Alma Mater Studiorum · Università di Bologna

Scuola di Ingegneria e Architettura
Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Corso di Laurea in Ingegneria Informatica

Tesi di Laurea in:
Amministrazione di Sistemi T

SVILUPPO DI UN'INFRASTRUTTURA VIRTUALE PER L'EROGAZIONE DI SERVIZI DI CALCOLO CON SLURM

Relatore:
Prof. Marco Prandini

Presentata da:
Massimo Valerio Zerbini

Correlatore:
Ing. Andrea Giovine

Appello V
Anno Accademico 2023/2024

"It is only when they go wrong, that machines remind you how powerful they are."

— Clive James

Abstract

Nel presente elaborato di tesi viene descritto e analizzato il processo di sviluppo di un'infrastruttura virtuale SLURM, per l'integrazione di specifici servizi di calcolo in ambito *High Performance Computing* (HPC). In particolare, si implementano la condivisione delle risorse di computazione, la definizione di una priorità di *scheduling* e la federazione di due *cluster* virtuali. La progettazione dell'ambiente avviene applicando il principio di *Infrastructure as Code* (IaC), tramite strumenti come Vagrant e Ansible. Durante il processo di sviluppo, si affrontano e analizzano le varie difficoltà legate alla corretta configurazione dell'intera infrastruttura distribuita. Le funzionalità integrate vengono infine convalidate tramite l'esecuzione di test mirati.

Indice

1 | Introduzione

Il contesto dell'High Performance Computing (HPC) rappresenta l'avanguardia per la risoluzione di problemi complessi e computazionalmente avanzati. Esso consiste nel coordinamento di più unità di elaborazione (raggruppate in "*cluster*") per fornire esecuzioni ad alte prestazioni, ricorrendo tipicamente al calcolo simultaneo [1]. La nascita del primo standard di programmazione parallela risale al 1990, ma in tempi recenti la potenza di calcolo ha raggiunto ordini di grandezza significativi per la simulazione di scenari reali.

L'applicazione di ambienti HPC viene principalmente associata all'ambito della ricerca scientifica, ma il suo utilizzo è esteso a qualunque disciplina che necessiti di calcolo a elevate prestazioni; alcuni esempi sono:

- la riproduzione e lo studio del clima globale (Climatologia);
- la risoluzione di equazioni fluidodinamiche (Fisica);
- la ricerca di metodi di conservazione per antichi testi e scritture (Archeologia);
- lo studio delle proteine per la cura di malattie degenerative (Medicina).

La progettazione e la manutenzione di un ambiente HPC coinvolgono tecnologie e aspetti tipici dell'ingegneria informatica, tra cui l'amministrazione di sistemi. È infatti di cruciale importanza la corretta impostazione della comunicazione tra i diversi nodi interessati.

In questo elaborato di tesi viene descritto e analizzato il procedimento di sviluppo di un'infrastruttura virtuale distribuita, dedicata all'implementazione di diverse funzionalità inerenti al contesto HPC. In particolare, verranno integrate:

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

In primo luogo, si introdurranno le tecnologie e i particolari strumenti scelti per l'implementazione, discutendone il funzionamento e l'architettura.

In seguito, verrà progettato e definito l'ambiente virtuale IaC, affrontando problematiche come la configurazione di rete, la risoluzione DNS e la registrazione delle attività in un Database.

In ultimo, verranno condotti test mirati per la verifica di ciascuna delle funzionalità implementate.

2 | Stato dell'arte

2.1 Virtualizzazione

Una Virtual Machine (VM) (in italiano, *macchina virtuale*) rappresenta un'emulazione virtuale di un calcolatore, ottenuta tramite astrazione delle risorse hardware della macchina "*host*" (processo di virtualizzazione) [2]. Esistono diversi tipi di virtualizzazione:

- **Virtualizzazione nativa (o totale):** l'hardware viene completamente virtualizzato grazie all'uso di *hypervisor* (figura 2.1);
- **Virtualizzazione non nativa:** l'emulazione a livello software è basata su hardware diverso da quello fisicamente utilizzato, e necessita di codice aggiuntivo di basso livello per la compatibilità;
- **Paravirtualizzazione:** non viene simulato hardware in modo diretto, ma viene offerta una speciale Application Programming Interface (API) per l'interazione tra macchine virtuali e risorse fisiche;
- **Containerizzazione:** implementa la virtualizzazione a livello di *kernel* del sistema operativo, permettendo l'esecuzione contemporanea e isolata di istanze *user-space*, senza *overhead* dovuto all'emulazione hardware.

Ciascun tipo di virtualizzazione offre un diverso equilibrio tra flessibilità e prestazioni; in particolare, la virtualizzazione totale si è dimostrata specialmente adatta per:

- la condivisione di un sistema tra diversi utenti;
- l'isolamento dei programmi (e degli utenti) presenti sulle VM;
- l'emulazione di nuovo hardware per il raggiungimento di affidabilità, produttività e sicurezza.

2.1.1 Vagrant

Vagrant è un software *open source* per la gestione di VM, focalizzato sulla riproducibilità affidabile degli ambienti virtualizzati [3]. Nasce nel 2010 come progetto personale di Mitchell Hashimoto, ma cresce in popolarità dopo la fondazione di HashiCorp nel 2012.

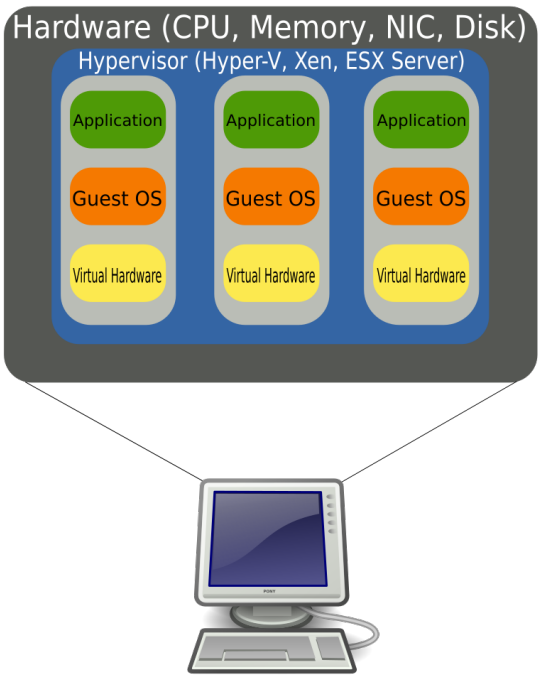

Figura 2.1: Virtualizzazione hardware totale [2]

Architettura

Vagrant dipende da software di virtualizzazione esterni (detti "*provider*" o "*hypervisor*") come VirtualBox[1] e VMware[2], ma grazie al suo alto livello di astrazione, risulta portabile rispetto a essi.

