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

Ke Wang
Illinois Institute of Technology
Los Alamos National Laboratory
kwang22@hawk.iit.edu

Abhishek Kulkarni
Indiana University
adkulkar@cs.indiana.edu

Michael Lang
Los Alamos National Laboratory
mlang@lanl.gov

Dorian Arnold
University of New Mexico
darnold@cs.unm.edu

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

ABSTRACT

Owing to the significant high rate of component failures at extreme scales, system services will need to be failure-resistant, adaptive and self-healing. A majority of HPC services are still designed around a centralized paradigm and hence are susceptible to scaling issues. Peer-to-peer 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 service workloads to support extreme-scale systems. The simulator is validated against existing distributed KVS-based services. 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 HPC service workloads into the simulator and presenting a KVS-based distributed job launch prototype.

Keywords

Key-Value Store, System Services, Discrete Event Simulation, Extreme Scales

1. INTRODUCTION

Due to the expected increases in component count and failure rates for extreme-scale supercomputers, system software and services will need to be fault-tolerant, self-healing and adaptive for efficient system utilization and sustained operation. However, leadership-class systems have traditionally been managed using manual or semi-automatic approaches under a single management domain. Additionally, many (if not most) HPC services are still designed around a centralized server with at most a single fail-over server and hence suffer from a single point of failure and scalability problems.

Large-scale systems have worked around some of these issues using hierarchical partitions of smaller functional units. However, these hierarchically partitioned services are often very loosely-coupled and do not present a coherent system for management or maintenance. From a management perspective, such systems are more laborious to maintain and pose serious scalability concerns for systems with hundreds of thousands of nodes. Such concerns suggest a move toward fundamentally scalable distributed system designs – a move further motivated by the growing amount of data that servers need to access quickly, reliably and in a consistent manner.

The broad goal of this work is to design and develop a framework that allows us to explore systematically the design space for HPC services and to evaluate the impact of different design choices. **The specific goal of this work is to evaluate the different distributed key-value store designs for extreme-scale systems, as we now motivate.**

SC '13 November 17-21, 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2503210.2503239

1.1 Key-Value Stores and HPC

In this work, we generally target high-performance computing services, such as those that support system booting, system monitoring, hardware or software configuration and management, I/O forwarding and run-time systems for programming models and communication libraries [15] [2] [31] [28]. For extreme-scale systems, these services all need to operate on large volumes of data in a consistent, resilient and efficient manner at extreme scales. We observe that these services commonly and naturally comprise of access patterns amenable to NoSQL abstractions, a data storage and retrieval paradigm that admits weaker consistency models than traditional relational databases.

These requirements are consistent with those of large-scale distributed data centers, for example, Amazon, Facebook LinkedIn and Twitter. In these commercial enterprises, NoSQL data stores – Distributed Key-Value Stores (KVS), in particular – have been used successfully [5] [18] [8]. 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.

By encapsulating distributed system complexity in the KVS, we can simplify HPC service designs and implentations. 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 [28]. Application developers from Sandia National Laboratory [14] 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 execu-

tion [23] [26] [22] [24] fabric – MATRIX [34] where KVS is used for task submission, dependency, and progress information; and the fusion distributed file system, FusionFS [37], where the KVS is used in tracking file-system metadata. Another example of a many task computing system is the Swift parallel programming system [35].

The many different use cases impose a varied set of requirements on the KVS. In this work, we developed a four-dimensional taxonomy to classify and specify these requirements; the various points along these four dimensions represent different KVS design choices. We then use this taxonomy to develop a simulator for evaluating different KVS designs. A simulation-based approach allows us to evaluate designs for extreme-scale platforms, which do not yet exist, and easily execute future studies of services other than KVS.

The key contributions of this work are:

- a taxonomy for classifying HPC system services;
- a simulation tool to explore KVS design choices for large-scale system services;
- and an evaluation of KVS design choices for extreme-scale systems using both synthetic and real workload traces.

The rest of this paper is organized as follows. In Section 2, we present our service taxonomy that allows us to explore the design space and categorize various system services for simulation. In Section 3, we describe the design and implementation of the KVS simulator. We describe our experimental setup and simulation results in Section 4 and present related work in Section 5. Finally, we conclude and propose extensions to this work in Section 6.

2. A TAXONOMY FOR HPC KEY-VALUE STORES

We developed a taxonomy to help us reason about distributed services in several ways: (1) the taxonomy gives us a systematic way to decompose services into their basic building block components; (2) the taxonomy allows us to categorize services based on the features of these building block components, and (3) the taxonomy suggests the configuration space to consider for evaluating service designs via simulation or implementation. Our taxonomy has five components. The first component is the *service model*, which is a high-level description of the service's functionality. Throughout the rest of this paper, we focus on the KVS service model. We now describe the other four components, namely *data model*, *network model*, *recovery model* and *data consistency model*, by combining specific instances of these components we then can define a *service architecture*.

(i) **Data Model** The data model defines how a service distributes and manages its data. For example, in a *centralized* data model, a single central component maintains and manages all data. Alternatively, data can be distributed amongst multiple servers – *partitioned*. The data can have different levels of replication; *unique* (no replication), *mirrored* (full replication) or *overlapped* (partial replication).

(ii) **Network Model** The *network model* dictates the interconnection topology of a service's components. Components can form a structured overlay, such as a ring, tree (of different shapes including binomial, k-ary and radix trees) or graph. Components can also comprise an unstructured overlay network. The network model may also differentiate between deterministic and non-deterministic data routing. While some overlay networks imply a complete membership set (eg. fully-connected), others may assume a partial membership set (eg. binomial graphs). These distinctions impact the communication overhead that can be attributed to each network model.

(iii) **Recovery Model** The *recovery model* specifies how a service deals with component failures. Common recovery techniques include *fail-over* and *checkpoint-restart* and *roll-forward* protocols. Other techniques like *triple modular redundancy* and *erasure codes* also ensure data integrity. Some of the recovery models are self-contained – recovery via logs from persistent storage and some require communication with peers to retrieve replicated state.

(iv) **Consistency Model** The *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. Depending on the data model and the corresponding level of replication, a system service might employ differing levels of consistency. The level of consistency is a trade-off between the server response time and how tolerant clients are to stale data. It can also compound the complexity of recovery mechanisms. Servers could employ *weak*, *strong*, or *eventual* consistency model.

Some specific instantiations of service architectures from the taxonomy are depicted shown in Figure 1 and Figure 2. For instance, c_{tree} is a service architecture with a *centralized* data model and a *tree-based* overlay network, consistency and recovery model are not depicted, but would need to be identified to define a complete *service architecture*. d_{fc}has a *distributed* data model with a *fully-connected* overlay network whereas $d_{chord}$is a distributed data model and has a Chord overlay network [29] with partial membership, again the consistency and recovery model are not show graphically.

Table 1 classifies existing KVS systems according to our taxonomy. These common and most-tested KVS architectures come mostly from the Internet domain. Services developed for the Internet generally are designed for high failure rates and loosely-synchronized components and state over a widely distributed area. In contrast, HPC services generally target a more tightly-coupled workload model. Our taxonomy allows us to understand qualitatively how these different usage models render different system design and to evaluate quantitatively the impact of these different design choices on a service architecture.

Using the taxonomy we can narrow the parameters and focus on the major components of KVS services. Simulation can then be used to narrow the design space for any specific KVS service application before any implementation has begun. Additionally we can eventually create modular KVS components to allow easy creation of extreme-scale services, but in this work we focus on the simulation of KVS for HPC services using our taxonomy as a basis.

