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  
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 hierarchical partitions 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 manage 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.

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] [28] [25]. 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 implementations. 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 [25]. 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 execution fabric.
where KVS is used for task submission, dependency, and progress information; and the fusion distributed file system, FusionFS [33], where the KVS is used in tracking file-system metadata.

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 (e.g., fully-connected), others may assume a partial membership set (e.g., 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, $c_{\text{tree}}$ is a service architecture with a central 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. $d_{\text{bin}}$ has a distributed data model with a fully-connected overlay network whereas $d_{\text{chord}}$ is a distributed data model and has a Chord overlay network [26] 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.
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 PUTs and GETs. 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, \( d_f \) 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 service 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 overhead includes client send (\( CS \)), client receive (\( CR \)), server send (\( SS \)), server receive (\( SR \)) and local request processing (\( LP \)). CS is comprised of message serialization time, \( t_{msg} \), and message transmission time, \( t_{lat} \), calculated as \( t_{msg} + BW \times lat \), where \( BW \) is the peak network bandwidth and \( lat \) is half of the round-trip time (RTT). CR is the message deserialization overhead, \( t_{ds} \). SS includes the time when the server finishes the last queued task in the communication queue \( t_{q} \), the overhead of packing a message by the server \( t_{p} \), and \( t_{ms} \). SR is the summation of \( t_{q} \), the overhead of unpacking a message by the server \( t_{u} \). LP includes the time when the server finish processing the last queued request \( t_{lp} \), and the request processing time \( t_{r} \). For a locally resolved query, the time to finish it consists of client send overhead \( GS \), 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 remotely resolved request in \( d_f \), the time to finish it includes client send overhead \( GS \), 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 in \( d_{chord} \) involving \( k \) hops to find the predecessor of the responsible server, the time to finish the request includes client send overhead \( GS \), server receive overhead \( SR \), server forwarding request overhead \( SS+SR \), server processing 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 (\( t_{HR} \)) is given by

\[
\begin{align*}
    t_{LR} &= CS + SR + LP + SS + CR \\
    t_{HR} &= t_{LR} + 2 \times (SS + SR)
\end{align*}
\]

For \( d_f \): \( t_{HR} = t_{LR} + 2 \times (SS + SR) \)

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

where \( k \) is the number of hops to find the predecessor.

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

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
<th>Data Model</th>
<th>Network Model</th>
<th>Recovery Model</th>
<th>Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voldemort</td>
<td>Key-Value Store</td>
<td>Distributed</td>
<td>Fully-connected</td>
<td>N-way Replication</td>
<td>Eventual</td>
</tr>
<tr>
<td>Cassandra</td>
<td>Key-Value Store</td>
<td>Distributed</td>
<td>Fully-connected</td>
<td>N-way Replication</td>
<td>Both</td>
</tr>
<tr>
<td>DHT</td>
<td>Key-Value Store</td>
<td>Distributed</td>
<td>Hierarchical</td>
<td>N-way Replication</td>
<td>Strong</td>
</tr>
<tr>
<td>Pastry</td>
<td>Key-Value Store</td>
<td>Distributed</td>
<td>Partially-connected</td>
<td>N-way Replication</td>
<td>Strong</td>
</tr>
<tr>
<td>ZHT</td>
<td>Key-Value Store</td>
<td>Distributed</td>
<td>Fully-connected</td>
<td>Replicas</td>
<td>Eventual</td>
</tr>
</tbody>
</table>

Table 1: Representative KVS-based services categorized in the taxonomy.
3.3 Data and Network Models

Our simulator supports different data and network models: centralized server (Csingle), centralized server with multiple second-tier aggregation servers in a tree overlay (Ctree), distributed servers with fully connected topology (dFC), distributed servers with chord protocol topology (dChord).

For Csingle and Ctree, all data is stored in the centralized server. The only difference is that Ctree 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. Therefore, they only do request aggregation. In the future, it is possible to simulate PUT caches in these servers for read-intensive workloads.

For dFC and dChord, the key space along with the associated data is evenly partitioned among all the servers, ensuring a perfect load balancing. In dFC, data is hashed to the server in an interleaved way (key modulo the server id), while in dChord, consistent hashing [16] is the method for distributing data. The servers in dChord have global knowledge of all servers, while in dChord, 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) [26], 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 that it has received all of its state.

