YAR: Yet Another Reddit*

*Work developed within the course of Cloud Computing Systems in the lective year 2019/20

André Rosa
48043
af.rosa@campus.fct.unl.pt

Rúben Silva
47134
rfc.silva@campus.fct.unl.pt

***Abstract*—For the practical evaluation of the course Cloud Computing Systems, we have been proposed to create a web application similar to Reddit, resorting to Azure Cloud Services. Therefore, we present YAR (Yet Another Reddit), our implementation of such an application. Generally speaking, YAR consists on a massive collection of forums, called communities, where users can post whatever they pretend, like sharing news or publishing questions, and can also comment on other users' posts.**

We deployed our solution, in a geo-replicated setting in West Europe and Central North America, and we evaluated the performance of YAR in both scenarios.

***Index Terms*—Reddit, Azure, CosmosDB, Azure Functions, Redis, Cloud**

I. Introduction

Cloud computing has become an important catalyst and promoter for the development of new Web Services, mainly Web Apps. The most important reason is that Cloud brought to the table what is called Computing as a Service: the ability of progressively scale and deploy a web service on-demand and in a reasonably short amount of time. Therefore, new creative ideas no longer held on paper due to the necessity of investing in expensive hardware, that also needs to be maintained and updated. Additionally, developers also gained access to new services (such as new Database Systems, Caching solutions, Data Mining, etc.), that enables building better applications, with high availability and low latency, that ultimately lead to reaching more and happier clients.

For the practical evaluation of the Cloud Computing Systems course, we have been proposed to create a web application similar to Reddit [1], resorting to Microsoft's Cloud Services and Infrastructure: Azure [2]. Therefore, we present YAR (Yet Another Reddit), our implementation of such an application. Generally speaking, YAR consists on a massive collection of forums, called communities, where users can publish (post) whatever they want, such as sharing news, publishing questions, add comments, and like or dislike other user's posts.

We deployed our solution in a geo-replicated setting, in West Europe and Central North America, and we evaluated the performance of YAR in both scenarios.

The remainder of this report is structured as follows: on Section II we introduce the design of the system, including the architecture with its main components and how these components interact with each other; on Section III we introduce the implementation details that are worth being highlighted; on Section IV we present the experimental evaluation of the system; and, on Section V we conclude the report with some final remarks.

II. Design

In this section, we present YAR's data model (II-A) and then its architecture (II-B).

A. Data Model

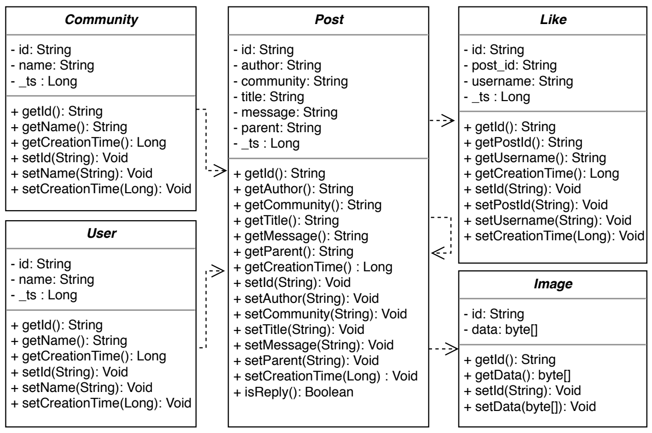

Fig. 1. YAR's Data Model

The data model of our system is presented in figure 1, and can be summarized as follows:

- **Communities** YAR contains a set of communities, where users can insert new posts. A community serves to group posts about a similar theme/subject. Each community has two unique identifiers (*id* and *name*) and a timestamp (*_ts*, in seconds) of the moment when it was created.
- **Users** Each user has two unique identifiers (*id* and *name*) and a timestamp (*_ts*, in seconds) of the moment when they registered in the system. (Note: Passwords and other user's details were out of scope of the project requirements).
- **Images** Each image has an unique identifier (*id*), that corresponds to the hash of its contents. This enables the

system to keep only a copy of each image when it is uploaded more than one times.

