Design, Prototype Implementation, and Comparison of Scalable Web-Push Architectures on Amazon Web Services Using the Actor Model

Hussachai Puripunpinyo
Print and Recognition Services
Workday
Pleasanton, CA, USA
hussachai.puripunpinyo@workday.com

M.H. Samadzadeh
Computer Science Department
Oklahoma State University
Stillwater, OK, USA
samad@cs.okstate.edu

***Abstract*— Push technology has become an essential part of many web applications that require near real-time notification. The Hypertext Transfer Protocol (HTTP) is widely used for exchanging information between clients and servers. The growth of the Internet of Things (IoT) and the expansion/penetration of Internet access to new areas have resulted in the ever-rapidly increasing number of Internet users. Consequently, scalability has become one of the key requirements of modern web applications. One of the challenging problems of scalability is scaling out web-push applications since they are constrained by unshareable resources and their stateful nature. Amazon Web Services (AWS) is the leading cloud service provider that offers a variety of services. Web-push applications can leverage the services that AWS provides. Developers can build their entire service stack on Amazon Elastic Compute Cloud (EC2) if other services cannot fulfill their business requirements. To enable the scalability feature, a centralized message mediator is needed to orchestrate and coordinate messages across the web-push nodes. The publish-subscribe pattern should be implemented from upstream to downstream in order to optimize the flow of data. This paper examines the technologies that could be used as building blocks to create scalable web-push applications, and proposes three architectures using three different messaging services, i.e., Amazon Simple Notification Service (SNS), Redis Pub/Sub, and Apache ActiveMQ. A couple of push and pull models as well as an adaptive push/pull model are introduced and experimentally compared in this paper. The actor model, on account of its being a robust model, is used for handling concurrency and for coordinating the flow of messages between a messaging server and a web client. We created a chat room application to evaluate the architectures with different criteria and conditions, and the push technology used our experimentation was WebSocket. From our experiment, we found that Amazon SNS performed relatively poorly compare to the other two due to the overhead of web services calls. The architectures using Redis Pub/Sub and ActiveMQ performed equally well even though the ActiveMQ server took more time to process messages. We acknowledge that there are a large number of factors involved in and potentially affecting the performance, and that the response time is merely one of the properties that ought to be considered in making a design decision.**

Keywords- Push Technology; Real-Time Notification; Amazon Web Services; AWS; Publish-Subscribe; Amazon SNS; Redis; ActiveMQ; Push Model; Pull Model; WebSocket;

I. INTRODUCTION

Hypertext Transfer Protocol (HTTP) was initially designed to be generic and stateless. Generally, a web browser does not keep an HTTP connection open once it gets a resource from a server. Typically, an HTTP request is initiated by a web browser, and then a web server returns an HTTP response that corresponds to the address and content of the request. However, in certain applications, a server is required to send data to a web browser without an initial request. Examples include notifications, page updates, and chats, among many others. Before the coming of HTML 5, there were several techniques that have been used to implement a push application, e.g., Comet, Pushlet, and third party browser plugins. Server-Sent Events (SSE) and the WebSocket have been introduced in HTML 5. The former is a simplex protocol that operates on HTTP, while the latter is a full-duplex protocol that operates on TCP directly [1].

A web-push model resembles a publish-subscribe pattern where a publisher is a web server and a subscriber is a web client. A web-push technology requires a persistent connection between a server and a client to function. Generally, it is difficult to scale a persistent connection horizontally due to its stateful nature. The technique of scaling out web-push applications is rather different from scaling out web applications, which is based on a request-response model. They however have something in common, which is that the user information should be stored externally. A session ID and other information can be stored in an external database which all nodes in a cluster can access. In contrast, a network socket is a logical resource that is mapped to an underlying hardware; thus, it is specific to a machine and cannot be transferred or shared across machines. A web-push application has to maintain the mapping between a user ID and a network socket. In UNIX, a network socket is identified by a file descriptor [2] as well as other types of I/O resources such as a file and a terminal. The limit of the number of sockets depends on a system, but the actual limit might be the bandwidth and capability of underlying hardware, operating system, and infrastructure. In order to scale a web-push application, besides storing the states externally, two additional requirements need to be fulfilled. The first requirement is that a server needs an in-memory data structure to store the mapping of a user ID and the network socket. The other requirement is to use a message mediator that implements the publish-subscribe model for routing the flow of data to the corresponding users correctly and efficiently.

This paper focuses on building a scalable web-push application by leveraging AWS services. The actor framework is used for handling messages between clients and messaging servers. Some of the important components for building a scalable push-server application will be introduced in Section II. In Section III, we propose three different architectures using different messaging services: Amazon SNS, Redis Pub/Sub, and Apache ActiveMQ. The experimental results of different load testing scenarios on each architecture are discussed in Section IV. Finally, Section V concludes this paper and outlines some possible areas of future work.

