Lezione 1

Parte I : deployment di una applicazione web su EC2

Applicazione di esempio

Consideriamo una semplice applicazione web, *FlaskPlot*, che permette di creare grafici di funzioni matematiche. L'applicazione è scritta in Python usando il framework *Flask*[1]. Si tratta di una semplice applicazione 'giocattolo' (funzionalita' minimali, poco robusta, ...).

Per il deployment dell'applicazione durante la lezione useremo il server di sviluppo fornito da Flask:

- Semplice da configurare
- **Da non usare in scenari reali!!**
- Per approfondire: https://flask.palletsprojects.com/en/1.1.x/tutorial/deploy/

Per avviare il server con la nostra applicazione bastano 2 comandi:

```
$ export FLASK_APP=flaskplot.py
$ flask run -h 0.0.0.0 -p <numero di porta>
```

Deployment su EC2

Per iniziare, useremo una macchina virtuale del servizio EC2 di AWS per installare la nostra applicazione web.

- Selezioniamo la **regione** AWS da utilizzare (e.g., Francoforte)
- Creiamo una **Key Pair**: coppia di chiavi pubblica-privata che servira' per collegarci in SSH alla macchina virtuale
  - La chiave pubblica viene installata in automatico sulle istanze
  - La chiave privata deve essere conservata in maniera sicura sul proprio PC
  - Non e' possibile scaricare più di una volta la chiave privata da AWS
- Avviamo una istanza EC2
  - **AMI**: come immagine di partenza possiamo usare Amazon Linux 2
  - Usiamo un'istanza poco costosa (e.g., t2.nano/micro)
  - Creiamo un **security group** che lasci passare tutto il traffico SSH e HTTP
  - Lasciamo le altre impostazioni al valore di default
- Possiamo ottenere informazioni sulla nuova istanza dalla dashboard di EC2 (e.g., stato dell'istanza, IP pubblico, ...)
- Colleghiamoci in SSH alla nuova istanza (come utente **ec2-user**)

```
$ ssh -i <chiave privata.pem> ec2-user@<Public IP/Public DNS>
```

- Installiamo il software di cui abbiamo bisogno (i.e., Python e Flask) per eseguire FlaskPlot:

```
$ sudo yum install python3
$ sudo pip3 install flask
$ sudo pip3 install matplotlib
```

- Copiamo sulla macchina la nostra applicazione web con `scp`:

```
$ scp -i <chiaveprivata.pem> -r <cartellalocale> \
  ec2-user@<istanza ec2>:/home/ec2-user/
```

[1] `https://flask.palletsprojects.com/en/1.1.x/`

- Avviamo l'applicazione:

```
$ cd flaskplot/
$ sh run.sh
```

- Proviamo a collegarci con un browser.

Parte II : deployment con EC2 e Load Balancer

La configurazione attuale non è assolutamente scalabile né tollerante ai guasti. Vogliamo estendere la nostra infrastruttura prevedendo la possibilità di avere più repliche del web server. Useremo il servizio di *load balacing* di AWS per distribuire le richieste in arrivo alle varie repliche.

Operazioni preliminari

Creiamo un semplice servizio *systemd* per avviare il server automaticamente all'avvio. Creiamo il file `/etc/systemd/system/flaskplot.service`:

```
[Unit]
Description=Simple systemd service for FlaskPlot.

[Service]
Type=simple
WorkingDirectory=/home/ec2-user/flaskplot
ExecStart=/bin/bash /home/ec2-user/flaskplot/run.sh

[Install]
WantedBy=multi-user.target
```

Abilitiamo e avviamo il servizio:

```
$ sudo systemctl daemon-reload
$ sudo systemctl start flaskplot.service
$ sudo systemctl enable flaskplot.service
```

Creiamo anche una **AMI** a partire dall'istanza EC2 corrente: in questo modo sarà sufficiente avviare delle istanze EC2 a partire dall'AMI creata per avere subito l'applicazione configurata.

Attenzione: ogni AMI creata viene associata ad uno *snapshot* del disco virtuale dell'istanza. Conservare degli snapshot a un costo: `https://aws.amazon.com/premiumsupport/knowledge-center/ebs-snapshot-billing/`

Virtual Private Cloud

AWS ci permette di avere un controllo a grana fine sulla rete (virtuale) che le nostre istanze e gli altri servizi useranno per comunicare tra loro e con il resto di Internet. Se non richiesto esplicitamente, tutte le istanze EC2 condividono la stessa sottorete virtuale.

