UNIVERSIDADE DO VALE DO RIO DOS SINOS - UNISINOS
UNIDADE ACADÊMICA GRADUAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RAFAEL AUGUSTO EYNG

PUSH STREAM AS A SERVICE: A Server Push Service Implementation for the Tsuru Platform

São Leopoldo
2019

RAFAEL AUGUSTO EYNG

PUSH STREAM AS A SERVICE: **A Server Push Service Implementation for the Tsuru Platform**

Artigo apresentado como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação pela Universidade do Vale do Rio dos Sinos - UNISINOS

Adviser: Prof. MSc. Ernesto Lindstaedt

São Leopoldo
2019

PUSH STREAM AS A SERVICE: A Server Push Service Implementation for the Tsuru Platform

Rafael Augusto Eyng [1]
Ernesto Lindstaedt [2]

Abstract: Cloud computing is a mature model of offering hardware and software components as services to clients, and constitutes a solid foundation for applications to be developed upon. Modern applications have performance, scalability and functionality demands that can largely benefit from the cloud ecosystem, where an application can be thought as a set of services and components working cooperatively to deliver a feature set. Those components can be very generic, like computing nodes, network infrastructure and databases, or more specific and feature-oriented.

Several cloud computing systems were built to ease the adoption and usage of cloud computing by application developers. Tsuru is an open-source system for cloud computing, and offers the Platform-as-a-Service cloud service model. Tsuru provides a platform to run client applications, and a number of pluggable services that application developers can provision and integrate with the applications.

This work proposes an implementation of a cloud service for the Tsuru system to provide server push functionality. Server push is a programming model on client-server architectures where data is sent from the server to the client without an explicit request for that data. This functionality is usually found implemented at application level, coupling it with business and application logic, and bounding its scalability with the application. The model presented in this work decouples the server push functionality from the application, allowing independent management and scaling, as well as providing the functionality as a cloud service for the Tsuru platform, provisioning software and hardware components automatically for the application developer upon a call to the provisioner service.

Keywords: Cloud Service. Server Push. Cloud Computing. PaaS. Tsuru.

1 INTRODUCTION

Several applications currently available to both public and private audiences have aggressive scalability demands, and usually provide a feature-rich experience to their users, as “these applications are designed to have high user interactivity with low user-perceived latencies” (ESTEP, 2013). Cloud computing has brought a number of new possibilities to application developers (GARCIA-GOMEZ et al., 2012), backed by a diversity of cloud models and services that can be explored in order to enable developers to meet their application needs, either in performance, scalability or functionality.

In this scenario, the paradigm of service-oriented computing “becomes a common paradigm for developing applications in a cost-effective manner” (LA; HER; KIM, 2013), with the fundamental notion of reusing services across several applications.

There are several software and hardware components that are now offered as a service. On the literature can be found Database-as-a-Service, Storage-as-a-Service, Monitoring-as-a-

[1]Graduando em Ciência da Computação pela Unisinos. Email: rafaeleyng@gmail.com
[2]Orientador, Mestre em Ciência da Computação. Email: linds@unisinos.br

Service, to name a few (KÄCHELE et al., 2013; DUAN et al., 2015; SHARMA, 2015). The plurality is such that the "Everything-as-a-Service" (XaaS) trend can be identified, where everything that can viably be offered as a service can eventually be so (DUAN et al., 2015).

1.1 Objectives

1.1.1 General Objective

The present work objectives to implement and evaluate an open-source cloud service for the Tsuru PaaS system to provide server push functionality for applications developed to run on Tsuru's platform.

1.1.2 Specific Objectives

- develop the Push Service system, a micro-services based system that leverages the Nginx Push Stream (PUSH STREAM, n.d.) module to provide server push technology, with a horizontally scalable architecture to overcome limitations of the Push Steam module, while providing an extra set of features such as authentication of message producers, and configurable expiration of communication channels.

- develop the PushaaS system, a service that provisions Push Service instances upon service calls, allowing clients to use the service while abstracting its complexities of provisioning and configuration.

- integrate the PushaaS system to the Tsuru platform, allowing application developers on the Tsuru platform to integrate the Push Service with their applications by making calls to the Tsuru command-line interface (Tsuru CLI).

- evaluate the implementation in terms of flexibility of its architecture, adequacy to the Tsuru ecosystem, and convenience for application developers who want to integrate the service with client applications.

1.2 Delimitation

The scope and limitations of the present work will be presented now. This work is aimed to result in a fully functional implementation of the PushaaS system. Nonetheless, some limitations will be present:

- the Push Service system was designed to allow for viable horizontal scalability, but the present work does not provide means for actually performing sizing and scaling actions on Push Service instances.

- the PushaaS component provisions Push Service instances relying on an abstract concept of a provisioner. The system was designed to allow further development of multiple provisioners, allowing multiple infrastructure providers to be used to provide infrastructure resources. The initial implementation of this work only provides implementation for a provisioner that works with the Amazon Elastic Container Service (ECS) as the infrastructure provider.

- in order to better fit with real-world organizations needs, the Push Service supports authentication on API calls. Only HTTP basic authentication was implemented on this work.

In order to avoid software incompatibilities, this implementation is based on the following software versions:

- Tsuru (TSURU, n.d.) platform on the 1.6 or 1.7 version, and other compatible versions.
- the server push functionality is specifically provided using the open source Push Stream module for Nginx server. The module version is 0.5.4.
- software components developed on this work will be developed and built with the Go programming language on the 1.12 version, in order to result in small binaries with good runtime performance, features of the Go language.
- the Amazon ECS provisioner implemented on the PushaaS works with the launch type Fargate (FARGATE, n.d.), specifically using Docker containers. Docker containers are a convenient way of deploying applications, abstracting complexity from application developers, and Amazon ECS is a mature tool that provides the cluster infrastructure needed to run Docker containers. The Fargate launch type goes an extra step further by totally abstracting the underlying computational nodes required to run the cluster.

1.3 Organization of this Paper

Section 2 provides a theoretical framework as a foundation for several concepts and technologies that will be explored throughout the paper. Section 3 analyzes related work to the topics of cloud computing, PaaS and server push techniques. Section 4 exposes the methodological approach applied on this work.

Section 5 presents the system architecture and important implementation details for the proposed solution. Section 6 presents and discusses the results of the system evaluation. Section 7 finishes with a brief analysis of the implementation and achieved results, and section 8 discusses several aspects that could be explored and further developed on future work.

2 THEORETICAL FRAMEWORK

Several technologies are needed to implement the proposed solution. Here they are examined, divided in three main axis:

- tools that provide server push functionality.
- cloud computing systems that can be used to build the system upon.
- tools needed to build the proposed service's custom logic.

2.1 Server Push Functionality

Server push is a programming model where, in a client/server architecture, a server sends data to the client without an explicit client request for that data. Here the technology employed in providing this functionality is studied.

2.1.1 Nginx

Nginx is a “high-performance, low-footprint web server that follows the non-blocking, event-driven model” (MARINOS; WATSON; HANDLEY, 2014). Nginx can be compiled with the Push Stream module extension in order to provide the server push functionality, and expose the Nginx server to the application users. It is the Nginx (with the module enabled) that is responsible for actually pushing the content to the clients.

2.1.2 Push Stream module

The Push Stream module is an open-source piece of software that can be compiled together with Nginx in order to provide server push functionality.

It provides an endpoint for message producers and the infrastructure to deal with the message consumers, supporting several techniques for serving the message consumers. A browser-side Javascript library to connect to a "Push Stream"-enabled Nginx is also provided, featuring all the techniques supported server-side (Forever Iframe, Long Polling, Websockets and Server-Sent Events).

2.2 Cloud Computing Systems

This work proposes a cloud service, so it is necessary to run it on top of existing cloud systems. Here some technologies that might be employed to build the service will be analyzed.

2.2.1 Amazon Web Services

AWS is a feature-rich cloud platform that provides computing and networking functionality needed for the implementation, such as Virtual Private Clouds and Subnets, Access Control Rules, Virtual Machine instances and a Container Scheduling platform.

2.2.1.1 Amazon Elastic Container Service

Since this work proposes an "as a Service" functionality, it is necessary to provision new computational resources upon the creation of every new instance of the service, as well to remove it upon removal of a service instance. The proposed tool for this is Amazon Elastic Container Service (Amazon ECS).

2.2.2 PaaS

The cloud service proposed is meant to be provided to applications running on a PaaS, which is common service models that cloud providers offer. According to the NIST definition (MELL; GRANCE et al., 2011), under the PaaS cloud service model, “the capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure [...]”. So a “main goal of PaaS systems is to relieve users of the burden of managing underlying resources” (COSTACHE et al., 2017).

