Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Ottimizzazione del deployment di architetture a microservizi utilizzando programmazione a vincoli

Relatore:
Chiar.mo Prof.
Roberto Amadini

Presentata da:
Giovanni Spadaccini

Corelatori:
Chiar.mo Prof.
Gianluigi Zavattaro,
Dott.
Simone Gazza

I Sessione
Anno Accademico 2023/2024

Indice

1 Introduzione **1**
1.1 Progetto FREEDA 1
1.2 Obiettivi della Tesi 2

2 Background **3**
2.1 Programmazione lineare e Programmazione a Vincoli 3
2.2 Strumenti utilizzati 4

3 Struttura Applicazione e Infrastruttura **5**
3.0.1 Risorse 5
3.1 Applicazione 6
3.1.1 Componente 7
3.1.2 Requisiti 7
3.1.3 Budget 8
3.2 Infrastruttura 8
3.2.1 Nodi 8
3.2.2 Links 8

4 Linguaggio Intermedio **11**

5 Modellizzazione del problema in programmazione a vincoli **13**
5.1 Modellizzazione 13
5.1.1 Variabili 13
5.1.2 Vincoli 13
5.1.3 Requisiti dei componenti 14
5.1.4 Obiettivo 16
5.2 Modifiche Apportate 17
5.2.1 Variabili 17
5.2.2 Vincoli 17
5.2.3 Obbiettivo 21

6 Implementazione **23**
6.1 Implementazione del Parser 23
6.2 Differenze nella modellazione 24
6.3 Benchmark 24

7 Conclusioni **27**
7.1 Risultati raggiunti 27
7.2 Sviluppi futuri 27

A Esempi Yaml dei file di applicazione e infrastruttura **I**
A.1 Esempio Risorse I
A.1.1 Esempio completo Applicazione II
A.2 Esempio Componente III
A.2.1 Dei Requirement V
A.2.2 Esempio Completo Infrastruttura VII

B Versione alternativa della funzione obbiettivo **IX**

Capitolo 1

Introduzione

L'adozione diffusa di *smart devices* connessi e la crescente potenza computazionale di questi dispositivi stanno portando a un'evoluzione del Cloud computing verso ambienti distribuiti. Questi nuovi ambienti sfruttano le capacità computazionali ai margini della rete. Allo stesso tempo, le applicazioni basate su microservizi (MSA) stanno diventando sempre più comuni nelle applicazioni aziendali, richiedendo supporto per il loro deployment su infrastrutture Cloud-IoT.

Il deployment dei servizi che compongono una MSA su infrastrutture Cloud-IoT deve essere pianificato in modo globale, tenendo conto dei diversi requisiti di deployment dei servizi stessi. Questi requisiti includono la complessità delle MSA e delle infrastrutture Cloud-IoT, i cui servizi e nodi possono fallire per vari motivi, rendendo necessaria l'implementazione di meccanismi di resilienza ai fallimenti nelle MSA distribuite. Inoltre, tali requisiti devono essere allineati con le direttive dell'UE per un'IT sostenibile, che includono la riduzione del consumo energetico.

La combinazione di tutti questi requisiti rende difficile configurare il deployment di MSA su infrastrutture Cloud-IoT. I requisiti di deployment possono essere in conflitto tra loro. Ad esempio, aumentare la resilienza replicando le istanze dei servizi può incrementare il consumo energetico. Questo richiede nuove tecniche e strumenti che aiutino gli ingegneri DevOps a gestire i requisiti di deployment.

1.1 Progetto FREEDA

Il progetto FREEDA [VSA+24] ha lo scopo di permettere il deployment integrato di una MSA su un'infrastruttura Cloud-IoT, bilanciando i suoi requisiti di deployment, che possono essere anche in conflitto. FREEDA consente di analizzare una MSA e un'infrastruttura Cloud-IoT, insieme alle informazioni sui deployment attuali e precedenti della MSA, per identificare ulteriori requisiti che garantiscano la resilienza ai fallimenti e la sostenibilità energetica del deployment della MSA. Inoltre, FREEDA ottimizza il deployment della MSA utilizzando un risolutore di

vincoli, bilanciando adeguatamente i requisiti di deployment, specialmente quando questi sono in conflitto.

FREEDA mira quindi a proporre tecniche e strumenti per abilitare il deployment, resiliente ai fallimenti ed energeticamente sostenibile di una MSA su un'infrastruttura Cloud-IoT esistente, ottimizzando i compromessi tra i requisiti di deployment spesso conflittuali.

1.2 Obiettivi della Tesi

Il proposito della presente tesi è la creazione di un parser che possa generare, a partire dai file di configurazione del deployment, in più formati utilizzabili da solver di programmazione a vincoli o programmazione lineare.

Utilizzando poi il solver appropriato sul file di output si troverà la soluzione ottimale per il deployment della MSA su un'infrastruttura Cloud-IoT, bilanciando i requisiti di deployment dei microservizi e tenendo conto dei vincoli di risorse, delle dipendenze tra i servizi e dei requisiti di resilienza e sostenibilità energetica.

Capitolo 2

Background

In questo capitolo verranno delineate le basi della programmazione a vincoli utilizzate nella presente tesi. Si fornirà una panoramica sul paradigma della programmazione a vincoli, sulle sue varianti principali, e sugli strumenti e linguaggi utilizzati per modellare e risolvere i problemi.

2.1 Programmazione lineare e Programmazione a Vincoli

Sia la programmazione a vincoli che la programmazione lineare sono dei paradigmi di programmazione che permettono di modellare e risolvere problemi complessi in modo dichiarativo. Invece di specificare il processo per risolvere un problema, il programmatore definisce i vincoli che devono essere soddisfatti. Sarà compito del risolutore trovare una soluzione che rispetti tutti i vincoli definiti. Questo approccio consente di separare la definizione del problema dalla sua risoluzione, permettendo di concentrarsi maggiormente sulle specifiche del problema stesso.

La Programmazione Lineare (PL) è una branca fondamentale della ricerca operativa che si occupa dell'ottimizzazione di una funzione obiettivo lineare soggetta a vincoli anch'essi lineari. In questo contesto, un problema viene modellato utilizzando variabili continue, che possono assumere qualsiasi valore all'interno di un intervallo definito. La funzione obiettivo è tipicamente una somma pesata delle variabili, e i vincoli sono espressi come disuguaglianze o uguaglianze lineari.

La Programmazione Lineare Intera (ILP, dall'inglese Integer Linear Programming) estende il concetto di PL introducendo la restrizione che alcune o tutte le variabili devono assumere valori interi. Questa caratteristica rende l'ILP particolarmente adatta a problemi dove le decisioni devono essere prese in unità discrete, come ad esempio nella pianificazione della produzione, nella gestione delle risorse e nella logistica.

La programmazione non lineare a vincoli (NLP) è una tecnica utilizzata per affrontare problemi di ottimizzazione in cui sia la funzione obiettivo che i vincoli presentano termini non lineari. A differenza della programmazione lineare, dove le relazioni tra le variabili sono rappresentate da equazioni lineari, nella programmazione non lineare queste relazioni possono essere espresse da equazioni o disuguaglianze più complesse, includendo polinomi, esponenziali, logaritmi, e altre forme non lineari.

2.2 Strumenti utilizzati

Esistono varie implementazioni di strumenti per la programmazione a vincoli, che permettono di esprimere e risolvere questi problemi. Alcuni strumenti che verranno utilizzati in questa tesi sono:

- **MiniZinc** [MSKS14]: un linguaggio di modellazione che permette di definire problemi di programmazione a vincoli in modo conciso e leggibile. MiniZinc è utilizzato insieme a vari risolutori per trovare soluzioni ottimali.
- **Z3** [DMB08]: un solver SMT (Satisfiability Modulo Theories) sviluppato da Microsoft Research, che supporta diversi tipi di problemi di programmazione a vincoli, inclusi quelli lineari e non lineari. Offre molteplici interfacce tra cui una libreria Python.
- **MPS** [Wik24b]: è un formato di file che permette di memorizzare problemi scritti in programmazione lineare.
- **SMT-LIB** [C+11]: è un formato di file standard per la rappresentazione di problemi Satisfiability Modulo Theories (SMT).

