Andes: a highly scalable persistent messaging system

Charith Wickramarachchi, Srinath Perera, Shammi Jayasinghe, Sanjiva Weerawarana

WSO2 Inc.
Mountain View, CA
USA
{charith, srinath, shammi, sanjiva}@wso2.com

Abstract—A combination of factors: expanding user bases, the ubiquity of mobile communications, and newer technologies such as cloud computing and multi-core computing, are pushing today's systems to grow larger and larger. With their loosely coupled nature, distributed messaging systems often play a key role in such architectures. However, just like other parts of the architecture, these messaging systems also need to scale up, and they need to do so in three dimensions: quantity of messages, number of users, and size of messages. Although most current systems handle the first two dimensions, few of them efficiently support the third dimension. This paper proposes a novel method to implement a scalable and persistent broker that supports a publish/subscribe model and distributed queues using a NoSQL database and a coordination framework. We will discuss the design that uses recent advances in scalable database management and distributed coordination middleware, and we will compare the proposed models with other distributed message brokers.

Keywords—publish/subscribe, messaging, distributed queues, scale

I. INTRODUCTION

In the last few years, most Internet applications have seen an explosion of users and workloads. This has been caused by the wide-reach and ubiquity of Internet and advances in mobile technologies. As a result, the amount of connections between applications and the amount of data those applications have to handle have also increased. Handling these applications requires large-scale systems.

As the size of the systems grow, it becomes increasingly difficult to design them and keep them running. To avoid these difficulties, most such architectures use loosely coupled technologies. These systems, comprised of a wide range of users and components, need communication middleware to connect the different parts together. However, the direct use of point to point technologies, such as remote procedure call (RPC), forces this middleware to be tightly coupled. For example, conventional communication mediums will assume that both the server and client are available at the same time (transient communication); the client will wait for the server, and the client always sees the servers address.

Messaging technologies have been proposed to enable loose coupling in several dimensions, including time, synchronization, and space (e.g., Eugster et al. [2]). For example, technologies such as publish/subscribe and distributed queues enable users address their components using logical names, to send and receive messages in asynchronous pattern when the client or server is ready to receive them, and to communicate in a persistent manner with higher reliability.

For example, let us consider the following two usecases.

1) A large super market chain transfers prize updates from a central point to hundreds of branches, collects the transaction data and stock updates from the store, and sends them back to a central processing cluster. Due to geographical distributed and heterogeneous nature of branches, the usecase needs loosely coupled, interoperable, and persistent communication between different participants to ensure that messages are not lost due to node or communication failures.

2) Consider a large-scale, long-running computation that distributes data across large number of nodes, enables communication between these nodes to carry out the execution, and finally collects the results to build the final output. Communication between the nodes on such computations are best done through a persistent messaging model to recover from communication and node failures. For example, Amazon teams have proposed an Amazon Simple Queuing Service (SQS) based model to perform large-scale computations in the cloud.

To handle such scenarios and other similar cases, we need highly scalable message frameworks that support persistent communication.

Messaging systems support two messaging patterns: distributed queues and a publish/subscribe model. The first model is based on Queues and second is based on Topics. With both the models, users send messages to a Queue or a Topic while other users subscribe to the Queue or Topic. With Queues, the message is delivered to one of the subscribers; with Topics, it is delivered to all of the subscribers.

