ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
(SEDE DI BOLOGNA)

Anno Accademico 2024/2025

RELAZIONE DI FINE TIROCINIO CURRICULARE

svolto dallo studente:

Massimo Valerio Zerbini (matricola: 0000969932)

iscritto al Corso di Studio in:

Ingegneria Informatica T (codice: 9254)

presso:

ULISSE — Dipartimento di Informatica – Scienza e Ingegneria (DISI)

sul seguente argomento:

Integrazione di funzionalità su infrastruttura virtuale SLURM

Studente:

_____
(firma)

Tutor Accademico &
Referente Struttura Ospitante:
Prof. Marco Prandini

_____
(firma per approvazione della relazione finale)

Indice

Capitolo 1

Introduzione

1.1 Obiettivi

L'attività di tirocinio svolta presso il gruppo Unibo Laboratory of Information and System Security (ULISSE) del Dipartimento di Informatica – Scienza e Ingegneria (DISI) ha avuto come obiettivo lo sviluppo di un ambiente virtuale distribuito per l'integrazione di funzionalità relative al settore del High Performance Computing (HPC). In particolare, si è richiesta l'implementazione di:

- **federazione di *cluster***, ossia il coordinamento tra molteplici infrastrutture per l'esecuzione di calcoli;
- **condivisione delle risorse di computazione** disponibili, al fine di raggiungere il pieno utilizzo della potenza di calcolo dei nodi;
- **priorità di *scheduling*** per un determinato utente, ristretta a una risorsa di computazione specifica.

1.2 Tecnologie coinvolte

Le competenze richieste per la realizzazione del progetto sono basate principalmente sul corso di Amministrazione di Sistemi. L'attività di studio e documentazione sui nuovi strumenti è stata autonomamente svolta online. Di seguito vengono presentate le principali tecnologie utilizzate.

1.2.1 Git

Git è un Version Control System (VCS) distribuito, in grado di registrare i cambiamenti durante lo sviluppo di un progetto, mantenendone la cronologia. È caratterizzato da una gestione efficiente e scalabile delle informazioni. Il principale vantaggio

nell'uso dei VCS è dato dal forte supporto allo sviluppo non lineare, che permette la diramazione ("*branching*") e la successiva fusione ("*merging*") di nuove funzionalità.

Git si basa sul concetto di *repository*, ovvero una "copia" del progetto. Per la collaborazione tra sviluppatori viene spesso utilizzato un repository remoto, sempre accessibile; il singolo sviluppatore lavora (anche offline) sulla propria copia locale e, quando lo ritiene opportuno, aggiorna il repository remoto. GitHub e GitLab sono esempi di servizi che offrono hosting di repository remoti.

1.2.2 Vagrant

Vagrant è un software di gestione di Virtual Machine (VM) focalizzato sulla riproducibilità affidabile degli ambienti virtualizzati. Dipende da programmi di virtualizzazione esterni (detti "*provider*"), come VirtualBox e VMware, ma risulta portabile rispetto a essi.

L'immagine di una VM di base (denominata "*box*") è facilmente reperibile da un deposito online contenente un'ampia varietà di sistemi operativi. Vagrant fa utilizzo di un file di configurazione (`Vagrantfile`) per ottenere i parametri di esecuzione.

1.2.3 Ansible

Ansible è un software di configurazione automatica di ambienti informatici, nota come Infrastructure as Code (IaC). Ansible lavora in architettura "*agentless*", dove il controllore comunica ed esegue le operazioni necessarie ("*provisioning*") su uno o più nodi oggetto, mediante una semplice connessione Secure Shell (SSH) e senza quindi l'esigenza di software specifico sui nodi da configurare (escludendo il server SSH).

Ansible utilizza i "*playbook*", un elenco di istruzioni da eseguire che coinvolgono i nodi destinatari. I playbook sono scritti in linguaggio Yet Another Markup Language (YAML), di facile interpretazione.

1.2.4 MariaDB

MariaDB è un Database Management System (DBMS) relazionale, utilizzato per l'interfacciamento con un Database (DB). MariaDB è basato su MySQL, estendendone le funzionalità e migliorandone le prestazioni.

1.2.5 MUNGE

MUNGE Uid 'N' Gid Emporium (MUNGE) è un servizio di autenticazione per la creazione e validazione di credenziali utente, progettato appositamente per l'uso in ambienti HPC. La codifica e decodifica delle credenziali si basa su una chiave comune, condivisa ai soli nodi fidati.

1.2.6 SLURM

Simple Linux Utility for Resource Management (SLURM) è un software di gestione e coordinamento di nodi di calcolo, finalizzato alla pianificazione e l'esecuzione di programmi ("*jobs*"). SLURM è altamente scalabile, rendendolo adatto anche in contesti HPC a elevato uso di risorse. Di default utilizza MUNGE per l'autenticazione dei nodi.

SLURM integra tre funzioni principali:

- allocazione esclusiva (e non) delle risorse computazionali agli utenti, per un periodo di tempo configurabile;
- inclusione di un *framework* per la sottomissione e il monitoraggio dell'esecuzione di programmi;
- gestione della contesa di risorse mediante una coda di esecuzione.

È possibile includere funzionalità aggiuntive, quali registrazione delle attività ("*accounting*") e limitazione dell'uso di risorse, tramite il caricamento di *plugin* opzionali.