Immagini di VM di base (denominate "*box*") sono facilmente reperibili e condivisibili su depositi online: ciò rappresenta un punto di forza per Vagrant, che solleva gli sviluppatori dal laborioso processo di inizializzazione degli ambienti virtuali. L'uso della virtualizzazione hardware permette l'esecuzione di ambienti e sistemi operativi indipendenti dalla macchina *host*.

A tempo di esecuzione, Vagrant utilizza un file di configurazione (`Vagrantfile`) per ottenere i parametri definiti, quali immagine di base, specifiche hardware, cartelle condivise e indirizzi di rete locali.

2.2 Provisioning

Il *provisioning* rappresenta l'insieme delle azioni preparatorie necessarie a rendere un sistema o servizio pienamente operativo [4]. Nell'ambito dell'amministrazione di sistemi, esempi tipici sono installazione/aggiornamento di software, configurazione di rete e gestione degli utenti.

[1] `https://www.virtualbox.org/`
[2] `https://www.vmware.com/`

2.2.1 Ansible

Ansible è un pacchetto di strumenti software *open source* per la configurazione e gestione automatizzata di ambienti informatici, tramite file di definizione (principio noto come Infrastructure as Code (IaC)) [5] [6]. Ansible nasce nel 2012 e viene acquisito dalla nota compagnia software RedHat nel 2015; è stato inizialmente sviluppato per sistemi Unix-like, ma ha esteso il proprio supporto anche a nodi Windows. Ansible è compatibile con computer fisici, macchine virtuali e interi ambienti *cloud*.

Architettura

Ansible si basa su interazione "*agentless*", dove il controllore, mediante una semplice connessione Secure Shell (SSH), esegue le operazioni definite sui nodi interessati, senza quindi l'esigenza di particolare software sul sistema destinatario (escludendo il server SSH).

I nodi da gestire sono identificati tramite "*inventory*", un file di testo che li elenca e li raggruppa. Per definire le operazioni da eseguire, Ansible utilizza i "*playbook*", file descritti in linguaggio Yet Another Markup Language (YAML) di semplice interpretazione.

L'elenco dei moduli responsabili delle azioni sui nodi è costantemente aggiornato e consultabile nella documentazione ufficiale [7]. Essi sono sviluppati aspirando a un'alta astrazione dal sistema destinatario e all'idempotenza delle singole operazioni.

2.3 DNS & DHCP

Il protocollo Domain Name System (DNS) nasce in seguito all'esigenza di collegare indirizzi Internet Protocol (IP) a nomi di host, facilmente interpretabili e memorizzabili dagli esseri umani [8]. Per ciascuna zona di autorità è presente almeno un *name server*, dove vengono mantenute le corrispondenze tra *hostname* e indirizzi di competenza del dominio.

La risoluzione DNS è un procedimento trasparente alle applicazioni e agli utenti finali. La richiesta di traduzione del nome di host viene trasmessa al *resolver* locale, il quale ne controlla la presenza in memoria *cache*; in caso di non disponibilità del record, la richiesta viene inoltrata in autonomia ai *resolver* di livello superiore (figura 2.2).

Il protocollo Dynamic Host Configuration Protocol (DHCP) svolge invece la funzione di assegnamento automatico e dinamico di indirizzi IP, nell'ambito di una Local Area Network (LAN) [9].

DHCP lavora in architettura *client/server*: al momento della connessione alla rete locale, il cliente invia una richiesta in *broadcast*, e il server DHCP di competenza risponde assegnando un indirizzo IP locale all'interno di un intervallo configurabile.

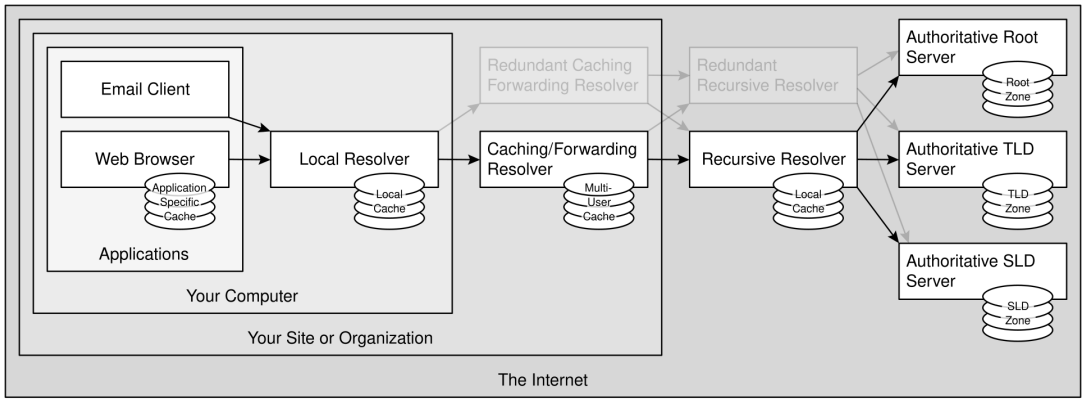

Figura 2.2: Sequenza di risoluzione DNS [8]

2.3.1 dnsmasq

`dnsmasq` è un software *open source* leggero e facilmente configurabile, in grado di fornire servizi DNS/DHCP su piccola scala [10].

In caso di richieste di risoluzione DNS, `dnsmasq` accede alla *cache* locale ed eventualmente inoltra l'operazione a un effettivo *resolver* ricorsivo. Per richieste DHCP, `dnsmasq` si affida invece alla configurazione locale per allocazioni statiche o dinamiche. Il servizio DHCP si integra in modo naturale con il servizio DNS, consentendo la registrazione automatica delle corrispondenze *hostname*/*address* in seguito al *leasing* degli indirizzi.

2.4 Base di dati

Un Database (DB) (in italiano, *base di dati*) è una collezione organizzata di dati accessibili tramite un Database Management System (DBMS), un software di interfacciamento per amministratori, applicazioni e utenti finali [11].

L'interazione con il DB è regolata da linguaggi specializzati:

- Data Control Language (DCL) per il controllo dell'accesso ai dati;
- Data Definition Language (DDL) per la definizione di dati e delle relazioni tra di essi;
- Data Manipulation Language (DML) per la manipolazione dei dati (inserimento, modifica, rimozione);
- Data Query Language (DQL) per la ricerca di informazioni estrapolate dai dati.

Structured Query Language (SQL) raggruppa le funzionalità di DDL, DML e DQL in un singolo linguaggio, rappresentando uno standard *de facto* nell'ambito delle basi di dati.

Le tipologie dominanti di DB sono:

- **Relazionale:** i dati sono strutturati in *tabelle* collegate univocamente tra loro tramite *relazioni* e *chiavi* identificative (figura 2.3); ciascuna tabella simboleggia un'entità: le colonne indicano gli *attributi*, mentre le righe rappresentano le sue *istanze*; grazie alle *transazioni*, i dati godono delle proprietà di Atomicity, Consistency, Isolation, Durability (ACID);
- **Not only SQL (NoSQL):** estensione dei DB relazionali, con supporto verso ulteriori strutture (chiave:valore, documenti, colonne, grafi, ...); è indicato per applicazioni a bassa latenza ed elevato *throughput*.

