Building a Distributed, Consistent Real-Time Cloud Computing Application

Kevin Chabreck
Duke University
Department of Electrical and Computer Engineering
kevin.chabreck@duke.edu

Yijie Zhuang
Duke University
Department of Electrical and Computer Engineering
yijie.zhuang@duke.edu

ABSTRACT

There is an ever-increasing emphasis on collaborative work in team-based environments. Many software solutions have emerged in recent years that attempt to facilitate such work; Google's Google Docs and Microsoft's Office 365 are two extremely popular web-based tools that allow users to simultaneously edit documents to a collaborative end. Factors such as perceived latency, document consistency, and system scalability play an important and often critical role in applications of this nature. Many optimization techniques that target these aspects of application performance exist, but often the only way to meet system goals and requirements under extreme load is to distribute work across multiple physical hosts.

PaintChat is a consolidated cloud-based system that allows users to collaboratively draw on a shared canvas. In an effort to gain hands on experience designing, developing, and deploying an actual distributed system, we have extended the PaintChat application into a functionally equivalent clustered system called *PaintChat v2*. In addition to the overarching goal of crafting a distributed application, particular focus was given to reducing response time during heavy workloads, maximizing the consistency of the shared document, and crafting a highly scalable decentralized system design with support for dynamic addition and withdrawal of member nodes.

1. INTRODUCTION

With the increasing prevalence of the Internet, cloud-based applications have become popular in both commercial businesses and people's daily lives. Cloud-based application providers aim to supply an Internet-based computing service, often at a global scale and under extreme load. The latency experienced between a user action and the associated server feedback is a critical factor in applications such as these - companies like Amazon have reported massive losses in users and revenue when response delays surpass 100 ms.

While latency is of great importance, it is not the only factor required for the successful operation of a cloud service. Consistency of shared data between multiple clients is often vital as well. Though some application models can tolerate a varying degree of consistency, many services can not. For example, banks and other financial services are incapable of handling any number of conflicting writes or stale reads. A naive way of preventing this from happening is by executing data transactions sequentially in a consolidated system architecture. This approach, however, has several enormous disadvantages: it is slow, it is incapable of scaling for heavy load, and it subjects the system to a single point of failure.

In order to solve these problems, people tend to deploy and provide the service among multiple servers. The strengths of this approach are twofold: not only does serving requests from multiple hosts allow for a greater number of connections to allow more throughput and fault-tolerance, which is also called distributed computing. By distributing the service from a single node to multiple server nodes, people could get benefits from the more concurrent operation and faster response time. But at the same time more potential issues arise that need to be taken into consideration to guarantee correct system behavior; data consistency and failure-recovery are just a few of many such issues we personally encountered while implementing a distributed service called *PaintChat v2*.

PaintChat is a cloud service that allows multiple users to draw collaboratively on a browser canvas [8]. In the previous efforts, we have implemented the PaintChat on a single server. In order to build up a distributed cloud application, we distribute PaintChat by deploying the cloud service among multiple nodes. We built up the distributed PaintChat back-end on the Akka framework. In this project, our goal was not just to distribute the application, but to actually improve its performance by doing so. In order to measure this goal, we evaluated our distributed implementation against the original in a realistic cloud deployment environment. While it is not imperative to the operation of the service, we also made it a priority to maintain the highest possible degree of consistency while still optimizing for response time and tail latency.

1.1 Initial Implementation

The initial design of PaintChat (which will be referred to as PaintChat v0 for the remainder of this paper) introduces several technology choices and architectural elements that remained constant throughout its evolution [7, 8]. A high level depiction of the architecture is shown in Figure 1, and several of the core components are described below.

Web Server: PaintChat is built around an entity capable of receiving and responding to HTTP requests. This is an important feature as it is a core requirement for establishing a WebSocket connection. This functionality is also used to serve requests for static objects and to provide RESTful endpoints that return information about the current system state.

