ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Analisi e sviluppo di middleware DDS per la gestione dei consumi in sistemi HPC

Relatore
Prof. Andrea Bartolini

Candidato
Giacomo Madella

Ottobre 2023

Qualcosa è ancora da sistemare, e verrà fatto tra domani e mercoledì, come ad esempio la pagina bianca qui, numerazione delle pagine ed alcuni grafici.

Abstract

Nei sistemi di High-Performance Computing (HPC), la gestione energetica è diventata una delle principali preoccupazioni, non solo a causa dei costi monetari, ma anche per la sostenibilità ambientale e per la progettazione di nuove generazioni di supercomputer[1]. Perpendicolarmente all'aumento della potenza computazionale richiesta, le tecnologie associate allo sviluppo dei componenti che stanno alla base dei processori, si sono avvicinati sempre più ai loro limiti fisici. E' nato con questo, il concetto di Power Management cercando di definire un modello software con il compito di gestire la potenza di sistemi di HPC. Data l'eterogeneità di questi sistemi, nel corso degli anni sono stati proposti sempre più strumenti in grado di funzionare su una specifica configurazione hardware e cercando di risolvere un sottoinsieme limitato di problemi. Lo scopo di questa tesi è di testare un middleware di comunicazione basato su Data Distribution Service(DDS), che faciliterebbe lo scambio di informazioni tra diversi software di Power Management. Questo permetterebbe di creare un Power Stack interoperable e con una visione di insieme. Successivamente sarà stilato un modello di utilizzo, ed in collaborazione con il progetto REGALE, un esempio di implementazione degli attori coinvolti.

La tesi è organizzata nel seguente modo: nel primo capitolo viene introdotto il concetto di Power Management, nel secondo rappresentato lo stato dell'arte. Dopo una introduzione di DDS nel terzo viene presentato il progetto REGALE. Nel quarto verranno riportati i test effettuati con i relativi risultati nel sesto. Dopo una breve introduzione dei prototipi creati sarà presente una conclusione.

Va qui l'intro dei capitoli?

Indice

Introduzione

Il termine power management è stato usato nel corso degli anni per raggruppare problemi di diversa tipologia, ma che ruotano tutti attorno al concetto di energia. Tra questi infatti si può includere:

- Power management legata alla gestione della potenza assorbita, a sua volta suddivisibile in:
  - Thermal Design Power, potenza termica massima che un componente può dissipare;
  - Therm Design Current o Peak Current, legata alla massima corrente erogabile da alimentatori o dai processori;

Tutto ok qui?

- Thermal management, gestione temperatura dinamica o statica;
- Energy management, gestione della sostenibilità e del consumo di energia;

In questa tesi, si farà riferimento a questa parola per abbracciare tutti e tre i concetti che essa può rappresentare, offrendo così una visione olistica e completa del problema.

Il contesto nel quale viene definito un Power Management è spesso un sistema di *High-Performance Computing*, detto anche sistema ad alte prestazioni. Questi ultimi sono macchine computazionali composte da cluster di decine o a volte centinaia di nodi interconnessi tra di loro da reti a bassa latenza. Ogni nodo è composto a sua volta da decine di processori, ed acceleratori come CPU, GPU e TPU. Inoltre ogni nodo mette a disposizione memorie di diverso tipo, con capienze elevate e ad alta banda, condivisibile tra i processori al suo interno. Andando a considerare tutti i cluster nel loro insieme, si ottengo capacità computazioni che nei giorni nostri hanno raggiunto ordini del ExaFlops (10^{18} operazioni di Floating Point per secondo).

In contrasto a ciò, dagli anni '70 ad oggi si sono manifestate difficolta sempre più grandi nel ridimensionamento dei transistor, che ha portato ad una progressiva fine delle leggi di Denard e Moore[2][3]. Tali leggi, che avevano guidato l'industria informatica per decenni, prevedevano un consumo energetico costante al crescere della velocità e capacità computazionale. Quando la loro efficacia è venuta a mancare, il mantenimento

e ancora di più lo sviluppo di nuove generazioni di sistemi sono diventati compiti tutt'altro che banali[1], rendendo sempre più di vitale importanza i software in grado di automatizzarne la gestione. Dall'arrivo degli exa-computer[1], la potenza necessaria per alimentare questi sistemi ha superato la precedente soglia dei 20MWatt[4]. Se poi si considera che la maggior parte della potenza fornita, viene convertita in calore, si deve prendere in considerazione anche i consumi necessari per tenere raffreddati i sistemi. Infatti, se non adeguati comporterebbero grandi inefficienze in termini di energia, che si traducono anche in degradazioni di prestazioni computazionali. Prendendo in considerazione ogni aspetto, i centri che ospitano queste macchine necessitano di decine di MWatt di potenza per ogni exa-computer che hanno in funzionamento. Ordini di grandezza di questo tipo non sono facilmente raggiungibili e anche quando lo sono, hanno costi estremamente elevati. Al fine di definire dei power budget, e utilizzare efficientemente la potenza richiesta si sono resi necessari strumenti in grado di operare su diversi livelli di astrazione. Sono nati così i primi concetti di Power Management, componenti per controllare l'utilizzo di energia utilizzando diverse strategie, al fine di ridurre gli sprechi energetici e, allo stesso tempo, garantire un temperatura di funzionamento sicura. L'insieme dei componenti software e hardware che svolgono il compito di Power Management vanno a formare un Power Stack(PS) in grado di gestire la potenza assorbita di macchine HPC.

Ci sono centri con piu exa-computer?

Mentre sono state proposte diverse soluzioni in tale argomento, la maggior parte di esse si è rivelata essere una rimedio per soddisfare singoli obiettivi di ottimizzazione o per un singolo sistema di HPC. Infatti molti dei prodotti attualmente disponibili svolgono compiti senza una visione globale e spesso in conflitto gli uni con gli altri. Peraltro non sono neanche mai state definite o modellizzate interfacce di comunicazione tra i vari software, lasciando agli amministratori dei sistemi di HPC, l'onere di farlo.

Exa-computer esiste?

[1] Supercomputer in grado che raggiunge prestazioni di ExaFlops

Ci sono termini migliori di Supercomputer?

Stato dell'arte

Un Power-Stack deve gestire e monitorare la potenza assorbita, le frequenze e le temperature di processori all'interno dei sistemi ad alte prestazioni. Questo deve poter essere fatto anche a diversi livelli come intero sistema, singoli nodi e singoli elementi all'interno dei nodi. E' quindi necessario poter accedere agli attuatori e sensori presenti nei core sia in modo diretto che da "remoto". Normalmente per farlo ci sono due vie disponibili basate su due interfacce differenti:

- in-band
- out-of-band

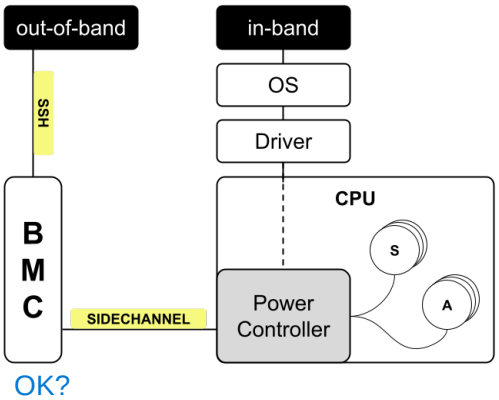


Figura 2.1. Differenza tra le due interfacce

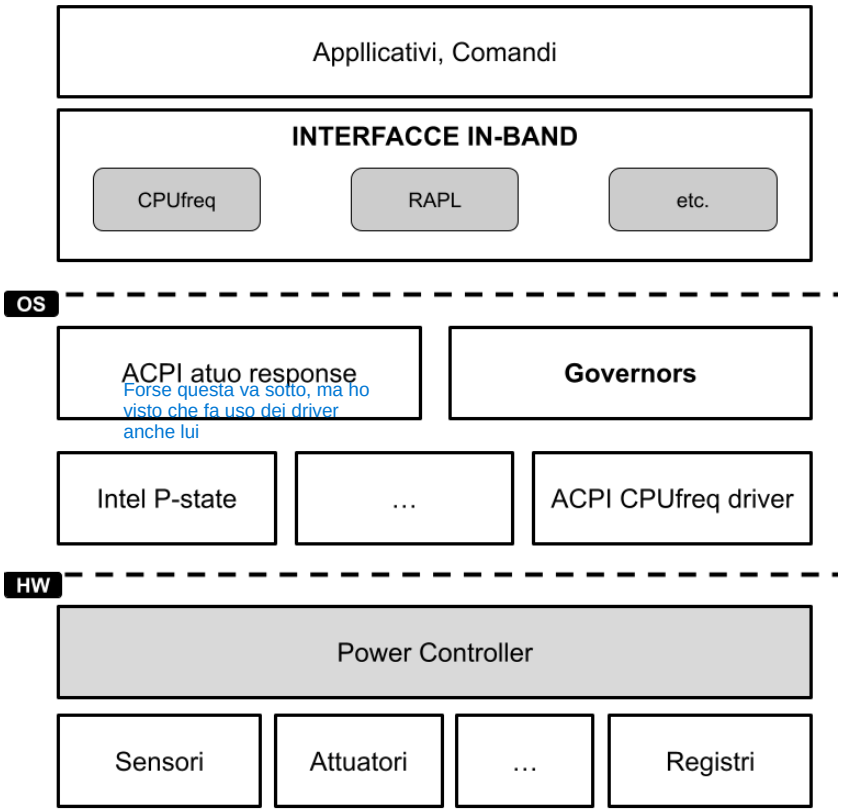
Nella figura 2.1 viene schematizzato l'accesso ai dispositivi hardware che si occupano della gestione del power management su sistemi HPC. Sarebbe in realtà possibile accedere a questi componenti anche tramite altri meccanismi specifici, ad esempio la mappatura in memoria condivisa dei componenti hardware, tuttavia a causa della loro natura altamente specializzata, tali approcci non saranno considerati nell'ambito di questa tesi.

2.0.1 Servizi in-band

I servizi in-band accedono alle risorse hardware tramite codice che esegue sul processore stesso. Questi sono resi possibili da infrastrutture come CPUfreq o RAPL che tramite dei driver, espongono a livello utente le manopole per gestire e monitorare frequenze e informazioni della cpu. Questo passaggio viene reso possibile da interfacce fornite dal sistema operativo. Queste ultime possono essere gestite in automatico in base al carico di sistema, in risposta ad eventi ACPI oppure in modo manuale. Una volta scelti

i driver come *ACPI CPUfreq driver* o *Intel P-state* è possibile scegliere tra diversi governors (o governatori) disponibili, che permettono di agire con delle policy differenti. Per esempio *CPUfreq* fornisce diversi governors per soddisfare diversi tipi di situazioni, come:

- performance: forza la CPU ad eseguire alla frequenza massima disponibile;
- powersave: forza quella minima;
- ondemand: comportamento dinamico in base all'utilizzo di sistema;
- userspace: permette ai selezionati user-space di impostare la frequenza;
- conservative: come ondemand ma con più inerzia al cambiamento;



Questo schema è un pò difficile da fare, siccome ci sono tante configurazioni possibili. Cosi può andare?

Figura 2.2. Struttura interfacce in-band: divise su più livelli tra cui Sistema Operativo (SO) e Hardware (HW)

Il vantaggio di usare queste interfacce è che permettono di operare in Real-Time ed in modo dinamico. I lati negativi invece risiedono nelle stesse peculiarità di questi

strumenti, ovvero che possono ottenere solo le informazioni dei core sui quali i processi vengono eseguiti.

2.0.2 Servizi out-of-band

Contrariamente alle interfacce in-band, i servizi out-of-band fanno utilizzo di *sidechannels* ovvero canali di accesso alternativi per ottenere i dati richiesti. Questo meccanismo permette a processi esterni[1] di accedere alle informazioni contenute nel processore che si vuole analizzare. Per di più questo permette di monitorare i componenti anche quando ci sono errori ed eccezioni che normalmente bloccherebbe il servizio. Un componente tra i più importanti che svolge questa funzione è il Baseboard Management Controller (BMC), solitamente un micro-controllore animato da sistemi embedded linux, e accessibile tramite un canale separato (solitamente provvisto di una propria interfaccia di rete e/o bus specifici). Il suo principale scopo è quello di monitorare in modo dettagliato lo stato di tensioni, temperature, ventole e prestazioni dei processori e fornire contemporaneamente servizi di power capping sia a livello di sistema (non possibile tramite le interfacce in-band) che di singoli processori. Recentemente alcuni produttori di BMC introducono anche dispositivi FPGA da affiancare al BMC per aumentarne la flessibilità e le prestazioni. E' corretta la parte delle FPGA?