II. MESSAGE MEDIATORS

A. *Cloud Message Services*

AWS is a leading cloud service provider that is operated by Amazon. It is categorized as Infrastructure as a Service (IaaS) since users have full control over their rental virtual machines. Besides providing a compute engine called Amazon Elastic Compute Cloud (EC2), AWS provides many services that facilitate the common use-cases in software development such as database, queue, service monitoring, notification, email, and many others [3]. More importantly, most of those services are auto-scalable; therefore, developers do not have to concern themselves with the work load since the scaling is managed by the cloud provider.

1) Amazon Simple Notification Service (SNS)

Amazon SNS is a push notification service that supports many different protocols such as HTTP, HTTPS, Email, SMS, as well as protocols for other Amazon services [4]. This publish-subscribe service provides unified APIs that can be used with different protocols seamlessly. Developers can incorporate this service into their web-push architectures. The major advantages of using the AWS service over running services on one's own servers can be described as follows. 1) Scalability – The service can handle massive loads since resources and bandwidths are not limited by a specific amount that the provider allocates, rather by the capability of the provider itself. 2) Availability – Hardware and software are being monitored 24x7 by AWS. 3) Flexibility – The protocols that Amazon SNS supports cover almost all common use cases. Despite the many advantages that Amazon SNS has, developers may find it is more suitable and beneficial to use other approaches as we will discuss in more detail in Sections III and IV.

2) Amazon ElastiCache

Amazon ElastiCache is a web service that abstracts the underlying cache system to make it easy to use and scale since it is fully managed by the cloud provider. Developers can choose either Memcached or Redis as the underlying cache system [5]. Nonetheless, Redis implements the publish-subscribe messaging paradigm, called Pub/Sub, while Memcached does not.

The Redis Pub/Sub protocol is stateless, and the messages are delivered in a fire-and-forget fashion, which means the server does not expect any responses back from subscribers. The protocol is simple and has very little overhead. To subscribe to a channel, the protocol starts with "SUBSCRIBE" followed by a channel name. After the server receives this command, the client will be subscribed under the specified channel name [6]. When Redis Pub/Sub is used in a cluster mode, the node receiving the "PUBLISH" command broadcasts a message to the other nodes as depicted in Fig. 1 [7]. This model may lead to a bandwidth saturation situation when there are too many messages sending and receiving between nodes in the same cluster. As a result, adding more nodes may have an adverse effect when the network bandwidth reaches a limit and becomes a bottle neck in the pipeline.

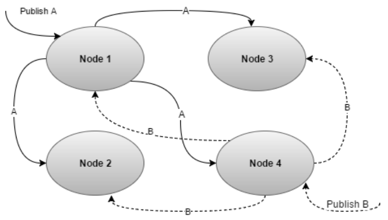

Figure 1. Messages propagation in Redis Pub/Sub cluster [7].

B. *Message Oriented Middleware (MOM)*

MOM is a system that provides a method of communication between distributed systems [8]. The communication is based on an asynchronous interaction model which allows a sender to continue processing once a message has been received by MOM, and the receiver neither has to consume the message nor be ready for a message [8].

There are many MOM implementations in many languages, the one that will be used in this paper is ActiveMQ which is the popular open-source MOM that complies with the Java Message Service (JMS) specification [9]. It supports a variety of cross language clients, messaging protocols, transport protocols, enterprise integration pattern, and many other API integrations. JMS provides two communication models: Queue and Topic [10]. JMS Queue is a point-to-point communication that implements the load balancer semantics while JMS Topic implements the publish-subscribe semantics (see Fig. 2).

The model to retrieve data, i.e., whether it is a push or a pull model, depends on the implementation. ActiveMQ uses a pull model along with the long polling technique to mitigate the busy-waiting issue. While a push model is considered to be more responsive, a pull model has many advantages such as the following items. (a) It is difficult for a push-based system to control the rate of data transfer because of diverse consumers, and it may cause a buffer overflow in a fast-producer-slow-consumer scenario. A pull-based system lets consumers pull the data at their speed, and it is easy to scale out the messaging system to maintain the quality of service. (b) It is more effective to retrieve data in batch for a pull model. A push-based system has to make a decision whether to push data immediately or accumulate it to a certain number before sending. A wrong choice may lead to a situation of overwhelming the consumers as mentioned previously.

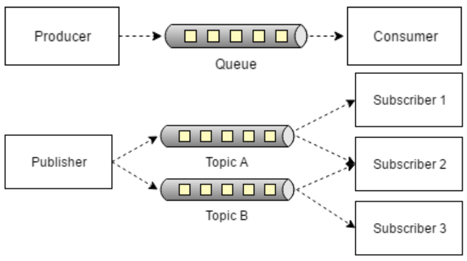

Figure 2. A Queue model (above) and a topic model (below) [8].

