Untangling Cluster Management with Helix

Kishore Gopalakrishna, Shi Lu, Adam Silberstein,
Ramesh Subramonian, Kapil Surlaker, Zhen Zhang

LinkedIn
Mountain View, CA, USA
{kgopalakrishna,slu,asilberstein,rsubramonian,ksurlaker,zzhang}@linkedin.com

Distributed data systems systems are used in a variety of settings like online serving, offline analytics, data transport, and search, among other use cases. They let organizations scale out their workloads using cost-effective commodity hardware, while retaining key properties like fault tolerance and scalability. At LinkedIn we have built a number of such systems. A key pattern we observe is that even though they may serve different purposes, they tend to have a lot of common functionality, and tend to use common building blocks in their architectures. One such building block that is just beginning to receive attention is cluster management, which addresses the complexity of handling a moving system with many servers. This includes issues like bootstrapping, data placement constraints, load balancing, etc. All of this shared complexity, which we see in all of our systems, motivates us to build a cluster management framework, Helix, to solve these problems once in a general way.

Helix lets systems declare how they want their cluster managed, primarily by defining a state model that enumerates its components' possible states, transitions between those states, and constraints that dictate the system's valid settings. Helix does the heavy lifting of ensuring the system satisfies that state model in the distributed setting, while also meeting the system's goals on load balancing and throttling state changes. We detail several Helix-managed production distributed systems at LinkedIn and how Helix has helped them avoid building custom management components. We describe the Helix design and implementation and present an experimental study that demonstrates its performance and functionality.

Categories and Subject Descriptors: H.3.4 [Systems and Software]: Distributed systems
General Terms: Design, Algorithms

1. INTRODUCTION

The number and variety of distributed data systems (DDSs) have grown dramatically in recent years and are key infrastructure pieces in a variety of industrial and academic settings. These systems cover a number of use cases including online serving, offline analytics, search, and messaging. The appeal of such systems is clear: they let developers solve large-scale data management problems by letting them focus on business logic, rather than hard distributed systems problems like fault tolerance, scalability, elasticity, etc. And because these systems deploy on commodity hardware, they let organizations avoid purchasing and maintaining specialized hardware.

At LinkedIn, as at other web companies, the pieces of our infrastructure stack follow the above model [14]. Our stack includes offline data storage, data transport systems Databus and Kafka, front end data serving systems Voldemort and Espresso, and front end search systems. We see the same model repeated in a variety of other well-known systems like Hadoop [2], HBase [4], Cassandra [1], MongoDB [7] and Hedwig [6]. These systems serve a variety of purposes and each provides unique features and tradeoffs, yet they share a lot of functionality. As evidence of shared functionality, we observe systems frequently reusing the same off-the-shelf components. A primary example is at the storage layer of DDSs. PNUTS [11], Espresso [14], and others use MySQL as an underlying storage engine, different Google systems build on GFS [10, 15], and HBase [4] builds on HDFS. These storage engines are robust building blocks used here for lowest-level storage, while other key features like partitioning, replication, etc. are built at higher layers.

The storage layer, however, is by no means the only component amenable to sharing. What has been missing from LinkedIn's and other DDS stacks is shared cluster management, or an operating system for the
cloud [16]. DDSs have a number of moving parts, often carrying state. Much of a system's complexity is devoted toward ensuring these components are all alive and running effectively, bootstrapping new components to replace failed ones, etc.

Cluster Management The term *cluster management* is a broad one. We define it as the set of common tasks required to run and maintain a DDS. These tasks are:

- *Resource management*: The resource (database, index, etc.) the DDS provides must be divided among different nodes in the cluster.
- *Fault tolerance*: The DDS must continue to function amid node failures, including not losing data and maintaining read and write availability.
- *Elasticity*: As workloads grow, clusters must grow to meet increased demand by adding more nodes. The

SOCC'12, October 14-17, 2012, San Jose, CA USA

DDS's resources must be redistributed appropriately across the new nodes.

- *Monitoring*: The cluster must be monitored, both for component failures that affect fault tolerance, and for various health metrics such as load imbalance and SLA misses that affect performance. Monitoring requires followup action, such as re-replicating lost data or re-balancing data across nodes.

We more formally define DDS cluster management tasks in Section 2.2.

Despite so much common ground, we do not see a large or mature body of work on making cluster management a generic building block, certainly compared to what we observe at the storage layer. That said, there is a burgeoning body of work in this area (as seen in YARN [3] and other systems discussed in Section 6). There are a number of possible reasons, but we believe the key reason is that the separation between DDS-specific and generic functionality is not nearly so obvious in cluster management as it is at the storage layer. In the storage layer, an engine like MySQL is clearly at the bottom of the DDS stack. In cluster management, the responsibility is much more symbiotic: DDSs must communicate their correct and desired behavior, and current status to the cluster manager, while the manager must monitor the DDS and take action on it.

While defining and building a generic cluster management system is a daunting challenge, the benefits are immense for a company like LinkedIn. We are building out many DDSs. If each of these can rely on a single cluster manager, we gain a lot of benefit from simpler and faster development time, as well as operational simplicity.

Helix This paper presents Helix, a generic cluster management system we have developed at LinkedIn. Helix drastically simplifies cluster management by distilling a common set of concepts for describing DDSs and then exposing to DDSs pluggable interfaces for declaring their correct behavior using these concepts.

Key among the pluggable interfaces is an augmented finite state machine, which the DDS uses to encode its valid settings, *i.e.*, its possible states and transitions, and constraints over these. The DDS may also specify, using an optimization module, goals for how its resources are best distributed over a cluster, *i.e.*, how to place resources and how to throttle cluster transitions. Helix uses these to guide the actions it executes on the various components of the DDS. This includes monitoring system state for correctness and performance, and executing system transitions as needed. Helix automatically ensures the DDS is always valid, both in steady state and while executing transitions, and does its best to respect the DDS's optimization goals. This approach lets DDS designers focus on modeling system behavior, rather than the complexity of choreographing that behavior.

In this paper we make the following contributions:

- A model for generic cluster management, including both defining the common components of a DDS and the functionality a cluster manager must provide (Section 2.2).
- The design and implementation of Helix. This includes its pluggable interfaces and examples of how we use it at LinkedIn to support a diverse set of production systems. We initially targeted a diverse set of systems: Espresso, a serving store; Databus, a change capture and transport system, and a Search-as-a-service system. (Sections 3 and 4).
- An experimental study of Helix. We show that Helix is highly responsive to key events like server failure. We also show how through generic cluster management we make it possible to debug DDS correctness while remaining agnostic to the DDS's actual purpose. (Section 5).

We review related work in Section 6. We conclude and handle future work in Section 7.

2. MOTIVATION

At LinkedIn we have built and deployed a number of DDSs. These DDSs have many common characteristics in how they are deployed, monitored, expanded, etc. and differences in specific optimization goals, constraints, etc. on how those commonalities must happen. The goal of building a general cluster manager is very appealing for LinkedIn and the community. Providing a common way for a system to declare its own model, without worrying about how to force the system to conform to that model, makes creating that model easy and uniform. Moreover, it simplifies operational support, since there is only one cluster manager executing cluster changes across different systems, and we can concentrate on that manager's correctness.

Our goal in this work is to formally identify the common characteristics of DDSs and then build a cluster manager that both handles those requirements for the DDS, while exposing interfaces so the DDS can declaratively describe their own system-specific model of behavior. The cluster manager itself does the heavy lifting on the DDS.