[1] In esecuzione su processori diversi dai quali si vuol reperire dati

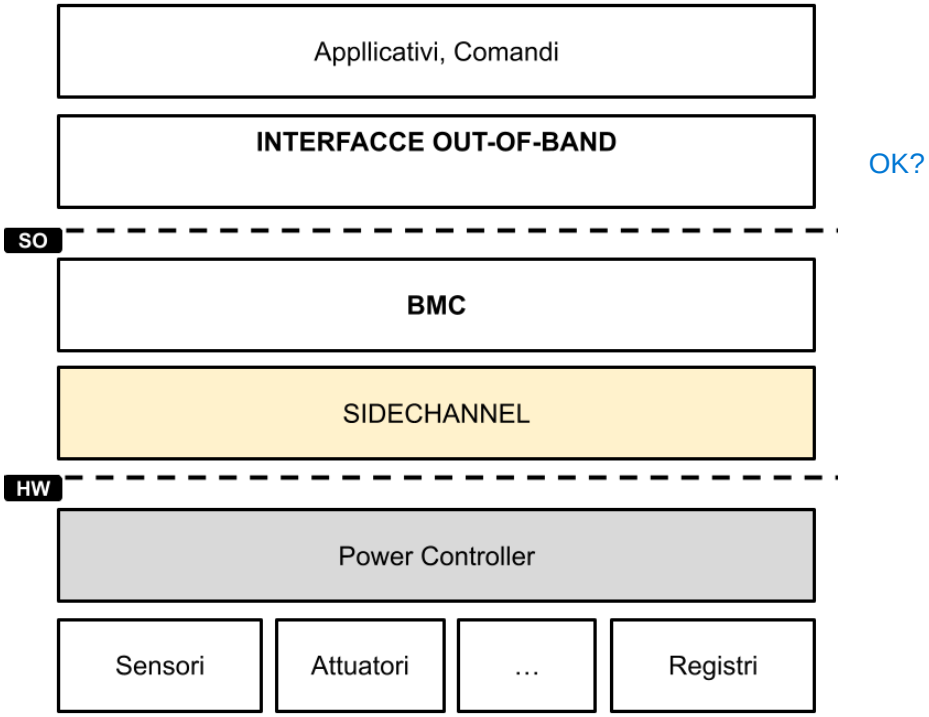


Figura 2.3. Struttura interfacce out-of-band

2.1 Interfacce di alto livello

Nel corso degli anni con l'obbiettivo di ottimizzare e automatizzare l'interazione con questi meccanismi hardware sono stati sviluppati diversi software di più alto livello che utilizzano sia interfacce in band che interfacce out of band. Si possono ricordare i più famosi: *Variorum* (LLNL), *GEOPM* (Intel)[5], e *HDEEM* (Atos)[6]. Tutti questi rappresentano però un tentativo di fornire una soluzione ad un sottoinsieme di problemi per la gestione dell'energia o potenza, piuttosto che ad un software con visione globale di Power Management per sistemi di calcolo ad alte prestazioni.

2.2 Modello di power stack

Con Power-Stack si intende un insieme di software che cooperando riescono a fornire ad applicazioni, utenti e amministratori gli strumenti per un servizio completo di Power Management. Una volta definito il problema, e i componenti che possono essere utilizzati, è possibile definire un modello di interazione e responsabilità dei vari attori.

Di seguito vengono riportati quelli che sono i ruoli necessari al fine di coordinare un sistema HPC dall'allocazione di un applicativo, fino alla gestione delle tensioni.

- Workflow engine (WE)
- System Power Manager (SPM)
- Job Manager (JM)
- Job Scheduler (JS)
- Resource Manager (RM)
- Node Manager (NM)
- Monitor (M)

Qui ho fatto un pò di modifiche in base a quello che mi sembrava più intuitivo. Ho scomposto il System Manager nei sui componenti (SPM, JS, RM), sono giusti?

2.2.1 Workflow engine

Il workflow o *flusso di lavoro* è un insieme task che devono essere svolti per risolvere un determinato problema. Il workflow engine si occupa di analizzare le dipendenze e le richieste di risorse di ogni workflow e decide dinamicamente come dividerlo nei jobs che verranno successivamente assegnati al Job Scheduler.

2.2.2 System Power Manager

Il System Power Manager si occupa di comunicare con tutti i Node Manager all'interno del sistema, per impostare eventuali limiti di potenza. Questi ultimi vengono solitamente impostati manualmente dagli amministratori di sistema, oppure in modo automatico comunicando con altri attori, come monitor e NM. Una volta fissati i limiti, vengono monitorati i dati relativi a potenza ed energia, e controlla di conseguenza i budget, e la *user-fairness*.

2.2.3 Job scheduler

Il job scheduler dopo aver ricevuto come input un insieme di jobs li schedula all'interno del sistema, e in modo indicativo decide quando schedulare ogni job, su quale nodo, e con quale power budget. In particolare la serie di compiti che si trova a svolgere è il seguente:

1. L'utente schedula i jobs da svolgere in una o più code, definite dal Workflow Engine?. Corretto?

2. Il Job scheduler esamina tutte le code e i job in esse contenute, e decide dinamicamente, quale sarà l'ordine di esecuzione, e il tempo massimo in cui viene assegnata una risorsa.

Generalmente si cerca di ottimizzare alcune caratteristiche come il tempo di utilizzo del sistema oppure l'accesso veloce alle risorse per alcuni sottoinsiemi di jobs. Inoltre le code definite, possono avere diverse priorità o può essere ristretto l'accesso a soli alcuni utenti. I job scheduler possono condividere un nodo anche con più utenti contemporaneamente, in base all'utilizzo che devono farne. Per farlo il nodo viene allocato e diviso in partizioni virtuali, che vengo "sciolte" una volta finiti i job in esecuzione. Questo permette di utilizzare al massimo i componenti messi a disposizione dal sistema HPC.

2.2.4 Resource Manager

Per riuscire a svolgere questo lavoro il Job Scheduler interagisce con uno o più Resource Manager. Questi sono software che hanno il compito di dividere (o condividere) le risorse computazionali e fisiche del sistema HPC, ai vari utenti che lo utilizzano. Queste risorse includono diversi componenti:

- Nodi
- Processori
- Memorie
- Dischi
- Canali di comunicazione (compresi quelli di I/O)
- Interfacce di rete

Per esempio quando un Job Scheduler deve eseguire un job, richiede al RM di allocare core, memorie, dischi e risorse di rete in base alle specifiche di esecuzione del job. Infine in alcuni casi il RM è anche responsabile di gestire la distribuzione elettrica e raffreddamento di alcune parti dei centri di calcolo[7].

2.2.5 Job Manager

Lo scopo del job manager è quello di effettuare ottimizzazioni job-centriche considerando le prestazioni di ogni applicazioni, il suo utilizzo di risorse, la sua fase e qualsiasi interazione dettata da ogni workflow in cui è presente. In breve il job manager decide i target delle manopole del Power Management, come (i) CPU power cap, (ii) CPU clock frequency oltre ad eseguire ottimizzazione del codice.

2.2.6 Node Manager

Il node manager fornisce accesso ai controlli e monitoraggio hardware a livello del nodo. Volendo permette anche di definire delle policy di power management. Ha infine lo scopo di preservare integrità, sicurezza del nodo sia in termini informatici che fisici.

2.2.7 Monitor

Il monitor è responsabile di collezionare tutte le metriche in-band e out-of-band che riguardando prestazioni, utilizzo e stato delle risorse, potenza ed energia. Tutto questo deve essere fatto con il minor impatto possibile sul sistema dove sta agendo, collezionando, aggregando e analizzando le metriche e dove necessario, scambiandole ad altri attori. A sua volta il *Monitor* è scomponibile in tre sotto-moduli:

- Gestione Firma che genera una firma che identifica univocamente il job;
- Estimatore che valuta le proprità dei job o dello stato del sistema usando la firma generata precedentemente;
- Dashboard che fornisce le funzionalità da mostrare agli sviluppatori.

Per concludere viene mostrato uno schema in figura 4.1a che vuole mostrare la gerarchia e le possibili interazioni tra i vari attori.

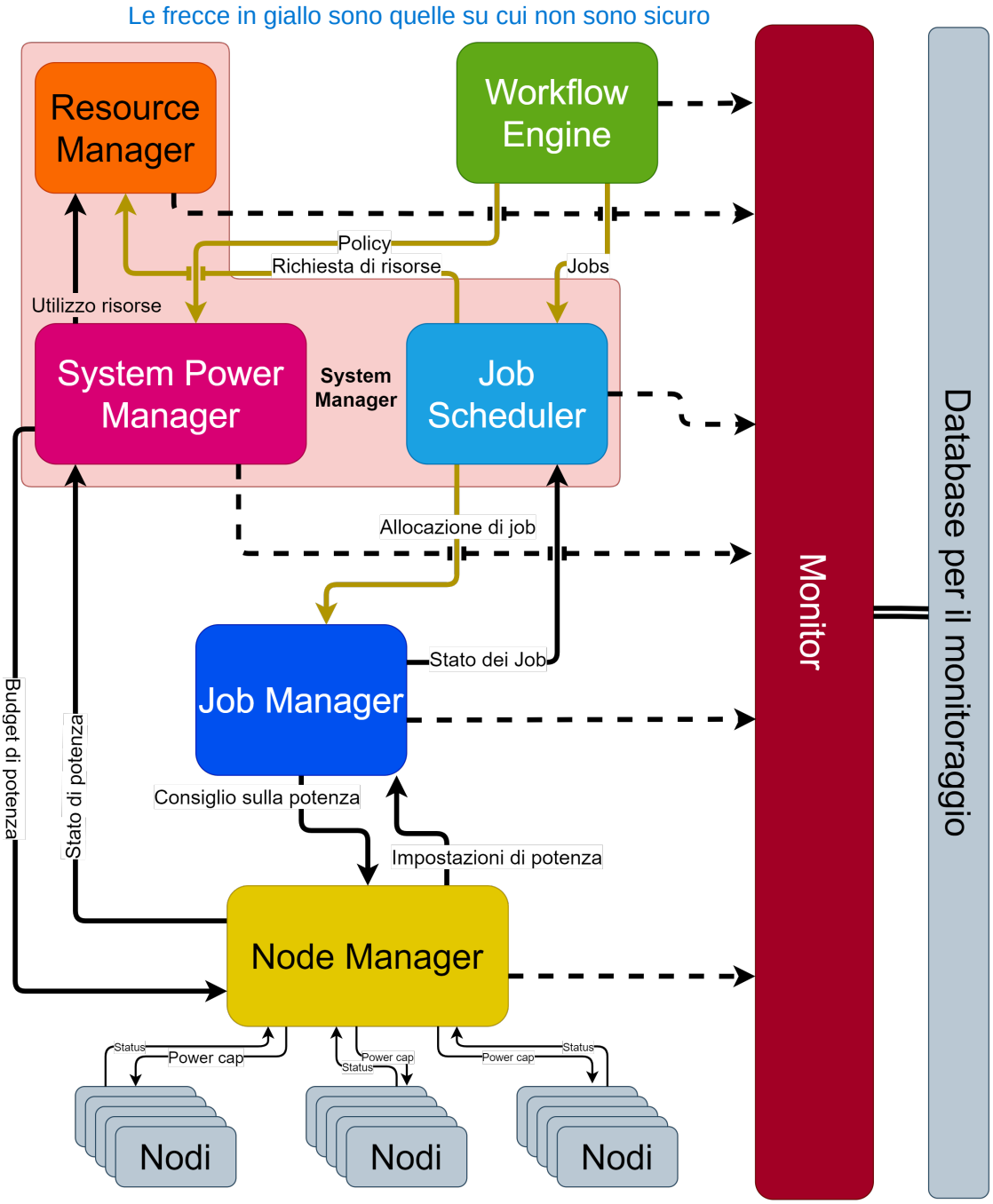


Figura 2.4. Modello di power stack

DDS & RTPS

Data Distribution Service (DDS)[8] e Real-Time Publish-Subscribe (RTPS)[9] sono strumenti che rivestono un ruolo importante nelle comunicazioni in sistemi distribuiti e Real-Time. Infatti, anche se a diversi livelli, si occupano della la trasmissione di dati tra applicazioni e dispositivi interconnessi, giocando un ruolo importante in scenari con diverse centinaia di attori come i sistemi ad alte prestazioni come HPC.

3.1 Implementazione usasta

In particolare, DDS e RTPS sono due differenti protocolli di comunicazione che accoppiati forniscono i servizi sopracitati . Ci sono state diverse implementazioni di questi protocolli da diversi società e organizzazioni, come:

- FastDDS (eprosima)
- CycloneDDS (Oracle)
- ConnextDDS
- GurumDDS

In tutti i successivi capitoli verrà preso come riferimento FastDDS ed in particolare la sua versione 2.11.2 [10]. E' stato scelto di utilizzare questa implementazione dato il supporto per le comunicazione Real-Time, e la vasta possibilità di impostazioni delle Qualità del servizio(QoS) che la rendevano adeguatamente configurabile per un utilizzo su sistemi di HPC.

3.2 DDS

Data Distribution Service è un protocollo incentrato sullo scambio di dati per sistemi distribuiti. Questo si basa su modello chiamato Data-Centric Publish Subscribe (DCPS). I principali attori che vengono coinvolti sono:

- Publisher: responsabile della creazione e configurazione dei DataWriter. Il DataWriter è l'entità responsabile della pubblicazione effettiva dei messaggi. Ciascuno avrà un Topic assegnato sotto il quale vengono pubblicati i messaggi;
- Subscriber: responsabile di ricevere i dati pubblicati sotto i topic ai quali si iscrive. Serve uno o più oggetti DataReader, che sono responsabili di comunicare la disponibilità di nuovi dati all'applicazione;
- Topic: collega i DataWriter con i DataReader. È univoco all'interno di un dominio DDS;
- Dominio: utilizzato per collegare tutti i publisher e subscriber appartenenti a una o più domini di appartenenza, che scambiano dati sotto diversi topic. Il DomainParticipant funge da contenitore per altre entità DCPS, e svolge anche la funzione di costruttore di entità Publisher, Subscriber e Topic fornendo anche servizi di QoS;
- Partizione: costituisce un isolamento logico di entità all'interno dell'isolamento fisico offerto dal dominio;

Inoltre DDS definisce le cosiddette Qualità di Servizio (QoS policy) che servono configurare il comportamento di ognuno di questi attori.