C. *Actor Framework*

The actor model is a programming model for handling concurrency. It focuses on message passing and has the following properties: encapsulation of states, fair scheduling, location transparency, and mobility [11]. This model has gained more popularity because of the success of the implementation in Erlang [12]. Subsequently, many implementations emerged in many languages including the most popular actor framework for Java Virtual Machine (JVM), called the Akka toolkit, which we will use as an important component in Section III. The actor model is a more preferred approach to coping with the concurrency problem than using threads and locks directly because a) locks are not composable, b) locks are easy to be misplaced and overused, c) locks may introduce deadlocks and race conditions, and d) error recovery is difficult when using locks [12].

To use the actor model as a message coordinator in architectures, a dedicated actor will be assigned to each network connection in order to receive and respond to message events. It is worth mentioning that an actor has its own mailbox [12] which is a message queue. The mailbox can buffer messages when the actor receives messages at a faster rate than it can process.

D. *Event Bus*

Event Bus implements a publish-subscribe pattern, and it does not provide any data persistence mechanisms. There are some use cases where data persistence for an individual item is not necessary such as resource monitoring, clickstream, and stock price. The data are not bound to specific items, rather to the channels they are being streamed to. Users which subscribe to channels will not see data in the past since messages are sent out in a fire-and-forget fashion unless some sort of persistent mechanisms is implemented.

Event Bus can be used as a primary message coordinator of a web-push application when persistence and durability are not the main concerns. Some frameworks, such as Vert.x, Redis, and Akka, support a distributed Event Bus while some other frameworks, such as Guava, do not. An in-memory Event Bus, it can be used with other distributed message mediators for mapping a message channel to network connections. This use case will be demonstrated in Section III when an Event Bus is used together with Amazon SNS.

E. *Reactive Streams And Back-Pressure*

A buffer is often used in a consumer to prevent a fast producer from overrunning its capability. The problem is that a producer does not know the status of a consumer. So, a producer may keep sending data at a higher speed than a consumer can consume, thus eventually filling the bounded buffer. The excessive data will be lost if there is no alternative data handling mechanism available. When the buffer is full, and a message requires an acknowledgement to confirm a successful transmission, a producer may resend a message to a consumer. As a result, the flood of traffic can cause denial of service on a consumer if a producer keeps resending messages. Alternatively, a producer may notice the increasing number of failures and adjust the speed of transmission accordingly. However, this is not an efficient method to handle this issue because it does not prevent the failure from recurring, and, more importantly, the service might be disrupted.

Reactive streams specification was invented to mitigate the fast-producer-slow-consumer problem and improve the efficiency of transferring messages by providing a standard way for asynchronous stream processing with non-blocking back-pressure [13]. The back-pressure protocol is responsible for controlling the flow of the data stream where speed and stability are not predetermined [13]. Network transmission is a good example. A producer and a consumer have to work together in order to adjust the speed of the flow according to the current statuses from both sides in an asynchronous fashion. The mode of transmission will be determined whether it is pull or push at runtime, depending on how fast the downstream components can process. This method takes advantage of both modes and it is known as dynamic push/pull [14]. Reactive Streams is used for processing a potentially unbounded number of elements in sequence. The data pass between components in an asynchronous fashion with mandatory non-blocking back-pressure. The objective of this mechanism is to give consumers the power to control the pace of the stream they are consuming.

III. PROPOSED ARCHITECTURES

There are virtually a countless number of possible architectures that can bring scalability to web-push applications. This paper focuses on utilizing the services provided by AWS. The first two proposed architectures use the messaging services provided by AWS, and the third one uses Amazon EC2 to host a third-party messaging software. The controller component, which is part of the proposed architectures, is a web component that handles the flow of input/output.

For brevity, we will use the name of the messaging service to refer to the architecture using that service. For instance, Amazon SNS may (depending on a context) refer to the architecture using Amazon SNS as a message mediator.

We developed a chat room application for testing. The channel name refers to a chat room to which people who are in the room can send messages, and where they can also see messages from other people.

A. Cloud-Based Service Approach Using Amazon SNS

Amazon SNS is a push-based system [4]. A subscriber can be an HTTP endpoint that Amazon SNS can send messages to through a topic [4]. In order to subscribe to a topic, the controller has to make a request containing a topic name and an endpoint address to Amazon SNS. Amazon SNS then returns the topic's Amazon Resource Name (ARN), which is a unique identifier for the topic being created, back to the controller. If the topic already exists, the topic's ARN will be returned without creating a new topic [4]. Since the entity that makes a request for a topic creation does not have to be a subscriber, the process can be abused to initiate a distributed denial of service (DDoS) attack. That is why a subscriber needs to confirm a subscription in order to enable the topic, as depicted in Fig. 3.