ServerWorker: This actor is responsible for accepting incoming HTTP requests, managing connection, membership, and canvas state, and facilitating communication between concurrent users. Nearly all information sent to a PaintChat server will pass through the ServerWorker, which will handle it accordingly; this most frequently requires broadcasting an incoming message to all connected clients. Most messages also need to be logged or otherwise used to update the state of the system. This allows future users to learn of the system state upon connection. This also serves as a means of ordering updates and simplifies consistency concerns within the node.

ClientWorker: These actors are instantiated by the ServerWorker whenever a new HTTP request is received, and are responsible for serving the request. In the event that the request is an "Upgrade to WebSocket" request, this worker establishes a TCP connection with the client. This connection allows for bilateral communication between the client and the ClientWorker. Incoming messages from the client/WebSocket are received by the ClientWorker and passed to the ServerWorker. Outgoing messages received from the ServerWorker are forwarded to the client over the WebSocket connection.

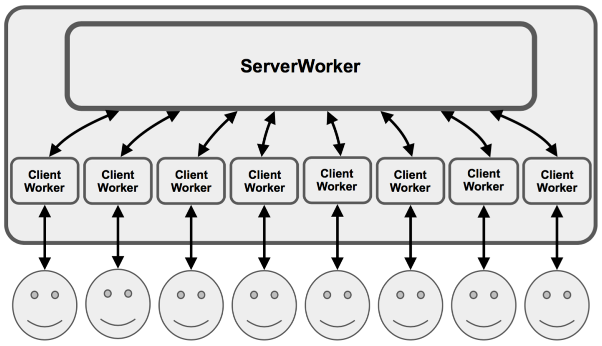


Figure 1: A high-level depiction of PaintChat v0's consolidated architecture shown hosting 8 client connections.

Application Layer and Communication Protocol: PaintChat v0 features a user-facing application layer written in vanilla JavaScript, and uses a simple protocol for communicating information between the browser and the back end server. In the interest of drawing a fair comparison between the initial implementation and our clustered alternative, we made the effort to implement this protocol as closely as the design would allow and to minimize the number of modifications made to the application. For the most part, we were successful in this goal and only a few modifications were necessary. [1]

2. DISTRIBUTED IMPLEMENTATION

While we began this project knowing that we wanted to distribute the PaintChat system, we weren't immediately sure what the best approach to doing so would be. Distributed services are fickle systems that demand extremely thorough planning, and even more thorough analysis when attempting to diagnose points of inconsistencies, bottlenecks, or races. We first present the high level architecture of our final implementation, followed by sections that drill down into each of the major facets of the system.

2.1 High Level Architecture

From a birds eye view, the general structure of PaintChat v2 is very similar to that of its predecessor. A highly simplified depiction of this architecture is shown in Figure 2. There are still ClientWorker actors responsible for client communication, and there are still ServerWorker actors charged with managing client membership, and receiving and broadcasting messages to and from all of the ClientWorkers local to the server. However, now there is more than just one ServerWorker in the system. Each node in the system has its own ServerWorker that not only receives messages from local ClientWorkers, but from remote ones as well.

It is important to clarify here that while the nodes in this system are aware of each other, *no actual communication takes place directly between remote ServerWorker actors.* Figure 2 is a bit misleading in this sense and could easily be interpreted this way, but is only intended to show the general flow of messages through the system. As a side note, this diagram coincidentally is reflective of one of our earlier approaches to distribution that was later scrapped in the favor of using Akka's own solution for this problem.

2.2 Akka Cluster

Akka provides multiple ways for sending messages between actors on different physical host machines or in

[1]Given a bit more time to fine-tune the consistency protocol, this may have been a different story.

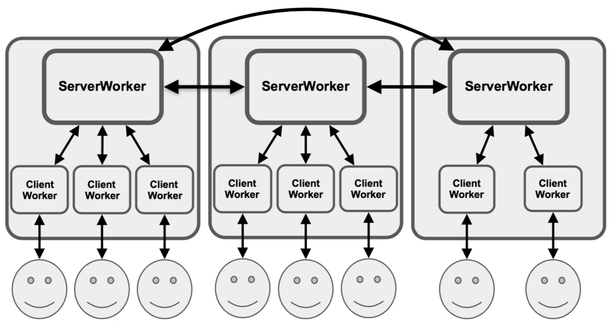


Figure 2: A high-level depiction of PaintChat v2's distributed architecture. Note: this is an extreme simplification, and is only meant to show the general flow of messages in the system.

different JVMs. However, we found that their Cluster module served the needs of our application particularly well. Not only does this package provide the means for remote communication between actors, but it also provides an extremely helpful set of abstractions and primitives geared towards developing resilient, asynchronous distributed systems that demand tight coupling with a stratified topology. One of the aspects of this module that we found incredibly helpful was the fact that it manages membership of the cluster for you automatically.