Architettura

Un cluster SLURM è coordinato dal demone controllore `slurmctld`, il cui compito è gestire le richieste di esecuzione e le risorse disponibili. Ciascun nodo di calcolo esegue `slurmd` per comunicare con il controllore ed effettuare i task richiesti. Il demone `slurmdbd`, connesso a un DBMS, diventa essenziale per la registrazione in un singolo DB delle attività di molteplici cluster.

Federazione

SLURM permette la creazione di federazioni di cluster, dove la sottomissione dei job viene inoltrata a tutti i componenti. Ciascun cluster tenta dunque di portare a termine la richiesta di esecuzione, in modo indipendente. Per il corretto coordinamento all'interno della federazione è necessario che tutte le istanze di `slurmctld` (una per cluster) possano comunicare tra loro, oltre che con la singola istanza di `slurmdbd`, dedicata alla registrazione delle attività.

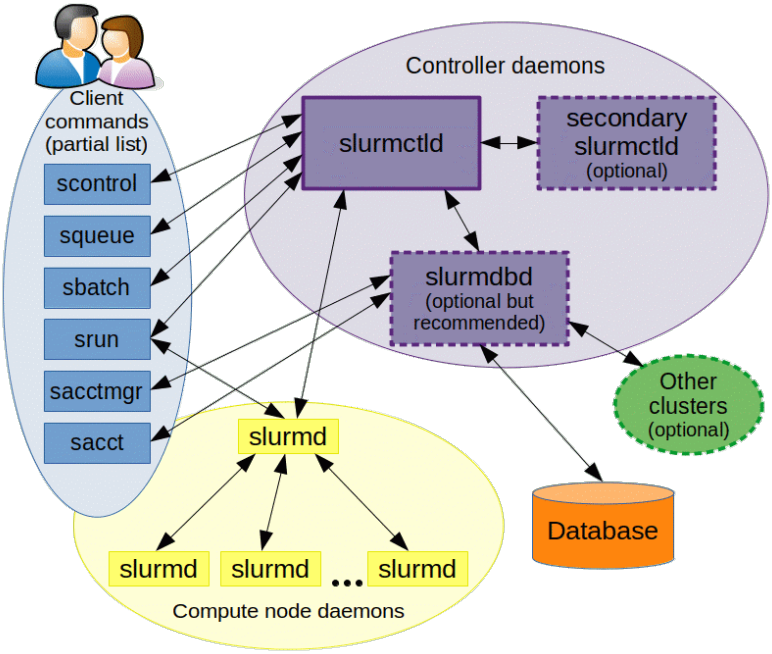

Figura 1.1: Componenti SLURM

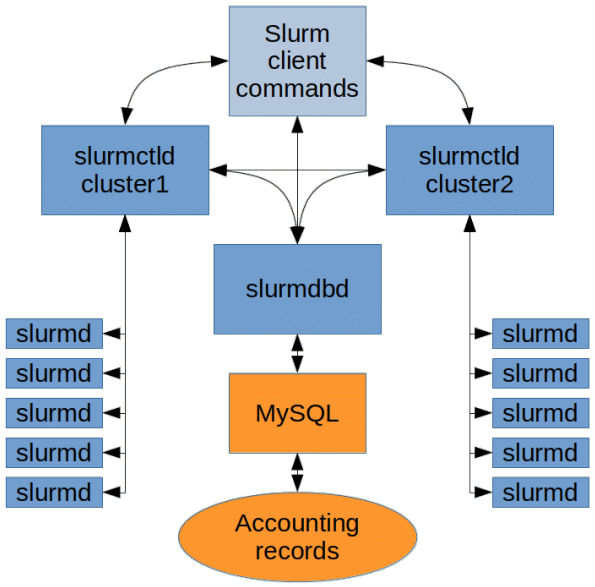

Figura 1.2: Comunicazione in una federazione SLURM

Capitolo 2

Attività svolte

2.1 Repository

Per lo sviluppo del progetto mi è stato concesso l'accesso a un repository Git remoto sulla piattaforma GitLab interna al DISI, autenticandomi da pagina web con le credenziali istituzionali.

Affinché potessi lavorare su copia locale e salvare le modifiche sul repository remoto, è stato necessario generare una chiave asimmetrica SSH e inserire la parte pubblica nel mio account GitLab. In seguito, ho aggiunto una voce *host* al file locale di configurazione SSH, per indicare a `git` come autenticarsi correttamente:

/home/max/.ssh/config

```
...
Host gitlab.disi.unibo.it
  User git
  PreferredAuthentications publickey
  IdentityFile /home/max/.ssh/id_gitlab_unibo
...
```

Ho quindi istanziato una copia locale del progetto ("*cloning*"). Il repository conteneva una prima stesura di un cluster virtuale SLURM, con un nodo controllore e due nodi worker. Come da prassi nello sviluppo software di nuove funzionalità, ho anzitutto generato un nuovo *branch*, denominato "tirocinio", sul quale avrei lavorato da quel momento in poi.

2.2 Risoluzione DNS

Prima di estendere l'infrastruttura con un secondo cluster, ho dovuto verificare il funzionamento di quello già presente. SLURM basa le proprie comunicazioni

sugli *hostname* e non su indirizzi Internet Protocol (IP): ciò implica una corretta risoluzione Domain Name System (DNS), ovvero la "traduzione" da nome a indirizzo IP corrispondente.

