UNIVERSITA' DEGLI STUDI DI PADOVA
Dipartimento Di Ingegneria Dell'Informazione
Corso Di Laurea In Ingegneria Informatica

PariCluster: supporto per simulazioni di reti su cluster

RELATORE: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi
CORRELATORE: Dott. Michele Bonazza

LAUREANDO: Luca Demo

Anno Accademico 2012/2013

Alla mia famiglia
mio grande sostegno

Indice

Introduzione

Questo elaborato descrive la progettazione e l'implementazione di *PariCluster*, software di supporto per la simulazione di una rete di *client* su cluster in funzione ai requisiti espressi. Saranno messe in evidenza l'integrazione della gestione del cluster attraverso il programma di resource manager "slurm" e alcuni aspetti del linguaggio Python scelto per la scrittura del codice.

1 PariPari

In questo primo capitolo verrà presentata una descrizione generale del progetto *PariPari* e al contesto che ha portato alla necessità di sviluppo del tool *PariCluster*.

1.1 Cos'è *PariPari*

PariPari è una rete *Peer to Peer serverless* in sviluppo presso il dipartimento di ingegneria dell'informazione dell'università di Padova. Allo sviluppo di questo progetto sono ogni anno interessati studenti del dipartimento il cui contributo ha permesso di mettere le basi per la costruzione di una nuova piattaforma di file sharing e storage distribuito su rete internet. Il *client* di *PariPari*, che è scritto quasi interamente in linguaggio *Java*, si presenta come un sistema fortemente modulare, infatti il suo nucleo, il *Core*, è progettato per essere esteso mediante dei plugins sviluppati in modo indipendente dai team che li seguono. I plugin, ognuno dei quali offre una particolare funzionalità, si possono suddividere in due fondamentali gruppi: plugin della *cerchia interna* e della *cerchia esterna*. I primi sono i plugin fondamentali che forniscono le funzionalità necessarie per la gestione della rete *PariPari* stessa e permettono agli altri plugin di operare. Tra di essi si possono riconoscere:

- **Core**: è il componente fondamentale, contiene il punto di ingresso dell'intero programma e gestisce l'interazione fra tutti gli altri plugin.
- **Connectivity**: fornisce a tutti i plugin le funzionalità di accesso alla rete
- **DHT (Distributed Hash Table)**: realizza la rete *PariPari* serverless permettendo la ricerca e il recupero di risorse all'interno della rete stessa
- **Local Storage**: gestisce l'archiviazione nella memoria di massa dei dati per conto dei plugin

I plugin della cerchia esterna estendono *PariPari* offrendo funzionalità aggiuntive e permettendo tra l'altro l'accesso ad altre reti P2P o l'uso di protocolli di comunicazione già esistenti. Alcuni di essi sono per esempio:

- **Mulo**: *client* per l'accesso alla rete P2P eDonkey.
- **Torrent**: *client* per l'accesso alla rete BitTorrent
- **IRC – IM**: plugins che permettono l'uso dei servizi di chat Internet Relay Chat e Instant Messenger.
- **Distributed Storage**: funzioni di archiviazione distribuita nella rete *PariPari*
- **VoIP**: rende possibile una conversazione telefonica sfruttando la connessione internet

- **GUI**: fornisce una interfaccia grafica al programma

1.2 Perchè PariCluster

Come ogni software di una certa complessità anche *PariPari* durante lo sviluppo richiede una continua attività di testing.
I programmatori di *PariPari* durante la scrittura del codice sviluppano parallelamente delle routine di test con l'ausilio di *JUnit*, un framework di testing per *Java*, per verificare che le singole porzioni di codice operino nelle modalità previste fornendo i risultati attesi. Questa tecnica di testing è definita *"unit testing"* perché relativa alle più piccole unità di codice indipendenti testabili, siano esse funzioni, metodi o classi. Tuttavia, pur garantita la stabilità e la correttezza di un codice, ciò che prima di ogni release è necessario verificare è la correttezza e l'effettivo funzionamento del sistema.
E' immediatamente chiaro che una verifica di tale qualità non è possibile limitandosi al testing delle singole componenti, ma è necessario un completo collaudo del programma o, nel caso di *PariPari*, dei singoli *plugin*.

Riferendoci alla suddivisione dei *plugin* proposta nel paragrafo precedente, si può intuire come quelli della *cerchia esterna* siano in gran parte collaudabili direttamente in quanto relativi a protocolli, reti e sistemi già esistenti e stabili, come le reti *eDonkey* e *BitTorrent*. Questi usano inoltre solo una parte delle funzionalità dei plugin interni in quanto non fanno propriamente uso della rete *PariPari*. Arrivati ad uno stadio di sviluppo soddisfacente, si è reso necessario collaudare la rete *PariPari* facendo interagire un numero significativo di *client* per verificarne il comportamento in una rete di una certa complessità. Test di questo tipo non possono sicuramente essere delegati a singole persone in possesso di una copia del software, sia perché la disponibilità di risorse umane non è sufficiente, sia perché tali procedure richiederebbero molto tempo e coordinazione. Si pensi per esempio se si dovesse testare la rete nella quale diversi gruppi di peer hanno una specifica configurazione diversa dagli altri o se si dovessero simulare la morte di nodi o cambiare continuamente la configurazione dei singoli peer. E' evidente che risulta necessario un approccio più automatizzato e diretto, che permetta un controllo immediato dei singoli peer o gruppi di essi da parte di un solo operatore. La scelta a questo punto è stata quella di simulare una rete *PariPari* su un cluster capace di operare con centinaia di istanze del programma, da qui il nome del tools oggetto di questa discussione, appunto *PariCluster*.

2 Cluster

In questo secondo capitolo verrà proposta una trattazione sommaria dei sistemi *cluster* e analizzata l'architettura di quello ad uso del dipartimento di ingegneria dell'informazione.

2.1 Cos'è un cluster

Un cluster di computer può essere definito come un insieme di calcolatori, detti nodi del cluster, connessi fra loro e capaci di operare parallelamente ed apparire, sotto certi aspetti, come un singolo sistema di calcolo. Un cluster è quindi costituito da diversi calcolatori fra loro indipendenti, ognuno dei quali dispone di un sistema operativo e di proprie risorse fisiche (memoria centrale, memoria di massa, cpu...) ed eventuali periferiche.
Ogni nodo del cluster esegue in locale un software di controllo che permette l'acquisizione, l'esecuzione e il controllo dei *job*, ovvero le richieste di "lavoro". Tale software rappresenta inoltre l'interfaccia attraverso la quale l'utente gestisce il cluster. Lo scheduling dei *job* e la loro assegnazione è gestita da un nodo del cluster definito master node, mentre l'esecuzione dei *job* stessi avviene sui nodi definiti compute node.

2.1.1 Scheduling

Esistono diverse politiche di scheduling dei processi e algoritmi decisionali per l'assegnazione dei *job* che determinano le caratteristiche del particolare cluster. In base a queste si possono individuare tre principali tipi di cluster:

- **Load Balancing Cluster**: sono cluster nei quali i nuovi *job* sono assegnati ai compute node meno carichi, nell'ottica di sfruttare il numero maggiore di risorse possibile in un modo ottimale.
- **High Performance Cluster**: sono progettati per offrire la massima potenza di calcolo in caso a *job* altamente parallelizzabili.
- **Fail-Over Cluster**: sono cluster nei quali ogni nodo dispone di un secondo nodo di backup che subentra nel caso in cui il primo risulti non disponibile. Questo tipo di cluster è usato in ambiti critici nei quali è fondamentale la continua disponibilità del sistema.

2.2 Eridano

Nel capitolo precedente si è detto che la simulazione di una rete *PariPari* avverrà mediante l'uso di un cluster, in particolare quello in dotazione al dipartimento di Ingegneria dell'Informazione dell'università di Padova. Il nome del cluster in questione è Eridano. Eridano è composto da 16 compute node connessi fra loro mediante due network fisiche operanti una a 10Gbps per fornire una intercomunicazione veloce fra i nodi e una a 1Gbps per l'accesso dall'esterno e la gestione del cluster. Ogni nodo dispone di un processore intel core i7 950 (4 core – 8 threads), 12GB di memoria RAM DDR3 Triple Channel e 6 HDD magnetici da 1TB ciascuno. Su ogni nodo è installato il sistema operativo Debian nella versione 6.0 e il software di controllo del cluster è *Slurm*. L'accesso alle macchine del cluster avviene mediante ssh attraverso il server Stargate, esterno al cluster.

