Hop: Elastic consistency for exascale data stores*

Latchesar Ionkov and Michael Lang

Los Alamos National Laboratory, Los Alamos NM 87544, USA

Abstract. Distributed key-value stores are a scalable alternative for relational databases and file systems. Different systems try to provide the right balance between scalability, consistency and reliability for a certain category of applications. Yet many applications don't have a single static set of requirements for all the data they create and use. In order to meet different requirements, the applications end up using multiple data stores, each with its own interface, or suffer the unnecessary performance degradation. This paper describes Hop, our attempt to define an interface that allows specification of different consistency and replication levels, and by allowing the assembly of different implementations of the interface, provides the right balance of consistency, scalability and availability for different parts of application data. We describe the Hop interface and evaluate some of the implementations we created, as well as how they can be combined to provide flexible distributed deployments.

1 Introduction

In recent years many applications try to replace the standard data stores, like relational databases and file systems with distributed key-value stores. The main reason for the change is that the new data stores provide different balance between scalability, reliability and consistency, usually focusing more on the scalability, than reliability and consistency. There is a large variety of storage systems available, each with different functionality, optimizations and guarantees. Most of these systems are designed for particular application or workload, and there is rarely a single system, that fits perfectly for other applications, with slightly different requirements.

Our experience with implementing system services and scientific simulation applications shows that even within a single application, the consistency and performance requirements vary widely across subsets of the data sets, or for the same subsets during different stages of the application execution. For example, a scientific simulation doesn't require strong data consistency within the distributed store while performing a checkpoint (dumping its state to stable storage), performance is more important. Analytic and visualization applications, on other hand, require the same data set to be in consistent state before they can process it. Depending on the way the application partitions the overall work, consistency can be relaxed based on regions of the data sets that are shared by groups of nodes.

* LA-UR-13-27636

Same diverse requirements apply for data replication. In most cases, it is beneficial for the overall performance if the replicas don't share the same physical location (rack, data center, etc.) which usually leads to higher latency and lower performance. In some cases, for instance when the data is temporary and all users are clustered physically, it makes more sense if the replicas are close. For example, there is no reason to keep the list of which nodes within a rack are powered-on replicated outside the rack itself. Power or network failure to the whole rack will make the data unavailable, but as the resources the list describes are unavailable, that loss is irrelevant.

Distributed data stores are currently being evaluated for many exascale tasks where centralized services have reached the limits of scalability. Global services such as namespaces, service registration, provisioning, monitoring, job and resource management, I/O forwarding, fabric management; runtime systems supporting newer distributed programming models all need distributed data services at exascale. In the application space these stores are being evaluated for static and dynamic data tables, such as Equation of State (EOS) for multi physics, and as a general way of dealing with asynchronous access to distributed data structures. Hop is a robust, scalable class of solutions designed for these exascale requirements.

The main goal of the Hop project is to try to define an interface for data access that can be easily used by application and middle-ware developers and implemented by data stores. It should allow data store developers to implement and express various levels of consistency and reliability. The application developers should be able to express the consistency and data replication requirements for particular parts of the data sets and phases of the application. Middle-ware developers should be able to write software that optimizes for particular workloads without sacrificing the requirements defined by the application.

The Hop interface allows users of a data store to define the level of consistency they need while reading the data. The choice varies from "don't care" to "not less recent than" to strong consistency. It allows formation of *consistency domains* where a set of application instances ensure mutual consistency, without paying the performance degradation for strong consistency outside the domain.

Each data entry in Hop is assigned numeric *version* value, which can be used to define the *freshness* of the entry's value. The entry's version is monotonically increasing each time when its value is updated. When retrieving data, the application can specify version value, and will be guaranteed to receive data that is more recent that the specified version. All update operations return the new version value. If an instance that updated some data needs to ensure that another instance will see its updates, it needs to pass its version value to the other instance, which in turn uses that value when reading the entry. There is a special value, `Newest` that forces strong consistency among all instances. Clients can also use future version values as a mechanism to wait until the entry is modified. This feature is useful in cases when it may take some time until updates are propagated across multi-layered systems as well as for implementing publish/subscribe features on top of Hop.

Hop also defines a set of atomic operations on a single data entry, improving the usability of a distributed store without sacrificing the performance. The atomic operations are serialized at the node, responsible for the entry, thus ensuring their correct behavior within the distributed store.