Capitolo 3

Struttura Applicazione e Infrastruttura

Nel progetto FREEDA si è deciso di dividere i dati in due gruppi distinti, i dati relativi all'Applicazione e quelli relativi all'Infrastruttura:

- **Applicazione**: i dati definiti all'interno di questo gruppo hanno il compito di definire tutti i servizi che compongono l'applicazione e le dipendenze tra di essi. Ogni servizio è descritto con dettagliate specifiche tecniche, che includono i requisiti di risorse, e le dipendenze da altri servizi. In questo modo, è possibile avere una visione completa della struttura dell'applicazione, facilitando la gestione e la manutenzione del sistema.
- **Infrastruttura**: i dati al suo interno definiscono tutti i nodi su cui i servizi possono essere distribuiti, fornendo dettagliate informazioni su ciascun nodo. Queste informazioni includono le risorse hardware disponibili, come CPU, memoria e storage, le configurazioni di rete come la latenza o servizi implementati come SSL o storage criptato. Inoltre, il file descrive le relazioni e i collegamenti tra i vari nodi.

Questa suddivisione in due gruppi vengono definiti in due file distinti permettendo una gestione modulare del sistema, facilitando eventuali aggiornamenti o modifiche future. L'uso di una struttura ben definita in nel formato YAML [1], offre sia possibilità di generare e elaborare questo file in automatico che una facile comprensione e scrittura da parte di un umano.

3.0.1 Risorse

Le risorse sono dei parametri che servono sia nei componenti che nell'infrastruttura. Esse rappresentano requisiti che definiscono i componenti e che devono essere rispettati dai nodi nei quali verranno disposti e dai loro collegamenti.

Le risorse possono essere divise in due gruppi principali: consumabili e non consumabili.

[1] YAML è un linguaggio facile da comprendere anche per gli umani, per serializzare dati. YAML

Risorse Consumabili

Le risorse consumabili sono quelle che devono essere aggregate per tutti i componenti in un nodo. Queste includono:

- `cpu`: numero di core.
- `ram`: quantità di memoria volatile disponibile.
- `storage`: spazio di archiviazione disponibile.
- `bwIn`: larghezza di banda in ingresso.
- `bwOut`: larghezza di banda in uscita.

Risorse Non Consumabili

Le risorse non consumabili devono essere soddisfatte per il nodo, e sono indipendenti da altri componenti distribuiti sulla stessa infrastruttura. Queste includono:

- `security`: definisce il livello di sicurezza richiesto. I valori attualmente definiti sono:
  - `SSL`: Secure Sockets Layer, protocollo di sicurezza.
  - `encrypted storage`: archiviazione crittografata.
  - `firewall`: sistema di sicurezza per la gestione del traffico di rete.
- `latency`: tempo di risposta del sistema, fondamentale per applicazioni in tempo reale.
- `availability`: percentuale di tempo in cui il sistema deve essere operativo e accessibile, cruciale per servizi critici.

Considerazioni sulle risorse

Tutti questi campi vengono definiti da un valore intero, tranne security che è una lista di stringhe con i valori sopra definiti.

Da notare che queste sono le risorse che sono state delineate per ora, in un futuro del progetto FREEDA potrebbero essere modificate.

Un esempio in formato YAML delle risorse può essere trovato nell'appendice A.1

3.1 Applicazione

Un'applicazione è formata da una tupla `name, components, requirements` e `budget`.

- `name`: il nome dell'applicazione.

- `components`: definisce i componenti.
- `requirements`: definisce le necessità sia dei componenti stessi che delle loro relazioni.
- `budget`: definisce i massimi consumi in termini di CO2 e in budget monetario.

Un esempio di applicazione può essere trovato nell'appendice A.1.1.

3.1.1 Componente

La nozione di componente può essere vista come un servizio che vogliamo disporre nella nostra infrastruttura.

`components` è formato dai campi `name, type, flavours`.

- `name`: è univoco tra i componenti.
- `type`: non è molto significativo per quanto concerne questa tesi, ha un valore tra {`service`, `database`, `integration`} e denota il tipo del componente.
- `flavours`: definisce vari modi in cui il componente può essere dispiegato e, con quel determinato `flavour`, quali altri componenti gli servono.

I flavours sono diversi modi in cui possiamo distribuire un componente dell'applicazione, un esempio pratico potrebbe essere un backend che definisce due flavour, uno di medie dimensioni e di grandi dimensioni . Il medio potrà sostenere meno carico ma avrà bisogno anche di meno risorse, mentre il grande dovrà utilizzare più risorse ed altri servizi come un database.

Inoltre nel campo `flavours` dopo aver definito i nomi dei vari flavour, si può aggiungere la chiave `uses` che contiene un array dei componenti necessari per dispiegare il componente con quel flavour.

Un esempio può essere trovato nell'appendice A.2.

3.1.2 Requisiti

I requisiti definisco le condizioni per le quali i componenti possono essere distribuiti.

Il campo `requirements` è definito a sua volta da due sottocampi principali: `components` e `dependency`.

Il sottocapo `components` contiene, per ogni componente (definito dal nome 3.1.1), le risorse descritte nella sezione 3.0.1. Queste risorse possono essere specificate per un flavor particolare e, in tal caso, vengono inserite all'interno del campo `flavor-specific`. Se invece le risorse sono definite per tutti i flavor, vengono inserite sotto `common`. Questo approccio permette una flessibilità nell'allocazione delle risorse, adattandosi a diverse configurazioni.

Nel sottocampo `dependency` vengono invece definite le dipendenze tra i vari servizi. Qui possiamo specificare le relazioni di dipendenza tra due nodi, attualmente solo le risorse `availability` e `latency` sono utilizzate tra quelle descritte nella sezione risorse 3.0.1.

Nell'appendice si può trovare un esempio dei componenti A.2.1.

3.1.3 Budget

Il campo `budget` è strutturato in due sottocampi principali: `cost` e `carbon`.

Il sottocampo `cost` definisce il costo in termini monetari, fornendo un limite chiaro e quantificabile al budget disponibile per l'implementazione, che definiremo dopo attraverso il parametro `profile` 3.2.1 dell'infrastruttura.

Il sottocampo `carbon`, invece, definisce il massimo costo in termini di CO2 consumata.

3.2 Infrastruttura

L'infrastruttura è definita da tre componenti principali: `name`, `nodes` e `links`. La componente `name` è utilizzata per assegnare un nome significativo all'infrastruttura. La componente `nodes` rappresenta i nodi dell'infrastruttura, ciascuno dei quali è dotato di risorse specifiche. Infine, `links` definisce le risorse e i parametri relativi alle connessioni tra i nodi.

Un esempio completo dell'infrastruttura si trova nell'appendice A.2.2.

3.2.1 Nodi

La componente `nodes` contiene un elenco dettagliato dei nodi presenti nell'infrastruttura, un nodo può essere immaginato come un server (virtuale o non) su cui può essere disposto uno o più servizi. Per ciascun nodo, vengono definite le `capabilities`, ovvero le risorse di cui quel nodo dispone, come descritto nella sezione 3.0.1. Inoltre, ogni nodo ha un parametro aggiuntivo denominato `profile`, che specifica i costi associati all'utilizzo. I costi sono suddivisi in due categorie principali: il parametro `carbon`, che rappresenta l'impatto ambientale in termini di emissioni di CO2 dovute all'uso delle risorse, e il parametro `cost`, che indica i costi economici associati all'utilizzo delle risorse del nodo.

