An Evaluation of ZooKeeper
for High Availability in System S

Cuong M. Pham,
Zbigniew Kalbarczyk,
Ravishankar K. Iyer
University of Illinois at Urbana-Champaign
{pham9, kalbarcz, rkiyer}@illinois.edu

Victor Dogaru
IBM Software Group
Oakland, CA, USA
vdogaru@us.ibm.com

Rohit Wagle,
Chitra Venkatramani
IBM T.J Watson Research Center,
Yorktown Heights, NY, USA
{rwagle, chitrav}@us.ibm.com

ABSTRACT

ZooKeeper provides scalable, highly available coordination services for distributed applications. In this paper, we evaluate the use of ZooKeeper in a distributed stream computing system called System S to provide a resilient name service, dynamic configuration management, and system state management. The evaluation shed light on the advantages of using ZooKeeper in these contexts as well as its limitations. We also describe design changes we made to handle named objects in System S to overcome the limitations. We present detailed experimental results, which we believe will be beneficial to the community.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – *performance measures*

D.2.4 [**Software Engineering**]: Software/Program Verification – *reliability*

General Terms

Reliability, Performance

Keywords

Distribute systems, high availability, stream processing, distributed coordination.

1. INTRODUCTION

ZooKeeper (ZK) [1] is a scalable, highly available, and reliable coordination system for distributed applications. The primitives exposed by ZK can be leveraged for providing dynamic configuration management, distributed synchronization, group and naming services in large-scale distributed systems. This paper evaluates the use of ZK as the coordination backbone for System S (commercialized as InfoSphere Streams [4][8]), a distributed streaming middleware system. We present our findings from a detailed experimental study to understand the application of ZK in System S, both to replace some of the existing services and to provide new capabilities. We also detail the design changes we made to System S to better utilize ZK capabilities. We expect the findings will be useful to distributed systems designers looking to leverage ZooKeeper as the coordination backbone.

System S applications are developed to analyze high-volume, continuous data from a variety of sources. The programming model supports application specification in the form of a dataflow graph, with analytics components or operators interconnected by streams, which carry tuples of a fixed schema. The System S runtime hosts applications from multiple users, deploys the compiled operators (called Processing Elements or PEs) across a distributed system, manages their streaming interconnections, monitors and manages their resource usage and lifecycle. Some requirements of the system that make ZK a good candidate as a coordination backbone include:

-   *   **High Availability**: System S applications are long running, and process data continuously. High availability is a crucial requirement both from an infrastructure and an application point of view. If any of the components fail, the system has to detect it and take appropriate recovery actions [3].

-   *   **High Performance**: System S is a high-performance system supporting a dynamic application execution environment. An application can change its topology during runtime based on analysis results, and new applications that connect to existing ones can be launched and removed dynamically. Supporting these features requires a high-performance control and coordination backbone.

-   *   **Scalability**: System S can support a very large set of applications and is scalable over hundreds of nodes. This requires a scalable coordination service to manage a large number of named entities, and a large number of clients.

-   *   **Management Simplicity**: Currently System S leverages different services to provide system recovery (DB2), system coordination and configuration (file-system). Simplifying this to a single system makes management in terms of deployment and troubleshooting easier.

The ZK architecture satisfies these requirements for System S and is a good candidate due to its scalability, resiliency, in-memory implementation, event-based interface, and a wait-free data-object interface. In this paper, we evaluate ZK for the set of functions outlined below:

-   *   Resilient name server – providing a highly available name service, which stores information about all named entities in the system, such as PEs, and stream end-points, supporting a dynamic execution environment.

-   *   Dynamic System Configuration – providing a configuration service that supports dynamic updates and notifications to configuration parameters.
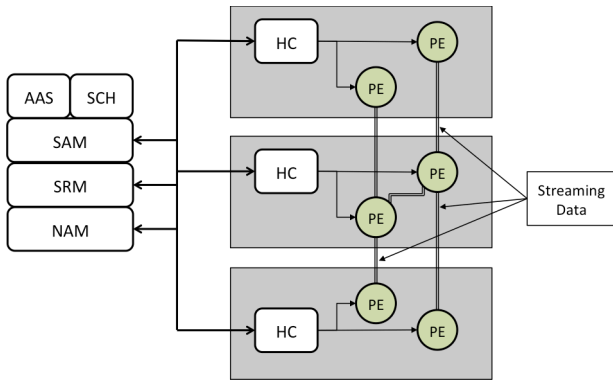
Figure 1: System S Runtime Architecture

- System state management and update notifications for the runtime state of system entities such as PEs, streams, and applications.

In this study, we designed and carried out a set of performance measurements for ZK in conditions which simulate specific System S application workloads. We report our findings on the advantages and shortcomings of using ZK in the System S context. We also report on changes to the System S runtime design to better leverage ZK capabilities. Specifically, we run experiments, which align our performance measurements with existing benchmarks, and compare existing System S performance to an alternative implementation using ZK for each of the functions outlined above.