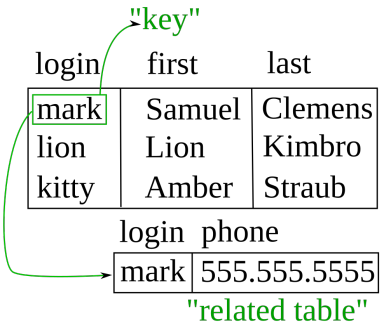

Figura 2.3: Modello relazionale [11]

2.4.1 MariaDB

MariaDB è un DBMS relazionale *open source*, basato sul diffuso MySQL, con il quale aspira a mantenere un'alta compatibilità nonostante l'introduzione di funzionalità avanzate, come nuovi *storage engine* [12].

2.5 Workload Management

Il concetto di *job scheduling* nasce con l'avvento dei primi sistemi informatici, in grado di eseguire una coda di schede perforate descriventi sequenze di istruzioni macchina [13]. I software moderni per la gestione di job sono tipicamente strutturati in architettura distribuita *master/agent*, con un singolo nodo di controllo e numerosi nodi di esecuzione.

2.5.1 SLURM

Simple Linux Utility for Resource Management (SLURM) è un software *open source* per la gestione e il coordinamento di nodi di calcolo, finalizzato alla pianificazione e l'esecuzione di programmi ("*jobs*") [14]. SLURM è altamente scalabile, rendendolo adatto anche in contesti High Performance Computing (HPC) a intenso utilizzo di risorse.

Vengono integrate tre funzioni principali:

- allocazione esclusiva (e non) delle risorse computazionali, 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 code 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

Come illustrato in figura 2.4, un *cluster* SLURM è coordinato dal demone controllore `slurmctld`, il cui compito è gestire le richieste di esecuzione e allocare 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 delle attività di molteplici cluster in un singolo DB; con esso vengono configurati gli *account* per il controllo degli utenti abilitati alla sottomissione di programmi. È inoltre possibile raggruppare i nodi di computazione in partizioni logiche assestanti e configurabili individualmente.

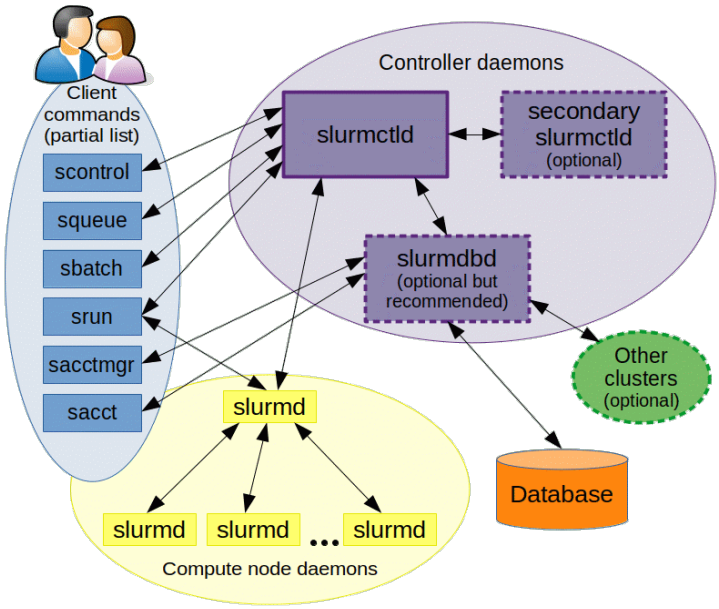

Figura 2.4: Componenti di un cluster SLURM [14]

Autenticazione

Le interazioni tra i vari demoni SLURM avvengono tramite Remote Procedure Call (RPC): è dunque necessario verificare l'autenticità della fonte del messaggio. Di default si utilizza MUNGE Uid 'N' Gid Emporium (MUNGE), un servizio per la creazione e validazione di credenziali utente, progettato appositamente per ambienti HPC [15].

La codifica/decodifica delle credenziali si basa su una chiave comune, condivisa ai soli nodi fidati. Un prerequisito fondamentale per il corretto funzionamento di MUNGE è la coerenza di utenti e gruppi in tutti i nodi interessati [16].

Condivisione di risorse

Di default, SLURM alloca in modo esclusivo interi nodi di computazione per l'esecuzione dei programmi: ciò significa che, anche in caso di basso utilizzo di risorse, nessun altro job verrà inoltrato allo stesso nodo. Per superare l'inefficienza di questo scenario, è possibile attivare la condivisione di risorse, quali Central Processing Unit (CPU) e Random Access Memory (RAM) [17]; configurazioni più granulari sono applicabili per le componenti della CPU, come *socket* e *core* (figura 2.5).

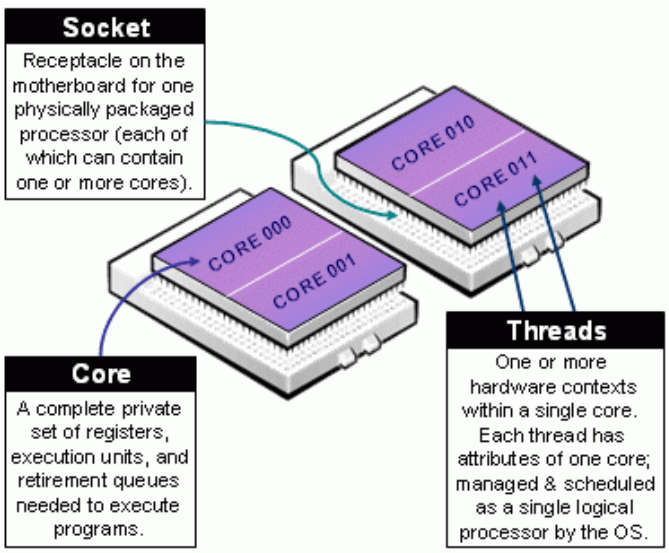

Figura 2.5: Definizioni di *socket*, *core* e *thread* [18]

Ulteriori tipologie di risorse, come le Graphics Processing Unit (GPU), vengono supportate sotto la classificazione di Generic Resource (GRES): per esse è necessaria la precisa definizione tramite file di configurazione. È inoltre possibile la loro condivisione tra i diversi job in coda [19].

Priorità di scheduling

SLURM offre numerosi metodi per impostare la priorità di *scheduling* tra i job; i principali sono:

- configurazione del **peso di molteplici fattori** ("*Multifactor Priority*") per la selezione del job da eseguire, come ad esempio la sua dimensione, la quantità di risorse richieste e il tempo trascorso in coda [20];
- definizione di **Quality of Service (QoS)** per partizioni o singoli nodi [21], con la possibilità di limitare l'uso di risorse specifiche [22].