Come DNS *resolver* ho scelto `dnsmasq`, un software leggero in grado di svolgere anche la funzione di Dynamic Host Configuration Protocol (DHCP) server, per l'assegnamento automatico degli indirizzi IP in una Local Area Network (LAN). Il programma viene eseguito su `controller`, con indirizzo IP statico `192.168.10.254`, e su cui è presente il file di configurazione:

dnsmasq.conf

```
interface=eth1
no-hosts
no-resolv

# NAMESERVER USED FOR THE INTERNET (host running the VMs)
server=10.0.2.3

dhcp-range=192.168.10.1,192.168.10.253,12h
dhcp-option-force=option:dns-server,192.168.10.254
```

In questo modo i nodi worker (denominati `slurm1` e `slurm2`) ottengono un indirizzo IP all'interno del range, causando la registrazione della corrispondenza *hostname*:*indirizzo*. Nel momento in cui si richiede la risoluzione di un nome di host, `dnsmasq` risponde con l'indirizzo IP associato.

2.3 MariaDB & `slurmdbd`

Essendo prevista la registrazione delle attività (*accounting*) in contesti multi-cluster SLURM, è stata necessaria la configurazione di un DB e relativo DBMS (in questo caso, MariaDB).

Per non esporre credenziali nei file utilizzati da Ansible, sono state definite delle apposite variabili di esecuzione:

secrets.yml

```
mariadb:
  root_password: THIS_IS_THE_ROOT_DB_PASSWORD
  slurm_password: THIS_IS_THE_SLURM_DB_PASSWORD
```

Al fine di semplificare e isolare l'installazione del DB su `controller`, è stato utilizzato il software di containerizzazione `podman`, che recupera ed esegue l'immagine di MariaDB opportunamente configurata:

roles/mariadb/tasks/main.yml

```
...
- name: Pull a MariaDB container
  become: true
  become_user: 'mariadb'
  containers.podman.podman_image:
    name: 'mariadb:latest'

- name: Run MariaDB container
  become: true
  become_user: 'mariadb'
  containers.podman.podman_container:
    name: 'mariadb'
    image: 'mariadb:latest'
    state: started
    ports: "127.0.0.1:3306:3306"
    env:
        MARIADB_ROOT_PASSWORD: "{{ mariadb.root_password }}"
...
```

In questo modo MariaDB risulta in ascolto sulla porta Transmission Control Protocol (TCP) `3306` di `controller`.

Seguendo poi la documentazione SLURM per il corretto interfacciamento tra il DBMS e `slurmdbd`, si generano DB e utente dedicati:

roles/mariadb/tasks/main.yml

```
...
- name: Ensure 'slurm_db' database is present
  community.mysql.mysql_db:
    name: slurm_db
    state: present
    login_host: 127.0.0.1
    login_port: 3306
    login_user: root
    login_password: "{{ mariadb.root_password }}"
  retries: 6
  delay: 10

# NOTE: wildcard '%' (any host) as host for 'slurm' is necessary to connect via
# TCP/IP ports
- name: Ensure 'slurm' MySQL user is present with password
  community.mysql.mysql_user:
    name: slurm
    password: "{{ mariadb.slurm_password }}"
```

```
    host: '%'
    login_host: 127.0.0.1
    login_user: root
    login_password: "{{ mariadb.root_password }}"
    state: present
...
```

Si impostano poi i privilegi appropriati, assicurandosi che vengano salvati e applicati ("*flushing*"):

roles/mariadb/tasks/main.yml

```
...
- name: Grant usage privilege to the user
  community.mysql.mysql_user:
    name: slurm
    priv: '*.*:USAGE'
    append_privs: yes
    host: '%'
    login_host: 127.0.0.1
    login_user: root
    login_password: "{{ mariadb.root_password }}"
    state: present

- name: Grant all privileges on the database to the user
  community.mysql.mysql_user:
    name: slurm
    priv: 'slurm_db.*:ALL'
    append_privs: yes
    host: '%'
    login_host: 127.0.0.1
    login_user: root
    login_password: "{{ mariadb.root_password }}"
    state: present

- name: Flush privileges (update & reload grant tables)
  community.mysql.mysql_user:
    login_user: root
    login_password: "{{ mariadb.root_password }}"
    check_implicit_admin: yes
    append_privs: yes
    login_host: 127.0.0.1
    state: present
    name: slurm
    host: '%'
```

```
34 ...
```

A questo punto ho verificato manualmente l'accesso al DB presentandomi come user `slurm`, lo stesso che verrà poi utilizzato da `slurmdbd`:

```
vagrant@controller:~$ mariadb -u slurm --port=3306
↪  --password=THIS_IS_THE_SLURM_DB_PASSWORD slurm_db
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 12
Server version: 11.5.2-MariaDB-ubu2404 mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [slurm_db]>_
```

Ho quindi proseguito installando il demone stesso e importando il file di configurazione specifico:

roles/mariadb/tasks/main.yml

```
1  ...
2  - name: Install 'slurmdb' package
3    ansible.builtin.apt:
4      name: 'slurmdbd'
5      update_cache: true
6
7  - name: Copy slurmdb config file
8    ansible.builtin.template:
9      src: 'slurmdbd.conf'
10     dest: '/etc/slurm/slurmdbd.conf'
11     mode: '0600'
12     owner: 'slurm'
13     group: 'slurm'
```

Il file di configurazione per `slurmdbd` era così composto:

slurmdbd.conf

```
1  AuthType=auth/munge
2  DbdHost=controller
3  DbdPort=6819
```

```
DebugLevel=verbose
StorageHost=localhost
StorageLoc=slurm_db
StoragePass="{{ mariadb.slurm_password }}"
StoragePort=3306
StorageType=accounting_storage/mysql
StorageUser=slurm
LogFile=/var/log/slurm/slurmdbd.log
PidFile=/run/slurmdbd.pid
SlurmUser=slurm
```