2.3 Slurm

Slurm ("Simple Linux Utility for Resource Management") è un software open-source di gestione e monitoraggio di *job* per cluster linux ed è attualmente utilizzato per il controllo di Eridano. In un cluster gestito con *Slurm*, su ogni compute node gira un demone slurmd mentre sul master node il demone slurmctld . Ogni elaboratore è considerato una risorsa di tipo node ed è possibile definire delle partizioni che corrispondono a set di nodi. *Slurm* fornisce un insieme completo di strumenti per la gestione dei *job* nel cluster e in particolare:

1. Permette l'allocazione in modo esclusivo o non esclusivo delle risorse del cluster
2. Fornisce un framework per avviare, monitorare e fermare il flusso di lavoro su uno o più compute node allocati
3. Gestisce la contenzione delle risorse del sistema cluster (nodi, core, thread, memoria) mediante l'uso di una coda per i *job* in attesa.

2.3.1 Risorse

Quando un cluster è gestito mediante *Slurm*, il concetto di risorsa consumabile del cluster può assumere un significato più specifico del singolo nodo o gruppo di nodi. Secondo la configurazione standard, ogni nodo è considerato come una singola risorsa che viene allocata e assegnata in modo esclusivo a un unico *job* per volta. In Eridano è però attivo un plugin di *Slurm* chiamato "Consumable Resource Node Allocation Plugin" (CRNA) che permette una gestione più capillare delle risorse del cluster. Il concetto di risorsa consumabile viene quindi associato alle

seguenti risorse fisiche di ogni nodo:

- **CPU**: non c'è nozione di core, thread e *socket*. Nella configurazione standard:
  - nei sistemi con singolo core senza hyperthreading, la risorsa CPU rappresenta la cpu stessa
  - in sistemi multi-core, rappresenta i singoli core
  - in sistemi single o multi-core con hyperthreading rappresenta i singoli thread (detti anche core logici)
- **Socket**: questa risorsa corrisponde al *socket* fisico di installazione del processore
- **Core**: risorsa corrispondente a un core del processore
- **Memoria**: corrisponde alla memoria centrale dei nodi, di default è considerata risorsa condivisa

In Eridano esistono quindi 16 nodi per un totale di:

- 16 *socket*
- 64 cores
- 128 threads
- 16 risorse di memoria condivise fra i thread e cores dello stesso nodo

2.3.2 Allocazione delle risorse

In *Slurm*, ogni risorsa, prima di essere utilizzata e quindi assegnata per l'esecuzione di un *job*, deve essere allocata. L'allocazione consiste nella assegnazione a un *job* della data risorsa in modo che questa sia considerata non più disponibile se l'allocazione è esclusiva, o comunque non idle (inattiva) se è previsto un certo livello di condivisione.
L'allocazione di una risorsa in *Slurm* avviene mediante l'uso del comando salloc dalla console di uno dei nodi.
I parametri del comando salloc permettono all'utente di allocare una generica risorsa del cluster, la cui entità dipende dalla configurazione di *Slurm*. Nella configurazione standard viene allocato un intero nodo, se attivo il plugin CRNA visto in precedenza viene invece allocato un core e tutta la memoria sul nodo di appartenenza del core stesso a meno che non sia specificato diversamente.

2.3.3 Job e Job Step

Una volta allocate, le risorse possono essere utilizzate per l'esecuzione del *job* a cui sono destinate. Il significato di *Job* fino ad ora emerso è quello generale

di “lavoro”, ovvero di un indefinito carico elaborativo da eseguire. Tuttavia in *Slurm* i *job* assumono un significato più direttamente collegato alle risorse che all’esecuzione del lavoro vero e proprio. Si è visto che il comando salloc alloca e mette a disposizione dell’utente un certo quantitativo di risorse, il risultato di tale operazione è la creazione di un *job* associato a tali risorse. Esiste dunque un rapporto diretto tra il concetto di *Job* e risorse allocate allo stesso. In *Slurm* ogni *job*, identificato da un codice univoco, rappresenta quindi un insieme di risorse allocate pronte ad essere utilizzate o già in uso. Allocate le risorse e ottenuto il *job* relativo, è possibile chiedere al cluster l’esecuzione dei comandi o degli applicativi per cui si è richiesto il *job*, tali richieste assumono il nome di *Job* Step e fino al loro completamento potranno usare tutte o una parte delle risorse assegnate al *Job*. Il comando usato per l’inizio dello Step è srun, alternativamente *Slurm* permette l’esecuzione di script di comandi srun mediante il comando sbatch.

3 Progettazione

In questo capitolo verranno presentati i requisiti ai quali *PariCluster* dovrà offrire risposta. Si introducono inoltre alcuni aspetti che è utile affrontare fin dalle prime fasi di sviluppo.

3.1 Requisiti

Come accennato nel primo capitolo di questa trattazione, il progetto di *PariCluster* nasce con lo scopo di fornire un tool di supporto al team di sviluppo di *PariPari* per facilitare l'attività di testing mediante simulazioni di reti di client. In particolare *PariCluster* dovrà essere realizzato considerando fondamentali i seguenti requisiti:

- Automatizzare l'avvio e l'arresto di un numero arbitrario di istanze di *client PariPari*
- Permettere una gestione delle istanze in modo singolo o in forma di gruppi di esse.
- Possibilità di inviare comandi testuali per mezzo di *socket* a singole istanze o gruppi di esse.
- Salvataggio e successivo ripristino dello stato della rete simulata

3.2 Ambiente operativo

L'ambiente che ospiterà *PariCluster* corrisponde a un cluster, in particolare al cluster Eridano del dipartimento di Ingegneria dell'Informazione. E' utile ora studiare il sistema in questione e porre attenzione a quelle caratteristiche che risultano vincolanti o che è possibile sfruttare.

a) Ricordiamo che il sistema operativo comune a tutto il cluster è Debian 6.0, per cui possiamo affermare che:
   - il software da realizzare sarà sviluppato necessariamente per lavorare in ambiente GNU/Linux
   - sarà utile scegliere un linguaggio di programmazione che disponga di librerie sufficientemente sviluppate e stabili per l'ambiente linux

b) In accordo alle caratteristiche tipiche di un cluster, la cartella home è condivisa fra tutte le postazioni attraverso il protocollo NFS (*Network File*

System)e quindi un dato all'interno di esse è univocamente identificabile dalla sua path e accessibile da tutte le macchine del cluster. Conseguenza di questo è che anche ogni modifica apportata da uno dei nodi sarà visibile a tutti gli altri.
Si evince quindi che:

- è possibile sfruttare la condivisione della cartella home per accedere ai dati in modo trasparente da ogni nodo usando percorsi interni alla cartella home.
- Installando il programma nella cartella home, il codice sarà disponibile su ogni nodo del cluster, rendendo immediato l'utilizzo del software o di parte di esso (es: script) su ogni nodo.

c) Il software che gestisce il cluster è *Slurm*, tuttavia le macchine sono tutte raggiungibili mediante connessione ssh. Si possono valutare quindi due possibilità: usare *Slurm* per assegnare le istanze di *PariPari* oppure avviare le singole istanze mediante connessioni ssh.

3.3 Idea Risolutiva

Da una prima analisi si intuisce che *PariCluster* sarà costituito da un programma di controllo centrale identificabile come un server che fornirà l'interfacciamento con l'utente.
In modo duale esisterà una componente decentralizzata che potrebbe corrispondere alle singole istanze dei *client PariPari*.