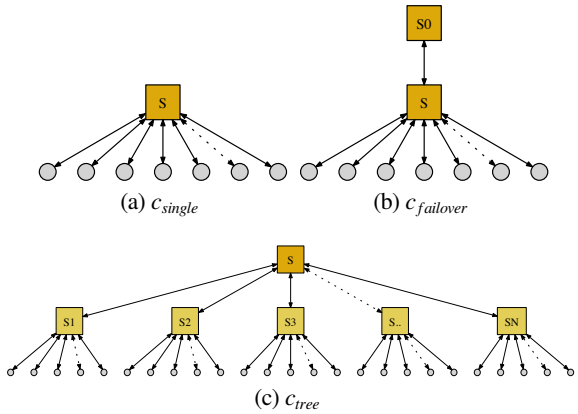
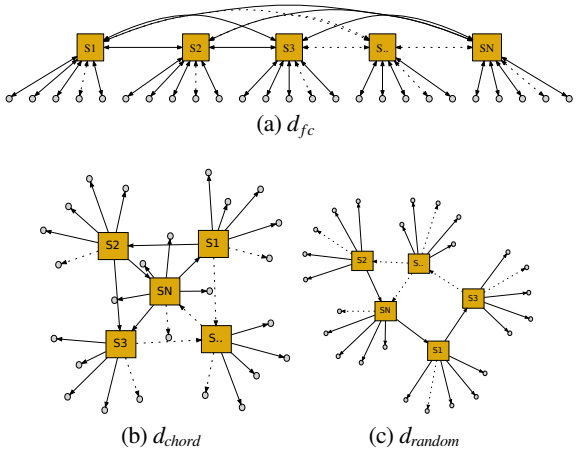


Figure 1: Centralized Service Architectures

3. SIMULATING KEY-VALUE STORES

In this work, we use simulation to study the various points in the design space of KVS systems. A simulation-based approach gives us a framework for not only studying KVS but also other HPC services



(a) d_{fc}

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

Figure 2: Distributed Service Architectures

in the future. This methodology also allows us to make sound design choices before we develop real prototypes and to evaluate design choices at scales for which real platforms do not exist yet. In this section, we describe our simulator's design and implementation. Our KVS service taxonomy dictates the major KVS components and, thus, directly informs the configuration space of our simulator. Our current simulator allows us to explore the previously described KVS network models, namely c_{single}, c_{tree}, $d_{fc}$and d_{chord}, here we assume a centralized data model for $c_{single}$and c_{tree}, and a distributed data model for $d_{fc}$and d_{chord}. The simulator is extendable to other network and data models. The above models can be configured with N-way replication for the recovery model and either eventual or strong for the consistency model.

3.1 Simulator Overview

Each simulation consists of millions of clients that connect to thousands of shared servers, the number of clients and servers are configurable, and how a server is selected by client can be preconfigured or random, and is easily modified.

The workload for the KVS simulation is a stream of `PUT`s and `GET`s. At simulation start, we model unsynchronized clients by having each simulated client stall for a random time before submitting its requests. This step is skipped when modeling synchronized clients. At this point, each client connects to a server (as described below) and sends synchronous (or blocking) `GET` or `PUT` requests as specified by a workload file. After a client receives successful responses to all its requests, the client-server connection is closed.

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, however the requests within one queue are processed sequentially. Since clients send requests synchronously, each server's average number of queued requests is equal to the number of clients that server is responsible for.

For our distributed architectures, $d_{fc}$and d_{chord}, our simulator supports two mechanisms for server selection. In the first mechanism, *client selection*, each client has a membership list of all servers; a client selects a server by hashing the request key and using the hash as an index into the server list. Alternatively, a client may choose a random server to service their requests. In the second mechanism, *server selection*, each server has the membership list of part or all of the servers and clients submit requests to a dedicated server. Client selection has the benefit of lower latency, but leads to significant overhead in updating the membership list when servers fail. Server selection, on the other hand, puts a heavier burden on the servers.

3.2 Cost parameters

The simulation results are dependent on the attribution of the communication and processing costs due to the sevice architecture, we explain and justify the cost parameters in this section. Figure 3 shows the scenario of server selection with five enqueued operations, three of which are resolved locally and forwarded to the local operations queue, and two are resolved remotely by forwarding to other servers. The composite overheads include client send, (CS), client receive, (CR), server send (SS), server receive (SR) and local request processing, (LP). CS is comprised of message serialization time, t_{ser}, and message transmission time, t_{com}, calculated as $msgSize/BW + lat$, where $msgSize$ is the message size in bytes, BW is the peak network bandwidth and lat is half of the round-trip time (RTT). CR is the message deserialization overhead, t_{des}. SS includes the time when the server finishes the last queued task in the communication queue t_{qc}, the overhead of packing a message by the server t_{ss}, and t_{com}. SR is the summation of t_{qc}, the overhead of unpacking a message by the server t_{sr}. LP includes the time when the server finish processing the last queued request t_{qp}, and the request processing time t_p. For a locally resolved query, the time to finish it consists of client send overhead CS, server receive overhead SR, locally processed time LP, server send to client overhead SS, and client receive overhead CR. This is applicable to all the architectures. For a remotely resolved request in d_{fc}, the time to finish it includes client send overhead CS, server receive overhead SR, server forwarding request overhead SS+SR, locally processed time LP, server returning processing result overhead SS+SR, server send to client overhead SS, and client receive overhead CR. For a remotely resolved request of d_{chord} involving k hops to find the predecessor of the responsible server, the time to finish the request includes client send overhead CS, server receive overhead SR, overhead of finding the predecessor $2\times k\times(SS+SR)$, server forwarding request overhead to the responsible server SS+SR, locally process time LP, server returning processing result overhead SS+SR, server send to client overhead SS, and client receive overhead CR. 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}: $t_{RR} = t_{LR} + 2\times(SS+SR)$

For d_{chord}: $t_{RR} = t_{LR} + 2\times k\times(SS+SR)$

where k is the number of hops to find the predecessor.

Service	Description	Data Model	Network Model	Recovery Model	Consistency
Voldemort	Key-Value Store	Distributed	Fully-connected	N-way Replication	Eventual
Cassandra	Key-Value Store	Distributed	Fully-connected	N-way Replication	Both
D1HT	Key-Value Store	Distributed	Hierarchical	N-way Replication	Strong
Pastry	Key-Value Store	Distributed	Partially-connected	N-way Replication	Strong
ZHT	Key-Value Store	Distributed	Fully-connected	Replicas	Eventual

Table 1: Representative KVS-based services categorized in the taxonomy

All of the architectures under study are derived from this basic design using communication and processing costs.

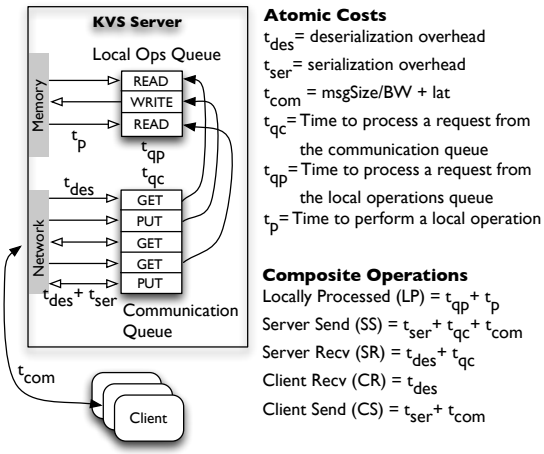


Figure 3: KVS client/server simulator design

3.3 Data and Network Models

Our simulator supports different data and network models: centralized server (c_{single}), centralized server with multiple second-tier aggregation servers in a tree overlay (c_{tree}), distributed servers with fully connected topology (d_{fc}), distributed servers with chord protocol topology (d_{chord}).

For c_{single} and c_{tree}, all data is stored in the centralized server. The only difference is that c_{tree} has a layer of aggregation servers from whom the client submits requests to. The aggregation size (number of requests being packed before sending) of each individual aggregation server is dynamically changing according to the loads. Basically, it is equal to the number of clients, which have more requests left to be processed, among all the clients that the aggregation server is responsible for. The upper bound is the number of clients the aggregation server serves. Currently, they only do request aggregation. In the future, it is possible to simulate `PUT` caches in these servers for read intensive workloads.

For d_{fc} and d_{chord}, the key space along with the associated data is evenly partitioned among all the servers ensuring a perfect load balancing. In d_{fc}, data is hashed to the server in an interleaved way (key modulo the server id), while in d_{chord}, consistent hashing [16] is the method for distributing data. The servers in d_{fc} have global knowledge of all servers, while in d_{chord}, each server has only partial knowledge of the other servers; specifically this is logarithmic of the total number of servers with base 2 and is kept in a table referred to as the *finger table* on each server.