2.2.3 Tsuru

Tsuru is an "open source and polyglot platform for cloud computing" (VERGARA et al., 2014), developed by the brazilian media company Globo.com, since 2012. It is in fact a PaaS system, well suited for running web applications (although not limited to that), written in several programming languages, like Python, PHP, Node.js, Go, Ruby or Java (COSTACHE et al., 2017).

Tsuru supports the concept of services, which, inside Tsuru's nomenclature must be understood as a "well-defined API that Tsuru communicates with to provide extra functionality for applications" [3]. A Tsuru service is a software that exposes an HTTP API that implements a specification supplied by Tsuru's documentation, and is registered to a Tsuru target (a particular installation of Tsuru upon some infrastructure). After registering the service, a Tsuru user can issue command-line commands to interact with the service, like creating a new instance or binding it to an application. When this is done, the commands will be issued against the Tsuru target, and Tsuru itself will call the registered service so it can perform the actions needed to comply with the user's commands.

2.3 Tools Needed to Build The Proposed Service

Here are analyzed the tools needed to build the service to provide server push functionality, that will be deployed and evaluated on the cloud stack described in the previous topic.

2.3.1 The Go programming language

For the new software components that must be developed to fulfill this proposal the Go programming language is an adequate choice. The language is developed by Google, and aims at providing good performance and simple primitives for building concurrent systems.

As stated by Westrup e Pettersson (2014), Go is a compiled, performant "general purpose language which is especially popular for writing web servers and related tasks", which produces small binaries. Go is specially well suited for systems development, being used on several large projects, included some related to the present work, like Tsuru itself, Docker, Terraform, and also other well-known systems like Kubernetes [4] and etcd [5].

2.3.2 Redis

In order to achieve great flexibility and scalability on the push service instances, a pub/sub mechanism is proposed between the API that the message producer calls and the Nginx with the Push Stream module that the application user connects with to receive the server push.

Besides that, each instance of the push service will need to store a very limited set of information, so a storage system is needed. Redis, being a key-value storage and a highly-optimized pub/sub system (GASCON-SAMSON et al., 2015), provides both features.

[3]https://docs.tsuru.io/1.6/understanding/concepts.html#services
[4]https://github.com/kubernetes/kubernetes
[5]https://github.com/etcd-io/etcd

3 RELATED WORK

3.1 Systems That Implement Server Push Functionality

3.1.1 A RESTful web notification service

Lo et al. (2016) proposes a RESTful notification system using server push, attached to the application implementation, avoiding a separate middleware. The argument in avoiding this extra component is to reduce complexity and ease development, but should be noted that this paper doesn't consider the development and deployment on a cloud environment, where infrastructure components doesn't have the same burden in provision as in traditional environments. Besides that, coupling of server push functionality system to the application in this implementation doesn't allow reuse of the solution.

The authors discuss several server push techniques. Polling and long-polling are discarded because of resource consumption. WebSockets are considered but would require the server-side to be aware of two protocols (WebSockets, besides the HTTP already). The implementation was done used only Server-Sent Events. To scope notifications, the work proposes a criteria filtering.

3.1.2 Research on server push methods in web browser based instant messaging applications

Shuang e Kai (2013) conducts a study on using server push on instant messaging applications, exploring different techniques, like HTTP polling, HTTP long-polling, HTTP streaming and WebSockets. The four methods provided different experiences on the delivery efficiency. HTTP polling is easier to implement but either requires the usage of an interval between polls (which delays the message delivery), or makes unnecessary requests. HTTP long-polling, on the other hand, avoids the unnecessary requests and delivers the messages more timely, but requires extra implementation server side. HTTP streaming presents an improvement, specially under a high volume of messages, where several messages can be sent as response to a single request, but can have a number of different issues with buffering proxies and firewalls. All these methods rely on HTTP and suffer from the same issue of HTTP adding some weight to the messages, in the form of unnecessary headers data. WebSockets have features to answer all the problems, at the cost of being unsupported in some older browsers or networking environments.

Similarly to the previous work, this work implements the solution inside the application. As a future work, the author proposes to create a framework to unify the handling of all four methods.

3.2 Cloud Computing Systems

3.2.1 Framework for evaluating reusability of component-as-a-service

La, Her e Kim (2013) proposes a framework to evaluate the reusability of components offered as services.

The framework presents eight aspects that are further broken down into more granular objective metrics, allowing to determine a number between 0 and 1 to represent a component reusability. The considered aspects are modularity, adaptability, composability, commonality, comprehensibility, interface soundness, QoS assurance and publicity. The modularity aspect of an application, for instance, could be computed by evaluating the metrics of Cohesiveness of Services and Minimality of Dependency. Cohesiveness of Services is calculated from the relationship found in mapping of functionalities to components, where the ideal would be one functionality being provided by one component, yielding a metric of 1. Minimality of Dependency is found by subtracting from 1 the division of the number of the service's functions that rely on external services by the service's total number of functions, where a service with no external dependencies would also yield a metric of 1. The other aspects can also be objectively computed to reveal the component reusability.

3.2.2 Beyond IaaS and PaaS - An Extended Cloud Taxonomy for Computation, Storage and Networking

Kächele et al. (2013) analyzes the taxonomy of cloud services, proposing an extended and more uniform nomenclature for computational, networking and storage resources, dividing each resource type into different abstraction levels, and classifying based on "the topmost level that is fully managed by the cloud provider".

The paper proposes five layers of abstraction for computational resources, instead of the commonly seen division in IaaS, PaaS and SaaS. IaaS can be understood as node-based infrastructure and divided in two separate layers: Hardware-as-a-Service (HWaaS) offers bare hardware to the tenant, allowing the tenant control starting from the operating system kernel and boot loader level, while Operating System as a Service (OSaaS) allows the tenant the control over the daemons and processes, with a single-node view. PaaS doesn't provide the vision of a computational node, and can be divided in two separate layers: Runtime Environment as a Service (RaaS), where the tenant deploys his own application with full control over the application, and Framework as a Service (FaaS), where the tenant only deploys code to be invoked by the execution environment. SaaS "is the provisioning of entire applications as a resource", while all hardware and software is managed by the provider. The application can either be or not be accessible to clients.

3.2.3 An efficient multi-task PaaS cloud infrastructure based on docker and AWS ECS for application deployment

Tihfon et al. (2016) proposes a multi-task PaaS cloud infrastructure, developed on top of Amazon infrastructure. Developers using the service can build applications to run on Docker containers, and the proposed system uses Amazon EC2 Container Service to distribute the load on the cluster, and manages other services and load balancers.

3.2.4 Challenges for the comprehensive management of Cloud Services in a PaaS framework

Garcia-Gomez et al. (2012) describes the 4CaaSt project, a European Union-funded project aimed to develop a PaaS framework. The paper presents the concept of blueprints, which describe every application, component or service in terms of its various aspects like lifecycle and associated resources. The blueprinting approach introduces the usage of blueprint templates as a simplified method for provisioning cloud services. Software providers describe all relevant aspects of a service in a so-called source blueprint, that service providers can use to create a service offering. A service developer can rely on existing services by declaring this dependency on its own service's blueprint. This system also proposes a cloud marketplace which can offer several cloud services and provide them based on blueprints.

On elasticity, 4CaaSt provides support to execute PaaS deployments on different infrastructure components, unlike other PaaS offerings (Google App Engine, Amazon Beanstalk etc), which rely on homogenous infrastructure resources.

3.3 Cloud Services

3.3.1 Evolution of as-a-Service Era in Cloud

Sharma (2015) observes the paradigm shift toward cloud environments, in order to obtain robust and affordable services, noting a "rise to the accelerated evolution of 'as-a-Service' (aaS) framework in almost all the domains". The paper then lists 68 services found in literature being provided in the "as a Service" model. Some of the services that were found to be somewhat similar to the proposed on the current paper are: Database-as-a-Service (DBaaS), Hadoop-as-a-Service (HDaaS), Storage-as-a-Service (StaaS), Testing-as-a-Service (TeaaS).

3.3.2 A XaaS savvy Automated Approach to Composite Applications

Debnath, Sharma e Kaulgud (2015) analyzed several services provided by cloud providers, both in terms of functional and non-functional attributes, and concluded that these service descriptions are not structured. The paper then defines a web service description schema with attributes to represent the service's basic information (name, category, description), inputs and outputs, and tags (that are used to express the core features of a service) and proposes an automated algorithm to receive a composer's input to match available services and create a manifest which is used to orchestrate the application, using the configuration tool Puppet.