3.3.1 Executor

Per avviare una istanza del *client* di *PariPari* è necessario richiamare la virtual machine di Java da riga di commando specificando l'esecuzione del file .jar `Core.jar` contenente il punto di ingresso del client.
E' tuttavia necessario porre attenzione a un aspetto fondamentale del *client PariPari*: una volta avviato in modalità remota, tenterà di aprire in ascolto la porta TCP 10000 su cui ricevere i comandi.
L'immediato inconveniente è che se la porta è già in uso da un altro programma o semplicemente da un altro *client PariPari* avviato in precedenza sullo stesso nodo, quella porta non sarà più disponibile. Il *client* di *PariPari* prevede, in tale circostanza, di usare la prima porta libera a partire dalla successiva.
Tuttavia in questo caso:

- Se la prima porta non è libera, il *client* ne userà un'altra, ma il controllore non lo saprà

- Se il *client* fallisce l'avvio, non verrà aperta nessuna porta, ma al controllore non giungerà nessuna notifica di tale situazione

Per rimediare a queste situazioni di incertezza si può pensare all'uso di un intermediario tra il controllore e il *client* di *PariPari*, per esempio uno script che potrà scegliere una porta locale libera da indicare al client. Si è in un primo momento valutata proprio questa ipotesi, tuttavia, dopo qualche tentativo, si sono resi subito evidenti i limiti di questo approccio:

- Il processo iterativo di ricerca della porta è potenzialmente lungo e inefficiente quando molti *client* sono stati già avviati
- L'avvio di due o più script in rapida successione può portare diversi script a identificare come libera la stessa porta, portando inevitabilmente a una erronea associazione di più *client* alla stessa, nonostante solo un *client* effettivamente utilizzi la data porta.

Si è quindi valutata l'opzione di lasciare al *client* l'onere di scegliere la porta da usare, sapendo che in tal caso nessun *client* potrà avviarsi usando una porta già utilizzata da un altro. In modalità remota, specificando cioè il non caricamento del plugin GUI della console grafica, il *client* presenta un output piuttosto dettagliato nel descrivere il processo di avvio. In particolare comunica su quale porta il *client* è in ascolto per i comandi impartibili da remoto.
Con un parsing dell'output del client, *PariCluster* può quindi identificare la porta e comunicarla al manager.
Questo approccio è fin da subito risultato comodo e immediato, ed è stato considerato come il più valido e quindi adottato.
Un altro aspetto da non ignorare è la chiusura del client, in quanto, anche in questo caso, nessuna notifica è prevista da parte del *client* stesso. E' tuttavia possibile rimediare lasciando lo script in attesa della chiusura del *client* che esso stesso ha lanciato e terminare solo dopo che questo è a sua volta terminato.
In questo modo, le ultime istruzioni dello script potrebbero essere la notifica al controllore dell'effettiva chiusura del *client* associato. Tale approccio si è dimostrato valido e si è deciso quindi di adottarlo con l'uso di uno script al quale viene dato il nome di "Executor" in quanto "esegue" il lancio del client.

3.3.2 Manager

Al modulo di controllo di *PariCluster* è stato dato, ai fini della trattazione, il semplice nome di "Manager" in quanto gestisce la simulazione della rete. A differenza di Executor, il codice di Manager durante una simulazione è attivo con una sola istanza su un solo nodo. Al Manager competono tutte le funzioni di

controllo della simulazione tra cui:

- Avvio dei *client* mediante richiesta di esecuzione dello script Executor su un nodo
- Controllo dei *client* mediante connessioni *socket*
- Organizzazione logica dei *client* definendo dei gruppi di client
- Arresto dei *client* "gentile" mediante invio del comando di chiusura.
- Salvataggio dello stato della simulazione e caricamento dello stesso in successive simulazioni

L'interfaccia per l'utente sarà costituita, almeno per le prime versioni del software, da una riga di comando.

3.3.3 Gestione della simulazione

Eridano non è un cluster ad uso esclusivo di *PariPari* e diversi utenti possono accedervi anche contemporaneamente per usarlo.
Il fatto che tutti i nodi siano raggiungibili mediante connessione ssh, rende possibile la richiesta di esecuzione dello script Executor anche senza usare *Slurm*. Questo approccio, per quanto semplice, presenta però diversi svantaggi nel caso in esame:

- non si conosce lo stato di utilizzo delle risorse del cluster
- le probabilità di possibili interferenze con altri processi è relativamente alta
- è necessario conoscere gli indirizzi di rete dei nodi e verificare se questi sono disponibili

L'utilizzo di *Slurm* è sicuramente una scelta migliore in quanto non presenta i suddetti problemi, e porta anche dei vantaggi non trascurabili:

- Al progetto *PariCluster* è assegnata una partizione del cluster, quindi il nostro software delegherà a *Slurm* la selezione e l'allocazione delle risorse fra quelle esplicitamente dedicate alla simulazione.
- *PariCluster* dovrà solo richiedere l'allocazione di uno o più *job* , e successivamente, per ogni istanza del *client PariPari*, creare un *Job* Step. Vediamo ora come è possibile integrare l'uso di *Slurm* in *PariCluster*.

Nodi Prima di poter avviare un processo, che nel nostro caso è un *client* di *PariPari*, è necessario aver allocato delle risorse, ovvero aver richiesto l'inizio di un *job* a *Slurm*.

Il comando *Slurm* per l'apertura di un *job* , come si è visto, è salloc ed è chiamato aprendo dallo script un sottoprocesso di *shell* bash. Salloc supporta l'uso di molti parametri per specificare i dettagli della richiesta di allocazione, tuttavia è importante solo specificare che verranno usate le risorse fornite dalla partizione destinata alla simulazione.
Come conseguenza delle impostazioni del cluster Eridano, salloc allocherà una risorsa di tipo Core che corrisponde a un Core fisico fra i quattro a disposizione di ogni nodo.
Una volta eseguito con successo il comando salloc, un nuovo *job* sarà allocato e pronto per l'utilizzo. Ciò che però a una prima analisi può sfuggire è il fatto che il sottoprocesso *bash* usato per l'allocazione ha a sua volta ottenuto due altri sotto processi.

```
lucademo@eridano10:~$ ps U lucademo
  PID TTY      STAT   TIME COMMAND
 7899 ?        S      0:00 sshd: lucademo@pts/2
 7900 pts/2    Ss     0:00 -bash
 9336 pts/2    R+     0:00 ps U lucademo
lucademo@eridano10:~$ salloc -p paripari
salloc: Granted job allocation 4905
lucademo@eridano10:~$ salloc -p paripari
salloc: Granted job allocation 4906
lucademo@eridano10:~$ ps U lucademo
  PID TTY      STAT   TIME COMMAND
 7899 ?        S      0:00 sshd: lucademo@pts/2
 7900 pts/2    Ss     0:00 -bash
 9340 pts/2    Sl     0:00 salloc -p paripari
 9343 pts/2    S      0:00 /bin/bash
 9417 pts/2    Sl     0:00 salloc -p paripari
 9420 pts/2    S      0:00 /bin/bash
 9472 pts/2    R+     0:00 ps U lucademo
lucademo@eridano10:~$
```

Figure 3.1: Dopo l'allocazione di due *Job* sono presenti 4 nuovi processi

Questi processi in background sono vitali per il ciclo di vita del *job* e la chiusura del processo padre a questo punto provocherebbe la propagazione del segnale SIGHUP di chiusura a tutti i subprocess aperti da salloc deallocando conseguentemente il *job* e terminando tutti i *job* Step, ovvero i *client PariPari*. Esiste un comando POSIX che può essere usato per evitare la propagazione del segnale di chiusura, ovvero il comando nohup.
Si noti tuttavia che la propagazione del segnale non è per forza da considerare un comportamento problematico e anzi può essere sfruttato per evitare che alla chiusura (eventualmente inattesa) di *PariCluster*, rimangano

`allocati tutti i` *`job`* .
E' quindi sufficiente prendere atto di questo legame e non chiudere il processo *shell* avviato per effettuare l'allocazione del *job* .
All'interno di *PariCluster* la gestione delle risorse del cluster è trasparente all'utente e si assume che quest'ultimo sia solo consapevole che esiste una certa entità collegata alle risorse del cluster, di nome "Nodo" che "ospita" un certo numero di *client* di *PariPari*.
Ogni nodo corrisponde quindi a un *job* e l'utente, mediante un comando, potrà crearne nella quantità desiderata, ovviamente entro i limiti concessi dalla quantità di risorse disponibili nel cluster.