Federazione

SLURM permette la definizione di federazioni di *cluster*, dove la sottomissione dei job è inoltrata a tutti i componenti [23]. Ciascuno di essi 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 2.6).

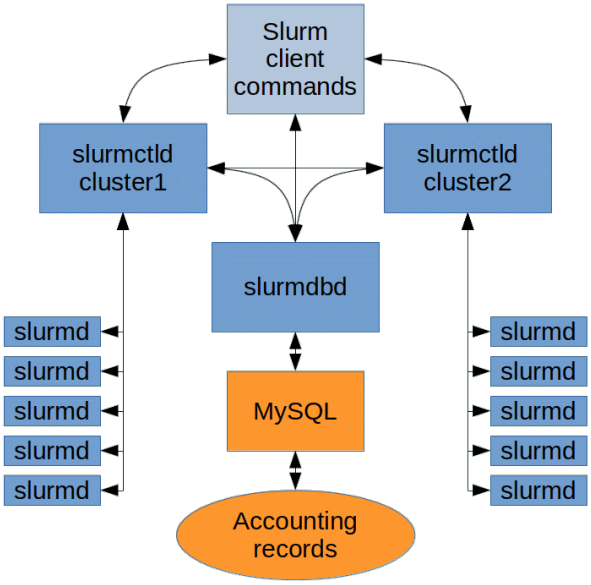

Figura 2.6: Comunicazione in una federazione SLURM [24]

3 | Analisi progettuale

Il progetto si pone di sviluppare, in stile Infrastructure as Code (IaC), un'infrastruttura di VM dedicata all'integrazione di particolari funzionalità SLURM. Di seguito sono analizzati i passi necessari.

Macchine virtuali

L'implementazione richiede anzitutto la scelta di un *hypervisor* (in grado di fornire virtualizzazione totale) e di una *box* Vagrant di riferimento; si è dunque optato per VirtualBox[1], nativamente supportato da Vagrant, e Debian Bookworm[2], un sistema operativo basato su Linux e noto per la sua stabilità ed efficienza.

Una funzionalità interessante di VirtualBox sono le *linked clone* [25], macchine virtuali basate sulle differenze con il disco della *box* di base, al fine di evitarne la copia totale (figura 3.1). In questo modo, l'*overhead* dovuto alla creazione di molteplici VM viene ridotto al minimo.

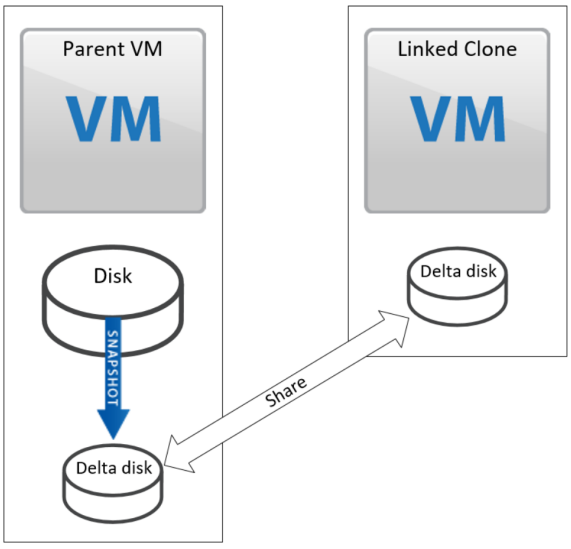

Figura 3.1: Relazione tra VM di base e *linked clone* [26]

Il *provisioning* delle macchine virtuali viene svolto da diversi *playbook* Ansible.

[1] `https://www.virtualbox.org/`
[2] `https://www.debian.org/releases/bookworm/`

DHCP & DNS

SLURM basa le proprie comunicazioni su nomi di host, astraendo da indirizzi IP; è quindi necessaria la corretta risoluzione DNS all'interno della LAN. La scelta di `dnsmasq` offre inoltre funzionalità DHCP per più reti.

Accounting

Essendo prevista la registrazione delle attività ("*accounting*") in ambienti multi-cluster SLURM, è necessaria la configurazione di un DB e relativo DBMS; è stato scelto MariaDB per familiarità con il contesto SQL. L'interazione tra il DBMS e il demone `slurmdbd`, responsabile dell'*accounting*, è configurabile seguendo la documentazione appropriata [27].

Topologia di rete

Una federazione SLURM, per definizione, è composta da più *cluster* in coordinamento tra loro; in contesti concreti, è verosimile che ciascuno di essi sia confinato nella propria LAN, dove la trasmissione verso l'esterno viene mediata da un *router* (o *gateway*). La figura 3.2 visualizza la topologia di rete per due *cluster* in federazione.

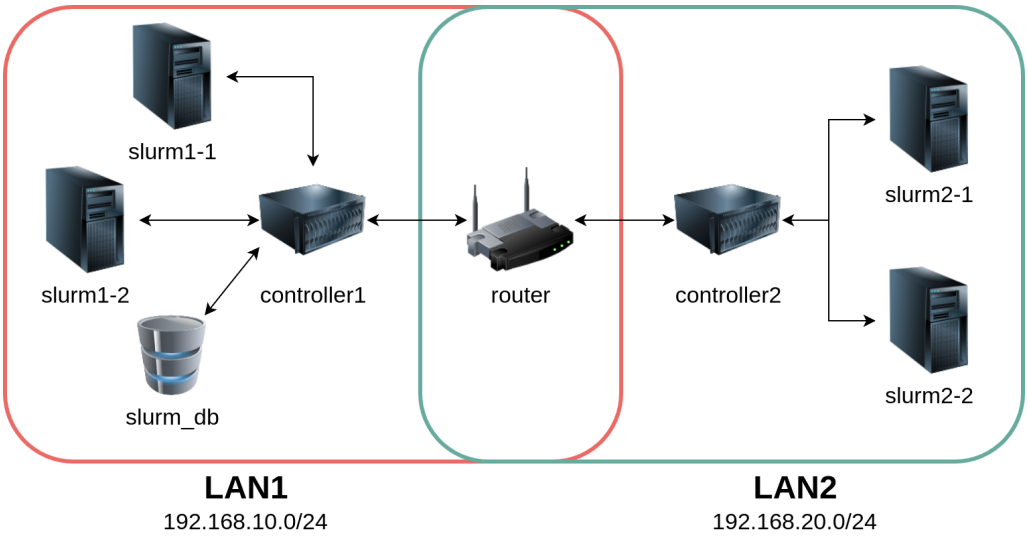

Figura 3.2: Topologia virtuale desiderata

In particolare, la configurazione della macchina virtuale `router` deve permettere la corretta comunicazione tra le due reti, inoltrando i pacchetti interessati.

Condivisione di risorse

La condivisione delle risorse disponibili su un nodo è configurabile grazie al *plugin* `select/cons_tres` [17]. Il progetto si pone, nello specifico, la condivisione di CPU, RAM e GPU; essendo quest'ultima catalogata come Generic Resource (GRES), è necessaria una configurazione aggiuntiva per il corretto interfacciamento [19].