For example, lets assume there are two broker nodes A and B. In broker node A, there is a subscriber to topic “FOO,” and in broker B, there is a subscriber to the same topic FOO. When a message is published to the topic FOO, all the subscribers for the topic FOO need to get this message. In contrast, with Queues, each message will be sent to only one subscriber. A distributed messaging system consists of
many such broker nodes, and users can connect to either to
perform any operation without seeing a difference.
Generally we consider three different dimensions of scal-
ability in a messaging system.
1) Handling a large number of users (publisher and
subscribers) or large message rate for a single queue
2) Handling a larger number of queues/topics
3) Handling bigger messages
As we shall discuss in the related works section, there are
existing broker networks that could transfer messages across
a network of brokers in scalable manner. However, most of
those systems are built to handle small messages, and they
only support the first two dimensions of scale. Furthermore,
although many message brokers support persistence, they
are often built on top of RDBMS technologies that are not
optimized for fast writes or to support scale.
When a system has different parts that comprise of
different operating systems and technologies, interoperable
message protocols are essential to ensure seamless integra-
tion. WS-Eventing and WS-Notifications define a SOAP
based message protocol for publish/subscribe systems, and
that is the defacto answer in the SOA world. However, most
top enterprise usecases use a different protocol called AMQP,
which define an interoperable protocol for publish/subscribe
and queue based communication. Although there are many
brokers that support either protocol, none of them support
interoperability between two protocols.
This paper proposes a highly scalable messaging system
called “Andes,” which is build on top of Apache Cassan-
дра and Apache Zookeeper. Furthermore, it supports both
the AMQP message protocol as well as WS-Eventing and
seamlessly interoperates from one protocol to the other. We
have implemented the proposed architecture by extending
Apache Qpid, a well-known message broker that supports
AMQP based message exchanges, thus reusing many parts
from Qpid and providing support for the JMS specification.
We argue that advances in scalable database management
systems, such as Apache Cassandra [4] and distributed
coordination frameworks like Apache Zookeeper [1] have
yielded a sweet-spot, which enables a highly scalable mes-
sage broker that covers all three dimensions of scale. By
using Cassandra, which is highly scalable and optimized
for very fast reads and writes, the proposed implementation
significantly improves both performance and support for
large files.
The primary contribution of this paper is a proposing
a novel message broker architecture that uses advances in
scalable database management and distributed coordination
middleware to build a interoperable message broker that
scales on all three aforementioned dimensions of scale,
as well as how to implementing it and demonstrating its
viability.
Rest of the paper is organized as follows. The next section
discusses related work in distributed messaging systems,
and the following section discusses the architecture of the
proposed Andes messaging system. The following section
provides a performance comparison of Andes with other
JMS brokers and a scalability analysis of the system. The
performance analysis is followed by a discussion of pro-
posed architecture and how it makes a difference, and finally,
the last section concludes the discussion.

II. RELATED WORK

Distributed message broker networks are a topic that have
been addressed in detail. The main design goal of these
message broker networks is providing scalability and fault
tolerance. For example, message brokers like NaradaBro-
kening [9], Gryphon [10], Oracle Advanced Queuing [7],
TIBCO Rendezvous [8], IBM WebSphere MQ [6], and
Padres [11] consists of many brokers placed in some topol-
y and they work by routing messages to the other brokers
in the broker network to hide the fact that the broker network
consists of many nodes from the end user.
Generally message broker networks do not provide mes-
sage persistence. When they do, persistence is provided at
the edges of the network by storing messages in the node
that first receives the message into the network or the node
that is supposed to deliver the message to the subscriber.
This model works well with smaller messages and is very
fast when message persistence is not required. However the
model creates multiple copies of the message.
JMS brokers such as ActiveMQ [5], HorentMQ [12], Rab-
bitMQ [13], Qpid [14] etc provides another set of solutions.
Most of these support persistence by using a database to
write messages.
Scaling the publish/subscribe model is relatively easier
than distributed queues. For example, replication of all sub-
scriptions across all the nodes in the broker network would
allow the system to scale as that would allow each node to
sends any published messages at that node to subscribers.
Furthermore, since all subscribers receive all messages,
no coordination is required among the participating nodes.
Distributed broker network algorithms like NaradaBrokering
further optimizes message delivery.
In contrast, nodes in a distributed queue implementation
need tighter coordination to ensure that each message is
sent to only one subscriber. JMS implementations like
RabbitMQ, HorentMQ, Qpid and ActiveMQ include support
for message queues. Those systems call a distributed broker
network as a cluster. In a RabbitMQ Cluster, queues are
created and live in a single node, and all nodes know about
all the queues. When a node receives a request to a queue
that is not available in the current node, it routes the request
to the node that has the queue. To provide high availability
(HA), RabbitMQ has a mirrored queue arranged in the
master-slave fashion, and messages are replicated between
master and slave, so the slave can take over if the master
has died.
With HorentMQ, the clients define the clusters. Clients create cluster connections that load balance messages across all nodes in the cluster. It also supports message redistribution, which means if the broker does not have any subscriptions, the messages received by that broker are rerouted to other brokers that have subscriptions. RabbitMQ and HorentMQ scale for large numbers of queues, but they cannot handle a large number of messages per queue. With QPid, both the data and metadata are replicated across the brokers, which increase the replication overhead. However, clients can talk to any node without seeing a difference.

ActiveMQ supports the master/slave model and the network of brokers model. In the network of brokers model of ActiveMQ, the brokers are arranged in a topology, and subscriptions are propagated through the topology until messages reach a subscriber.