PariPariCores Quando nell'ambiente di simulazione *PariCluster* è presente almeno un nodo, l'utente può chiedere su questi l'avvio di uno o più *client* di *PariPari*. Anche in questo caso l'operazione è trasparente all'utente, il quale dovrà solo usare un comando per avviare il numero desiderato di client. L'Executor, avviato con un comando srun, a questo punto contatta il master e in seguito a un positivo avvio del *client PariPari*, gli trasmette indirizzo e porta aperti da quest'ultimo.
Il manager riceve il messaggio e aggiunge un nuovo elemento alla lista dei *client* attivi.
Quando lo script executor terminerà, in seguito alla chiusura del client, il manager riceverà un messaggio di notifica e la voce relativa a quel *client* sarà rimossa dalla lista.
Il programma memorizza quindi la lista dei *client PariPari* aperti come una tupla composta da:

- Identificativo del client
- Indirizzo nella rete cluster
- Porta di comunicazione

Per l'utente, all'interno di *PariCluster*, i *client* sono noti con il nome di "*PariPariCores*".

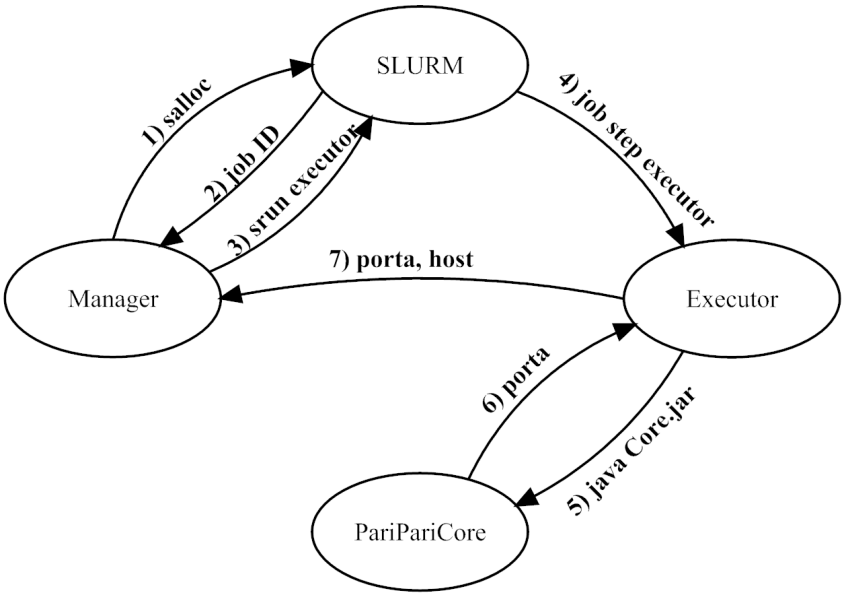

Figure 3.2: Fasi di avvio di un *client PariPari* con *Slurm*

Gruppi Tra i requisiti di *PariCluster* è presente quello di avere la possibilità di gestire gruppi di *client* oltre che i *client* singoli. Dal punto di vista progettuale, non si evidenziano particolari problemi in quanto l'organizzazione in gruppi è interna a *PariCluster* e può essere gestita mediante semplici liste di client.

3.3.4 Identificazione delle risorse

L'utente di *PariCluster* è consapevole dell'esistenza di Nodi, *PariPari*Cores e Gruppo, ai quali deve potersi riferire per l'uso di gran parte dei comandi. E' quindi necessario proporre un intuitivo meccanismo di identificazione di queste entità.
Si può pensare di adottare una soluzione semplice e immediata: ognuna di queste entità disporrà di un identificativo (ID) univoco durante l'intera sessione di lavoro e tale identificativo individua anche le risorse correlate alle entità, come le working directory dei *client* che assumeranno lo stesso nome dell'identificativo dei *client* stessi.
Si posso usare degli identificativi alfanumerici composti da un prefisso di caratteri seguito da una sequenza di cifre. Il prefisso specifica il tipo di entità, mentre la sequenza di cifre identifica l'entità tra l'insieme di quelle dello stesso tipo. Per i nodi è possibile adottare il prefisso "N", per i *PariPari*Cores (i *client* di *PariPari*) il prefisso "PP", mentre per i gruppi "G".

3.3.5 Layouts

L'ultimo requisito proposto nel secondo capitolo esprime la possibilità di salvare lo stato della rete *PariPari* per poterla ricaricare in una successiva simulazione. Possiamo definire "stato" della simulazione l'insieme delle seguenti informazioni:

- Il numero di *client PariPari* attivi e i relativi identificativi
- I gruppi di client
- Il numero di nodi usati
- Le working directory dei client

Il salvataggio dello stato può essere effettuato semplicemente scrivendo in un file di testo gli identificativi dei *client* attivi, le definizioni dei gruppi creati, il numero dei nodi attivi e il percorso della directory di lavoro dei client.
Lo "stato" della simulazione è chiamato "*layout*" in quanto rappresenta la struttura della rete di client.

3.4 Linguaggio di Programmazione

L'adozione di un particolare linguaggio di programmazione per la scrittura di un programma non è quasi mai scontata e comporta spesso la scelta di avvalersi di alcune caratteristiche a discapito di altre.
Nella scelta del linguaggio è in particolare utile considerare le seguenti caratteristiche:

- Il sistema operativo in cui il programma girerà: i vari linguaggi dispongono infatti di librerie diverse in relazione al sistema operativo di sviluppo ed esecuzione. E' necessario scegliere un linguaggio che garantisca una buona stabilità e che disponga di librerie ben documentate.
- Livello di astrazione: un software che richiede alte prestazioni computative o un controllo più esteso delle funzionalità più vicine all'hardware deve essere sviluppato con un linguaggio che permetta tale livello di controllo.
- Produttività: diversi linguaggi offrono diverse sintassi e organizzazione del codice. In relazione al livello di astrazione del linguaggio, tipicamente i linguaggi più produttivi sono quelli che propongono maggiore astrazione che consente una notevole riduzione della mole di codice da scrivere. Il principale svantaggio di tali linguaggi sono le prestazioni, generalmente inferiori,

talvolta sensibilmente.

Vogliamo ora applicare questi tre concetti al progetto di *PariCluster*.
L'executor per sua natura ben si presta ad essere realizzato come uno script in quanto deve eseguire solo poche istruzioni e rimanere in attesa di un evento. Il manager sarà fondamentalmente un programma che riceve degli input da tastiera e comunica con un numero imprecisato di *client* mediante *socket*. Non è quindi necessaria l'adozione di un linguaggio a basso livello in quanto non sono previste operazioni computazionalmente pesanti e la quasi totalità dei linguaggi ad alto livello offre funzionalità di connessione mediante *socket*.
Il tempo utile di sviluppo del software è previsto di tre mesi, un lasso di tempo sufficientemente lungo per l'adozione di un linguaggio dalla media produttività. In base a tali osservazioni è possibile scegliere fra un vasto ventaglio di opzioni, fra le quali si decide di optare per un linguaggio di scripting con un alto livello di produttività, diffuso e supportato in ambiente Linux.
Il linguaggio scelto per la realizzazione di *PariCluster* è Python.

3.4.1 Python

Python è un linguaggio di programmazione ad alto livello che supporta molteplici paradigmi di programmazione e presenta una elevata leggibilità ed espressività. Python è spesso usato come linguaggio di scripting in quanto è un linguaggio interpretato, ma mediante l'uso di tools esterni è possibile distribuire il codice come programma stand-alone per diverse piattaforme.
Permette al programmatore di adottare una scrittura orientata agli oggetti così come alle funzioni, è possibile creare moduli contenenti funzioni o classi importabili da altri script ed è anche possibile integrare porzioni di codice scritte in linguaggio C qualora fosse necessario. I diversi paradigmi possono essere usati anche all'interno dello stesso software e all'interno dello stesso script.
Esistono diverse implementazioni della virtual machine di Python, l'implementazione ufficiale è CPython ed è scritta in C conforme allo standard C89. Lo script python viene interpretato, riorganizzato e tradotto in forma di bytecode ed infine la Virtual Machine (CPython) legge ed esegue il bytecode.
Python è un linguaggio nato ufficialmente nel 1989 da una idea di Guido Van Rossum, e fino ad oggi ha subito tre maggiori update portandolo attualmente alla versione 3, versione adottata anche per lo sviluppo di *PariCluster*.

4 Implementazione

In questo quarto capitolo verranno discussi gli aspetti relativi alla fase di scrittura del codice e presentate alcune problematiche strettamente legate al linguaggio di programmazione scelto.