3.3 RTPS

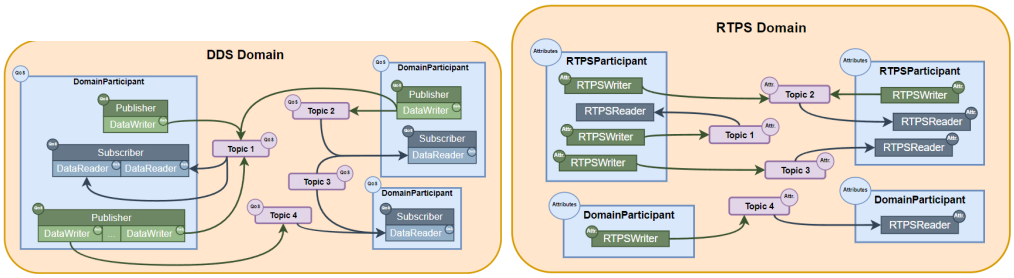
Real-Time Publisher Subscribe è un middleware[1] utilizzato da DDS per gestire la comunicazione su diversi protocolli di rete come UDP/TCP e Shared Memory. Il suo principale scopo è quello di inviare messaggi real-time, con un approccio best-effort e cercando di massimizzare l'efficienza. E' inoltre progettato per fornire strumenti per la comunicazione unicast e multicast. Le principali entità descritte da RTPS sono:

- RTPSWriter: endpoint capace di inviare dati;
- RTPSReader: endpoint abilitato alla ricezione dei dati;

Ereditato da DDS anche RTPS ha la concezione di Dominio di comunicazione e come questo, le comunicazioni a livello di RTPS girano attorno al concetto di Topic prima definito. L'unità di comunicazione è chiamata **Change** che rappresenta appunto un cambiamento sui dati scritti sotto un certo topic. Ognuno degli attori registra questi *Change* in una struttura dati che funge da cache. In particolare la sequenza di scambio è:

[1] Formalmente è un protocollo a se stante, utilizzato da DDS come middleware

1. il *change* viene aggiunto nella cache del RTPSWriter;
2. RTPSWriter manda questa *change* a tutti gli RTPSReader che conosce;
3. quando RTPSReader riceve il messaggio, aggiorna la sua cache con il nuovo *change*.



(a) Schema DDS **(b)** Schema RTPS

Figura 3.1. Confronto tra architettura DDS e RTPS

3.4 ROS

Questo approccio distribuito per lo scambio dei dati tra vari attori utilizzando un middleware basato su DDS non è idea inedita. Infatti dalla sua seconda versione il software open-source **Robot Operating System**[11] meglio conosciuto come ROS ha deciso di usare questi strumenti introducendo un ulteriore livello che permette di usare diverse implementazioni di DDS. Questa idea è stata e sarà di grande ispirazione per il completamento di questo progetto. Nello specifico, è stata creata una libreria chiama rmw_dds_common(ros middleware) come mostrato nella 3.2[2] sopra il quale la community ha creato le implementazioni di DDS desiderate, andando a creare così diverse possibilità di configurazione dello stesso servizio di distribuzione dati. Inoltre per cambiare tra le diverse versioni di DDS si deve semplicemente impostare una variabile di ambiente, rendendo il procedimento facile per tutti i possibili fruitori di ROS2.

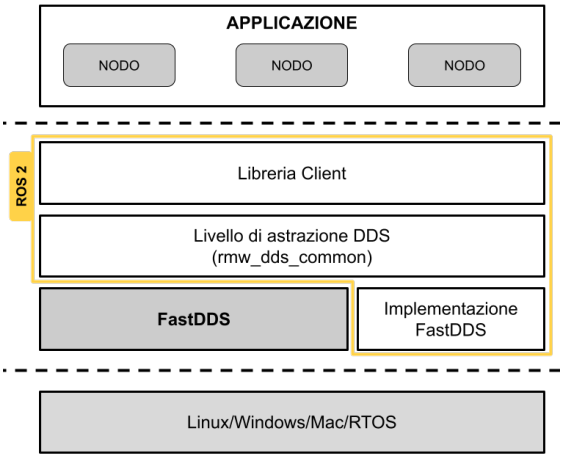


Figura 3.2. Ros Middleware per DDS

[2]L'implementazione di FastDDS vuole essere un esempio tra le diverse soluzioni disponibili per ROS2

Tuttavia è stato scelto di utilizzare direttamente l'implementazione DDS invece che *Ros middlware* per i seguenti motivi:

- Potenzialità: ROS mette a disposizione solo alcuni degli strumenti resi disponibili dallo strato DDS, andando a limitare la possibilità di sfruttamento di tutte le impostazioni e QoS di FastDDS;
- Granularità: oltre alla mancanza di alcune funzionalità, in ros sono predisposti dei pacchetti preconfigurati di entità. Per andare a studiare più approfonditamente gli strumenti su architetture diverse, è più pratico avere la possibilità di cambiare ogni piccola configurazione.
- Flessibilità: Per andare a definire delle strutture dati di ROS, al fine di scambiare messaggi DDS usando il middleware offerto, era necessario creare diverse strutture dati che combaciassero con le interfacce ROS;
- Diversa natura: In ROS, la maggior parte delle comunicazioni fatte sono intra-processo e raramente viene utilizzata una infrastruttura di rete, per andare a collegare tutti i nodi. Per questo sono maggiormente ottimizzate quel tipo di comunicazioni, piuttosto che quelle su rete, come in questo caso.
- Facilità di implementazione: Implementare completamente *rmw_dds_common* richiedeva un impegno e uno studio non indifferente della architettura sottostante a ROS, che seppur ben documentata, sarebbe costata diverso tempo.

Ne tiro via un pò?

REGALE

REGALE[12] è un progetto finanziato dall'UE[13] nato ad Aprile 2021 che opera nell'ambito del Power Management in sistemi ad High-Performance Computing ed in particolare si è focalizzato su sistemi Exascale[1].

4.1 Obbiettivi

Il loro principale obbiettivo è aprire la strada alla prossima generazione di applicazioni dell'HPC, riunendo importanti parti interessate, accademici e centri europei di supercalcolo. Il progetto si pone di definire un'architettura open-source con l'intenzione di costruire un prototipo in grado di dotare i sistemi di HPC dei meccanismi e delle politiche necessari per garantire un utilizzo delle risorse efficace[13]. Per farlo sono state definite delle parole chiave che si è imposto di rispettare durante lo sviluppo di tutto il progetto:

- Effettivo utilizzo delle risorse disponibili, ottenibile tramite miglioramenti delle performance delle applicazioni, aumento del throughput del sistema, e la minimizzazione della *Performance Degradation* sotto vincoli di potenza;
- Ampia applicabilità ottenibile attraverso l'inseguimento di concetti come scalabilità, indipendenza dalle piattaforme ed estensibilità;
- Facilità di implementazione ottenibile tramite la creazione di una infrastruttura flessibile, e che gestisca in automatico le risorse.

4.2 Power Stack

L'intero progetto, durante il suo sviluppo si è basato su strumenti come MPI library[14], SLURM[15], o DCDB[16]. Inoltre, ha deciso di considerare l'introduzione di molti

[1] Exascale: capace di eseguire operazioni nell'ordine di ExaFlops (10^{18})

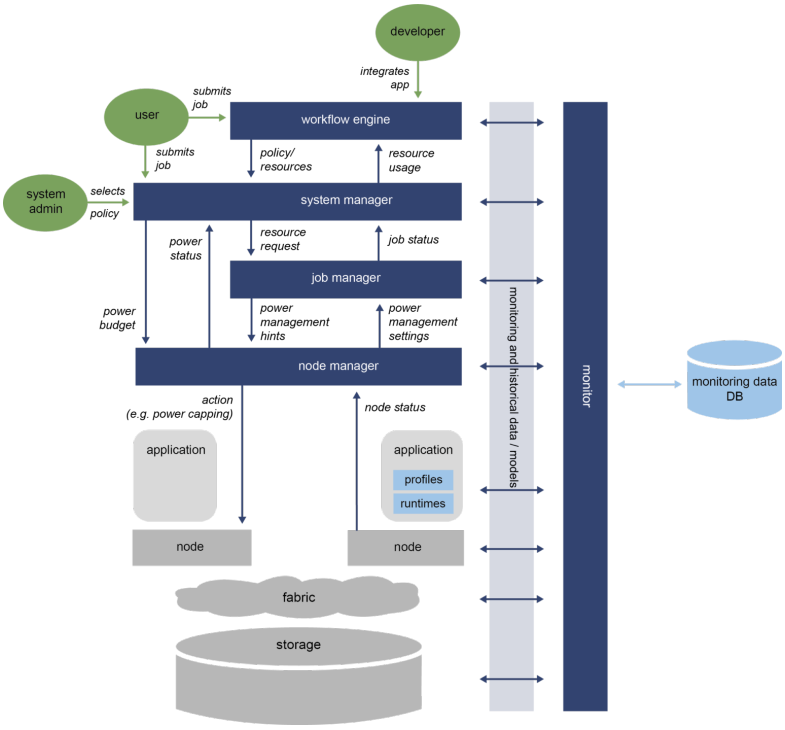
software open-source che potessero soddisfare le esigenze modello di Power Stack 4.1a. Infatti sono stati valutati e selezionati diversi applicativi (molti dei quali prodotti dai partner, come mostrati in tabella 4.1) anche con ruoli analoghi, per soddisfare a diverse esigenze.

Tool	**Partner**	**Ruolo all'interno di REGALE**
SLURM	TUM	System Manager
OAR	UGA	System Manager
DCDB	LRZ	Monitor, Monitoring Data
BEO	ATOS	Monitor, Node Manager, Monitoring Data
BDBO	ATOS	Monitor, Job Manager
EAR	BSC	Monitor, Node Manager, Job Manager, Monitoring Data
Melissa	UGA	Workflow Engine
RYAX	RYAX	Workflow Engine
Examon	E4/UNIBO	Monitor, Monitoring Data
COUNTDOWN	CINECA/UNIBO	Job Manager
PULPcontroller	UNIBO	Node Manager
BeBiDa	RYAX	System Manager

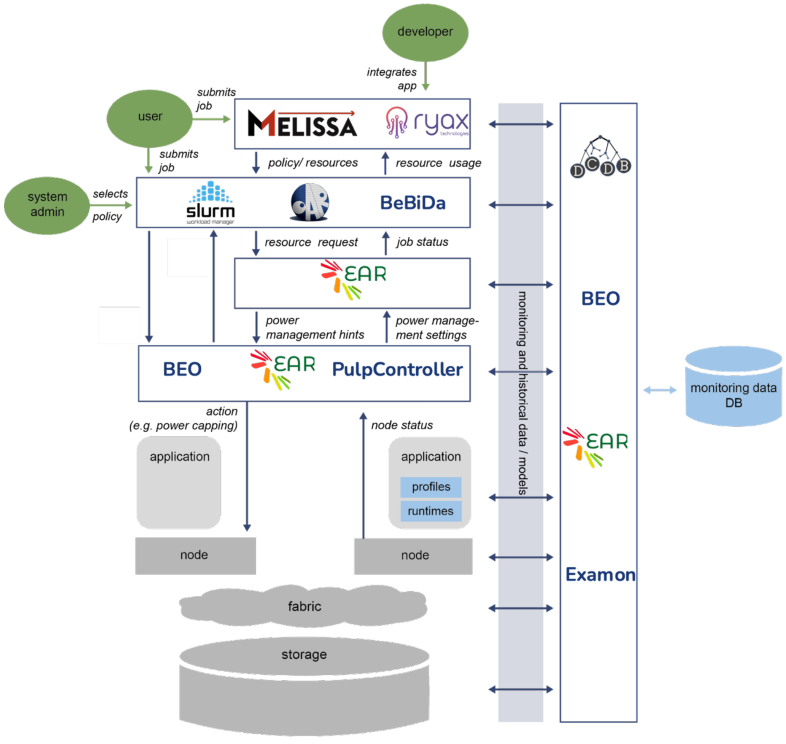
Tabella 4.1. Software introdotti all'interno di REGALE con il partner che li ha prodotti e il loro ruolo

4.3 Integrazione

Vista la natura dei software introdotti nel progetto, non era previsto che questi potessero comunicare tra di loro, in quanto nati per essere usati singolarmente. Serviva perciò uno strumento che fosse in grado di far comunicare due a due ogni attore del power stack da loro introdotto. tto questo pretesto si è scelto di testare varie soluzioni, tra cui anche quella di un **middlware DDS**.



[H] **(a)** Modello di Power Stack Regale



(b) Copertura componenti

Figura 4.1. Descrizione delle immagini

Test

Il tema principale di questa tesi, è stata quelli di generare un modello, analizzare e alla fine implementare quello che potrebbe essere l'infrastruttura sulla quale tutti gli attori di un Power-Stack possano comunicare in modo completamente distribuito tramite DDS. Per poter creare l'infrastruttura necessaria, sono stati utilizzati sistemi di High-Performance Computing sui quali andare a provare empiricamente i vari esperimenti. Per supportare questo lavoro, sono stati resi disponibili due supercomputer uno da Cineca[17] e uno da E4[18] nel tentativo di ottenere risultati affidabili. Di seguito le specifiche dei sistemi utilizzati:

Parameter	**Cineca**	**E4**
Number of nodes used	3	
Processor	Intel CascadeLake 8260 S	
Number of sockets per node	2	
Number of cores per socket	24	
Memory size per node	384 GB	
Interconnect	Mellanox Infiniband 100GbE	
OS	CentOS Linux	
MPI	Open MPI 4.1.1	

Non mi hanno ancora dato tutti i ~~dati~~, poi aggiorno

Tabella 5.1. Tabella hardware dei sistemi utilizzati

5.1 Strumenti utilizzati

I test effettuati in questa sezione sono stati generati da diversi tipi di componenti ognuno di essi con uno o più compiti specifici, in modo da avere un discreto controllo sull'avan-

zamento e la gestione dei dati. Nella figura 5.1 viene riportato uno schema riassuntivo di tutte le tecnologie utilizzate.