Noting the cost reduction of operating databases inside a service, both on licensing and administrative costs, specially because of the need of specialized operation and configuration for commercial DBMSs to achieve a good performance, Curino et al. (2011) presents Relational Cloud, a Database-as-a-Service developed by the authors.

The paper proposes a workload-aware system, which is used to split tenants databases across a pool of nodes. In this system, instead of provisioning new resources to the users when a new service instance is created, the pool of computational nodes is already provisioned, and new instances actually create new databases inside the database servers running on the nodes.

Users have access to the SQL features of the DBMS without being responsible for provisioning hardware resources, configuring software and tuning performance, security, access control and data privacy.

3.3.3 ORESTES - a Scalable Database-as-a-Service Architecture for Low Latency

Gessert, Bücklers e Ritter (2014) presents a Backend-as-a-Service/Database-as-a-Service called ORESTES (Objects RESTfully Encapsulated in Standard Formats), specially focused on providing low latency for users' requests, normally coming from web and mobile applications, and with read-intensive operations. The system also includes a REST interface and a server-side schema management, and stores data on an underlying NoSQL database, to benefit from it's scalable nature. While users can choose the specific database, like MongoDB or Cassandra, based on her own specific needs, the REST interface provides a uniform CRUD interface. But the database-specific querying functionalities are also allowed. To mitigate latency, cache is used on top of the HTTP interface, using a Bloom filter algorithm to achieve cache consistency.

The system uses cloud-hosted NoSQL databases, exposing them through the mentioned REST interface, but adds ACID transactions in a middleware level, leverages web caching to optimize responses and to achieve global content distribution, adds schema management and provides autoscaling. The system uses a complex coordination of six cache levels to achieve low-latency for the end user, and to provide this in a globally distributed access pattern.

3.4 Server Push as a Service

No work mentioning a cloud service meant to provide the server push functionality was found. It could be because of the usual approach of dealing with server push on the application code. The fact that server push technology is a relatively mature field, with the rise of cloud services no efforts of providing it as a service were done.

4 METHODOLOGY

This is an applied research focused on solving the specific problem of providing a given functionality in the context of the Tsuru platform. This work is constituted of the implementation and deployment of a solution, and a subsequent evaluation of the system, with the analysis of several non-quantifiable variables, as well as with other quantifiable variables. Because of this nature, a mixed approach of both qualitative and quantitative methods will be employed to evaluate the implementation.

On the evaluation phase, 25 employees of the brazilian media company Globo.com were invited to participate, from which 15 finished the evaluation, by submiting the questionnaire. The choice of inviting Globo.com employees was due to Globo.com involvement in the related projects. The company supports the active development of Tsuru, participated on the Nginx Push Stream module development, and currently uses both tools on production systems with millions of users. So, both the employees general experience and specific know-how about these tools would help them to better evaluate the system.

The steps on the evaluation process consisted of:

- watch a video with a general exposition of the system, specially the Nginx Push Stream module, the Push Service system and the PushaaS system.
- watch a video with a demonstration of the system.
- run exactly the same demonstration of the previous step. Because running the demonstration required installed software (the Tsuru CLI), this step was optional.
- answer an electronic questionnaire.

For the demonstration referred above, the application push-service-demo-app [6] was developed. This application is a simulation of a news website, where content can be consumed in a news feed, and also features a publishing interface, to simulate the publication of new content. This application has two modes of operation. If the application is unable to find an instance of Push Service bound to the application, the web application only load news upon a page refresh or navigation. If the application can find an instance of Push Service bound to the application, the web application will connect to the subscribing interface of the Push Service, and receive news and updates in real time, without the need of a refresh.

The demonstration was recorded on video and each participant received credentials that allowed him to reproduce the demonstration, if desired. The demonstration consisted of running a series of commands and performing a series of actions. These can be summarized as:

[6]https://github.com/pushaas/push-service-demo-app

- create an application on the Tsuru cluster and deploy the push-service-demo-app code to it.
- open the application on the browser, and use it. The application would not have an instance of Push Service created and bound to it, so it would work, but without real-time updates.
- request the creation of a Push Service instance.
- wait while the Push Service instance is being provisioned.
- bind the created Push Service instance to the demonstration application.
- open again the application on the browser, and use it. Now the application would have an instance of Push Service created and bound to it, so it would work with real-time updates.

The questionnaire is divided in sections. The first section is about the interviewee background on software development and on technologies related to the proposed implementation, which provides explanatory variables that help to analyze the remaining questions (VALLI, 2017). The remaining sections are focused on evaluation of different aspects of the system.

The questions about the system itself concerning more abstract, non quantifiable aspects of the system, so most questions gave room for the interviewee to analyze a particular aspect and expose perceptions, as well as raise attention points, that were collected and analyzed. Four questions asked the interviewee to answer using a Likert scale of five points, but besides the scale answer, a justification for the answer was asked. In all the cases the scale was used to directly measure satisfaction with aspects of the system, instead of using a traditional scale of agreement, which would add an undesired level of indirection (in this case, agreement with satisfaction levels) (SILVA JÚNIOR; COSTA, 2014).

Besides the questionnaire, a brief experiment measuring the provisioning time of Push Service instances on Amazon ECS was conducted by the author, to obtain a rough baseline of the required time to provision a new instance of the service.

5 IMPLEMENTATION

Several software components were developed to implement a server push service for the Tsuru platform.

The development strategy adopted was to first develop the more independent components, and add extra layers of complexity or integration later. So the development followed these steps, which will be examined in detail next:

- develop the Push Service, a micro-services based system to provide a server push service on top of the Nginx Push Stream module.
- develop the PushaaS system, with the capability of provisioning instances of Push Service in response to a service call.
- deploy the whole ecosystem to a cloud infrastructure and integrate the PushaaS with a Tsuru cluster, in order to perform tests and evaluate the implementation.

5.1 Push Service

The Push Service was developed as a micro-services system to augment the Nginx Push Stream module capabilities, by enabling a more scalable architecture, a more flexible deployment and a layer of extra functionality for users.

It helps to understand the role played by the Push Service once the limitations of the simple use of the Nginx Push Steam module are understood. The module works by being compiled together with the Nginx code, and this compilation results in a binary with the Nginx server with the Push Stream enabled. An instance of Nginx with Push Stream module could be run and configured to provide the server push functionality, by enabling on the Nginx both a publishing endpoint and a subscribing endpoint. This could be enough for very simple scenarios, but presents shortcomings for more complex needs. The main shortcoming is that horizontal scalability is not supported. While multiple instances of Nginx with the module could be deployed, there is no built-in coordination mechanism that would make all the instances work in collaboration. Even if a load balancer is used to distribute clients connecting to the subscribing endpoint, there is still no mechanism for coordinating and synchronizing the publishing of content across all instances. Besides this shortcoming, another aspect is that, while access to the publishing interface can be limited via configuration to only be allowed from specific networks, the module does not offer authentication on the publishing endpoint.

At a high-level, the Push Service can be understood as a publish-subscribe system between instances of a message-publishing interface (accessed by applications operated by a user responsible for publishing content, like some content management system) and a message-consuming interface (accessed by content consumers, typically end-users of some web application).

The Push Service is composed of four different software components, that can be seen in Figure 1. Briefly, the four components can be summarized as:

- push-api [7]: acts as the publishing interface of the system. Applications that want to interact with the Push Service call the push-api HTTP API. The push-api then publishes the received messages to the push-redis.

- push-redis [8]: acts as a publish-subscribe bus and as a data store.

- push-agent [9]: subscribes to messages on the push-redis, an upon a message, publishes it to the associated push-stream instance.

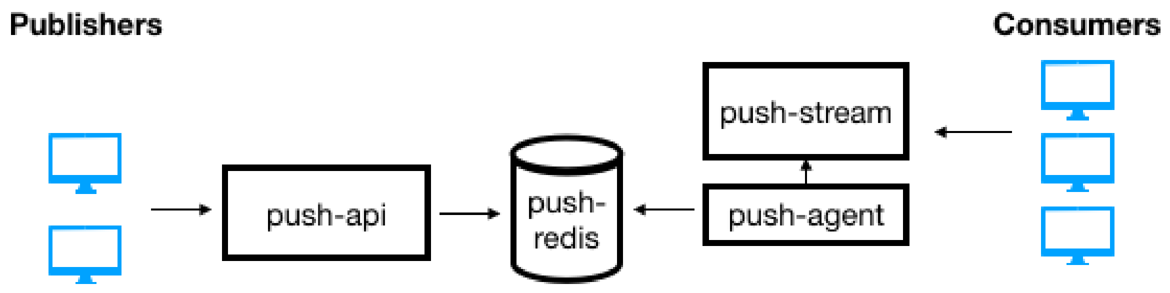
- push-stream [10]: receives subscriptions from end-users, and accepts publication from its associated push-agent instance.