3.4.1 Failure/Recovery

Fail/recovery events are generated because of servers failing and re-joining the dynamic overlay network. The servers can fail the system very frequently in an extreme-scale system, in which the mean-time-to-

Figure 3: KVS client/server simulator design

Atomic Costs

\[ t_{des} = \text{deserialization overhead} \]
\[ t_{ser} = \text{serialization overhead} \]
\[ t_{com} = \text{msgSize/BW + lat} \]
\[ t_{qc} = \text{Time to process a request from the communication queue} \]
\[ t_{tp} = \text{Time to process a request from the local operations queue} \]
\[ t_{qp} = \text{Time to perform a local operation} \]

Composite Operations

Locally Processed (LP) = \( t_{des} + t_{p} \)
Server Send (SS) = \( t_{ser} + t_{qc} + t_{com} \)
Server Recv (SR) = \( t_{des} + t_{qc} \)
Client Recv (CR) = \( t_{des} \)
Client Send (CS) = \( t_{ser} + t_{com} \)

\[ \text{Client} \]
\[ \text{KVS Server} \]
\[ \text{Local Ops Queue} \]
\[ \text{Communication Queue} \]

3.5 Consistency Model

Our simulator implements two consistency models: strong consistency and eventual consistency [29].

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 find-
ing the correct server (one-hop hashing for \( d_{1 \log N} \) or logN hops routing for \( d_{shard} \)), the requests are sent to a randomly chosen 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” [29].

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 \( V \) are smaller than or equal to all corresponding update counters in a vector clock \( V' \), then \( V \) precedes \( V' \), and can be replaced by \( V' \). If \( V \) overlaps with \( V' \), 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 responds success; otherwise, responds 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, thePut request is considered successful.

3.6 Implementation

After evaluating several simulation frameworks such as OMNET++ [27], 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) [30] 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.

4.1.2 Experiment Environment

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BW</td>
<td>Network bandwidth</td>
</tr>
<tr>
<td>lat</td>
<td>Network congestion latency</td>
</tr>
<tr>
<td>msgSize</td>
<td>Message size</td>
</tr>
<tr>
<td>idLength</td>
<td>Key length (in bits)</td>
</tr>
<tr>
<td>tss</td>
<td>Server message packing overhead</td>
</tr>
<tr>
<td>tso</td>
<td>Server message unpacking overhead</td>
</tr>
<tr>
<td>tcs</td>
<td>Client message packing overhead</td>
</tr>
<tr>
<td>tco</td>
<td>Client message unpacking overhead</td>
</tr>
<tr>
<td>tp</td>
<td>Time to process a request locally</td>
</tr>
<tr>
<td>numReqPerClient</td>
<td>Number of request per client</td>
</tr>
<tr>
<td>numClientPerServer</td>
<td>Number of clients per server</td>
</tr>
<tr>
<td>FailureRate</td>
<td>Failure frequency</td>
</tr>
<tr>
<td>numReplica</td>
<td>Number of replicas</td>
</tr>
<tr>
<td>R</td>
<td>Number of successful Get responses</td>
</tr>
<tr>
<td>W</td>
<td>Number of successful Put responses</td>
</tr>
<tr>
<td>numTry</td>
<td>Number of retries before a client talks to replica</td>
</tr>
</tbody>
</table>

Table 2: Parameter names and descriptions
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 lightweight 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, \(num_{ReqPerClient}\) and \(num_{ClientPerServ}\), from Table 3. For \(d_c\) and \(d_{load}\), 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 \(<\text{type}, \text{key}, \text{value}>\) are generated with a uniform random distribution (URD) (50% Getts 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 (MA-TRIX), 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 BlueGene/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 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 \(d_c\) 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 convincing 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 \(d_c\) vs \(d_{load}\)

In the section, we compare two of the distributed architectures (\(d_c\), and \(d_{load}\)) with a very basic scenario (no replication, no failures or consistency model). The synthetic workload is used to investigate the tradeoffs...
between these service architectures at increasingly large scales.