Based on our experiments, we find that:

- ZK is a better alternative to the current name services in System S, which is either based on a shared file system or a non-recoverable service. ZK does not impact the system performance, while providing crash tolerance and eliminating the dependency on a shared file system across the System S cluster.
- ZK is a more easily manageable and higher performance alternative to the current system recovery feature in System S based on DB2 [3]. Although using DB2 is more reliable, configuring DB2 HADR is onerous. In the course of applying ZK to the above two scenarios, we had to make appropriate design choices to get the required functionality while maintaining high performance. One of the limitations in ZK is the size of each zNode. We had to ensure that System S state objects were appropriately sized and organized to get the best performance from ZK. We also discovered that the ZK C++ client significantly outperforms the Java client. Since most System S infrastructure components are written in C++, we could clearly leverage this benefit. In this paper, we quantify this difference for the awareness of application writers, when they design high-performance System S applications.

The rest of the paper is organized as follows. Section 2 provides background information and presents related work. Section 3 presents our experimentation methodology and setup. Sections 4, 5, and 6 present the results from the evaluation of ZK for the three functions outlined before. Section 7 concludes the paper.

2. BACKGROUND AND RELATED WORK

2.1 ZooKeeper Overview

ZK is a service, which provides wait-free coordination for large-scale systems. ZK can be used as the kernel for building more complex coordination services at clients.

ZK uses client-server architecture. The server side, called *ensemble*, consists of one *leader* and several *followers* to provide a replicated service. It requires that a majority of servers has not crashed, to provide continuous service. Crashed servers are able to recover and rejoin the ensemble. If the leader server crashes, the rest will elect a new leader. Only the leader can perform update operations; it then propagates the incremental state changes to the followers using the Zab protocol [2]. Each server keeps a copy of the data in its memory, but saves the transaction logs and snapshots of the data in persistent storage for recovery.

Application clients implement their logic on top of ZK client libraries, which handle network connection and provide APIs for invoking ZK primitive operations. Currently ZK supports C/C++, Java and Python bindings for clients. A ZK client can establish a session with a ZK service, and sessions enable clients to move transparently among the servers. Sessions have timeouts to keep track of the liveness of the client and server.

The ZK *data model* provides its clients an abstraction of a hierarchical name space, like a virtual file system. The data node is called *zNode*. ZK *consistency model* guarantees that write operations are linearizable, but read operations are not. All write operations have to go through the leader, which is then responsible for propagating the updates to other followers. To boost the performance, read requests are handled locally by the server that the client is connected to. As a result, a read might return a stale value.

The ZK *consistency model* guarantees that write operations are linearizable, but read operations are not. All write operations have to go through the leader, which is then responsible for propagating the updates to other followers. To boost the performance, ZooKeeper has local reads. That means read requests are handled locally by the server that the client is connected to. As a result, a read might return a stale value.

ZK implements a useful feature for coordination, called *watches*. The idea is to allow the client to monitor, or watch for modifications on zNodes. Clients set watches on zNodes they want to monitor, and then they will be notified asynchronously when watched zNodes are modified.

2.2 ZooKeeper in Other Systems

Many distributed applications have adopted ZK as an integral part of their systems, such as Distributed HBase [5]. Distributed HBase [5], which can consist of thousands of nodes, uses ZK to manage cluster status. For instance, HBase clients can query ZK to find the cluster to connect to. In addition, ZK is used to detect and trigger repairing process for node failures. HBase also intends to extend the usage of ZK for other purposes, such as monitoring table state and schema changes.

Several other real-time streaming analytics systems, such as Stormy [6] and Twitter Storm [7], have also integrated ZK in their implementations. While Stormy [6] employs ZK to provide consistent leader election, Twitter Storm uses ZK to implement Reliable Runtime with auto restart, and Dynamic Configuration changes.
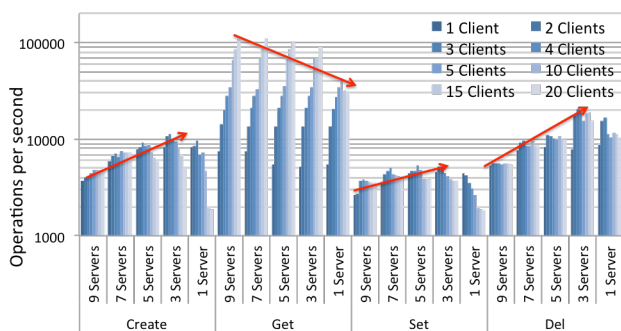
Figure 2: ZK basic operations throughput

This paper describes in details the intended use cases of ZK in System S, as well as presents our in-depth performance and availability analysis of these use cases.

2.3 System S Overview

System S [4][8] comprises of a middleware runtime system and an application development framework, geared towards supporting the development of large-scale, scalable and fault-tolerant stream processing applications. An application is essentially a flowgraph in which operators carry out portions of the data processing analytics by consuming and producing new streams, leading to the extraction of relevant results [9]. Once an application is compiled, a set of runnable processing elements is created. A processing element (PE) is a runtime container for portions of the flowgraph, i.e., a collection of operators and their stream interconnections. PEs belonging to an application can be logically grouped together to form jobs. A System S user can then start up the application by submitting it to the middleware runtime system, thereby creating one or more jobs. The jobs can then be monitored, moved and canceled using system management tooling.