> The cluster membership used in Akka is based on Amazon's Dynamo system and particularly the approach taken in Basho's' Riak distributed database. Cluster membership is communicated using a Gossip Protocol, where the current state of the cluster is gossiped randomly through the cluster, with preference to members that have not seen the latest version. [3, Akka, *Cluster Specification*]

With the use of this module you need only to specify a few seed nodes, instantiate a `ClusterListener` actor and subscribe to the cluster events that you want to receive messages for, such as members joining or leaving the group. This also allows you to be notified of changes in the leader node, a cluster node selected behind the scenes by Akka responsible for "committing" information about membership status. We found this feature particularly useful for early approaches at a consistency protocol [3].[2]

Messaging within the cluster can be accomplished in a variety of ways, but we found that Akka Cluster's `DistributedPubSubMediator` actor fit our messaging model quite well [4]. This extension lets you subscribe or publish updates to a topic from anywhere in the cluster by simply sending a `Subscribe` or a `Publish` message to a mediator actor. Each node in the cluster will have one mediator. The mediator will route the messages to the appropriate actors, sending them first to remote subscribers and then to local ones. We found this to be a fantastically simple abstraction, and used it to set up our messaging structure in a very small amount of time.[3] Rather than ClientWorkers sending messages to their parent ServerWorkers and having to spend time sorting them out there, we publish every incoming client message to the DistributedPubSubMediator under the `client_update` topic, which is subscribed to by all of the ServerWorkers in the cluster.

2.3 Consistency

A primary concern we had for our application was the consistency of the document. We wanted to ensure not only that all messages were delivered to all clients, and that they arrived in the same order at every destination. This was simple and barely even a concern with the initial implementation. PaintChat v0's single ServerWorker combined with Akka's rule of only processing one in-order message at a time essentially guaranteed a global ordering of updates, barring any packet loss or buffer overflows. However, the introduction of multiple ServerWorkers complicated the situation. We were still able to achieve an ordering common to clients connected to the same cluster node by adhering to the general structure of PaintChat v0, but this did nothing to guarantee a common global order among all nodes in the cluster. In order to establish an absolute update order, we had to create our own consistency protocol.

The PaintChat requirements are somewhat unique in that they require both a strict degree of consistency as well as a prioritization of latency. It was immediately apparent to us that we wouldn't be able to meet our latency goals using a hierarchical structure like that the initial implementation. We knew we would need some kind of decentralized entity determining update order, and we knew that we would most likely have to settle for a system that was eventually consistent. We also wanted to avoid creating a central point of failure. These goals were realized using a combination of the ClusterSingleton and the Persistence modules.

ClusterSingleton is another great resource to come out of Akka Cluster [2]. It allows you to define an actor that will only ever be instantiated on a single node in a cluster at any given point in time. If that node were to fail, a new instance of the singleton will be created on a surviving node (assuming there is one available). Actors

[2]In later iterations of PaintChat v2 we actually ended up phasing a lot of responsibility out of our ClusterListener, favoring more specific extensions within the Cluster package to do the heavy lifting.

[3]Though this was extremely quick and easy, in hindsight it may have been *too* easy. Though it greatly reduces the logical complexity of message flow within the cluster, we suspect the mediator actors may be a bottleneck in our system.

can communicate with the singleton via a `ClusterSingletonProxy` actor that can be optionally instantiated on every node. All messages sent to the proxy will be routed to the singleton regardless of its location in the cluster, resulting in a simple and constant means of contact.