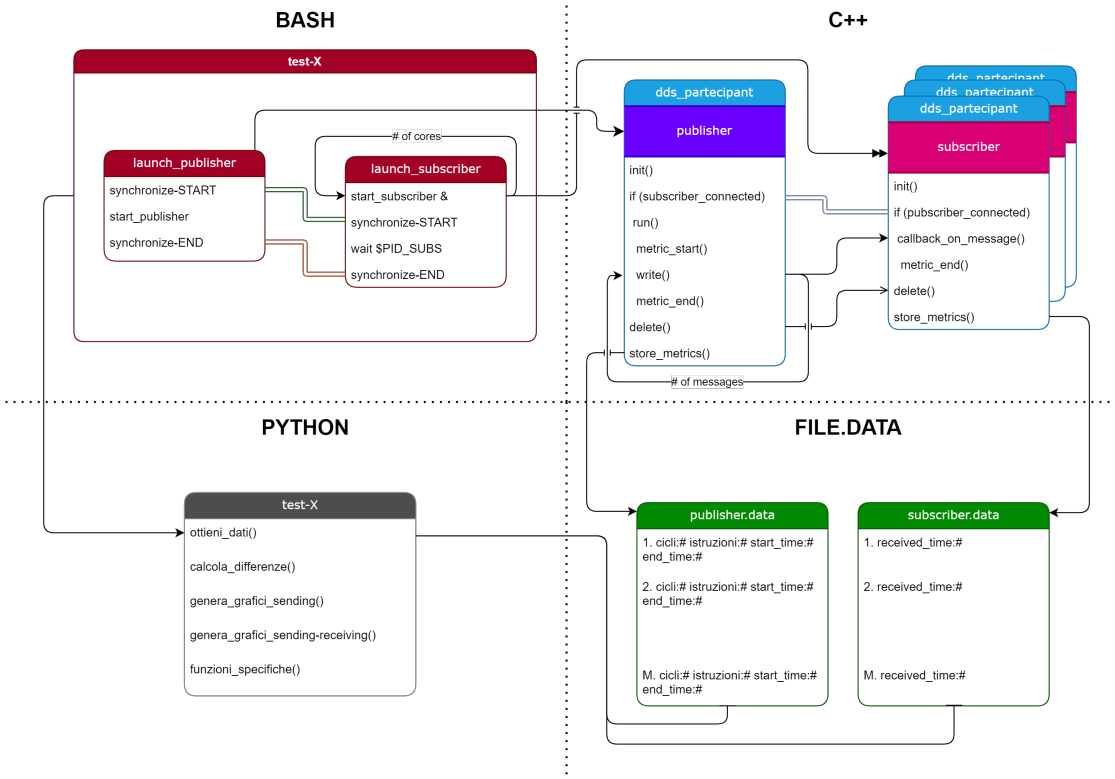


Figura 5.1. Struttura test

5.1.1 Bash

Vista la necessità di lanciare diversi publisher e diversi subscriber ogni volta con dei parametri variabili è stato conveniente usare programmi di scripting come Bash. Infatti questi gestivano i diversi parametri da passare agli attori, inizializzavano le variabili d'ambiente, decidevano quali core dovevano essere utilizzati da ogni partecipante (taskssetaffinity) e mantenevano sincronizzati i test per evitare che alcuni attori fossero inizializzati troppo presto. Infine ripulivano e ordinavano i dati una volta terminato i test ed andavano ad eseguire gli script python che processavano i dati, nelle cartelle corrette.

5.1.2 C++

E' stato scelto di realizzare una unica implementazione publisher e subscriber dove i valore delle funzionalità che si volevano testare dovevano essere passati come parametro lato bash (5.1.1) in modo da poter avviare tutti i test con gli stessi codici, renden-

do più semplice la gestione dei diversi test, e più robusto ad errori dovuti a diverse configurazioni.

Struttura

Per scambiarsi dei messaggi all'interno di infrastruttura basata su DDS, sono necessari: (i) un topic, (ii)un publisher ed (iii) un subscriber. Inoltre nel topic è necessario definire il tipo dato o struttura di dati che si va a scambiare. La struttura che si è scelta di utilizzare per i test è stata la seguente:

```
struct DDSTest
{
    unsigned long index;
    std::string message;
};
```

Dove index era necessario per definire una corrispondenza stretta tra i messaggi inviati e quelli ricevuti, mentre la stringa era comoda per definire un oggetto di dimensione molto variabile (anche dinamicamente durante i test).

Una volta studiata la documentazione ufficiale di eProsima FastDDS, è stato sviluppato un codice in grado di integrare tutte le funzionalità di DDS ed alcuni strumenti per l'ottenimento di metriche precedentemente concordate con Cineca[17]. Nello specifico sono state scelte:

- Tempo di invio
- Istruzioni Perf-Event per invio
- Cicli TSC (read_tsc) per invio
- Tempo di invio e ricezione

5.1.3 Lettura TSC

Il Time Stamp Counter, è un registro a 64 bit, presente nella maggior parte dei processori moderni. Il registro fornisce informazioni sul tempo, in termini di cicli di clock del processore, e viene spesso utilizzato per effettuare misure di queste metriche. Per leggere questo valore, che viene fatto prima e dopo l'istruzione da misurare, è necessario eseguire la seguente istruzione:

```
unsigned int lo, hi;
__asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
return ((uint64_t)hi << 32) | lo;
```

rdtsc è l'istruzione assembly per leggere il registro Timestamp Counter, =a (lo) e =d (hi) sono i vincoli di output che specificano come i risultati dell'istruzione rdtsc devono essere restituiti al programma. In lo ("=a") viene riportat il valore a 32 bit meno significativo ed in hi, il valore a 32 bit più significativo.

5.1.4 Conteggio istruzioni

Per le istruzioni invece è stato usato lo strumento **Perf**, un software offerto da Linux ed incluso anche nel suo kernel, per la profilazione delle performance tramite i *performance_counter*. Questa suite è estremamente avanzata e permette di ottenere delle metriche specifiche senza troppa difficolta. In questo caso è stata usata una chiamata alla libreria **perf_event** nel codice per il valore *PERF_COUNT_HW_INSTRUCTIONS*.

5.1.5 Ottenimento dei tempi

In sistemi molto complessi come può essere considerato un supercalcolatore, la gestione degli orologi è tutt'altro che banale. Infatti dopo aver deciso una tra le tante politiche di sincronizzazione disponibile come centralizzata, distribuita, GPS e tante altre, è necessario applicarle e continuare a tenere questi orologi sullo stesso tempo assoluto. Nei sistemi utilizzati in questo progetto, ed in particolare nei sistemi 5.1, lo strumento adottato è Network Time Protocol (fornito da ntpd). Quest'utlimo ogni intervallo di tempo impostato, va a rendere disponibile ai vari nodi ed ai rispettivi orologi locali un tempo di riferimento che ha la funzione di punto fisso. Questo intervallo è normalmente fissato ogni 1024s, ma non disponendo dei diritti necessari ad utilizzarlo, non mi è stato possibile recuperare l'informazione.

Detto questo, per ottenere le differenze di tempi su sistemi Linux, è ricorrente utilizzare una funzione chiamata **clock_gettime()** che restituisce il tempo istantaneo alla chiamata. Se si esegue la differenza tra due diverse clock_gettime(), si ottiene il tempo trascorso tra queste due. Per questa funzione è possibile ottenere diverse metriche tra cui:

- CLOCK_REALTIME: ottiene il tempo assoluto, sincronizzato dei vari sistemi da ntpd
- CLOCK_MONOTONIC: ottiene un tempo relativo, da un punto non preciso dall'avvio del sistema
- CLOCK_MONOTONIC_RAW: come sopra, ma non influenzato da ntpd
- CLOCK_PROCESS_CPUTIME_ID: timer dei processori ad alta risoluzione
- CLOCK_THREAD_CPUTIME_ID: tempo dei thread dei processori

Tra queste è stato utilizzato il MONOTONIC, visto che il REALTIME con aggiornamenti di ntpd di 1024s subisce variazioni di alcuni millisecondi[19], di gran lunga superiore all'ordine di grandezza da misurare (microsecondi, a volte anche nanosecondi). Inoltre dato che per eseguire i test è stato usato un solo sistema per volta (o Cineca, o E4) MONOTONIC_RAW non era necessario (anche in caso di aggiornamento ntp, viene diffuso in egual modo su tutti i nodi). Il problema di usare la MONOTONIC, è che su sistemi con orologi diversi, questi sono sfasati di diverse migliaia di secondi. E' stato necessario aggirare questo problema, e per farlo sono stati usati 2 approcci completamente diversi:

- Sincronizzazione dei nodi
- RTT

Entrambi i tentativi verranno approfonditi nelle successive sezioni

UML

Lo schema UML di funzionamento degli attori DDS è riassunto e schematizzato dalla figura 5.2

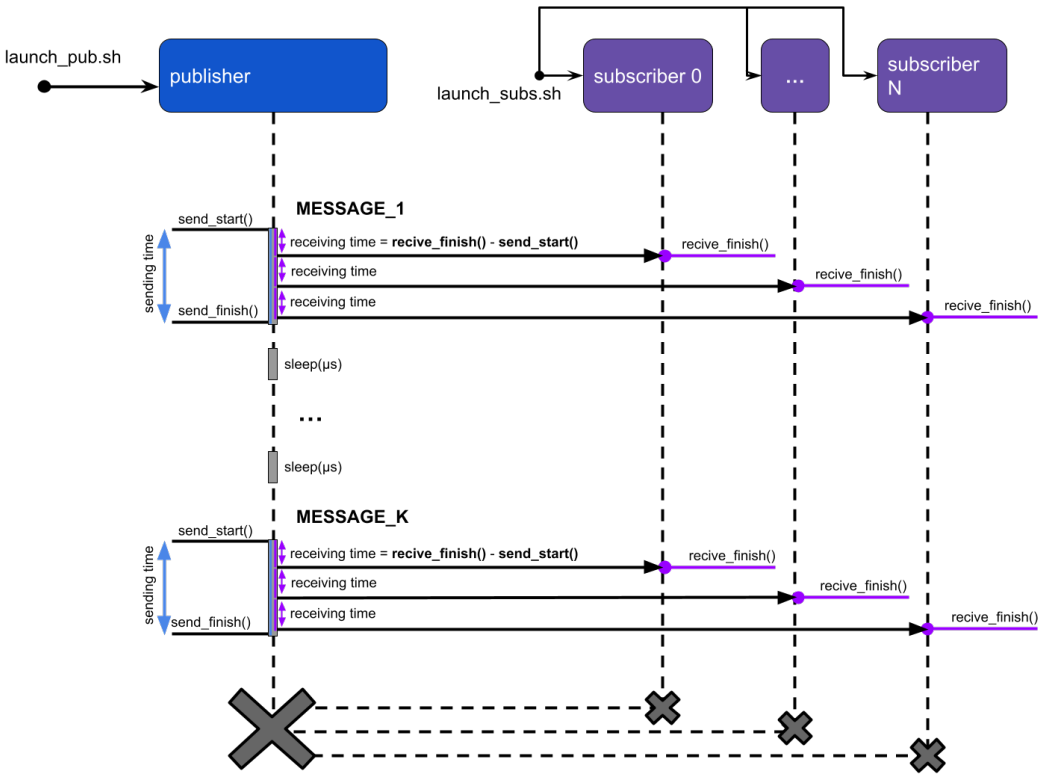


Figura 5.2. Schema UML

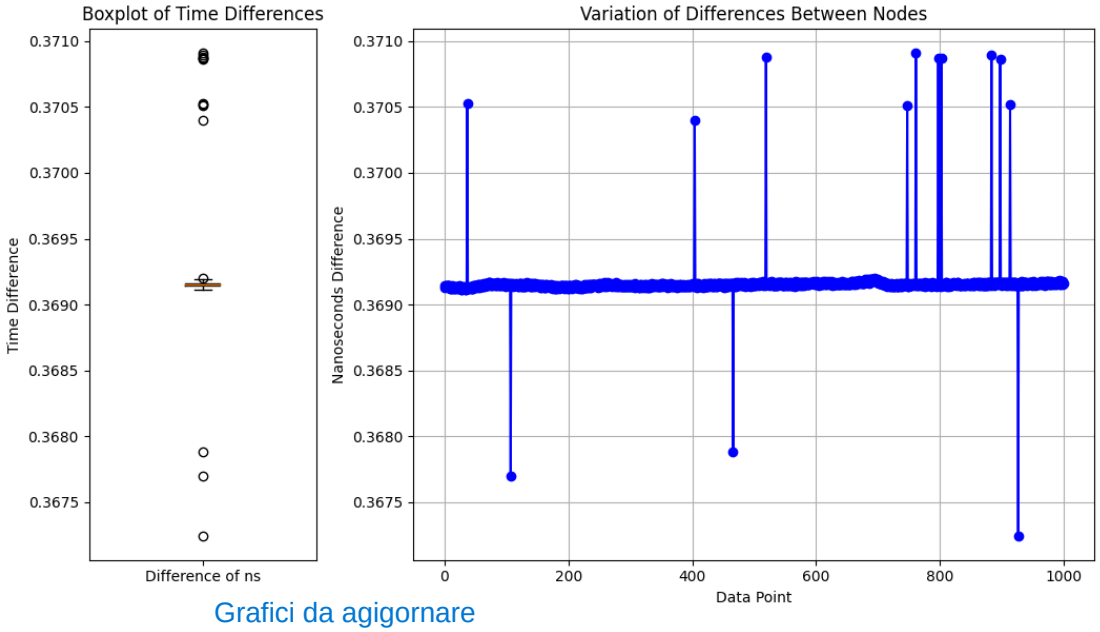
In particolare nel Publisher 3.2 prima e dopo la chiamata a funzione di write() si sono presi i valori tempo-invio, istruzioni, e TSC, mentre al lato ricevente, di Subscriber 3.2 è stato preso il tempo al momento dell'arrivo del messaggio. Segue uno schema uml della base di ognuno dei test.

5.2 DataMiners

Considerando che ogni publisher genera 10 mila messaggi da inviare a tutti e 48 i subscriber, per ogni protocollo di trasporto, e in alcuni casi in partizioni differenti si è arrivato ad avere per ogni test fino a $1'960'000$ messaggi scambiati e le relative metriche per ogni messaggio da processare. Gli script python sono stati utili a organizzare e processare tutti i dati prodotti dai vari test. Inoltre sono stati fondamentali per poter generare tutti i grafici che sono stati in questa tesi.