The performance comparison between \( dfc \) and \( d_{chord} \) is shown in Figure 5. Doubling client counts indicates doubling server counts because each server is responsible for 1K nodes. With \( dfc \), we observe that the efficiency has a fairly constant value (67%) at extreme scales, meaning that \( dfc \) 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 \( dfc \) and \( \log_2 N \) hops for \( d_{chord} \). We show the per client throughput speedup of \( dfc \) 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 \( dfc \) is about 2 times 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.

![Figure 6: Service architecture replication overhead](image)

4.3 Replication overhead

This section investigates the replication overhead associated with \( dfc \) 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 \( dfc \) and \( d_{chord} \), we see that \( dfc \) 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 \( dfc \), the relatively low routing overhead results in a larger impact on efficiency. In \( dfc \), the first added replica adds over 20% overhead (67% to 46%) and decreases the efficiency by 29.9% (20% to 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.

![Figure 5: \( dfc \) and \( d_{chord} \) performance comparison](image)

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 \( dfc \) network, whereas in the \( d_{chord} \) network, it is sent to \( \log_2 N \) nodes resulting \((\log_2 N)^2 \) 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 \( dfc \) 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 \( dfc \), we have at most 10M client forwarding messages and failure events only require 5×1K messages (small compared to 10M), while for \( d_{chord} \) we have \( 1M \times \log(1K) = 10M \) forwarding message, and \( 5(\log(1K))^2 = 500 \).
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 logarithmically, 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](image)

### 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 may 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 \( d_{chord} \) 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 \( d_{fc} \), 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](image)

### 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 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 real traces of three HPC system services: job launch using SLURM, monitoring by Linux Syslog, and I/O forwarding using FusionFS [33] distributed file system. We also implemented a distributed job launch
Table 4: Number of messages for $dfc$, $d_{chord}$ with and without failure and replica (C&R) (in thousands)

<table>
<thead>
<tr>
<th># Clients</th>
<th>$dfc$</th>
<th>$d_{chord}$</th>
<th>$dfc$ (C&amp;R)</th>
<th>$d_{chord}$ (C&amp;R)</th>
<th>$dfc$</th>
<th>$d_{chord}$ (C&amp;R)</th>
<th>$dfc$</th>
<th>$d_{chord}$ (C&amp;R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>414.46</td>
<td>184.956</td>
<td>312.426</td>
<td>246.153</td>
<td>0.033</td>
<td>4.549</td>
<td>217.669</td>
<td>87.106</td>
</tr>
<tr>
<td>8192</td>
<td>307.282</td>
<td>491.178</td>
<td>404.126</td>
<td>596.151</td>
<td>0.042</td>
<td>0.645</td>
<td>175.384</td>
<td>170.838</td>
</tr>
<tr>
<td>16384</td>
<td>634.826</td>
<td>1207.952</td>
<td>726.112</td>
<td>1472.551</td>
<td>0.065</td>
<td>28.615</td>
<td>336.498</td>
<td>377.072</td>
</tr>
<tr>
<td>32768</td>
<td>1290.604</td>
<td>2825.860</td>
<td>1368.706</td>
<td>3039.180</td>
<td>0.114</td>
<td>0.712</td>
<td>665.388</td>
<td>661.926</td>
</tr>
<tr>
<td>65536</td>
<td>2601.066</td>
<td>6409.968</td>
<td>2698.502</td>
<td>6637.741</td>
<td>0.210</td>
<td>0.590</td>
<td>1322.836</td>
<td>1315.460</td>
</tr>
<tr>
<td>131072</td>
<td>5222.814</td>
<td>14256.736</td>
<td>5319.564</td>
<td>14542.027</td>
<td>0.402</td>
<td>0.888</td>
<td>2632.354</td>
<td>2627.996</td>
</tr>
<tr>
<td>262144</td>
<td>10465.436</td>
<td>31279.104</td>
<td>10564.664</td>
<td>37702.242</td>
<td>0.786</td>
<td>1.096</td>
<td>5251.380</td>
<td>5247.638</td>
</tr>
<tr>
<td>524288</td>
<td>20950.976</td>
<td>67947.732</td>
<td>21030.058</td>
<td>6637.741</td>
<td>1.554</td>
<td>1.127</td>
<td>10492.476</td>
<td>10485.882</td>
</tr>
<tr>
<td>1048576</td>
<td>41922.386</td>
<td>147105.826</td>
<td>42008.038</td>
<td>42008.038</td>
<td>3.090</td>
<td>1.274</td>
<td>20985.110</td>
<td>20979.146</td>
</tr>
</tbody>
</table>