- **Posts** Posts are the core part of YAR. A Post can either be a root post or a reply (or comment) to another post. Each post has a title (optional), an author (name of the user who created it), a community (name of the community where it belongs), a message, a timestamp (*_ts*, in seconds) of the moment when it was created, and a parent (optinal - id of the parent post). When a post has a parent, it means that it is a reply to another post. In this case, we have the invariant that their communities must be the same.

- **Likes** Users can express their enjoyment of a post by liking it. A user can only like a given post a single time. Each like has the id of the post (*post_id*), the name of the user who liked the post (*username*), an unique identifier(*id*), that results of the concatenation of the id of the post and the name of the user, and a timestamp (*_ts*, in seconds) of the moment when it was created.

B. Architecture

As previously mentioned, we built YAR leveraging on Azure Cloud Services and Infrastructure. Services made available in this platform are called Resources.

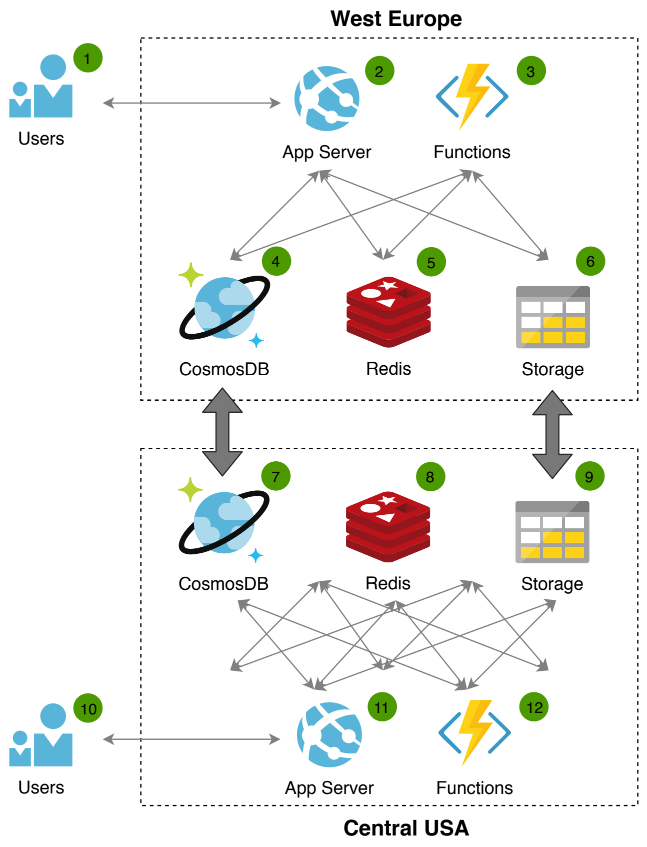

Fig. 2. YAR's Architecture

The Architecture of our system is presented in figure 2, presenting Azure's resources we utilized. They are separated into 2 regions: West Europe and Central USA. In each, we have the following resources: an App Server, a CosmosDB, a Blob Storage, a Redis, and Azure's Functions. Each of this resources are presented, in more detail, next.

1) App Server: The App Server corresponds to a virtual machine, runnig our application code, that deals/answers to client requests, communicating with other resources if necessary, and thus, it acts as a gateway between the clients and the system. It exposes a REST API with the following endpoints:

- **POST /community** Create a new community.

- **GET /community/{*name*}** Retrieve the community identified by *name*.

- **POST /user** Register a new user.

- **GET /user/{*name*}** Retrieve the user identified by *name*.

- **POST /image** Upload a new image.

- **GET /image/{*id*}** Download the image identified by *id*.

- **POST /post** Creates a new post.

- **GET /post/{*id*}** Retrieves the post identified by *id*.

- **POST /post/{*id*}/like/{*username*}** Inserts a new like, of the user identified by {*username*} into the post identified by {*id*}.

- **DELETE /post/{*id*}/dislike/{*username*}** Removes an existing like, of the user identified by {*username*} into the post identified by {*id*}.

- **GET /page/thread/{*id*}?ps={*page_size*}&t={*token*}&d={*depth*}** Retrieves a thread: the post identified by *id* and a sub set of its replies, of cardinality *page_size*. It has a recurssive pattern, having also, for each reply, a sub set of their replies, up to a given *depth*. To retrieve more replies of a given post, a continuation *token* must be passed. This token is returned, after a request, if there are more replies to be retrieved. To obtain the first page, the token must be empty. This endpoint can also be called with an id of a reply, to further obtain more levels of (deeper) replies, i.e., replies that are beyond the current request's selected {*depth*}.