Priorità di scheduling

Una volta attiva la condivisione di risorse, si può impostare la priorità di *scheduling*, ottenuta grazie alla definizione di una partizione SLURM separata e all'utilizzo del *plugin* `priority/multifactor` [20].

Sulla partizione viene applicata una Quality of Service (QoS) prioritaria [21], accessibile dall'utente desiderato e ristretta a una singola GPU del nodo di computazione [22].

Federazione di cluster

Ciascun *cluster* SLURM è descritto interamente da un proprio file di configurazione, che deve essere coerente su tutti i suoi nodi. Per la generazione di due installazioni, è dunque necessario definire due file separati, da inoltrare correttamente alle rispettive macchine virtuali.

Una prerogativa essenziale per l'impostazione di una federazione SLURM è la registrazione, sullo stesso DB, di tutti i cluster coinvolti, tramite l'uso di una singola istanza di `slurmdbd` [23].

4 | Implementazione

L'intero codice sviluppato per l'implementazione è consultabile pubblicamente su un *repository* Git[1].

4.1 Infrastruttura

4.1.1 Macchine virtuali

Il primo passo è la definizione delle macchine virtuali, seguendo il grafico e la nomenclatura presentati in figura 3.2; è necessario prestare attenzione ai seguenti aspetti:

- creazione delle VM come *linked clone* della *box* di base;
- corretto assegnamento delle reti interne VirtualBox (`LAN1` e `LAN2`);
- delega del *provisioning* a un *playbook* Ansible.

Il file di definizione 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 = "1024"
    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
```

[1] https://github.com/maxzrbn/Tirocinio-SLURM

```
  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
```

Ciascun nodo è quindi virtualmente dotato di 4 CPU e 1 Gigabyte (GB) di RAM. Da notare l'uso di iterazioni per la definizione compatta di molteplici VM.

Il *provisioning* Ansible viene definito in `site.yml`, attribuendo "ruoli" alle varie macchine virtuali; ciascuno di essi esegue una particolare sequenza di istruzioni.

site.yml

```
---
- hosts: router
  become: true
  roles:
    - common
    - router

- hosts: controller1
  become: true
  roles:
    - common
    - mariadb
    - slurmcommon
    - cluster1
    - slurmctlr

- hosts: controller2
  become: true
```

```
19    roles:
20      - common
21      - slurmcommon
22      - cluster2
23      - slurmctlr
24
25  - hosts: slurm1-1:slurm1-2
26    become: true
27    roles:
28      - common
29      - slurmcommon
30      - cluster1
31      - slurmwkr
32
33  - hosts: slurm2-1:slurm2-2
34    become: true
35    roles:
36      - common
37      - slurmcommon
38      - cluster2
39      - slurmwkr
```

4.1.2 Configurazione di rete

Svolgendo la funzione di DHCP/DNS server, la macchina virtuale `router` dispone di due interfacce di rete con indirizzi IP locali statici; è possibile configurarle grazie al seguente file, opportunamente inoltrato alla VM:

roles/router/files/eth_cfg

```
1  auto eth1
2  iface eth1 inet static
3    netmask 255.255.255.0
4    address 192.168.10.254
5
6  auto eth2
7  iface eth2 inet static
8    netmask 255.255.255.0
9    address 192.168.20.254
```

Affinché si possano effettivamente instradare i pacchetti tra le due LAN, è necessario attivare, su `router`, il *forwarding* a livello di *kernel*, modificando un particolare file di configurazione:

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
...
```

Le interfacce delle restanti VM vengono automaticamente configurate grazie al protocollo DHCP.

4.1.3 DHCP & DNS

Il servizio DHCP/DNS è svolto, per entrambe le reti, da `dnsmasq`, in esecuzione su `router`. La sua configurazione è la seguente:

roles/router/files/dnsmasq.conf

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

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

# ADDRESS RANGES FOR LEASING
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

# SPECIFY SELF AS NAMESERVER (for both LANs)
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 l'instradamento dei pacchetti provenienti da una LAN e destinati all'altra.

4.1.4 Accounting

La registrazione delle attività ("*accounting*") viene svolta dal demone `slurmdbd`, in esecuzione su `controller1`.

Database

Per integrare l'inserimento di password nel *provisioning*, si definiscono apposite variabili di esecuzione, eventualmente crittografabili con l'ausilio di Ansible Vault [7]:

secrets.yml

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

Al fine di isolare l'installazione del DB, si può utilizzare il software di containerizzazione `podman`[2], in grado di recuperare ed eseguire 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 `controller1`.

Seguendo poi la documentazione SLURM per il corretto interfacciamento tra il DBMS e `slurmdbd` [27], si generano DB e utente dedicati:

[2] `https://podman.io/`

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

- 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 (operazione di "*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: '%'
...
```

A questo punto è possibile verificare manualmente l'accesso al DB, presentandosi come utente `slurm` (utilizzato poi da `slurmdbd`):

```
vagrant@controller1:~$ 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]>_
```

slurmdbd

Si prosegue con l'installazione del demone `slurmdbd`:

roles/mariadb/tasks/main.yml

```
...
- name: Install 'slurmdb' package
  ansible.builtin.apt:
    name: 'slurmdbd'
    update_cache: true
...
```

Il suo file di configurazione è così composto:

slurmdbd.conf

```
AuthType=auth/munge
DbdHost=controller1
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
```

Da notare le opzioni:

- `AuthType=auth/munge`: attivazione del *plugin* MUNGE per l'autenticazione delle trasmissioni RPC;
- `DbdPort=6819`: porta TCP di ascolto per le comunicazioni con gli altri componenti SLURM.

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

```
vagrant@controller1:~$ 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 sul messaggio evidenziato si è appreso che `slurmdbd`, specificata l'opzione `StorageHost=localhost`, tenta la connessione al DB tramite *socket* locale di default (`/run/mysqld/mysqld.sock`) [27]; eseguendo però il DBMS all'interno di un *container*, la posizione della *socket* diventa relativa al *filesystem* astratto:

```
vagrant@controller1:~$ sudo find / -type s | grep 'mysqld.sock'
/home/mariadb/.local/share/containers/storage/overlay/e7d63d4ddde910afaf1d257
↪  4c3345fc74c42372bffc95f84528ead29c2f52d7e/diff/run/mysqld/mysqld.sock
```

È dunque necessario aggiornare il file di configurazione con `StorageHost=127.0.0.1`, in modo da forzare il demone all'utilizzo di porte TCP/IP.