3.2.2 Links

La componente `links` è un array che descrive le connessioni tra i nodi dell'infrastruttura. Ogni elemento di questo array rappresenta una connessione specifica tra due nodi contenuti nel campo `connected_nodes`, e per ciascuna connessione, vengono definiti parametri cruciali come

la latenza (`latency`) e la disponibilità (`availability`), che devono essere definite nel campo `capabilities`.

Capitolo 4

Linguaggio Intermedio

Il linguaggio intermedio rappresenta una trascrizione strutturata dei file di input, che si avvicina alla descrizione in programmazione a vincoli del problema. Questo approccio facilita la successiva traduzione in diversi tipi di solver anche molto differenti tra loro, e la possibilità di aggiungerne facilmente altri. Inoltre rappresenta un astrazione sul modello definito in precedenza, permettendo di essere più flessibile (per esempio prendendo le risorse come parametro), rispetto al modello definito nella capitolo precedente.

Il linguaggio intermedio è definito dai seguenti elementi:

- `Comps`: insieme di tutti i nomi dei componenti dell'applicazione.
- MustComps: insieme di tutti i nomi dei componenti che devono essere distribuiti su un nodo, perché si possa eseguire l'applicazione.
- `Flavs`: insieme di tutti i nomi dei flavours. I flavours rappresentano diverse configurazioni o modalità di implementazione per ciascun componente.
- `Nodes`: insieme di tutti i nomi dei nodi dell'infrastruttura. I nodi sono le entità fisiche o virtuali su cui vengono disposti i componenti.
- `CRes`: insieme di tutte le risorse consumabili, come descritto nella sezione 3.0.1.
- `NRes`: insieme di tutte le risorse non consumabili, anch'esse descritte nella sezione 3.0.1.
- `Res`: insieme di tutte le risorse, risultante dall'unione di `CRes` e `NRes`, come definito nella sezione 3.0.1.
- `Flav`: sottoinsieme di `Comps` $\times$ `Flavs`, in cui per ogni componente vengono associati i flavours che può avere.

- `Uses`: sottoinsieme di `Comps` × `Flav` × `Comps`, che definisce le dipendenze tra i componenti e i flavours.
- `comReq`: sottoinsieme di `Comps` × `Res`, che definisce i requisiti di risorse per ciascun componente.
- `depReq`: sottoinsieme di `Comps` × `Comps` × `Res`, che definisce le dipendenze di risorse tra i componenti.
- `cost`: valore intero che definisce il costo massimo consentito per l'implementazione dell'applicazione.
- `cons`: valore intero che definisce il massimo consumo energetico consentito, misurato in termini di emissioni di CO2.
- `nodeCap`: sottoinsieme di `Nodes` × `Res`, che definisce le capacità di risorse per ciascun nodo.
- `linkCap`: sottoinsieme di `Nodes` × `Nodes` × `Res`, che definisce le capacità di risorse per le connessioni tra i nodi.

Questo modello è quello che utilizzeremo anche per modellizzare il problema con la programmazione a vincoli nel prossimo capitolo.

Capitolo 5

Modellizzazione del problema in programmazione a vincoli

Questa sezione inizia con una descrizione dettagliata della modellizzazione originale, quindi prosegue illustrando le semplificazioni introdotte durante il lavoro di ricerca.

5.1 Modellizzazione

5.1.1 Variabili

Definiamo $|\mathsf{Comps}|$ matrici $\mathcal{D}^c$ di variabili binarie di dimensioni $|Flav(c)| \times |\mathsf{Nodes}|$ tali che, per ogni $c \in \mathsf{Comps}$:

$$\mathcal{D}^c_{i,j} = \begin{cases} 1 & \text{se il componente } c \text{ è dispiegato nel flavour } i \text{ sul nodo } j \\ 0 & \text{altrimenti.} \end{cases}$$

In questo modo, per ogni $c \in \mathsf{Comps}$, possiamo definire una variabile ausiliaria:

$$node_c = \sum_{i \in Flav(c), j \in \mathsf{Nodes}} j \cdot \mathcal{D}^c_{i,j}$$

che, grazie ai vincoli definiti in (5.1), rappresenta il nodo in cui c sarà dispiegato: $node_c > 0$ *se e solo se* c è dispiegato su $node_c$.

5.1.2 Vincoli

Ogni componente è dispiegato in al massimo un flavour, su al massimo un nodo:

$$\forall c \in \mathsf{Comps} : \sum_{i \in Flav(c),\ j \in \mathsf{Nodes}} \mathcal{D}^c_{i,j} \leq 1 \tag{5.1}$$

Tutti i componenti in MustComps devono essere distribuiti:

$$\forall c \in \mathsf{MustComps} : node_c > 0 \ \wedge \sum_{i \in Flav(c), j \in \mathsf{Nodes}} \mathcal{D}^c_{i,j} = 1 \tag{5.2}$$

Si noti che il vincolo $node_c > 0$ è implicato da $\sum_{i \in Flav(c), j \in \mathsf{Nodes}} \mathcal{D}^c_{i,j} = 1$, ma aggiungere questo vincolo ridondante potrebbe essere utile, poiché il risolutore potrebbe non essere in grado di inferire questa informazione.

Se il componente c è dispiegato nel flavour i, tutti i componenti utilizzati da c in quel flavour devono essere distribuiti su qualche nodo:

$$\forall c \in \mathsf{Comps}, i \in Flav(c), c' \in Uses(c,i) : \sum_{j \in \mathsf{Nodes}} \mathcal{D}^c_{i,j} \leq \sum_{\substack{i' \in Flav(c'), \\ j' \in \mathsf{Nodes}}} \mathcal{D}^{c'}_{i',j'} \tag{5.3}$$

5.1.3 Requisiti dei componenti

Se il componente c dispiegato nel flavour i richiede una certa quantità di risorsa r, allora $node_c$ deve avere capacità per r:

$$\begin{aligned} &\forall c \in \mathsf{Comps} : node_c > 0, i \in Flav(c), r \in Req(c,i) : \\ &\quad comReq(c,i,r) \cdot \mathcal{D}^c_{i,node_c} \leq nodeCap(node_c, r) \end{aligned} \tag{5.4}$$

Questa non è una formulazione lineare poiché $node_c$ è una variabile il cui valore generalmente è sconosciuto a priori.

Se invece preferiamo una codifica lineare, il vincolo (5.4) diventa:

$$\begin{aligned} &\forall c \in \mathsf{Comps}, i \in Flav(c), r \in Req(c,i) : \\ &comReq(c,i,r) \cdot \sum_{j \in \mathsf{Nodes}} \mathcal{D}^c_{i,j} \leq \sum_{j' \in \mathsf{Nodes}} nodeCap(j', r) \cdot \mathcal{D}^c_{i,j'} \end{aligned} \tag{5.5}$$

Le formulazioni di cui sopra sono valide per tutte le risorse in Res. Inoltre, per le risorse consumabili, dobbiamo garantire che un nodo soddisfi i requisiti di risorsa per tutti i componenti distribuiti su di esso. Sia `CRes` l'insieme di tutte le risorse consumabili disponibili sul nodo j. Dobbiamo imporre che:

$$\forall j \in \mathsf{Nodes}, r \in \texttt{CRes} : \sum_{\substack{c \in \mathsf{Comps}, \\ i \in Flav(c) : r \in Req(c,i)}} comReq(c,i,r) \cdot \mathcal{D}^c_{i,j} \leq nodeCap(j,r) \tag{5.6}$$

In questo modo, ci assicuriamo che ogni nodo j abbia una quantità sufficiente di una certa risorsa r per soddisfare le esigenze di tutti i componenti distribuiti su di esso.