Unlike the commonly used data stores, where clients use a single monolithic service, we envision that Hop will be used similarly to file systems, where each application will use multiple Hop implementations, each providing different functionality, and only some of them used for persistently storing the data to disks. Custom Hop implementations may include caching middle-ware suitable for particular applications, aggregation of data entries, etc.

As part of the project, we developed a number of simple Hop implementations that provide useful functionality and can be used to assemble more complicated configurations that can be used for real-world installations. Our caching service provides support for local and cooperative caching. It also implements a novel concept of consistency domains that allow clients to easily group together and ensure consistency for a subset of the key-value store entries.

The project is novel in that it provides a minimal, easy to implement API, as well as a set of basic building blocks from which to compose larger services based on key-value stores. It allows definition of flexible coherency domains for subsets of the data. Other key contributions are : configurable consistency and replication of individual elements, definition of entries that provide side-effects upon access, support for publish/subscribe over key/value store interface, and a unique aggregation service that redirects based on a simple prefix key.

2 Related work

Distributed key-value store is an important system service, which serves requests very fast. Memcached [5] is a popular service that stores key/values pairs in RAM and is designed to serve as a caching layer between applications and persistent data stores like databases. Dynamo [2] is a highly available key-value storage implemented by Amazon to provide users with a reliable resource. Cassandra [9] is a distributed storage system developed by Facebook to satisfy the requirements of the Inbox Search problem. It provides SQL-like query language and failure recovery. ZHT [11] is a zero-hop distributed hash table for managing the parallel file system meta-data, and serves as a building block for future distributed system services. Chord [14] is a distributed protocol, which defines how servers communicate with other and how data is distributed among all the servers. As part of the Hop evaluation, we implemented both Chord-like and ZHT-like schemes providing choice to the services so they can select a method that best fits its requirements.

There are systems that use some form of versioning to manage consistency. Project Voldemort [4] is an advanced key-value store that provides multi-version concurrency control for updates. In order to get up-to-date view, the application needs to read data from the majority of the replicas. Similarly to Voldemort, Riak is a data store that uses MVCC and vector clocks to order updates. It also provides tunable consistency at read and write operations by specifying how many replicas must respond to operation.

Comet [6] is a distributed data storage system that allows executable code (handlers) to be attached to the stored entries, making them “active”. The handlers may run on specific operations (get, set, etc.), or on timers, at the place where the entry is stored.

COPS [12] tries to improve the eventual consistency, usually used in distributed systems, by presenting a scalable causal consistency with convergent conflict handling.

OceanStore [8] is an infrastructure that can be used to connect data stores across wide-area network and provide provide clients with flexible consistency guarantees. It uses replicas as caching capabilities to provide fail-over and to improve performance. It uses entries’ versions to to keep track of the freshness of the data in the replicas and caches. OceanStore also defines algorithms that are used for data placement. The consistency level is defined per session and if client has different consistency requirements for groups of data, it needs to use multiple sessions.

3 Design

The main objective of the project is to define an interface for storing and retrieving data values in distributed systems, while allowing the data users to specify diverse consistency and replication levels for the different types of data they store.

The data store and retrieval requirements for the applications are not uniform, and in many cases they vary even for different data entries within an application. For example, in some cases fast data retrieval is more important than the data freshness. Some of the data is relevant only to the clients in a location (i.e. server rack) and don’t need to be replicated outside that location. There are cases when consistency is required for groups of clients while the consistency restrictions across groups can be relaxed.

Generally, a balance is necessary between the number of operations defined in an API and the complexity of the parameters they accept. Small set of operations makes it easier for clients to use it. It also makes the implementation of simple services that support it more straightforward. For complex functionality, though, a small number of operations usually leads to a complex parameter space with too many values with special semantics. In Hop’s design we tried to create what we believe is the right balance between the number of operations and parameter idiosyncrasies.

In order to keep the interface simple, Hop defines a single type of keys (string) and a single type of values (byte array). Our evaluation in section 4 shows that these restrictions still allow reasonable level of flexibility.

We intentionally omitted support for partial entry retrieval and update. We believe that large monolithic entries allow applications to encapsulate their data in “file” formats that leave off important metadata information and make it harder for data to be shared across applications. Instead of using a singly entry with all data in it, we encourage developers to adopt more fine-grained approach to data storage. The atomic operation `Replace` provides support for partial data update, which we hope is not going to be abused by the developers.