Nel tentativo di avviare il demone si sono però presentati i seguenti messaggi di errore:

```
vagrant@controller:~$ sudo slurmdbd -D -vvv
slurmdbd: debug2: accounting_storage/as_mysql: init: mysql_connect() called for
↪  db slurm_db
slurmdbd: debug2: Attempting to connect to localhost:3306
slurmdbd: error: mysql_real_connect failed: 2002 Can't connect to local server
↪  through socket '/run/mysqld/mysqld.sock' (2)
slurmdbd: error: The database must be up when starting the MYSQL plugin.
↪  Trying again in 5 seconds.
```

Indagando sull'errore evidenziato ho appreso che `slurmdbd`, nel momento in cui si specifica `StorageHost=localhost`, tenta la connessione al DB tramite *socket* locale di default (`/run/mysqld/mysqld.sock`). Eseguendo però il DBMS all'interno di un container, la posizione della socket diventa relativa al *filesystem* "virtuale":

```
vagrant@controller:~$ sudo find / -type s | grep 'mysqld.sock'
/home/mariadb/.local/share/containers/storage/overlay/e7d63d4ddde910afaf1d257
↪  4c3345fc74c42372bffc95f84528ead29c2f52d7e/diff/run/mysqld/mysqld.sock
```

Ho così aggiornato il file di configurazione:

slurmdbd.conf

```
...
StorageHost=127.0.0.1
...
```

In questo modo il demone viene forzato a utilizzare porte TCP/IP per stabilire la connessione al DB. Ora `slurmdbd` viene avviato senza errori.

2.4 Cluster singolo

Nel seguente file di configurazione, utilizzato da entrambi `slurmctld` e `slurmd`, viene descritto il generico comportamento del cluster:

slurm.conf

```
ClusterName=cluster
SlurmctldHost=controller
MpiDefault=none
ProctrackType=proctrack/cgroup
ReturnToService=1
SlurmctldPidFile=/var/run/slurmctld.pid
SlurmctldPort=6817
SlurmdPidFile=/var/run/slurmd.pid
SlurmdPort=6818
SlurmdSpoolDir=/var/slurm/slurmd
SlurmUser=slurm
SlurmdUser=root
StateSaveLocation=/var/slurm/slurmctld
SwitchType=switch/none
TaskPlugin=task/affinity
#
# SCHEDULING
SchedulerType=sched/backfill
SelectType=select/linear
#
# LOGGING AND ACCOUNTING
AccountingStorageType=accounting_storage/slurmdbd
JobAcctGatherType=jobacct_gather/none
SlurmctldLogFile=/var/log/slurm/slurmctld.log
SlurmdLogFile=/var/log/slurm/slurmd.log
#
# COMPUTE NODES
NodeName=slurm[1-2] CPUs=4 State=UNKNOWN
PartitionName=debug Nodes=ALL Default=YES MaxTime=INFINITE State=UP
```

Avendo indicato `AccountingStorageType=accounting_storage/slurmdbd`, `slurmctld` attiva la registrazione delle attività comunicando con `slurmdbd`, tramite la porta di default `6819`.

Per verificare l'intero funzionamento del cluster è necessario anzitutto avviare i nodi worker (`slurm1` e `slurm2`), che si collegano in automatico al controllore, risolvendo il nome di host indicato da `SlurmctldHost=controller`. In particolare, la comunicazione avviene sulla porta di default `6817`. Una volta attivi, si può verificare lo stato dei nodi eseguendo:

```
vagrant@controller:~$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
debug*       up   infinite      2   idle slurm[1-2]
```

Un esempio di sottomissione di job è il seguente (l'opzione `-N` indica il numero di nodi worker su cui svolgere il comando):

```
vagrant@controller:~$ srun -N2 hostname
slurm1
slurm2
```

È quindi evidente che il programma `hostname` sia stato eseguito da entrambi i nodi di computazione, confermando il corretto funzionamento del cluster.

2.5 Topologia di rete

Una federazione SLURM, per definizione, è composta da (almeno) due cluster in comunicazione tra loro; in contesti reali, è verosimile che ciascuno di essi venga confinato all'interno della propria LAN, dove la trasmissione verso l'esterno è mediata da un *router*. Il seguente schema visualizza il concetto, applicandolo al caso di tirocinio:

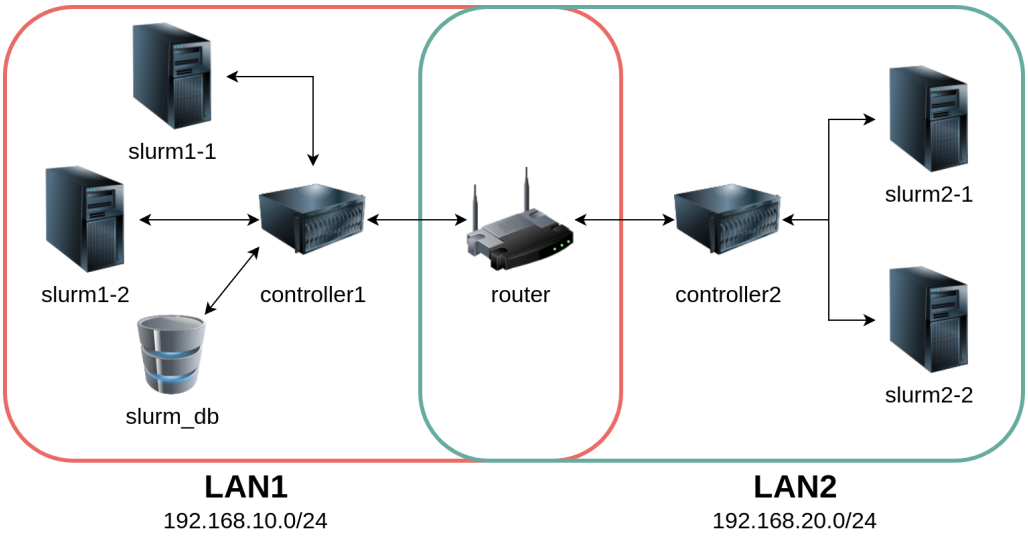

Figura 2.1: Topologia virtuale desiderata

Ho dunque esteso l'infrastruttura con una seconda rete di host virtuali e un'ulteriore VM per svolgere la funzione di router. Il file di configurazione di Vagrant risulta così composto:

Vagrantfile

```
Vagrant.configure("2") do |config|
  #
  # VM BOX CONFIGURATION
  config.vm.box = "debian/bookworm64"
  config.vm.provider "virtualbox" do |vb|
    vb.linked_clone = true
    vb.memory = "2048"
    vb.cpus = "4"
  end
  #
  # VIRTUAL MACHINES
  config.vm.define "router" do |machine|
    machine.vm.hostname = "router"
    machine.vm.network "private_network", virtualbox__intnet: "LAN1",
    ↪ auto_config: false
    machine.vm.network "private_network", virtualbox__intnet: "LAN2",
    ↪ auto_config: false
  end
  (1..2).each do |i|
    config.vm.define "controller#{i}" do |machine|
      machine.vm.hostname = "controller#{i}"
      machine.vm.network "private_network", virtualbox__intnet: "LAN#{i}",
      ↪ auto_config: false
    end
    (1..2).each do |j|
      config.vm.define "slurm#{i}-#{j}" do |machine|
        machine.vm.hostname = "slurm#{i}-#{j}"
        machine.vm.network "private_network", virtualbox__intnet: "LAN#{i}",
        ↪ auto_config: false
      end
    end
  end
  #
  # PROVISIONING
  config.vm.provision "ansible" do |ansible|
    ansible.playbook = "site.yml"
  end
end
```

Da notare l'uso di iterazioni per la definizione compatta di molteplici VM.

Ho assegnato a `router` gli indirizzi IP statici `192.168.10.254` e `192.168.20.254`, su due interfacce separate; essendo appartenente a entrambe le LAN, ho poi trasfe-

rito su di esso l'esecuzione di `dnsmasq`, modificandone la configurazione per servire richieste DHCP e DNS originanti da entrambe le reti:

dnsmasq.conf

```
interface=eth1
interface=eth2
no-hosts
no-resolv

# NAMESERVER USED FOR THE INTERNET (host running the VMs)
server=10.0.2.3

dhcp-range=interface:eth1,192.168.10.1,192.168.10.253,12h
dhcp-range=interface:eth2,192.168.20.1,192.168.20.253,12h

dhcp-option-force=interface:eth1,option:dns-server,192.168.10.254
dhcp-option-force=interface:eth2,option:dns-server,192.168.20.254

# AVOID ROUTING ALL TRAFFIC THROUGH THIS VM
dhcp-option=3

# STATIC ROUTE FOR CROSS-LAN COMMUNICATION
dhcp-option=121,192.168.20.0/24,192.168.10.254,192.168.10.0/24,192.168.20.254
```

L'opzione evidenziata specifica il *gateway* tra le reti, ossia il "percorso" da scegliere in caso di pacchetti provenienti da una LAN e destinati all'altra. Affinché `router` possa effettivamente instradare i pacchetti, è necessario attivare il *forwarding* a livello di kernel:

roles/router/tasks/main.yml

```
...
- name: Allow IPv4 forwarding
  ansible.builtin.lineinfile:
    path: '/etc/sysctl.conf'
    line: 'net.ipv4.ip_forward=1'
    state: present
  notify: Reload kernel config
...
```

Ora l'infrastruttura è pronta per l'integrazione del secondo cluster SLURM.

2.6 Federazione SLURM

Ciascun cluster SLURM è descritto interamente dal file `slurm.conf` corrispondente, che deve risultare identico su tutti i nodi. Dovendo separare l'ambiente, ho generato

due file di configurazione indipendenti; di seguito sono indicate le loro differenze rispetto alla struttura a cluster singolo:

roles/cluster1/templates/slurm.conf

```
# CLUSTER 1 CONFIG
ClusterName=cluster1
SlurmctldHost=controller1
FederationParameters=fed_display
...
NodeName=slurm1-[1-2] ...
...
```

roles/cluster2/templates/slurm.conf

```
# CLUSTER 2 CONFIG
ClusterName=cluster2
SlurmctldHost=controller2
FederationParameters=fed_display
...
AccountingStorageHost=controller1
...
NodeName=slurm2-[1-2] ...
...
```