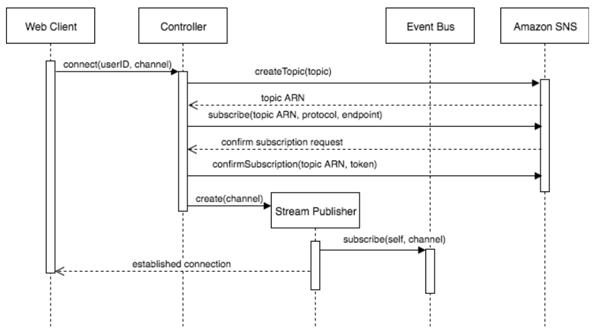

Figure 3. The subscription process on the serve-push architecture using Akka EventBus and Amazon SNS.

Stream publisher is a component provided by Akka Streams. It is used for keeping track of the subscription lifecycle and the requested elements [14]. This component also implements the back-pressure protocol [14]. Note that the subscription in this context means a network connection between a web client and a controller. Thus, it acts as an adapter that channels messages from Amazon SNS to a web client. The connection is ready when the stream publisher subscribes itself to the Event Bus.

The role of the Event Bus is to map messages received from Amazon SNS to the stream publisher that holds a network connection between a web client and a controller (web server). The in-memory Event Bus classifies messages, in this case by a channel name, and routes them to the subscribers. The subscriber of the Event Bus component is the stream publisher component. Fig. 4 provides a clear picture of how each component interacts when a push event occurs. A push event starts when Amazon SNS receives a message. The message then is forwarded to all subscribers of a specified topic, and a subscriber type is an HTTP endpoint in this case. After the controller receives a message through an HTTP POST request, it parses the message in JavaScript Object Notation (JSON) format and publishes to the Event Bus. The actor, which is subscribed to the specified channel of the Event Bus, will receive a message and will forward it through a TCP socket that has previously been bound to the stream publisher. A message can be either processed or transformed at any step depending on a decision at design time.

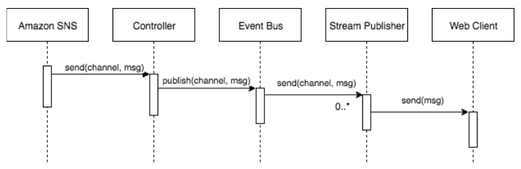

Figure 4. The process of publishing a message to a consumer (web browser)

Amazon SNS can push messages to the Amazon Simple Queue Service (SQS), which is a point-to-point protocol [4]. So, this architecture can be modified from being a push model to a pull model, and, as a consequence, the connection initiation phase will take more time because it would involve additional web service calls. Moreover, the architecture will be more complex.

B. Distributed Publish-Subscribe Approach Using Redis

Similar to Amazon SNS, Redis Pub/Sub is a push-based system [6]. Amazon ElastiCache manages all administrative tasks and hides the complexity of the configuration from developers. The Redis cluster should be accessed through the Amazon EC2 instance that resides in the same virtual private cloud (VPC) for security and performance purposes [5].

Due to the simplicity of the Redis protocol and the Redis client that integrated with an actor, the architecture becomes straightforward as shown in Fig. 5.

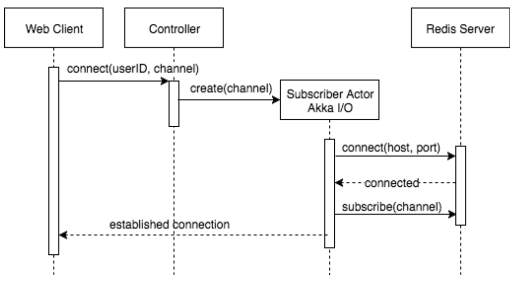

Figure 5. The subscription process of web-push architecture using Redis Pub/Sub.

The primary component in this architecture is the subscriber actor which is responsible for both coordinating messages and mapping the channel to the network connections. The subscriber actor component is implemented using Akka I/O and the network abstraction API which is event-driven, non-blocking, and asynchronous. A subscriber can subscribe to a single channel using the exact name or multiple channels using a regular expression.