Figure 1 showed the basic architecture of the Push Service, with a single instance of each component. The components could be replicated in order to achieve higher availability or to support higher loads, and load balancers could be added to distribute the load between the system interfaces, like shown in Figure 2.

Now the specific components will be examined in greater detail.

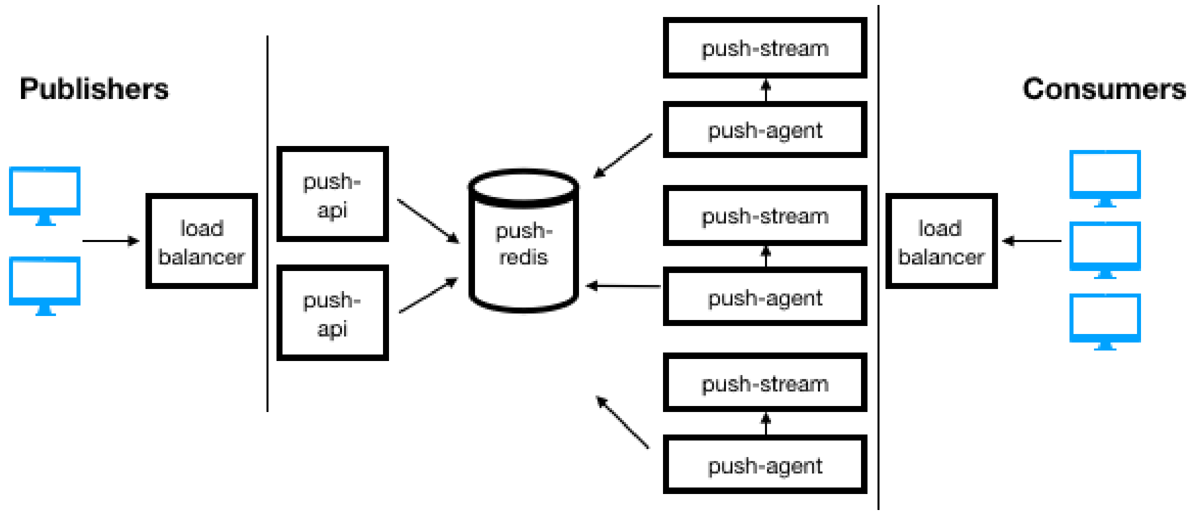
[7]https://github.com/pushaas/push-api
[8]https://github.com/pushaas/push-redis
[9]https://github.com/pushaas/push-agent
[10]https://github.com/pushaas/push-stream

Figure 1 – Push Service basic architecture



Source: The author

Figure 2 – Push Service scaled



Source: The author

5.1.1 push-stream

The Nginx Push Stream module is an open-source third party module for the Nginx web server. It is a mature tool, used by large real-life applications to enable server push functionality. But the plain usage of the module does not answer needs of horizontal scalability and coordination between instances that large applications require.

The Nginx Push Stream module works with the abstraction of channels, where messages can be published to or consumed from. A channel is a logical isolation of messages, providing some kind of scoping for messages. For instance, on a news website, each web page that requires real-time updates could be represented by a different channel.

An instance of Nginx with the Push Stream module installed will accept subscriber connections on channels, and, upon publication of messages on those channels, will distribute the message to all subscribers. The computational power of this instance will determine the number of existing channels, connected subscribers and published messages that the system can handle at once, and the module does not provide any means of horizontal scalability.

Is important to notice that the channels on the Push Stream module are ephemeral. They are allocated when a message is published on them, and consumers can subscribe to them while they exist. But after around 15 minutes of inactivity, they are deallocated and cease to exist, and attempts to subscribe to them will fail. If a persistent channel is to be provided, it must be kept alive with periodic messages just to keep the connections open. The Push Service as a whole provides this functionality of persistent channels, that will be explained on the push-api item.

The number of existing channels tends to pose a lesser problem in real-life applications, because it is usually associated with a dependent business resource, like a channel per different web page, so it doesn't tend to outgrow. But the number of connected subscribers in large applications can easily reach from thousands to millions of users. If multiple instances of the Nginx with Push Stream module were to be deployed to handle this load, a load balancer could be placed in front of the instances and distribute the subscribers between them. But because the load balancing is channel-agnostic, every instance should hold information of every channel, and message publishing should be replicated across all instances, so each instance could notify its subscribers. The Nginx Push Stream module does not provide means for this publishing replication across instances, nor the Push Service system provides the load-balancing capacity in the proposed implementation, but its architecture makes it viable to implement.

The push-stream component of the Push Service is a custom build of the Nginx web server, compiled with the Push Stream module, and a custom configuration file to expose the publishing and subscribing interfaces on Nginx. A very simple configuration exposing the publish-subscribe interface was added, and used to generate a Docker image for this component. The push-stream component does not overcome the limitations presented by the Nginx Push Stream module, only provides a basic Nginx build with the module, which, in coordination with the other components of the Push Service, will be used to overcome such limitations.

The Nginx with Push Stream module exposes routes both for publishing and subscribing to channels. The configuration file provided in the build of the push-stream component should be such that the subscribing interface is exposed and allowed for the targeted users (usually users of a public website), while the the publishing interface should only be exposed to the push-agent component (explained in the next item) corresponding to the push-stream instance.

5.1.2 push-agent

The push-agent is a component designed to work as a sidecar application to the push-stream, in order to better integrate it and coordinate with other parts of the system. It consists of a program that, while executed in a different machine or container than the push-stream itself, should guard a one-to-one relationship with the instances of push-stream of the system. For each instance of push-stream deployed, there should be deployed a corresponding push-agent instance, connected to that push-stream instance.

The push-agent is a small program developed in Go, aiming for a small binary with good runtime performance and low resource footprint. It fulfills a twofold role in the system:

- it subscribes to the push-redis waiting for messages. When a message arrives, push-agent publishes the message to the publishing interface of its corresponding push-stream instance. The messages were modeled as containing the message data and a list of channels to which publish this data. In most scenarios a message would be published only to a single channel, but this opens some possibilities for optimization of persistent channels, that will be explored later.
- it polls the push-stream for status and statistics, and save this data on the push-redis, giving visibility of this data separated by push-agent instances.

The existence of the push-agent listening to a bus that notifies message publishing events is a solution for coordinating message publishing across multiple push-stream instances, in the case of multiple instances being used to provide horizontal scalability.

5.1.3 push-redis

The Push Service system needs both a data storage system to store information about persistent channels, and a publish-subscribe mechanism to communicate message flow from push-api instances to push-stream instances (through the corresponding push-agent instances). The Redis data store was chosen because it can fulfill both these roles, with high performance and low resource footprint.

The push-redis component is essentially a Docker image of a simple Redis with extra configuration. By default, Redis provides memory-only storage, but to better tolerate failures, the configuration provided activates on-disk storage. This way, in the case of a failure, the persistent channels data are safely stored on disk and can be recovered upon a restart.

5.1.4 push-api

It was said on the push-stream item that the Nginx with the Push Stream module provides a publishing interface. But, because of the need of scaling horizontally the push-stream instances, the access to that interface was limited to a corresponding push-agent associated with the push-stream instance. In that way, feeding new messages to the push-agent instances is done by listening to the push-redis. The missing part so far is the publishing interface that can be called by the application that wants to deliver messages to its clients, and this role is played by the push-api.

The push-api is a Go program that both exposes an HTTP API and runs a background worker. The HTTP API exposes to client applications the ability to create and maintain persistent channels, and to publish messages on the channels. The background worker is responsible for keeping the persistent channels alive.

The HTTP API exposed by the push-api interacts with the push-redis. In the case of a channel creation or deletion, a corresponding record is added or removed to the Redis storage. Since Redis provides key expiration out of the box, it was trivial to implement a time-to-live functionality that can be configured on channels on its creation.

In the case of message publishing, the push-redis is used not as storage, but as a the bus. Hence, a message published on a push-api instance is delivered to the push-redis, which acts as a bus and notifies all listening push-agent instances, which in turn publish the message to its corresponding push-stream instance.