3.4 Recovery Model

The recovery model defines how a node recovers its state and how it rejoins the system after a failure. This not only involves the way the recovered server gets back all of its data, but also considers how to update the replica information of other servers which are affected due to the recovery. The first replica of a failed server is notified by an external mechanism (EM) [29], which could be another distributed service, that knows the status information of all servers when the primary server recovers. Then it sends all the replicated data (including the data of the recovering server, and other servers for which the recovering server acts as a replica host) to the recovered server. The recovery is then finalized once the server acknowledges it has received all of its state.

3.4.1 Failure/Recovery

Fail/recover events are generated because of servers failing and rejoining the dynamic overlay network. The servers can fail the system very frequently in an extreme-scale system, in which the mean-time-to-failure (MTTF) could be in the order of hours [25]. For simplicity, in our simulator, we make an assumption that fail/recover events happen at fixed periods and there is only one server failing or recovering in the system at a given time. At the very beginning, all servers are up in the system. When fail/recover event occurs, the simulator randomly picks one server and flips its status (up to down, down to up). In order to notify other servers about a node's failure, we implement both the eager and lazy methods. In the eager method, we assume an EM sending a failure message to notify other online servers. This would be done either with a broadcast message (in d_{fc}) or with chaining in which every node notifies the left node in their finger tables (for d_{chord}). In the lazy method, servers are not notified but realize the failure of a node when they try to communicate with it. When a node recovers, it gets the membership list from the EM and then does a broadcast (d_{fc}), or it receives its finger table from the EM and then notify all the servers that should have the joined node in their finger tables (d_{chord}) [29]. In our simulations, the failure event without a replication model impies that when a server fails, all the messages (requests coming from the clients, or forwarding messages from other servers) would fail. In addition, the clients wouldn't try the failed requests again, even if the failed server recovers. The purpose of this is to isolate the effect of failures so that it can be measured separately.

3.4.2 Failure/Recovery with Server Replication

We implement a replication model in the simulator for the purpose of handling failures. In c_{single} and c_{tree}, one or more failovers are added, while in d_{fc} and d_{chord}, each server replicates its data in the consecutive servers. Failure events complicate server replication clients would try again several times for failed requests before they turn to the next replica in case the failed server recovers at a later time. In all cases, we assume that clients know or can trivially find the replicas of servers to whom they send requests. When the primary server fails, the first replica sends the failed server's data to one more other server to ensure that there are enough replicas. In addition, all the servers that replicate data on the failed server would also send their data to one more server.

3.5 Consistency Model

Our simulator implements two consistency models: strong consistency and eventual consistency [32].

3.5.1 Strong Consistency

In strong consistency, every replica observes every update in the same order. Updates are made with atomicity guarantee so that no two replicas may store different values for the same key at any given time. In the case of strong consistency, a client sends requests to a dedicated server (primary replica). The `Get` requests are processed and returned back to client immediately while the `Put` requests are first processed locally and then sent to the replicas. The primary replica waits for an acknowledgement from each other replica before it responds back to the client. When a node recovers from failure, before the node gets all its data (see section 3.4), the node caches all the requests that directed to it. In addition, the first replica (notified by the EM) of the newly recovered server migrates all pending `Put` requests, which should have been served by the recovered server, to the recovered server. This ensures that only the primary replica processes `Put` requests at anytime in the system while there may be more than one replica processing `Get` requests.

3.5.2 Eventual Consistency

In eventual consistency, given a sufficiently long period of time over which no changes are sent, all updates can be expected to propagate eventually through the system and all the replicas will be consistent.

Although different replicas may be storing different versions of the same key at a given time, all replicas will eventually receive every update and will eventually agree on all values. In our simulator, after finding the correct server (one-hop hashing for d_{fc}or $logN$ hops routing for d_{chord}), the requests are sent to a randomly choosen replica, which is called the "coordinator". This is to model inconsistent updates of the same "key" in different replicas, and also achieves load balancing among all the replicas. There are three key parameters that change the behavior of the consistency mechanism: the number of replicas–N, the number of replicas that must participate in a successful `Get` request–R, and the number of replicas that must participate in a successful `Put` request–W. We satisfy $R+W>N$ to guarantee "read our writes" [32]. Similar to Dynamo and Voldemort, our simulator uses version clock to track different versions of data and detect conflicts. A version clock is a vector of <*serverId*,*counter*> pairs for each key in each server. It specifies from one server's point of view, how many updates have been processed by each server for a specific key. If all update counters in a vector clock $V1$ are smaller than or equal to all corresponding update counters in a vector clock $V2$, then $V1$ precedes $V2$, and can be replaced by $V2$. If $V1$ overlaps with $V2$, then there is a conflict. For a `Get` request, the coordinator first reads the value locally, sends the request to other replicas, and waits for <*value*,*version clock*> responses. When a replica receives a `Get` request, it first checks the corresponding version clock. If the version clock precedes the coordinator's version clock, then the replica reponses success; otherwise, responses failure with its own <*value*,*version clock*> pair. The coordinator waits for $R-1$ successful responses. If there are multiple versions of data, the coordinator returns all the versions to the client who is responsible for reconciliation (according to an application specific rule such as "largest value wins") and writing back the reconciled version. For a `Put` request, the coordinator generates a new vector clock by incrementing its counter of the current vector clock by 1, and writes the new version locally. Then the coordinator sends the request along with the new vector clock to other replicas and waits for responses. If the new vector clock is preceded by a replica's vector clock, the replica accepts the update and responds success; otherwise, responds failure. If at least $W-1$ replicas respond success, the `Put` request is considered successful.

3.6 Implementation

After evaluating several simulation frameworks such as OMNET++ [30], OverSim [3], SimPy [17], we chose to develop the simulation using a peer-to-peer system simulator, PeerSim [21], because of its support for extreme scalability and dynamicity. Among the two simulation engines it provides, we use the discrete-event simulation (DES) [33] engine because it is more scalable and realistic compared to the cycle-based one for large-scale simulations. Every behavior in the system is converted to an event and tagged with an occurrence time. All the events are inserted in the global event queue, which is sorted based on the event occurrence time. At each iteration, the simulation engine fetches the first event and executes the corresponding actions resulting in insertion of zero or more events in the queue. The simulation terminates when the queue is exhausted or a termination condition is satisfied.

There are several parameters that define the simulator environment and operation. The simulation is built on top of PeerSim, which is developed in Java, and has about 10,000 lines of code. The input to the simulation is a configuration file, which specifies the system architecture, the values of the parameters, etc. The names and descriptions of the parameters are shown in Table 2. There are no other dependencies.

4. EVALUATION

The goal of the experimental evaluation is to give insight into the design space of distributed KVS for HPC, and to show the capabilities of our simulation tool to expose costs inherent in design choices. To accomplish these goals we use KVS simulation to evaluate the overhead of different service architectures as we vary the major parameters that we identified in section 2, taxonomy. We present these experimental results by incrementally adding complex features such as replication, server response to failure/recovery, and consistency, so that we can measure the individual contributions to the overhead due to support for these distributed service features and their associated protocols. This overhead is reflected in the communication intensity of the specific service architectures and the additional communication as a result of the additional features.

But first we need to setup the experimental environment by describing the metrics being used to evaluate the simulation, present the workloads to be applied, and validate our simulation against two real systems: ZHT [19], a zero-hop distributed KVS and Voldemort [8], an open-source implementation of Amazon's Dynamo [5] KVS.

4.1 Experimental Setup

4.1.1 Metrics

The metrics being used to evaluate our simulation include *Aggregated Server Throughput*, *Per-Client Throughput Speedup*, *System Efficiency*, and *Number of Messages*. The specifications and reasons for choosing them are as follows:

- **Per Client Throughput Speedup**: This represents the relative average throughput of each client from the client's perspective. Per client throughput is calculated as number of requests finished divided by the time to finish the requests, it measures how fast a client's requests are processed as viewed by the client.
- **System Efficiency**: This is calculated as measured aggregated server throughput divided by the ideal aggregated server throughput. The aggregated server throughput is calculated as number of total requests / simulation time. The ideal throughput is calculated with zero communication overhead. This metric shows the system utilization, the proportion of time when the system is actually processing requests. The higher the efficiency is (close to 1), the more fully utilized the system would be.
- **Number of Messages**: This is comprised of number of messages for processing requests, failure events, replication, and consistency – strong or eventual. Via message counters in the simulator, we have a numerical view of the overhead of each property.