C. *MOM Approach using ActiveMQ*

If Amazon SNS and Amazon ElastiCache do not fulfil the requirements, developers can use their choice of message orchestration software, and install it manually into Amazon EC2. We used ActiveMQ as a message mediator in this work, but it can be substituted easily with any MOMs that complies with JMS. As depicted in Fig. 6, the flow starts when the web client sends a request to the controller. The controller then looks up for the topic-poller actor. If the actor does not exist, the controller will create a new one. Once the topic-poller actor is created, it creates a session which is a context for producing and consuming messages [10]. The session is a lightweight object and not thread-safe [10], thereby encapsulating the session inside the actor guarantees that the single-threaded session will not be accessed concurrently. The topic can be created using a session object. This action is idempotent which means it will not create a new one if the topic already exists [10]. The consumer is the object that is used for retrieving messages in JMS. After the consumer is created, the topic-poller actor will continue polling messages from the messaging server until either the actor or the destination topic is destroyed.

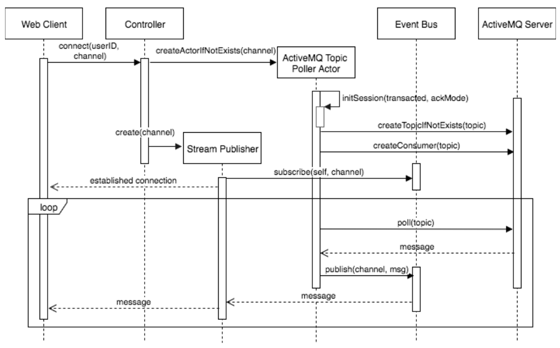

Figure 6. The subscription process of web-push architecture using ActiveMQ.

ActiveMQ client uses a pull model to retrieve data from a message broker on the server side. The loop in the diagram (see Fig. 6) can be implemented using a message passing model, rather than a traditional loop, in order to free a thread and give the opportunity to the other actors to run. The ActiveMQ client supports a long polling technique which makes it fit well with the message passing approach.

Once the actor gets messages from a topic, it publishes those messages to the EventBus component that propagates them to the corresponding subscriber, which is the stream publisher component. EventBus pushes messages directly to the stream publisher via a message passing mechanism. The stream publisher then forwards the messages to a WebSocket connection to which it is registered. Creating a new actor object is an asynchronous operation, which means that the controller can continue doing something else because the actor is running in another thread.

This architecture is more complex than the other two because of the pull model. The stream publisher is the actor that is mapped one-to-one with a WebSocket connection. If it is responsible for polling messages, the number of actors polling messages will be increased at the same rate as the number of connections. Since the polling operation blocks a thread, having many of them will eventually block all active threads and result in the system becoming unresponsive. The global topic-poller actor and Event Bus are introduced to minimize the number of blocking operations.

IV. EXPERIMENTAL DATA AND RESULTS

A. *Software Used and Experimental Factors*

The experiment was performed on Amazon EC2 C4.XLarge instance which has 4 virtual CPUs, 7.5 GB of memory, and high network performance [15]. Despite the architecture and hardware, the software and its versions can affect the benchmark result. Table I shows some details of the software that we used in our implementation.

TABLE I. SOFTWARE AND VERSION USED IN THE EXPERIMENTS

Software	Description	Version
Scala	Language and runtime	2.11.8
Play Framework	Web framework	2.5.12
Akka Toolkit	Actor model implementation	2.4.14
AWS Java SDK	A client for Amazon SNS	1.11.86
Rediscala	Reactive Redis client library	1.8.0
Redis Server	A key-value store database that supports publish-subscribe.	3.2.6
ActiveMQ	MOM that complies with JMS	5.14.3
Async HTTP Client	Asynchronous and non-blocking I/O HTTP client.	2.0.27

We created a load generator using the async-http-client library to test and collect the result because we found that web browsers did not provide reliable results due to the interference of background processes. Furthermore, we could not control the creation of threads. Simultaneous firing relies solely on a single-threaded non-blocking I/O event loop.

There are two measurements that we use throughout the experiments: round-trip time (RTT) and message turnaround time (MTT). RTT is the length of time it takes for a message to be sent by a client plus the length of time it takes for that message to be received by a client (it does not have to be the same client). Message turnaround time (MTT) is the turnaround time of a message between a web server and a messaging server. The push technology used in our experiments is WebSocket.

B. *Evaluation of Performance Using Low Concurrent Setting*

The first case is to have 4 threads sending 200 requests sequentially to a web server and capturing RTT and MTT. Each thread will not send the next request until it receives back a message belonging to a previous request, which means that there are four potentially concurrent requests in the same millisecond. Table II shows the result of each thread, and it suggests that all threads are being processed fairly since there is no one thread dominating others. This result conforms to the fairness property of an actor model.