In most real-life scenarios, the publishing interface of the system should have a lower load when compared to the subscribing interface (i.e., usually on most systems there are more users consuming content than publishing content). Nonetheless, the push-api is a stateless application and might be easily horizontally scaled, simply by adding a load balancer in front of multiple instances, as long as they all publish messages to the same push-redis instance.

The Push Stream module does not provide authentication, and this might allow for unintended users publishing messages. In the setup proposed in the Push Service, this is acceptable because the Nginx configuration should only allow message publishing coming from the corresponding push-agent instance. But since in the Push Service the publishing surface is extended from the push-stream to the push-api, support for authentication was implemented on the push-api so that only authorized applications can publish content. Currently only HTTP basic authentication support was implemented, but several authentication methods could be easily implemented and made available through configuration.

Besides the HTTP API, the push-api is also composed by the aforementioned background worker. The worker is responsible for keeping alive the channels that were created to be persistent, overcoming the limitation of ephemeral channels imposed by the Push Stream module. The worker runs periodically and checks all channels saved on the push-redis. It then publishes a single message to the push-redis pub-sub system, containing the list of all channels to revive, and a well-known message data that represents a keep-alive message. Later, each instance of push-agent listening to the push-redis will receive the message containing all the channels, and will publish the keep-alive message to its corresponding push-stream instance, on all channels.

The push-api also serves the Push Service Admin, a web application containing an admin interface for the Push Service instance that can be accessed by the appropriated route on the push-api. The Admin application can be accessed using the same credentials that are used to interact with the push-api. The Admin application consists of an overview of the Server Push service. There is a main dashboard with information and statistics that each push-agent stores on the push-redis, and there is a list of all the persistent channels stored on the system. Information about each persistent channel can be seen, such as expiration date (if set), number of connected subscribers and others. Also, a view of messages being published on the channel through the push-api can be seen, as well as an interface for sending messages from the Admin, to be used for debugging purposes.

5.2 PushaaS

The PushaaS is the system responsible for provisioning instances of a complete Push Service (with all its four components) in response to a service call made by a developer who wants to create a new Push Service instance to be used by his application.

It consists of a Go program called pushaas [11] that exposes an HTTP API and runs some background jobs. It also requires a Redis instance to be used both as storage for instances

[11] https://github.com/pushaas/pushaas

information and to be used as a queue for background jobs.

The HTTP API exposes routes that allow clients to request instances of the Push Service to be provisioned and deprovisioned, as well as other operations like binding the service to applications, which will be further explored on the item about Tsuru Service API.

Upon a call to the appropriate route, the pushaas system will create a job to provision new infrastructure. To make the system more flexible and able to work on different environments and setups, pushaas internally works with the abstraction of provisioners, which are software components that implement a common interface with provision and deprovision methods, and are responsible for taking the appropriate actions on a corresponding infrastructure provider in order to provision or deprovision resources for a Push Service instance on that infrastruture provider.

The infrastructure provisioner choice was implemented as a pushaas application-level configuration, so different organizations using the pushaas system could choose to provision Push Service instances on the infrastructure provider of their choice, like on many public cloud offerings (Amazon EC2, Google Cloud Platform, Microsoft Azure, Digital Ocean) as well as on on-premises solutions (Apache CloudStack, OpenStack). For simplicity, it is an application-level configuration, not a per Push Service instance configuration (i.e., all Push Service instances created via the pushaas will be provisioned on the same infrastructure provider, through the application-level configured provisioner). Currently only a single infrastructure provisioner (Amazon ECS) is implemented, and the default configuration uses it, but the addition of new implementations to the codebase should be trivial.

To provide greater flexibility, the pushaas supports the concept plans for the service. Plan is a concept from Tsuru services, that will be further explored in a later section, but it essentially refers to the amount of computational resources allocated for a service. The pushaas was developed with support for plans, but the current implementation only supports a default "small" plan.

Besides the HTTP API, a web application containing an admin interface for the pushaas was developed, and can be accessed by the appropriated route on the pushaas. The admin application can be accessed using the same credentials that were used to register the service on the Tsuru cluster, and it consists of a tool for administrators to have a unified view of all the Push Service instances provisioned using the PushaaS, and their associated resources.

5.2.1 Amazon ECS Provisioner

The role of the provisioner is to create (or remove) resources for a complete Push Service instance to be fully operational, so it can later be bound to an application that will use the service it provides.

The Amazon ECS (Elastic Container Service) was chosen to be the infrastructure provider used to implement the default provisioner, to validate and evaluate the pushaas system, because it offers a light-weight container system with low cost that was convenient for the validation purpose.

Amazon ECS works upon a few important concepts that will be used on the provisioning process:

- cluster: is a logical grouping of tasks and services. Tasks and services are defined inside of the scope of a cluster.

- task definition: is a description of an abstract task, containing information like the required resources (CPU, memory) for the task, environment variables and containers definitions. A single task definition can be composed of multiple different containers, if needed.

- service: is an actual execution of a task. A service is defined inside a cluster, and runs a configured number of instances of a task definition, exposing network addresses for the instances.
- service discovery service: to allow services to be referenced by names (instead of IPs) internally on the AWS private network (which is a more convenient and flexible way of referencing services), it is necessary a service discovery mechanism that allow services to be registered by name, creating an indirection to their actual addresses.

The provisioning of a new Push Service instance with the Amazon ECS provisioner on pushaas follows these main steps:

- the instance data is stored on the pushaas Redis instance, containing information like its name, status and owner team.
- provision the Push Service push-redis.
  - all instances of Push Service can reuse the same task definition, since there is no custom configuration per different instance. So the task definition for the push-redis was previously created once.
  - a service discovery service is created so the new push-redis service can be referenced internally by other components by its service name.
  - a service is created using the task definition and the service discovery service. The service is created with the appropriated security groups and on the appropriate subnet.
  - the system waits for the service to become on the "running" state.
- provision the Push Service push-stream and push-agent.
  - a specific task definition is created for this instance of Push Service. This task definition defines that is composed of two containers, one from the push-stream image and one from the push-agent image. The push-agent definition receives environment variables that define the appropriate addresses to connect to the corresponding push-stream and to the push-redis of the Push Service instance.
  - a service discovery service is created so the new push-stream service can be referenced internally by other components by its service name,
  - a service is created using the task definition and the service discovery service. The service is created with the appropriated security groups and on the appropriate subnet.
  - the system waits for the service to become on the "running" state.
- provision the Push Service push-api.
  - a specific task definition is created for this instance of Push Service. This task definition defines that is composed of one container from the push-api image. The container definition receives environment variables that define the appropriate address to connect to the push-redis of the Push Service instance, as well as generated

credentials to be used by applications to connect to the push-api instance, and the public address for clients who want to connect to the push-stream. Here there is a point of technical debt that was overlooked for sake of simplicity on the current implementation. End-user applications who want to receive real-time updates must be connected to the push-stream component to receive the updates. These applications can know the address of the push-stream to connect by asking the push-api on the appropriate route. The technical debt is that, since currently no load balancers are being added in front of the push-stream component (because in the existing small plan there is only a single instance of each component), the push-stream service is being exposed publicly and its public IP is being used for clients to connect, which is less flexible than in ideal situation where a load balancer would abstract these details from the client application.

  - a service discovery service is created so the new push-api service can be referenced internally by other components by its service name.
  - a service is created using the task definition and the service discovery service. The service is created with the appropriated security groups and on the appropriate subnet.
  - the system waits for the service to become on the "running" state.

- the instance data is updated with the running status, so users can query and know that the provisioning is done.

Since the pushaas supports plans, but only a "small" plan is currently implemented, all instances of the Push Service are being provisioned with the same amount of computational resources. The current implementation creates only a single instance of each component and does not add load balancers in front of the push-api or the push-stream. This was done to simplify the process of provisioning in the current state of development of the project, but poses a limitation that should be solved in a future development to allow for greater flexibility and scalability. All containers in the "small" plan are created with 256 CPU units (a computational resource measurement used on AWS ECS), and 512 MiB of memory.

5.2.2 Tsuru Service API

The pushaas as explained so far has the ability to provision and deprovision Push Service instances. But this work intends to offer this functionality as a Tsuru Service.

The process of integrating a service to a Tsuru cluster is trivial:

- the service must be registered on the Tsuru cluster.
- the service must offer some HTTP routes that will be called when the Tsuru user interacts with the service.

Registering the service on the Tsuru cluster is done by creating a manifest file that configures service informations (like endpoints and credentials) and running the appropriate commands on the Tsuru CLI [12].