4.1.5 Cluster SLURM

Il file di configurazione `slurm.conf` viene utilizzato da entrambi `slurmctld` e `slurmd`; in esso è descritto il generico comportamento del *cluster*:

roles/cluster1/templates/slurm.conf

```
# CLUSTER 1 CONFIG
#
ClusterName=cluster1
SlurmctldHost=controller1
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
AccountingStorageUser=slurm
JobAcctGatherType=jobacct_gather/cgroup
SlurmctldLogFile=/var/log/slurm/slurmctld.log
SlurmdLogFile=/var/log/slurm/slurmd.log
#
# COMPUTE NODES
NodeName=slurm1-[1-2] CPUs=4 State=UNKNOWN
PartitionName=debug Nodes=ALL Default=YES MaxTime=INFINITE State=UP
```

Con l'opzione evidenziata, `slurmctld` attiva e delega la registrazione delle attività a `slurmdbd`, in esecuzione sul medesimo nodo (`controller1`).

La configurazione del secondo *cluster* presenta invece alcune differenze:

roles/cluster2/templates/slurm.conf

```
# CLUSTER 2 CONFIG
#
ClusterName=cluster2
```

```
4 SlurmctldHost=controller2
5 ...
6 AccountingStorageHost=controller1
7 ...
8 NodeName=slurm2-[1-2] ...
9 ...
```

L'opzione in evidenza, specificata solo nel *cluster* senza DB, indica il nodo da contattare per l'*accounting* delle attività, dove viene eseguito `slurmdbd`.

4.2 Condivisione di risorse

Per la condivisione delle risorse è necessario modificare opportunamente i file di configurazione; di seguito vengono indicate le variazioni relative al primo *cluster*, ma le differenze con il secondo interessano esclusivamente la nomenclatura dei nodi:

roles/cluster1/templates/slurm.conf

```
1 ...
2 SelectType=select/cons_tres
3 SelectTypeParameters=CR_CPU_Memory
4 ...
5 AccountingStorageTRES=gres/gpu
6 ...
7 GresTypes=gpu
8 NodeName=slurm1-[1-2] CPUs=4 RealMemory=1024 Gres=gpu:rtx2080ti:2 State=UNKNOWN
9 ...
```

In particolare, le opzioni indicano [17]:

- `SelectType=select/cons_tres`: *plugin* per la gestione delle risorse ("*consumable trackable resources*"), permettendo l'esecuzione di più job sullo stesso nodo;
- `SelectTypeParameters=CR_CPU_Memory`: risorse ordinarie da condividere (nel caso in questione, CPU e RAM);
- `AccountingStorageTRES=gres/gpu`: attivazione dell'*accounting* relativo alle GPU;
- `GresTypes=gpu`: tipologie di Generic Resource (GRES) da condividere;
- `RealMemory=1024`: memoria totale in Megabyte (MB) allocabile per nodo di computazione (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.

È inoltre fondamentale introdurre nei nodi di computazione due ulteriori file di configurazione, per il corretto interfacciamento con le GRES [19]:

roles/slurmwkr/templates/cgroup.conf

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

In `gres.conf` vanno specificati:

- il nome del nodo di computazione;
- le precise caratteristiche delle risorse presenti.

gres.conf

```
NodeName=slurm1-1 Name=gpu Type=rtx2080ti Count=2 File=/dev/nvidia[0-1]
```

Una volta configurate, è possibile verificare la disponibilità delle risorse tramite il comando `sinfo`:

```
vagrant@controller1:~$ sinfo -o "%15N  %10c  %10m  %20G"
NODELIST         CPUS        MEMORY      GRES
slurm1-[1-2]     4           1024        gpu:rtx2080ti:2
```

4.3 Priorità di scheduling

L'applicazione di una priorità di *scheduling* necessita anzitutto di un contesto verosimile di utilizzo; a tal proposito vengono definiti utenti/gruppi di prova, tra cui quelli interessati dalla priorità (in questo caso, `user1`).

Affinché venga mantenuta la coerenza di utenti/gruppi all'interno del *cluster*, la loro definizione avviene con task comuni a tutti i nodi:

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
```

```
14    ansible.builtin.user:
15      name: "{{ item.name }}"
16      uid: "{{ item.id }}"
17      group: "{{ item.name }}"
18      state: present
19    loop:
20      - { name: 'user1', id: '2001' }
21      - { name: 'user2', id: '2002' }
22      - { name: 'user3', id: '2003' }
23      - { name: 'user4', id: '2004' }
24  ...
```

Il file di configurazione SLURM è stato modificato nel seguente modo:

roles/cluster1/templates/slurm.conf

```
1  ...
2  PriorityType=priority/multifactor
3  PriorityWeightQOS=10000
4  PriorityWeightTRES=GRES/gpu=100
5  ...
6  AccountingStorageEnforce=limits
7  JobAcctGatherType=jobacct_gather/cgroup
8  ...
9  PartitionName=gpupart Nodes=slurm1-[1-2] QOS=gpuprio AllowAccounts=user1acct
   ↪  State=UP
```

In particolare, le opzioni specificano:

- `PriorityType=priority/multifactor`: *plugin* per la priorità a molteplici fattori [20]; per il caso in questione si sono privilegiate, tramite le opzioni `PriorityWeight-`:
  - l'appartenenza a Quality of Service (QoS) prioritarie;
  - la richiesta di GPU per l'esecuzione;
- `AccountingStorageEnforce=limits`: imposizione dei limiti sulle risorse specificati nel DB; implica la presenza di un'associazione (utente, account, partizione, ...) valida per tutte le richieste di esecuzione [22];
- `PartitionName=gpupart`: partizione aggiuntiva ristretta all'account `user1acct`, con QoS `gpuprio` (definiti in seguito).

4.3.1 Impostazione del DB

È fondamentale la registrazione nel DB (tramite il comando `sacctmgr`) delle informazioni per il comportamento desiderato, ovvero:

- la QoS `gpuprio` [21];
- gli account `user1acct` (prioritario) e `noprio` [27].

```
vagrant@controller1:~$ 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@controller1:~$ sudo sacctmgr add account user1acct QOS=gpuprio
 Adding Account(s)
  user1acct
 Settings
  Description     = Account Name
  Organization    = Parent/Account Name
 Associations =
  C = cluster1    A = user1acct
 Settings
  QOS           = gpuprio
Would you like to commit changes? (You have 30 seconds to decide)
 (N/y): y
vagrant@controller1:~$ sudo sacctmgr add account noprio
 Adding Account(s)
  noprio
 Settings
  Description     = Account Name
  Organization    = Parent/Account Name
 Associations =
  C = cluster1    A = noprio
Would you like to commit changes? (You have 30 seconds to decide)
 (N/y): y
```

L'account `noprio` risulta essenziale affinché gli utenti non prioritari possano richiedere l'esecuzione di job. Da notare il confinamento della priorità a una singola GPU, tramite l'attributo `GrpTRES` [22].

Gli utenti, durante la registrazione nel DB, vengono collegati a un account di default, pertinente al comportamento desiderato:

```
vagrant@controller1:~$ sudo sacctmgr add user user1 DefaultAccount=user1acct
 Adding User(s)
  user1
```