3.1 Hop Operations

The Hop interface defines 6 operations:

Create	Creates a new entry for a key and sets its initial value.
Remove	Removes an entry associated with a key.
Get	Retrieves the value, associated to a key.
Set	Updates the value, associated with a key.
TestSet	Compares the value of an entry and sets it to a new value if the comparison is true.
Atomic	Performs an atomic operation on an entry.

All Hop operations receive the name (key) of the entry as a parameter. Other approaches, for example assigning a temporary numeric value to an entry (like the file descriptors in the POSIX API) have some advantages, like performing the name look-up and permission checking once at the beginning of a set of operations, smaller message overhead, etc. That approach implies state kept on the servers for each client. In a distributed system, these benefits are smaller than the disadvantages of keeping (and replicating in case of a failure) the state. Many of the key-value stores implicitly create entries, when a value for a key is `set` first. We chose to define an explicit create operation. The main reason is to allow the users to list specific requirements for the entry: how many replicas should be supported, the location of the replicas, etc. Additionally, we prefer to preserve symmetry: an interface with a function that deletes an entry is more complete if it also explicitly creates it.

The interface doesn't define any specific values for the entry's flags, the description what parameters are supported is left for the specific implementations.

In distributed systems, many problems arise when multiple clients concurrently modify an entry's value. Even strong consistency guarantees don't alleviate all the complications. For example, there is no guarantee that incrementing a value is going to produce the correct results. The most common solutions for the problems are support for transactions, or atomic operations. We chose to provide the latter, mostly because distributed transactions are complex to implement, don't scale well and complicate the key/value store interface. Transactions offer many benefits, and we plan to explore them in the future based on the atomic primitives we already defined.

All atomic operations serialize the access to an entry. The value of an entry can't be modified or retrieved while an atomic operation is being processed. The most commonly implemented atomic operation is test-and-set. It compares the value of an entry to a specified value, and if both match, sets it to a new value. Test-and-set returns the value of the entry, as set at the end of the operation. If the two values don't match, the user receives the current entry's value, otherwise the user receives the new value.

In a distributed configuration, where clients and servers might be located on different nodes with high latency network connecting them, test-and-set might be an inefficient method to atomically modify an entry. While a client retrieves the value, modifies it and performs test-and-set, another client may change the entry, forcing the first client to loop multiple times until it succeeds. To improve the latency, we define a set of commonly used operations that are atomically executed by the Hop implementation at the location where the entry is stored. Additional atomic operations can be built on these.

Add	Atomically adds a number to the entry's value.
Sub	Atomically subtracts a number from the entry's value.
BitSet	Atomically sets to 1 one bit in the entry's value that was previously set to 0. Returns the new value as well as the address of the bit that was modified.
BitClear	Atomically sets to 0 one bit in the entry's value that was previously set to 1. Returns the new value as well as the address of the bit that was modified.
Append	Atomically appends the value specified in the operation to the entry's current value.
Remove	Atomically removes from the entry's current value all subarrays that match the value specified in the operation.
Replace	Atomically replaces all subarrays from the entry's current value that match the first value specified in the operation with the second value.

Although the Hop values are arrays of bytes, the arithmetic atomic operations assume the arrays to be big-endian encoding of 8-, 16-, 32-, or 64-bit integers (depending on the size of the array). The Hop implementations can define additional atomic operations.

3.2 Entry Versions

The Hop interface assigns a 64-bit *version* value to each data entry. The version is increased every time the value of the entry is updated. When data is retrieved, in addition to the value, the user receives its current version. The application can specify what version of the value it requires. The Hop service cannot return values associated with versions older than the specified one. They might, however return more recent values. If the value the service is not recent enough, it is expected to wait until the entry reaches the specified version. Caching services may use requests for newer versions as triggers for updating the cached entries. In distributed systems it is possible for requests to arrive in a different order to the servers, therefore versions ensure later requests wait for the earlier ones.

Valid entry version have values between 1 and 2^{63}. The rest of the values can be used to define some special version values that fine-tune the semantics of the data retrieval. The special values that all Hop services are required to support are:

Any	Retrieve any value, no matter how fresh.

Newest	Retrieve the newest value. Instructs all intermediary services to ignore their stored entries and consult the authoritative service that contains the entry.
Wait	Ignore all intermediaries, wait until the entry is modified once, and return the new value. Can be used for implementation of publish/subscribe mechanisms.
Uncommitted	Return values even if they are not yet replicated as required.