Since this service was planned from the beginning to be integrated to Tsuru, the routes for service instances provisioning were created following the Tsuru Service API Specification [13].

[12]https://docs.tsuru.io/1.7/services/build.html#creating-a-service-manifest

[13]https://docs.tsuru.io/1.7/services/api.htmlapi-specification

Besides the routes for provisioning, to fulfill the specification, some routes are needed for binding service instances to applications. The process of binding consists basically of:

- tracking internally that the service instance is related to a given application.
- returning a set of environment variables that will be set on the application, with information the application needs to reach the service (like endpoints and credentials).
- taking any kind of extra actions, like allowing network access to resources. In the current setup and with the AWS ECS provisioner, no extra actions were required.

All the provisioning and binding functionality was developed and exposed accordingly to the Tsuru Service API Specification, and with the service registration, the service was exposed as a Tsuru service.

5.3 Infrastructure on AWS

To validate the implementation of the whole PushaaS system (the deployment of the pushaas app, the integration with Tsuru applications and the provisioning of Push Service instances), it is necessary to deploy the system as a whole to some infrastructure.

The main components that need to be deployed are:

- the Tsuru cluster, with control nodes - where Tsuru runs its own internal components, and with poll nodes - where the client applications get deployed.
- the pushaas app, and its Redis instance.
- the Push Service instances that get provisioned upon calls to the pushaas API.

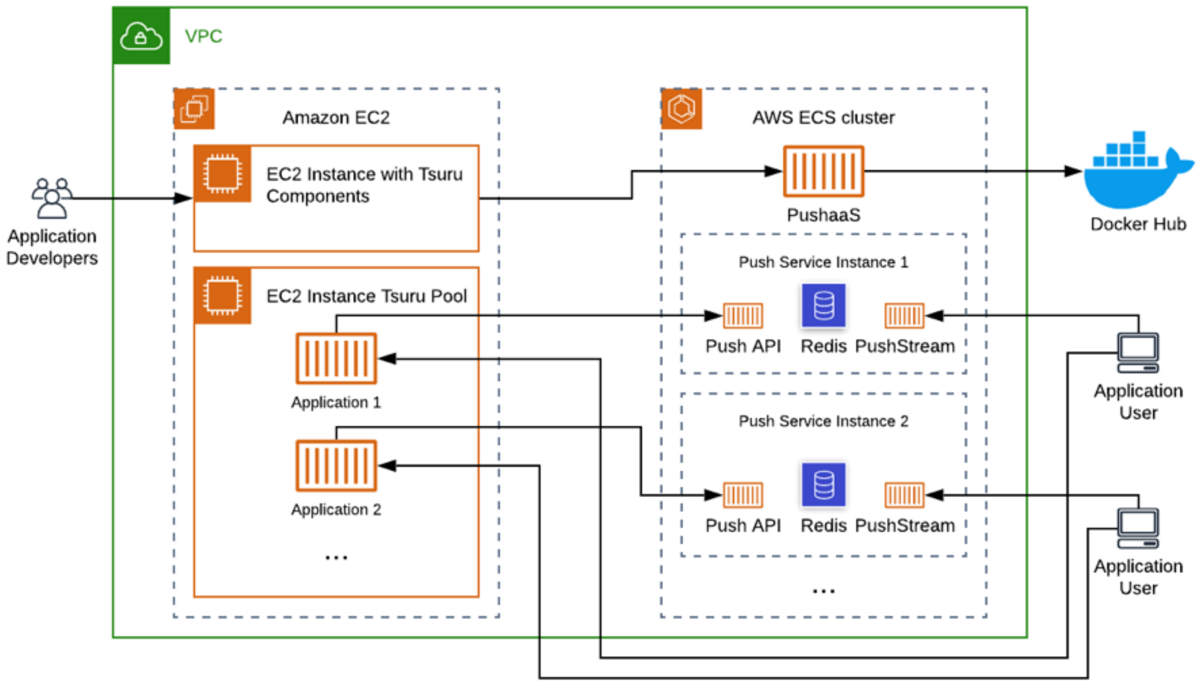
The chosen infrastructure provider for this validation was AWS, and Terraform was used to automate processes and achieve better reproducibility [14]. Figure 3 shows the overview of the architecture used in the evaluation of the system, presenting the Tsuru cluster, the PushaaS system deployed on Amazon ECS, and instances of the Push Service provisioned on Amazon ECS and bound to applications running on the Tsuru cluster, being accessed by end-users. Note that in the figure, instances of push-agent are not represented explicitly, but represented unified with its associated push-stream instance, for simplicity.

The main components of infrastructure needed to deploy and validate the PushaaS system will be examined now. The following list contains the main steps taken to deploy the system:

- a VPC (virtual private cloud) was created, so the PushaaS system could be all deployed to a single isolated virtual private cloud inside AWS infrastructure.
- Tsuru was installed on Amazon EC2. The Tsuru installer was used, and it created a virtual machine for the Tsuru components, and a separate machine to be the Tsuru pool, where applications get deployed. Since it was a simple validation of a integrated service, a pool with a single machine was enough to hold the client applications.
- Tsuru roles and teams were configured, so later they could be used to grant permissions to developers during the validation process.
- a private DNS namespace was created, to allow for registering DNS records of services, so services could be addressed by their names instead of IPs.

[14]https://github.com/pushaas/pushaas-aws-ecs-config

Figure 3 – PushaaS Infrastructure



Source: The author

- an Amazon ECS cluster was created, to hold task definitions and services that would later be created.

- an ECS service was created running an instance of Redis, to be used by the pushaas.

- an ECS service was created running the pushaas.

- the pushaas ECS service address was used to generate a manifest file and register the service on the Tsuru cluster.

- an ECS task definition for the push-redis was created, as noted previously that since all instances of Push Service would run an exact copy of the same push-redis image, the task definition could be shared among all instances.

- the NodeJS platform was added to the Tsuru cluster. This makes Tsuru capable of running NodeJS applications. The test application develop for the validation process is a NodeJS application, so it could now be deployed and run on Tsuru.

6 RESULTS ANALYSIS

6.1 Questionnaire

There were 15 respondents to the questionnaire. The questionnaire's sections will be analyzed on the following sections.

6.1.1 Interviewee Background

The first section of the questionnaire was used to determine the interviewee's degree of experience in the software development field, as well as familiarity with technologies used on this project. Table 1 shows the number of years of experience in the software development field of the interviewees.

0 to 5 years	6 to 10 years	11 to 15 years	16 to 20 years
2 people	3 people	6 people	4 people

Table 1 – Interviewees experience in the software development field

All the interviewees work currently as software developers, devops, system administrators or managers of development teams.

About the familiarity with the Tsuru platform, all the interviewees were familiar with the platform. The familiarity ranged from around 1 year of daily usage as a client of the platform, to Tsuru team members, like core developer and and product owner, with several years of experience on the platform.

About the familiarity with the Push Stream module, nine interviewees either did not know the project, or only knew it by name, without any practical experience. The other six interviewees experience varied between being a user, managing teams that work with the module, and being a developer of the project.

It was important to know the interviewees background in the software development field and with the technologies associated with the present work, because it helps to better analyze the subsequent questions, specific about the project.

6.1.2 Server Push

In the single-question section about the server push functionality, the interviewees were asked to speak about the relevance of server push functionality on modern applications.

All the responses considered it relevant, with some noting it as highly relevant. The main reasons were:

- clients can get updates as soon as possible, which is a relevant feature for the application business.
- promotes a more engaging experience.
- the freshness sensation improves the overall user experience on a digital product.
- better resource usage (both on server-side and on client-side) when compared to polling techniques.

6.1.3 Push Service

On this section, the interviewees were asked to evaluate the Push Service system as a standalone solution to implement the server push functionality, without taking into account its integration to the Tsuru platform or the provisioning of instances upon a service call (examined on the next section).

The first question of the section was about the perception of the horizontal scalability of the Push Service architecture. The general perception was that the components can easily be horizontally scaled with the addition of multiple instances of each component, mainly attributed to the usage of Redis as a pub-sub mechanism to uncouple the other components. It was noted that the ability of scaling the Push Service system can avoid the need of scaling the client application. Some interesting attention points noted were:

- the Redis itself, if not properly scaled, can become the system bottleneck on scenarios of high load on the publishing interface. This should not be a problem on most systems since is reasonable to expect a higher load always on the subscribing interface than on the publishing interface, but nonetheless is a real possibility and constitutes a good point of development for future work.