Among those solutions, both queues that are placed in a single node and the master/slave model are not scalable. The broker networks like ActiveMQ distribute messages in consumer priority mode where brokers that are close to the point of origin are more likely to receive the messages. The challenge is how to load balance those messages. In each of the above queue implementations, persistence is part of the individual nodes logic and not part of the routing algorithms. Also, in case of a node failure or deletion of subscriptions, store and forward networks have to redistribute the messages and hence lose the in-order delivery.

Broker nodes in the above architectures use message routing to communicate and ensure that all messages are delivered to right place. An alternative model is to write the messages to the database and have other clients read them from the database where message transfer between clients happens through the database. For example, in the centralized message broker called WS-Messenger [15], a demon that sends message to right subscribers and message acceptors communicate using a database.

Ekenayake et al. [3] have proposed a pub/sub model built on top of Azure NoSQL style data storage service. When a client publishes a message, the system stores the message in Azure Blob cache, obtains a URL for that blob, and sends the URL to subscribers. Then subscribers directly read the message from Azure Blob cache. Brokering message URL is done using a queue based model or a TCP Endpoints. In queue-based model, clients places the URL in Azure queue, and a worker node picks up the URL and sends it to subscribers. In the TCP endpoints based model, all URL passing happens through direct TCP connections. However, Ekenayake et al. does not address how multiple brokers can coordinate to deliver messages. Furthermore, it exposes the storage to end-users directly.

The proposed Andes broker builds on the Cassandra NoSQL database and Apache Zookeeper. Cassandra is a column family based NoSQL database that provides better scalability, performance, and tunable consistency by relaxing the data model from relational databases and adopting (distributed hash table (DHT) based architecture as proposed by Amazon Dynamo. A Cassandra ring consists of the multiple nodes that can scale up by adding new nodes. Apache Zookeeper is a distributed coordination framework, which enables simple implementation of most coordination algorithms by providing a shared memory space that uses Pxsos algorithm for coordination. More details can be found from [1].

III. ANDES ARCHITECTURE

Andes is a broker network that consists of multiple broker nodes, and they communicate by storing messages in a shared Apache Cassandra storage. Nodes use Apache Zookeeper for coordination when absolutely necessary. As shown by Figure 1 right, Andes supports both AMQP and WS-Eventing, and except for message parsing code, both support by the same Andes core. This enables AMQP users to receive messages published by WS-Eventing and vice versa, thus supporting protocol conversion as well. Andes extends Apache Qpid for AMQP support and provides WS-Eventing based Service API using Apache Axs2.

As described before, Andes supports both distributed queues and publish/subscribe model. Publishers publish to or Subscribers subscribe to a Queue or a Topic by talking to any broker node through either AMQP or WS-Eventing. In each case, users may publish messages to the topic or the queue through any broker, and the system delivers the message to the right parties regardless of the broker from which the subscriptions are placed. The existence of multiple brokers is transparent to the end users.

For each AMQP subscription, the client and the broker node maintain a persistent TCP connection, and any matching messages are sent to the subscriber through that TCP connection. For WS-Eventing subscriptions, connections as opened as needed.

When a broker receives a publish message, it stores it in the message store column family in Cassandra, and then places the message ID in a queue representation within Cassandra. Andes have implemented a queue representation using the Cassandra column model, and uses that queue to store Message IDs. Andes also stores the subscription details in a Cassandra column family, and each broker loads those details to memory and periodically refreshes them from Cassandra database.

Andes assigns an unique ID to each message, which is a combination of the timestamp and identity of the node where we received the message. It has the timestamp as the prefix, and as a result cassandra queue implementation stores messages sorted by their timestamp. Since each message has a unique ID, brokers can write to Cassandra without coordinating with each other. However, participating nodes need to coordinate with each other when reading and deleting
messages while ensuring reliable and in-order delivery. Let us look at how Andes implements each of the models.

A. Pub-Sub Architecture

In the pub/sub scenario, Andes keeps a queue in Cassandra database per each subscription, which we will call a “subscription queue". When a user subscribes, the node that receives the subscription updates the subscription table in Cassandra database and creates a subscription queue for that queue in the storage. When a broker receives a publish message, the broker first writes the message to a message store in cassandra. The message always has a topic associated with it, and the broker knows all subscriptions for the topic and writes the message ID to the queue associated with each subscription.

When a subscription is created, apart from creating the subscription queue, it starts a thread identified as “publisher thread" to poll it’s subscription queue. Since the queue is only used by this thread, it can access it without a need for coordination. The thread pulls out the messages id from this subscription queue, which are already sorted by their timestamp due to message IDs, loads the actual message for that message ID from message store, sends it to the actual subscriber using AMQP or Web Services, and finally removes the message ID from the subscription queue.