Name	Description
BW	Network bandwidth
lat	Network congestion latency
$msgSize$	Message size
$idLength$	Key length (in bits)
t_{ss}	Server message packing overhead
t_{sr}	Server message unpacking overhead
t_{cs}	Client message packing overhead
t_{cr}	Client message unpacking overhead
t_p	Time to process a request locally
$numReqPerClient$	Number of request per client
$numClientPerServ$	Number of clients per server
$FailureRate$	Failure frequency
$numReplica$	Number of replicas
R	Number of succesful `Get` responses
W	Number of succesful `Put` responses
$numTry$	Number of retries before a client talks to the replica

Table 2: Parameter names and descriptions

4.1.2 Experiment Environment

The software versions used are: Sun 64-bit JDK version 1.6.0_22; PeerSim jar package version 1.0.5. The simulations were run on a single node; the largest amount of memory required for any of the simulations was 25GB and the longest run-time was 40 minutes (millions of clients, thousands of servers, and tens of millions of requests). Given how light-weight our simulator is, an extremely large scale range can be explored.

4.1.3 Parameters

The parameters presented in Table 2 are displayed in Table 3 with their values. The network parameters are chosen to reflect large-scale systems such as IBM BlueGene/P [1], and the Kodiak cluster from the Parallel Reconfigurable Observational Environment (PRObE) [11]. The base request-processing time is taken from samples of processing time from services such as `memcached` [9] and ZHT [19].

4.1.4 Workloads

The simulations are performed with up to 1 million clients each submitting 10 `Get` or `Put` requests. We did experiments to verify that higher numbers (e.g. 100, 1k, 10k) of get/put requests gave nominally the same results. These values are configurable by changing the parameters, *numReqPerClient* and *numClientPerServ*, from table 3. For d_{fc} and d_{chord}, we increment the number of clients by 1024 and the number of servers by 1 as we scale.

In exploring the overhead of different distributed system service features (Sections 4.2 though 4.6), we use the synthetic workload, in which, 10M tuples of <*type*,*key*,*value*> are generated with a uniform random distribution (URD) (50% `Gets` and 50% `Puts`) and placed in a workload file. Each client would then read 10 requests in turn and execute their workloads.

Realistic workloads are also employed. They are described in more detail and applied in section 4.7, where we feed the KVS simulator with these distributed HPC service traces to show the generality of KVS.

4.1.5 Validation

We validate our simulator against two real systems: a zero-hop KVS, ZHT [19], and an open-source implementation of Amazon Dynamo KVS, Voldemort [8]. Both systems serve as building block for system services. ZHT is used to manage metadata of file systems (FusionFS), monitor task execution information of job scheduling systems (MATRIX), and to store the resource and job information for our distributed job launch prototype which is under improvement, while Voldemort is used to store data for the LinkedIn professional network.

In the case of validating against ZHT, the simulator was configured to match the *client selection* that was implemented in ZHT. ZHT was run on the IBM Blue Gene/P machine (BG/P) in Argonne National Laboratory with up to 8K nodes and 32K cores. We used the published network parameters of BG/P in our simulator. We used the same workload as that

Parameter	BG/P	Kodiak
BW	6.8 Gbps	1.0 Gbps
lat	100 μs	120 μs
msgSize	10 KB	10 KB
idLength	128 bits	128 bits
t_{ss}	50 μs	40 μs
t_{sr}	50 μs	40 μs
t_{cs}	50 μs	40 μs
t_{cr}	50 μs	40 μs
t_p	500 μs	1500 μs
numReqPerClient	10	10
numClientPerServ	1024	1024
numTry	3	3

Table 3: Simulation Parameters

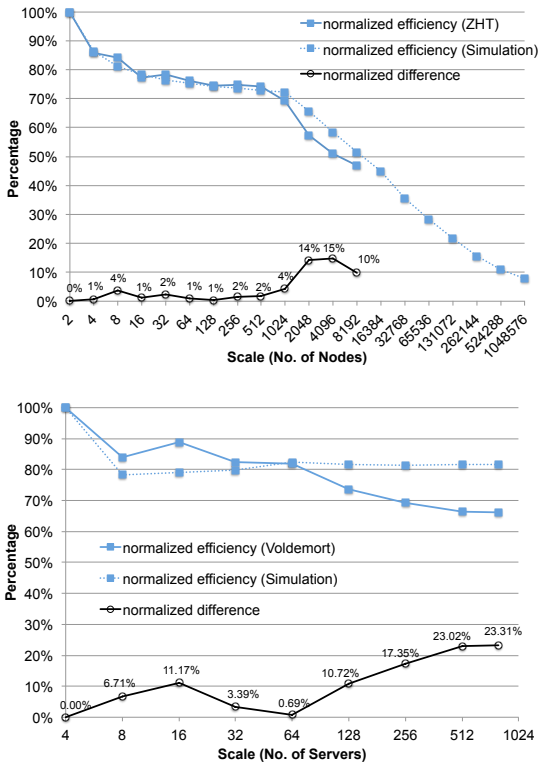


Figure 4: Validation of the simulator: Simulating ZHT and Voldemort

used to in ZHT: each node has a client and a server, each client submits 10K requests with URD, the length of the key is 128 bits, and the message size is 134 bytes. The result in Figure 4 shows that our simulator matches ZHT with up to the largest scale (8K nodes with 32K cores) that ZHT was run. The biggest difference was only 15% at large scales. The ZHT curve depicts decreasing efficiency after 1024 nodes, because each rack of BG/P has 1024 nodes. Within 1024 nodes (one rack), the communication overhead is small and relatively constant, leading to constant efficiency (75%). After 1024 nodes, the communication spans multiple racks, leading to more overhead and decreasing efficiency.

In the case of Voldemort, we focused on validating the eventual consistency model of the simulator. The simulator was configured to match the *server selection* d_{fc} model, with each server backed by 2 replicas and responsible for 1024 clients, and an associated eventual consistency protocol with versioning and read-repair. We ran Voldemort on the Kodiak cluster from PROBE with up to 800 servers, and 800k clients. Each client submitted 10 random requests. As shown in Figure 4, our simulation results match the results from the actual run of Voldemort within 10% up to 256 nodes. At higher scales, due to resource over-subscription, an acute degradation in Voldemort's efficiency was observed. Resource over-subscription means that we ran way too many client processes (up to 1k) on one physical node. At the largest scale (800 servers and 800 nodes), there will be 1k client processes running on each node, leading to serious resource over-subscription.

Given the good validation results, we believe that the simulator can offer convincible performance results of the various architectural features we are interested in. This allows us to weigh the service architectures and the overheads that are induced by the various features.

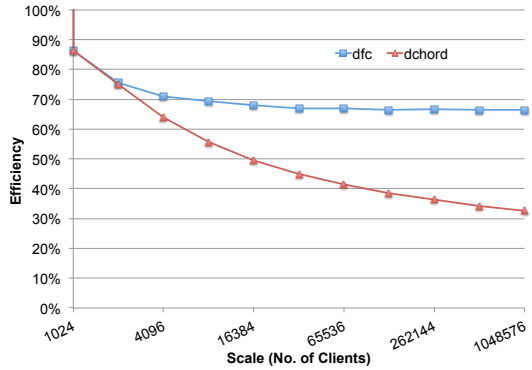
4.2 Architecture Comparisons of d_{fc} vs d_{chord}

In the section, we compare two of the distributed architectures ($d_{fc}$and d_{chord}) with a very basic scenario (no replication, no failures or consis-

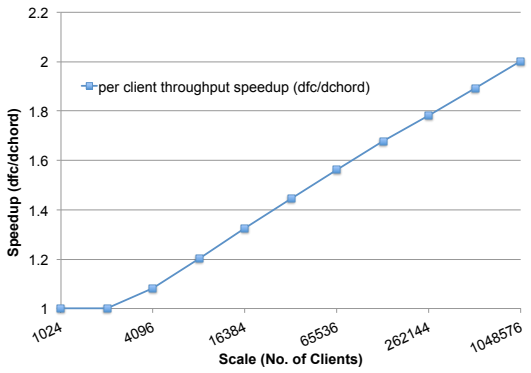
tency model). The synthetic workload is used to investigate the tradeoffs between these service architectures at increasingly large scales.