The System S middleware runtime architecture (Figure 1) separates the logical system view from the physical system view. The runtime contains two distinct sets of components – the centralized components are responsible for accepting job management and monitoring requests, deploying and tracking streaming applications on the runtime environment and the distributed components, which are responsible for managing application pieces deployed on individual hosts. Specifically, the Streams Application Manager (SAM) is the centralized gatekeeper for logical system information related to the applications running on System S. SAM pro- vides access to this information to the administration and visualization tooling. SAM also functions as the system en- try point for job management tasks. The Streams Resource Manager (SRM) is the centralized gatekeeper for physical system information related to the software and hardware components that make up a System S instance. SRM is the middleware bootstrapper, carrying out the system initialization upon an administrator request. In the steady-state, SRM is responsible for collecting and aggregating system-wide metrics, including the health of hosts that are part of a System S instance and the health of the System S componentry itself, as well as relevant performance metrics necessary for scheduling and system administration.

The runtime system also includes additional components, which we briefly describe here. The Scheduler (SCH) is the component responsible for computing placement decisions for applications to be deployed on the runtime system [10][11]. The Name Service

(NAM) is the centralized component responsible for storing service references enabling inter-component communication by associating symbolic names with resource endpoints that can be registered, unregistered and remotely queried. The Authentication and Authorization Service (AAS) is the centralized component that provides user authentication as well as inter-component cross authentication, vetting interactions between the components.

The runtime system has two distributed management components. The Host Controller (HC) is the component running on every application host and is responsible for carrying out all local job management tasks including starting, stopping and monitoring processing elements on behalf of requests made by SAM. The HC is also responsible for acting as the distributed monitoring probe on behalf of SRM ensuring that the distributed pieces of applications remain healthy. Finally, a System S runtime instance typically includes several instances of the Processing Element Container (PEC), which hosts the application user code embedded in a processing element. To ensure physical separation, there is one PEC per processing element running on the system.

Some System S components (SAM, AAS, SRM and NAM) maintain an internal state in order to carry out their operations. Each component's internal state reflects a partial view of the overall System S instance state to which the component belongs. This internal state must be durable if the component is to recover from failure.

Stateful centralized services currently save their state in DB2 to support recovery from failure. Distributed services recover state either by querying the centralized servers or the environment.

3. EXPERIMENTAL SETUP

All experiments are conducted on a selected group of RedHat 6 hosts. Each host contains one Intel Xeon 3.00 GHz CPU (4 cores) and 8GB RAM. All the hosts are interconnected via 1 Gigabit Ethernet with approximate ping time is steady at 0.095- millisecond round-trip. ZK server hosts also have local hard-disks where the ZK snapshots and transaction logs are stored.

3.1 Primitive Operations Throughput

In order to establish the baseline of ZK's performance on our experimental cluster, we present the following experiment, which examines the throughput of ZK primitive operations under varying workloads and varying the number of servers in a quorum. The number of clients in each test ranges from 1 to 20. Each client issues 100,000 asynchronous requests and then waits for all of them to finish at the server side to report the throughput independently from other clients. The throughput of a server ensemble is the aggregated throughput of all the clients running concurrently. The result is shown in Figure 2.

As expected, the throughputs of Write request, including Create, Set, and Delete, increase as the number of servers in a quorum decreases, except in the case of standalone server. This behavior is expected as ZK is using primary-backup model [2]: only the leader can make updates, then broadcast atomically to other following servers; the more following servers, the greater the time to complete the broadcast.

As Read requests are distributed evenly to all the available servers, where they are processed locally, the throughput increases as the number of server increases. However, when the servers are under-utilized, adding more servers does not improve the throughput. As shown in Figure 2, the throughputs of ensembles, containing from 1 to 9 servers are the same with 1 to 4 clients, as

even the standalone server is underutilization. Read throughput is one order of magnitude faster than the Write throughput.

In summary, adding more servers into the quorum, on one hand, increases Read throughput and number of tolerable server crashes. But on the other hand, it consumes more compute resources and decreases the Write throughput.

4. ZOOKEEPER AS A NAME SERVICE SERVER

4.1 Use Case Description

The Name Service (NAM) is responsible for presenting an interface where System S components and applications can register and locate remote resource endpoints. The space in NAM is organized in a directory-based hierarchy, where an object can be placed anywhere in the directory structure.

Currently System S offers two implementations for NAM: Distributed NAM and File System NAM. Distributed NAM is a scalable daemon suitable for large deployments. The service is not backed by durable storage; therefore it is a single point of failure in the system. The File System NAM implementation is suitable for simple deployments as well as development and testing environments. This implementation relies on an NFS shared file system to store and propagate entries, which is intrinsically recoverable.

Although NAM does not significantly impact the performance of the stream applications, it plays a critical role in the availability of the streaming system. According to our performance profiling, even during then job submission time when NAM experiences a peak of activity, the applications spend only about 1% of their execution time invoking NAM. However, System S cannot tolerate NAM unavailability, as this service is in the critical path of inter-component communication. During job submission, SAM contacts the HCs in order to start PE instances on a subset of the instance hosts. During initialization, each PE is responsible for establishing data connections with the other PEs in order to send data streams, which results in a large number of NAM requests. Specifically, a PE (i) registers its data input ports with NAM; (ii) queries NAM for the host and port of the PE it has to connect to.