When a subscriber receives a message, it sends an acknowledgement back to the broker node. Then, the broker node removes the message ID from the subscription queue, but cannot remove the actual message from the message store as there may be other subscribers delivering the same message. To circumvent this problem, Andes uses a timeout based approach. There is a demon running in each broker node that periodically checks the messages in the message store and removes messages that are older than a given timeout period.

B. Distributed Queues

Distributed queues provide two guarantees: reliable delivery and ordered delivery. With strict ordering, Andes delivers messages from the same queue to all subscribers in the same order as they are published. However, to provide this guarantee, the system must deliver messages to subscribers one after the other.

To understand this better, let’s consider the following scenario. Lets assume there is queue Q1, and subscribers A and B have subscribed to Q1. Lets also assume that the two messages m1 and m2 have been sent to the queue. If the system is delivering messages in parallel, the following scenario can occur.

1) Broker starts sending m1 to A and m2 to B
2) m1 delivery fails as the subscriber A has failed or left.
3) However, by the time failure is known, m2 could have been delivered to B. Now unless there are other subscribers in the system, the system should either deliver m1 out of order or drop the message.

The same limitation is also present in other JMS brokers such as ActiveMQ and HorentMQ, where the different broker nodes in the topology deliver messages in parallel. If the delivery has failed, the retry will deliver messages out of order. The only way to avoid this problem is to wait for each message to be delivered before delivering the next message. Therefore, we provide two modes of operations: strict ordering and best-effort ordering.

1) Distributed Queues with Strict Ordering: Andes maintains a single Cassandra queue per each distributed (AMQP) queue in the messaging system, which we will call the “strict queue". Whenever, a new message is received by the queue, the broker places that message in the Cassandra message store and adds the message ID to the strict queue. Also, when a new subscription is made, the broker starts a the polling thread. Polling threads check for messages in the strict queue periodically and sends those messages to the subscriber associated with the subscription. To avoid race conditions when multiple polling threads accessing the same Cassandra queue, each node acquires a lock using the Zookeeper coordination framework, and polling thread only releases the thread after the message has been delivered and an acknowledge has been received. The lock ensures that messages are delivered one after the other, and this avoids the failure condition we discussed earlier that leads to out of order messages delivery.

2) Distributed Queues with Best-effort Ordering: With the best effort model, Andes creates one Cassandra queue per each distributed queue, which we will call a landing queue. We also assigns a broker node for each queue as the queue manager. This assignment is achieved through a coordinator that is elected through Zookeeper. Furthermore, Andes creates a cassandra queue per each node and topic pair, which we will call the node-topic queue. As shown by Figure 2, when a publisher publishes a message through a broker node to Q1 Queue, the broker node writes the message to message store and stores the MessageID in to the Q1 landing queue. Then the queue manager for Q1 reads MessageIDs from the landing queue Q1 and writes them to node topic queues associated with Q1 (e.g. Q1-Node1 and Q1-Node2) in a round-robin fashion.

Each broker may have multiple subscribers (e.g. subscription1 and subscription2 for Node6). Messages placed in node-topic queue are intended for any of those subscribers. Each broker runs a thread per each node-topic queue for that node (e.g. for Node6, it is Q1-Node6 and Q1-Node6), and this thread reads messages from the node-topic queue and sends them to the subscribers. Since only that thread reads from the queue, coordination through zookeeper is not required. Unlike the strict mode, this model delivers messages in parallel. However, if all subscriptions on a specific node has been removed, any message that has been
copied to node-topic queues need to be transferred to a different node-topic queue. When this happens, Andes might be forced to deliver messages in out of order.

In all three cases, messages are written to Cassandra once they are received and deleted only when they have been delivered to subscribers.

Andes uses Zookeeper for all coordination, and once started, it elects a coordinator using the election recipe of Zookeeper, which re-elects another node if the coordinator has failed. Each broker node when started creates an ephemeral z-node within Zookeeper, and Zookeeper will notify the coordinator if a node has failed. If a broker has failed, the coordinator detects the failure through Zookeeper and removes all subscriptions to that broker and moves any messages to other node-topic queues. Also, if the failed broker has been assigned as a queue manager for some queues, those queues are reassigned to a different broker.