2.1 Use Cases

Before delving into cluster manager requirements, it helps to review a sample of DDSs that we target. We describe three LinkedIn systems. We have purposefully chosen them for their diversity, to ensure the commonalities we draw from them are truly found in most or all DDSs. These use cases are good as examples, but are also the some of the earliest use cases at LinkedIn that motivated our cluster management work.

Espresso Espresso is a distributed, timeline consistent, scalable, document store that supports local secondary indexing and local transactions. Espresso runs on a number of *storage node* servers that store and index data and answer queries. Espresso databases are horizontally partitioned into a number of partitions, with each partition having a certain number of replicas distributed across the storage nodes.

Espresso designates one replica of each partition as *master* and the rest as *slaves*; only one master may exist for each partition at any time. Espresso follows a timeline consistent where only the master of a partition can accept writes to its records, and all slaves receive and apply the same writes through a replication stream. For load balancing, both master and slave partitions are assigned evenly across all storage nodes. For fault tolerance, it adds the constraint that no two replicas of the same partition may be co-located.

To maintain high availability, Espresso must ensure every partition has an assigned master in the cluster. When a storage node fails, all partitions mastered on that node must have their mastership transferred to other nodes and, in particular, nodes hosting slaves of those same partitions.

When this happens, the load should be as evenly distributed as possible.

Espresso is elastic; we add storage nodes as the number and size of databases, and the request rate against them, increases. When nodes are added we migrate partitions from existing nodes to new ones. The migration must maintain balanced distribution in the cluster but also minimize unnecessary movements. Rebalancing partition in Espresso requires copying and transferring significant amounts of data, and minimizing this expense is crucial. Even when minimized, we must throttle rebalancing so existing nodes are not overwhelmed.

Skewed workloads might cause a partition, and the storage node hosting it, to become a hot spot. Espresso must execute partition moves to alleviate any hot spots.

Search-as-a-service LinkedIn's Search-as-a-service lets internal customers define custom indexes on a chosen dataset and then makes those indexes searchable via a service API. The index service runs on a cluster of machines. The index is broken into partitions and each partition has a configured number of replicas. Each cluster server runs an instance of the Sensei [8] system (an online index store) and hosts index partitions. Each new indexing service gets assigned to a set of servers, and the partition replicas must be evenly distributed across those servers.

Unlike Espresso, search indexes are not directly written by external applications, but instead subscribe to external data streams such as Databus and Kafka [14] and take their writes from them. Therefore, there are no master-slave dynamics among the replicas of a partition; all replicas are simply *offline* or *online*.

When indexes are bootstrapped, the search service uses snapshots of the data source (rather than the streams) to create new index partitions. Bootstrapping is expensive and the system limits the number of partitions that may concurrently bootstrap.

Databus Databus is a *change data capture* (CDC) system that provides a common pipeline for transporting events from LinkedIn primary databases to caches within various applications.

Databus deploys a cluster of relays that pull the change log from multiple databases and let consumers subscribe to the change log stream. Each Databus relay connects to one or more database servers and hosts a certain subset of databases (and partitions) from those database servers. Databus has the same concerns as Espresso and Search-as-a-service for assigning databases and partitions to relays.

Databus consumers have a cluster management problem as well. For a large partitioned database (e.g. Espresso), the change log is consumed by a bank of consumers. Each databus partition is assigned to a consumer such that partitions are evenly distributed across consumers and each partition is assigned to exactly one consumer at a time. The set of consumers may grow over time, and consumers may leave the group due to planned or unplanned outages. In these cases, partitions must be reasssigned, while maintaining balance and the single consumer-per-partition invariant.

2.2 Requirements

The above systems tackle very different use cases. As we discuss how they partition their workloads and balance them across servers, however, it is easy to see they have a number of common requirements, which we explicitly list here.

- **Assignment of logical resources to physical servers** Our use cases all involve taking a system's set of logical resources and mapping them to physical servers. The logical entities can be database partitions as in Espresso, or a consumer as in the Databus consumption case. Note a logical entity may or may not have state associated with it, and a cluster manager must be aware of any cost associated with this state (e.g. movement cost).
- **Fault detection and resource reassignment** All of our use case systems must handle cluster member failures by first detecting such failures, and second rereplicating and reassigning resources across the surviving members, all while satisfying the system's invariants and load balancing goals. For example, Espresso mandates a single master per partition, while Databus consumption mandates a consumer must exist for every database partition. When a server fails, the masters or consumers on that server must be reassigned.
- **Elasticity** Similar to failure detection and response requirement, systems must be able to incorporate new cluster physical entities by redistributing logical resources to those entities. For example, Espresso moves partitions to new storage nodes, and Databus moves database partitions to new consumers.
- **Monitoring** Our use cases require we monitor systems to detect load imbalance, either because of skewed load against a system's logical partitions (*e.g.*, an Espresso hot spot), or because a physical server become degraded and cannot handle its expected load (*e.g.*, via disk failure). We must detect these conditions, *e.g.*, by monitoring throughput or latency, and then invoke cluster transitions to respond.

Reflecting back on these requirements we observe a few key trends. They all involve encoding a system's optimal and minimally acceptable state, and having the ability to respond to changes in the system to maintain the desired state. In the subsequent sections we show how we incorporate these requirements into Helix.

3. DESIGN

This section discusses the key aspects of Helix's design by which it meets the requirements introduced in Section 2.2. Our framework layers system-specific behavior on top of generic cluster management. Helix handles the common management tasks while allowing systems to easily define and plug in system-specific logic.

In order to discuss distributed data systems in a general way we introduce some basic terminology:

3.1 DDS Terminology

- *Node*: A single machine.
- *Cluster*: A collection of nodes, usually within a single data center, that operate collectively and constitute the DDS.
- *Resource*: A logical entity defined by and whose purpose is specific to the DDS. Examples are a database, search index, or topic/queue in a pub-sub system.
- *Partition*: Resources are often too large or must support too high a request rate to maintain them in their entirety, but instead are broken into pieces. A partition is a subset

of the resource. The manner in which the resource is broken is system-specific; one common approach for a database is to horizontally partition it and assign records to partitions by hashing on their keys.

- *Replica*: For reliability and performance, DDSs usually maintain multiple copies of each partition, stored on different nodes. Copies of the same partition are known as replicas.
- *State*: The status of a partition replica in a DDS. A finite state machine defines all possible states of the system and the transitions between them. We also consider a *partitions's* state to be the set of states of all of its replicas. For example, some DDSs allow for one replica to be a *master*, which accepts reads and writes, or a *slave* which accepts only reads.
- *Transition*: A DDS-defined action specified in a finite state machine that lets a replica move from one state to another.

This list gives us a generic set of concepts that govern most or all DDSs; all of the DDSs we have at LinkedIn follow this model. From the definitions, however, it is clear these concepts are quite different across DDSs. For example, a partition can be as diverse as a chunk of a database versus a set of pub/sub consumers. The goals for how these partitions should be distributed across nodes may be different. As a second example, some systems simply require a minimum number of healthy replicas, while others have more nuanced requirements, like master vs. slave.

We next introduce how Helix lets DDSs define their specific cluster manager requirements, and how Helix supports the requirements of any DDS without custom code.

3.2 Declarative System Behavior

The two key aspects of Helix that provide our desired "plug-and-play" capability are (1) an *Augmented Finite State Machine (AFSM)* that lets a system define the states and transitions of its replicas and constraints on its valid behavior for each partition; and (2) an optimization module through which systems provide optimization goals that impact performance, but not correctness. We now discuss these in sequence.