The performance comparison between d_{fc} and d_{chord} is shown in Figure 5. Doubling client counts indicates doubling server counts because each server is responsible for 1K nodes. With d_{fc}, we observe that the efficiency has a fairly constant value (67%) at extreme scales, meaning that d_{fc} scales perfectly linearly with respect to the scale. On the other hand, in d_{chord}, as we scale up, the efficiency decreases smoothly (Figure 5(a)). This is due to the additional routing required by d_{chord} to satisfy requests: one-hop maximum for d_{fc} and log_N hops for d_{chord}. We show the per client throughput speedup of d_{fc} with respect to d_{chord} in Figure 5(b). As the system scales up, the speedup is increasing. Up to 1M clients, the per client throughput of d_{fc} is about 2 times of that of d_{chord}. This is again due to the extra hops required to find the correct server responsible for the key in d_{chord}.

The conclusion is that at the base case (no replication, failure, or consistency model), the partial connectivity of d_{chord} results in higher latency to satisfy client requests, due to the additional routing. Fully connected topology results in faster response to client requests (twice as fast as partial connectivity), because it needs at most one hop routing.



(a) Server efficiency

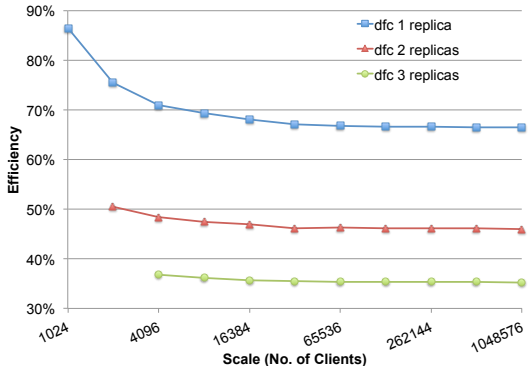


(b) Per client throughput speedup

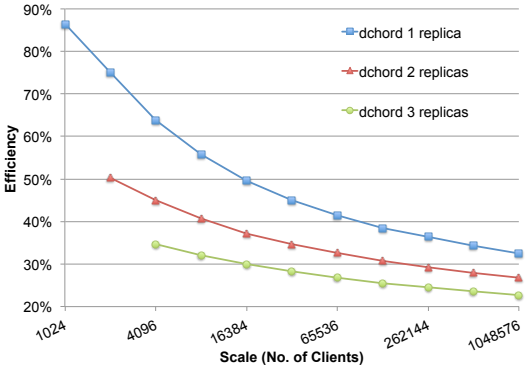
Figure 5: d_{fc} and d_{chord} performance comparison

4.3 Replication overhead

This section investigates the replication overhead associated with d_{fc} and d_{chord}. The data is collected for comparison of 1 to 3 replicas and the results are shown in Figure 6. It shows that there is always additional cost for additional replicas due to the added communication and processing overhead involved in propagating the `Put` requests to the extra replicas. Comparing d_{fc} and d_{chord}, we see that d_{fc}has more overhead than d_{chord} when adding extra replicas. This is due to the low efficiency of d_{chord}, since d_{chord} has higher overhead for routing, the additional fixed overhead for the replicas is relatively small when comparing with the routing overhead. In d_{fc} the relatively low routing overhead results in a larger impact on efficiency. In d_{fc}, the first added replica adds over 20% overhead (67% to 46%) and decreases the efficiency by 29.9% (20% / 67%), the second added replica introduces over 10% overhead (46% to 35%) and decreases the efficiency by 23.9%, while in d_{chord}, overheads of the first and second added replicas are 6% (33% to 27%) and 4% (27% to 23%), and efficiency decreased by 18.2% and 14.8%, respectively at the largest scale.



(a) d_{fc}



(b) d_{chord}

Figure 6: Service architecture replication overhead

4.4 Server Failure Effects

We add failure events (servers fail and possibly recover) to the simulator to emulate the failure rates of extreme-scale class systems. Here we do not use extra replicas and a request to a "failed" server would be dropped and the client wouldn't retry. This is to measure the overhead of dynamicity of the service architectures. A failure event has to be forwarded to every other server in the d_{fc} network, whereas in the d_{chord} network, it is sent to log_N nodes resulting $(log_N)^2$ messages. Different failure frequencies (high 60/min, medium 20/min, and low 5/min) are studied with the results shown in Figure 7, for both d_{fc} and d_{chord}.

As seen in Figure 7, the higher the failure frequency is, the more overhead introduced (lower efficiency curve for higher failure frequency). But the dominating factor is the client messages. These client request-processing messages dwarf the number of communication messages of the failure events, which is a secondary factor even at the frequency of 60 events/min. Furthermore, this effect is getting dominating as the system scales up; the efficiency gaps are getting smaller and smaller until they disappear at the largest scales. For example: given 1M clients each sending 10 requests, 1K servers, and 5 failure events, for d_{fc}, we have at most 10M client forwarding messages and failure events only require $5{\times}1K$ messages (small compared to 10M), while for d_{chord}, we

have $1M \times log(1K) = 10M$ forwarding message, and $5(log(1K))^2 = 500$ failure messages. This illustrates how client-request messages dominate even with the added messages required to deal with server failures and recovery. Figure 7 shows that d_{fc} is more efficient than d_{chord} at the studied failure rates.

In order to validate the correctness of failure events represented in our simulator, we show the number of communication messages of one failure event in Figure 8. In this experiment we expect to see d_{fc} increase linearly while d_{chord} should increase logrithmically, we turn off the client workload messages and configure the simulator to process merely 10 failure events, and collect the average number of messages. The regression models in Figure 8 validate the linear and logarithmic relationships of the number of messages with respect to the number of servers for d_{fc} and d_{chord}, respectively. The R-Square values of the models are 0.865 and 0.998, which demonstrate that our models are representative.

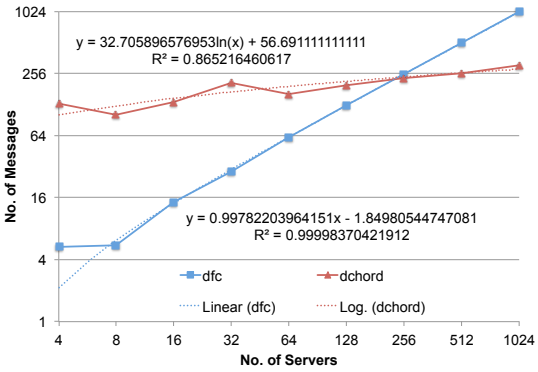


Figure 8: Regression of the number of failure event messages with respect to the system scale

4.5 Server Failures with Replication

This section explores the overhead of failure events when a service is configured to keep updated replicas for resilience. The clients try to resend the failed requests to the primary several times (specified by the *numTry* parameter, which is set to a default value of 3) before it turns to the next replica. The reason is the "failed" server might recover at later time or a server may respond slowly due to high load. The results are shown in Figure 9 which displays the efficiency comparison between d_{fc} and d_{fc} configured with failures and supporting replication, and between d_{chord} and d_{chord} configured with failure events and supporting replication, respectively. We fixed the parameters at 3 replicas, and a low failure rate (5 failure events per minute), with a strong consistency model.

As seen in Figure 9, both d_{fc} and d_{chord} have significant efficiency degradation when both failures and replication were enabled (blue solid line vs blue dotted line, red solid line vs red dotted line). The performance degradation of d_{fc} is more severe than that of d_{chord}, 44% (67% to 23%) for d_{fc} vs, 17% (32% to 15%) for d_{chord}. The reason for this is hidden in Table 4.

This table lists the number of messages of each property (process request, failure, strong consistency of replicas) for both d_{fc} and d_{chord}. We see that at extreme-scale the request-process message count (dominant factor) does not increase much when turning on failures and extra replicas for both d_{fc} and d_{chord}. The failure event message count is negligible and the number of strong consistency messages of replicas increases significantly about the same rate for both d_{fc} and d_{chord}. However, these added messages account for about 1/3 (20M/60M) for d_{fc}, and less than 1/8 (20M/170M) for d_{chord}. Due to the high request process message count in d_{chord} the performance degradation of d_{fc} seems more severe than that of d_{chord}. The overhead of replication is costly as can be seen from Figure 9, tuning a service to the appropriate replication will have a large impact on performance.