Requisiti di dipendenza

Se $c' \in Uses(c,i)$, esiste una dipendenza tra c e c' che potrebbe essere soggetta a un requisito infrastrutturale specifico sul collegamento che connette $node_c$ e $node_{c'}$. Per garantire il soddisfacimento di questi requisiti di dipendenza, imponiamo il seguente vincolo: [1]

$$\forall c \in \mathsf{Comps} : node_c > 0, i \in Flav(c), c' \in Uses(c,i) : node_{c'} > 0, r \in Req(c,c') : \\ depReq(c,c',r) \leq linkCap(node_c, node_{c'}, r) \tag{5.7}$$

Anche questo, è un vincolo non lineare. Una possibile codifica lineare per (5.7) è:

$$\forall c \in \mathsf{Comps}, i \in Flav(c), c' \in Uses(c,i), i' \in Flav(c'), r \in Req(c,c') : \\ depReq(c,c',r) \leq \sum_{j,j' \in \mathsf{Nodes} \,:\, j<j'} linkCap(j,j',r) \cdot z_k \tag{5.8}$$
$$z_k \in \{0,1\} \quad z_k \leq \mathcal{D}^c_{i,j} \quad z_k \leq \mathcal{D}^{c'}_{i',j'} \quad z_k \geq \mathcal{D}^c_{i,j} + \mathcal{D}^{c'}_{i',j'} - 1$$

dove le variabili ausiliarie z_k sono tali che $z_k = 1 \Leftrightarrow \mathcal{D}^c_{i,j} \cdot \mathcal{D}^{c'}_{i',j'} = 1$. Questo evita un prodotto non lineare e impone che il requisito sul collegamento $\{j, j'\}$ sia attivato solo quando $\mathcal{D}^c_{i,j} = \mathcal{D}^{c'}_{i',j'} = 1$.[2]

Requisiti di budget

Definiamo due variabili ausiliarie tot_{cost} e tot_{cons} che denotano, rispettivamente, il costo totale e il consumo energetico del deployment:

$$tot_{cost} = \sum_{c \in \mathsf{Comps}, i \in Flav(c), r \in Req(c,i), j \in \mathsf{Nodes}} comReq(c,i,r) \cdot cost(j,r) \cdot \mathcal{D}^c_{i,j} \tag{5.9}$$

$$tot_{cons} = \sum_{c \in \mathsf{Comps}, i \in Flav(c), r \in Req(c,i), j \in \mathsf{Nodes}} comReq(c,i,r) \cdot cons(j,r) \cdot \mathcal{D}^c_{i,j} \tag{5.10}$$

Se β_{cost} è il budget per il costo totale che l'amministratore è disposto a pagare per il deployment, e β_{cons} è il budget per la quantità totale di energia che l'amministratore è disposto a consumare, allora dobbiamo imporre $tot_{cost} \leq \beta_{cost}$ e $tot_{cons} \leq \beta_{cons}$.

[1] In questa sezione assumiamo, per semplicità, solo risorse non consumabili.

[2] Utilizziamo la notazione z_k come scorciatoia per $z_{c,i,j,c',i',j'}$.

5.1.4 Obiettivo

Si possono definire diverse funzioni obiettivo. Un possibile approccio sarebbe assegnare pesi (normalizzati) γ e ε per il costo e il consumo energetico, rispettivamente. Formalmente, la funzione obiettivo da minimizzare sarebbe:

$$\gamma \cdot tot_{cost} + \varepsilon \cdot tot_{cons} \tag{5.11}$$

La parte critica consiste nella determinazione di valori ragionevoli per i pesi, poiché essenzialmente mescoliamo costi delle risorse (ad esempio, il costo unitario per RAM e CPU misurato in una certa valuta) con costi energetici (ad esempio, emissioni di CO_2 espresse in gCO2-eq/KWh). L'approccio più semplice è concentrarsi su obiettivi individuali, cioè un deployment *orientato al costo* dove $\gamma = 1, \varepsilon = 0$, o un deployment *orientato all'energia*, dove $\gamma = 0, \varepsilon = 1$.

Un approccio alternativo per evitare i problemi di cui sopra è concentrarsi su quanto sia *importante* distribuire un componente in un certo flavour. Per fare ciò, si può utilizzare una funzione di importanza $imp : \mathsf{Comps} \times \mathsf{Flavs} \to \mathbb{N}$ tale che $imp(c, f)$ sia l'importanza di distribuire il componente c quando è dispiegato nel flavour $f \in Flav(c)$ (assumiamo che un valore più alto di imp sia migliore). La funzione obiettivo da massimizzare sarebbe quindi:

$$\sum_{\substack{c \in \mathsf{Comps}, \\ i \in Flav(c)}} imp(c, i) \cdot \sum_{j \in \mathsf{Nodes}} \mathcal{D}^c_{i,j} \tag{5.12}$$

Una prima implementazione della funzione di importanza (imp) potrebbe essere quella di dare ad ogni flavour un numero incrementale, questo può dare problemi in quanto si potrebbe preferire involontariamente un flavour minore.

Ad esempio siano f è f' due flavour, dove f è più importante di f'. Definiamo $\mathsf{Comps} = \{c, c'\}$ quindi definiamo $c \in \mathsf{Comps} : imp(c, f) = 2$ e $c \in \mathsf{Comps} : imp(c, f') = 3$ e definiamo uses come $Uses(c, f) = \emptyset$, $Uses(c, f') = \{c'\}$ e infine $\mathsf{MustComps} = \{c\}$ a questo punto il solver potrebbe fare il deploy di entrambi piuttosto che il deploy di quello più grande.

Per non avere questo problema possiamo lasciare degli spazio tra i valori. Ordiniamo i flavour dal più al meno importante. Siano λ il numero di flavour, e $f^{(i)}$ il flavour con importanza i e $n_{f^{(j)}}$ il numero di componenti con il flavour $f^{(i)}$ allora definiamo la funzione di importanza come:

$$imp(c, f^{(i)}) = \begin{cases} 1 & \text{se } i = 1 \\ 1 + \sum_{j=1}^{i-1} n_j \cdot imp(c, f^{(j)}) & \text{altrimenti} \end{cases} \tag{5.13}$$

Che può essere scritta anche come:

$$1 + \sum_{j=1}^{i-1} n_j \cdot imp(c, f^{(j)}) = \prod_{j=1}^{i-1} (n_j + 1) \tag{5.14}$$

Con questo funzione obbiettivo la somma di tutti i flavour con una priorità è minore della somma di ogni flavour di priorità maggiore. Questo però fa si che la funzione sia esponenziale dove il valore massimo è n^f dove n è il numero di componenti e f è il numero di flavour.

5.2 Modifiche Apportate

In questa sezione si procederà alla rimodellazione di alcune parti definite in precedenza. Sebbene questa nuova modellizzazione possa apparire più complessa, essa consente, come verrà mostrato successivamente, di introdurre semplificazioni derivanti da precomputazioni.

5.2.1 Variabili

Le variabili $node_c$ definite nella sezione 5.1.1 verranno eliminate, in quanto non necessarie. Si manterranno invece le variabili D definite nella sezione 5.1.1, le quali saranno utilizzate per indicare se un componente con un determinato flavour è dispiegato sul nodo corrispondente.

5.2.2 Vincoli

I vincolo di MustComps 5.15 che utilizza la variabile $node_c$, che non utilizziamo più, diventa:

$$\forall c \in \mathsf{MustComps} : \sum_{i \in Flav(c), j \in \mathsf{Nodes}} \mathcal{D}^c_{i,j} = 1 \tag{5.15}$$

Non vi sono modifiche significative, poiché come era già stato segnalato il vincolo $node_c > 0$ nella sezione 5.1.2 era ridondante.