All operations that return an entry's value also return its version. Successful operations are required to return a valid version (i.e. between 1 and 2^{31}). If an operation returns 0 for a version, the client is expected to retry the operation (similarly to EAGAIN error code in Unix). In a distributed environment there might be cases when successful completion of the operation is hard, or even impossible to achieve, but returning an error would be the wrong outcome. For example, if an instance in distributed service fails and its responsibilities are assigned to another instance, it is easier to abandon the currently pending operations and ask the clients to retry using the updated configuration.

Instead of keeping only the latest version of an entry, Hop allows the implementations to keep as many older versions as they may see fit. However, Hop would have to be extended to provide any operations for version management.

3.3 Entry Names and Values

Entries' names in Hop are variable-sized UTF-8 encoded strings up to 65535 bytes long. Like most key-value stores, Hop supports a flat namespace. Because Hop doesn't support key enumeration (see section 4.1 though), there is no reason to group keys in hierarchical namespaces. Flat namespaces are also easier to partition in distributed implementations.

An entry's value is an array of bytes, with maximum size of 4GB. The Hop interface doesn't support partial retrieval of values, and there is no practical reason for bigger values if they are stored and retrieved over the network.

3.4 Entry side-effects

Hop implementations can define some, or all of the entries as special entries that have side effects or produce results based on the server or client configurations. One example of that behavior is the `#/keys:`*regular-expression* entry that will be defined in section 4.1.

4 Implementations and Evaluation

The Hop interface is intentionally kept simple. We envision multiple services implementing it assembled from hop building block services. Real-world installations using a combination of multiple implementations to provide the required levels of consistency, scalability, and reliability. Currently we have created 7 Hop implementations, the more complicated ones using the basic ones as building blocks. In this section we will describe these implementations as examples of how the basic Hop interface can be used and to assemble complex real-world services.

4.1 General Functionality and Guidelines

All of the implementations below provide extra functionality not defined in the basic Hop interface that is useful in building system services.

Local Entries Entries with names starting with `#/` are considered local and may return different values depending to which node in a distributed system the client connects. Most of these entries are used by the implementation itself to maintain its basic functionality. All our Hop implementations provide `#/id` entry. It is immutable and provides the name of the Hop implementation, as well as additional information about its configuration.

Name Lookup The basic Hop interface doesn't provide functionality that allows checking the names of existing entries. Our Hop implementations allow name lookup using two special local entries: `#/keys` and `#/keys:`*regexp*. The value of `#/keys` contains the names of all local entries, `#/keys:`*regular-expression* is a special *dynamic* entry that returns all keys that match the specified regular expression. For example, getting the value of `#keys:foo.*bar` will return all keys, with prefix `foo` and suffix `bar`.

The standard rules for Hop versions also apply for the special keys. For example, the user can watch a Hop instance for newly created entries by calling the `Get` operation on `#/keys` with version value `Wait`. The operation will hang until a new entry is created.

If the service has a lot entries, waiting on `#/keys` is impractical, because it will transfer huge amounts of data back to the client. For that reason, we defined another special entry, `#/keynum` that returns the number of keys present in a Hop instance. Waiting on it will also return when an entry is created or removed.

In our future implementations we are planning to use dynamic key names for other purposes, like returning partial values, debugging, etc.

Distributed Implementations Instead of defining additional protocols, we use the Hop interface and special local entries to implement the communication between the instances in a distributed service. That serves two main purposes: makes the coding of distributed services fast and simple, and tests if the Hop interface is flexible enough for real-world applications. We believe that the inability of using the Hop interface for our own distributed implementations would be an argument for its incompleteness and/or inefficiency.

4.2 Basic Building Blocks

Rmt The standard Hop interface doesn't define or provide any support for networked deployment of Hop implementations. The simplicity of the standard Hop interface makes it easier to implement and doesn't tie it to specific network protocol or framework.

Rmt is a package that provides networked client-server support for Hop. It consists of a tightly integrated pair: rmt.srv and rmt.clnt. Rmt.clnt is a Hop implementation that serializes all Hop operations and sends them over a network to rmt.srv. Rmt.srv receives a Hop implementation as a parameter and makes it available over a network connection. When it receives a message, it deserializes it, calls the appropriate Hop operation, serializes the result and sends it back.

Rmt supports TCP and Infiniband Verbs protocols. It allows up to 65536 simultaneous requests for a client-server pair. The requests are multiplexed and the responses are demultiplexed over a single connection.