4.1 Client Side

La componente *client* del programma *PariCluster*, consta di un unico script che, avviato dal modulo di controllo attraverso un comando srun di *Slurm*, lancia e configura un *client PariPari*.
Nel complesso questo script, contenuto nel file `executor.py`, è uno script piuttosto breve il cui funzionamento è descrivibile nella seguente sequenza di operazioni:

1. Lo script viene lanciato specificando dei parametri di configurazione quali:
   - il nome associato all'istanza, necessario per l'identificazione dei messaggi in risposta al manager
   - la porta su cui il manager è in ascolto
   - l'host su cui il manager è in ascolto
   - la prima porta che il *client* deve tentare di usare
   - il percorso dove si trova il Core di *PariPari*
   - il percorso dove creare la copia di lavoro del client
2. Viene aperto un *socket* verso il manager usando i dati ricevuti come parametri, se la connessione fallisce, lo script si interrompe in quanto il manager risulta non raggiungibile.
3. Lo script avvia come subprocess il *client* di *PariPari* usando le impostazioni ricevute dal manager.
4. L'output del *client* viene letto e da esso ricavata l'effettiva porta aperta dal client.
5. L'avvio del *client* e la relativa porta utilizzata vengono notificate al manager e il *socket* viene chiuso.

6. Lo script si mette in attesa della terminazione del processo del client.

7. Quando il *client* termina, lo script prosegue aprendo un nuovo *socket* verso il manager per notificare l'avvenuta chiusura del processo.

8. Lo script Executor chiude il *socket* e termina.

Nelle prime versioni dello script era presente anche una funzione di log interna che tuttavia è stata rimossa in seguito, in quanto si è preferito semplicemente stampare in output il log che di default viene catturato e salvato su file da *Slurm*. Una ulteriore funzione di log risultava superflua e lo script è stato quindi semplificato lasciando unicamente a *Slurm* tale compito.

4.2 Server Side

La componente server di *PariCluster* è costituita da un insieme di script che gestiscono i vari aspetti della simulazione. Una volta avviato, lo script di ingresso al programma esegue una routine di setup dell'ambiente terminata la quale sarà pronto a ricevere i comandi dell'utente.

4.2.1 Session Setup

Allo script contenente il punto di ingresso del programma è stato dato il nome di `PariCluster.py`.
Essendo il primo script ad essere eseguito, dovrà provvedere al primo setup dell'ambiente di lavoro ed eseguirà dei test per verificare lo stato del cluster.
Brevemente è possibile riassumere il setup nei seguenti step:

- Vengono importati i moduli della libreria di Python necessari e i moduli di servizio del programma

- Vengono letti i file di stringhe per caricare in memoria i messaggi testuali da usare nel ciclo di vita dell'applicazione

- Legge e interpreta i parametri eventualmente forniti all'avvio

- Permette la conferma o la modifica della configurazione

- Crea le directory di lavoro

- Apre il *socket* in ascolto per ricevere le notifiche dagli script Executor

- Controlla lo stato di *Slurm*

4.2.2 Configurazione

La configurazione di un programma è data dall'insieme di valori e parametri che ne definiscono almeno in parte il comportamento.
PariCluster memorizza queste impostazioni in un file di testo nella cartella principale del programma. Nel file di configurazione sono per esempio indicati il nome della partizione *Slurm* da usare, il percorso di salvataggio delle cartelle di lavoro, il percorso in cui si trova il *client PariPari*, la verbosità.
All'avvio dello script all'utente viene proposta la configurazione attualmente salvata e si chiede se confermarla o modificarla. Come emerso in fase di testing, l'utente può trovare superfluo dover confermare ogni volta la stessa configurazione per cui è stata aggiunta la possibilità di saltare questa fase specificando il parametro `{noconf` all'avvio dello script.
Quando la configurazione, subite le eventuali modifiche, è confermata, le impostazioni che sono necessarie a tutti gli script del programma sono copiate in un modulo dedicato, chiamato shared.
Il modulo che si occupa della modifica del file di configurazione è separato e il codice è contenuto nel file `configure.py`, mentre il modulo in cui sono definite le variabili globali condivise è definito nel file `shared.py`.

4.2.3 Gestione delle stringhe

Durante la scrittura di un programma, uno degli inconvenienti che più spesso si presentano è l'interferenza tra il codice di controllo, gestione e di interfacciamento. Un design pattern che regola questo comportamento è il cosiddetto "Model-View-Controller" che suggerisce la separazione netta tra i tre blocchi di codice. Tuttavia, non esistendo una interfaccia grafica complessa, questo schema non è stato formalmente adottato.
Mentre infatti il codice di gestione del cluster si trova in uno script separato, le funzioni di interfacciamento sono in parte integrate nel codice che controlla il flusso del programma.
Man mano che il programma veniva scritto è risultato evidente come un livello di verbosità sufficientemente comprensibile richiedesse una gestione più completa di qualche comando di output a console, e che l'inserimento di questi comandi all'interno del codice di controllo sarebbe risultato pesante dal punto di vista della leggibilità e manutenibilità del programma.
Su esempio delle modalità di output a console adottate nel progetto *PariPari*, si è pensato di realizzare, in uno script a parte, alcune funzioni di log che permettessero di lavorare a più livelli di verbosità, ovvero di "visibilità" dell'output, in modo da

unire la stampa di log di debug su file a quella di output a video con l'uso di una singola funzione.
Le principale funzione di log che è stata sviluppata permette di specificare un messaggio e un livello di verbosità dello stesso. In base al livello specificato e a quello minimo richiesto per la stampa a video (indicato dall'utente), il messaggio sarà o meno visualizzato nella console con un colore relativo al tipo del messaggio. In tutti i casi, il messaggio è comunque scritto nel file di log del programma. Allo stato attuale sono stati definiti i seguenti tipi di messaggio:

- User message: sempre visualizzato a schermo, di colore verde
- Log message: visualizzato a schermo se l'utente ha settato verbosità maggiore di 0, di colore azzurro
- Debug message: visualizzato a schermo se l'utente ha settato verbosità maggiore di 1, di colore blu
- Warning message: sempre visualizzato, di colore giallo
- Error message: sempre visualizzato, di colore rosso
- Hidden message: mai visualizzato, solo log su file

Questa gestione aggiunge una notevole espressività all'output del programma verso l'utente, ma non ha ancora portato sufficienti vantaggi al programmatore. Infatti il più grande inconveniente nell'avere istruzioni di output insieme a istruzioni di controllo del flusso del programma è lo spazio che i messaggi di testo vanno a occupare nelle righe di codice.
Il rischio è quello di scrivere un codice inutilmente lungo e di difficile lettura. Inoltre, nel caso in cui risultasse necessario apportare delle modifiche ai testi, sarebbe necessario cercare la riga e correggere il testo che a sua volta potrebbe essere un messaggio usato più volte in diverse parti del codice rendendo la modifica frustrante.
Una soluzione a questo problema viene dall'adozione di un file di testo a parte nel quale ogni riga, identificata da un codice, rappresenta un messaggio. Durante la fase di setup, il programma carica le stringhe in un array associativo così da poter essere utilizzate semplicemente specificandone il codice. La stessa stringa, per esempio un messaggio di errore, usata in più parti del codice, può così essere modificata agendo solo sulla corrispondente riga nel file di testo.

4.2.4 Directory di Lavoro

Nel capitolo 3 si è discusso sulla condivisione nel cluster della cartella home evidenziando come il contenuto di questa sia accessibile da ogni nodo fisico del cluster, e quindi anche da ogni nodo logico interno di *PariCluster*.
Questo dettaglio è di fondamentale importanza in quanto permette di avere una gestione dei file su disco totalmente trasparente dalla struttura del cluster. Oltre a quelle che contengono gli script, *PariCluster* dovrà accedere ad altre due cartelle:

- la cartella dove si trova il Core di *PariPari*
- la cartella dove vengono salvate le working directory dei client
- la cartella dove vengono raccolti i log della sessione

In particolare è necessario, ad ogni avvio del programma, creare una cartella che raccolga di dati dei *client* nella sessione di lavoro e una cartella per i log, attribuendo loro un nome univoco. Tale nome potrebbe per esempio essere un timestamp.
Tuttavia un timestamp risulta poco espressivo se non convertito nella corrispondete data e ora, quindi si è deciso di usare una sequenza composta nell'ordine da: anno, mese, giorno, ora, minuti e secondi.
La cartella di lavoro di *PariCluster*, così come quella di log, viene creata in fase di setup all'interno della cartella specificata nella configurazione. All'interno di essa i *client PariPari* depositeranno le proprie working directory con nome corrispondente all'identificativo loro assegnato.