TABLE II. MEASURED TIME OF 4 CONCURRENT THREADS WHERE EACH THREAD SENDS 200 MESSAGES SEQUENTIALLY TO AMAZON SNS

Thread	RTT		MTT	
	Avg. (ms)	**Std. Dev. (ms)**	**Avg. (ms)**	**Std. Dev. (ms)**
1	133.76	37.53	109.63	33.70
2	130.92	35.52	107.29	32.02
3	135.20	37.54	111.78	32.73
4	132.22	32.55	108.22	27.63

The average RTT of the Amazon SNS approach is considered fast compared to the duration of a blink of an eye which lasts 100-400 ms on average [16]. As is characteristic of the actor model, a message is processed by one thread at a time which guarantees that no concurrent modifications can occur. Of course, that does not imply that the actor is bound to only one thread, even though it can be configured in that manner, it just means that a message can be processed by any threads in a thread pool.

Before measuring the performance in a highly concurrent setting, it is important to see how each architecture performs under a low concurrency setting. The sudden drop of network speed and other factors might affect RTT and MTT; therefore, testing on a few messages could yield an unreliable result. We tested by sending 200 messages sequentially to each HTTP endpoint that represents a different architecture. Table III shows the average and standard deviation of RTT and MTT for all three architectures. The average MTT of Amazon SNS is relatively high compared to the other two mainly because of the overhead of HTTP, request signing, authentication, and authorization. In addition, the public cloud service cannot guarantee a stable result as we can see that the standard deviation is 28.85% of the average time. Another factor that might affect this result is the AWS Java SDK that we used. Even though the standard deviation of Redis Pub/Sub and ActiveMQ are also high compared to their respected average MTTs, the numbers are small enough to be neglected due to the unreliable network latency.

According to Table III, Redis Pub/Sub performs better than the other two architectures. The light weight of the protocol along with the push model makes Redis Pub/Sub efficient in propagating messages in this particular case. The average RTT is corresponding to the average MTT in this experiment. Message latency is calculated by subtracting RTT by MTT, and we have 23.80, 11.36, and 6.68 ms for Amazon SNS, Redis Pub/Sub, and ActiveMQ, respectively. Message latency may give us a hint about the capability of the architecture in delivering messages to clients. However, this indicator may be inaccurate when blocking I/O operations are involved.

TABLE III. COMPARISON OF AMAZON SNS, REDIS PUB/SUB, AND ACTIVEMQ ON 4 CONCURRENT THREADS AND EACH THREAD SENDS 200 MESSAGES SEQUENTIALLY.

Architecture	RTT		MTT	
	Avg. (ms)	**Std. Dev. (ms)**	**Avg. (ms)**	**Std. Dev. (ms)**
Amazon SNS	133.03	35.79	109.23	31.52
Redis Pub/Sub	29.08	17.72	1.48	0.77
ActiveMQ	39.32	32.64	3.19	1.38

C. *Evaluation of Performance Using High Concurrent Setting*

In this evaluation, we sent 600 messages to a web server concurrently, and had 3 WebSocket connections to receive the responses back at 200 messages per connection. Fig. 9 shows the benchmark results of Amazon SNS, Redis Pub/Sub, and ActiveMQ from left to right. The baseline is the result of sending 200 concurrent requests and receiving all responses back by a single WebSocket connection. It is interesting that ActiveMQ performed better than Redis Pub/Sub when the number of concurrent messages is increased by 150 times despite having higher MTTs (see Fig. 9c, d, e, and f). There are at least two factors that make ActiveMQ outperform Redis Pub/Sub in certain cases. First, the stream publisher component implements the reactive streams protocol which is an adaptive system that adjusts the transfer rate dynamically. Second, ActiveMQ uses a pull model to retrieve messages, and this model is more efficient than a push model in batch fashion. We may infer that a chance of having more than one message in a queue is proportional to the degree of concurrency.

From Fig. 9a and Fig. 9b, the average RTT and MTT of Amazon SNS are 12,562.61 and 10,156.1 ms, respectively. The bottleneck is obviously the messaging service. There are also other overheads involved in this process such as the HTTP protocol, JSON message parsing, and the blocking I/O used by the HTTP client. Amazon SNS is a cost effective solution as a fee is calculated based on usage plus a generous monthly free-tier quota [4]. Developers do not have to rent a virtual machine that costs hourly even when there is no traffic, but the hidden cost may come from the service latency considering the fact that web applications may have to scale out in order to maintain an acceptable response time.

D. *Finding the Optimum Number of Concurrent Requests*

We have tested the systems in both low and high concurrent settings. This evaluation was carried out to find the optimum degree of concurrent requests that the architectures can handle in an acceptable time frame. The acceptable value is subjective; thus, we present a series of results in Fig. 7 and Fig. 8.