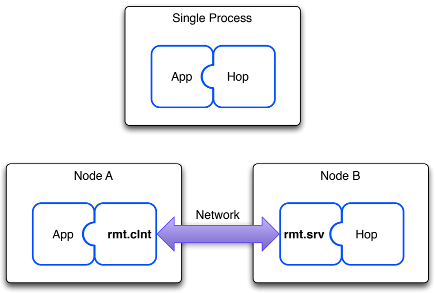

Fig. 1. Example of single-process and remote Hop deployment

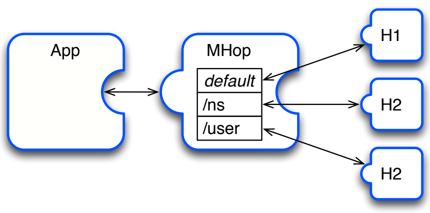

Fig. 2. Example of MHop deployment

Figure 1 shows deployment of a Hop interface can be made within a single process, where the Hop implementation and the application are linked together in a single binary, or over the network where rmt.clnt serves as a proxy to the remotely deployed Hop implementation. No changes to the Hop implementation is required in order to make it available over the network.

KHop: Hop Reference Implementation KHop implements a key-value store in RAM. It is a basic building block that we use in other Hop implementations. KHop implements all required functionality and can be regarded as a reference Hop implementation.

In addition to the default storage in RAM, KHop allows its users to provide per-entry behavior for the entries they create. If the entry objects (created outside the Hop interface operations) implement all or some of the Hop operations, the custom operations are called when required. This functionality allows easy implementation of special entries with dynamic content and/or side effects.

MHop: Hop Aggregation Service MHop is a proxy implementation that combines the entries from multiple Hop instances into a single namespace. It is somewhat equivalent to the Unix Virtual File System (VFS) for file systems. MHop allows a Hop instance to be "mounted" to a prefix, passing all operations to keys that start with the prefix to that instance. MHop uses trie structures to minimize the impact of the additional key processing on the performance. Figure 2 shows how MHop can be used to redirect operations to three Hop instances: all operations for keys starting with `/ns` are redirected to Hop2, the keys starting with `/user` are redirected to Hop3, all operations for other keys are redirected to the default Hop1 instance.

The purpose of MHop is to provide a centralized access to all available Hop instances and hide the details of their deployment from the applications.

CHop: Distributed Caching Service One of the main reasons to introduce version numbers in Hop was to allow trading strict consistency for performance. One of the ways to improve performance is to use some caching services. We implemented simple distributed caching that utilizes the Hop features.

Upon creation of CHop, the user provides a Hop instance to be cached, and restrictions on the memory and number of entries the cache should use. The CHop instance can also join a CHop group of instances.

Each instance of CHop keeps local cache of the most recently used entries. It uses the version number provided to the Get operation to decide whether to return the locally stored value, or update it from the original Hop instance. All Set operations also update the local cache.

In addition to the local cache, CHop allows definition of *consistency domains*. All applications that use a specific consistency domain are guaranteed to see the updates that other applications within the domain make, regardless of the version number. There are no guarantees when updates made outside of the consistency domain would be available. Using specific (more recent than the cache or `Newest`) version while retrieving a value will always retrieve the specified version from the original Hop instance, and ensure that from that moment on, no other application within the consistency domain will receive an older value.

Figure 3 shows an example of a group of six CHop instances, with two consistency domains. The applications using C1 and C3, or C4 and C6 will see consistent views of the entries, provided by the Hop instance, while C2 and C5 may see inconsistencies.

The definition of consistency domains in CHop is implemented by using special prefix `#/cache/`*domain-name/*. By prepending the prefix while accessing the keys, multiple users of CHop instances can ensure that they will get consistent view of the values. For example, applications A and B can use consistent domain `bar` to ensure consistent view of key `foo` by accessing entries with names `#/cache/bar/foo`. For example, for a resource manager job information(process, memory layout) would be consistent within the nodes of a job but not consistent to jobs on other nodes.

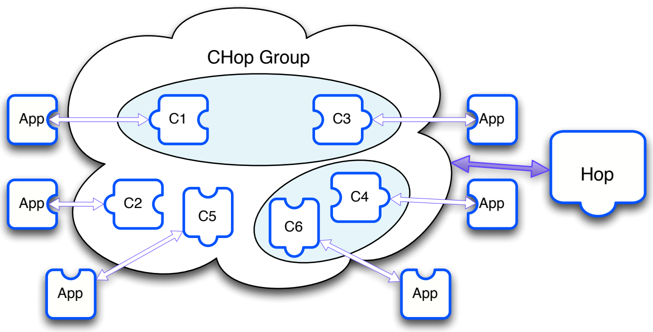

Fig. 3. Example of CHop deployment with 6 instances and 2 consistency domains

In addition to the consistency domain prefixes, CHop allows the users to create a special Hop instance that hides the prefixes and uses the consistency domain for all entries simplifying implementation.