4.2.5 Problema di sincronizzazione tra user input e socket

In queste prime versioni di *PariCluster* l'unica modalità disponibile all'utente per impartire comandi è una semplice riga di comando testuale. Tipicamente il programma, nel suo ciclo di vita, dovrà ricevere un numero indefinito di comandi da parte dell'utente per cui la lettura dell'input da tastiera sarà sicuramente un ciclo indefinito.
Il primo approccio che si è tentato è stato quello di leggere l'input da tastiera con un comando di readline, per poi verificare a quale comando corrispondesse attraverso una serie di costrutti condizionali if.
Si noti che in Python non esiste il costrutto *switch*, per cui si usa di norma una catena di *if-elif*. L'input da tastiera da parte dell'utente non è però l'unico input che giunge al modulo manager di *PariCluster*. Infatti, come abbiamo visto in fase di progettazione, è prevista anche la ricezione di dati da parte degli script executor per notificare lo stato dei *client* e comunicare le porte di comunicazione

per il controllo degli stessi. Tali comunicazioni avvengono via *socket* e devono essere considerate di importanza uguale a quella degli input da tastiera.
Lo stato dei *client* è infatti una informazione importante anche per l'utente e quest'ultimo sicuramente è interessato ad esserne aggiornato.
Si tratta quindi di gestire due input:

- Stream *stdin* da tastiera per gli user input
- Buffer in ricezione del *socket* per gli aggiornamenti dagli executor

Il problema che si è subito presentato in fase di test del codice è che l'uso di una lettura bloccante come quella prevista da `readline` bloccava lo script e il buffer del *socket* non veniva letto.
Anche inserendo la lettura del *socket* all'interno del ciclo di readline, il *socket* sarebbe comunque letto solo ad ogni newline da tastiera. Una idea iniziale è stata quella di usare un approccio multithread separando l'acquisizione da tastiera dalla lettura dei *socket*. Rimaneva tuttavia il problema nell'unire queste due componenti in modo da fornire all'utente il quadro completo dello stato della simulazione pur rimanendo sempre pronti a ricevere l'input.
Una possibilità che offre Python in queste circostanze è l'uso dei segnali, comparabili agli eventi in altri linguaggi come Java. Tuttavia l'unione dell'architettura multithread all'uso dei segnali si è fin da subito rivelata non così semplice e immediata e si è prima voluto provare altre soluzioni. Una alternativa la si è infatti trovata con l'uso della funzione `select` di Python che permette l'accesso alla omonima funzione disponibile in molti sistemi operativi UNIX. Il comportamento di questa funzione è molto semplice e potente, permette infatti, data una lista di *streams*, di effettuare un poll di controllo identificando gli stream pronti ad essere letti.
Nel caso in esame questa funzione è sembrata fin da subito molto interessante e anche in fase di implementazione, dopo alcuni test, è risultata effettivamente efficace e immediata.
Creato quindi un ciclo indefinito, al suo interno si è inserito il comando `select` che è bloccante e di seguito la gestione degli input da *socket* e da tastiera. Quando l'utente inserisce un comando e preme il tasto invio (carattere newline), lo stream *stdin* sarà pronto per essere letto e il `select` ritorna una lista contenente lo stream. Nelle successive righe di codice viene verificato se effettivamente lo stream di input è leggibile, in caso affermativo viene letta e interpretata la prima riga in esso.
Più complesso il caso in cui è il *socket* in ascolto sul manager a risultare leggibile. Il *socket* in ascolto risulta "leggibile" quando riceve una nuova connessione, in questo caso la connessione viene accettata e lo stream relativo ad essa viene immediatamente aggiunto alla lista di input del `select`. Alla successiva iterazione del ciclo

di lettura, il `select` estenderà il poll di controllo anche alla singola connessione e uno stato leggibile di questa, corrisponderà ai dati trasmessi dall'altro capo della connessione, nel nostro caso l'executor. Il manager legge quindi i dati in arrivo e dopo averne verificato la provenienza dallo script esecutore, effettua le operazioni necessarie, siano esse l'aggiunta o la rimozione di un nuovo *client* alla lista di gestione.
Quando l'executor chiude la connessione, il `select` rileverà la connessione relativa nuovamente come leggibile. Tuttavia in questo caso i dati letti corrispondono a un valore nullo, a significare che la connessione si è effettivamente chiusa ed è possibile rimuoverla dalla lista di polling.

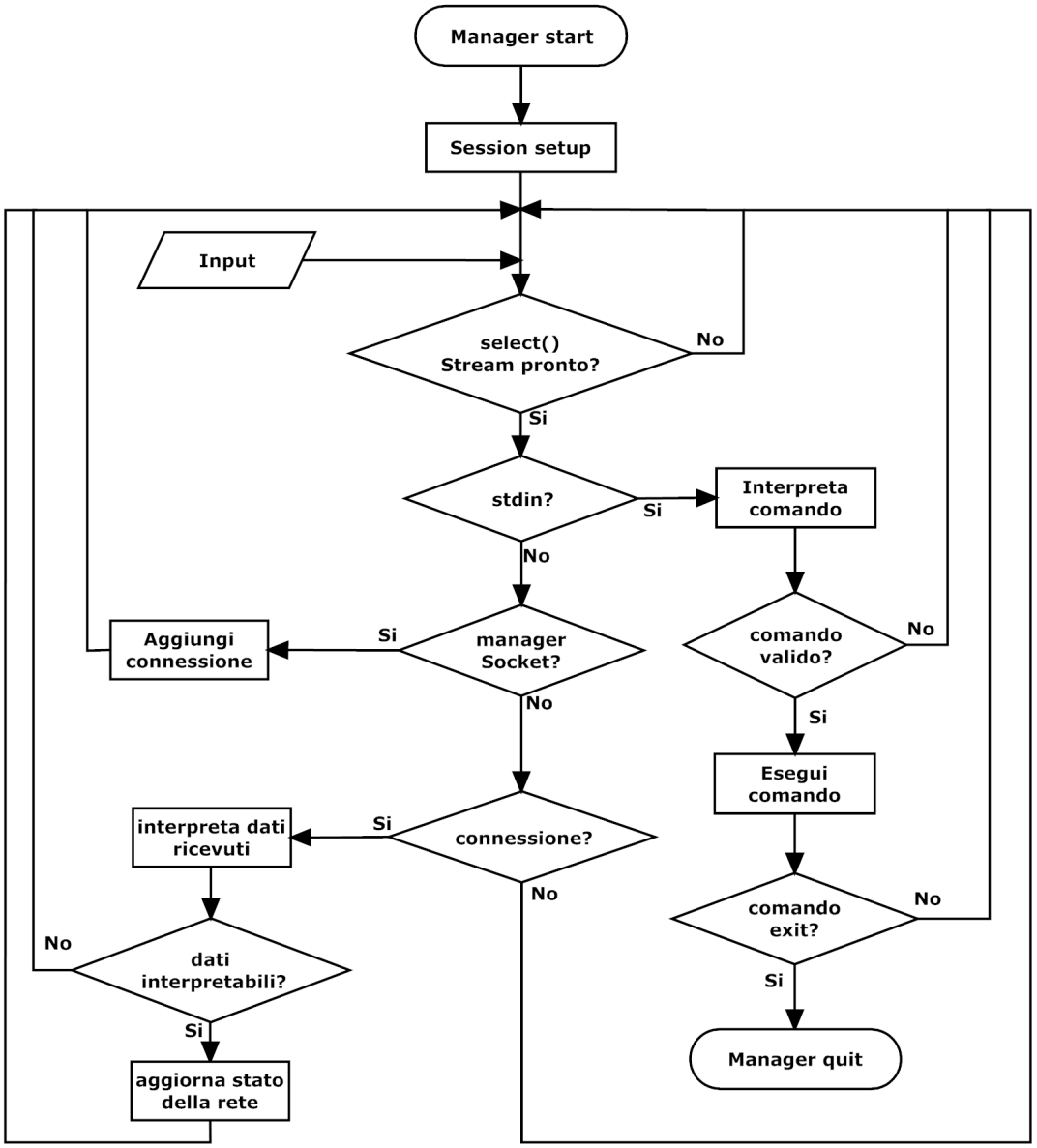

Figure 4.1: Gestione degli input da *stdin* e *socket* con `select`

L'intero ciclo di lettura degli input da tastiera e da *socket* e la relativa interpretazione, costituisce quello che viene definito "main cycle" del programma ed è implementato nel file `PariCluster.py`, successivamente alle istruzioni di setup. La condizione di uscita dal ciclo è la lettura del comando di terminazione del programma. Una volta uscito dal ciclo, lo script termina.