- on the subscribing interface, the connected clients can be distributed across multiple push-stream instances, thus not posing a problem. But since there is no sharding of channels across the push-stream instances (i.e., all channels will exist on all push-stream instances), if the number of persistent channels grows too much over time, the instances might run out of memory to store all channels data. On most real-life systems the number of channels should not be a problem, and the usage of channels with an appropriate TTL configured should keep the number of existing channels to a reasonable number.

The second question was about the perception of independent scaling of both the publishing and the subscribing interface of the system. The general perception is that both ends of the system can be scaled independently to accommodate the client application need of extra load on the publishing or subscribing interfaces. Again, the usage of the Redis as a pub-sub was pointed as a reason for this. One interesting attention point noted was how new instances of push-stream and push-agent can be initialized with previously published data, in order to constitute a valid initial state that clients can request when connecting to a push-stream instance. This point is specially interesting because the Push Stream module supports a backtrack mechanism. A client that will connect to a push-stream instance can specify a number of seconds to backtrack, e.g., if the client specifies 60 seconds, it will receive from the push-stream messages published on the previous 60 seconds. The Push Stream module handles this natively by holding the messages in memory for a specified period of time, but the current implementation of the Push Service does not provide any means of initializing push-agent instances with a state of previously published messages on other instances.

The third question asked about the satisfaction about the extra features that the Push Service system provides on top of the plain usage of the Push Stream module, namely the authentication, and ability of creating persistent channels, with optional expiration. This question used a Likert scale and the respondent was asked to justify the answer. Table 2 shows the distribution on the scale. The main observations were that authentication was an important feature to be added. The resources savings suggested by having expiration on channels was also pointed as a very important feature, while one answer demonstrated skepticism about the actual amount of resources savings that this would enable.

1 (very dissatisfied)	2 (dissatisfied)	3 (neutral)	4 (satisfied)	5 (very satisfied)
0	0	1	8	6

Table 2 – Satisfaction about extra features

Lastly, the users were asked about the satisfaction with the Push Service Admin. This question used a Likert scale and the respondent was asked to justify the answer. Table 3 shows the distribution on the scale. The simplicity, presence of metrics and ability of viewing the publishing stream were regarded as important features.

1 (very dissatisfied)	2 (dissatisfied)	3 (neutral)	4 (satisfied)	5 (very satisfied)
0	0	2	3	10

Table 3 – Satisfaction about Push Service Admin

6.1.4 PushaaS

On this section, the interviewees were asked to evaluate the offering of Push Service instances as a service, taking into account its integration with the Tsuru platform.

The first question of the section asked the interviewees to point positive aspects about the offering of a server push service as a cloud service for the Tsuru platform. All interviewees brought up positive aspects, and the main aspects mentioned were:

- simplicity and quickness of applications to use the service.
- autonomy for applications developers working with Tsuru to provision instances.
- the gain in scale that the service allows (i.e., implement the service once and all applications on the platform can benefit from it with little extra effort).
- expected low maintenance cost.
- isolation between instances for different applications, allowing for independent failures.
- facilitated billing, because the isolation of instances allows for independent billing for applications.
- simplicity to integrate with applications.

The second question asked the interviewees to point negative aspects about the offering of a server push service as a cloud service for the Tsuru platform. Six interviewees brought up negative aspects, and the main aspects mentioned were:

- less flexibility when compared to a service provisioned manually.
- the multiplication of instances could required more idle resources, when publishing is sporadic. Each instance would require a minimum infrastructure allocated to run each of the Push Service components.
- integration with Tsuru could be overkill in very simple scenarios.

The third question asked the interviewees to evaluate the convenience for application developers to use the Push Service as a cloud service offered by Tsuru. The process for application developers to use the service consist of asking the creation of the instance via the Tsuru CLI, waiting for the instance to become available, and asking for the service instance do be bound to the application, also via the Tsuru CLI. This process is rather standard for any Tsuru service. This question used a Likert scale and the respondent was asked to justify the answer. Table 4 shows the distribution on the scale. The process was noted to follow the expected steps for any Tsuru service. The only attention point brought up was about the need for waiting the service

1 (very dissatisfied)	2 (dissatisfied)	3 (neutral)	4 (satisfied)	5 (very satisfied)
0	0	0	1	14

Table 4 – Satisfaction with the convenience of Push Service usage as Tsuru service

instance to be provisioned before asking for the bind, which is part of the provisioning of any Tsuru service, not particular to the PushaaS.

The fourth question asked the interviewees to express their satisfaction level about the time needed to provision a new Push Service instance. This question used a Likert scale and the respondent was asked to justify the answer. Table 5 shows the distribution on the scale. The time needed to provision an full instance of the Push Service is largely determined by the chosen provisioner and the service plan, which will determine in which infrastructure and how many infrastructure components will actually be provisioned, so it must be understood that the interviewees perception of time during the system evaluation is based upon a particular provisioner (Amazon ECS) provisioning components for a particular service plan (the "small" plan).

1 (very dissatisfied)	2 (dissatisfied)	3 (neutral)	4 (satisfied)	5 (very satisfied)
0	0	1	2	12

Table 5 – Satisfaction with the provisioning time of a Push Service instance

The last question asked about general impressions about the usage of the PushaaS system to implement the server push functionality on applications executed on the Tsuru platform. The perception was overall positive on all answers, and some aspects were highlighted:

- the system leverages existing technologies and integrates them in a easy and inexpensive way.
- the PushaaS usage is very familiar for Tsuru users, fulfilling the application developer expectations.
- isolation of service instances is a beneficial aspect for organizations with multiple applications that consume the service.
- the service architecture favors scalability.
- the extra features added are relevant.
- the system solves some problems that the plain use of the Nginx Push Stream module would present.

Some attention points also were brought up:

- the need to evaluate the horizontal scalability of Push Service in real-life applications.
- for small use-cases, the system is too complex and application-level handling of server push functionality could be preferable.
- for better integration with organizations, other authentication mechanisms should be supported besides HTTP basic.

- the Nginx Push Stream module does not support channels sharding. Future work could be done in the Push Stream module itself to support sharding and a cluster mode of operation, which could allow for easier horizontal scalability even replace some components of the proposed Push Service presented in this work.

6.2 Provisioning Benchmarks

In order to obtain some baseline about the time needed to provision an instance of the Push Service, a benchmark of the provisioning of ten instances sequentially was conducted. Here again is important to notice that the provisioning time is highly dependent on the provisioner and on the instance plan. The benchmarks were conducted using the "small" plan and the Amazon ECS provisioner. Table 6 shows time that the provisioning of each instance took.

Instance	Time (in seconds)
1	236
2	205
3	331
4	244
5	239
6	244
7	199
8	198
9	186
10	259

Table 6 – Benchmark of time to provision instances of Push Service

The average time was 234 seconds, with the median being 237. The times the interviewees experimented on the demonstration was likely similar to the observed times on this benchmark.

7 CONCLUSION

This work aimed to provide an implementation of a server push service for client applications running on top of the Tsuru platform, in a convenient way to allow for easy provisioning and integration with applications. Besides the convenience, the system also focused on delivering a well-designed architecture to allow for horizontal scaling and handling of significant loads.

The Push Service was able to leverage the existing Nginx Push Stream module, a high-performance tool used in large real-world system, to compose a larger system focused on providing horizontal scalability and an extra layer of features useful for organizations. The PushaaS was able to take the Push Service and abstract its usage by offering it as a cloud service. The integration of the service with the Tsuru platform was successful and its usage via Tsuru was perceived by Tsuru users as very familiar and similar to other existing and more established services.

The evaluation process conducted showed a positive reception of the service by Tsuru users, which considered relevant the problem the system solves, and considered the implementation a savvy approach to the problem. While the general reception was positive, various attention points were raised, constituting a valuable source of improvements to be made on future work.

The system presents some important extension points, that while received at least a basic implementation, should be extended in order for the system to really achieve a desired level of flexibility to make it useful in more scenarios:

- provisioners: the development and evaluation processes provisioned infrastructure components on Amazon ECS. Organizations willing to adopt the system would likely have other infrastructure providers, so the development of new provisioners would be needed. This is the main extension point in the system, and the implementation of new provisioners should be fairly trivial. The pushaas application already supports configuration of the chosen provisioner as an application-level configuration.
- plans: somewhat related to the provisioners, Push Service instances, when created, are sized according to the chosen plan for the service instance. The system was implemented with a single, default, "small" plan for service instances. While this is acceptable for the evaluation purposes, usage of the system on real-world scenarios requires allowing the developer to have control over the sizing of the service instance, both on the vertical aspect (the computational power to be allocated for components) and on the horizontal aspect (the number of replicas of components, to achieve high availability and allow load balancing between instances). Also, aspects of auto-scaling were ignored on this work, but should be considered as an improvement on top of plans.
- authentication: the system proposed an authenticated publishing interface, overcoming the limitation of the Nginx Push Stream module. The current implementation only supports HTTP basic authentication, but organizations often need more complex authentication schemes, specially to allow integration with existing platforms. The addition of extra authentication schemes should also be fairly trivial.