Inoltre al posto della disequazione 5.4, utilizziamo questa codifica:

$$\begin{aligned} &\forall c \in \mathsf{Comps}, i \in Flav(c), r \in Req(c,i), \forall n \in \mathsf{Nodes} : \\ &\quad comReq(c,i,r) > nodeCap(n,r) \implies \mathcal{D}^c_{i,n} = 0 \end{aligned} \tag{5.16}$$

Mentre per la disequazione 5.7 diventa:

$$\begin{aligned} \forall c \in \mathsf{Comps}, i \in Flav(c), c' \in Uses(c,i), r \in Req(c,c'), f' \in Flav(c') : \\ \forall n \in \mathsf{Nodes}, n' \in \mathsf{Nodes} : \\ (depReq(c,c',r) > linkCap(n,n',r)) \implies D^c_{f,n} + D^{c'}_{f',n'} \leq 1 \end{aligned} \tag{5.17}$$

Poiché tali modifiche non sono ovvie e immediate, verranno giustificate nel capitolo dedicato all'implementazione.

Dimostrazione dell'equivalenza del vincolo dei nodi

Per sostituire la formula 5.4 alla formula 5.16 dobbiamo dimostrare la loro equivalenza. Partendo da:

$$\begin{aligned} \forall c \in \mathsf{Comps}, i \in Flav(c), r \in Req(c,i) : \\ node_c > 0 \implies comReq(c,i,r) \cdot \mathcal{D}^c_{i,node_c} \leq nodeCap(node_c, r) \end{aligned} \tag{5.18}$$

deve dimostrare l'equivalenza con:

$$\begin{aligned} \forall c \in \mathsf{Comps}, i \in Flav(c), r \in Req(c,i), n \in \mathsf{Nodes} : \\ comReq(c,i,r) > nodeCap(n,r) \implies \mathcal{D}^c_{i,n} = 0 \end{aligned} \tag{5.19}$$

Utilizzando l'espansione universale per i valori di $node_c$, si trasforma in:

$$\begin{aligned} \forall c \in \mathsf{Comps}, i \in Flav(c), r \in Req(c,i), n \in \mathsf{Nodes} : \\ node_c = n \wedge node_c > 0 \implies comReq(c,i,r) \cdot \mathcal{D}^c_{i,n} \leq nodeCap(n,r) \end{aligned} \tag{5.20}$$

Innanzitutto, si sostituire $node_c$ con n, poi possiamo riscrivere il vincolo $node_c = n$ che viene espanso come $\sum_{\forall f \in \mathsf{Flavs}(c), \forall nt \in \mathsf{Nodes}} D^c_{f,nt} \cdot nt = n$. Questo vincolo si può semplificare in quanto può esistere solo un flavour su un solo componente dispiegato su un determinato nodo 5.1), quindi possiamo trasformarlo in $D^c_{f,n} = 1$. Di conseguenza, il vincolo $node_c > 0$ diventa ridondante, pertanto:

$$\begin{aligned} \forall c \in \mathsf{Comps}, i \in Flav(c), r \in Req(c,i), n \in \mathsf{Nodes} : \\ D^c_{f,n} = 1 \implies comReq(c,i,r) \leq nodeCap(n,r) \end{aligned} \tag{5.21}$$

Passando alla contro-nominale, da notare che $D^c_{f,n}$ può assumere solo i valori 0 e 1 quindi:

$$\begin{aligned} &\forall c \in \mathsf{Comps}, i \in \mathit{Flav}(c), r \in \mathit{Req}(c,i), n \in \mathsf{Nodes} : \\ &\quad \mathit{comReq}(c,i,r) > \mathit{nodeCap}(n,r) \implies D^c_{f,n} = 0 \end{aligned} \tag{5.22}$$

Dimostrazione dell'equivalenza del vincolo di link

Per sostituire la formula 5.7 alla formula 5.17 dobbiamo dimostrare la loro equivalenza. Partendo da:

$$\begin{aligned} &\forall c \in \mathsf{Comps}, i \in \mathit{Flav}(c), c' \in \mathit{Uses}(c,i), r \in \mathit{Req}(c,c'), : \\ &\mathit{node}_c > 0 \wedge \mathit{node}_{c'} > 0 \wedge D^c_{f,\mathit{node}_c} = 1 \implies \mathit{depReq}(c,c',r) \leq \mathit{linkCap}(\mathit{node}_c, \mathit{node}_{c'}, r) \end{aligned} \tag{5.23}$$

Si deve dimostrare l'equivalenza con:

$$\begin{aligned} &\forall c \in \mathsf{Comps}, i \in \mathit{Flav}(c), c' \in \mathit{Uses}(c,i), r \in \mathit{Req}(c,c'), f' \in \mathit{Flav}(c') : \\ &\qquad \forall n \in \mathsf{Nodes}, n' \in \mathsf{Nodes} : \\ &(\mathit{depReq}(c,c',r) > \mathit{linkCap}(n,n',r)) \implies D^c_{f,n} + D^{c'}_{f',n'} \leq 1 \end{aligned} \tag{5.24}$$

Procediamo utilizzando l'espansione universale per i valori di node_c e node'_c, questo diventa:

$$\begin{aligned} &\forall c \in \mathsf{Comps}, i \in \mathit{Flav}(c), c' \in \mathit{Uses}(c,i), r \in \mathit{Req}(c,c'), n \in \mathsf{Nodes}, n' \in \mathsf{Nodes} : \\ &\qquad \mathit{node}_{c'} = n' \wedge \mathit{node}_c = n \wedge \mathit{node}_c > 0 \wedge \\ &\mathit{node}_{c'} > 0 \wedge D^c_{f,\mathit{node}_c} = 1 \implies \mathit{depReq}(c,c',r) \leq \mathit{linkCap}(\mathit{node}_c, \mathit{node}_{c'}, r) \end{aligned} \tag{5.25}$$

Si rimuovono i vincoli non utili per esempio $\mathit{node}_c > 0$ e $\mathit{node}_{c'} > 0$, poichè per avere $\mathit{node}_{c'} = n'$ e $\mathit{node}_c = n$ è implicato che siano maggiori di zero. Inoltre si procede a sostituire node_c con n e $\mathit{node}_{c'} = n'$.

$$\begin{aligned} &\forall c \in \mathsf{Comps}, i \in \mathit{Flav}(c), c' \in \mathit{Uses}(c,i), r \in \mathit{Req}(c,c'), n \in \mathsf{Nodes}, n' \in \mathsf{Nodes} : \\ &\mathit{node}_{c'} = n' \wedge \mathit{node}_c = n \wedge D^c_{f,n} = 1 \implies \mathit{depReq}(c,c',r) \leq \mathit{linkCap}(n,n',r) \end{aligned} \tag{5.26}$$

Si osserva che il vincolo $node_c = n$ è ridondante con $D^c_{f,n} = 1$, infatti $node_c = n$ viene espansa come $\sum_{\forall f \in \mathsf{Flavs}(c), \forall nt \in \mathsf{Nodes}} D^c_{f,nt} \cdot nt = n$. Considerando che un componente può essere disposto su un solo nodo con un solo flavour, come indicato nel vincolo 5.1, si può riscrivere tale vincolo come $D^c_{f,n} = 1$.

$$\forall c \in \mathsf{Comps}, i \in Flav(c), c' \in Uses(c,i), r \in Req(c,c'), n \in \mathsf{Nodes}, n' \in \mathsf{Nodes}: \quad node_{c'} = n' \wedge D^c_{f,n} = 1 \implies depReq(c,c',r) \leq linkCap(n,n',r) \tag{5.27}$$

Successivamente, si espande $node_{c'} = n'$, come segue:

$$\sum_{\forall f' \in \mathsf{Flavs}(c'), \forall nt \in \mathsf{Nodes}} D^{c'}_{f',nt} \cdot nt = n'$$

Questo vincolo è equivalente a la condizione:

$$\forall f \in Flav(c') : D^{c'}_{f,n'} = 1$$

in virtù del vincolo 5.1.

