RadarScanner@Cloud

Cloud Computing and Virtualization
Project - 2020-21
MEIC / METI / MECD - IST - ULisboa

1 Introduction

The goal of the CCV project is to design and implement a simulated radar scanning service hosted in an elastic cluster of web servers. The service executes a computationally-intensive task that simulates radar scanning (e.g., by a drone) and whose goal is to identify and highlight regions of a map which, similarly to radars and sonars in real-life, are limited by both range and boundaries. Regions such as mountains or frontiers of a map should be identified on-demand, following a set of scanning strategies (implemented as simple map covering algorithms), and the area reached by the scanning highlighted in the map.

The system will receive a stream of web requests from users. Each request is to perform scanning on a given terrain (map), based on a source point (the radar location initial coordinates), and the maximum range intended for the scan (for convenience, horizontal and vertical). A radar scanning can be performed using a number of strategies. In this project we make use of three specific ones for demonstrative purposes and to drive the simulation of a computationally intensive task that will be executed using cloud resources. The example algorithms using simple `Java` code include: GRID_SCAN (a brute force approach that scans every point/location within range to assess whether it is reachable to the radar scan), PROGRESSIVE_SCAN (an incremental strategy that increases the distance of points/locations scanned to assess, until it reaches maximum range or finds obstacles/boundaries), and GREEDY_RANGE_SCAN (a strategy that initially scans farther way points/locations to assess their reachability, thus establishing as fast as possible whether a sizable region is within radar range and not hidden by obstacles/boundaries).

Each request can result in a task of varying complexity to process, as different scanning approaches will take different number of steps to explore each of the different terrain maps. To achieve scalability, good performance and efficiency, the system should attempt to optimize the selection of the cluster node for each incoming request and to optimize the number of active nodes in the cluster.

This project specification is accompanied by a Frequently Asked Questions Document, to be made available at: https://tinyurl.com/cnv-faq-20-21

2 Architecture

The *RadarScanner@Cloud* system will be run within the Amazon Web Services ecosystem. The system (see Figure 1) will be organized in four main components:

- **Web Servers** receive web requests to perform radar scanning tasks, i.e. assessing in a map image which points/locations are reachable from the starting (source) coordinates of the radar scan (i.e., within range and not beyond any boundary). In *RadarScanner@Cloud*, there will be a varying number of identical web servers. Each will run on an AWS Elastic Compute Cloud (EC2) instance;
- **Load Balancer** is the entry point into *RadarScanner@Cloud* system. It receives all web requests, and for each one, it selects an active web server to serve the request and forwards it to that server;
- **Auto-Scaler** is in charge of collecting system performance metrics and, based on them, adjusting the number of active web servers;
- **Metrics Storage System** uses one of the available data storage mechanisms in AWS to store web server performance metrics regarding to execution of requests. These will help the load balancer choose the most appropriate web server.

2.1 Web Servers

Web servers in *RadarScanner@Cloud* are system virtual machines running an off-the-shelf `Java`-based web server application on top of Linux. The web server application will serve a single web page that receives an HTTP request providing the necessary information, i.e., the scenario to be scanned (a given image map of a given size and layout), the coordinates (xS, yS) for the source position of the radar scan, the top-left (x0,