5.3 Sincronizzazione

Una delle prime soluzioni che è stata provata, è stata quella di sincronizzare gli orologi locali dei diversi nodi tramite l'utilizzo di librerie sviluppate per la programmazione parallela come Message Passing Interface (MPI). Quest'ultimo è un protocollo di comunicazione molto utilizzato nei sistemi HPC per la programmazione parallela. Nello specifico è stata utilizzata una funzionalità chiamata MPI_barrier, che permette di bloccare i processi, fino all'arrivo di un punto in comune, dopo il quale tutti procedono insieme. Questo serve per sincronizzare i processi tra di loro, ma non è ideata nello specifico per sincronizzare gli orologi. Gli strumenti strumenti utili alla mera sincronizzazione dei tempi dei nodi sono altri, come il già citato Network Time Protocol, ma essendo ntpd un servizio di amministrazioni non è stato possibile interagirci e quindi usarli. Nella figure 5.3 **?? ??** sono stati riportati i dati ottenuti grazie a questo meccanismo.



Grafici da agigornare

Figura 5.3. Scostamento del tempo su nodi diversi

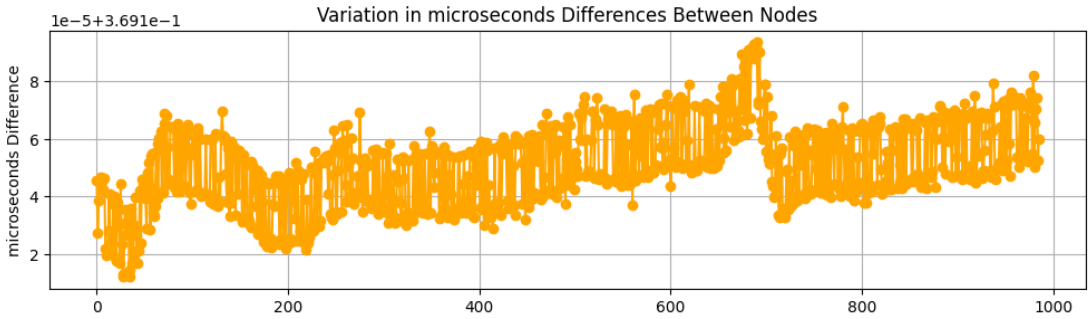


Figura 5.4. Scostamento senza outliers

Come possibile notare nella figura 5.3 nonostante le mpi_barrier, gli scostamenti di tempo tra 2 nodi durante i diversi tentativi effettuati (1000), cambia notevolmente, arrivando a differenze fino ad un massimo di 290 microsecondi. Questo rende il metodo appena mostrato utilizzabile solo nel caso in cui non sia necessario sapere il tempo assoluto di qualche azione, ma le varianze, che rimarrebbero costanti se si utilizzano sempre gli stessi nodi.

5.4 RTT

Nonostante la sincronizzazione, fosse idealmente il metodo più preciso per ottenere i tempi di invio-rivezione, essendo l'errore possibile dello stesso ordine di grandezza dei tempi di ricezione, per alcuni test si è utilizzato un approccio che non richiedesse sincronizzazione. Il metodo più intuitivo è utilizzare il Round Trip Time (RTT). Il RTT è un metrica che viene solitamente utilizzata per misurare la latenza di una rete, e si basa sull'idea di calcolare il tempo che intercorre tra lìinvio di un segnale e la ricezione della conferma di arrivo dello stesso. Ovviamente il valore ottenuto risulta nel caso ideale più che raddoppiato vista la necessità di un messaggio di risposta. Nel diagramma 5.2 non sarebbe stato possibile condurre questa misura, perchè un subscriber non può inviare un messaggio di risposta. Per farlo è stato necessario rivedere gli attori coinvolti, ed introdurre in quello che prima venivano chiamati publisher e subscriber, un publisher e un subscriber a testa. Per semplificarne la comprensione viene riportato lo schema modificato:

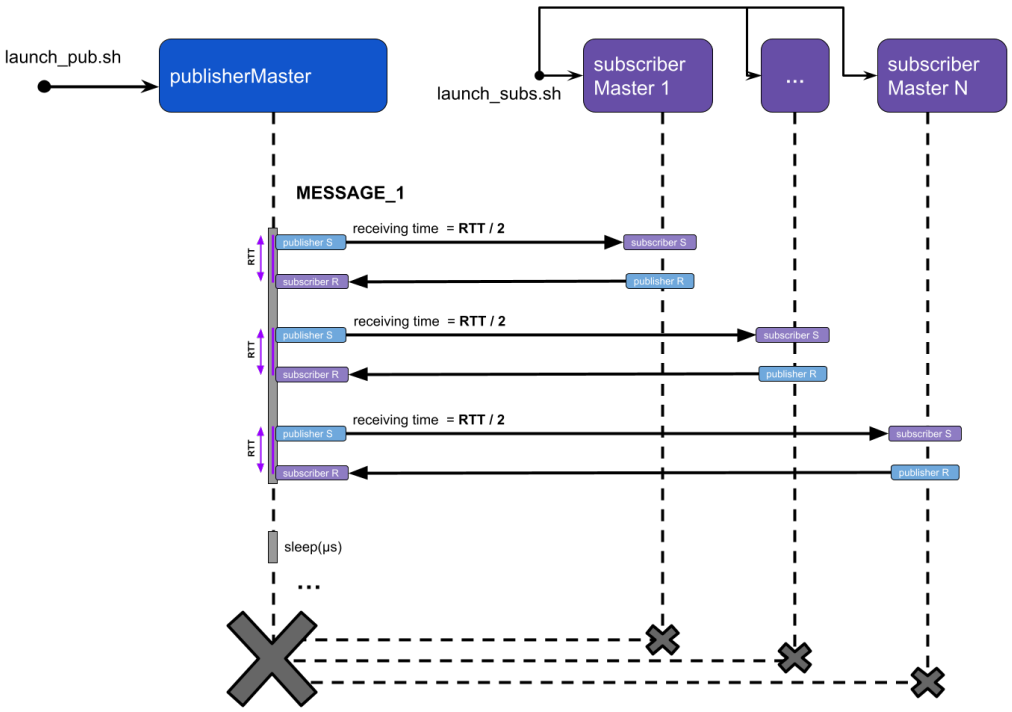


Figura 5.5. Schema UML RTT

Ovviamente questo metodo può comportare qualche ritardo intrinseco di dover gestire due entità per ogni attore, ma sono tempi infinitesimali in confronto al tempo necessario per inviare il messaggio su rete (dove è stato usato questo approccio).

5.5 Schema

Sono stati svolti diversi test al fine di trovare un modello ottimale di utilizzo e per la caratterizzazione di DDS, all'interno di sitemi HPC, nel contesto del Power Management. Nello specifico i test sono stati utili a capire il peso che avesse una singola configurazione o modello di utilizzo al fine di trovare quello più adeguato per una futura implementazione. I test effettuati sono:

- test-0: discovery
- test-1: protocollo di comunicazione

- test-2: partizioni e wildcards
- test-3: throughput

Al fine di condurli nel modo più trasparente e corretto possibile sono stati resi pubblici [20] tutti i codici utilizzati durante lo svolgimento di questi test.

5.5.1 Test-0

La prima fase di DDS consiste nel riconoscere altri *dds-partecipant* appartenenti allo stesso dominio. Questa viene chiamata **fase di Discovery**. Essa ha un ruolo estremamente importante siccome senza non sarebbe possibile far comunicare nessun attore all'interno della stessa rete e nemmeno nello stesso nodo. Una delle peculiarità di DDS è che permette di eseguire tutto in modo completamente distribuito, e anche questa fase non viene fatta diversamente (se non esplicitamente configurato). Il problema è che durante la discovery, quando si hanno diverse migliaia di componenti, si genera un overhead di pacchetti scambiati e di conseguenza anche le computazioni necessarie che crescono esponenzialmente. Sono disponibili in DDS, anche per questo motivo, meccanismi di discovery centralizzati al fine di evitare questo problema. In questo esperimento si andranno a comparare le due diverse implementazioni e per farlo si useranno gli strumenti **perf** e **TCPdump**.

5.5.2 Test-1

In DDS ed in particolare nel layer sottostante di RTPS, per scambiare messaggi anche tramite rete, e non solo nello stesso nodo, è possibile scegliere come mezzo diversi tipi di protocolli:

- udp: fornisce due versioni v4 e v6 e importa l'omonimo protocollo di trasporto
- tcp: fornisce due versioni v4 e v6 e importa l'omonimo protocollo di trasporto
- udp-multicast: una versione modificata del semplice udp, dove tutti i subscriber collegati allo stesso topic, hanno un indirizzo comune di ricezione dei dati, permettendo così al publisher di inviare un singolo messaggio che viene condiviso tra tutti i subscriber
- shared-memory: analogo al metodo precedentemente, ma invece di utilizzare un indirizzo IP, viene utilizzato un indirizzo di memoria. E' possibile solo quando i due processi che comunicano sono sullo stesso nodo, con memoria condivisa.

Nel primo test si è valutata la differenza di queste implementazioni utilizzando la rete infiniband 5.1 su diversi nodi di un supercalcolatore.

5.5.3 Test-2

Un concetto fondamentale nelle comunicazioni tra attori con gerarchie diverse, in sistemi con diverse centinaia di migliaia di entita, come cluster, nodi, processori, workflow, job (etc.), sono le possibilità di instradare, segmentare e rendere gerarchiche le comunicazioni. Come spiegatolo nel capitolo 3 in DDS ci sono diversi strumenti disponibili per farlo. Tra di loro differiscono per alcuni aspetti, come flessibilità, costo (in performance) e livello di segmentazione.

Dominio

Il dominio è la segmentazione di più "forte" e di più alto livello. Va a partizionare gli attori presenti in un dominio in modo del tutto fisico (cambiando per ogni dominio porte e indirizzi di comunicazione) e per nulla flessibile. Per cambiare il dominio è necessario distruggere e creare di nuovo il partecipante. Inoltre il dominio non permette nessun tipo di gerarchia.

Topic

All'interno di un dominio i topic definiscono il metodo principale di instradamento dei messaggi, essendo però limitato dal tipo di messaggio che si vuole inviare. Infatti topic diversi supportano tipi di dato diversi, e non sono modificabili a run-time. Inoltre il topic non permette gerarchie ed è difficilmente modificabile a run-time

Partizione

Questo strumento risulta molto interessante, in quanto all'interno di un topic permette di definire gerarchie (è possibile sottoscriversi a più partizioni contemporaneamente), definisce wildcards e crea una segmentazione virtuale. Inoltre è facilmente modificabile a run-time.

Wildcards

Le wildcards sono un costrutto appartenente alle partizioni, che permette di definire dei pattern testuali sulla base del quale vengono instradati i messaggi. Un esempio può essere *Node** che va a corrisponde a tutti i messaggi sotto il topic precedentemente definito, a tutte le partizioni che iniziano con Node.

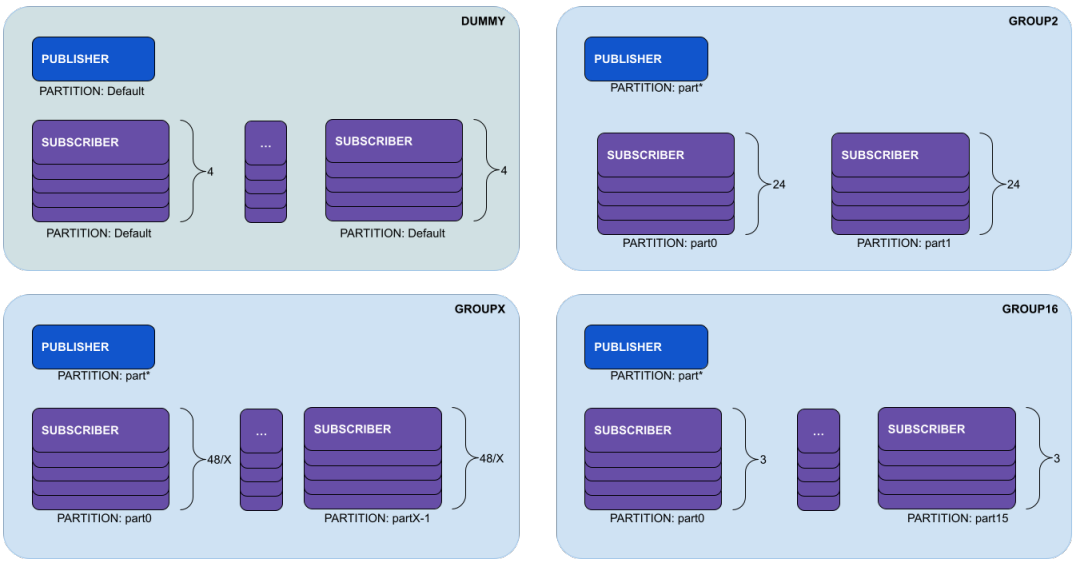


Figura 5.6. Schema test wildcards

Definiti questi concetti è stato pensato un test rappresentato dalla figura 5.6 per valutare se utilizzare la partizioni, e quindi sfruttare le gerarchie negli instradamenti avesse un prezzo in termini di performance da pagare.

5.5.4 Test-3

Nel l'ultimo test ci si è focalizzati sui risultati numerici dell'utilizzo di questo middleware. Ci si è focalizzati principalmente sul throughput, un valore particolarmente importante quando si parla di sistemi con volumi di scambio estremamente elevati. Sono stati eseguiti i test su tutti e 4 i protocolli messi a disposizione, al fine di fornire una panoramica completa, con risultati chiari. Inoltre per calcolarlo sono state cambiate le configurazioni in modo che potessero simulare un carico di comunicazioni elevato tra attori di HPC. Nella tabella 5.2 vengono riportate i parametri scelti.