- **GET /page/initial?ps={*page_size*}&p={*page_number*}** Retrieves the set of posts that belong to the initial page. It has 2 parameters: *page_size*, that corresponds to the amount of posts of each page, and *page_number*, that corresponds to the number of the page to be retrieved.

- **POST /page/search** Submits a search query into the system and returns a set of posts that match with it.

*2) **CosmosDB**:* Azure CosmosDB is Microsoft's globally distributed, multi-model database service that enables building highly responsive and highly available applications worldwide. It elastically and independently scales throughput and storage across any number of Azure regions, scaling up from thousands to hundreds of millions of requests per second, enabling to deal with unexpected spikes in workloads, without having to over-provision for the peak. Additionally, Cosmos also transparently replicates the data wherever the users are, so they can interact with a replica of the data that is closest to them, decreasing the latency. Therefore, leveraging on Cosmos DB, we can build highly available planet scale applications. Furthermore, Cosmos offers tunable consistency, which enables to not have to make extreme tradeoffs between consistency, availability, partition tolerance [3], since its multi-master replication protocol was carefully designed to offer five well-defined consistency levels on the consistency spectrum: strong, bounded staleness, session, consistent prefix, and eventual. For YAR we opted by Session Consistency, because we want to maintain causality of events, for instance, when a reply is added to a post, the client should always be able to read that post (assuming no delete occurred).

A Cosmos database consists of a set of Cosmos containers, which serve as the logical units of distribution and scalability. In each region, data within a container is partitioned and distributed by using a provided partition-key (local distribution), and each partition is also replicated across geographical regions (global distribution). The containers, as well as their partition-keys, utilized in our system are presented in table I.

TABLE I
YAR'S COSMOSDB CONTAINERS

Container	Partition-Key
Communities	name
Users	name
Posts	community
Likes	post_id

*3) **BlobStorage**:* Azure Blob Storage is Microsoft's storage solution for the cloud, optimized for storing massive amounts of unstructured data, i.e., data that does not adhere to a particular data model or definition, such as text or binary data (e.g. images). In a BlobStorage, data are organized into containers, that consist on a set of blobs, similar to a directory in a file system. It supports three types of blobs: Block Blobs, that store blocks of data that can be managed individually, Append Blobs, that also store blocks but are optimized for append operations, and Page Blobs, that store random access files (virtual hard drive (VHD) files and serve as disks for Azure virtual machines).

We resorted to a BlobStorage to save the images uploaded to YAR, in a container of Block Blobs.

*4) **Redis**:* Redis [4] is an in-memory data structure key-value store. Among many uses cases, Redis is typically used in Web Services as an application level cache, to improve the performance of systems that rely heavily on backend storage (databases or blobstorages), by temporarily copying frequently accessed data to fast storage located close to the application.

For our application, we have decided to cache the following information:

- **Most Recently Accessed Communities**: To enable faster verification if a community exists upon creating a new post.

- **Most Recently Accessed Users**: To enable faster verification if a user exists upon creating a new post.

- **Most Recently Accessed Images**: To enable faster retrieval of popular images.

- **Most Recently Accessed Posts**: To enable faster retrieval of popular posts.

- **Most Recently Accessed Replies Pages**: To enable faster retrieval of popular replies pages of a given post.

- **Approximate Number of Total Likes**: To enable faster retrieval of the number of likes of a given post. Stored in HyperLogLogs.

- **Approximate Number of Dayly Likes**: To enable faster retrieval of the number of likes, in the last 24h, of a given post. Stored in HyperLogLogs. Used to compute the initial page.

- **Approximate Number of Total Replies**: To enable faster retrieval of the number of replies of a given post. Stored in HyperLogLogs. Used to compute the initial page.

- **Approximate Number of Dayly Replies**: To enable faster retrieval of the number of replies, in the last 24h, of a given post. Stored in HyperLogLogs. Used to compute the initial page.