If a broker has failed just after a message has been delivered and just before the message has been removed from Cassandra queue, other broker nodes that will take over the message handling have no way of detecting that the message has been delivered. In such cases, they would deliver the message twice. This is a known limitation of the current architecture.

Cassandra is designed to run all broker nodes within the same LAN (data center). Andes also assume the same. Following the general Cassandra deployment patterns, broker nodes and Cassandra communicate through unsecure
connections and they must be placed within the firewall, and only the broker communication ports should be accessible from the outside. This model is used to allow for very fast connections between Cassandra and the broker nodes. User calls to broker networks can use HTTPS, and Andes support username and password based authentication and authorization. It is worth noting that this limitation applies only to brokers nodes, and clients can be outside of the LAN where broker nodes resides.

To provide support for large messages, Andes uses several techniques. When a message is routed in a conventional broker network, each hop would create a new copy, and for large messages, this can be prohibitive. Andes writes the messages to Cassandra once it is received, performs coordination using message ID, and reads the message back only when Andes sends it the subscriber. Since data written to Cassandra can be read from any Broker Node, this avoids the need to make further copies. Furthermore, to avoid the need to load the complete message into memory when processing, Andes breaks each message into small blocks and stores it in Cassandra. AMQP can support both reading and writing in streaming fashion as the protocol natively supports blocks. We support streaming in WS-Eventing through Apache Axis2 streaming XML data model, which internally handles streaming. With WS-Eventing, if users send messages as pure XML, the message size may be high. Therefore, we recommend using MTOM based attachments to send the content of the messages. WS-Eventing API for Andes supports both pure XML-based content as well as attachment-based content.

IV. PERFORMANCE RESULTS

We have performed the following two experiments to empirically evaluate the Andes broker. The first evaluates the single node performance of the new model and compares it with existing message brokers while the second evaluates the scalability of Andes.

A. Comparative Study

This test compares the steady state message throughput of Andes with other well-known brokers.

For the test, we have setup one broker node and 40 concurrent publishers. After the system has processed 5000 messages for warming up, we measured the message throughput as seen by the consumer side by sending 10,000 messages. We have repeated the test with message brokers HornetQ and Qpid with different message sizes. Following graph depicts the results.

As Figure 3 on right depicts, throughput decreases with the size of the messages. Andes outperform Qpid in all occasions by factor of 2 to 3, and given that it works using Qpid parsing code, we believe that is a useful results. For smaller messages, HornetMQ is faster, which is likely due to overhead of storage based message exchange model used by Andes. However, for messages larger than 100KB, Andes become faster.

B. Scalability Study

For this test, we have setup broker networks with 2 to 8 nodes in best effort mode. Also we created a distributed queue called “BrokerTest” and added 20 subscribers distributed across those broker nodes. In each setup, we evaluated the consumer throughput while changing the number of concurrent publishers. For each number, we have sent 5000 messages of size 32KB and measured the throughput by considering the number of messages received within a fix interval. We run these tests using 6 Intel(R) Xeon(R) 16-core machines each having 16GB memory. We ran Cassandra and Zookeeper in a single node, ran two brokers per each node, and ran clients from a single node.

Figure 3 on the left depicts the results as seen by the consumer side. As depicted by the graph, adding more nodes increases throughput at both producer and consumer side, but more concurrency did not yield higher throughput. This suggests that 10 publishers can fully utilize the system. However, it worth noting that even with higher number of consumers, there is no degradation of throughput.

Time to publish a messages consistently was less than 1ms, and publish throughput changed from 120 at 2 nodes to 600 at the 8 nodes. Unlike with consumer throughput, the publisher throughput increases with both the number of nodes as well as the number of publishers.

V. DISCUSSION

Andes is optimized for routing large messages through a broker network, and while routing, only one copy is made as it will be stored and read from Cassandra distributed database. As the performance results suggest, this model is faster for large messages. However, unlike brokers such as QPId and RabbitMQ, Andes brokers do not have an in-memory model and they always store and serve messages from the persistent datastore. Consequently, with smaller messages Andes have much higher latency. Hence we believe, Andess model is good for large messages and persistent delivery whereas broker network mode is good for small messages.

We argue that Andes is scalable in all three dimensions of scale: large number of users, large number of queues or topics, and large messages.

Following are few examples classes of usecases where Andes broker can be useful.

1) Distributed computations often transfer large messages across the processors in the computation, and with long running processes persistent communication is useful. For example, Ekenayake et al. [3] have built a map reduce framework over pub/sub network and Andes can be very useful for such scenarios.
2) Uses cases that need persistent batch data transfers e.g. multiple retail stores may collect their transactions logs to a single data center.