To test this use case, we developed a ZK-based NAM implementation and integrated it with System S. System S's components are linked against the ZK client library to make requests to the ZK NAM. Current NAM operations translate into zNode Create, Get, and Delete operations.
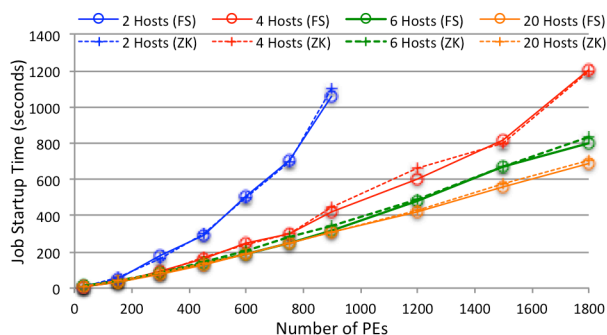

Figure 3: Job startup time File System vs. ZK NAM

We compared the startup time of a Streams job using ZK NAM under normal working conditions, versus during ZK leader crashes. We also compared ZK NAM and file-based NAM under normal conditions but not during server failures, since the underlying NFS implementation was not configured for high availability.

4.2 Failure Free Execution

This experiment examined the performance of ZK NAM in normal working conditions. Particularly, we compared the *startup time* of stream jobs using ZK and File System NAM. We also inspected closer at the Read and Write request arrival rates at NAM during stream job startup.

We used the Long Chains performance benchmark to generate workload for the experiment. Each Long Chains job consists of one input operator and one output operator joined using a set of relay operators linked in chains. The input operator sends tuples to a given number of operator chains. Each chain has a configurable length. All the chains are joined at the output operator. The number of chains is also customizable for each job.

Figure 3 shows the comparable startup times of ZK NAM and File System NAM under varying number of PEs and varying computing resources (number of hosts running the job). This result confirms the expectation mentioned above, as NAM should not significantly impact the performance of the application.

Figure 4a shows the Write and Read request arrival rate at NAM during job startup time. At peak, NAM receives >350 Read requests and >50 Write requests per second. These rates are low compared to ZK throughput (Section 3.1), which shows that the load posed by job submission is under the capacity of our ZK
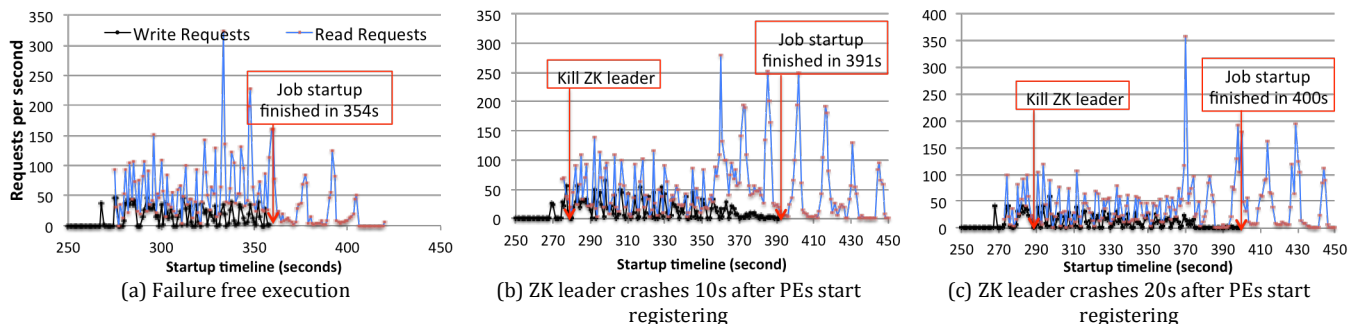

| (a) Failure free execution | (b) ZK leader crashes 10s after PEs start registering | (c) ZK leader crashes 20s after PEs start registering |

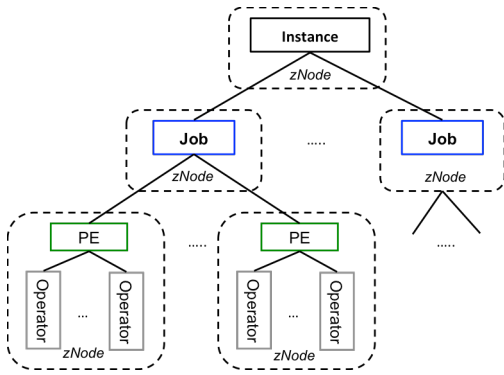**Figure 4: Request arrival rate at NS during job submission (Long Chains benchmark with 900 PEs running on 4 hosts)**

Figure 5: Persisting the model of a System S job to ZooKeeper zNodes

Figure 6: Multi-op performance
Create (black lines): one Multi-Op per job; Create1 (blue lines): multiple Multi-Op per job. Experimented with two sets of three and five ZK servers. zNode size is 512B or 1KB in each experiment.

installation. One ZK server ensemble can accommodate multiple job submissions at the same time, and still has available bandwidth for other tasks as well.

Figure 4a also shows that Read requests arrive at NAM about three times faster than Write requests. This is a good ratio for ZK. The number of Write requests depends on the number of PEs, as each PE needs to register its input ports once. The number of Read requests depends on the topology of the PEs: how many neighbors each PEs needs to query. This ratio is aligned with Long Chains topology. In general, this ratio tends to be greater, which makes it a good workload pattern for ZK.