We can portray the information from Fig. 7 as the scenario of a chat room. Suppose we have 400 people in a chat room sending messages simultaneously. They will notice that the delay of their messages is about 7.5, 3, and 2.5 seconds for Amazon SNS, Redis Pub/Sub, and ActiveMQ, respectively. If the expected response time is 2 seconds, the number of concurrent requests should not be more than 200 for Redis Pub/Sub and ActiveMQ and 100 for Amazon SNS. We also found that increasing the number of WebSocket connections to 4 does not have any significant impact on the architectures as shown in Fig. 8. The actor runs in a single thread, and we may assume that having 4 actors running on 4 threads may gain more performance. In this case, it seems like a thread pool is already utilized.

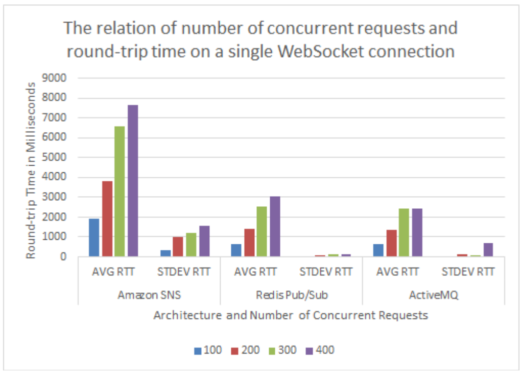

Figure 7. Comparison of RTTs among architectures on a single WebSocket connection.

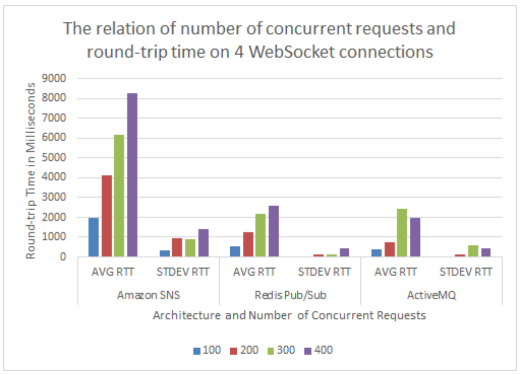

Figure 8. Comparison of RTTs among architectures on 4 WebSocket connections.

V. CONCLUSION

A regular web application can be scaled up by storing states in a shared distributed data store. So, users can access any node in a cluster and see the same data. A web-push application is a web application that has the push capability. The push technology relies on a persistent connection between a web client and a web server. Network connection is not a transferable resource. Thus, the architecture should be designed so that it supports different persistent connections linked to the same person. A publish-subscribe pattern should be applied from upstream to downstream in order to make the flow efficient. A centralized message mediator is needed to coordinate messages across nodes in a cluster. Developers can choose either push or pull depending on the situation, since the two models have non-identical advantages and disadvantages.

The actor model is an alternative approach for handling concurrency using asynchronous message passing. It is intuitive to use each actor representing one network connection, and let the actor system manage the lifecycle of the connection.

We evaluated three scalable web-push architectures on various conditions in order to make a comparison among them. Based on our observations, Amazon SNS is not a good choice for our use case except in certain situations, since its web service protocol seems to have a lot of overhead. Even though the Amazon SNS approach performed relatively poorly compared to the other approaches, its messaging server is more scalable and cost-effective. Nevertheless, a hidden cost may emerge from the application servers that need to be scaled out to meet the business requirement. We showed that the architecture using Amazon SNS for our particular case is not suitable for highly concurrent environments. We also provided two other architectures that are portable and can be run in any IaaS provider. The architecture using ActiveMQ performs slightly better than Redis Pub/Sub in the high concurrent settings despite having more MTT. Both Amazon SNS and Redis Pub/Sub are push-based system. As it is well-known, consumers might get overwhelmed if producers produce at a faster rate for a certain period. To avoid this issue, developers may consider an alternative approach. ActiveMQ is a MOM (Message Oriented Middleware) that complies with JMS. It has some advantages over the other two besides the fact that it uses a pull model. ActiveMQ supports transaction, persistence, and durability as well as many protocols and API integrations (these are in fact typical strengths of MOM). The alternative approach, after all, requires a lot of administrative tasks which make it less attractive. However, a number of configuration management and orchestration tools such as Puppet, Chef, and Ansible can automate the deployment and many other administrative tasks in a cluster environment.

This paper proposes a number of possible architectures that can be used for scaling out web-push applications on the AWS platform. There may obviously be better solutions, and a single architecture may not fit all use cases. The architecture that is most suitable does not have to be the fastest one, but it is the one that is suitable for a specific circumstance, for example, time to market, expertise of a team, or a particular business contract.

Further work in this area could be on evaluating the alternatives across cloud platforms using their native support. The Redis Pub/Sub architecture seems to have a limitation in its design, further investigation on quantifying its capabilities in different circumstances would be interesting. Lastly, the pull model system has been shown to be efficient in our experiment, but it has some disadvantages. One of those is that actors have to listen to too many topics. The long polling technique could improve the response time, but it is a blocking operation. Developers will have to find the optimal waiting time or endeavor to provide an adaptive mechanism in order to strike a balance between responsiveness and overall system performance, since either one affects the other.