4.3 Examples Of Using the Building Blocks

The Hop implementations described in this section utilize the basic Hop building blocks to create services that can be used in real deployments.

SHop SHop builds on KHop by adding special entries that are standard for deployable Hop services. SHop defines the following special entries:

`#/id`	Hop implementation identification.
`#/keys`	Returns all keys in the data store.
`#/keys:`*regexp*	Returns all keys that match the specified regular expression.
`#/keynum`	Returns the number of keys in the data store.

SHop also uses rmt.srv to provide single-instance data store, available over the network.

D2Hop D2Hop is a distributed Hop implementation that partitions the key space across multiple D2Hop instances, running on different nodes. D2Hop calculates a 32-bit hash value for each key and redirects the operation to the appropriate instance responsible to that value. Each instance is handles a range of hash values. Currently D2Hop supports FNV-1a[3] and Adler-32 hash functions. D2Hop implementation is built on top of the KHop and Rmt building blocks.

Although D2Hop handles most instance failures, it doesn't replicate the entries' data across multiple instances and failures may lead to data loss, this is currently being developed.

The D2Hop instances belong to one of the three categories: leader, followers, or clients. The leader is responsible for keeping track of the followers health and handles joining and leaving the service. Failure of a leader causes failure of the whole service. Followers are responsible for handling the operations for part of the keys' namespace as well as redirecting the operations for the rest of the keys to the appropriate instance. Each follower is connected to all other followers and the leader. The clients are connected to some, or all followers and redirect the operations to the appropriate instance.

The maintenance of the D2Hop infrastructure is implemented by using Hop operations on a special entry `#/conf`. The content of the entry is a string, with first line describing the leader, and following lines for each of the followers. A line contains the instance network address as well as a list of hash ranges the instance is responsible for. To join the service, the instance performs atomic `Append` with its address to the leader's `#/conf` entry. The leader updates its configuration and returns the new content of the entry, that includes the new instance and what keys it is responsible for. Each follower and client keep track of version of the configuration they have and constantly tries to retrieve the next version. Once an instance joins or leaves the service, they receive the new configuration and its updated version.

D2Hop can run any Hop implementation as distributed service, its instances simply redirect the calls they receive to the Hop instance, specified on initialization.

Although D2Hop doesn't handle failures, it can be used for deployment where failures are unlikely and serves as a good example how to implement distributed Hop implementations.

ChordHop ChordHop is a Hop implementation that uses the Chord [14] distributed service for partitioning the key space. It implements Chord's strong stabilization algorithm. CHordHop is built on top of KHop and Rmt.

Similarly to D2Hop, the Chord maintenance protocol is implemented as Hop operations on special entries. Each Chord node defines the following entries:

`#/chord/successor:`*ID*	The value of the entry is the address of the Chord node that is successor of the specified *ID*. The current node might contact other nodes to find out the successor.
`#/chord/predecessor`	Returns the address of the predecessor of the Chord. `TestSet` operation on the entry notifies the node that another node might be its predecessor while returning the old predecessor.
`#/chord/finger`	Returns the current node's finger table. Used for debugging only.
`#/chord/ring`	Returns description of Chord's ring. Used for debugging only.

ChordHop doesn't replicate the entries, so even though Chord stays connected when failures occur, data might be lost. In the future we plan to add replication.

ChordHop clients act similarly to the Chord nodes, maintain correct finger table, etc. But unlike the Chord nodes, they don't join the Chord ring.

5 Results

We evaluated the performance of some of our Hop implementations at small scale and compared one of them to two other key-value stores. All tests were run on a 128-node cluster with Infiniband QDR interconnect. Each node has 16 cores and 32GB RAM.

To evaluate the performance, we created a simple benchmark that executes random key-value store operation on randomly generated keys and random value sizes. The probability for each type of operation is weighted, giving higher chance of retrieval operations over the ones that create, remove or modify an entry. For the Hop centric tests (Figures 4 and 6), 55% of the operations are `Get`, 35% are `Set`, 5% are `Create`, 4% are `Remove`, 3% are `TestSet`, and 3% are `Atomic`. For the comparison with other key-value stores (Fig. 5, we removed the atomic operations and had tested with 60% `Get` operations, 30% `Set`, 5% `Create`, and 5% `Remove`.