Akka's **Persistence** library offers a means of recording an actor's state, and recovering it upon failure [6]. This is accomplished with event sourcing [5]. Event sourcing describes the pattern of defining certain incoming messages as events, and persisting them to a entity called a *journal*. These events can be retrieved and replayed to the persistent actor during recovery.

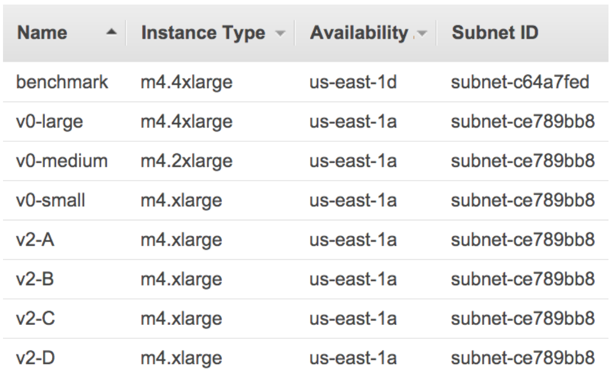
Journals are intended to be self-contained elements external of the actor system, and usually store data to a non-volatile medium. Unfortunately, Akka doesn't provide an official solution for a production-grade journal, so developers are left to pick from a large list on the community projects page. For our journal platform we selected Cassandra, a distributed and replicated database. We picked Cassandra for its clustered architecture in an attempt to avoid a central point of failure, and for its impressive benchmarks which we hoped would help us shave precious milliseconds from our latency numbers [9].

With a persistent ClusterSingleton subscribed to the `client_update` topic we created a means of establishing a global event ordering accessible from anywhere in the cluster, even in the presence of node failure. We used this actor to implement a rudimentary version of a corrective technique called Read Repair, which involves performing error correction of data upon it being read. For us the data was the update buffer cache kept by the ServerWorkers, which was read every time the ServerWorker accepted a new client WebSocket connection. Given more time we would be interested in extending this into something similar to Riak's Active Anti-Entropy protocol, fixing ordering errors before they are read and actively propagating corrections to the client [1].

3. EXPERIMENTAL SETUP

3.1 Benchmark

We set up a benchmark to measure the latency of the PaintChat service and checking for the consistency. For measuring the latency, we compared the the latency in different configuration settings. Basically we are interested in comparing the performance of single-node implementation and cluster implementation. There are three configuration settings in our experiment: 1) a small-size single-node implementation and a single-node cluster implementation; 2) a medium-size single-node implementation and a two-node cluster implementation; 3) a large-size single-node implementation and a four-node cluster implementation. The single server node is implemented without using any cluster specification provided by Akka. The latency would increase when many concurrent PAINT request are sending to the server. The cluster setting here implemented the service in the cluster mode by using the cluster specification provided by Akka. This would involves with some extra overhead of message broadcasting to keep consistency. In each configuration, we kept the capacity of the hardware resource the same over the two different implementations. In this way, we could compare how cluster implementation could affect the performance.



Name	Instance Type	Availability	Subnet ID
benchmark	m4.4xlarge	us-east-1d	subnet-c64a7fed
v0-large	m4.4xlarge	us-east-1a	subnet-ce789bb8
v0-medium	m4.2xlarge	us-east-1a	subnet-ce789bb8
v0-small	m4.xlarge	us-east-1a	subnet-ce789bb8
v2-A	m4.xlarge	us-east-1a	subnet-ce789bb8
v2-B	m4.xlarge	us-east-1a	subnet-ce789bb8
v2-C	m4.xlarge	us-east-1a	subnet-ce789bb8
v2-D	m4.xlarge	us-east-1a	subnet-ce789bb8

Figure 3: EC2 instances provisioned for the purpose of benchmarking the performance of the cluster.

The basic idea of the test bench is by creating certain number of clients that connected to the single server or Amazon DNS server. In our case, we would create 10 and 100 clients connection respectively. Each client would send out 10 or 100 different PAINT message to the server, as shown in the Table 1.The objectivity of changing the number of client connections and the messages number at the same time is that we want to see how does the number of clients and messages affect the performance respectively. The response latency are expected to increase when more and more request are flooding into the same server node. In order to run the benchmark to get the latency, we need to first set up a server node in the Amazon EC2 cluster. At first we tried to run the benchmark remotely in the local computer by sending messages to the server node in the Amazon EC2, it turned out that certain number of the packages would be dropped due to the network connectivity and makes it harder to finish the test-bench. So we move to run the test-bench in another node in the Amazon EC2, in which way we could get much better delivery of the packages.