While delivering similar performance, ZK can tolerate server failures and reduce stress to the file system, which is often the IO bottleneck to many distributed system. The next section examines NAM's behavior under ZK server crashes.

4.3 ZooKeeper Server Failure Execution

Figure 4b and 3c demonstrate that NAM can sustain ZK Server crashes. A ZK leader crash impacts a streaming job start time by increasing the total duration of the start operation.

Two possible causes of the additional delay are: (i) increasing workload for the rest of the servers; and (ii) execution stalled during session migrations from failed server to other servers. The first cause does not happen in this case. Because adding additional requests, that other servers have to handle for the failed server, does not exceed each server's capacity. That rules out the possibility that increasing workload on each active server causes increasing the running time. The second cause is what really happens in this case. In order to confirm this argument, we experimented with different server crash duration, or Mean-Time-To-Repair (MTTR), and different crash points.

When varying the MTTR of the server failure, we did not observe any changes in the startup time. For example, if the crashed server is restarted after 60 seconds, while the job is still starting-up, the startup time is the similar to the startup time when the crashed server is not restarted. This experiment also confirms the observation in section 3.1: once ZK servers are underutilized, adding one ZK server to the ensemble does not impact the application performance.

However, the crash point in time of the server does affect the performance. As showed in Figure 4c, where the crash point was
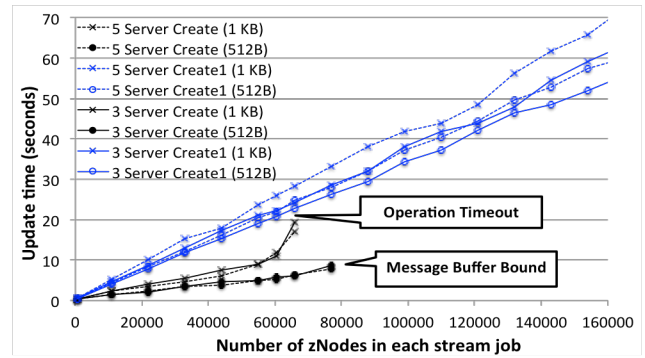
moved further 10 seconds back in comparison with Figure 4b, we see a slight increase in the startup time. The further away we are from the startup time, the more PEs have established connections with NAM. Therefore, if a server crashes, there are more ZK clients which have to migrate their sessions to the other servers. That causes a longer delay for name registering and querying.

These two experiments again confirm the cause of the increasing running time is the session migration due to server crashes.

5. ZOOKEEPER AS A RECOVERY DATABASE

5.1 Use Case Description

A System S instance runs one or more streaming applications logically managed as jobs [3]. An application is essentially a graph in which the vertices are the data flows and the nodes are operators running the Streams application code. A PE is a runtime container, which can host one or more operators. Operators and PEs have ports, which are connected to create data streams. These entities are structured in a hierarchical model, which for the purpose of our experiments is mapped to a hierarchy of zNodes, as illustrated in Figure 5.

In the System S architecture, SAM has two responsibilities:

- *Instance Management*: accepting job management requests for deploying streaming applications and updating the associated instance state.

- *Instance State Access*: providing access to logical system information related to the applications running on System S (the instance model) to administration and visualization clients.

SAM processes job submission requests in stages, which generate updates to the System S instance model. The current System S implementation uses two building blocks, which together provide system-wide fault tolerance: a reliable communication infrastructure (CORBA), and a relational database (IBM DB2). Instance model updates and messages to remote components within each stage are persisted within a single transaction.
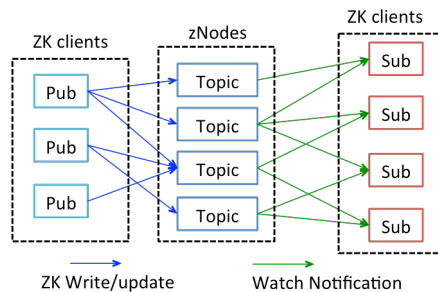
Figure 7: Publisher-subscriber model for Dynamic System Configuration

Figure 8: Watch & Read latency

C: number of subscriber clients per machine. M: number of machines running the subscriber. The black (for Java clients) and red (for C++ clients) arrows show the increasing trend of latencies when increasing the number of zNode each client monitors

With ZK, a set of related state updates can be executed using a ZK multi-operation (multi-op), which allows a batch of create, delete, update or version-check operations to succeed or fail together.

To help evaluate multi-op performance when using ZK, we want to measure the maximum zNode multi-op rate achievable for various node numbers.

5.2 Single Multi-op for One Stream Job

Multi-ops are submitted to ZK as one single request. Even though the request contains a list of operations, ZK applies the same *message size boundary* and *operation timeout* as for a single primitive operation request, which sets a limit for the number of operations packed in each multi-op.

In order to increase the number of operations batched in one multi-op, we configured ZK server to accept a message size up to 45MB, as well as extend the operation timeout to 30 seconds.

The black lines in **Figure 6** show that the multi-op execution time depends on both the number of batched operations packed into one message and the data size of each operation. The time taken for the multi-ops, which creates multiple 1KB zNodes per request, starts growing quickly after reaching approximately 50,000 zNodes per request due to CPU bound at the server. We start encountering operation timeouts on reaching 66,000 zNodes per request. When operating with a 512B sized nodes, the number of zNodes created by each Multi-Op is limited by the message buffer size.