L'opzione evidenziata, presente solo nel cluster senza DB, indica l'host dove viene eseguito `slurmdbd`, da contattare per l'*accounting* delle attività. La registrazione di entrambi i cluster sullo stesso DB è una prerogativa essenziale per l'impostazione della federazione:

```
vagrant@controller1:~$ sudo sacctmgr add federation testfederation
↪  clusters=cluster1,cluster2
 Adding Federation(s)
  testfederation
 Settings
  Cluster       = cluster1
  Cluster       = cluster2
Would you like to commit changes? (You have 30 seconds to decide)
 (N/y): y
```

Utilizzando `sacctmgr` e `scontrol`, ho confermato la corretta comunicazione tra i due cluster:

```
vagrant@controller1:~$ sacctmgr show federation
Federation    Cluster ID             Features     FedState
---------- ---------- -- -------------------- ------------
testfeder+   cluster1  1                            ACTIVE
testfeder+   cluster2  2                            ACTIVE
vagrant@controller1:~$ scontrol show federation
Federation: testfederation
Self:       cluster1:127.0.0.1:6817 ID:1 FedState:ACTIVE Features:
Sibling:    cluster2:192.168.20.194:6817 ID:2 FedState:ACTIVE Features:
↪  PersistConnSend/Recv:Yes/Yes Synced:Yes
```

Il comando `sinfo` presenta ora anche lo specifico cluster di provenienza dei nodi:

```
vagrant@controller1:~$ sinfo
PARTITION CLUSTER  AVAIL  TIMELIMIT  NODES  STATE NODELIST
debug*    cluster1    up   infinite      1   idle slurm1-[1-2]
debug*    cluster2    up   infinite      1   idle slurm2-[1-2]
```

Grazie all'opzione `--clusters` in fase di sottomissione, è possibile specificare una lista di cluster sul quale tentare di eseguire il job richiesto:

```
vagrant@controller1:~$ srun --clusters=cluster2 -N2 hostname
slurm2-1
slurm2-2
vagrant@controller2:~$ srun --clusters=cluster1 -N2 hostname
slurm1-1
slurm1-2
```

Da notare come, in entrambe le sottomissioni di esempio, il controllore di partenza non appartenga al cluster specificato per l'esecuzione, verificando quindi il corretto comportamento della federazione.

2.7 Condivisione di risorse

La seconda funzionalità richiesta dal tirocinio riguarda la condivisione delle risorse di computazione dei nodi, in particolare di Central Processing Unit (CPU), Random Access Memory (RAM) e Graphics Processing Unit (GPU). La configurazione del comportamento è descritta in `slurm.conf` e può raggiungere alti livelli di personalizzazione, in base alle esigenze specifiche.

Allo scopo di semplificare l'implementazione, ho lavorato su un singolo cluster SLURM, composto da nodo controllore e nodo di computazione. La configurazione presenta le aggiunte:

slurm.conf

```
...
SelectType=select/cons_tres
SelectTypeParameters=CR_CPU_Memory
...
AccountingStorageTRES=gres/gpu
...
GresTypes=gpu
NodeName=worker CPUs=4 RealMemory=1024 Gres=gpu:rtx2080ti:2 State=UNKNOWN
...
```

In particolare, le seguenti opzioni specificano:

- `SelectType=select/cons_tres`: plugin apposito per la gestione delle risorse ("*consumable trackable resources*") e quindi l'esecuzione di più job sullo stesso nodo;
- `SelectTypeParameters=CR_CPU_Memory`: risorse da condividere (la GPU viene contata separatamente);
- `AccountingStorageTRES=gres/gpu`: registrazione dell'attività delle GPU;
- `GresTypes=gpu`: tipi di Generic Resource (GRES) da condividere;
- `RealMemory=1024`: memoria totale (in Megabyte (MB)) allocabile per nodo worker (opzione obbligatoria per la condivisione della memoria; deve essere minore o uguale alla RAM riconosciuta dal sistema operativo);
- `Gres=gpu:rtx2080ti:2`: Specifiche GRES presenti nel nodo (in questo caso, due GPU).

È stato poi necessario introdurre nel nodo `worker` due ulteriori file di configurazione, per l'interfacciamento con le risorse presenti:

cgroup.conf

```
CgroupPlugin=autodetect
CgroupAutomount=yes
ConstrainCores=no
ConstrainRAMSpace=no
ConstrainDevices=yes
```

gres.conf

```
NodeName=worker Name=gpu Type=rtx2080ti Count=2 File=/dev/nvidia[0-1]
```

A questo punto, tramite il comando `sinfo`, ho controllato la presenza delle giuste risorse nel nodo di computazione:

```
vagrant@controller:~$ sinfo -o "%15N  %10c  %10m  %20G"
NODELIST         CPUS        MEMORY      GRES
worker           4           1024        gpu:rtx2080ti:2
```

Per verificare il funzionamento, ho allocato quattro job da 1 CPU (`-c1`) e 256 MB di RAM (`--mem=256`):

```
vagrant@controller:~$ srun -c1 --mem=256 sleep 60 &
[1] 18254
vagrant@controller:~$ srun -c1 --mem=256 sleep 60 &
[2] 18264
vagrant@controller:~$ srun -c1 --mem=256 sleep 60 &
[3] 18274
vagrant@controller:~$ srun -c1 --mem=256 sleep 60 &
[4] 18284
```

Con il comando `squeue` ho poi accertato l'esecuzione contemporanea dei job (tutti in stato *running*) sul nodo `worker` da 4 CPU e 1 GB di RAM, a dimostrazione della condivisione dei processori e della memoria:

```
vagrant@controller:~$ squeue
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
    1     debug    sleep  vagrant  R       0:15      1 worker
    2     debug    sleep  vagrant  R       0:14      1 worker
    3     debug    sleep  vagrant  R       0:13      1 worker
    4     debug    sleep  vagrant  R       0:12      1 worker
```