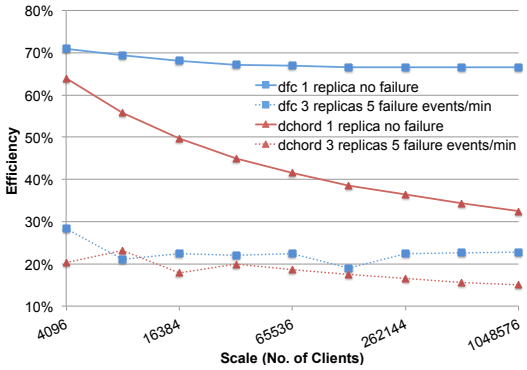


Figure 9: The effect of failure events with servers using replication

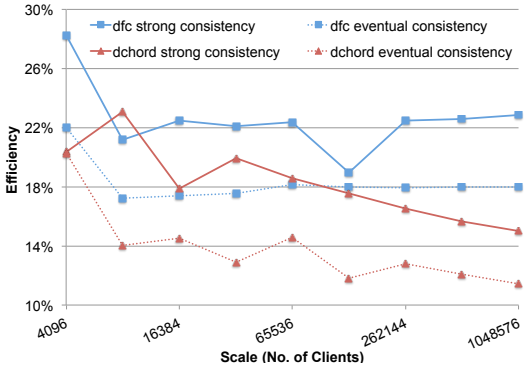


Figure 10: Strong consistency and eventual consistency

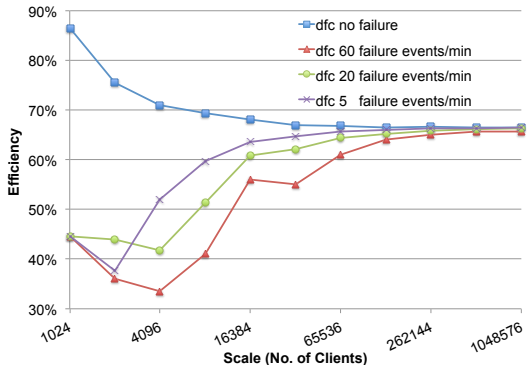
4.6 Strong and Eventual Consistency

We compare the overhead of strong and eventual consistency models in this section. We enable server failures with 5 failure events per minute and add 2 extra replicas. For eventual consistency, we configure (N, R, W) to be (3, 2, 2), which is the typical configuration of Amazon Dynamo [5]. The efficiency comparison between strong and eventual consistency for both d_{fc} and d_{chord} is shown in Figure 10. We also list the number of messages for request processing, failure events, and consistency models in Table 5.

We see in Figure 10 that eventual consistency has more overhead than the strong consistency for both d_{fc} and d_{chord}. From strong to eventual consistency, efficiency reduces by 4.5% for d_{fc} and 3% for d_{chord} at extreme-scale. In Table 5, we see that the request-process message count doesn't vary much for both $d_{fc}$and d_{chord}. However, for consistency messages, eventual consistency introduces about twice (41M/21M) the number of messages than strong consistency. This is because in eventual consistency each request would be forwarded to all other N=3 replicas and the server waits for R=2 and W=2 successful acknowledgments. Whereas, with strong consistency, just the `Put` requests would be forwarded to all other replicas. Eventual consistency gives faster response times to clients but there is a cost with respect the to the communication overhead as shown in Table 5. This need to be considered when designing services.

4.7 KVS Applicability to HPC Services

In this section, we show that KVS can be used as a general building block for developing HPC services. First, we evaluate the two architectures through our simulator with different workloads: job launch, monitoring, and I/O forwarding. These three workloads were obtained from



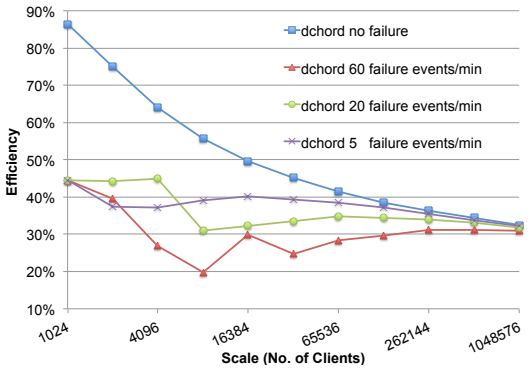


Figure 7: d_{fc} and d_{chord} effects of failure events

	process message count				failure message count		strong consistency count	
# Clients	d_{fc}	d_{chord}	d_{fc} (C&R)	d_{chord} (C&R)	d_{fc} (C&R)	d_{chord} (C&R)	d_{fc} (C&R)	d_{chord} (C&R)
4096	143.426	184.956	312.426	246.153	0.033	4.549	217.669	87.106
8192	307.282	491.178	404.126	596.151	0.042	0.445	175.384	170.838
16384	634.826	1207.952	726.112	1472.551	0.066	28.615	336.498	377.072
32768	1290.604	2825.860	1386.706	3039.180	0.114	0.712	665.388	661.926
65536	2601.066	6409.968	2698.502	6637.741	0.210	0.590	1322.836	1315.460
131072	5222.814	14256.736	5319.564	14542.027	0.402	0.888	2632.554	2627.996
262144	10465.436	31279.104	10564.664	31702.242	0.786	0.996	5251.380	5247.638
524288	20950.976	67947.732	21050.058	68422.326	1.554	1.127	10492.476	10485.882
1048576	41922.386	146570.7	42020.528	147105.826	3.090	1.274	20985.110	20979.146

Table 4: Number of messages for d_{fc}, d_{chord} with and without failure and replica (C&R) (in thousands)

	process message count				failure message count				consistency message count			
	sc		ec		sc		ec		sc		ec	
# Clients	d_{fc}	d_{chord}	d_{fc}	d_{chord}	d_{fc}	d_{chord}	d_{fc}	d_{chord}	d_{fc}	d_{chord}	d_{fc}	d_{chord}
4096	312.426	246.153	141.468	211.356	0.033	4.549	0.030	0.355	217.669	87.106	167.198	164.472
8192	404.126	596.151	391.926	682.689	0.042	0.445	0.054	0.588	175.384	170.838	340.236	328.208
16384	726.112	1472.551	733.134	1466.287	0.066	28.615	0.086	23.705	336.498	377.072	668.216	655.372
32768	1386.706	3039.180	1356.776	3076.448	0.114	0.712	0.15	0.827	665.388	661.926	1317.964	1312.000
65536	2698.502	6637.741	2675.156	6647.217	0.210	0.590	0.210	0.768	1322.836	1315.460	2628.146	2621.648
131072	5319.564	14542.027	5324.970	14776.662	0.402	0.888	0.534	1.093	2632.554	2627.996	5257.768	5251.656
524288	21050.058	68422.326	21033.874	68689.058	1.554	1.127	2.070	1.398	10492.476	10485.882	20982.712	20972.940
1048576	42020.528	147105.826	42008.038	147975.997	3.090	1.274	4.118	1.573	20985.110	20979.146	41954.552	41951.924

Table 5: No. of messages of strong consistency (sc), and eventual consistency (ec) for both d_{fc} and d_{chord} (in thousands)

real traces of three HPC system services: job launch using SLURM, monitoring by Linux Syslog, and I/O forwarding using FusionFS [37] distributed file system. We also implemented a distributed job launch prototype based on SLURM and the distributed KVS, ZHT.

