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
**Correlatore**
Dr. Federico Tesser

**Candidato**
Giacomo Madella

Ottobre 2023

# Abstract

Nei sistemi di High-Performance Computing (HPC), la gestione energetica è diventata una delle principali preoccupazioni, non solo a causa dei costi monetari di esercizio, ma anche per la sostenibilità ambientale e per la progettazione di nuove generazioni di supercalcolatori[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. Il concetto di Power Management si propone di gestire la potenza di sistemi di HPC tramite approcci software. Inoltre data l'eterogeneità dei supercalcolatori, nel corso degli anni sono stati proposte soluzioni di Power Management, legate a specifiche configurazioni hardware e con una visione non olistica. Lo scopo di questa tesi è di: (i) sviluppare in collaborazione con i partner del progetto EuroHPC JU REGALE un middleware di comunicazione basato su Data Distribution Service (DDS), per facilitare lo scambio di informazioni tra i diversi software di Power Management; (ii) caratterizzare le prestazioni dei componenti di DDS al fine di creare un modello costi/benefici relativo alle scelte progettuali effettuate; (iii) realizzare un insieme di prototipi di attori di power management integrati con il middleware sviluppato.

# 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:

1. Power Management per la potenza assorbita, a sua volta suddivisibile in:
    - Thermal Design Power, potenza termica massima che un componente può dissipare;
    - Therm Design Current o Peak Current, massima corrente erogabile da alimentatori o dai processori;
2. Thermal management, gestione temperatura dinamica o statica;
3. 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 (PM) 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 crescenti nella riduzione della dimensione dei transistor, che ha portato al progressivo termine delle leggi di Dennard e Moore[2][3]. Tali leggi, che hanno 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 la realizzazione di software in grado di automatizzarne la gestione. Dall'arrivo degli exa-computer[1], infatti la potenza necessaria per alimentare questi sistemi ha superato la precedente soglia dei 20MWatt[4]. Considerando che la maggior parte della potenza fornita, viene convertita in calore, diventa importante considerare anche i consumi necessari al raffreddamento dei sistemi. Infatti, se il raffreddamento risulta inadeguato comporta inefficienze energetiche, che si traduce in degradazioni di prestazioni computazionali. Per questo motivo, il Power Management dei Data Center è diventato ogni anno un compito più complesso.

Al fine di ridurre il consumo di potenza di esercizio, si cerca un approccio in grado di imporre vincoli andando ad agire a diverse granularità: intero data-center, cluster di calcolo, nodo, job. Sono nati così i primi strumenti, che controllano l'utilizzo dell'energia tramite la riconfigurazione dinamica dei parametri operativi nei componenti di calcolo (Nodi, CPU, GPU, TPU) al fine di ridurre gli sprechi energetici e, allo stesso tempo, garantire una temperatura di funzionamento sicura. L'insieme dei meccanismi software e hardware che svolgono il compito di Power Management vengono racchiusi all'interno di un Power Stack(PS) unificato.

Ad oggi, sono state proposte diverse soluzioni per il Power Management di apparati HPC. La maggior parte di esse è stata sviluppata per sopperire a problematiche specifiche legate ai singoli vendor, e singoli centri di calcolo. Manca quindi una visione globale e la possibilità di poter comporre diverse soluzioni in modo sinergico. Non sono inoltre mai state definite o standardizzate interfacce di comunicazione comuni tra i vari software, lasciando agli amministratori dei sistemi di HPC l'onere di farlo: limitando de-facto l'uso e la pervasività di tali soluzioni.

## 1.1 Contributi

I contributi di questa tesi sono stati in primo luogo, lo studio e caratterizzazione di una specifica implementazione di DDS all'interno di sistemi HPC portando una visione più completa sullo strumento utilizzato. Successivamente sono stati valutati dei modelli basati sui risultati come *use-case*. Infine, in collaborazione con Cineca[5], BSC[6] ed E4[7], sono stati creati i middleware ed i prototipi, in grado di simulare vere comunicazioni nell'ambito del Power Management.

[1] Supercalcolatore in grado che raggiunge prestazioni di ExaFlops

# Stato dell'arte

Al fine di poter operare correttamente, il 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. Partendo dal livello più basso, è 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 interfacce differenti:

- in-band
- out-of-band

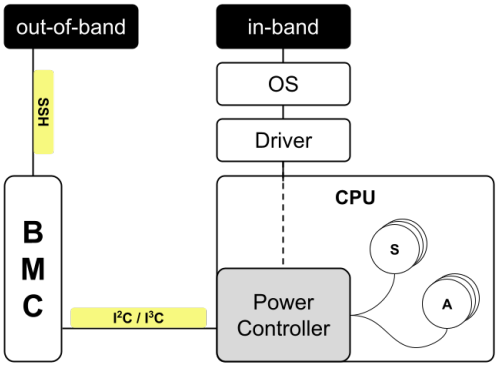



**Figura 2.1.** Differenza tra le interfacce in-band e out-of-band

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.