Allo stesso modo, ho allocato due job richiedenti 1 GPU ciascuno (`--gpus=1`) e verificato la loro esecuzione parallela:

```
vagrant@controller:~$ srun --gpus=1 --mem=256 sleep 60 &
[1] 18294
vagrant@controller:~$ srun --gpus=1 --mem=256 sleep 60 &
[2] 18304
vagrant@controller:~$ squeue
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
    5     debug    sleep  vagrant  R       0:08      1 worker
    6     debug    sleep  vagrant  R       0:07      1 worker
```

2.8 Priorità di scheduling

La terza funzionalità richiesta dal tirocinio è stata l'impostazione della priorità di esecuzione per uno specifico utente, su una singola GPU del nodo di computazione. L'idea di base consiste nel generare un'ulteriore partizione SLURM, prioritaria (tramite Quality of Service (QOS)), attiva su una GPU e accessibile solo dall'utente desiderato.

Sono partito definendo degli utenti/gruppi di test, coerenti su ciascun nodo del cluster; nel mio caso ho scelto `user1` come utente prioritario:

roles/slurmcommon/tasks/main.yml

```
...
- name: Ensure test groups exist
  ansible.builtin.group:
    name: "{{ item.name }}"
    gid: "{{ item.id }}"
    state: present
  loop:
    - { name: 'user1', id: '2001' }
    - { name: 'user2', id: '2002' }
    - { name: 'user3', id: '2003' }
    - { name: 'user4', id: '2004' }

- name: Ensure test users exist
  ansible.builtin.user:
    name: "{{ item.name }}"
    uid: "{{ item.id }}"
    group: "{{ item.name }}"
    state: present
  loop:
    - { name: 'user1', id: '2001' }
    - { name: 'user2', id: '2002' }
    - { name: 'user3', id: '2003' }
    - { name: 'user4', id: '2004' }
...
```

Il file di configurazione SLURM è stato modificato in questo modo:

slurm.conf

```
...
PriorityType=priority/multifactor
PriorityWeightQOS=10000
PriorityWeightTRES=GRES/gpu=100
...
AccountingStorageEnforce=limits
```

```
7 JobAcctGatherType=jobacct_gather/cgroup
8 ...
9 PartitionName=gpupart Nodes=worker QOS=gpuprio AllowAccounts=user1acct State=UP
```

In particolare, le seguenti opzioni specificano:

- `PriorityType=priority/multifactor` : plugin che coinvolge più fattori per determinare la priorità di un job; per il caso in questione ho aumentato il peso di QOS e GPU, tramite le opzioni `PriorityWeight-`;
- `AccountingStorageEnforce=limits` : imposizione dei limiti (sulle risorse) descritti nel DB; impone inoltre, per tutte le richieste di esecuzione di job, la presenza di un'associazione (utente, account, partizione, ...) valida;
- `PartitionName=gpupart` : partizione aggiuntiva ristretta all'account `user1acct`, con QOS `gpuprio`.

Ho dunque registrato nel DB (tramite il comando `sacctmgr`) le informazioni per il comportamento desiderato:

```
vagrant@controller:~$ sudo sacctmgr add qos gpuprio Priority=10
↪  GrpTRES=gres/gpu=1
 Adding QOS(s)
  gpuprio
 Settings
  Description    = gpuprio
  GrpTRES                 = gres/gpu=1
  Priority                = 10
Would you like to commit changes? (You have 30 seconds to decide)
 (N/y): y
vagrant@controller:~$ sudo sacctmgr add account user1acct QOS=gpuprio
 Adding Account(s)
  user1acct
 Settings
  Description     = Account Name
  Organization    = Parent/Account Name
 Associations =
  C = testcluster A = user1acct
 Settings
  QOS           = gpuprio
Would you like to commit changes? (You have 30 seconds to decide)
 (N/y): y
vagrant@controller:~$ sudo sacctmgr add account noprio
 Adding Account(s)
  noprio
```

```
 Settings
  Description     = Account Name
  Organization    = Parent/Account Name
 Associations =
  C = testcluster A = noprio
Would you like to commit changes? (You have 30 seconds to decide)
 (N/y): y
```

L'account `noprio` è essenziale affinché gli utenti non prioritari possano richiedere l'esecuzione di job. Da notare il confinamento della priorità di `gpuprio` a una singola GPU, tramite l'opzione `GrpTRES`.

Gli utenti, durante la registrazione, vengono collegati a un account di default:

```
vagrant@controller:~$ sudo sacctmgr add user user1 DefaultAccount=user1acct
 Adding User(s)
  user1
 Settings
  Default Account = user1acct
 Associations =
  C = testcluster A = user1acct            U = user1
Would you like to commit changes? (You have 30 seconds to decide)
 (N/y): y
vagrant@controller:~$ sudo sacctmgr add user user2,user3,user4
↪  DefaultAccount=noprio
 Adding User(s)
  user2
  user3
  user4
 Settings
  Default Account = noprio
 Associations =
  C = testcluster A = noprio               U = user2
  C = testcluster A = noprio               U = user3
  C = testcluster A = noprio               U = user4
Would you like to commit changes? (You have 30 seconds to decide)
 (N/y): y
```