Parametri	**Valore**
[#] publisher	1
[#] subscriber	40
[#] messaggi scambiati (per attore)	10 000
Dimensione del messaggio	16 Byte

Tabella 5.2. Valori usati per il test 3

Risultati

Nella sezione corrente, si riportano tutti i risultati rilevanti ottenuti durante la fase di testing, stilando un possibile modello di use-case utile alle finalità di Power Management.

6.1 Discovery centralizzata contro distribuita

Come previsto, nella fase di discovery un numero elevato di entità genera una quantità di operazioni da svolgere che cresce in modo esponenziale. In questo caso sono stati usati tutti i core di due nodi, con un totale di 96 entità. Già nel primo grafico è possibile notare un distacco tra i due approcci, con sole 30 unità 6.1. Nelle figure dei cicli 6.2 e delle istruzioni 6.3 viene confermato l'andamento non lineare.

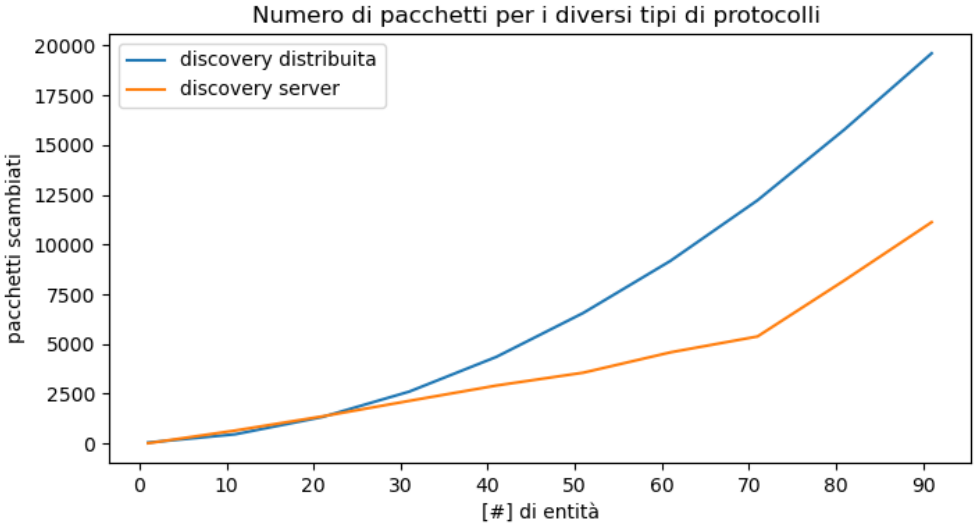


Figura 6.1. Numero di pacchetti scambiati durante la discovery all'aumentare di entità

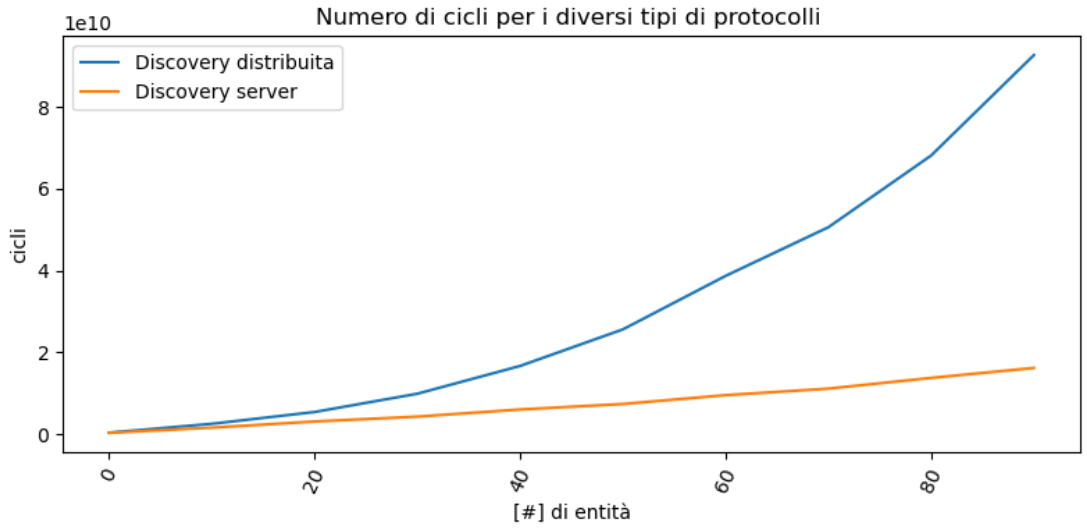


Figura 6.2. Numero di cicli durante la discovery all'aumentare di entità nelle diverse opzioni

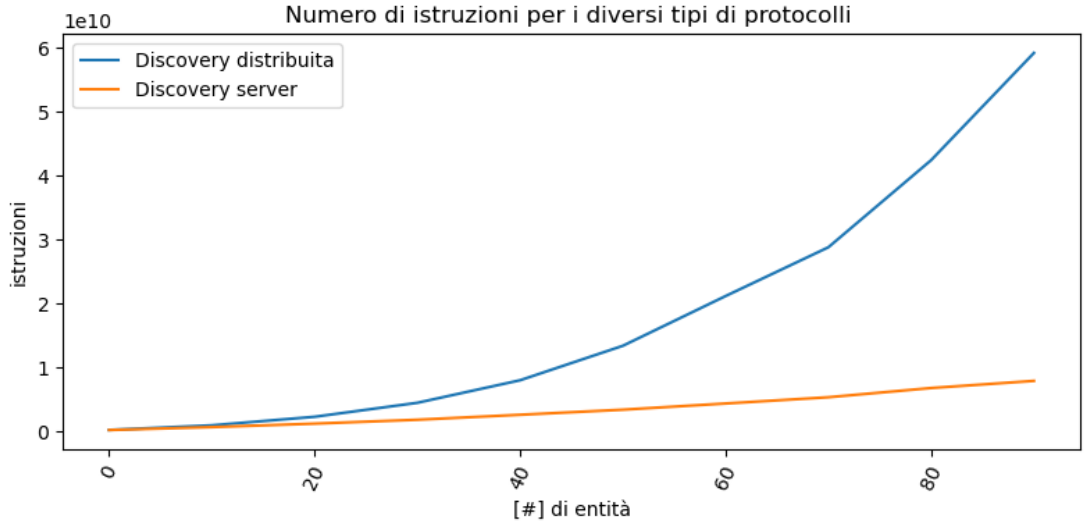


Figura 6.3. Numero di cicli durante la discovery all'aumentare di entità nelle diverse opzioni

In un caso reale serve valutare in primo luogo il numero di entità in un determinato dominio, e successivamente i costi-benefici di ogni implementazione considerando anche l'impatto che si può avere nel caso di fallimento del server (nonostante sia possibile avere un server di backup che viene automaticamente attivato, nel caso il primo fallisse)

6.2 Impatto del numero di sub iscritti ad un topic

Visto lo schema 5.2 si può capire, che il numero di subscriber presenti in un dominio ed iscritti ad un topic, comporta un overhead di comunicazione che va ad influenzare sia i tempi, che cicli e istruzioni impiegate nella singola *publish* come viene dimostrato nella figura 6.4 e 6.5. L'effetto è particolarmente pronunciato anche in protocolli come *UDP* che non dovrebbero possedere concetti di connessione. Questo potrebbe essere dovuto al costo di inizializzazione ed invio dei messaggi, ad indirizzi di rete diversi.

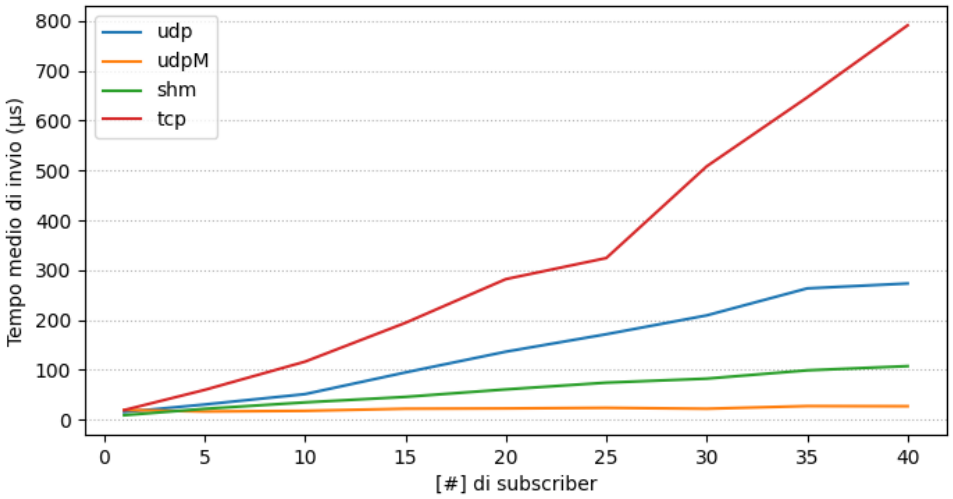


Figura 6.4. overhead sulla publish all'aumentare dei subscriber

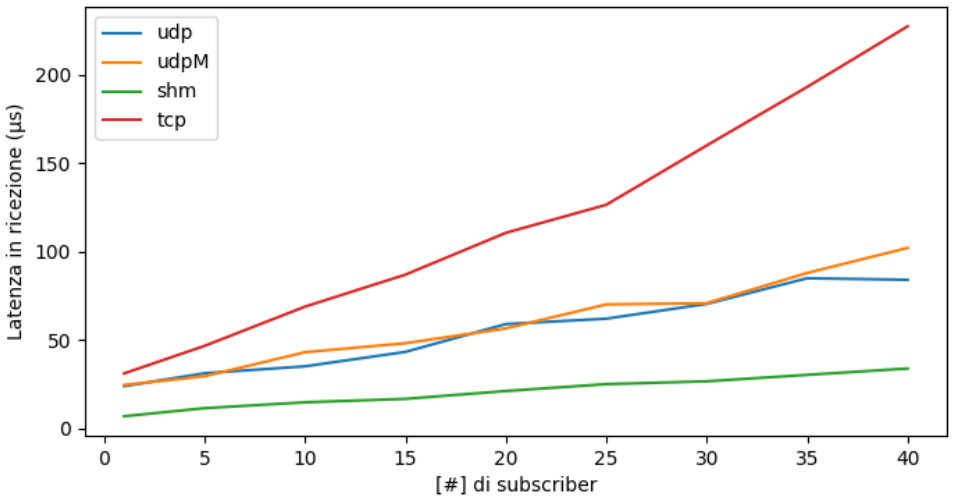


Figura 6.5. latenza di ricezione all'aumentare dei subscriber

Ovviamente l'impatto è poco significativo in quei protocolli che applicano strutture di come udp-Multicast e Shared-Memory, che verranno

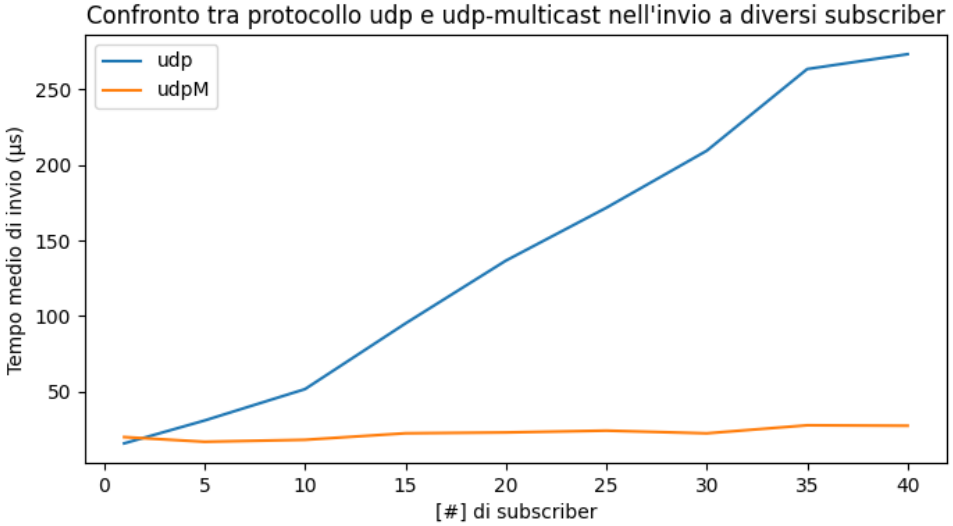


Figura 6.6

Da questo si può concludere che sia il publisher che subscriber risentono della presenza di molteplici *dds-partecipant* in ascolto su un topic. Questo problema è migliorato nel caso vengano utilizzati protocolli che si basano su multicast.

6.3 Primo messaggio

E' stato notato con tutti i protocolli utilizzati un ritardo, di un ordine di grandezza superiore, che riguarda esclusivamente il primo messaggio. Tuttavia, non è stato chiarito il motivo di questo overhead, presente anche in comunicazioni locali[1]. Anche se non dimostrato una delle possibili motivazioni potrebbe essere la necessità di allocare memoria durante la prima fase di comunicazione, da entrambi gli attori (potenzialmente amplificato nel caso 5.5).