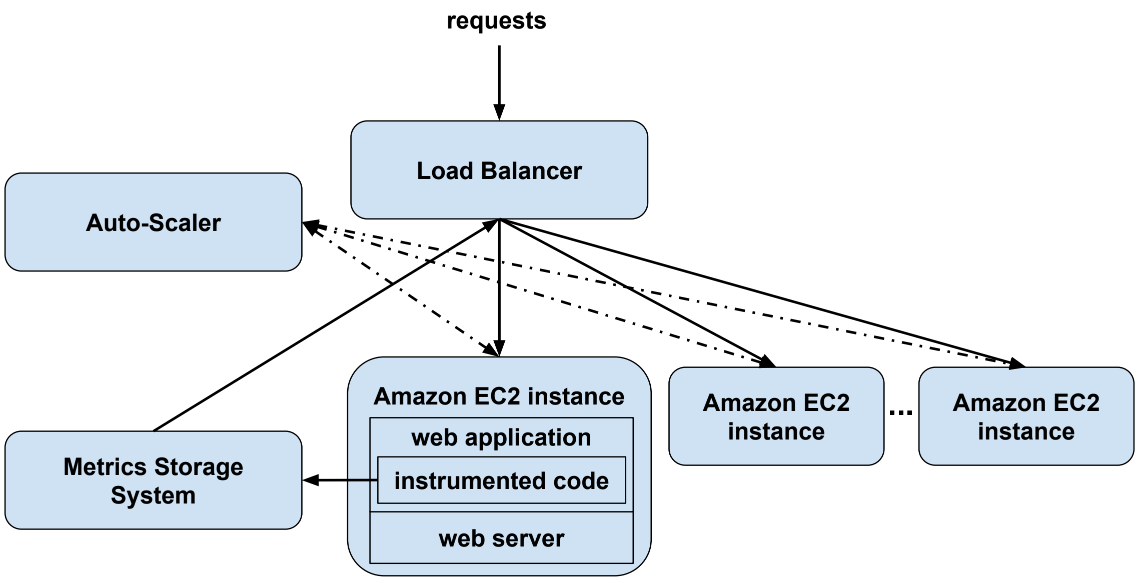

Figure 1: Architecture of *RadarScanner@Cloud*

y0) and bottom-right (x1, y1) corners of the maximum range of scanning area, and the scanning strategy (GRID_SCAN, PROGRESSIVE_SCAN, GREEDY_RANGE_SCAN) to use. The page serving the requests will perform the scanning and assessing online and, once it is complete, reply to the web request. On success, the reply shows the map updated with the scanned area within radar range highlighted and an IST flag on the radar source position.

2.2 Load Balancer

The load balancer is the only entry point into the system. It receives web requests and selects one of the active web server cluster nodes to handle each of the requests. In a first phase, this job can be performed by an off-the-shelf load balancer such as those available at Amazon AWS. Later in the project, you should design a more advanced load balancer that uses metrics obtained in earlier requests (stored in the Metrics Storage System, MSS) to pick the best web server node to handle a request.

The load balancer can estimate the approximate complexity and workload of a request based on the request's parameters combined with data previously stored in the MSS. The load balancer may know which servers are busy, how many and what requests there are currently executing, what the parameters of those requests are, their current progress and, conversely, how much work is left taking into account a cost estimate that was calculated when the request arrived.

2.3 Auto-Scaler

For this project, you will design an auto-scaling component that adaptively decides how many web server nodes should be active at any given moment. Students should design the auto-scaling rules that will provide the best balance between performance and cost (initially, it can be a simple AWS Auto-scaling group). The Auto-Scaler should detect that the web app is overloaded and start new instances and, on the other hand, reduce the number of nodes when the load decreases.

2.4 Metrics Storage System

The *RadarScanner@Cloud* system should also include a Metrics Storage System (MSS) that stores load and performance metrics collected from the web server cluster nodes. These nodes will process the *RadarScanner@Cloud* requests using code that was previously instrumented by the students, in order to collect relevant dynamic performance metrics regarding the application code executed (e.g. number of bytecode instructions or basic blocks executed, data accesses, number of function calls executed, invocation stack depth, and/or others deemed relevant). These metrics allow estimating the task complexity realistically, irrespective of variable wall-clock time delays, that can be caused by frequent resource overcommit, incurring VM slowdown, done by the cloud provider.

The final choice of the metrics extracted, instrumentation code, and system used to store the metrics data is thus subject to analysis and decision by the students. Students should consider the usefulness/overhead trade-offs of all utilized metrics. The selected storage system can be updated directly or you may resort to some intermediate transfer mechanism. For realism, you must take into account that continuously querying this storage system may eventually become a bottleneck for the load balancer component and resource overuse.

2.4.1 Code Instrumentation

The code of the application that performs map radar scanning (invoked by the web server) is written in the `Java` programming language and compiled into bytecode. The application is to be further instrumented with a `Java` instrumentation tool (`BIT`) in order to extract and persistently store the dynamic performance metrics regarding the code being executed. The explicit duration (wall-clock time) of each request handled should not be considered or stored in the MSS.

2.5 Implementation

The system and all its parts could also be deployed on a single machine, but instead, you will be using Amazon Web Services. AWS provides components for deciding autoscaling, storing data, running computing instances and load balancing. It is up to each individual group to decide how and where (e.g., in which VM) to implement and deploy each of the solution's components.

3 Checkpoint

For the initial checkpoint, students should submit the *RadarScanner@Cloud* system with a running and instrumented web server, load balancer and auto-scaler. Note that the algorithms for load balancing, auto-scaling and the MSS need not be fully implemented at this stage (you can use AWS ELB and AutoScale and metrics can be stored temporarily in the computing nodes), but it is expected that some logic is already developed and, at the very least, they should be already thought out. The code submitted for the checkpoint will be evaluated on the following labs. The checkpoint submission bundle must include an intermediate report (2-page, double column) clearly describing: a) what is already developed and running in the current implementation (architecture, data structures and algorithms); and, b) the specification of what remains to be implemented or completed. A first bubmission should be made in the Fénix system until 23:59 on April 27th, 2021. If needed, updates can be submitted until 23:59 on April 30th, 2021.

4 Final Submission

The final submission should include all the checkpoint features and additionally:

- The connection between the web server instrumentation and the MSS.
- An adequate auto-scaling algorithm that aims to balance cost and performance efficiently.
- An adequate load balancing algorithm that uses the metrics collected in the MSS.

Student groups should submit the solution's code and a report (up to 6 double column pages) describing the implemented solution, clearly explaining and justifying the algorithm design and tuning decisions, as well as any measurements and analysis that support the design decisions and configurations.

Groups are encouraged to provide information regarding queries to the system with enough load to trigger the auto-scaling mechanism.

The code of the final submission should be well organized and adequately commented.

The final submission should be made in the Fénix system until 23:59 on May 25th, 2021.