- **Initial Page**: Stores the initial page, with all its sub-pages (page 1, 2, ...) for faster retrieval.

*5) **AzureFunctions**:* Azure Functions is a service for easily running just the code, or "functions", needed for the problem at hand, without worrying about a whole application or the infrastructure to run it in the cloud. Additionaly, they can be developed in a panoply of programming languages, and thus make development even more productive. This functions are triggered by events, which start their execution.

For YAR, we defined the following functions:

- **Compute Inital Page**: Periodically (every 24h) recomputes the initial page and inserts it in the cache. The algorithm used to compute the page is presented in III-B.
- **Clear HyperLogLogs**: Since we resorted to hyperlogs to estimate the total number of likes they need to be recomputed when there is a dislike. Additionaly, the dayly hyperlogs must be reset every 24h.
- **Geo-Replica Blob Storage**: To support geo-replication with writes on both regions, we resorted to a function that copies blobs written to a particular region, to the other region's BlobStorage.

*6) **Cognitive Search**:* Azure Cognitive Services enables to easily add cognitive features into applications. We resorted to this to support advanced search on the contents of posts.

Algorithm 1: Get Thread

```
Function GetThread($id$,$depth$,$page\_size$,$cont\_token$):
  post ⟵ getPost($id$); // Get post from DB
  queue ⟵ {post};
  current_level ⟵ 0;
  amount_posts_cur_lvl ⟵ 1;
  While #queue> 0 do:
    current_post ⟵ poll(replies); // From DB
    amount_posts_cur_lvl ⟵ amount_posts_cur_lvl;
    ($cont\_token$, replies) ⟵
    post_id ⟵ current_post.getId();
      getReplies(post_id, $cont\_token$, $page\_size$);
  current_post.setReplies(replies);
  current_post.setContToken($cont\_token$);
  total_likes ⟵ getTotalLikes(p);
  current_post.setLikes(total_likes);
  If current_level<depth do:
    queue ⟵ queue ∪ replies;
  If amount_posts_cur_lvl= 0 do:
    current_level ⟵ current_level+1;
    amount_posts_cur_lvl ⟵ #queue;
  Return post;
```

III. Implementation

In this section, we introduce the implementation details that we think are worth being highlighted.

A. Continuation Tokens on Replies

(Explicar o pseudocódigo)

The algorithm to retrieve a thread is presented in algorithm 1.

When fetching replies from a given post, only a subset (of $page\_size$ size) is returned, along with a continuation token. In order to retrieve the next replies, the continuation token must be passed as a parameter in another call to this function. This continuation token is generated by cosmosDB, upon returning a result for a query, when there are more docments to be returned. However, this token is in JSON format, and, in order to be passed as a HTTP query parameter, we encoded it using our variant of Base64 representation (Base64 representation where the character '/' is replaced by '-'). After retrieving a subset of replies, if the current depth is belong $depth$, the replies are queued to be process later. Then, the number of likes of the current post being processed is retrieved. Upon finish processing the current post, the next one is fetch from the queue and the process continues until the maximum $depth$ is reached. Finally, a tree of posts, with appended number of likes and respective continuation tokens is returned.

B. Initial Page Computation

The algorithm to compute the initial page is presented in algorithm 2. It traverses all the posts on the database and keeps the ones with the highest scores in a sorted set, in order to maintain them organized by their score. Upon evaluating a post, first, its score is computed regarding simultaniously several metrics: popularity (many likes and/or replies), freshness (how long it was posted), trending (was recently accessed), and hotness (many likes and/or replies in the last 24h). Then, if the set is not full, the posts are simply added to it. However, when it is full, if the current post being considered has higher score than the post with minimum score on the set, it replaces it. Otherwise, if they both have the same score, the later replaces the first with 50% probability. Otherwise, the current post is discarded, and the next one is fetched.

IV. Evaluation

A. Experimental Setting

In order to evaluate the performance of our system, we resorted to artillery [5], a load-testing toolkit, written in JavaScript.

We tested our service on the following scenarios:

- **Random Reader**: In this scenario, we test our service based on normal user interaction. A user first requests the initial page and all it's media components (after getting the initial page data, a sequence of requests blobs (images) occur). We then simulate normal user browsing through randomly selecting a few posts to interact with and advancing in the initial page's posts list, with some probability. With the randomly selected posts, we either trigger a like on that post, post a reply with an image or not (also randomly decided) or do both.