[1] comunicazioni effettuati in localhost o in shared memory

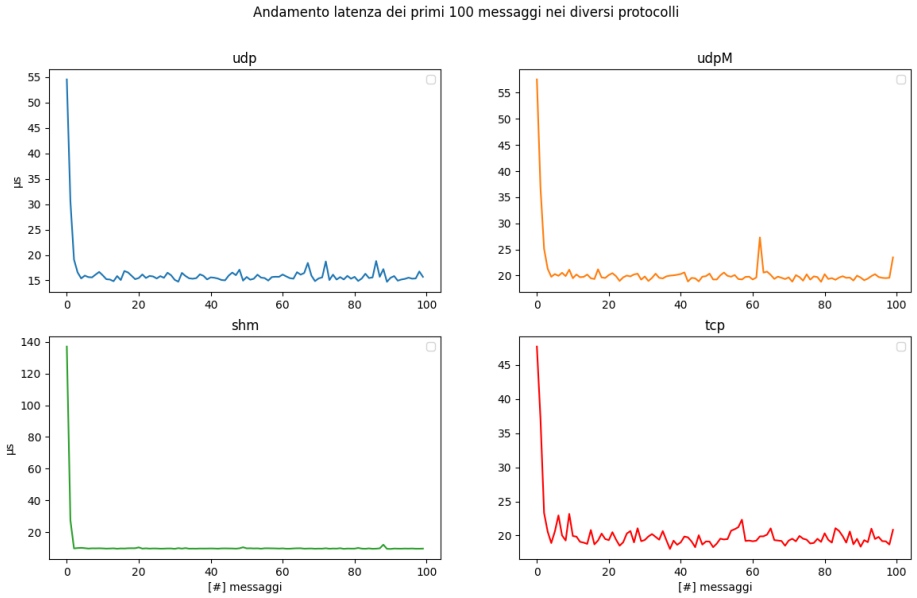


Figura 6.7

Data la complessità necessaria per andare così a fondo nel problema, non è stato approfondito ulteriormente.

6.4 Protocolli di comunicazione

Visti gli schemi 5.1 è prevedibile che in un vero Power Stack, siano predilette le comunicazioni non locali. In merito a ciò, nonostante ci siano di mezzo molti più livelli per comunicare con udp e tcp, i risultati trovati sono stati decisamente interessanti e non così lontani dal più veloce *Shared Memory*.

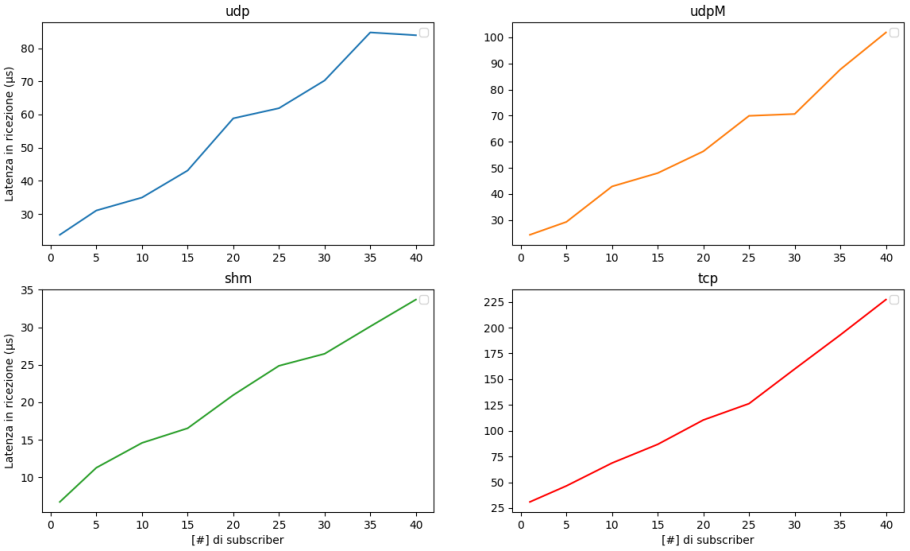


Figura 6.8. differenza tra solo publish e publish-subscribe per ogni protocollo

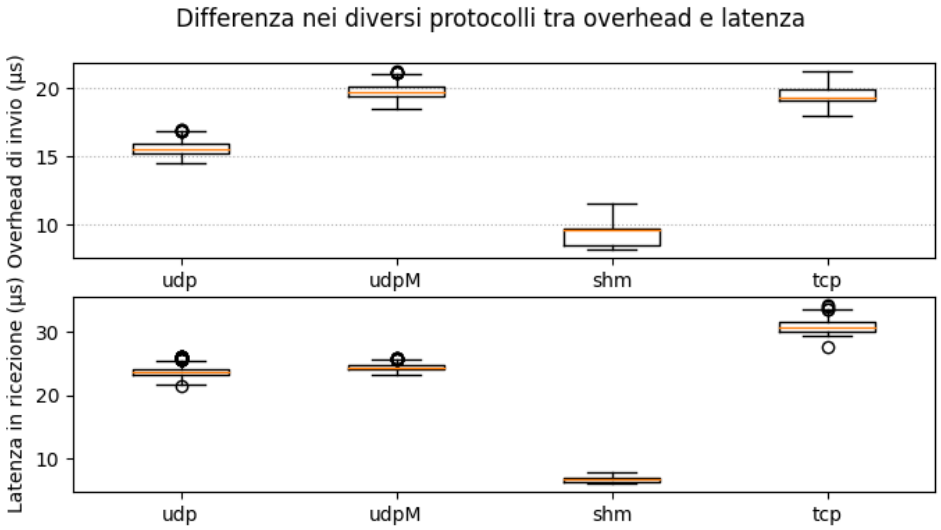


Figura 6.9. diagramma a scatola nei vari protocolli di comunicazione con un publisher e un subscriber

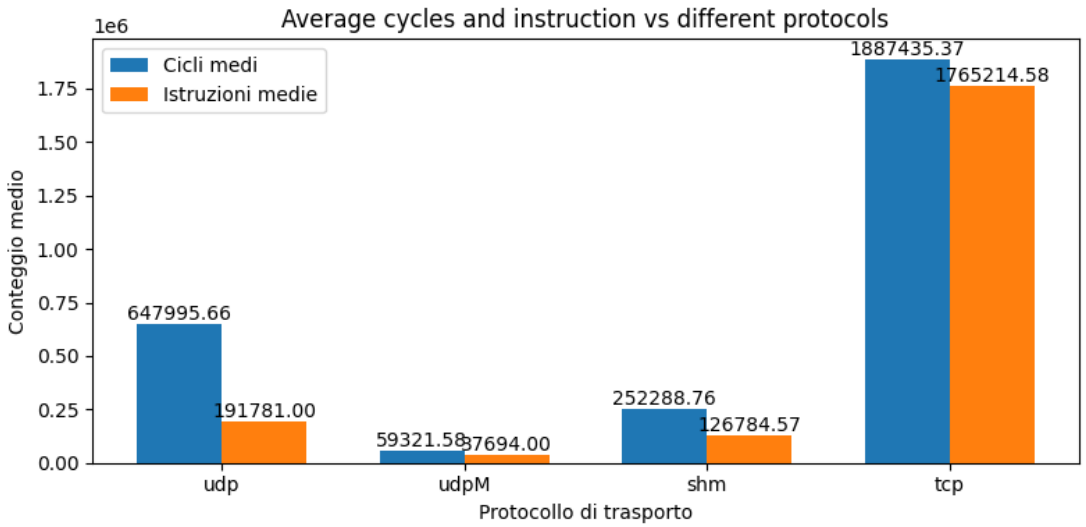


Figura 6.10. Conteggio cicli e istruzioni per ogni protocollo

E' quindi preferibile, ove possibile, usare *Shared Memory* sia per prestazioni, che per evitare di saturare la rete. In secondo luogo, se in presenza di diversi subscriber, per non caricare il publisher usare *UDP Multicast*. E infine utilizzare *TCP* solo dove vi è necessità di connessione, visto le notevoli latenze si in scrittura che in lettura. In tutti gli altri casi, dove non si è in locale, e dove il numero di subscriber non supera qualche decina, risulta più che adeguato *UDP*.

6.5 Domini, Partizioni e Wildcards

Nei test effettuati con topic e partizioni, non sono state notate differenze degne di nota in termini di performance nell'usare uno strumento piuttosto che un altro (6.11). Lo sono stati invece tra questi ultimi e i Domini. Tuttavia i domini non offrono alcun tipo di flessibilità e richiede il riavvio dei *dds-partecipant* nel caso si necessiti di un qualsiasi cambiamento. Per questo è consigliato utilizzo di domini diversi, solo tra entità che non hanno necessità di comunicare, e che anzi, magari anche per motivi di sicurezza devono stare isolati.

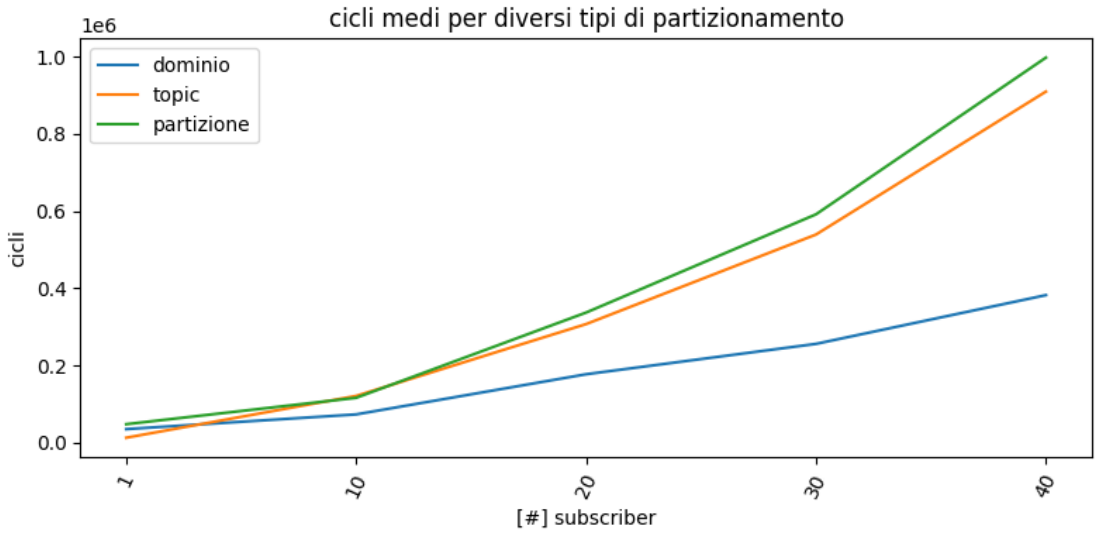
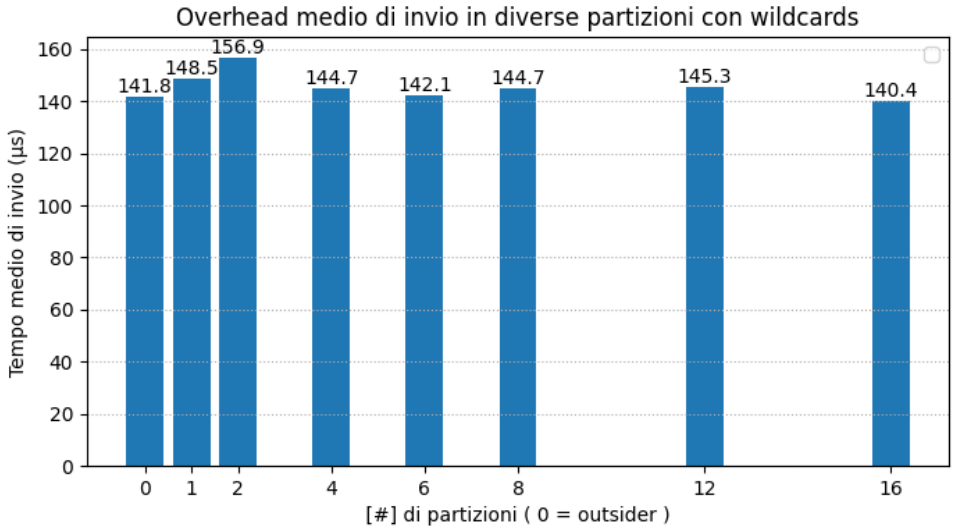


Figura 6.11. Peso in termini di cicli nell'usare partizioni topic e domini

Per quanto riguarda entità che necessitano di gerarchie, è conveniente usare le partizioni con le Wildcards visto che non hanno un peso significativo 6.12. Infine lasciando i topic come mezzo di partizionamento, per il tipo di dati, e il tipo di istruzioni da utilizzare.



la colonna 10 e 14 sono state tolte per incosistenza del numero dei subscriber, Magari poi lo dico

Figura 6.12. La differenza tra dummy, 1 partizione o 16 partizioni non ha particolari costi in termini di invio con utilizzo di wildcards

6.6 Throughput

L'ultimo test è stato utile a comprendere tutte le potenzialità di questo strumento più che creare un modello di utilizzo. Infatti come si può vedere dai seguenti grafici, i numeri di KByte (6.13) che è possibile inviare ogni secondo da un publisher ad un subscriber supera il MB/s, e permettendo comunicazione di diverse decine di Hz (6.13). Inoltre questo valore raggiunge numeri ancora più elevati qualora si usassero comunicazioni multicast, come notabile in figura 6.14.

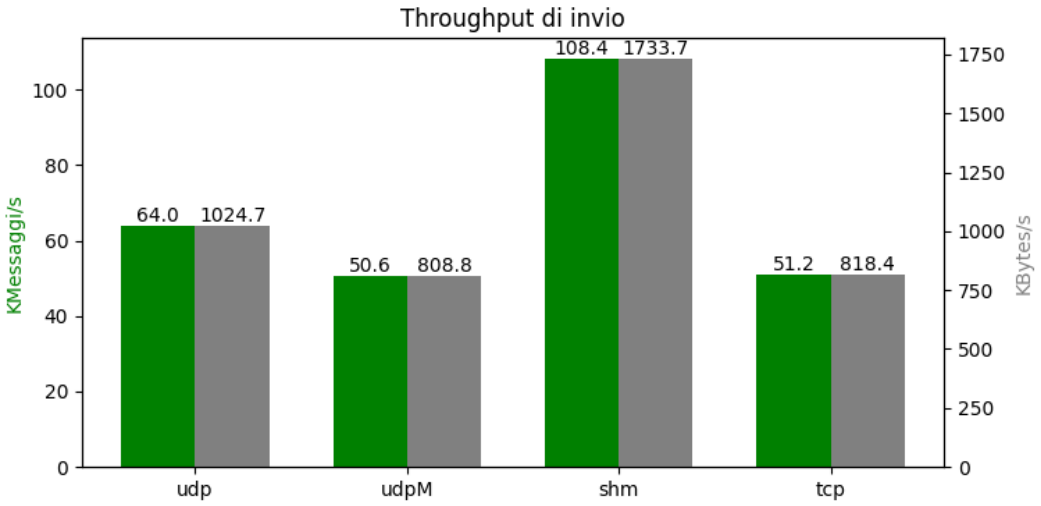


Figura 6.13. throughput di invio da parte di un singolo attore

Sull'asse delle Y, a sinistra, vengono mostrati in verde i il valore di KMessaggi/s[2] raggiungibili dai vari protocolli di comunicazione, mentre in grigio, a sinistra i KByte/s.