4.2.6 Comandi

Ogni stringa inserita a tastiera da parte dell'utente è considerata un comando. Molti comandi possono essere impartiti specificando dei parametri aggiuntivi, per cui non solo il comando, ma anche i parametri dovranno essere interpretati. Per rendere più ordinato e pulito il codice di interpretazione, sono state scritte delle funzioni di parsing che delocalizzano parti ripetitive della procedura. In particolare, per i comandi utente, sono state scritte due funzioni il cui codice è contenuto nel modulo `utils.py`.

- `splitStrip` usata per dividere la stringa in token separati da spazi ed eliminando gli spazi iniziali e finali di ogni token. Il primo token sarà quindi il comando, i successivi saranno gli eventuali parametri
- `parseCommandOptions` usata per riorganizzare i parametri del comando in un array associativo di facile accesso.

Se il comando è riconosciuto e gli eventuali parametri sono stati correttamente inseriti, lo script esegue le operazioni richieste, altrimenti sarà comunicato un errore in cui viene specificato che il comando non è valido o che il parametro è sconosciuto o il valore relativo non valido. L'interpretazione dei comandi avviene nel main cycle e quindi il codice relativo è contenuto nel file `PariCluster.py`.

Nella versione attuale i comandi disponibili all'utente sono:

- `Help`: visualizza una pagina di help con le regole d'uso e una lista dei comandi
- `Ver`: visualizza la versione di *PariCluster*
- `Nodeadd`: Aggiunge uno più nodi alla simulazione
- `Start`: Avvia uno o più *client PariPari*
- `Stop`: Termina uno o più *client PariPari*
- `Groupadd`: aggiunge un nuovo gruppo
- `Groupdel`: elimina un gruppo
- `Groupmod`: modifica un gruppo
- `Clean`: rimuove dai gruppi i *client* terminati

- `Nodes`: visualizza la lista dei nodi attivi
- `Pps`: visualizza la lista dei *client* attivi
- `Groups`: visualizza la lista dei gruppi definiti
- `Savelayout`: salva il *layout* corrente
- `Loadlayout`: carica un *layout* precendente
- `Quit`: termina *PariCluster*

4.2.7 Identificativi

Nel terzo capitolo sono state presentate le entità Nodo, *PariPari*Core e Gruppo e si è evidenziato come sia necessario un sistema di identificazione di esse. Si è deciso di usare degli identificativi alfanumerici composti da un prefisso di caratteri seguito da una sequenza di cifre.
L'utente può specificare solo la parte numerica dell'identificativo in fase di creazione dell'entità, ma il prefisso è imposto dal programma. Naturalmente gli identificativi sono univoci.
Lo stesso ID può essere espresso in due diversi modi:

- Full Qualified ID: è l'identificativo completo di prefisso
- Non-Full Qualified ID (o "ID breve"): è l'identificativo privo di prefisso

Per alcuni comandi in cui è sottinteso il tipo di entità a cui ci si riferisce, l'utente può usare l'ID nella forma breve, negli altri casi è necessario usare l'id completo. E' possibile specificare anche sequenze di ID separati da virgola, e blocchi contigui di essi nella forma "prefisso[A-B]" dove A e B sono i limiti del blocco. Alcune funzioni di utilità sono state create per la gestione degli ID nella forma breve o completa e per la gestione di sequenze di ID.
Le relative funzioni più usate nel codice sono:

- `areValidIds (names, tags)`: verifica che l'array di ID fornito sia valido
- `parseMultipleIds (text, tags)`: data la stringa relativa alla sequenza e i prefissi attesi, restituisce una lista dei singoli identificativi. Permette la traduzione dei gruppi nella lista dei corrispondenti PPCores e l'eliminazione dei duplicati.
- `addPrefix (ids, prefix)`: aggiunge un prefisso agli id brevi

- `remPrefix (ids, prefix)`: rimuove un prefisso agli id completi

4.2.8 Gestione Nodi e *PariPari*Cores

La gestione del cluster avviene mediante *Slurm*, per cui la creazione (allocazione) dei nodi e l'avvio dei *client* di *PariPari* avverrà richiamando comandi del framework di *Slurm*.
Quando l'utente richiede la creazione di un nodo mediante il comando nodeadd, lo script, dopo aver interpretato anche gli eventuali parametri che specificano il numero di nodi da aprire e il nome da assegnare loro, chiama la funzione newNode definita nello script `clusterManager.py`.
L'iter che la funzione è il seguente per ognuno dei nodi che devono essere avviati:

- viene aperto un sottoprocesso *bash* mediante la funzione `os.Popen` di Python specificando il comando `salloc`
- si legge l'output del processo per verificare se il *job* viene creato
- se il *job* è avviato, una nuova voce è inserita nel dizionario *ppNodes* nel quale la chiave è l'identificativo del nodo e il valore è un array associativo (dizionario) contenente l'id del *job* , il riferimento al processo relativo al *job* e una eventuale descrizione del nodo.

Un fatto anomalo, che ha determinato qualche difficoltà in quanto inatteso, è che l'output del comando salloc non è leggibile nello stream stdout del processo come ci si può aspettare, bensì nello stream stderr di errore.
Per quanto riguarda i *PariPariCores*, il comando per avviare nuove istanze di *PariPari* è `start` e permette l'avvio di una o più *client* per volta e permette anche l'assegnamento degli ID, che altrimenti sono attribuiti secondo un ordine crescente.
La funzione che esegue l'avvio del nodo è newPPCore e anch'essa è contenuta nello script`clusterManager.py`.
Per ogni *client* da avviare, la funzione:

- recupera il processo relativo al nodo su cui il *client* verrà avviato
- nello stream di input di tale processo viene scritto il comando srun che avvia l'executor fornendogli tutti i parametri di configurazione come il nome dell'istanza e la cartella di lavoro da usare
- inserisce l'ID del *client* in una lista contenente gli ID dei *client* cui è stato richiesto l'avvio

Se l'avvio ha successo, l'executor comunicherà al server di *PariCluster* lo stato online del *client* e l'id relativo verrà prima rimosso dalla lista degli ID in attesa di conferma, e poi aggiunto alla lista dei *client* attivi assieme all'indirizzo della macchina e alla porta su cui è in ascolto.
Il controllo di un *client* avviene inviando comandi testuali alla porta su cui è in ascolto. In *PariCluster*, tale comunicazione è possibile indicando una lista, anche mista, di identificativi di *PariPariCore* o di *Gruppi*, seguita dal comando da impartire.
Per ognuno dei *PariPariCores* indicati viene aperta una connessione usando le informazioni contenute nella lista dei *client* attivi. Il messaggio viene quindi spedito e la connessione chiusa.