$$\forall c \in \mathsf{Comps}, i \in Flav(c), c' \in Uses(c,i), r \in Req(c,c'), n \in \mathsf{Nodes}, n' \in \mathsf{Nodes}: \quad D^{c'}_{f,n'} = 1 \wedge D^c_{f,n} = 1 \implies depReq(c,c',r) \leq linkCap(n,n',r) \tag{5.28}$$

Da qui passiamo alla contronominale.

$$\forall c \in \mathsf{Comps}, i \in Flav(c), c' \in Uses(c,i), r \in Req(c,c'), n \in \mathsf{Nodes}, n' \in \mathsf{Nodes}, \forall f' \in Flav(c'): \neg(depReq(c,c',r) \leq linkCap(n,n',r)) \implies \neg(D^{c'}_{f,n'} = 1 \wedge D^c_{f,n} = 1)$$

Utilizzando le leggi De Morgan[3], e considerando che il dominio di $D^c_{f,n}$ può assumere solo i valori 0 e 1, riscriviamo la formula come:

[3]Le leggi di De Morgan, sono relative alla logica booleana e stabiliscono relazioni di equivalenza tra gli operatori di congiunzione e disgiunzione logica.Wikipedia [Wik24a]

$$\forall c \in \mathsf{Comps}, i \in Flav(c), c' \in Uses(c,i), r \in Req(c,c'), n \in \mathsf{Nodes}, n' \in \mathsf{Nodes}, \forall f' \in Flav(c') : \\ dreq(c,c',r) > linkCap(n,n',r) \implies D^{c'}_{f',n'} = 0 \vee D^{c}_{f,n} = 0 \tag{5.29}$$

Il cui vincolo $D^{c'}_{f',n'} = 0 \vee D^{c}_{f,n} = 0$ è equivalente ad $D^{c'}_{f',n'} + D^{c}_{f,n} \leq 1$

5.2.3 Obbiettivo

La funzione obbiettivo definita in precedenza nella sezione 5.1.4, potrebbe dare dei risultati insoliti in quanto tutti i componenti hanno la stessa importanza a parità di flavour. Infatti, tale approccio massimizzerà un flavour di un componente inutilizzato anche a discapito dei componenti MustComps.

Per esempio, consideriamo due componenti uno MustComps e uno inutilizzato, dove le possibili distribuzioni sono:

- i due componenti MustComps a medium e quello inutilizzato a tiny
- i due componenti MustComps a tiny e quello inutilizzato a large

In questo scenario verrà scelto dal solver di effettuare il dispiegamento dei componenti di flavour tiny piuttosto che disporne due a medium e quello inutilizzato a tiny. La soluzione così trovata risulta talvolta essere controproducente.

Un'alternativa consiste nel definire tre fasce di importanza. La prima la più rilevante, mira a massimizzare i componenti MustComps. La seconda fascia comprende gli altri componenti. In modo da garantire che venga sempre data la priorità alla fascia più alta, prima di passare alla successiva.

$$\begin{aligned} \text{Must} &= \sum_{\forall c \in \mathsf{MustComps}, i \in \lambda, n \in \mathsf{Nodes}} D^{c}_{f^{(i)},n} \cdot i \\ \text{Others} &= \sum_{\forall c \notin \mathsf{MustComps}, i \in \lambda, n \in \mathsf{Nodes}} D^{c}_{f^{(i)},n} \cdot i \end{aligned} \tag{5.30}$$

La funzione obbiettivo diventa $\text{Must} \cdot (n \cdot \lambda) + \text{Others}$, che a differenza di quella definita in precedenza nella sezione 5.1.4, ha una crescita polinomiale $(n \cdot \lambda)^2$.

Una versione più complessa è descritta nell'appendice B.

Capitolo 6

Implementazione

6.1 Implementazione del Parser

Come precedentemente illustrato, il parser riceve in input due file, `Applicazione` e `Infrastruttura`, già definiti nel capitolo 3. Il programma carica il contenuto di questi file YAML in strutture dati di Python. Queste strutture dati facilitano la verifica e la validazione dei dati [1]. Successivamente partendo da questi dati, viene generato il formato intermedio come descritto nel capitolo 4.

Nel formato intermedio, i dati non vengono codificati in strutture rigide come nella fase precedente, ma sono trascritti come mappe e liste di Python. Questo approccio offre maggiore flessibilità, consentendo una manipolazione più agevole poiché i controlli di correttezza sono già stati effettuati.

Dallo schema intermedio sono state poi create quattro traduzioni in altrettanti diversi linguaggi di modellazione.

- **MiniZinc**: come già delineato in precedenza, è linguaggio di modellazione specifico per la programmazione vincolata, che permette un interfaccia attraverso una libreria omonima scritta in python, per creare e risolvere problemi scritti in questo linguaggio.
- **Pulp**: un formato standard per la programmazione matematica, creato utilizzando la libreria PuLP di Python. È ampiamente utilizzato per la modellazione di problemi di programmazione lineare e a interi misti.
- **Z3**: un potente risolutore SMT che supporta anche l'ottimizzazione di funzioni, che offre tramite le API scritte anche in Python.

[1] Per la validazione dei dati viene utilizzata la liberia Pydantic

- **pySMT**: offre delle API Python, per il formato smt-lib v2.

6.2 Differenze nella modellazione

La formulazione del problema nei diversi linguaggi di modellazione varia in modo significativo a seconda del linguaggio utilizzato. In particolare, la creazione dell'istanza in MiniZinc segue il modello presentato in precedenza 5.1, mentre per la creazione dell'istanza in MPS, Z3 e SMT sono state apportate le modifiche descritte nella sezione 5.2.

Come anticipato, la seconda modellazione è stata modificata al fine di rendere il problema più efficiente e più semplice da risolvere. Inoltre essendo che viene implementate per più solver, si è creata un interfaccia comune, per rende più facile l'aggiunta di nuove traduzioni.

Questa codifica permette di eseguire controlli statici durante la generazione dell'istanza del problema. Infatti i vincoli 5.16 e 5.17, possono essere controllati staticamente durante la generazione dell'istanza del problema. Ad esempio, con il vincolo dei requirements modificato 5.16, è possibile precomputare $comReq(c, i, r) > nodeCap(n, r)$ e, se questa condizione è vera, si aggiungere il vincolo $D^c_{f,n} = 0$. Questo consente di evitare che il solver debba dedurre tale vincolo durante la risoluzione del problema.

In modo analogo, anche per il vincolo di dipendenza 5.17 si possono effettuare controlli statici. Infatti, calcolando $depReq(c, c', r) > linkCap(n, n', r)$ in modo statico e, se questo vincolo risulta valido, si può aggiungere il vincolo $D^c_{f,n} + D^{c'}_{f',n'} \leq 1$. Queste ottimizzazioni migliorano notevolmente l'efficienza del processo di risoluzione, riducendo il carico di lavoro del solver.

6.3 Benchmark

In questa sezione si presentano i risultati dei benchmark eseguiti su quattro diversi solver. I test sono stati condotti su un numero limitato di esempi, generati randomicamente da un programma che crea i file di applicazione e infrastruttura parametrizzati in base al numero di componenti per applicazione, al numero di nodi e al numero di risorse.

I solver utilizzati per questo benchmark sono:

- **Gecode**: un solver per la programmazione vincolata, implementato in C++ e noto per la sua efficienza nell'implementazione di vari algoritmi di ricerca e propagazione. È stato utilizzato per risolvere i modelli MiniZinc, SMT e Z3.
- **CBC**: un solver open source per la mixed integer programming (MIP) e la programmazione lineare. È stato utilizzato per risolvere i modelli MPS.