3) File transfers over publish/subscribe style. The publish/subscribe nature would allow the interesting parties to receive the data updates e.g. collecting sensor data from many radars.

4) Handling work queues involving large files - queues are a well-known method for scheduling and managing work, and Andes would allow large files to be distributed across many workers.

5) Asynchronous communication allows involved parties to communicate with each other without both parties being ready to be communicate at the same time, and Andes would allow such communication with large files or messages.

As we discussed in the related work section, distributed queue implementations do not guarantee in-order message delivery in the clustered mode. Andes supports in order message delivery in the strict mode through a global lock acquired through Zookeeper. However, since this mode is very slow, Andes also provides a best effort mode that provides much higher throughput by only doing minimal coordination across brokers. Hence, Andes gives the end-users flexibility to choose the mode that best serves their use cases.

Andes also proposes a publish/subscribe algorithm that does not require any coordination between broker nodes. Andes can scale by adding more broker nodes, and scale is limited only by Cassandra, which itself is a highly scalable system. Each broker node only needs to be aware of subscriptions that are directly done to that node. Therefore, we argue that Andes is a highly scalable publish/subscribe alternative.

In contrast to publish/subscribe model, scaling distributed queues is complex, and there are three models proposed in earlier works. First is the master slave model where all messages received by a node are replicated to a second node, and this is mostly done for high availability as oppose to scale (e.g. RabbitMQ, Qpid). The second approach is to have a cluster of nodes and use a load balancing mechanism to route messages to the broker that handles the given queue (e.g. RabbitMQ, HorentMQ). This model scales for a large number of queues (scaling dimension 2), but does not scale for a large number of messages (high throughput) for the same queue. Third is the broker networks, which scales in first two dimensions. In contrast, Andes can scale on all three dimensions of scale.

Andes uses best effort mode as the default for distributed queues. Strict message ordering is slow and only provided for special cases. Best effort mode does not use coordination (Zookeeper) on a per message basis, and only use coordination to decide on the roles of nodes (e.g. queue manager), and that is needed only when a node or a subscription is added or removed. Zookeeper is known to scale up to 100,000 messages per second. Based on those two facts, we argue that we can scale up Andes by adding more broker nodes and in the best effort mode with scale limiting only on Cassandra.

Andes allows users to publish or subscribe to any broker node in the system, and the clients will receive the same output regardless of the broker it is talking to. In its default mode, Andes will provide a list of broker nodes to the end user, and they can talk to any one of them. If the current broker node has failed, clients should re-subscribe/publish to a different node. Alternatively, it is possible to assign the same domain name to all the nodes through DNS load balancing or using a Hardware load balancer.

All messages are stored in Cassandra in a replicated manner, and the system can recover even if some of the Cassandra nodes have failed, which is an additional benefit.
generally not available out of the box with other brokers. This eliminates the need for master slave queue replication (e.g. RabbitMQ, Qpid), which is very expensive. If a broker node has failed, other nodes will detect the failure through Zookeeper and reassign it’s roles to other nodes. Similarly, if a new node is added, other nodes detect the new node through Zookeeper and can incorporate it with the rest of the system.

VI. CONCLUSION

As discussed in the introduction, the primary motivator for Andes is the need for highly scalable persistent messaging with multi-size messages. Conventional message routing based pub/sub and distributed queue architectures can scale for large number of messages and large numbers of topics and queues. However, they are not optimized for persistent large messages. Andes incorporates a NoSQL storage (Cassandra) and a coordination framework (Zookeeper) to build a highly scalable messaging framework using a storage based approach.

The primary contributions of this paper are proposing an interoperable and highly scalable message broker architecture that uses advances in scalable databases and distributed coordination middleware, implementing it, and demonstrating its viability.

Andes provides a publish/subscribe model that does not need any coordination between broker nodes, a strict mode for distributed queues that provides in order delivery, and a best-effort mode for distributed queues. Each can scale by adding more broker nodes, where strict mode is limited on both Cassandra and Zookeeper while others only depend on Cassandra. Given that both Zookeeper and Cassandra are highly scalable systems, we argue that Andes also highly scalable.

Although messaging is a key enabler in SOA based integration, when dealing with large messages, current messaging implementation force users to couple it with a databases and write custom code to keep the data content outside the message, which leads to more complex systems. We believe Andes provides a cleaner alternative for such use cases. Andes is freely available as an open source project under Apache Licence.

REFERENCES