5.3 Multiple Multi-ops For One Streams Job

To ensure the scalability, a ZK-based implementation would have to use a combination of the following techniques:

- Structure instance model changes such that properties, which do not change during the life of a job, are grouped into a small number of "constant" zNodes. This technique simplifies the zNode management (for example, "constant" zNodes can be created in a separate multi-op).

- Split job submissions into several stages, which are atomically executed. In this case, the responsibility of restoring the system to the state prior to the job submission in case of a failure will partially fall onto the client.

The blue lines in **Figure 6** illustrate the execution time where the execution of a job submission is split into several multi-ops. We believe that the extra overhead of the multi-op logic causes longer overall update times for the same number of nodes (about three times longer than in the extreme case where each multi-op updates a single node). In a real-life implementation, we expect that combining multiple operations in the same multi-op can shorten
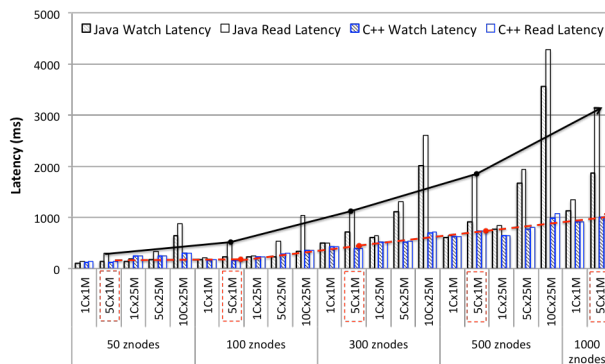
the overall update time. Further, where possible, by combining asynchronous execution of stage N with preparation of update operations for stage N+1, total execution time can be reduced.

6. ZOOKEEPER AS A PUBLISH-SUBSCRIBE MIDDLEWARE FOR DYNAMIC SYSTEM CONFIGURATION

6.1 Use Case Description

In this use case, we evaluate ZK as a mechanism for Dynamic System Configuration. System S contains tools, which help users inspect the state of an instance and retrieve PE-based and operator-based data flows for the set of applications running on that instance. The tools can depict the runtime environment using a topological visualization perspective with overlaid performance metrics. In order to provide a fresh view of the system, these tools run the following query types: (i) Retrieving instance topology and state; and (ii) Monitoring instance performance.

The tools periodically query SAM to retrieve the system state and topology and refresh their stream graph view. In an implementation based on ZK, instead of periodically polling SAM, clients would set watches on nodes of interest and let ZK send notifications when nodes are updated.

To simulate this usage pattern, we implemented a simple publish-subscribe system using ZK watches as illustrated in Figure 7. ZK servers act as the publish-subscribe middleware, where each topic is represented by one zNode, while the Subscribers and Publishers are ZK clients. The subscriber sets watches on the zNodes (the topics) that they wish to monitor. Meanwhile the publishers update the topics' content by writing to the corresponding zNodes. Upon each update, ZK servers send out notifications to the subscribers that have set watches on the updated zNode. Upon receiving a notification, each subscriber sends a Read request to the ZK servers to query the content of the zNode, and then resets the watch on that zNode.

The semantics is slightly different from a regular publish-subscribe system, where the middleware sends the updated content to the subscribers upon each notification. ZK only sends notifications telling the subscribers that there is a recent change in the watched zNode, and then the subscribers are responsible for

retrieving the updated content. In addition, the subs have to re-set the watch if they still want to monitor that zNode.

6.2 Failure Free Execution

It is of importance to know how much time it takes for the watch notifications and the updated data to reach all the subscribers under normal execution conditions.

In this experiment, we setup one publisher that updates N zNodes at the same time. There are M subscribers evenly distributed on P number of hosts. Each subscriber sets watches on all N zNodes. We measured:

- *The watch latency*: the time from a publisher starts updating all N zNodes until all the watch notifications arrive at all the M subscribers.

- *The read latency*: the time from a publisher starting to update all N zNodes until all the data (after subscribers issue read requests) arrive at all the M subscribers.
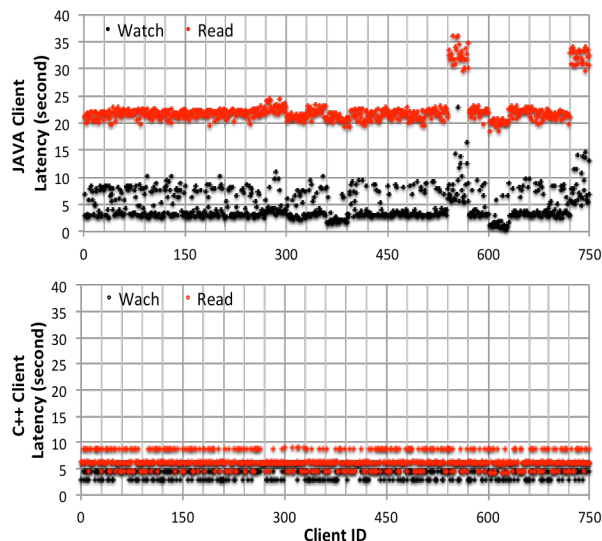
Since each subscriber performs a considerable amount of processing in this use case, we examined the performance of both C++ and Java implementations of the subscriber. The results of these experiments are shown in Figure 8.