I test sono stati eseguiti su un unico computer[2], con risultati che rappresentano la media di 8

[2]CPU:AMD Ryzen 5 7520U RAM:8 GB OS:Linux

esecuzioni. In ogni esecuzione sono stati generati 24 file di applicazione e 24 di infrastruttura, con il numero di componenti che varia da 2 a 25 e il numero di nodi che varia da 2 a 25. Su ciascuna coppia di queste istanze, sono stati eseguiti i vari solver, tutti con un timeout di 10 secondi.

Va notato che le modellazioni diverse da MiniZinc richiedono delle precomputazioni, come accennato in precedenza 6.2, e questi tempi di traduzione dal linguaggio intermedio al linguaggio di modellazione sono stati considerati nei risultati finali.

Di seguito vengono presentati una serie di grafici che illustrano i risultati dei benchmark. La Figura 6.1 mostra tutti i benchmark sovrapposti, permettendo un confronto visivo immediato delle prestazioni dei vari solver.

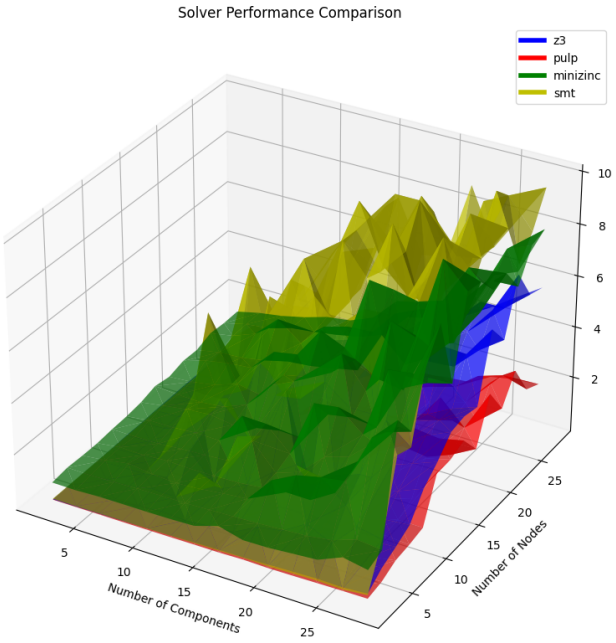

Figura 6.1: Tutti i benchmark sovrapposti

Le Figure 6.6 mostrano i benchmark individuali per ciascun solver. Questi grafici forniscono una visione dettagliata delle prestazioni di ogni solver su una serie di test specifici.

I risultati mostrano che MPS con CBC è il solver più veloce, seguito da Gecode con l'encoding di Z3, MiniZinc e SMT. Questo è probabilmente dovuto al fatto che la modellazione utilizzata è stata scritta in vincoli lineari, il che rende specifico per questo tipo di solver.

In particolare, Z3 può essere paragonato a MiniZinc poiché entrambi utilizzano lo stesso solver, e quindi la differenza di prestazioni può essere attribuita principalmente alla modellazione,

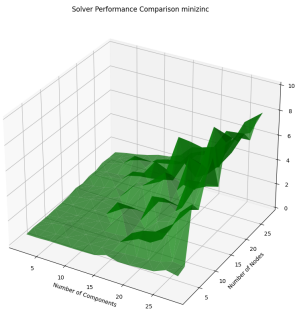

Figura 6.2: MiniZinc con Gecode

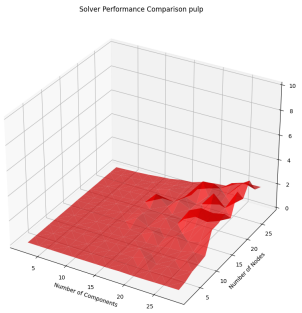

Figura 6.3: MPS con CBC

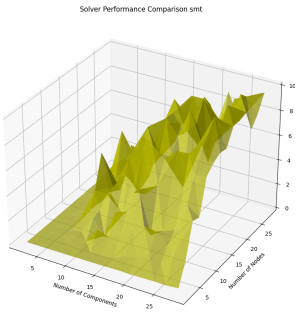

Figura 6.4: SMT con Gecode

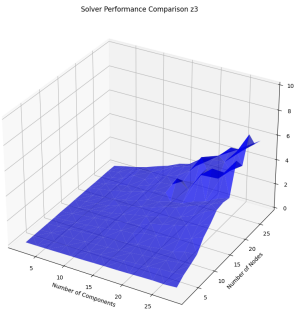

Figura 6.5: Z3 con Gecode

Figura 6.6: Benchmark individuali per ciascun solver

infatti come precedentemente specificato MiniZinc utilizza la prima modellazione presentata, mentre Z3 la seconda.

SMT con Gecode risulta essere il più lento. Questo è probabilmente dovuto al come la modellazione è stata scritta in SMT. Infatti implementazione utilizza variabili booleane, a differenza della implementazione Z3 che invece utilizza variabili intere con un bound da 0 a 1. Questo comporta l'aggiunta di operazioni per convertire da booleano a intero nella maggior parte dei vincoli, rendendo forse più difficile per il solver inferire informazioni.

Grazie a questo benchmark, abbiamo anche mostrato empiricamente l'equivalenza tra le diverse modellazioni e le implementazioni dei solver, poiché i risultati ottenuti sono stati equivalenti in termini di correttezza, nonostante hanno evidenziato differenze significative nei tempi di risoluzione. Questo aiuta a confermare la corretta implementazione delle traduzioni.

Capitolo 7

Conclusioni

7.1 Risultati raggiunti

Il lavoro svolto ha permesso di raggiungere diversi risultati significativi nel progetto FREEDA:

- **Sviluppo di un Parser**: È stato sviluppato un parser in grado di convertire i file di configurazione del deployment in un formato utilizzabile da un solver di programmazione a vincoli.
- **Modellazione del Problema**: È stata effettuata una modellazione dettagliata del problema di deployment, utilizzando la programmazione a vincoli.
- **Implementazione di Solver**: Sono stati implementati diversi solver, utilizzando linguaggi e strumenti come MiniZinc, PuLP, Z3 e pySMT.
- **Benchmarking**: È stata condotta una serie di benchmark per valutare le prestazioni dei diversi solver. I risultati hanno evidenziato che il solver basato su ILP con CBC è risultato il più veloce, mentre il solver Gecode è risultato il più lento.

7.2 Sviluppi futuri

Gli sviluppi futuri sono legati agli obiettivi del progetto FREEDA. In particolare, si prevede di sviluppare le seguenti tecniche:

- **Deployment Adattivo**: Tecniche per l'adaptive deployment, che permetteranno di adattare dinamicamente il deployment delle MSA in risposta ai cambiamenti nell'infrastruttura o nei requisiti dell'applicazione. Questo comporta l'utilizzo di pratiche di continuous reasoning per garantire che solo le parti necessarie del deployment vengano aggiornate, minimizzando così i tempi di inattività e le risorse utilizzate.

- **Resilienza ai Fallimenti**: Tecniche per il deployment resiliente ai fallimenti, che includono l'implementazione di meccanismi di tolleranza ai guasti e all'analisi continua delle dipendenze e delle vulnerabilità dei servizi per prevenire la propagazione dei guasti.
- **Sostenibilità Energetica**: Tecniche di deployment che considerano l'uso ottimale delle risorse energetiche, riducendo il consumo energetico complessivo delle applicazioni distribuite su infrastrutture Cloud-IoT.

