particle tracking in micro-injection molding simulated by mis. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*. IEEE, 2010.

[4] Fan Shi, Xiang Zhang, Qian Li, and Changyu Shen. Mould wall friction effects on micro injection moulding based on simulation of mis. *IOP Conference Series: Materials Science and Engineering*, 2010.

[5] SHI Fan, Zhang Xiang, LI Qian, and Shen Changyu. Numerical simulation of micro injection moulding based on mesh free method. *Sciencepaper Online*, 2010.

[6] Su Zhang, Xinwen Zhang, and Xinming Ou. After we knew it: empirical study and modeling of cost-effectiveness of exploiting prevalent known vulnerabilities across iaas cloud. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 317–328. ACM, 2014.

[7] Fan Shi, Aaron M Coffey, Kevin W Waddell, Eduard Y Chekmenev, and Boyd M Goodson. Nanoscale catalysts for nmr signal enhancement by reversible exchange. *The Journal of Physical Chemistry C*, 119(13):7525–7533, 2015.

[8] Tonglin Li, Ioan Raicu, and Lavanya Ramakrishnan. Scalable state management for scientific applications in the cloud. BigData Congress '14.

[9] Tonglin Li, Kate Keahey, Rajesh Sankaran, Pete Beckman, and Ioan Raicu. A cloud-based interactive data infrastructure for sensor networks. IEEE/ACM Supercomputing/SC'14.

[10] Tonglin Li, Kate Keahey, Ke Wang, Dongfang Zhao, and Ioan Raicu. A dynamically scalable cloud data infrastructure for sensor networks. ACM ScienceCloud 15.

[11] Tonglin Li, Ke Wang, Dongfang Zhao, Kan Qiao, Iman Sadooghi, Xiaobing Zhou, and Ioan Raicu. A flexible qos fortified distributed key-value storage system for the cloud. In *IEEE BigData Conference2015*. IEEE, 2015.

[12] I. Sadooghi, J. Hernandez Martin, T. Li, K. Brandstatter, Y. Zhao, K. Maheshwari, T. Pais Pitta de Lacerda Ruivo, S. Timm, G. Garzoglio, and I. Raicu. Understanding the performance and potential of cloud computing for scientific applications. 2015.

[13] Parallel reconfigurable observational environment. http://www.nmc-probe.org/. Accessed: 2014-11-30.

[14] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. SIGMETRICS '12, 2012.

[15] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: a holistic approach to fast in-memory key-value storage. NSDI'14.

[16] J.M. Wozniak, B. Jacobs, R. Latham, S.W. Son S. Lang, and R. Ross. C-mpi: A DHT implementation for grid and HPC environments. 2010.

[17] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, and Ioan Raicu. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. IPDPS '13.

[18] Tonglin Li, Raman Verma, Xi Duan, Hui Jin, and Ioan Raicu. Exploring distributed hash tables in highend computing. *SIGMETRICS Performance Evaluation Review*, 2011.

[19] Tonglin Li, Xiaobing Zhou, Ke Wang, Dongfang Zhao, Iman Sadooghi, Zhao Zhang, and Ioan Raicu. A convergence of key-value storage systems from clouds to supercomputers. *Concurr. Comput. : Pract. Exper.(CCPE)*, 2015.

[20] Tonglin Li and Ioan Raicu. Distributed nosql storage for extreme-scale system services. In *IEEE/ACM Supercomputing PhD Showcase*. IEEE/ACM, 2015.

[21] Tonglin Li. A convergence of NoSQL storage systems from clouds to supercomputers. *Illinois Institute of Technology, PhD Proposal*, 2014.

[22] Dongfang Zhao, Zhao Zhang, Xiaobing Zhou, Tonglin Li, Ke Wang, Dries Kimpe, Philip Carns, Robert Ross, and Ioan Raicu. Fusionfs: Towards supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In *Big Data, 2014 IEEE International Conference on*.

[23] Ke Wang, Xiaobing Zhou, Tonglin Li, Michael Lang, and Ioan Raicu. Optimizing load balancing and data-locality with data-aware scheduling. IEEE BigData'14.

[24] Ke Wang, Ning Liu, Iman Sadooghi, Xi Yang, Xiaobing Zhou, Tonglin Li, Michael Lang, Xian-He Sun, and Ioan Raicu. Overcoming Hadoop scaling limitations through distributed task execution. IEEE Cluster'15, 2015.

[25] Ke Wang, Kan Qiao, Iman Sadooghi, Xiaobing Zhou, Tonglin Li, Michael Lang, and Ioan Raicu. Load-balanced and locality-aware scheduling for data-intensive workloads at extreme scales. *Concurrency and Computation: Practice and Experience*, 2015.

[26] Iman Sadooghi, Ke Wang, Shiva Srivastava, Dharmit Patel, Dongfang Zhao, Tonglin Li, and Ioan Raicu. Fabriq: Leveraging distributed hash tables towards distributed publish-subscribe message queues. IEEE/ACM International Symposium on Big Data Computing (BDC), 2015.

[27] Tonglin Li, Chaoqi Ma, Jiabao Li, Xiaobing Zhou, Ke Wang, Dongfang Zhao, Iman Sadooghi, and Ioan Raicu. GRAPH/Z: A key-value store based scalable graph processing system. Cluster'15.