```
 Settings
  Default Account = user1acct
 Associations =
  C = cluster1    A = user1acct            U = user1
Would you like to commit changes? (You have 30 seconds to decide)
 (N/y): y
vagrant@controller1:~$ sudo sacctmgr add user user2,user3,user4
↪  DefaultAccount=noprio
 Adding User(s)
  user2
  user3
  user4
 Settings
  Default Account = noprio
 Associations =
  C = cluster1    A = noprio               U = user2
  C = cluster1    A = noprio               U = user3
  C = cluster1    A = noprio               U = user4
Would you like to commit changes? (You have 30 seconds to decide)
 (N/y): y
```

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

```
vagrant@controller1:~$ sacctmgr show qos format=name,priority,GrpTRES
      Name   Priority       GrpTRES
---------- ---------- -------------
    normal          0
   gpuprio         10    gres/gpu=1
vagrant@controller1:~$ 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
```

4.4 Federazione di cluster

L'implementazione di una federazione SLURM si basa sulla corretta impostazione dei singoli *cluster*, svolta in sezione 4.1.5.

La seguente opzione, non essenziale per il funzionamento in sé, imposta la visualizzazione federata per i comandi `sacctmgr` e `scontrol` [23]:

roles/cluster1/templates/slurm.conf

```
...
FederationParameters=fed_display
...
```

4.4.1 Impostazione del DB

La registrazione di entrambi i *cluster* sullo stesso DB è un requisito fondamentale per l'impostazione della federazione [23]:

```
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`, si è dunque confermata la corretta comunicazione tra di essi:

```
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
```

5 | Risultati

L'accertamento della funzionalità dei servizi SLURM proposti richiede, in primo luogo, l'avvio dell'intera infrastruttura. Tramite il comando "`vagrant up`", la creazione e il *provisioning* delle macchine virtuali vengono automatizzati in stile Infrastructure as Code (IaC).

Il comando `sinfo` convalida la disponibilità dei *cluster* aderenti alla federazione:

```
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]
```

5.1 Test della condivisione di risorse

Per verificare la condivisione delle risorse presenti su un nodo, vengono allocati quattro job da 1 CPU (`-c1`) e 256 MB di RAM (`--mem=256`), specificandone lo svolgimento su un singolo nodo (`-N1`) da 4 CPU e 1 GB di RAM:

```
vagrant@controller1:~$ srun -N1 -c1 --mem=256 sleep 60 &
[1] 18254
vagrant@controller1:~$ srun -N1 -c1 --mem=256 sleep 60 &
[2] 18264
vagrant@controller1:~$ srun -N1 -c1 --mem=256 sleep 60 &
[3] 18274
vagrant@controller1:~$ srun -N1 -c1 --mem=256 sleep 60 &
[4] 18284
```

Grazie al comando `squeue` è possibile confermare l'esecuzione contemporanea dei job (tutti in stato *running*), a dimostrazione della condivisione di processori e memoria:

```
vagrant@controller1:~$ squeue
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
    1     debug    sleep  vagrant  R       0:15      1 slurm1-1
```

```
    2     debug    sleep  vagrant  R       0:14      1 slurm1-1
    3     debug    sleep  vagrant  R       0:13      1 slurm1-1
    4     debug    sleep  vagrant  R       0:12      1 slurm1-1
```

Allo stesso modo, vengono allocati due job richiedenti 1 GPU ciascuno (`--gpus=1`) verificando la loro esecuzione parallela e quindi la condivisione delle GPU:

```
vagrant@controller1:~$ srun -N1 --gpus=1 --mem=256 sleep 60 &
[1] 18294
vagrant@controller1:~$ srun -N1 --gpus=1 --mem=256 sleep 60 &
[2] 18304
vagrant@controller1:~$ squeue
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
    5     debug    sleep  vagrant  R       0:08      1 slurm1-1
    6     debug    sleep  vagrant  R       0:07      1 slurm1-1
```

5.2 Test della priorità di scheduling

Per confermare la priorità di *scheduling* su una GPU, da parte di un determinato utente, vengono allocati tre job non prioritari e un job lanciato da `user1` (sulla partizione prioritaria `gpupart`):

```
vagrant@controller1:~$ sudo -u user2 srun -N1 --gpus=1 --mem=256 sleep 60 &
[1] 18314
vagrant@controller1:~$ sudo -u user3 srun -N1 --gpus=1 --mem=256 sleep 60 &
[2] 18324
vagrant@controller1:~$ sudo -u user4 srun -N1 --gpus=1 --mem=256 sleep 60 &
[3] 18334
vagrant@controller1:~$ sudo -u user1 srun -N1 --partition=gpupart --gpus=1
↪  --mem=256 sleep 60 &
[4] 18344
vagrant@controller1:~$ 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 slurm1-1
    7     debug    sleep    user2  R       0:17      1 slurm1-1
   10   gpupart    sleep    user1 PD       0:00      1 (QOSGrpGRES)
```

Il job `10` risulta quindi in concorrenza di risorse con il job `9` (entrambi in stato *pending*). Attendendo la liberazione di una GPU (ossia il termine del job `7`), la partizione prioritaria ha la precedenza di *scheduling*:

```
vagrant@controller1:~$ 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 slurm1-1
   10   gpupart    sleep    user1  R       0:04      1 slurm1-1
```

5.3 Test della federazione di cluster

Grazie all'opzione `--clusters` in fase di sottomissione, è possibile specificare una lista di *cluster* sulla quale tentare di eseguire il job richiesto (ad esempio, il programma `hostname`):

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

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

È importante notare come, per entrambe le sottomissioni di esempio:

- il controllore di partenza non appartenga al *cluster* specificato per l'esecuzione;
- venga richiesta l'esecuzione del job su due nodi distinti (opzione `-N2`).

Il risultato di `hostname` conferma l'inoltro dei job tra i due *cluster* e quindi il corretto comportamento della federazione.

6 | Conclusioni

Il progetto di tesi si è concentrato sullo sviluppo di un'infrastruttura virtuale, al fine di implementare particolari servizi di calcolo SLURM. Si sono scelti strumenti di gestione e configurazione automatica di macchine virtuali (nello specifico, Vagrant & Ansible), ottenendo una struttura Infrastructure as Code (IaC). Grazie all'uso di Git[1], un software *open source* per il controllo delle versioni, il processo di sviluppo è stato supportato da:

- cronologia delle modifiche;
- progettazione non lineare delle funzionalità.

In primo luogo, si sono dovute integrare le caratteristiche di base per l'operatività dell'infrastruttura, come assegnamento DHCP e risoluzione DNS (per entrambe le reti virtuali); a tal fine si è scelto lo strumento `dnsmasq`, di semplice impostazione.

Seguendo la documentazione appropriata, si sono poi configurati i vari componenti di un'installazione SLURM distribuita, prestando particolare attenzione alla configurazione di `slurmdbd` e DBMS associato.