We ran one key-value store server and 16 clients on each node.

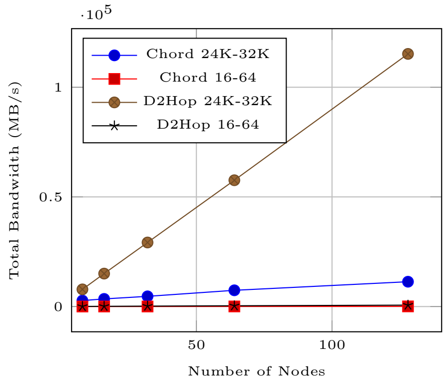

Fig. 4. Performance of D2Hop and Chord for different value sizes

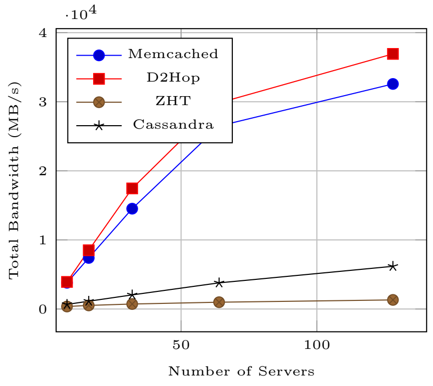

Fig. 5. Performance of Memcached, ZHT, and D2Hop

We run the benchmark for D2Hop and ChordHop implementations, scaling the number of key-value store nodes from 8 to 128. We run the tests for two sizes of the entry values – from 16 to 64 bytes, and from 24KB to 32KB. The tests run using the Infiniband Verbs API. Figure 4 shows the results from these tests. As expected, the D2Hop implementation scaled better than Chord due to the additional hops for key look-up required for Chord. Runs with smaller sized entries were latency bound and delivered much worse bandwidth values.

The second test we run compared the D2Hop implementation to some of the other key-value stores that we were able to run on the cluster. We compared our Hop implementation to Memcached [5], Cassandra and ZHT [11]. We used TCP over IB for running this set of tests, because Memcached, Casasandra and ZHT don't support running directly on Infiniband. We used value sizes between 24KB and 32KB for these tests. Both Memcached and ZHT were configured to disable data replication. ZHT was set to keep the data in RAM instead of storing it to disk. Cassandra was configured to store its data to ramdisk. We varied the number of servers handling constant number of clients, running on 128 nodes. Figure 5 shows the performance of the three key-value stores. Although D2Hop is not optimized for performance, it slightly outperforms Memcached. Cassandra's performance suffers because even though it doesn't store its data to real disks, it performs all operations (like creating commit logs, merging data files, etc.) that are required for persistence of the data.