3.2.1 AFSM

We reason about DDS correctness at the partition level. A finite state machine has sufficient expressiveness for a system to describe, at a partition granularity, all its valid states and all legal state transitions. We have augmented it to also express constraints on both states and transitions. Declaring that a system must have one exactly one master replica per partition is an example of a state constraint, while limiting the number of replicas that may concurrently bootstrap is an example of a transition constraint.

We introduce formal language for defining an AFSM. A DDS contains one or more resource. Each resource has a set of partitions P, each partition $p_i \in P$ has a replica set $R(p_i)$, $R_\sigma(p_i)$ is the subset of $R(p_i)$ in state σ, and $R_\tau(p_i)$ is the subset of $R(p_i)$ undergoing transition τ. Note for ease-of-use we differentiate states and transitions; internally, Helix treats both of these as states (changing from a state to a transition is instantaneous). N is the set of all nodes, and $R(p_i, n_j)$ denotes the subset of $R(p_i)$ replicas located on node n_j. We assume all nodes are identical, and defer handling heterogenous node types to future work.

We now take a particular use case, Espresso as described in Section 2.2, and formally define its AFSM.

1. Replica states are $\{Master(M), Slave(S), Offline(O)\}$
2. Legal state transitions are $\{O \rightarrow S, S \rightarrow M, M \rightarrow S, S \rightarrow O\}$
3. $\forall p_i, |R_M(p_i)| \leq 1$ (Partition has at most one master.)
4. $\forall p_i, |R_S(p_i)| \geq 2$ (Partition has at least two slaves.)
5. $\forall p_i, \forall n_j, |R(p_i, n_j)| \leq 1$ (At most one replica per partition on each node.)

3.2.2 Optimization Module

While the AFSM lets DDSs declare their correct behavior at a partition-level, the optimization module lets DDSs list optimizations at a variety of granularities: partition, node, resource, cluster. These optimizations can be broken into two types: *transition* goals and *placement* goals. Note that the optimization goals are just that. Helix tries to achieve them, but not at the cost of cluster correctness, and correctness does not depend on them.

When Helix must invoke multiple replica transitions, it must often choose an ordering for those transitions, both to maintain DDS correctness during the transitions, and in case of throttling. Transition goals let the DDS tell Helix how to prioritize those transitions. We denote Π is a *transition preference list* that ranks state transitions in order from highest to lowest priority.

Helix has many choices for how to place replicas on nodes. Placement goals let the DDS tell Helix how this should be done. The DDS can use these goals, for example, to achieve load balancing. Function $\sigma(n_j)$ returns the number of replicas in state σ (across all partitions) on n_j, and $\tau(n_j)$ returns the number of replicas in transition τ on a node n_j. $\tau(C)$ returns the number of replicas in τ cluster-wide.

Espresso's optimizations are as follows:

1. $\forall p_i, |R_M(p_i)| = 1$ (Partition has one master.)
2. $\forall p_i, |R_S(p_i)| = 2$ (Partition has two slaves.)
3. $\Pi = \{S \rightarrow M\}, \{O \rightarrow S\}, \{M \rightarrow S, S \rightarrow O\}$
4. $minimize(\max_{n_j \in N} M(n_j))$
5. $minimize(\max_{n_j \in N} S(n_j))$
6. $max_{n_j \in N}(O \rightarrow S)(n_j) \leq 3$.
7. $(O \rightarrow S)(C) \leq 10$.

Goals 1 and 2 describe Espresso's preferences for numbers of masters and slaves. These are nearly identical to constraints 3 and 4 in Espresso's AFSM. Whereas Espresso can legally have 0 or 1 masters per partition, it prefers to have 1 so that the partition is available; whereas Espresso must have at least 2 slaves for reliablility, it prefers to have 2 and not more. Goal 3 tells Helix to prioritize slave-to-master transitions above all others, followed by offline-to-slave. The transitions that move replicas offline are lowest. This prioritization is reasonable for Espresso, which cannot make partitions available for writes without a having a master, and

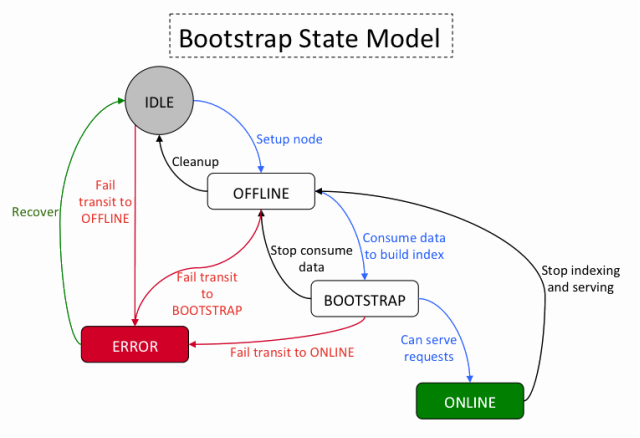

Figure 1: Search-as-a-service state machine.

has degraded fault tolerance when it lacks slaves. The order also minimizes downtime when new nodes are added, since slaves are then created on the new nodes before they are removed from existing nodes. Goals 4 and 5 encode Espresso's load balancing goals, which are to evenly distribute masters across all cluster nodes and to evenly distribute slaves across all cluster nodes. Goals 6 and 7 encode throttling goals, allowing no more than 3 concurrent offline-to-slave transitions per node and no more than 10 concurrent offline-to-slave transitions per cluster.

Figure 1 shows the detailed state machine for Search-as-a-service, another Helix-managed DDS. As with Espresso, the state model and constraints are configured in Helix. Search-as-a-service makes use of state machines to configure different resources differently based on the workflow. They also have optimization goal similar to Espresso to throttle the maximum number of offline→bootstrap transitions.

3.3 Helix Execution

Thus far we have described how DDSs declare their behavior within Helix. We now describe how Helix invokes that behavior on behalf of the DDS at run-time. Helix execution is a matter of continually monitoring DDS state and, as necessary, ordering transitions on the DDS. There are a variety of changes that can occur within a system, both planned and unplanned: bootstrapping the DDS, adding or losing nodes, adding or deleting resources, adding or deleting partitions, among others (we discuss detecting the unplanned changes in Section 4).

The most crucial feature of the Helix transition algorithm is that it is identical across of these changes and across all DDSs! Else, we face a great deal of complexity trying to manage so many combinations of changes and DDSs, and lose much of Helix's generality. The execution algorithm appears in Algorithm 3.3 and we now step through it.

In lines 2-3 we initialize two transition sets: validTrans and inflightTrans, whose purposes we describe shortly. The first step, in lines 5-6, is on a partition-by-partition basis, to read the DDS's current state and compute a target state, where target state is a distribution of a partition's replicas over cluster nodes that respects the DDS's constraints and optimization goals. Most of the time, the current state will match the target state, and there is nothing to do; they disagree only when the cluster changes (e.g. nodes lost, partitions added, etc.).