Si è dunque passati all'implementazione dei particolari servizi posti come obiettivo del progetto.

La condivisione delle risorse di computazione permette il loro assegnamento dinamico in base alle necessità dei job da eseguire; si è definita la condivisione di:

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

In particolare, il corretto interfacciamento con la GPU, catalogata da SLURM come Generic Resource (GRES), ha richiesto ulteriore configurazione.

Per implementare la priorità di *scheduling*, riservata a un determinato utente e ristretta a una specifica risorsa, si è aggiunta una partizione SLURM dedicata; essa risulta:

- prioritaria su una singola GPU, grazie alla definizione di Quality of Service (QoS);

[1] `https://git-scm.com/`

- accessibile esclusivamente dall'utente interessato, in accordo con l'*accounting* impostato nel DB.

L'attivazione della federazione di *cluster* SLURM riguarda essenzialmente la loro registrazione sul DB in comune; una volta inseriti, le richieste di esecuzione vengono inoltrate tra di essi, a seconda della necessità.

Si è infine svolta la fase di verifica delle funzionalità implementate, tramite la sottomissione mirata di programmi e l'accertamento della loro esecuzione.

6.1 Sviluppi futuri

Il modello di sviluppo IaC include numerosi vantaggi a beneficio dell'integrazione di un servizio o un'infrastruttura; tra i più rilevanti si ricordano:

- rapidità ed efficienza di sviluppo;
- riproduzione coerente degli ambienti;
- riduzione degli errori di configurazione dovuti alla gestione manuale.

Sono dunque possibili diversi sviluppi per l'infrastruttura progettata in questo elaborato, ad esempio:

- l'ampliamento delle funzionalità di calcolo disponibili, come *preemption* (sospensione di un job a bassa priorità per l'esecuzione di uno a priorità maggiore [28]) e *gang scheduling* (sospensione alternata dell'esecuzione di più job su una stessa risorsa [29]);
- l'implementazione di prassi *DevOps*, come test automatizzati, provocati dalla modifica dell'infrastruttura (principio noto come Continuous Integration & Continuous Delivery (CI/CD));
- l'applicazione in contesti reali, per la concreta erogazione di calcolo ad alte prestazioni.

Bibliografia

[1] *High Performance Computing (Wikipedia)*. URL: https://it.wikipedia.org/wiki/High_performance_computing (visitato il giorno 20/02/2025).

[2] *Virtualization (Wikipedia)*. URL: https://en.wikipedia.org/wiki/Virtualization (visitato il giorno 21/02/2025).

[3] *Vagrant (Wikipedia)*. URL: https://it.wikipedia.org/wiki/Vagrant_(software) (visitato il giorno 21/02/2025).

[4] *Provisioning (Wikipedia)*. URL: https://en.wikipedia.org/wiki/Provisioning_(technology) (visitato il giorno 22/02/2025).

[5] *Ansible (Wikipedia)*. URL: https://en.wikipedia.org/wiki/Ansible_(software) (visitato il giorno 22/02/2025).

[6] *Infrastructure as Code (Wikipedia)*. URL: https://en.wikipedia.org/wiki/Infrastructure_as_code (visitato il giorno 23/02/2025).

[7] *Ansible Documentation*. URL: https://docs.ansible.com/ (visitato il giorno 22/02/2025).

[8] *Domain Name System (Wikipedia)*. URL: https://en.wikipedia.org/wiki/Domain_Name_System (visitato il giorno 23/02/2025).

[9] *Dynamic Host Configuration Protocol (Wikipedia)*. URL: https://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol (visitato il giorno 23/02/2025).

[10] *Dnsmasq (Wikipedia)*. URL: https://en.wikipedia.org/wiki/Dnsmasq (visitato il giorno 24/02/2025).

[11] *Database (Wikipedia)*. URL: https://en.wikipedia.org/wiki/Database (visitato il giorno 24/02/2025).

[12] *MariaDB (Wikipedia)*. URL: https://en.wikipedia.org/wiki/MariaDB (visitato il giorno 24/02/2025).

[13] *Job Scheduler (Wikipedia)*. URL: https://en.wikipedia.org/wiki/Job_scheduler (visitato il giorno 25/02/2025).

[14] *SLURM Overview*. URL: https://slurm.schedmd.com/overview.html (visitato il giorno 25/02/2025).

[15] *SLURM Authentication Guide*. URL: https://slurm.schedmd.com/authentication.html (visitato il giorno 26/02/2025).

[16] *MUNGE Documentation*. URL: https://github.com/dun/munge/wiki (visitato il giorno 25/02/2025).

[17] *SLURM Consumable Resources Guide*. URL: https://slurm.schedmd.com/cons_tres.html (visitato il giorno 26/02/2025).

[18] *SLURM Multi-core/Multi-thread Support*. URL: https://slurm.schedmd.com/mc_support.html (visitato il giorno 26/02/2025).

[19] *SLURM Generic Resources Support*. URL: https://slurm.schedmd.com/gres.html (visitato il giorno 26/02/2025).

[20] *SLURM Priority Multifactor Guide*. URL: https://slurm.schedmd.com/priority_multifactor.html (visitato il giorno 27/02/2025).

[21] *SLURM Quality of Service Guide*. URL: https://slurm.schedmd.com/qos.html (visitato il giorno 27/02/2025).

[22] *SLURM Resource Limits Guide*. URL: https://slurm.schedmd.com/resource_limits.html (visitato il giorno 27/02/2025).

[23] *SLURM Federation Guide*. URL: https://slurm.schedmd.com/federation.html (visitato il giorno 28/02/2025).

[24] *SLURM Network Configuration Guide*. URL: https://slurm.schedmd.com/network.html (visitato il giorno 28/02/2025).

[25] *Vagrant Linked Clones*. URL: https://developer.hashicorp.com/vagrant/docs/providers/virtualbox/configuration#linked-clones (visitato il giorno 01/03/2025).

[26] *Cloning VMs*. URL: https://itgaiden.com/2019-02-25-cloning-vms-in-vsphere-part-1/ (visitato il giorno 01/03/2025).

[27] *SLURM Accounting Guide*. URL: https://slurm.schedmd.com/accounting.html (visitato il giorno 02/03/2025).

[28] *SLURM Preemption Guide*. URL: https://slurm.schedmd.com/preempt.html (visitato il giorno 06/03/2025).

[29] *SLURM Gang Scheduling Guide*. URL: https://slurm.schedmd.com/gang_scheduling.html (visitato il giorno 06/03/2025).

Acronimi

Elenco delle figure

Ringraziamenti

Ringrazio i miei relatori di tesi, il Prof. Prandini e l'Ing. Giovine, per il supporto e l'opportunità di approfondire l'argomento.

Ringrazio la mia famiglia, per il loro continuo sostegno e incoraggiamento verso la realizzazione dei miei obiettivi.

Ringrazio infine gli amici di studio, di liceo, di tennistavolo, per avermi accompagnato durante questo percorso.