Sudoku@Cloud

Cloud Computing and Virtualization
Project - 2019-20
MEIC / METI / MECD - IST - ULisboa

1 Introduction

The goal of the CCV project is to design and develop an elastic cluster of web servers that is able to execute a computationally-intensive task to discover the solution of a sudoku puzzle,[1] on-demand, by executing a set of Sudoku puzzle solving algorithms.

The system will receive a stream of web requests from users. Each request is to solve a given sudoku puzzle comprising a map (grid) with various number of elements missing that need solving. Sudoku puzzles can be solved using many algorithms. 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: BFS (Brute-Force Solver), DLX (Dancing Links), CP (Constraint Programming).

Each request can result in a task of varying complexity to process, as different solving approaches will take different number of steps to find the missing elements in the sudoku puzzle.

To have scalability, good performance and efficiency, the system will 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-19-20

2 Architecture

The *Sudoku@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**: The web servers receive web requests to perform puzzle solving, discovering missing elements in the grid and return the solved puzzle. In *Sudoku@Cloud* , there will be a varying number of identical web servers. Each will run on a rented AWS Elastic Compute Cloud (EC2) instance.
- **Load Balancer**: The load balancer is the entry point into the *Sudoku@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**: The auto-scaler is in charge of collecting system performance metrics and, based on them, adjusting the number of active web servers.
- **Metrics Storage System**: The metrics storage system will use one of the available data storage mechanisms at AWS to store web server performance metrics relating to requests. These will help the load balancer choose the most appropriate web server.

2.1 Web Servers

The *Sudoku@Cloud* web servers 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 a HTTP request providing the necessary information, i.e., the puzzle template (a given sudoku map of a given size), the solver strategy (BFS - Brute-Force Solver, DLX - Dancing Links, CP - Constraint Programming) to use, and the top threshold for the position of the last missing entry, driven by a series of prime numbers (with the default being the total number of entries of the puzzle). The page serving the requests will perform the solving online and, once it is complete, reply to the web request with a confirmation, and if successful by drawing the complete sudoku puzzle (i.e. solved).

[1] https://en.wikipedia.org/wiki/Sudoku

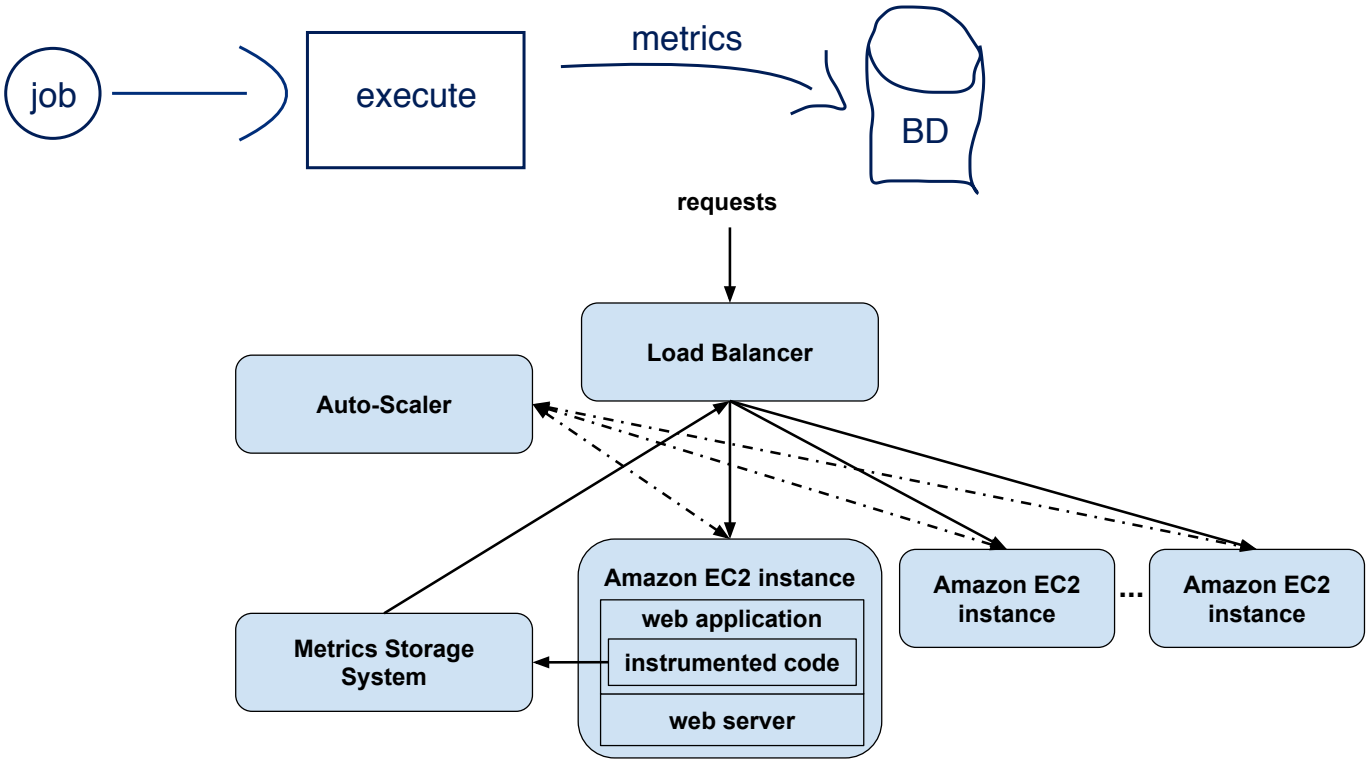

Figure 1: Architecture of *Sudoku@Cloud*

2.2 Load Balancer

The load balancer is the only entry point into the system: it receives a sequence of 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 data obtained in earlier requests, stored in the Metrics Storage System, to pick the best web server node to handle a request.

The load balancer can estimate the complexity, load and approximate duration of a request, based on the request's parameters combined with data previously stored in the MSS, that may be periodically or continuously updated by the MSS. The load balancer may know which servers are busy, how many and what requests they are currently handling, 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. It is up to the students the challenge to 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). It should detect that the web app is overloaded and start new instances and, conversely, reduce the number of nodes when the load decreases.

2.4 Metrics Storage System

The *Sudoku@Cloud* system will include a metrics storage system (MSS) that will store load and performance metrics collected from the web server cluster nodes. These nodes will process the *Sudoku@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 by the students from their analysis, by exploring the outcome of different instrumentations for typical requests). They will allow estimating 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 free and subject to analysis and decision by the students, regarding usefulness/overhead tradeoffs. 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 the sudoku solving (called 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 (such as BIT that will be presented in the labs) in order to extract and persistently store the dynamic performance metrics regarding the code 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 any of its parts could also be implemented on a single machine, but you will be using an Amazon Web Services (AWS) account. 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 checkpoint, students should submit the *Sudoku@Cloud* system with a running and instrumented web server, load balancer and auto-scaler. The algorithms for load balancing, auto-scaling and the MSS need not be fully implemented at this stage (metrics can be stored temporarily in the computing nodes) but, expectedly, some logic should already be 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) describing clearly: 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. Submission should be made in the Fenix system until 23:59 on April 24th, 2020.

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 Fenix system until 23:59 on May 18th, 2019.