Algorithm 1 Helix execution algorithm

```
1:  repeat
2:    validTrans = ∅
3:    inflightTrans = ∅
4:    for each partition p_i do
5:      Read currentState
6:      Compute targetState
7:      Read p_i pendingTrans
8:      inflightTrans.addAll(pendingTrans)
9:      requiredTrans = computeTrans(currentState, targetState, pendingTrans)
10:     validTrans.addAll(getValidTransSet(requiredTrans))
11:   end for
12:   newTrans = throttleTrans(inflightTrans, validTrans)
13: until newTrans == ∅
```

By default we use the RUSH [13] algorithm to produce the target state, though with enhancements to ensure we meet state machine constraints and to ensure we meet our optimization goals. Default RUSH relies on random hashing of a huge number of partitions to meet load balancing goals. Since DDSs can have as few as 10s of partitions per node, to avoid load skew and so better meet optimization goals, we additionally assign each node a budget that limits the number of partitions it may host. Helix makes it easy to plug in other algorithms; we discuss this more in Section 3.3.1.

Given a partition's target state, line 7-9 reads all pending transitions for the partition and then computes necessary additional replica transitions. Given current state and the state model, producing the set of all necessary transitions is straightforward and we omit the details.

The next part of the algorithm computes a set of valid transitions for the partition, taking into account the already pending transitions and those that still must occur. Our key insight is that when we trigger multiple transitions in parallel, these transitions are not instantaneous and we do not know in what order they will complete. if we simply trigger any arbitrary set of transitions (including all of them), the partition could easily become invalid (*e.g.*, having no master) while they are underway. In contrast, if we issue transitions one-at-a-time, we know the exact states of the partition, but then progress through the transitions very slowly.

Our solution is to produce valid *transition sets*; for a given partition, a transition set is valid if for every possible ordering of its transitions, the partition remains valid. Line 10 calls a method getValidTransSet that initializes a transition set to include all currently pending transitions for a partition and greedily adds other required transitions, as long as the transition set remains valid, until no more transitions can be added. It considers the transitions in priority order, according to the optimization goals given by the DDS.

Note we compute transition sets partition-by-partition. Since the AFSM is per-partition, we can safely execute a valid transition set per each partition without making any partition invalid. Thus, we have two potential dimensions for parallelism: by building transition sets per partition, and across partitions.

By line 12 we have a set of valid transitions to run across all partitions. We do not simply execute them all, but instead now take into account the DDS's throttling optimization goals. The throttleTransitions method takes the set of all inflight transitions (from prior rounds) and then selects as many additional transitions as possible, without violating constraints. Notice that any non-scheduled valid transitions are essentially forgotten and are considered anew in later rounds.

Finally, not shown in the algorithm is what happens when transitions complete. We are notified by callback and remove those transitions from their partitions lists of pending transitions.

The execution algorithm has two greedy key steps, to produce valid transitions and to choose transitions for execution, rather deriving both in a single optimization. While we could combine the two to produce a provably optimal solution, for the time being, we find the current approach produces a high level of parallelism, with highest priority transitions scheduled early.

3.3.1 Application-Customized Execution

By default Helix places replicas using modified RUSH, as described in Algorithm 3.3. We refer to this as *auto* placement. While this approach makes things very simple for some applications, it may not be powerful enough for all applications, such as those that want to customize placement of a single resource's partitions or even control placement of multiple resources' partitions. Helix offers a pluggable interface that lets DDSs customize placement. *Semi-auto* placement lets DDSs decide replica placement, while Helix still chooses the states of those replicas. *Custom* placement lets DDSs fully customize placement and state.

There are a number of reasons an application may want more explicit control over Helix's decisions. They might have application-specific requirements that cannot currently be expressed in Helix. For example, HBase can use *semi-auto* placement to co-locate its regions servers with the HDFS blocks containing those regions' data.

Finally, applications may also simply want explicit control over all placement, because they are very specialized, or because they are wary of handing over all control to Helix. We do not want to turn away such applications, and so permit customized placement in Helix, while allowing them to benefit from all of Helix's other features, rather than force such applications to build their own cluster management from scratch.

Helix offers a pluggable interface that lets DDSs add more customized placement. *Semi-auto* placement lets DDSs decide replica placement, while Helix still chooses the states of those replicas. This is useful for a system like HBase, whose region servers are ideally co-located with the HDFS blocks containing those regions' data. Finally, *custom* placement lets DDSs fully customize placement and state.

3.3.2 Execution example

We consider a case when a node is added to already existing cluster to illustrate the execution sequence in Helix. Suppose we start with an Espresso cluster with 3 nodes $n_0 \dots n_2$, 12 partitions $p_0 \dots p_{12}$, and a replication level of 3; there are 12 total replicas per node. We then add a node n_3. Intuitively, we want to rebalance the cluster such that every node is left with 9 replicas. In Helix execution we first compute a target state for each p_i. For 9 of 12 partitions the target state differs from the current state. In particular, 3 partitions get mastered on the new node and 6 partitions get slaved on the new node. We compute the required transitions for each partition. Suppose p_{10} will have its master replica moved from n_1 to n_3. n_1 must execute $t1 = (M \rightarrow S)$ for p_{10}, while n_3 must execute $t2 = (O \rightarrow S)$ and $t3 = (S \rightarrow M)$. We cannot, however, execute these transitions all in parallel since some orderings make the system invalid. In order to avoid p_{10} being mastered twice, $t3$ must execute only after $t1$ completes. Helix automatically enforces this since it issues a transition if and only if it does not violate any of the state constraints. There are two possibile valid groupings to reach this target state. One is to execute $t1$ and $t2$ in parallel, and then execute $t3$, while the other is to execute $t2$, $t1$, and $t3$ sequentially. Helix chooses the first approach.

Across all partitions, we produce a set of 18 valid transitions that we can execute in parallel; however, since this Espresso cluster prohibits more than 10 from running in parallel, we produce a transition set of size 10, and save all remaining transitions (delayed due to validity or throttling) to later rounds. Its important to note that Helix does not wait for all 10 transitions to complete before issuing the remaining transitions. Instead, as soon as the first transition is completed, it reruns the execution algorithm and tries to issue additional transitions.

3.4 Monitoring and Alerts

Helix provides functionality for monitoring cluster health, both for the sake of alerting human operators and to inform Helix-directed transitions. For example, operators and Helix may be want to monitor request throughputs and latencies, and at a number of granularities: per-server, per-partition, per-customer, per-database, etc. These metrics help systems detect if they are meeting or missing latency SLAs, detect imbalanced load among servers (and trigger corrective action), and even detect failing components.

Ultimately, the DDS, rather than Helix, should choose what to montitor. Helix provides a generic framework for monitoring and alerting. The DDS submits statistics and alert *expressions* they want monitored to Helix and then at run-time emit statistics to Helix that match those expressions. Helix is oblivious to the semantic meaning of the statistics, yet receives, stores, and aggregates them, and fires any alerts that trigger. In this paper we do not fully specify the framework, but instead give a few examples of its expressiveness and how it is used.

Helix stats follow the format `(aggregateType)(statName)`. For example, a system might create

```
window(5)(db*.partition*.reqCount)
```

We use *aggregation types* to specify how stats should be maintained over time. In this case, the aggregation type is a window of the last 5 reported values. This stat uses wildcards to tell Helix to track request count for every partition of every db. We also provide aggregate types *accumulate*, which sums all values reported over time and *decay*, which maintains a decaying sum.