C++ subscribers outperform the Java ones, and the gaps become more significant when increasing either the number of subscribers running on each machine or the number of zNodes monitored by each subscriber. When each machine has only one subscriber, and each subscriber monitors less than 1000 zNodes, the watch and read latencies of C++ and Java subscribers are comparable. However, as clearly shown by the trend lines in Figure 8, the watch and, especially, read latencies of Java subscribers increase faster than for C++. The same trend can be observed when keeping the number of zNodes constant, but increasing the number of subscribers running on each machine. With five subscribers on each machine and up to 1000 zNodes monitored by each client, read latency of the Java implementation is from 2 to 3 times slower than that of the C++ implementation.
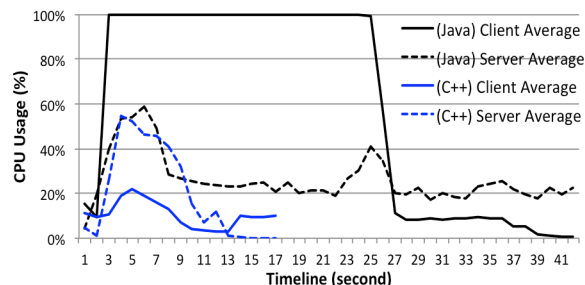
Figure 9b shows the average CPU utilization of Java and C++ subscribers across 25 machines; each machine has 30 subscribers; each subscriber monitors 1000 zNodes. The C++ lines are shorter because the C++ subscribers finished receiving the update faster than the Java ones. The CPU utilization peaks of ZK servers (the dashed lines) are similar when serving C++ and Java subscribers. However, while 30 C++ subscribers utilize at most 22% of the CPU resource, 30 Java subscribers consume the entire CPU resource of the machine. Java subscribers are the scalability bottleneck of the ZK based publisher-subscriber system.

It is also worth to note that each subscriber (in both C++ and Java implementations) runs as a single-threaded user process. Thus each Java subscriber requires a Java Virtual Machine (JVM). It could be more efficient to implement each subscriber as a thread, so that its footprint would be smaller. But we did not explore that proposal, because our design requires relatively isolation and independence between subscribers.

Figure 9a-b further visualize the latency differences between C++ and Java subscribers in the same setup. As we can see the watch notifications start arriving about 2 seconds after the update for both C++ and Java subscribers. However, as C++ subscribers are able to accommodate more requests at the same time, they finish faster than the Java subscribers. The performance of C++ subscribers is stable, as they all finished after 4-9 seconds. On the

(a) Detailed latencies on each client. The black dots and the red dots show the watch arrival time and read data completion time, respectively. Each machine hosts 30 clients (therefore each grid column represents the clients of one machine)

(b) Average CPU utilizations across 25 machines

Figure 9: Java vs. C++ clients (1000 watches/client)

other hand, there is a wide gap (~10 seconds) between watch arrival time and read completion in Java subscribers. It is also noticeable that Java clients are relatively unstable due to the fact that they are exhausting the resources of the machines. Therefore the performance of Java clients is more sensitive to the noise in the system (e.g. created by other system background services).

We did not compare Java and C++ clients in the other two use cases because these use cases mostly exercise the servers, therefore we anticipated that the performance of the clients would not significantly impact the performance of the overall service. On the other hand, since this particular use case involves the execution of the clients most heavily, we decided to examine the differences between Java and C++ clients.

6.3 ZooKeeper Failure Execution

In this experiment, we quantified the impact of ZK server crashes on the availability of our system. We used C++ clients in this experiment to achieve the highest server utilization.

In order to reduce the performance and network overhead, watches are managed locally in each server. The caveat of this
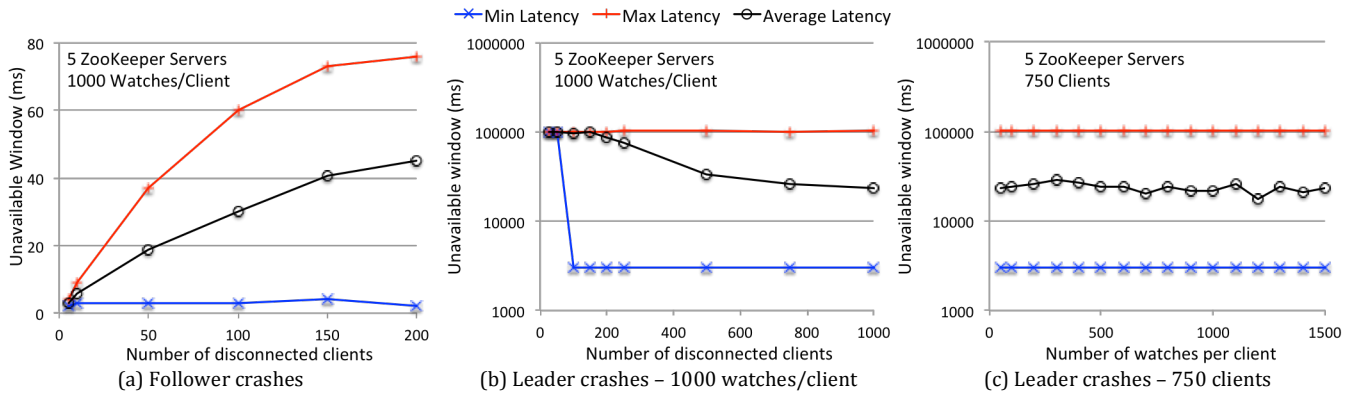
Figure 10: Unavailable window after (a) Follower crashes and (b, c) Leader crashes