The Push Service architecture was designed as a collaboration of several uncoupled and mostly stateless components, which can be replicated to support higher loads. While this architecture is known from the author experience to allow for horizontal scaling, and was generally perceived as able to horizontally scale by the interviewees, no actual performance tests or benchmarks were performed.

The PushaaS application itself resulted in an inexpensive application, with little infrastructure requirements - only a Redis instance for data storing and job scheduling, besides the application itself, so the adoption of the PushaaS system and its integration with an existing Tsuru cluster should be a simple process.

8 FUTURE WORK

While the PushaaS system is currently functional, it has several shortcomings that reduce its flexibility and applicability for real-world scenarios. Some of these shortcomings were known from the beginning and were intentional in order to reduce the scope of the current work, while others were noted by the interviewees on the validation process. Besides the shortcomings pointed, other approaches and techniques were also suggested, which can guide future work with different approaches than what was developed on this work.

Some aspects of the Push Service that could be evolved include:

- support for other authentication schemes, besides HTTP basic auth, making the Push Service more secure and fitter for usage on complex environments.

- the system could benefit from load tests, investigating how the system behaves under heavy load, and tuning the system so its components can be scaled evenly, avoiding bottlenecks.
- in the case of scaling horizontally by adding new instances of the push-stream component, the problem of initializing the state of new push-stream instances (so the newly created instance can support backtrack of messages) was pointed on the evaluation and is still an open problem in the current implementation.

Some aspects of the PushaaS that could be evolved include:

- support for multiple plans. As stated previously, this work only implemented the default "small" plan. Other plans could be supported. The plans should consider both aspects of vertical sizing of computational resources, as well as horizontal deployments and load balancing between instances of the components. Multiple instances of components such as the push-api or the push-stream are desired both for load distribution reasons as well as for high availability needs. Also the sizing of the push-redis component should be considered. Large plans, with higher needs of performance and availability could benefit from the usage of different Redis architectures, such as using Sentinel [15] for high-availability or a cluster mode for supporting higher loads.
- implementation of different provisioners. The pushaas application already supports application-level configuration of the chosen provisioner, but currently only Amazon ECS is implemented. For real-world scenarios, is likely that organizations would need different provisioners to fit their infrastructure stack.
- improvements on the Amazon ECS provisioner. While the provisioner is fully functional, it could be optimized by provisioning some components in parallel.

Some other approaches to achieve the results intended by this work were suggested on the evaluation and present interesting ways of evolving the solution:

- integration and usage of existing Tsuru services. DBaaS [16] (Database as a Service) and RPaaS [17] (Reverse Proxy as a Service) are more established Tsuru services that could be further integrated in order to provision parts of the infrastructure needed for Push Service instances.
- development of a "cluster mode" on the Nginx Push Stream module, allowing to deploy multiple instances of Nginx with the Push Stream module and connecting them in a cluster, allowing native sharding of channels and distribution of published messages between the instances, without the need of the external distribution system, performed on this work by the pub-sub on the push-redis and usage of the push-agent.

REFERENCES

COSTACHE, S. et al. Resource management in cloud platform as a service systems: analysis and opportunities. **Journal of Systems and Software**, [S.l.], v. 132, p. 98–118, 2017.

[15]https://redis.io/topics/sentinel
[16]https://github.com/globocom/database-as-a-service
[17]https://github.com/tsuru/rpaas

CURINO, C. et al. Relational cloud: a database-as-a-service for the cloud., [S.l.], 2011.

DEBNATH, P.; SHARMA, V. S.; KAULGUD, V. A xaas savvy automated approach to composite applications. In: IEEE 8TH INTERNATIONAL CONFERENCE ON CLOUD COMPUTING, 2015., 2015. **Anais...** [S.l.: s.n.], 2015. p. 734–741.

DUAN, Y. et al. Everything as a service (xaas) on the cloud: origins, current and future trends. In: IEEE 8TH INTERNATIONAL CONFERENCE ON CLOUD COMPUTING, 2015., 2015. **Anais...** [S.l.: s.n.], 2015. p. 621–628.

ESTEP, E. **Mobile html5**: efficiency and performance of websockets and server-sent events. 2013.

FARGATE. Accessed: 2019-05-21, `https://web.archive.org/web/20190520202950/https://aws.amazon.com/fargate/`.

GARCIA-GOMEZ, S. et al. Challenges for the comprehensive management of cloud services in a paas framework. **Scalable Computing: Practice and Experience**, [S.l.], p. 201–214, 2012.

GASCON-SAMSON, J. et al. Dynamoth: a scalable pub/sub middleware for latency-constrained applications in the cloud. In: IEEE 35TH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 2015., 2015. **Anais...** [S.l.: s.n.], 2015. p. 486–496.

GESSERT, F.; BÜCKLERS, F.; RITTER, N. Orestes: a scalable database-as-a-service architecture for low latency. In: IEEE 30TH INTERNATIONAL CONFERENCE ON DATA ENGINEERING WORKSHOPS, 2014., 2014. **Anais...** [S.l.: s.n.], 2014. p. 215–222.

KÄCHELE, S. et al. Beyond iaas and paas: an extended cloud taxonomy for computation, storage and networking. In: IEEE/ACM 6TH INTERNATIONAL CONFERENCE ON UTILITY AND CLOUD COMPUTING, 2013., 2013. **Anais...** [S.l.: s.n.], 2013. p. 75–82.

LA, H. J.; HER, J. S.; KIM, S. D. Framework for evaluating reusability of component-as-a-service (caas). In: INTERNATIONAL WORKSHOP ON PRINCIPLES OF ENGINEERING SERVICE-ORIENTED SYSTEMS (PESOS), 2013., 2013. **Anais...** [S.l.: s.n.], 2013. p. 41–44.

LO, W.-T. et al. A restful web notification service. **Journal of the Chinese Institute of Engineers**, [S.l.], v. 39, n. 4, p. 429–435, 2016.

MARINOS, I.; WATSON, R. N.; HANDLEY, M. Network stack specialization for performance. In: ACM SIGCOMM COMPUTER COMMUNICATION REVIEW, 2014. **Anais...** [S.l.: s.n.], 2014. v. 44, n. 4, p. 175–186.

MELL, P.; GRANCE, T. et al. The nist definition of cloud computing., [S.l.], 2011.

PUSH stream. Accessed: 2019-05-21, `https://web.archive.org/web/20190123045845/https://www.nginx.com/resources/wiki/modules/push_stream/`.

SHARMA, S. Evolution of as-a-service era in cloud. cornell university library. **Available at arxiv. org/ftp/arxiv/papers/1507/1507.00939. pdf**, [S.l.], 2015.

SHUANG, K.; KAI, F. Research on server push methods in web browser based instant messaging applications. **Journal of Software**, [S.l.], v. 8, n. 10, p. 2644–2651, 2013.

SILVA JÚNIOR, S.; COSTA, F. Measurement and verification scales: a comparative analysis between the likert and phrase completion scales. **Braz J Marketing, Opinion, and Media Research**, [S.l.], v. 15, 2014.

TIHFON, G. M. et al. An efficient multi-task paas cloud infrastructure based on docker and aws ecs for application deployment. **Cluster Computing**, [S.l.], v. 19, n. 3, p. 1585–1597, 2016.

TSURU. Accessed: 2019-05-21, `https://web.archive.org/web/20190523024243/https://docs.tsuru.io/1.6/`.

VALLI, R. Creating a questionnaire for a scientific study. **International Journal of Research Studies in Education**, [S.l.], v. 6, 2017.

VERGARA, G. F. et al. Deployment of secure collaborative softwares as a service in a private cloud to a software factory. In: CLOUD COMPUTING 2014, THE FIFTH INTERNATIONAL CONFERENCE ON CLOUD COMPUTING, GRIDS, AND VIRTUALIZATION, 2014. **Anais...** [S.l.: s.n.], 2014. p. 123–129.

WESTRUP, E.; PETTERSSON, F. Using the go programming language in practice. **Department of Computer Science, Faculty of Engineering LTH**, [S.l.], 2014.