Helix can maintain stats for their own sake; they are also a building block for alerts. An example alert is:

```
decay(0.5)(dbFoo.partition10.reqCount) > 100
```

Helix fires an alert if the time decaying sum of request counts on dbFoo's partition 10 exceeds 100. Similarly, we can instatiate multiple alerts at once using wildcards:

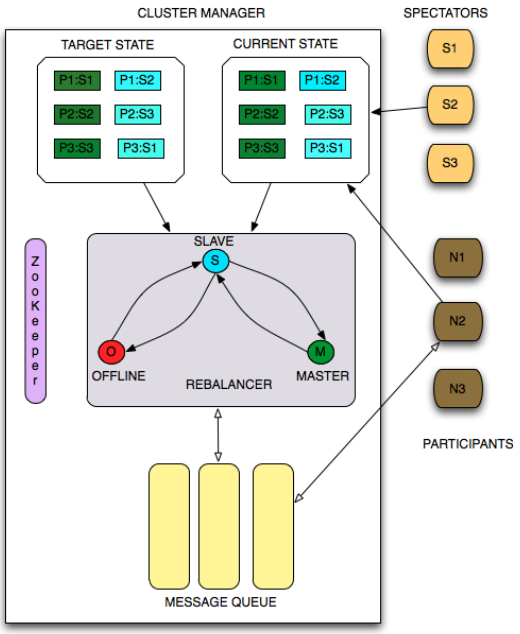

Figure 2: Helix Architecture

`decay(0.5)(dbFoo.partition*.reqCount) > 100`
The aggregator outputs a column of request counts and an alert fires for *each* value above 100.

Alerts support simple stats enumeration and aggregation using *pipeline operators*. Each operator expects tuples with 1 or more input columns and outputs tuples with 1 or more columns (valid number of input columns is operator specific). An example aggregating alert is:

```
decay(0.5)(dbFoo.partition*.failureCount,
dbFoo.partition*.reqCount)|SUMEACH|DIVIDE) > 100
```

This alert sums failure counts across partitions, sums request counts across all partitions, and divides to generate a database-wide failure rate.

The sets aggregator and operator types within Helix are themselves easy to expand. The only requirement in adding an aggregator is that pipeline operators and comparators (">" in above example) must be able to interpret its output. Pieline operators must specify their required number of input and output columns. This lets users apply arbitrary operator chains, since Helix can determine in advance whether those chains are valid. While have implemented a number of operators, the most commonly used for simple aggregations are SUMEACH, which sums each input column to produce an equal number of singleton columns, and SUM, which does row-wise summing.

4. HELIX ARCHITECTURE

Section 3 describes how Helix lets a DDS specify their model of operation and then executes against it. This section presents the Helix system architecture that implements these concepts.

4.1 Helix Roles

Helix implements three roles, and each component of a DDS takes on at least one of them. The *controller* is a pure Helix component. It hosts the state machine engine, runs the execution algorithm, and issues transitions against the DDS. It is the core component of Helix.

The DDS's nodes are *participants*, but run the Helix library. The library is responsible for invoking callbacks whenever the controller initiates a state transition on the participant. The DDS is responsible for implementing those callbacks. In this way, Helix remains oblivious to the semantic meaning of a state transition, or how to actually implement the transition. For example, in a *LeaderStandby* model, the DDS implements methods *OnBecomeLeaderFromStandby* and *OnBecomeStandbyFromLeader*.

The *spectator* role is for DDS components that need to observe system state. Spectators get notification anytime a transition occurs. A typical spectator is the router component that appears in data stores like
Espresso. The routers must know how partitions are distributed in order to direct client requests and so must be get up-to-date with any partition moves. The routers do not store any partitions themselves, however, and the controller never executes transitions against them.

The roles are at a logical level. A DDS may contain multiple instances of each role, and they can be run on separate physical components, run in different processes on the same component, or even combined into the same process.

Dividing responsibility among the components brings a number of key advantages. (A) All global state is managed by the controller, and so the DDS can focus on implementing only the local transition logic. (B) The participant need not actually know the state model the control is using to manage its DDS, as long as all the transitions required by that state model are implemented. For example, we can move a DDS from a MasterSlave model to a read-only SlaveOnly model with no changes to participants. The participants will simply never be asked to execute slave→master transitions. (C) Having a central decision maker avoids the complexity of having multiple components come to consensus on their roles.

4.2 Connecting components with Zookeeper

Given the three Helix components, we now describe their implementation and how they interact.
Zookeeper [9] plays an integral role in this aspect of Helix.

The controller needs to determine the current state of the DDS and detect changes to that state, *e.g.*, node failures. We can either build this functionality into the controller directly or rely on an external component that itself persists state and notifies on changes. When the controller chooses state transitions to execute, it must reliably communicate these to participants. Once complete, the participants must communicate their status back to the controller. Again, we either build a custom communication channel between the controller and participants, or rely on an external system. The controller cannot be a single point of failure; in particular, when the controller fails, we cannot afford to lose either controller functionality or the state it manages.

Helix relies on Zookeeper to meet all of these requirements. We utilize Zookeeper's group membership and change notification to detect DDS state changes. Zookeeper is designed to maintain system state, and is itself fault tolerant.

By storing all the controller's state in Zookeeper, we make the controller itself stateless and therefore simple to replace on a failure.

We also leverage Zookeeper to construct the reliable communication channel between controller and participants. The channel is modeled as a queue in Zookeeper and the controller and participants act as producers and consumers to this queue. Producers can can send multiple messages through the queue and consumers can process the messages in parallel. This channel brings side operational benefits like the ability to cancel transitions and to send other command messages between nodes.

Figure 2 illustrates the Helix architecture and brings together the different components and how they are represented in Zookeeper. The diagram is from the controller's perspective. The AFSM is specified by each DDS, but is then itself stored in Zookeeper. We also maintain the current states of all partition replicas and target states for partition replicas in Zookeeper. Recall that any differences between these states trigger the controller to invoke state transitions. These transitions are written to the messaging queue for execution by the participants. Finally, Helix maintains a list of participants and spectators in Zookeeper as *ephemeral nodes* that heartbeat with Zookeeper. If any of the nodes die, Zookeeper notifies the controller so it can take corrective action.

4.3 DDS Integration with Helix

This section describes how a DDS deploys with Helix. The DDS must provide 3 things.

Define Cluster The DDS defines the physical cluster and its physical nodes. Helix provides an admin API, illustrated here:

```
helix-admin --addCluster EspressoCluster
helix-admin --addNode EspressoCluster <esp10:1234>
```

Define Resource, State Model and Constraints Once the physical cluster is established, the next step is to logically add the DDS, including the state model that defines it and the resources it will supply. Again, Helix provides an admin API:

```
helix-admin --addStateModel MasterSlave states=<M,S,O>
            legal_transitions=<O-S, S-M, M-S, S-O>
            constraints="count(M)<=1 count(S)<=3"
helix-admin --addResource
           clusterName=EspressoCluster
   resourceName=EspressoDB numPartitions=8
   replica=3
   stateModelName=MasterSlave
```

Given the above admin commands, Helix itself computes an initial target state for the resource:

```
{
  "id" : "EspressoDB",
  "simpleFields" : {
    "IDEAL_STATE_MODE" : "AUTO",
    "NUM_PARTITIONS" : "8",
    "REPLICAS" : "1",
    "STATE_MODEL_DEF_REF" : "MasterSlave",
  },
  "mapFields" : {
    "EspressoDB_0" : { "node0" : "MASTER",
                       "node1" : "SLAVE"  },
    "EspressoDB_1" : { "node0" : "MASTER",
                       "node1" : "SLAVE"  },
    "EspressoDB_2" : { "node0" : "SLAVE",
                       "node1" : "MASTER" },
    "EspressoDB_3" : { "node0" : "SLAVE",
                       "node1" : "MASTER" },
  }
}
```

Implement Callback Handlers The final task for a DDS is to implement logic for each state transition encoded in their state machine. We give a partial example of handler prototypes for Espresso's use of MasterSlave.

```
EspressoStateModel extends StateModel
{
 void offlineToSlave(Message task,
     NotificationContext context)
 {
  // DDS Logic for state transition
 }
 void slaveToMaster(Message task,
     NotificationContext context)
 {
  //DDS logic for state transition
 }
}
```

4.4 Scaling Helix into a Service

The risks of having a single controller as we have described so far is that it can become a bottleneck or a single source of failure (even if it is easy to replace). We now discuss our approach for distributing the controller that actually lets Helix provide cluster management as a service.