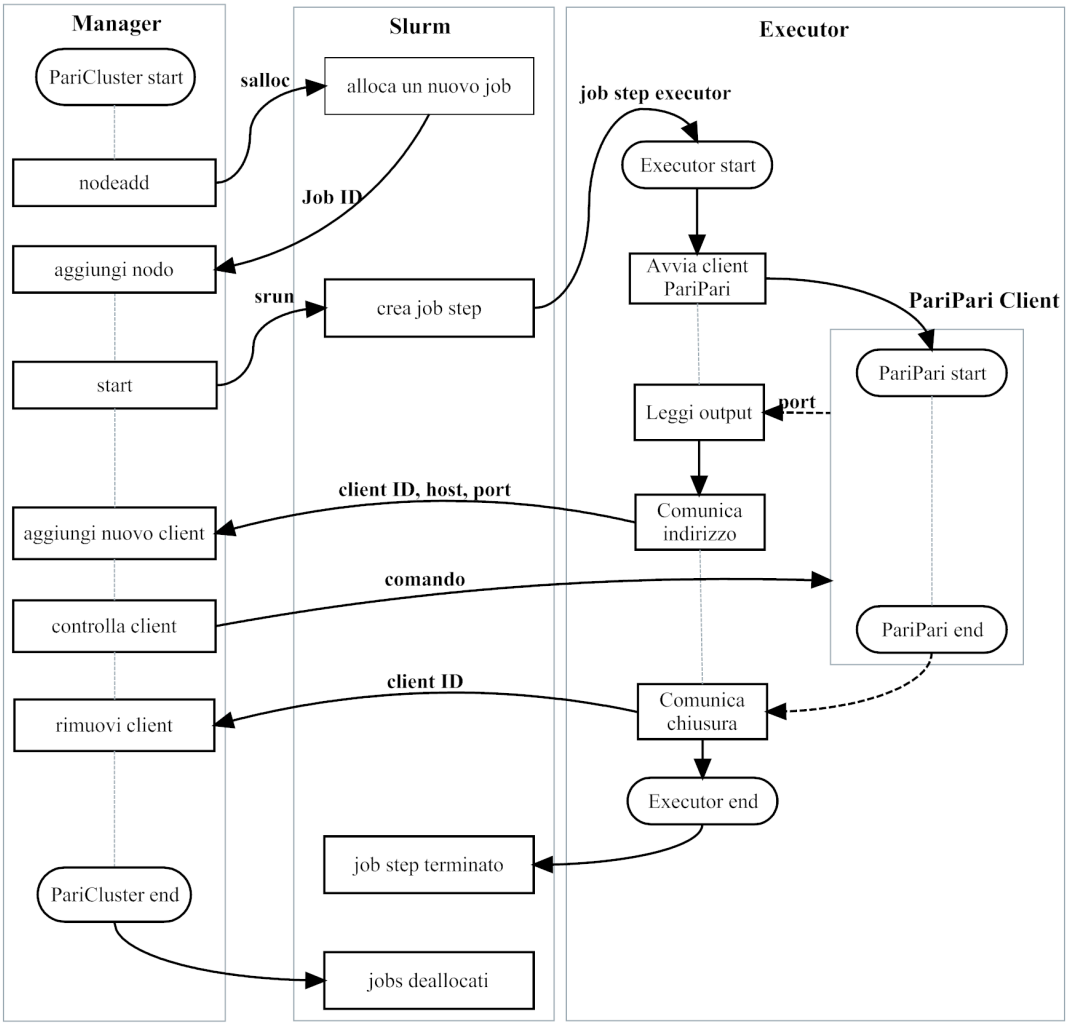

Figure 4.2: Gestione del cluster con Nodi e *PariPari*Cores

4.2.9 Salvataggio e caricamento dei Layout

I comandi per salvare e caricare uno stato della simulazione sono `savelayout` e `loadlayout`. Il salvataggio del *layout*, come analizzato in fase di progettazione, prevede la scrittura su file delle informazioni necessarie per la successiva ricostruzione dell'ambiente di lavoro.
Quando il comando `savelayout` viene invocato, seguito dal un nome identificativo, *PariCluster* apre un file di testo nella cartella dove risiedono i *layout* e al suo interno scrive:

- la lista degli identificativi dei *PariPariCores*
- la lista degli identificativi dei *Gruppi*
- la lista degli identificativi dei *Nodi*
- la directory di lavoro della sessione corrente

Questi dati sono sufficienti per il successivo ripristino in quanto la struttura logica della simulazione è data dagli identificativi, mentre la corrispondenza tra *client* e stato del *client* è assicurata dal fatto che ogni *client* che viene aperto in fase di caricamento del *layout*, utilizzerà la stessa working directory che utilizzava l'istanza del *client* da cui ha ereditato il nome.
Infatti *PariPari* separa il codice del core dai file che caratterizzano la particolare istanza (plugin scaricati e relativi file di configurazione, log...) memorizzandoli appunto nella working directory.
Il comando di `loadlayout`, quindi, legge il file *layout*, imposta la directory di lavoro della simulazione salvata come directory corrente e avvia lo stesso numero di nodi e *client* con gli stessi identificativi. Vengono inoltre ridefiniti i gruppi.
In fase di caricamento, il programma avvia un thread che esegue l'effettivo caricamento, proponendo una interfaccia più interattiva per permettere all'utente di seguire passo per passo le fasi di caricamento del *layout*, offrendo un riscontro visivo dell'avanzamento dell'operazione, che può durare diversi secondi.

5 Stato dell'arte e conclusioni

In questo ultimo capitolo si riassume lo sviluppo del progetto e partendo dallo stato attuale di questo, si presentano delle ipotesi di sviluppo futuro.

5.1 Stato dell'arte

Allo stato attuale una copia della prima versione di *PariCluster* è presente nel cluster Eridano ed è disponibile per attività di testing e report da parte dei programmatori di *PariPari*.
Nella versione corrente *PariCluster* mette a disposizione tutti i comandi per le funzionalità fondamentali che erano previste rispondendo quindi ai requisiti di partenza.
Eventuali bug o richieste sono gestite attraverso l'uso di Redmine, una piattaforma di gestione e bug-tracking usata anche per il progetto *PariPari*, e dalla prima versione rilasciata per la fase di test molti bug sono stati corretti e diverse funzionalità sono state aggiunte, tra cui:

- Reingegnerizzazione e ottimizzazione di funzioni di gestione del cluster
- Aggiunta di funzionalità ad alcuni comandi
- Introdotto un sistema di help per fornire documentazione dei singoli comandi

5.2 Conclusioni e Sviluppo Futuro

Nonostante *PariCluster* abbia raggiunto un sufficiente livello di stabilità e di funzionalità, lo sviluppo non si considera concluso.
Alcune richieste che sono state espresse nell'ultima fase di collaudo non hanno ancora trovato risposta nella loro realizzazione, e probabilmente il progetto subirà in futuro altri aggiornamenti e correzioni, e forse anche delle estensioni. Si pensa per esempio alla realizzazione di una GUI grafica che permetta funzioni come l'autocompletamento e lo storico dei comandi inseriti in precedenza, funzionalità che con l'implementazione attuale non sono di facile integrazione in quanto in conflitto con la gestione degli stream del comando select.
La scelta di non tentare uno sviluppo di queste funzioni allo stato attuale è dovuta sostanzialmente alla scadenza dei termini utili previsti per la realizzazione e la necessità di fornire un prodotto stabile ed efficiente nelle funzioni che offre,

piuttosto che un software non affidabile.
Inoltre è importante ricordare come lo sviluppo di *PariCluster* sia stato condizionato dalla poca esperienza in ambito cluster e di programmazione con il linguaggio Python dello sviluppatore, si prevede infatti una completa attività di reingegnerizzazione.
In questo elaborato si è voluto presentare l'evoluzione di *PariCluster* nelle varie fasi di realizzazione, dall'analisi dei requisiti, passando per la progettazione e arrivando all'implementazione vera e propria.
La trattazione non è sicuramente completa e non è sufficiente per esprimere settimane di prove, simulazioni, correzioni e riscritture di intere porzioni di codice, rese frequenti anche per l'adozione di un linguaggio di programmazione per me nuovo.
Si sono voluti portare alla luce quegli aspetti che più hanno segnato lo sviluppo del progetto, i problemi che man mano si sono presentati e che di volta in volta si sono superati o evitati, in modi sicuramente non sempre eleganti dal punto di vista della programmazione, ma adottati nell'ottica di fornire un software sufficientemente affidabile.

Riferimenti

JUnit
http://junit.org

Cluster: definizioni e architetture
http://www.ibm.com/
"Beowulf – HOWTO" di Jacek Radajewski and Douglas Eadline, 1998

SLURM: uso e configurazione
https://computing.llnl.gov/linux/slurm/

Python: API reference
http://www.python.org

nohup: comando per la non propagazione di SIGHUP
http://en.wikipedia.org/wiki/Nohup