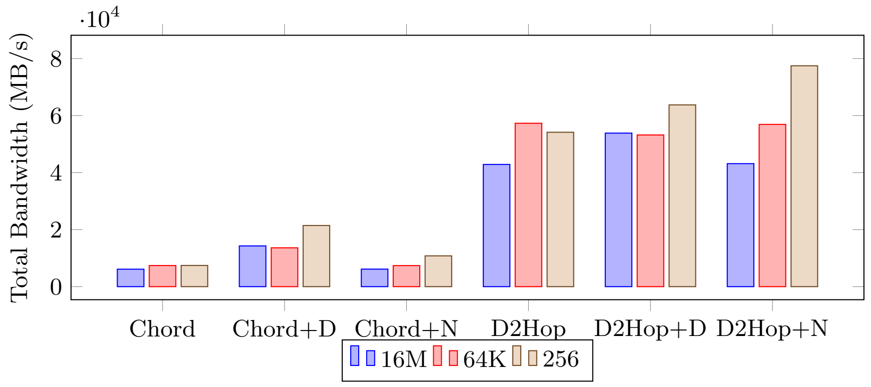

Fig. 6. Bandwidth for various Hop configurations for different key space sizes

The last test we run compared the performance of more complex configurations of Hop implementations, using combinations of some of the basic building blocks as well as D2Hop and ChordHop. The goal of the tests was to investigate how the utilization of caching with CHop, with and without consistency domains will affect the overall performance of the key-value store. The tests were run on 64 nodes, with value sizes between 24KB and 32KB. We compared the performance of running D2Hop and Chord directly (Chord, D2Hop lines) with running them in combination with CHop with consistency domains (CHord+D, D2Hop+D) and without consistency domains (Chord+N, D2Hop+N). When consistency domains are used, we create 8 domains, each with 8 nodes as members. As the efficiency of the cache depends on the key space size, we ran the tests for three different sizes of the key space – 256 keys, 64K keys and 16M keys. Half of the retrieval operations specify `Any` as version (making them candidates for being served from the CHop cache), while the other half specifies `Newest` ensuring that the operation reaches D2Hop or ChordHop. Figure 6 shows the results of the tests. For the small number of keys (256), all values can fit in the local cache, which makes the CHop without consistency domains best performer when request latency is not very high (as with D2Hop). For larger number of keys, as well as stores with higher latency (as ChordHop), the cooperative caching of

CHop's consistency domains allow higher cache hit rate within the domain and result in better performance.

6 Future Work

The most important missing functionality is proper replication for instance failures. Our plan is to implement some of the consensus algorithms (Paxos [10], Raft [13]) as operations on special entries, then build the entry replication on top of them. Similarly to the consistency domains, we are planning to provide support for *replication domains* that provide different replication levels for different sets of entries.

We are also planning to use Hop to implement an instance of Hobbes [1] Global Information Bus and related services. Hobbes is an operating system and runtime (OS/R) framework for extreme-scale systems, funded by the Department of Energy. The Global Information Bus is a software layer that provides mechanisms for sharing status information needed by other components in the OS/R. This include nodes status, locations and availability of various services, performance data, etc.

Another extension of this project we are exploring is using special keys as a way to read and write portions of complex entries. We are planning to create a subset of the DRepl[7] language designed for describing dataset layouts and use it to specify what subsets of the data entries are accessed.

7 Conclusion

The Hop interface allows the implementation and deployment of data store services, middleware and clients with different consistency and reliability requirements. The Hop project provides a set of building blocks as well as some examples on how to use them to create real-world services. The simplicity of the protocol and the support for special entry names allows it to be used as an RPC framework for exascale system services as well as to provide custom representation of data sets.

The services we implemented show that Hop is well fit for various system services implementations and can be used to hide some of the complexity of exascale systems from the user applications.

Acknowledgments

This work was performed at the Ultrascale Systems Research Center in the High Performance Computing Division at Los Alamos National Laboratory, and is supported by the U.S. Department of Energy DE-FC02-06ER25750.

References

1. Brightwell, R., Oldfield, R., Maccabe, A.B., Bernholdt, D.E.: Hobbes: composition and virtualization as the foundations of an extreme-scale os/r. In: Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers. pp. 2:1–2:8. ROSS '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2491661.2481427
2. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. pp. 205–220. SOSP '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1294261.1294281
3. Eastlake, D., Fowler, G., Vo, K.P., Noll, L.: The fnv non-cryptographic hash algorithm (2012)
4. Feinberg, A.: Project voldemort: Reliable distributed storage. In: Proceedings of the 10th IEEE International Conference on Data Engineering (2011)
5. Fitzpatrick, B.: Distributed caching with memcached. Linux J. 2004(124), 5– (Aug 2004), http://dl.acm.org/citation.cfm?id=1012889.1012894
6. Geambasu, R., Levy, A.A., Kohno, T., Krishnamurthy, A., Levy, H.M.: Comet: An active distributed key-value store. In: OSDI. pp. 323–336 (2010)
7. Ionkov, L., Lang, M., Maltzahn, C.: Drepl: Optimizing access to application data for analysis and visualization. In: Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on. pp. 1–11 (2013)
8. Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., et al.: Oceanstore: An architecture for global-scale persistent storage. ACM Sigplan Notices 35(11), 190–201 (2000)
9. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 35–40 (April 2010), http://doi.acm.org/10.1145/1773912.1773922
10. Lamport, L.: Paxos made simple. ACM Sigact News 32(4), 18–25 (2001)
11. Li, T., Verma, R., Duan, X., Jin, H., Raicu, I.: Exploring distributed hash tables in highend computing. SIGMETRICS Perform. Eval. Rev. 39(3), 128–130 (Dec 2011), http://doi.acm.org/10.1145/2160803.2160880
12. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. pp. 401–416. ACM (2011)
13. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm
14. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: ACM SIGCOMM Computer Communication Review. vol. 31, pp. 149–160. ACM (2001)