To avoid making the controller a single point of failure, we deploy multiple controllers. A cluster, however, should be managed by only one controller at a time. This itself can easily be expressed using a LeaderStandby state model with the constraint that every cluster must have exactly one controller managing it! Thus we set up multiple controllers as a participants of a *supercluster* comprised of the different clusters themselves as the resource. One of the controllers manages this supercluster. If that controller fails, another controller gets selected as the Leader for the supercluster. This guarantees that each cluster has exactly one controller.

The supercluster drives home Helix's ability to manage any DDS, in this case itself. Helix itself becomes a scalable DDS that in turns manages multiple DDSs. In practice we typically deploy a set of 3 controllers capable of managing over 50 clusters.

5. EVALUATION

This section describes our experience running Helix in production at LinkedIn and then presents experiments demonstrating Helix's functionality and production.

5.1 Helix at LinkedIn

From the beginning we built Helix for general purpose use, targeting Espresso, Search-as-a-service and Databus. As mentioned earlier, we chose these systems to ensure Helix became a truly general cluster manager and not, say, a cluster manager for distributed databases. The systems were themselves under development and in need of cluster management functionality, and so were likewise attracted to Helix. At this writing, all three of Espresso, Search-as-a-service and Databus run in production supported by Helix.

With every Helix-DDS integration we found and repaired gaps in the Helix design, while maintaining its generality. For example, Search-as-a-service needed more control than

auto replica placement offers, and so we added semi-auto and custom. We also added a *zone* concept to support grouping a set of nodes together. This is useful, for example, in rack-aware replica placement strategies.

The first Helix version did not include constraints and goals on transitions, but instead satisfied this requirement by overloading state constraints and goals. This complicated the algorithms for computing and throttling transitions. We added support for transitions to remove this complexity, as well as add support for throttling transitions at different cluster granularities.

DDS debugging and correctness checking One of the most interesting benefits of Helix falls out of its ability to manage a cluster without understanding its purpose. We have built a debugging tool capable to analyzing a DDS's correctness without understanding its purpose. Testing distributed systems is extremely difficult and laborious, and it cannot be done with simple unit tests. Generic deubugging tools that function in the distributed setting are especially valuable.

Our tool uses the "instrument, simulate, analyze" methodology. This consists of

- **Instrument** Insert probes that provide information about the state of the system over time. We use Zookeeper logs as "probes."
- **Simulate** We use robots to create cluster error, killing componets and injecting faults, both on the DDS and Helix sides.
- **Analyze** We parse the logs into a relational format, load them into a database as a set of tables and then write invariants on the DDS's constraints. Violation of the constraints signals a possible bug.

Every DDS that uses Helix automatically benefits from this tool and it has been crucial in improving performance and debugging production issues. One of the most difficult parts of debugging a DDS is understanding the sequence of events that leads to failure. Helix logging plus the debugging tool make it easy to collect system information and evaluate it. As a concrete example, the experiment on planned downtime we describe in Section 5.2.4 uses this tool to measure total downtime and partition availability during software upgrades. This lets us experiment with different upgrade policies and ensure they are performant and valid.

5.2 Experiments

Our experiments draw from a combination of production and experimental environments, using the DDS Espresso. The clusters are built from production-class commodity servers. The details of the production clusters are confidential, but we present details on our experimental cluster throughout our experiment discussion.

5.2.1 Bootstrap

Our first experiment examines Helix's performance when bootstrapping an Espresso cluster, (*i.e.*, adding the initial nodes and databases). We note that while for some DDSs bootstrapping is a rare operation, for others, like Databus consumers, it happens frequently, and fast bootstrap times are crucial.

We vary the number of allowed parallel transitions, add nodes and an Espresso database, and then measure the time taken for the cluster to reach a stable state. We repeat the experiment for 1024 partition and 2400 partition databases. Figure 3 plots the result and shows that bootstrap time decreases with greater transition parallelism, though the benefits diminish once parallelism is in the tens of transitions.

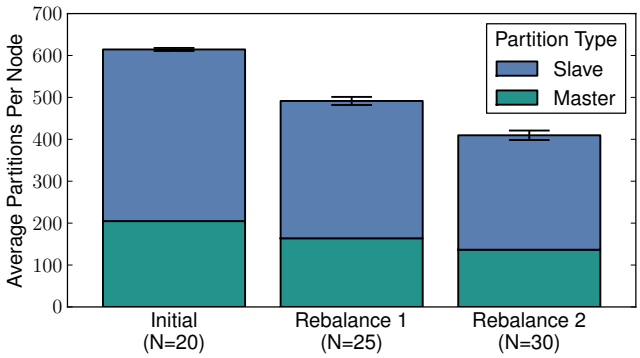

Figure 5: Elastic Cluster Expansion

The total bootstrap time is directly proportional to the number of transitions required and inversely proportional to the allowed parallelism. The number of required transitions is a function of the number of partitions, number of replicas per partition and the transitions drawn from Espresso's state machine. For example, an Espresso database with p partitions and r replicas requires pr offline→slave transitions and p slave→master transitions. Espresso sets allowed parallelism based on the number of cluster nodes and the number of concurrent transitions a node can handle. The final factor in deriving bootstrap time is the time required to execute each individual transition. These times are DDS dependent. For example, when an Espresso partition enters the slave state Espresso creates a MySQL database and all required tables for it; this takes tens of milliseconds. On bootstrap, this transition takes approximately 100 ms.

5.2.2 Failure Detection

One of Helix's key requirements is failure detection, and its performance in this area must match what a DDS might achieve for itself with a custom solution. In Espresso, losing a node means all partitions mastered on that node become unavailable for write, until Helix detects the failure and promotes slave replicas on other nodes to become masters. We run an experiment with an Espresso cluster of size 6, randomly kill a single node, and measure write unavailability. We vary the the number of Espresso partitions and the maximum cluster-wide allowed parallel transitions. Figure 4 plots the result and shows most importantly that we achieve low unavailability in the 100s of ms, and that this time decreases as we allow for more parallel transitions. It is interesting to note that earlier versions of Helix had recovery times of multiple seconds. The reason is that when so many partitions transitioned from slave to master at once, Helix actually bottlenecked trying to record these transitions in Zookeeper. We ultimately solved that bottleneck using techniques like group commit and asyncronous reads/writes; the key point is that we solved this problem for every DDS using Helix and masked internal Zookeeper details from them.

5.2.3 Elastic Cluster Expansion

Another of Helix's key requirements is its ability to incor-

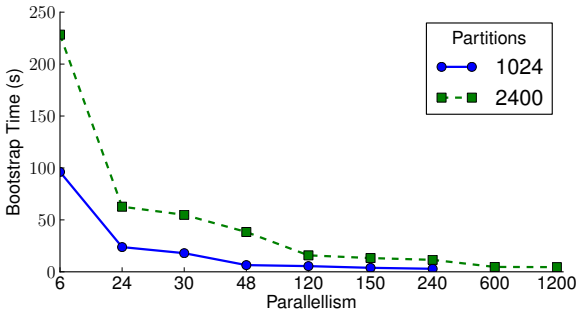

Figure 3: Espresso transition parallelism vs. bootstrap time.