## 2.1 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[8] o RAPL[9] (per architetture intel) 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* (in sistemi intel) 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;

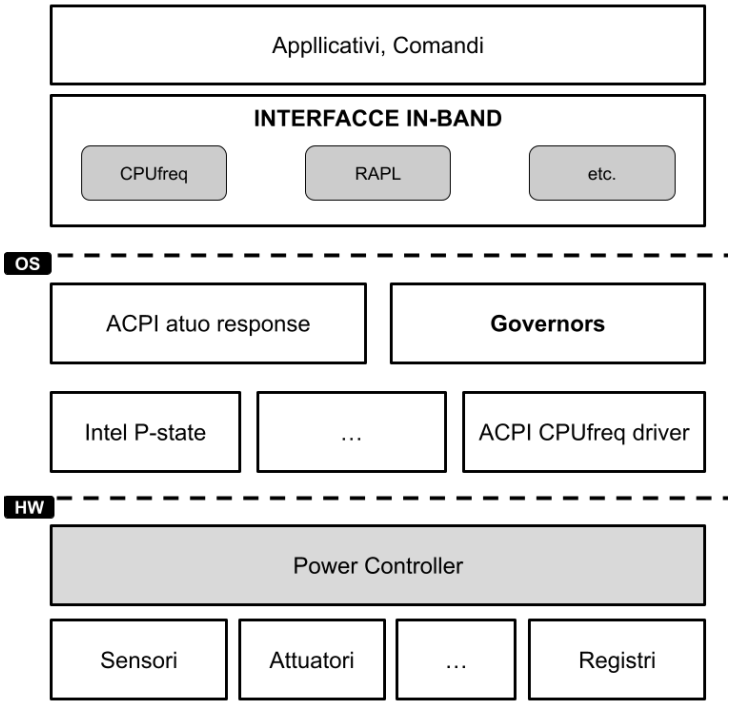



**Figura 2.2.** Struttura interfacce in-band: divise su più livelli tra cui Sistema Operativo (SO) e Hardware (HW)

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 le informazioni solamente nei core sui quali i processi vengono eseguiti.

## 2.2 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 le capacità.

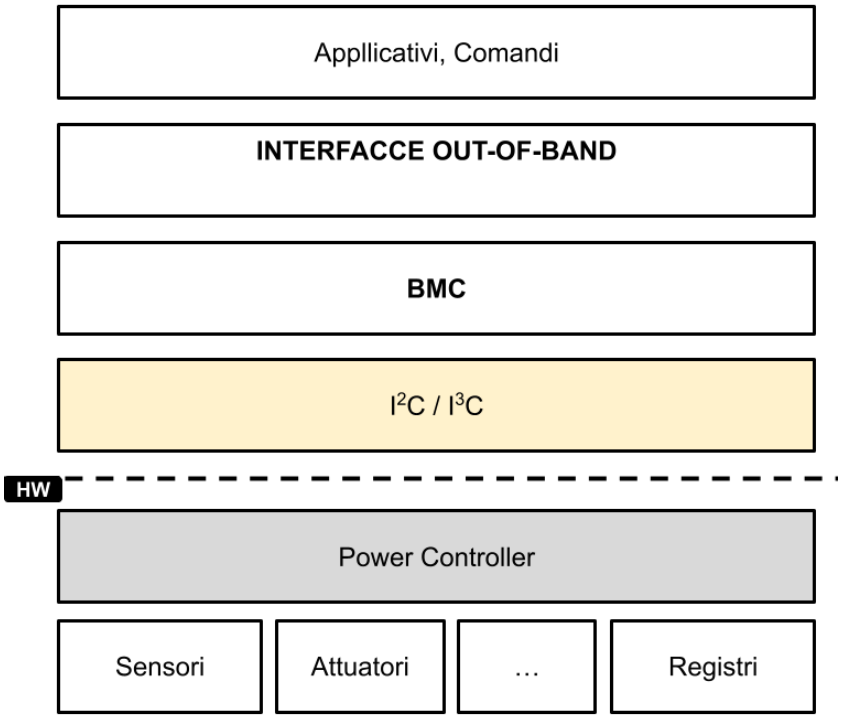



**Figura 2.3.** Struttura interfacce out-of-band

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

## 2.3 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)[10], e *HDEEM* (Atos)[11]. 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.

## 2.4 Modello di power stack

Con Power Stack si intende un insieme di software che cooperando riescono a fornire ad applicazioni, utenti e amministratori la gestione completa del Power Management. Una volta definito il problema, ed 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)

### 2.4.1 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.

### 2.4.2 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 Node Manager. Una volta fissati i limiti, vengono monitorati i dati relativi a potenza ed energia, e controlla di conseguenza i budget, e la *user-fairness*.