Algorithm 2: Initial Page Computation

```
Function ComputeInitialPage($page\_size$, $page\_number$,
        $MAX\_INITIAL\_PAGE\_POSTS$):
    set_size ⟵ min($page\_size$*$page\_number$,
            $MAX\_INITIAL\_PAGE\_POSTS$);
    selected_posts ⟵ {}; // Sorted Set
    posts ⟵ getAllPosts(); // Get all posts from DB
    Foreach $p \in$ posts do:
        score ⟵ getScore($p$);
        If #selected_posts $< set\_size$ do:
            selected_posts ⟵ selected_posts ∪ {(score , $p$)};
        Else:
            x ⟵ min{selected_posts};
            If $\pi_1$(x) $<$ score do:
                selected_posts ⟵ selected_posts / {x};
                selected_posts ⟵ selected_posts ∪ {(score , $p$)};
            Else If $\pi_1$(x) $=$ score do:
                If random() $\leq 0.5$ do:
                    selected_posts ⟵ selected_posts / {x};
                    selected_posts ⟵ selected_posts ∪ {(score , $p$)};
    Return { $p$ : $\exists_{e \in selected\_posts}$( $p = \pi_2(e)$ ) };

Function getScore($p$):
    popularity ⟵ getPopularity($p$);
    freshness ⟵ getFreshness($p$);
    trending ⟵ getTrending($p$);
    hotness ⟵ getHotness($p$);
    Return 0.12∗popularity+0.24∗freshness
        +0.24∗trending+0.4∗hotness;

Function getPopularity($p$):
    total_replies ⟵ getTotalReplies($p$);
    total_likes ⟵ getTotalLikes($p$);
    Return max(0.8∗total_likes+0.2∗total_replies,
            0.2∗total_likes+0.8∗total_replies);

Function getFreshness($p$):
    days ⟵ (now() $- p.getCreationTime() + 1)/(24 * 60 * 60)$;
    If days $< 0.3$ do:
        days ⟵ 0.3;
    Return 100/days;

Function getTrending($p$):
    Return isInCache($p$) ? 100 : 0;

Function getHotness($p$):
    dayly_replies ⟵ getDaylyReplies($p$);
    dayly_likes ⟵ getDaylyLikes($p$);
    Return max(0.8∗dayly_likes+0.2∗dayly_replies,
            0.2∗dayly_likes+0.8∗dayly_replies);
```

- **Troll**: This a simple scenario where all posts from the initial page are selected and then we post a reply with an image.
- **Create Root Post**: This a simple scenario where we simply test the creation of thread's initial post.
- **Searcher**: The Searcher scenario simulates a user that searches for something (a combination of three random words), and then evaluates the results and may or interact with some of the posts that compose the querie's result.

B. *Experimental Results*

After running our testing the results in terms of functionlanity were very satisfactory. Unfortunatly we do not have any data to present nor this nor the throughput. We ran our tests proior to the message by the teacher on how to correctly aggregate the results; by the time we tryed to apply those changes we had already ran out of money on both of our subscriptions.

V. Final Remarks

In this project, we developed YAR, a web application inspired by Reddit, that was implemented leveraging on Microsoft's Cloud Services: Azure. We deployed our solution in a geo-replicated setting, in West Europe and Central USA, and evaluated its performance in both regions. After developing this project we gained insight into how companies use Cloud Platforms to build their Services: selecting and linking different resources, configure deployments, managing costs. One of the most important things that we learned was that Cloud must be treated with a lot of caution and thoughts, despite all the benefits it has Cloud computing may become very expensive, even to the point that it may become no longer viable.

References

[1] "Reddit the front page of the internet," https://www.reddit.com/, accessed: 2019-11-23.

[2] "Microsoft Azure," https://azure.microsoft.com/, accessed: 2019-11-23.

[3] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/564585.564601

[4] "Redis," https://redis.io/, accessed: 2019-11-23.

[5] "Artillery," https://www.npmjs.com/package/artillery, accessed: 2019-11-24.