References

[1] V. Wang, F. Salim, and P. Moskovits, The definitive guide to HTML5 WebSocket. New York, NY: Apress, 2013.

[2] W. R. Stevens and S. A. Rago, Advanced programming in the UNIX environment. Ann Arbor, MI: Addison-Wesley, 2013.

[3] Amazon Web Services LLC, "Overview of Amazon Web Services," https://d0.awsstatic.com/whitepapers/aws-overview.pdf, [2017-01-13]

[4] Amazon Web Services LLC, "Amazon Simple Notification Service," https://aws.amazon.com/sns, [2017-01-13]

[5] Amazon Web Services LLC, "Amazon ElastiCache," https://aws.amazon.com/elasticache, [2017-01-13]

[6] Redis.io, "Pub/Sub," https://redis.io/topics/pubsub, [2017-01-13]

[7] Redis.io, "Redis cluster specification," https://redis.io/topics/cluster-spec, [2017-01-13]

[8] Q. Mahmoud, Middleware for communications. Chichester, England: John Wiley & Sons, 2004.

[9] ActiveMQ. http://activemq.apache.org, [2017-01-13]

[10] Java Community Process. "JSR-000343 JavaTM Message Service 2.0, "https://jcp.org/aboutJava/communityprocess/final/jsr343/index.html, [2013-05-21]

[11] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the JVM platform: a comparative analysis," Proc. the 7th International Conference on Principles and Practice of Programming in Java, ACM, Aug. 2009. pp. 11-20, doi:10.1145/1596655.1596658

[12] N. Raychaudhuri, Scala in action, Shelter Island, NY: Manning Publications, 2013.

[13] Reactive Streams, https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.0/README.md#specification, [2015-04-29]

[14] Lightened Inc, "Akka Scala documentation release 2.4.16," http://doc.akka.io/docs/akka/2.4/AkkaScala.pdf, [2016-12-19]

[15] Amazon Web Services LLC, "Amazon EC2 Instance Types," https://aws.amazon.com/ec2/instance-types, [2017-01-13]

[16] H. R. Schiffman, Sensation and perception: an integrated approach 5th edition. New York, NY: John Wiley & Sons, 2001.

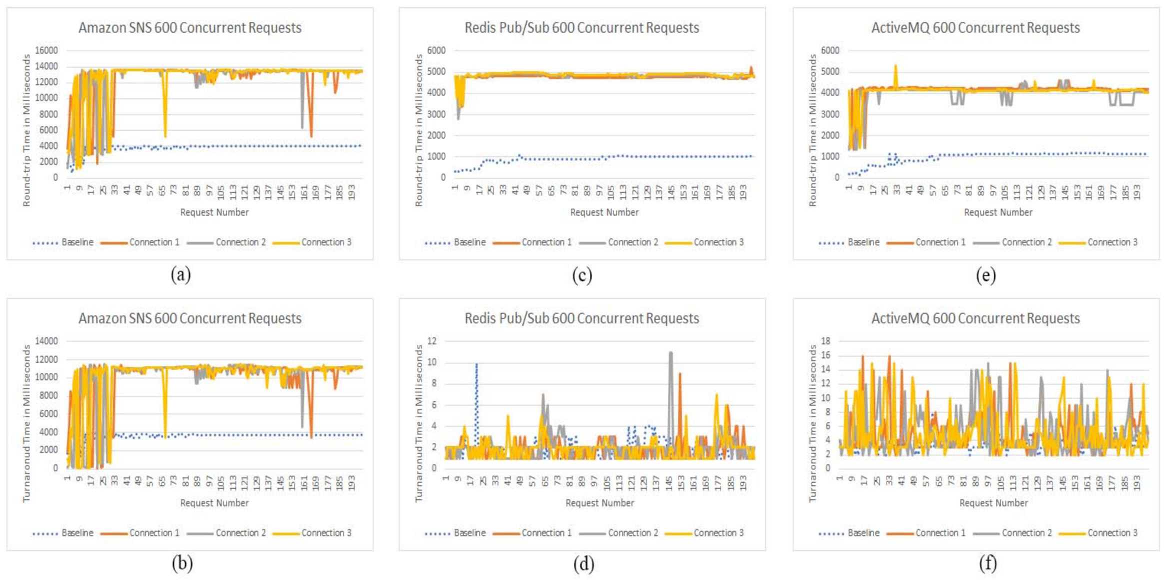

Figure 9. Evaluation of performance on 600 concurrent requests and 3 WebSocket connections using Amazon SNS.