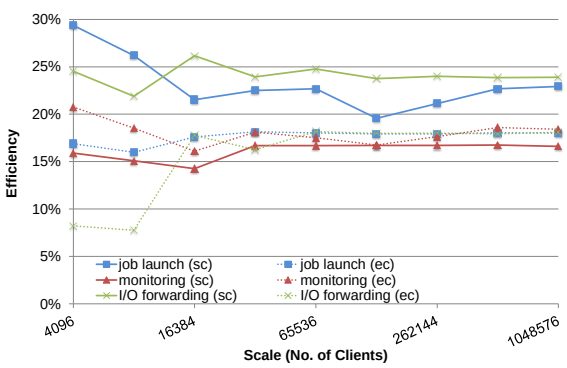
4.7.1 Simulation with Various Workloads

We run simulations with three workloads obtained from typical HPC services: job launch, monitoring, and I/O forwarding. The specification of each workload is listed below:

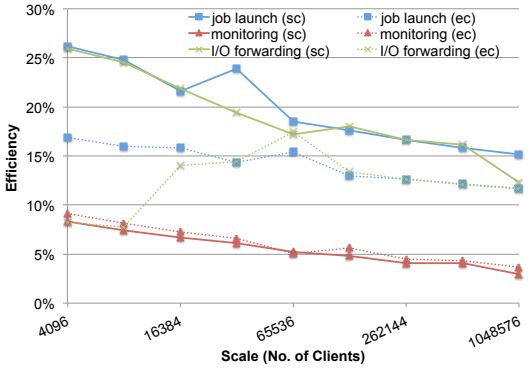
- **Job Launch**: The job launch workload is obtained by monitoring the messages between the server and client during a MPI job launch. Though the service is not implemented in a distributed fashion the messages to and from the clients should be representative regardless of server structure and this in turn drives the communication between the distributed servers. Job launch is characterized by control messages from the distributed servers (`Get`) and the results from the compute nodes back to the servers (`Put`).
- **Monitoring**: The monitoring workload is obtained from a 1600 node cluster's syslog data. This data was then categorized by message-type (denoting the key-space) and count (denoting the probability of each message). This distribution was then used to generate the workload which is completely `Put` dominated.
- **I/O Forwarding**: The I/O forwarding workloads is generated by running FusionFS distributed file system, which uses ZHT for metadata management. The client first creates 100 files, and then operates (reads or writes with 50% probability) each file once. We collect the log of the ZHT metadata server.

We extend and feed these real workloads to our simulator in order to investigate the applicability of our KVS simulator for HPC system services. The workloads obtained are not big enough for an extreme-scale system. For job launch and I/O forwarding workloads, we repeat the workloads several times until reaching 10M requests, and the key of each request is generated with URD within our 64-bit key space. For the monitoring workload, there are 77 kinds of message types with each one having a different probability. We generate 10M `Put` requests; the key is generated based on the probability distribution of the message types and is mapped to our 64-bit key space. We point out that these extensions are not enough to reflect every detail of these workloads. Nevertheless, they do reflect some important properties; the job launch and I/O forwarding workloads reflect the time serialization property and the monitoring workload reflects the probability distribution of all obtained messages.

Figure 11 shows the efficiency of these workloads with both strong



(a) d_{fc}



(b) d_{chord}

Figure 11: d_{fc} and d_{chord} with different workloads

and eventual consistency for d_{fc} and d_{chord}. 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. The ratio of the number of `Get` to `Put` requests is 50.9% to 49.1% for job launch, and 57.6% to 42.4% for I/O forwarding. For the monitoring workload, eventual consistency outperforms strong consistency because all the requests are `Put` type, which requires all $N-1$ acknowledgment messages from the other $N-1$ replicas in strong consistency whereas just $R-1$ or $W-1$ acknowledgment messages from the other $N-1$ replicas in eventual consistency. 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.

The above results demonstrate that our simulator is capable of simulating different kinds of system services as long as the workloads of these services could be transformed to `Put` or `Get` requests, which is true for the HPC services we have investigated.

4.7.2 *Distributed Job Launch*

In designing the next-generation distributed job management systems for HPC applications at extreme-scale, we developed a distributed job launch prototype based on the SLURM resource management system, combined with the distributed KVS, ZHT. The prototype is comprised of multiple controllers, each one managing several SLURM daemons in contrast to SLURM's centralized *slurmctrld*. The controllers are fully-connected, where each controller is aware of all of the other controllers (d_{fc}). ZHT is used to store the job metadata and is also used to resolve any contention for the resources (via the atomic compare and swap [13] operation of ZHT). By using ZHT to hide the complexities (e.g. failure, replica and consistency models) involved in creating distributed services, we show that a distributed KVS can be used as a building block for distributed services and can speed the development and deployment of these services.

Inspired by the work stealing [4] concept, we developed a novel resource stealing protocol, each controller uses this protocol for resource allocation when launching jobs. Each controller stores local free node list in ZHT and when launching a job it first checks the local free nodes. If there are enough available nodes locally, then the controller directly allocates the nodes; otherwise, it will query ZHT for other partitions from which it can steal resources. As long as there are not enough nodes to satisfy the allocation, the resource stealing protocol will randomly select a controller to steal nodes. When the selected controller (victim) has no available nodes, the stealing controller (stealer) sleeps and retries. If the stealer experiences several failures in a row because the victims have no free nodes, it will release all resources it has obtained, and then retries the resource stealing protocol again. The number of retries and lenght of sleep after a stealing failure are critical to the performance. After tuning these parameters, we have chosen to retry 3 times, and sleep 100ms, for the following experiments.

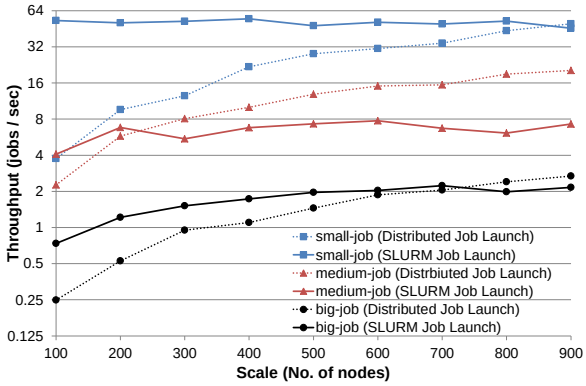


Figure 12: Comparison of SLURM and our distributed job launch prototype with various workloads

We configured each controller to manage 100 SLURM daemons, and compared SLURM job launch with our distributed job launch prototype under a small-job workload – job size is 1 node, a medium-job workload – job size is 1 to 50 nodes, and a big-job workload – job size is 50 to 100 nodes. We consider the simple sleep 0 job, which is a minimal job, but enough to measure the overhead of the two services. Figure 12 shows the comparison results with up to 9 controllers and 900 SLURM daemons (SLURM just has one centralized controller) running on the PRObE testbed system with the same total number of jobs. For a small-job workload, each controller launches 100 jobs with each job requiring 1 node; for medium- and big-job workloads, each controller launches 50 jobs, with each job requiring a random number of nodes ranging from 1 to 50, and from 50 to 100, respectively.

From Figure 12, we see that: (1) Under a small-job workload, with standard SLURM, the throughput has a decreasing trend as the number of nodes increase (53 jobs / sec at 100 nodes, down to 46 jobs / sec at 900 nodes), while for our prototype, the throughput increases linearly with respect to the scale, and this linear speedup trend is expected to continue at larger scales. By scales of 900 nodes, our prototype launches jobs faster than SLURM (50 jobs/sec vs. 46 jobs/sec); (2) Under a medium-job workload, for standard SLURM, as the number of nodes scales up, the throughput increases slightly (from 4 jobs / sec

at 100 nodes to 8 jobs/sec at 600 nodes), and then is almost constant or with a slow decrease, while for our prototype, the throughput increases approximately linearly with respect to the scale (from 2.8 jobs / sec at 100 nodes to 20.8 jobs / sec at 900 nodes). After 200 nodes, our prototype can launch jobs faster than SLURM, and the gap increases with the scale. At the largest scale, the distributed job launch prototype can launch jobs about 2.5 times faster than SLURM, and the trend implies that this speedup would continue at larger scales; (3) With a big-job workload, SLURM's throughput increases up to 700 nodes (from 0.75 jobs / sec at 100 nodes to 2.25 jobs / sec at 900 nodes), and then the throughput is almost constant (actually a little bit decreasing from 700 nodes to 900 nodes). However, like the previous two cases, our prototype experiences a linear increasing trend for throughput with respect to scale (from 0.28 jobs / sec at 100 nodes to 2.7 jobs / sec at 900 nodes). After 700 nodes, our prototype can launch jobs faster than SLURM.

The results show that employing a distributed KVS (ZHT), our prototype outperforms SLURM even with the added complexity of a distributed service and speculate on the potential positive impact such distributed job launch architecture could have at the extreme scales of tomorrow. We are now improving ZHT and our prototype, and will implement more distributed HPC system services, such as distributed monitoring, distributed queuing sevices, using ZHT in the future.