Appendice A

Esempi Yaml dei file di applicazione e infrastruttura

A.1 Esempio Risorse

Struttura generale è del tipo:

```yaml
nome_campo:
    value: any # può essere più tipi di dato
```

Alcuni esempi:

```yaml
cpu:
  value: 7
security:
  value:
  - encrypted_storage
```

A.1.1 Esempio completo Applicazione

```yaml
name: Example 2 component
components:
  frontend:
    type: service
    flavours:
      large:
        uses: [backend]
      medium:
        uses: []
  backend:
    type: service
    flavours: {tiny: []}
requirements:
  components:
    frontend:
      common:
        security: {value: []}
      flavour-specific:
        large:
          storage:
            value: 258
        medium:
          storage: {value: 652}
    backend:
      common:
        bwIn: {value: 58}
        bwOut: {value: 69}
      flavour-specific:
        tiny:
          security: {value: []}
          storage: {value: 242}
  dependencies:
    frontend:
      backend:
        availability: {value: 91}
        latency: {value: 99}
  budget:
    carbon: 99
```

```
    cost: 30358
```

A.2 Esempio Componente

La struttura di un componente è il seguente:

```
component_name:
    type: component_type
    flavours:
        flavour_name:
            uses: [ list_of_components ]
```

Dove:

- `component_name`: il nome del componente.
- `component_type`: il tipo del componente.
- `flavour_name`: il nome del flavour specifico.
- `list_of_components`: la lista dei componenti necessari per il flavour specificato.

Un esempio di un componente può essere:

```
frontend:
    type: service
    flavours:
      large:
        uses:  [backend]
      medium:
        uses:  [backend]
  backend:
    type: service
    flavours:
      large:
        uses:  [database]
      medium:
        uses:  [database]
      tiny:
        uses:  []
  database:
    type: database
    flavours:
```

```
20      large:
21        uses:  []
```

A.2.1 Dei Requirement

```yaml
components:
    frontend:
      common:
        bwIn:
          value: 1000
        bwOut:
          value: 1000
        availability:
          value: 98
          soft: true
        security:
          value: [ssl, firewall]
      flavour-specific:
        large:
          cpu:
            value: 2
            soft: true
          ram:
            value: 10
    backend:
      common:
        bwOut:
          value: 600
        security:
          value: [ssl]
      flavour-specific:
        large:
          cpu:
            value: 4
            soft: true
          ram:
            value: 80
        medium:
          cpu:
            value: 2
            soft: true
          ram:
            value: 40
dependencies:
```

```
    frontend:
      backend:
        latency:
          value: 50
```

A.2.2 Esempio Completo Infrastruttura

```yaml
name: Infrastruttura
nodes:
  node0:
    capabilities:
      availability: 95
      bwIn: 54
      bwOut: 27
      cpu: 19
      ram: 13
      security: [ssl, encrypted_storage, firewall]
      storage: 683
    profile:
      carbon: 9
      cost: {cpu: 10, ram: 6, storage: 4}
  node1:
    capabilities:
      availability: 100
      bwIn: 78
      bwOut: 49
      cpu: 1
      ram: 20
      security: [encrypted_storage, ssl]
      storage: 652
    profile:
      carbon: 4
      cost: {cpu: 8, ram: 2, storage: 9}
links:
- capabilities:
    availability: 100
    latency: 90
  connected_nodes: [node0, node1]
```

Appendice B

Versione alternativa della funzione obbiettivo

Partendo dalla funzione obbiettivo definita nella sezione 5.2.3, definiamo una versione più complessa creando una fascia intermedia dei componenti utilizzati direttamente dai MustComps.

Per definire questa nuova funzione, è necessario introdurre concetti nuovi. Innanzitutto, si osserva che gli *Uses* possono essere rappresentati anche come un grafo. Ogni volta che esiste un percorso che inizia da un componente MustComps con un determinato flavour verso un altro componente, si verifica se questo percorso è utilizzato. In tal caso si può aggiungere il costo di utilizzo.

Per controllare se un percorso che parte da un nodo MustComps è utilizzato introduciamo, $\mathbf{MustPath}(c)$ come l'insieme dei percorsi che partono da un MustComps e arrivano ad un componente.

Si fornisce il seguente esempio, siano $\mathsf{Comps} = \{c^1, c^2, c^3\}$, $\mathsf{Flavs} = \{t, m\}$ e $Uses(c^1, t) = \{c^2\}$, $Uses(c^1, m) = \{c^3\}$, $Uses(c^2, m) = \{c^3\}$

Allora il grafo dei percorsi diventa:

E di conseguenza $\mathbf{MustPath}(c2) = \{\{(c^1, t)\}\}$, mentre $\mathbf{MustPath}(c3) = \{\{(c^1, t), (c^2, m)\}$, $\{(c^1, m)\}\}$.

Per verificare che un componente c' sia utilizzato andiamo ad utilizzare la formula:

$$\bigvee_{\text{path} \in \mathbf{MustPath}(c')} \quad \bigwedge_{(c,f) \in \text{path}} \sum_{n \in \mathsf{Nodes}} D^c_{f,n} = 1$$

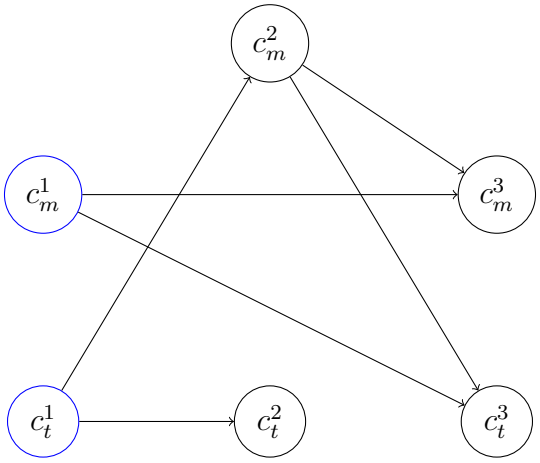

Figura B.1: I nodi blu rappresentano i componenti MustComps

Equivalente a:

$$1 - \prod_{\text{path} \in \textbf{MustPath(c')}} (1 - \prod_{(c,f) \in \text{path}} \sum_{n \in \text{Nodes}} D_{f,n}^{c})$$

Siano λ il numero di flavour, e $f^{(i)}$ il flavour con importanza i e n_{fj} il numero di componenti con il flavour, la fascia intermedia `Uses` si definisce come:

$$\texttt{Used} = \sum_{\substack{\forall c \notin \text{MustComps}, i \in \lambda, \\ n \in \text{Nodes}}} (1 - \prod_{\text{path} \in \textbf{MustPath(c)}} (1 - \prod_{(c',f) \in \text{path}} \sum_{n \in \text{Nodes}} D_{f,n}^{c'})) \cdot i$$

La funzione obbiettivo si modifica di conseguenza in $\text{Must} \cdot (n \cdot \lambda)^2 + \text{Used} \cdot (n \cdot \lambda) + \text{Others}$, che come quella definita in precedenza 5.1.4 ha una crescita polinomiale, ma con un esponente maggiore, infatti diventa $(n \cdot \lambda)^3$.

Questa proposta utilizza una funzione obbiettivo non lineare e abbastanza complessa. Per queste ragioni potrebbe implicare delle prestazioni peggiori.

Bibliografia

[C+11] David R Cok et al. The smt-libv2 language and tools: A tutorial. *Language c*, pages 2010–2011, 2011.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[MSKS14] Kim Marriott, Peter J Stuckey, LD Koninck, and Horst Samulowitz. A minizinc tutorial. *A minizinc tutorial*, 2014.

[VSA+24] Monica Vitali, Jacopo Soldani, Roberto Amadini, Antonio Brogi, Stefano Forti, Simone Gazza, Saverio Giallorenzo, Pierluigi Plebani, Francisco Ponce, and Gianluigi Zavattaro. Freeda: Failure-resilient, energy-aware, and explainable deployment of microservice-based applications over cloud-iot infrastructures. In *CAiSE Research Projects Exhibition*, pages 69–75, 2024.

[Wik24a] Wikipedia. Leggi di de morgan — wikipedia, l'enciclopedia libera, 2024. [Online; in data 24-giugno-2024].

[Wik24b] Wikipedia contributors. Mps (format) — Wikipedia, the free encyclopedia, 2024. [Online; accessed 24-June-2024].