Table 5: No. of messages of strong consistency (sc), and eventual consistency (ec) (in thousands)

<table>
<thead>
<tr>
<th># Clients</th>
<th>$sc$</th>
<th>$ec$</th>
<th>$sc$</th>
<th>$ec$</th>
<th>$sc$</th>
<th>$ec$</th>
<th>$sc$</th>
<th>$ec$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>312.426</td>
<td>246.153</td>
<td>141.468</td>
<td>211.356</td>
<td>0.033</td>
<td>4.549</td>
<td>0.030</td>
<td>0.355</td>
</tr>
<tr>
<td>8192</td>
<td>404.126</td>
<td>596.151</td>
<td>391.926</td>
<td>682.689</td>
<td>0.042</td>
<td>0.445</td>
<td>0.054</td>
<td>0.588</td>
</tr>
<tr>
<td>16384</td>
<td>726.112</td>
<td>1472.551</td>
<td>733.134</td>
<td>1466.287</td>
<td>0.066</td>
<td>28.615</td>
<td>23.705</td>
<td>336.498</td>
</tr>
<tr>
<td>32768</td>
<td>1386.706</td>
<td>3039.180</td>
<td>1356.776</td>
<td>3076.448</td>
<td>0.114</td>
<td>0.712</td>
<td>0.15</td>
<td>0.827</td>
</tr>
<tr>
<td>65536</td>
<td>2601.066</td>
<td>6409.968</td>
<td>2698.502</td>
<td>6637.741</td>
<td>0.210</td>
<td>0.590</td>
<td>0.210</td>
<td>0.768</td>
</tr>
<tr>
<td>131072</td>
<td>5222.814</td>
<td>14256.736</td>
<td>5319.564</td>
<td>14542.027</td>
<td>0.402</td>
<td>0.888</td>
<td>0.534</td>
<td>1.093</td>
</tr>
<tr>
<td>262144</td>
<td>10465.436</td>
<td>31279.104</td>
<td>10564.664</td>
<td>37702.242</td>
<td>0.786</td>
<td>1.096</td>
<td>1.096</td>
<td>2.092</td>
</tr>
<tr>
<td>524288</td>
<td>20950.976</td>
<td>67947.732</td>
<td>21030.058</td>
<td>6637.741</td>
<td>1.554</td>
<td>1.127</td>
<td>1.398</td>
<td>2.092</td>
</tr>
<tr>
<td>1048576</td>
<td>41922.386</td>
<td>147105.826</td>
<td>42008.038</td>
<td>42008.038</td>
<td>3.090</td>
<td>1.274</td>
<td>1.127</td>
<td>1.973</td>
</tr>
</tbody>
</table>

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

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 result messages 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 workload 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 7 shows the efficiency of these workloads with both strong and eventual consistency for $dfc$ 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 al-
most 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 slurmctrd. The controllers are fully-connected, where each controller is aware of all of the other controllers (\( d_{tc} \)). 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 length 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 11: \( d_{tc} \) and \( d_{hard} \) with different workloads](image1)

![Figure 12: Comparison of SLURM and our distributed job launch prototype with various workloads](image2)
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 queueing services, 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 [24]. However, none of the investigations are focused on HPC, or combine replication, failures and consistency. In this survey [20], the authors have investigated 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 [32], 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 four 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<sub>conn</sub>) 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<sub>pool</sub>) 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<sub>pool</sub> 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<sub>single</sub>, C<sub>core</sub>, and d<sub>pool</sub> 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

dl.acm.org/citation.cfm?id=1012889.1012894.
Computer Society. DOI: 10.1109/3PGCIC.2010.62. Available from http://dx.doi.org/10.1109/3PGCIC.2010.62.
//simpysourceforge.net/SimPyDocs/index.html.
dl.acm.org/citation.cfm?id=1416222.1416299.