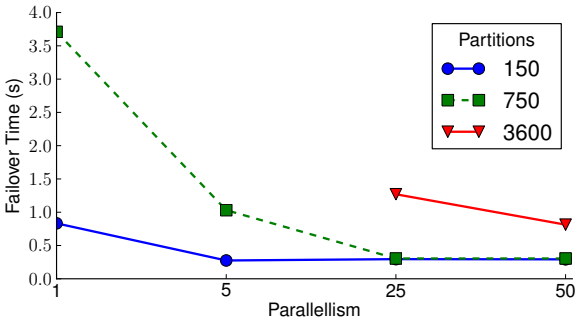

Figure 4: Espresso transition parallelism vs. failover time.

porate new nodes as clusters expand. Recall Algorithm 3.3 handles shifting partitions from existing to new nodes by computing target states. We want to ensure rebalancing transitions as few replicas as possible; in particular, in Espresso, we want to ensure that the number of replicas transitioned to either slave or master on new nodes is minimized, since these transitions typically involve the expensive step of copying state. We run an experiment that starts with a cluster with 20 nodes, 4096 partitions and 3 replicas per partition. We then add 5 new nodes, Helix recalculates the placement and we find that 19.6% of partitions are moved. The optimal percentage moved for full balance is clearly 20%; with this experiment's server and partition counts, achieving 20% movement requires moving partition fractions and so we instead achieve near balance.

Finally, we add an additional 5 nodes. After Helix recalculates the placement we see that 16% of partitions are moved, close to the optimal solution. Figure 5 shows we are minimal in the number of partitions moved to ensure even load distribution. While RUSH is very good at accomplishing this type of balancing, RUSH does not inherently support the concept of different replica states and is by default designed for handling numbers of partitions that are orders of magnitude greater than the number of nodes. As discussed in Section 3 we modified it to handle different states and possibly much smaller numbers of partitions and still achieve close to minimal movement.

5.2.4 *Planned Downtime*

One function that Helix manages for DDSs is planned downtime, which helps for a variety of administrative tasks, such as server upgrades. In Espresso bringing down a node involves moving every partition on it to the offline state and transitioning partitions on other servers to compensate. In fact, there is a direct tradeoff between total time to complete planned downtime (*e.g.*, to upgrade all nodes) versus accumulated unavailability over that time.

Since Helix provides fault tolerance, a DDS system can upgrade its software by upgrading one node at a time. The goal of this exercise is to choose a reasonable tradeoff between minimizing total upgrade time and minimizing unavailability during the upgrade. One simple approach is to upgrade one node at a time. Though this approach is common, it may leave a lot of parallelism on the table that the DDS can handle and would speed up upgrade time. Even though this is the most commonly used approach, it is not guaranteed to achieve the two state goals. The reason is if the ratio of transitions to max parallel transitions allowed

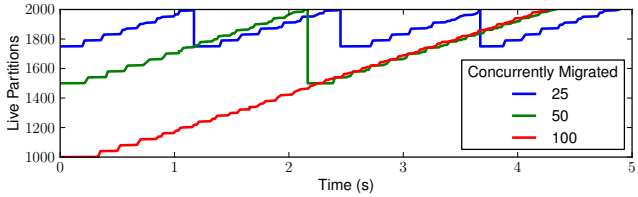

Figure 6: Percentage of partitions that are available over the course of a planned software upgrade.

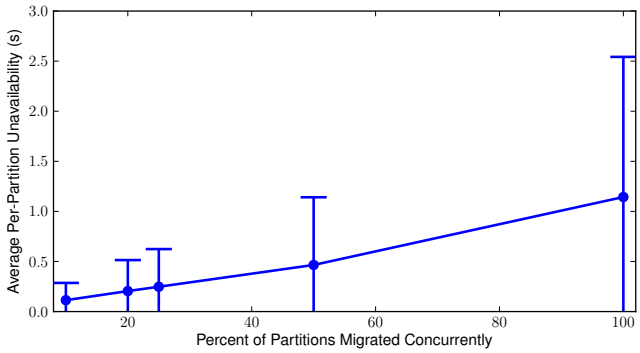

Figure 7: Average unavailability per partition during planned software upgrade.

on each node is very high, then its possible that overall availability of the system reduce. The ideal solution is to have this ratio close to 1.

In this experiment we showcase the impact of concurrent partition migration on a single node. We fix the total number of migrations at 1000 and control the batch size of partitions that may be concurrently migrated.

Figure 6 shows that by choosing a lower concurrency level, each migration is faster but the overall time taken to upgrade a node is larger.

Figure 7 shows a different perspective by plotting the average per partition unavailability. It also shows that the variance grows quite high as parallelism increases. While lower upgrade times are appealing, higher error counts during upgrades are not.

The ability to balance downtime and unavailability is important for DDSs so the upgrade process becomes smooth and predictable. Beyond what is illustrated in the scope of this experiment we also see cases where throttling transitions

is important to avoid load spikes that also affect availability. For example, bringing up too many Espresso partitions online at the same time, when they have cold caches, will cause failures. By controlling transitions at the partition granularity, Helix helps DDSs avoid availability and performance problems, and manage planned downtime duration.

5.2.5 Alerts

Recall from Section 3 that Helix provides DDSs a framework for defining monitored statistics and alerts. Here we give some examples of alerts Espresso runs in production.

Espresso monitors request latency to ensure it is meeting its SLAs or take corrective action otherwise. Specifically it registers the alert

```
decay(1)(node*.latency99th) > 100ms
```

Since the alert is wildcarded on node name, we separately monitor 99th percentile latency on each node in the Espresso cluster. The decaying sum setting of 1 means we monitor the latest reported value from each node, and ignore older values. Figure 8 shows a production graph displaying the current monitored value for a particular Espresso node. The x and y axes are wall time and requiest latency, respectively. A similar graph (not shown here) indicates whether the alert is actually firing and, when fired, emails notification to Espresso operators.

A second alert monitors the cluster-wide wide request error count:

```
decay(1)(node*.errorcount)|SUMEACH > 50ms
```

The alert is again wildcarded on node name, but now computes the sum of errors over all nodes. Figure 9 shows the production graph and, again, there is an unshown corresponding graph displaying the alert status.

6. RELATED WORK

The problems that systems dealing with cluster management attempt to solve fall in 3 main categories: (a) resource management, (b) enforcing correct system behavior in the presence of faults and other changes, and (c) monitoring.

There are generic systems that solve resource management and monitoring, but not fault-tolerance, in a generic manner. Other DDSs support fault-tolerance and enforce system behavior correctness in a way that is specific to that particular DDS.

6.1 Generic Cluster Management Systems

Systems such as YARN [3] and Mesos [5] implement resource management and some monitoring capabilities in a generic way. Applications using these systems are responsible for implementing the specific logic to react to changes in the cluster and enforce correct system behavior. We detail Mesos an example.