[2] 1000 Messaggi al secondo

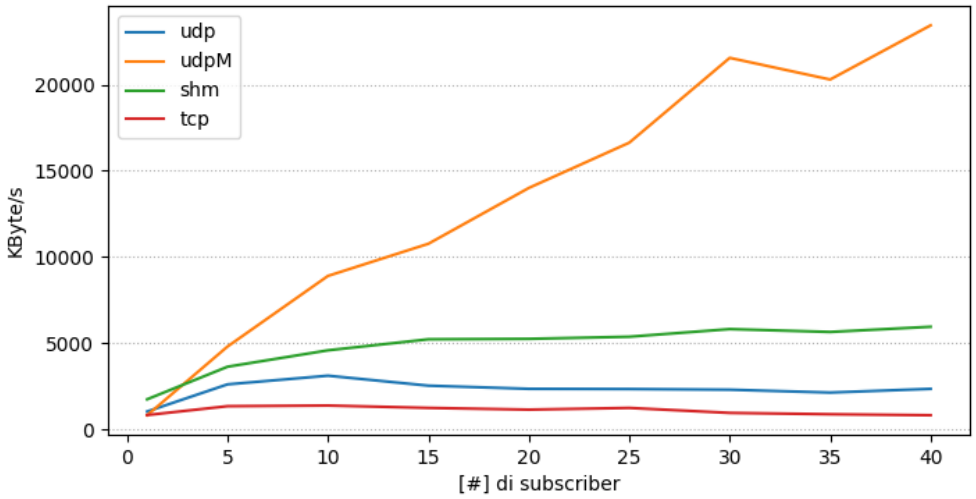


Figura 6.14. throughput a crescenti numeri di subscribers

Poi metto tutti i titoli. La descrizione di ogni asse è obbligatoria? mi sembra molto ripetitiva

Componenti dummy

Nel corso di questa tesi con la collaborazione dei membri del progetto REGALE, come Cineca[17], E4[18] e BSC[21] sono stati sviluppati dei prototipi di componenti del modello di Power-Stack per HPC. Questi ultimi oltre a fornire una prova delle potenzialità del middleware di DDS, sono utili anche come esempio per una sua effettiva implementazione all'interno dei software mostrati nel capitolo 4 che vogliono essere introdotti nell'infrastruttura. In merito a questo, nello schema 7.1 viene mostrato lo stato di avanzamento del Power Stack, illustrando quali di questi sono implementati e operativi, e quali in via di sviluppo.

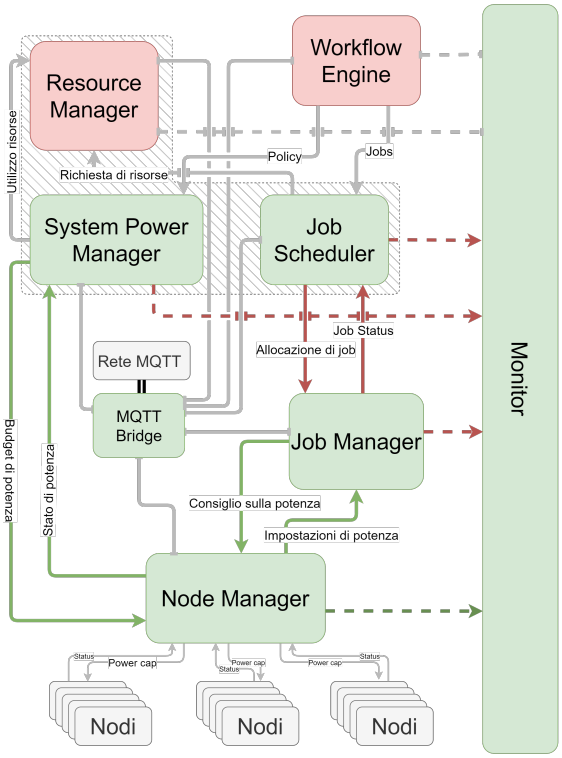


Figura 7.1. Schema componenti sviluppati: in verde completato, in grigio non previsto, in rosso ancora da sviluppare

L'infrastruttura DDS utilizzata, chiamata *REGALE Library* al momento permette di utilizzare i vari tipi di comunicazione, configurazioni di qos, e anche vari tipi di dati scambiati, tutti passabili tramite file **XML**.

I componenti creati sono dei programmi in c++ che vanno a simulare uno scambio di informazioni realistico. Al momento per motivi di semplicità tutti i dummy utilizzano dati di tipo *uint32_t* ed il loro comportamento si può riassumere nel seguente modo.

Job Scheduler dummy

Il JS interroga ogni 10 secondi il SPM per le informazioni sui servizi e la potenza totale del cluster riportandolo a schermo. Successivamente imposta il limite di potenza del cluster con un valore casuale tra 1000 e 1500.

Job Manager dummy

Il JM interroga il NM ogni 10 secondi e richiede il *powercap* impostato e le informazioni sulle frequenze (massima, minima e corrente) riportando a schermo tutti i valori ottenuti.

System power manager dummy

Il SPM nella sua implementazione server aspetta per le richieste in entrata. Al momento quelle previste sono:

- GET_INFO
- GET_POWER
- SET_POWER

Inoltre quando riceve una GET_POWER il SPM inoltra la richiesta al NM per ottenere la sua potenza prima di rispondere alla richiesta.

Node manager dummy

Il NM mentre ogni 30 secondi manda i dati al Monitor, aspetta per le richieste in entrata. Le richieste che accetta sono:

- GET_INFO_NODE
- GET_POWER_NODE
- GET_POWERCAP_NODE
- GET_MINFREQ_NODE

- GET_MAXFREQ_NODE
- GET_CURFREQ_NODE
- SET_POWER_NODE

Monitor dummy

Il Monitor semplicemente aspetta che i componenti gli inviino i dati, e quando li riceve li riporta a schermo.

MQTT Bridge

Questo componente, a differenza di tutti gli altri, non ha alcun ruolo nel Power Management ma serve solo a supporto di alcuni software che fanno ampio uso di comunicazioni **MQTT**[22]. Infatti il suo scopo è quello di intercettare e convertire tutte le comunicazioni provenienti da MQTT e DDS per infine inoltrare quelle desiderate nel protocollo opposto.

Ci sta la descrizione di questi componenti? oppure è inutile?

7.1 Struttura

I componenti mostrati fanno uso delle strutture di domini, partizioni e topic mostrati in tabella 7.2.

DUMMIES	
NAME	**USED**
NODE MANAGER DUMMY	P 0 default Monitor_report_job_telemetry P 0 default Monitor_report_node_telemetry P 0 default Monitor_report_cluster_telemetry
JOB SCHEDULER DUMMY	P 0 default SystemPowerManager_get P 0 default SystemPowerManager_set S 0 default SystemPowerManager_get_reply S 0 default SystemPowerManager_set_repl
JOB MANAGER DUMMY	S 0 default NodeManager_get P 0 default NodeManager_get_reply

SERVERS	
NAME	**OFFERED**
NODE MANAGER	S 0 default NodeManager_get S 0 default NodeManager_set P 0 default NodeManager_get_reply P 0 default NodeManager_set_reply
SYSTEM POWER MANAGER	S 0 default SystemPowerManager_get S 0 default SystemPowerManager_set P 0 default SystemPowerManager_get_reply P 0 default SystemPowerManager_set_reply
MONITOR	S 0 default Monitor_report_job_telemetry S 0 default Monitor_report_node_telemetry S 0 default Monitor_report_cluster_telemetry

Figura 7.2. Strutture di comunicazioni dei componenti implementati nel power stack.

In viola tutti i publisher, ed in giallo i subscriber. Successivamente nel rispettivo ordine vengono mostrati, domini, partizioni e topic utilizzati.

Conclusioni

Nel primo capitolo dopo una introduzione sui concetti di High-Performance Computing, e dei problemi che si riscontrano nei giorni nostri, sono stati approfonditi nel secondo, i concetti di Power Management e Power Stack analizzando lo stato dell'arte sulla gestione delle potenze, con tutti i componenti ad esso associati.

Nel terzo capitolo e' stato introdotto il framework DDS che si è voluto analizzare, ed in particolare la sua implementazione *FastDDS* che rappresenta la tecnologia alla base di questa tesi. Si sono inoltre accennati i meccanismi usati in ROS2 per risolvere i problemi analoghi a quello riscontrato nel Power Stack.

Nel quarto, vista la collaborazione con il progetto REGALE, sono stati illustrati i progetti e gli obbiettivi ricercati, introducendo brevemente anche i software open-source che saranno implementati nei vari ruoli del Power Stack.

Il quinto capitolo è stato utile a presentare in modo esaustivo tutti gli strumenti utilizzati per eseguire i test, oltre agli obbiettivi e al metodo di sperimentazione. Successivamente, nel sesto, sono stati mostrati i risultati degli esperimenti fatti. In primo luogo si è visto l'andamento esponenziale nella fase di discovery risolvibile tramite la sua implementazione di server. Il protocollo più performante si è rilevato essere quello di Shared Memory utilizzabile però solo in presenza di memorie condivise. Al secondo posto UDP e UDP Multicas, ma quest'ultima, solo in presenza di diversi subscriber. Infine non si sono notate particolari differenze nell'usare le wildcards, aprendo le strade alle comunicazioni di tipo gerarchico. Grazie ai risultati è stato possibile fornire un modello di utilizzo su come è possibile utilizzare e impostare al meglio questo middleware.

In conclusione sono stati mostrati tutte le implementazioni *Dummy* con le relative strutture di comunicazione, prodotte in collaborazione con il progetto REGALE.

Può andare come conclusione?

Glossario

Devo completarlo, ma è l'ultima cosa che faccio, che se non ho tempo, non credo sia fondamentale

assembly 26

kernel 26

ntpd 26

38

overhead 38

Bash Bourne Angain Shell, linguaggio di scripting 24

Real-Time In tempo reale, con latenze molto basse 9, 16

Piuttosto pensavo di mettere la pagina delle figure

Bibliografia

[1] Andrea Borghesi et al. «MS3: A Mediterranean-stile job scheduler for supercomputers - do less when it's too hot!» In: *2015 International Conference on High Performance Computing ands Simulation (HPCS)*. 2015, pp. 88–95. DOI: 10.1109/HPCSim.2015.7237025.

[2] R.H. Dennard et al. «Design of ion-implanted MOSFET's with very small physical dimensions». In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[3] Hadi Esmaeilzadeh et al. «Dark Silicon and the End of Multicore Scaling». In: *IEEE Micro* 32.3 (2012), pp. 122–134. DOI: 10.1109/MM.2012.17.

[4] TOP500. *TOP500 Supercomputer Sites*. 2023. URL: https://www.top500.org/.

[5] INTEL. *GEOPM*. 2017. URL: https://sc17.supercomputing.org/SC17%20Archive/tech_poster/poster_files/post176s2-file3.pd.

[6] Daniel Hackenberg et al. «HDEEM: High Definition Energy Efficiency Monitoring». In: *2014 Energy Efficient Supercomputing Workshop*. 2014, pp. 1–10. DOI: 10.1109/E2SC.2014.13.

[7] Matthias Maiterth et al. «Energy and Power Aware Job Scheduling and Resource Management: Global Survey — Initial Analysis». In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018, pp. 685–693. DOI: 10.1109/IPDPSW.2018.00111.

[8] Object Management Group. *Data Distribution Service*. 2004. URL: https://www.omg.org/spec/DDS/1.0.

[9] Object Management Group. *DDS Interoperability Wire Protocol*. 2008. URL: https://www.omg.org/spec/DDSI-RTPS/2.0.

[10] eProsima. *FastDDS*. 2022. URL: https://fast-dds.docs.eprosima.com/en/v2.11.2/.

[11] Open Robotics. *ROS 2 Documentation: Iron documentation*. https://docs.ros.org/en/iron/index.html. 2023.

[12] CSLab;NTUA. *Open Architecture for Future Supercomputers*. 2021. URL: https://regale-project.eu/ (visitato il 07/05/2023).

[13] *An open architecture to equip next generation HPC applications with exascale capabilities*. https://cordis.europa.eu/project/id/956560. Accessed: yyyy-mm-dd.

[14] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Knoxville, TN, USA: University of Tennessee. 1994.

[15] SLURM Development Team. *SLURM: A Highly Scalable Workload Manager*. https://slurm.schedmd.com/overview.html. 2002.

[16] Leibniz Supercomputing Centre. *DCDB: Modular and holistic Monitor for HPC*. https://gitlab.lrz.de/dcdb/dcdb. 2011.

[17] Cineca. *Cineca Galileo 100*. 2021. URL: https://www.hpc.cineca.it/hardware/galileo100.

[18] E4 System. *E4 HPC Systems*. URL: https://www.e4company.com/.

[19] Antonio Libri et al. «Evaluation of NTP/PTP fine-grain synchronization performance in HPC clusters». In: nov. 2018, pp. 1–6. ISBN: 978-1-4503-6591-8. DOI: 10.1145/3295816.3295819.

[20] Giacomo Madella. *github/tesiMagistrale*. 2023. URL: https://github.com/madella/tesiM.

[21] BSC. *Barcelona Supercomputing Center*. URL: https://www.bsc.es/.

[22] IBM. *Message Queuing Telemetry Transport*. https://mqtt.org/. 1999.