5. RELATED WORK

Work that is directly related to the simulation of services includes an investigation of peer to peer networks [7], telephony simulations [6], simulations of load monitoring [10], and simulation of consistency [27]. However, none of the investigations are focused on HPC, or combine replication, failures and consistency. In this survey [20], the authors have investigated 6 distributed hash tables and have categorized them in an algorithm taxonomy. This work focuses on the overlay networks, and presents a discussion on the performance. In [12], peer-to-peer file-sharing services are traced and these are used to build a parameterized model. Another taxonomy was developed for grid computing workflows [36], in which they use the taxonomy to categorize existing grid workflow managers to find their common features and weaknesses. But none of these investigations are targeting HPC workloads and services, and none of them use the taxonomy to drive features in a simulation, which then can be used in the design of services.

6. CONCLUSIONS AND FUTURE WORK

In this work we justified the usefulness of distributed KVS for HPC. We developed a service taxonomy and categorized KVS services into 4 components, and then used this taxonomy to drive the development of a KVS simulator that was parameterized across data, network, recovery, and consistency models.

The simulator was validated, and experiments were conducted to quantify and compare the overheads of fail/recover events, replication and different consistency models for these architectures under synthetic and realistic HPC workloads. With an extendable simulator as a tool, we can design system services for large-scale and make feature choices to reduce the effort of implementation.

The conclusions we draw are as follows: when the client requests dominate the communication–up to billions at extreme scales–the fully connected topology (d_{fc}) actually scales very well under moderate failures (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 partial-knowledge topology (d_{chord}) 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 d_{chord}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 includes extending the simulator to cover more of the taxonomy, adding network models and recovery models such as log-based replay. Additionally, we will use the simulator to model other system services and validate these at small scale, and 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 improving the distributed job launch prototype. Other service building block implementations will be developed to support c_{single}, c_{tree}, and $d_{chord}$with various properties from the taxonomy. This would allow other developers to select the best service architecture based on the simulations at the desired scale and build their services from the base implementations of these building blocks.

Acknowledgements

This work was supported by the U.S. Department of Energy under contract DE-FC02-06ER25750, and in part by the National Science Foundation under award CNS-1042543 (PRObE). This work was also in collaboration with the FusionFS project supported by the National Science Foundation grant NSF-1054974. This research also used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DEAC02-06CH11357. The publication has been assigned the LANL identifier LA-UR-12-25175. We thank Tonglin Li, Dongfang Zhao and Hakan Akkan for their help and suggestions.

7. REFERENCES

[1] Overview of the ibm blue gene/p project. *IBM Journal of Research and Development*, 52(1.2):199 –220, jan. 2008. ISSN: 0018-8646.

[2] Nawab Ali, Philip Carns, Kamil Iskra, et al. Scalable I/O Forwarding Framework for High-Performance Computing Systems.

[3] I. Baumgart, B. Heep, and S. Krause. Oversim: A flexible overlay network simulation framework. In *IEEE Global Internet Symposium, 2007*, pages 79 –84, may 2007.

[4] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, Sept., 1999.

[5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al. Dynamo: Amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, Stevenson, Washington, USA, 2007. ACM.

[6] Ibrahima Diane, Ibrahima Niang, and Bamba Gueye. A Hierarchical DHT for Fault Tolerant Management in P2P-SIP Networks. In *Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems*, ICPADS '10, pages 788–793, Washington, DC, USA, 2010. IEEE Computer Society.

[7] Tien Tuan Anh Dinh, Georgios Theodoropoulos, and Rob Minson. Evaluating Large Scale Distributed Simulation of P2P Networks. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, DS-RT '08, pages 51–58, Washington, DC, USA, 2008. IEEE Computer Society.

[8] A. Feinberg. Project Voldemort: Reliable Distributed Storage. ICDE, 2011.

[9] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004. ISSN: 1075-3583.

[10] Bogdan Ghit, Florin Pop, and Valentin Cristea. Epidemic-Style Global Load Monitoring in Large-Scale Overlay Networks.

In *Proceedings of the 2010 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 3PGCIC '10, pages 393–398, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Gary Grider. Parallel Reconfigurable Observational Environment (PRObE), October 2012. Available from http://www.nmc-probe.org.

[12] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, et al. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 314–329, Bolton Landing, NY, USA, 2003. ACM.

[13] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, Jan., 1991.

[14] Michael A. Heroux. Toward Resilient Algorithms and Applications, April 2013. Available from http://www.sandia.gov/~maherou/docs/HerouxTowardResilientAlgsAndApps.pdf.

[15] Morris A. Jette, Andy B. Yoo, and Mark Grondona. SLURM: Simple Linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *9th International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2003)*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60, Seattle, Washington, USA, June 24, 2003. Springer-Verlag.

[16] David Karger, Eric Lehman, Tom Leighton, et al. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, El Paso, Texas, United States, 1997. ACM.

[17] Tony Vignaux Klaus Muller. Simpy:documentation, May 2010. Available from http://simpy.sourceforge.net/SimPyDocs/index.html.

[18] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35–40, April 2010. ISSN: 0163-5980.

[19] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, et al. ZHT: A Light-weight Reliable Persistent Dynamic Scalable Zero-hop Distributed Hash Table. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS'13)*, IPDPS '13, Boston, MA, USA, 2013. IEEE Computer Society.

[20] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7(2):72–93, 2005.

[21] Alberto Montresor and Márk Jelasity. PeerSim: A scalable P2P simulator. In *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, September 2009.

[22] Ioan Raicu. Many-task computing: Bridging the gap between high-throughput computing and high-performance computing. Proquest, Umi Dissertation Publishing, 2009.

[23] Ioan Raicu, Ian Foster, Mike Wilde, et al. Middleware support for many-task computing. *Cluster Computing*, 13(3):291–314, September 2010. ISSN: 1386-7857.

[24] Ioan Raicu, Ian Foster, Yong Zhao, and Alex Szalay. Towards data intensive many-task computing. In *Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management, IGI Global Publishers, 2009*.

[25] Ioan Raicu, Ian T. Foster, and Pete Beckman. Making a case for distributed file systems at exascale. In *Proceedings of the third international workshop on Large-scale system and application performance*, LSAP '11, pages 11–18, San Jose, California, USA, 2011. ACM.

[26] Ioan Raicu, Ian T Foster, and Yong Zhao. Many-task computing for grids and supercomputers. In *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, pages 1–11. IEEE, 2008.

[27] M. Raihan Rahman, W. Golab, A. AuYoung, K. Keeton, and J.J. Wylie. Toward a Principled Framework for Benchmarking Consistency. 2012.

[28] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. MRNet: A software-based multicast/reduction network for scalable tools. In *in: Proc. IEEE/ACM Supercomputing '03*, 2003.

[29] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari 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.

[30] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, Simutools '08, pages 60:1–60:10, Marseille, France, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[31] Abhinav Vishnu, Amith R. Mamidala, Hyun-Wook Jin, and Dhabaleswar K. Panda. Performance Modeling of Subnet Management on Fat Tree InfiniBand Networks using OpenSM. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18 - Volume 19*, IPDPS '05, pages 296.2–, Washington, DC, USA, 2005. IEEE Computer Society.

[32] Werner Vogels. Eventually consistent. *Queue*, 6(6):14–19, October 2008. ISSN: 1542-7730.

[33] Ke Wang, Kevin Brandstatter, and Ioan Raicu. Simmatrix: Simulator for many-task computing execution fabric at exascale. In *21st High Performance Computing Symposia (HPC'13), Part of the SCS Spring Simulation Multiconference (SpringSim'13) in cooperation with ACM/SIGSIM*, San Diego, CA, USA, Apr., 2013.

[34] Ke Wang, Anupam Rajendranl, and Ioan Raicu. "matrix: Many-task computing execution fabric at exascale". 2013. Available from http://datasys.cs.iit.edu/projects/MATRIX/index.html.

[35] Michael Wilde, Ioan Raicu, Allan Espinosa, et al. Extreme-scale scripting: Opportunities for large task parallel applications on petascale computers. In *SCIDAC, Journal of Physics: Conference Series 180*, page 012046, 2009.

[36] Jia Yu and Rajkumar Buyya. A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing*, 3(3-4):171–200, 2005. ISSN: 1570-7873.

[37] Dongfang Zhao and Ioan Raicu. Distributed file systems for exascale computing. In *Doctoral Showcase, SC'12: Proceedings of the 2012 ACM/IEEE Conference on Supercomputing*, Salt Lake City, UT, November 2012.