For the consistency checking in the cluster mode, we create a url endpoint that could be used to retrieve the content in the paint buffer.When checking with consistency, we simply create a client instance for each existing node in the cluster and retrieve the paint buffer by using the url endpoint. We checked the consistency

Table 1: Benchmark Parameter Configurations

Clients	Msgs/Clients	Testing nodes			
10	10	v0sml	v2-1	v2-2	v2-4
10	100	v0sml	v2-1	v2-2	v2-4
100	10	v0sml	v2-1	v2-2	v2-4
100	100	v0sml	v2-1	v2-2	v2-4

**v0sml*: Single node implementation with small/medium/large server instances *v2-k*: Cluster implementation with k nodes setting up

here with a binary checking, whether or not it's consistent. Native method here is to compare the content of the buffer for each server, and if all of them are the same then the data is consistent in all the servers. Any differences in the ordering would be detected as inconsistent.

4. RESULTS

In the experiment setup, we compared the package latency of both the single-node implementation and cluster implementation while keeping the capacity of resources the same in the two settings. For example, the hardware resources in a single large instance is equal to that of a clustered four-node setting. The objectivity is to see if the the cluster implementation could get obtain better performance than the single-node implementation when feeding with same size of messages flow.

As we might expected, when feeding the server with more messages, the response latency would increase, as we could observe from Figure4-Figure6. Another observation from the figures is that the increment of the client connections would definitely bring more overhead than simply increasing the number of message packages. To be specifically, a new client connection would bring up a new execution context in the system, which would in fact increasing clock cycles spending on context switching.

In our experimental results, the performance of the cluster setting didn't actually output perform that of a