Il comando `sacctmgr` permette inoltre di verificare le informazioni registrate:

```
vagrant@controller:~$ sacctmgr show qos format=name,priority,GrpTRES
      Name   Priority       GrpTRES
---------- ---------- -------------
    normal          0
```

```
   gpuprio         10    gres/gpu=1
vagrant@controller:~$ sacctmgr show association user=user1,user2,user3,user4
↪  format=user,account,qos
      User    Account                  QOS
---------- ---------- --------------------
     user2     noprio               normal
     user3     noprio               normal
     user4     noprio               normal
     user1  user1acct              gpuprio
```

Per testare la priorità di scheduling su una GPU, ho allocato 3 job non prioritari e 1 job lanciato da `user1` sulla partizione `gpupart`:

```
vagrant@controller:~$ sudo -u user2 srun --gpus=1 --mem=256 sleep 60 &
[1] 18314
vagrant@controller:~$ sudo -u user3 srun --gpus=1 --mem=256 sleep 60 &
[2] 18324
vagrant@controller:~$ sudo -u user4 srun --gpus=1 --mem=256 sleep 60 &
[3] 18334
vagrant@controller:~$ sudo -u user1 srun --partition=gpupart --gpus=1 --mem=256
↪  sleep 60 &
[4] 18344
vagrant@controller:~$ squeue
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
    9     debug    sleep    user4 PD       0:00      1 (Resources)
    8     debug    sleep    user3  R       0:03      1 worker
    7     debug    sleep    user2  R       0:17      1 worker
   10   gpupart    sleep    user1 PD       0:00      1 (QOSGrpGRES)
```

Come evidenziato, il job `10` è in concorrenza di risorse con il job `9`, ma attendendo la liberazione di una GPU (ossia il termine del job `7`), la partizione prioritaria ha la precedenza di esecuzione:

```
vagrant@controller:~$ squeue
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
    9     debug    sleep    user4 PD       0:00      1 (Resources)
    8     debug    sleep    user3  R       0:50      1 worker
   10   gpupart    sleep    user1  R       0:04      1 worker
```

Capitolo 3

Conclusioni

L'esperienza di tirocinio presso Unibo Laboratory of Information and System Security (ULISSE) è iniziata con lo studio e l'approfondimento autonomo degli strumenti chiave interessati (in particolare SLURM), al termine del quale ho chiarito gli obiettivi posti dall'attività.

Successivamente ho dovuto integrare l'ambiente personale di lavoro con le tecnologie essenziali allo sviluppo del progetto. È stato dunque necessario:

- aggiornare Vagrant e Ansible, verificandone il funzionamento;
- configurare Git per l'accesso alla piattaforma remota GitLab del DISI;
- istanziare una copia locale del *repository*;
- generare un nuovo *branch* di lavoro.

A questo punto ho iniziato il lavoro sul progetto in sé:

- attivando la risoluzione DNS;
- impostando i privilegi adeguati nel DB e collegando a esso `slurmdbd`;
- abilitando la comunicazione tra i vari demoni SLURM coinvolti.

In questo modo ho ottenuto e verificato il corretto funzionamento del singolo *cluster* SLURM.

Sono poi passato all'estensione dell'infrastruttura virtuale, aggiungendo una seconda LAN contenente il secondo cluster. In particolar modo, ho dovuto:

- introdurre un *router* per il collegamento tra le due reti;
- estendere la configurazione DHCP/DNS a entrambe le LAN;
- generare una nuova configurazione SLURM per il secondo cluster, avendo cura di registrare le attività sul medesimo DB.

La prima funzionalità richiesta dall'attività di tirocinio è stata la federazione di cluster SLURM; per attivarne il comportamento ho dovuto semplicemente registrare la configurazione nel DB in comune.

La seconda funzionalità da integrare è stata la condivisione delle risorse di computazione, che permette il loro assegnamento dinamico in base alle necessità dei job da eseguire. Le risorse da poter condividere sono molteplici, nel mio caso ho dovuto implementare la condivisione di:

- Central Processing Unit (CPU);
- Random Access Memory (RAM);
- Graphics Processing Unit (GPU).

Questa attività ha richiesto molta documentazione autonoma, in particolare per la corretta configurazione delle Generic Resource (GRES), di cui le GPU fanno parte.

Il tirocinio si è concluso implementando l'ultima funzionalità SLURM, ossia la configurazione della priorità di *scheduling* per un utente specifico, su una risorsa specifica (in particolare, una GPU). Il vincolo della singola risorsa ha guidato l'attività di sviluppo verso l'aggiunta di una partizione SLURM dedicata, con priorità maggiore su una singola GPU, e accessibile esclusivamente dall'utente interessato.

Acronimi

CPU Central Processing Unit

DB Database

DBMS Database Management System

DHCP Dynamic Host Configuration Protocol

DISI Dipartimento di Informatica – Scienza e Ingegneria

DNS Domain Name System

GPU Graphics Processing Unit

GRES Generic Resource

HPC High Performance Computing

IaC Infrastructure as Code

IP Internet Protocol

LAN Local Area Network

MB Megabyte

MUNGE MUNGE Uid 'N' Gid Emporium

QOS Quality of Service

RAM Random Access Memory

SLURM Simple Linux Utility for Resource Management

SSH Secure Shell

TCP Transmission Control Protocol

ULISSE Unibo Laboratory of Information and System Security

VCS Version Control System

VM Virtual Machine

YAML Yet Another Markup Language