Mesos has 3 components: *slave daemons* that run on each cluster node, a *master daemon* that manages the slave daemons, and *applications* that run tasks on the slaves. The master daemon is responsible for allocating resources (cpu, ram, disk) across the different applications. It also provides a pluggable policy to allow for adding of new allocation modules. An application running on Mesos consists of a *scheduler* and an *executor*. The scheduler registers with the master daemon to request resources. The executor process runs on the slave nodes to run the application tasks. The master daemon determines how many resources to offer to each application. The application scheduler is responsible for deciding which offered resources to use. Mesos then launches the tasks on the corresponding slaves. Unlike Helix, Mesos does not provide a declarative way for applications to define their behavior and constraints on it and have it enforced automatically, but provides resource allocation functionality that is complementary to Helix.

6.2 Custom Cluster Management

As mentioned before, distributed systems like PNUTS [11], Hbase [4], HDFS [2], and MongoDB [7] implement cluster management in those specific systems. As an example, we examine MongoDB's cluster management.

MongoDB provides the concept of document collections which are sharded into multiple chunks and assigned to servers in a cluster. MongoDB uses range partitioning and splits chunks when chunks grow too large. When the load of any node gets too large, the cluster must be rebalanced by adding more nodes to the cluster and some chunks move to the new nodes.

Like any other distributed system, MongoDB must provide failover capability and ensure that a logical shard is always online. To do this, MongoDB assigns n servers (typically 2-3) to a shard. These servers are part of a replica set. Replica sets are a form of asynchronous master-slave replication and consist of a primary node and slave nodes that replicate from the primary. When the primary fails, one of the slaves is elected as the primary. This approach to fault tolerance is similar to the MasterSlave state model Espresso uses; Espresso gets it for free from Helix, while MongoDB had to build it.

Other distributed systems implement very similar cluster management capabilities, but each system reimplements it for itself, and not in a way that can be easily reused by other systems.

6.3 Domain Specific Distributed Systems

Hadoop/MapReduce [2, 12] provides a programming model and system for processing large data sets. It has been wildly successful, largely because it lets programmers focus on their jobs' logic, while masking the details of the distributed setting on which their jobs run. Each MR job consists of multiple map and reduce jobs and the processing slots get assigned to these jobs. Hadoop takes care of monitoring the tasks and restarting tasks in case of failures. Hadoop, then, very successfully solves the problems we outline for cluster management, but only in the MR context. Helix aims to bring this ease of programming to DDS development.

Zookeeper [9] is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. It is used by many distributed systems to help implement cluster management functionality, and we of course use it heavily as a building block in Helix. It is important to note that Zookeeper alone does not solve the cluster management problems. For example, it provides functionality to notify a cluster manager when a node has died, but does not plan and execute a response.

6.4 Distributed resource management

A number of older systems pioneered solutions in distributed resource management. Amoeba [**?**], Globus [**?**], GLUnix [**?**] and Legion [**?**] all manage large numbers of servers and present them as a single, shared resource to

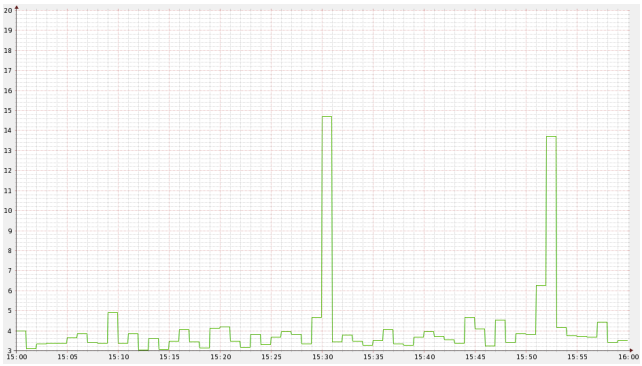

Figure 8: Alert reporting 99th percentile latency on a single Espresso node.

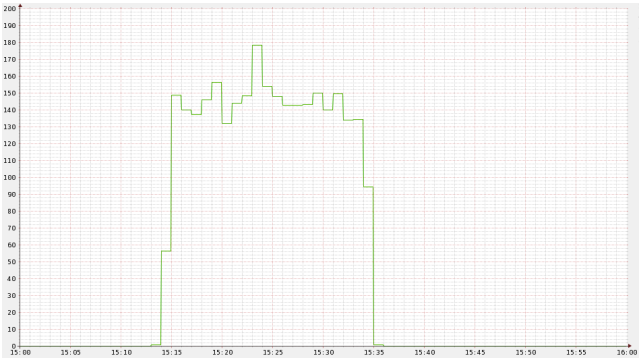

Figure 9: Alert reporting the cluster-wide Espresso error count.

users. The challenges they tackle most relevant to Helix include placing resources, scheduling jobs, monitoring jobs for failures, and reliably disseminating messages among nodes. Much of the scheduling work relates closely to how we distribute resource partitions among nodes, and how we hope to make this distribution more dynamic in the future, where Helix will move partitions aggressively in response to load imbalance. On the other hand, we address failure monitoring in Helix with Zookeeper, which itself relates closely to these systems. In general, it is easy to see the lineage of these systems reflected in Helix.

7. CONCLUSION

At LinkedIn we have built a sophisticated infrastructure stack of DDSs, including offline data storage, data transport, data serving and search systems. This experience has put us in great position to observe the complex, but common, cluster management tasks that pervade all of our DDSs. This motivated us to build Helix. We designed Helix with a diverse set of DDSs in mind to ensure its generality.

Helix lets DDSs declare their behavior through a set of pluggable interfaces. Chief among these interfaces are a state machine that lets the DDS declare the possible states of their partitions and transitions between them, and constraints that must met for each partition. In this way, DDS designers concentrate on the logic of the system, while the Helix execution algorithm carries out transitions in the distributed setting.

We have had a lot of success with Helix at LinkedIn. By providing the performance and functionality a DDS would normally target for itself, we have indeed offloaded cluster manager work from a number of DDSs. Helix even lets these DDSs make what would normally be drastic changes to the way they are managed with just minor changes to the Helix state model.

We have a number of future directions for Helix. One of them is to simply increase its adoption, a goal we expect our open-source release to accelerate. A second goal is to manage more complex types of clusters. The first challenge is to handle heterogenous node types; we plan to approach this with the notion of *node groups*. We cluster nodes by capabilities (cpu, disk capacity, etc.) and weight more performant groups to host larger numbers of partitions. The second challenge is to manage clusters over increasingly complex network topologies, including those that span multiple data centers.

A third goal is to push more load balancing responsibility into Helix. Helix's alerting framework lets it observe imbalance, and we must continue to extend the Helix execution algorithm to to respond to imbalance, yet remain completely generic across DDSs.

8. REFERENCES

[1] Apache Cassandra. http://cassandra.apache.org.
[2] Apache Hadoop. http://hadoop.apache.org/.
[3] Apache Hadoop NextGen MapReduce (YARN). http://hadoop.apache.org/.
[4] Apache HBase. http://hbase.apache.org/.
[5] Apache Mesos. http://incubator.apache.org/mesos/.
[6] Hedwig. https://cwiki.apache.org/ZOOKEEPER/hedwig.html.
[7] MongoDB. http://www.mongodb.org/.
[8] SenseiDB. http://www.senseidb.com/.
[9] Zookeeper. http://zookeeper.apache.org.
[10] F. Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.
[11] B. F. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, 2008.
[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[13] R. Honicky and E. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *IPDPS*, 2004.
[14] LinkedIn Data Infrastructure Team. Data infrastructure at LinkedIn. In *ICDE*, 2012.
[15] J. Shute et al. F1-the fault-tolerant distributed rdbms supporting google's ad business. In *SIGMOD*, 2012.
[16] M. Zaharia et al. The datacenter needs an operating system. In *HotCloud*, 2011.