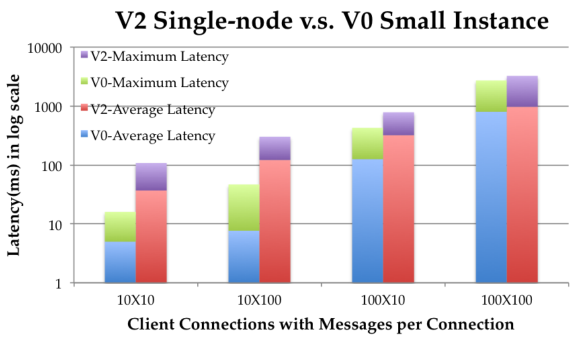


Figure 4: Latency results with v2 one-node setting and v0 small-instance

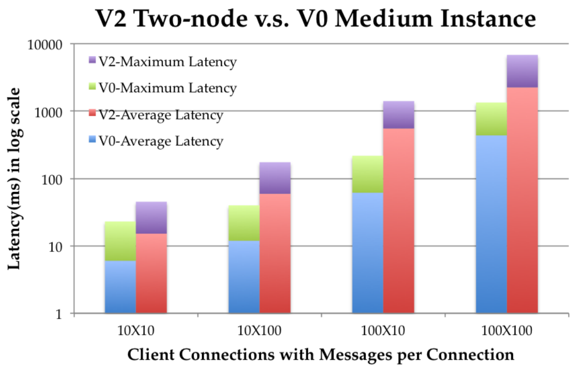


Figure 5: Latency results with v2 two-node setting and v0 medium-instance

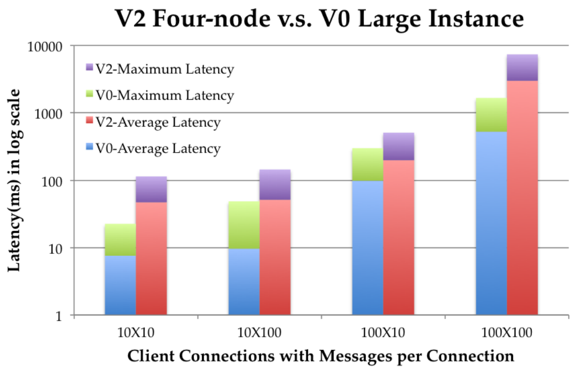


Figure 6: Latency results with v2 four-node setting and v0 large-instance

single server implementation without adding any clustering support in all of three experiment settings. As we could see in the Figure 4,the latency of the cluster implementation is almost twice as that of the single-node implementation, while this gap is narrowing down to 1.2X times when increasing the client connections.Same cases happened for the other two settings. The explanation here might be that the overhead of context switching didn't get exposed completely in the clustering setting. In the cluster setting, when the number of client connections increase, the connections would be allocated evenly to the nodes in the cluster, in which way the overhead of context switching got amortized by the number of nodes in the server.

Compare the implementation of single-node server and the cluster, we tried to figure the cause of the worse performance of the cluster. Here is some of our assumptions: 1) The implementation complexity adds more clock cycles. There is no doubt that the implementation of cluster node would involved with more execution actors and thus become more complex. As we said above, context switching across the actors would increase the latency; 2) The mediator in the cluster node become a bottleneck. One of the key differences of cluster implementation and single-node implementation is that the messages might need to be forwarded to other nodes in the cluster through the mediator in the server. The mediator would process with the client request in a FIFO order. But before sent back the package the local server worker, the mediator would first forward the packages to other nodes in the cluster. So the extra latency here would the *waiting time for the mediator buffer* plus with the *waiting time for mediator forwarding*. Theoretically, this extra latency would increase when the size of the cluster is scaling up.

5. CONCLUSION

In this study, we designed and developed an actual distributed system by extending the PaintChat application into a functionally equivalent clustered version. Besides distributing the PaintChat service over Amazon EC2, we also put efforts on improving the reliability and consistency of the system by persisting crucial events to distributed Cassandra instance. For performance evaluation, we evaluated the response latency of the distributed PaintChat by creating benchmarks to simulate client behavior and measure the message latency. We compare the performance result with the initial implementation of PaintChat. The experiment result indicates that the distributed version of Paintchat actually degraded the performance with the same hardware resource budget.The performance is degraded around 20%-50% in all the experimental settings.

6. FUTURE WORK

This project brought up some insights for the performance of a distributed service. There are many experiments that we could run to get some more interesting results. For instance, we could scale up the cluster size with more hardware resources allocated, to see how that affect the performance of the distributed service. Though we have implemented a rudimentary read-repair mechanism, some optimization techniques could be applied to further improve the performance. For example, we could try to implement mechanism that could persist consistency even before the read operation happen; Or we could try to implement a more efficient method to update buffer rather than throwing the entire buffer to the network.

Another interesting area to explore is the implementation of the distributed version. For simular functionality, Akka framework might provide different implementation methods, each of which might require different levels of developing efforts and bring different performance overhead. We are interesting to see if there are other implementation methods that could reduce the performance overhead and potentially outperform the performance of the singleton version.

7. REFERENCES

[1] Active anti-entropy. `http://docs.basho.com/riak/latest/theory/concepts/aae/`, May 2015.

[2] Cluster singleton. `http://doc.akka.io/docs/akka/2.4.1/scala/cluster-singleton.html`, May 2015.

[3] Cluster specification. `http://doc.akka.io/docs/akka/2.4.1/common/cluster.html`, May 2015.

[4] Distributed publish subscribe in cluster. `http://doc.akka.io/docs/akka/2.4.1/scala/distributed-pub-sub.html`, May 2015.

[5] Event sourcing pattern. `https://msdn.microsoft.com/en-us/library/dn589792.aspx`, May 2015.

[6] Persistence. `http://doc.akka.io/docs/akka/2.4.1/scala/persistence.html`, May 2015.

[7] K. Chabreck. Paintchat. `https://github.com/kevinchabreck/paintchat`, May 2015.

[8] K. Chabreck and D. Liu. Improving concurrent server performance with akka framework. Duke University, May 2015.

[9] A. Cockcroft and D. Sheahan. Benchmarking cassandra scalability on aws - over a million writes per second. `http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html`, May 2015.