- Creiamo un **Virtual Private Cloud (VPC)** nella regione corrente
- Al VPC viene associato un blocco di indirizzi IP (privati)
- Come blocco di indirizzi IPv4 possiamo scegliere qualcosa come `10.0.0.0/16`
- Nel VPC, avremo bisogno di definire delle **subnet** in ogni Availability Zone (AZ) che vogliamo usare
- Scegliamo una AZ, e creiamo una subnet (e.g., `10.0.1.0/24`)

- Se necessario, modifichiamo la subnet abilitando l'assegnazione automatica di IP pubblici IPv4 per le istanze create nella subnet
- Creiamo un **Internet Gateway** (IG), che consente l'accesso ad Internet per le istanze in esecuzione nel VPC, ed associamolo al nostro VPC
- Creiamo una **Route Table** per il nostro VPC ed associamola alle sottoreti create
- Aggiungiamo una regola di routing per indirizzi `0.0.0.0\0` con target l'IG

Creiamo dei **security group** (SG), dei firewall virtuali per i nostri servizi:

- Un SG per il load balancer: in ingresso accettiamo traffico HTTP da qualsiasi IP sorgente
- Un SG per i web server: in ingresso accettiamo HTTP **solo dal SG** del LB (e, se vogliamo, il traffico SSH dall'esterno)

Load Balancer

- 3 tipi di LB: Application LB, Network LB e Classic LB
- Per semplificare leggermente la configurazione, useremo il *Classic LB*
- Creiamo un LB in ascolto sulla porta TCP 80 che usa il SG definito in precedenza
- Per l'health check chiediamo di usare il protocollo HTTP, sulla porta 80, con richieste con URL `/`
- Avviamo delle istanze EC2, usando la AMI creata in precedenza, chiedendo che siano avviate nella nostra sottorete, usando il SG descritto sopra
- Dalla dashboard del LB, registriamo le nuove istanze EC2 con il LB
- NOTA: non e' necessario che le istanze EC2 abbiano un IP pubblico
- Attendiamo un paio di minuti prima di collegarci al nome DNS del LB

Parte III: deployment elastico con EC2 Auto Scaling

Il servizio di Auto Scaling consente di automatizzare l'avvio e lo spegnimento di istanze appartenenti allo stesso "gruppo" (e.g., per rispondere a variazioni nel carico dell'applicazione).

Configurazione del servizio di auto scaling

- Per prima cosa definiamo una *Launch Configuration* a partire dall'AMI creata in precedenza
- Creiamo l'Auto Scaling Group basato sulla Launch Configuration appena definita
- Specifichiamo che le istanze devono essere avviate nelle sottoreti del nostro VPC
- Possiamo scegliere qui una politica di scaling da utilizzare
- Per un semplice test, chiediamo di avere sempre 2 istanze attive
- Modifichiamo l'Auto Scaling Group, associandolo all'istanza di Classic LB creata in precedenza
- Verifichiamo che 2 istanze EC2 vengono create automaticamente

Benchmarking con *ab*

Per effettuare dei semplici test possiamo utilizzare il tool **ab**.[2]

Esempio (100 richieste GET con un limite di 10 richieste in parallelo):

```
ab -c 10 -n 100 <URL>
```

Per effettuare delle richieste di tipo POST:

[2] `https://httpd.apache.org/docs/2.4/programs/ab.html`

```
ab -p post.data  -c 10 -n 200 <URL>/plot
```

Il file `post.data` contiene i parametri della richiesta POST, codificati in maniera opportuna per il protocollo HTTP, e.g.:

```
function=x%2A%2A2&name=gabriele
```

Esercizi

Esercizio 1: ripetere la configurazione con il Load Balancer AWS a livello applicativo di nuova generazione, e utilizzando almeno 2 Availability Zones per migliorare la tolleranza ai guasti.

Esercizio 2: configurare una politica di Auto Scaling ragionevole, ed utilizzare `ab` per generare carico per l'applicazione. Verificare che il numero di istanze venga adattato in caso di variazioni nel livello di carico.
Monitorare il numero di richieste effettuate nel tempo, il numero di istanze in esecuzione, e delle metriche significative per valutare le prestazioni del sistema (e.g., tempo di risposta dell'applicazione).

Esercizio 3: modificare l'applicazione per fare in modo che lo stato interno (e.g., il conteggio dei grafici generati) non sia memorizzato localmente nell'application server, ma utilizzando un (semplice) servizio di memorizzazione di dati (e.g., Redis). Adattare di conseguenza l'infrastruttura Cloud utilizzata per il deployment, prevedendo la presenza di uno o più nodi per il deployment del nuovo servizio.