design is that the other servers are not aware of the watches that the crashed server was managing. Therefore, not only do the clients have to reconnect to the other servers, they have to resubmit all the active watches to the newly connected server. From when the server crashes to when the watches are resubmitted successfully, the clients will lose all the watch notifications that occur during that period. We call this period *watch unavailable window*, as depicted in Figure 11.

6.3.1 Zookeeper follower crashes

Figure 10a illustrates the watch unavailable window of the clients when a following ZK server crashes. In this experiment, the server ensemble consists of 5 servers; and each client monitors 1000 watches. Because the clients are evenly distributed to 5 servers, the number of disconnected clients due to one, follower crashes is one fifth of the total number of clients.

The *min latency* and *max latency* are the read latencies of the first and the last clients, respectively, that receive all the updated data. The chart shows that the min latency is constant (3 milliseconds), and the max latency increases gradually as the number of disconnected clients increases. This latency is negligible for many applications. For example interactive Stream Console users would not be able to notice this latency of update.

6.3.2 Zookeeper leader crashes

Figure 10b-c illustrate the watch unavailable window of the clients when a ZK leader crashes. This window ranges from 3 to 100 seconds. This is a considerable impact on the availability of the service. The reason for this long unavailable window is: ZK leader crashes force the rest of the servers to re-elect a new leader. During this re-election time, all clients are disconnected, thus no

request is severed. After the new leader is elected, all the servers start accepting connections. That also means there is a burst of watch resubmission requests initiated from all the clients.

7. CONCLUSION

This paper describes three intended use cases of ZooKeeper in System S: Resilient Name Service, Dynamic System Configuration using publish-subscribe model, and Recovery Database. Our in-depth analysis has shown that ZooKeeper is a viable coordination backbone, which will potentially improve the performance, reliability and availability of the next generation of System S Infrastructure.

REFERENCES

[1] Hunt, Patrick, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free coordination for Internet-scale systems." In *USENIX ATC*, vol. 10. 2010.

[2] Junqueira, Flavio P., Benjamin C. Reed, and Marco Serafini. "Zab: High-performance broadcast for primary-backup systems." In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pp. 245-256. IEEE, 2011.

[3] Wagle, Rohit, Henrique Andrade, Kirsten Hildrum, Chitra Venkatramani, and Michael Spicer. "Distributed middleware reliability and fault tolerance support in system S." In *Proceedings of the 5th ACM international conference on Distributed event-based system*, pp. 335-346. ACM, 2011.

[4] IBM InfoSphere Streams: http://www-01.ibm.com/software/data/infosphere/streams/

[5] Apache HBase: http://hbase.apache.org/

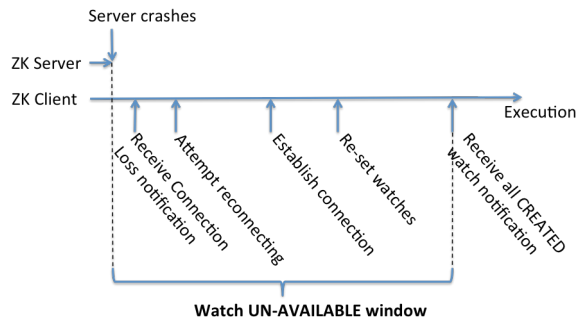[6] Loesing, Simon, Martin Hentschel, Tim Kraska, and Donald Kossmann. "Stormy: an elastic and highly available

Figure 11: ZooKeeper watch un-available window

streaming service in the cloud." In*Proceedings of the 2012 Joint EDBT/ICDT Workshops*, pp. 55-60. ACM, 2012.

[7] Marz, N., "A Storm is coming" http://engineering.twitter.com/2011/08/storm-is-coming-more-details-and-plans.html, August 2011

[8] Amini, Lisa, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. "SPC: A distributed, scalable platform for data mining." In Proceedings of the 4th international workshop on Data mining standards, services and platforms, pp. 27-37. ACM, 2006.

[9] Wu, Kun-Lung, Kirsten W. Hildrum, Wei Fan, Philip S. Yu, Charu C. Aggarwal, David A. George, Buğra Gedik et al.

"Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S." InProceedings of the 33rd international conference on Very large data bases, pp. 1185-1196. VLDB Endowment, 2007.

[10] Wolf, Joel, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. "SODA: An optimizing scheduler for large-scale stream-based distributed computer systems." In Middleware 2008, pp. 306-325. Springer Berlin Heidelberg, 2008.

[11] Wolf, Joel, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, and Kun-Lung Wu. "Job admission and resource allocation in distributed streaming systems." In *Job Scheduling Strategies for Parallel Processing*, pp. 169-189. Springer Berlin Heidelberg, 2009.