### 2.4.3 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.
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.

### 2.4.4 Resource Manager

Per riuscire a svolgere questo lavoro il Job Scheduler interagisce con uno o più Resource Manager[2]. Questi sono software che hanno il compito di dividere (o condividere) e orchestrare le risorse computazionali e fisiche del sistema HPC. Queste risorse includono diversi componenti:

- Nodi
- Processori
- Memorie
- Dischi

[2] A volte per questo il Job Manager ed il Resource Manager vengono collassati in un unico componente

- 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[12].

### 2.4.5 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.

### 2.4.6 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.

### 2.4.7 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 proprietà 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 mostra gli attori e le possibili interazioni che possono avere.

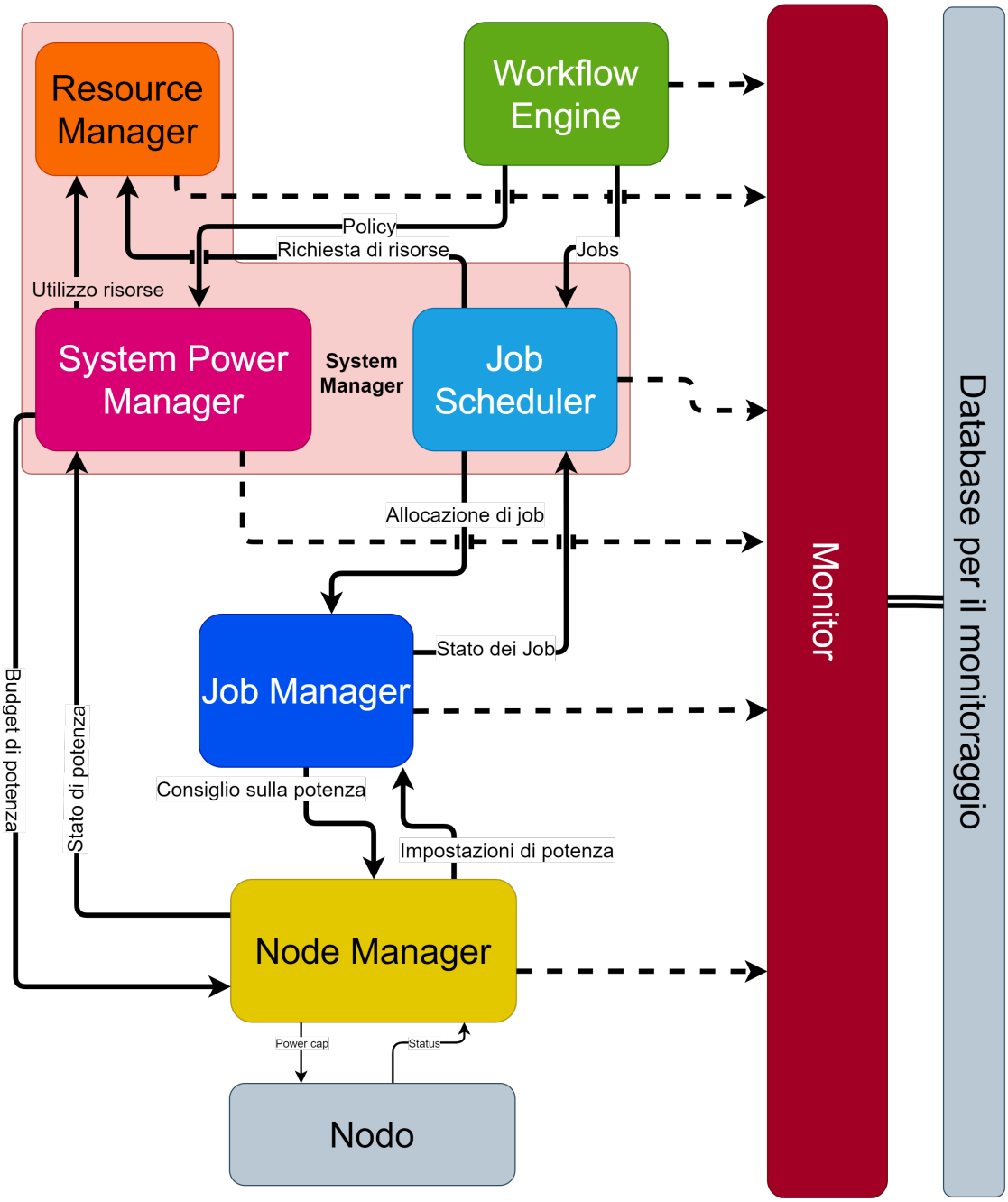



**Figura 2.4.** Modello di power stack

# DDS & RTPS

Data Distribution Service (DDS)[13] e Real-Time Publish-Subscribe (RTPS)[14] 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.

## 3.1 Implementazione usasta

In particolare, DDS e RTPS sono due differenti protocolli da cui sono state proposte diverse implementazioni da diversi vendor e produttori, 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 [15]. E' stato scelto di utilizzare questa implementazione dato il supporto per le comunicazione in tempo reale, 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 uno o più domini di appartenenza. Il DomainParticipant funge da contenitore per altre entità DCPS, e svolge anche la funzione di costruttore di entità Publisher, Subscriber e Topic;
- 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:

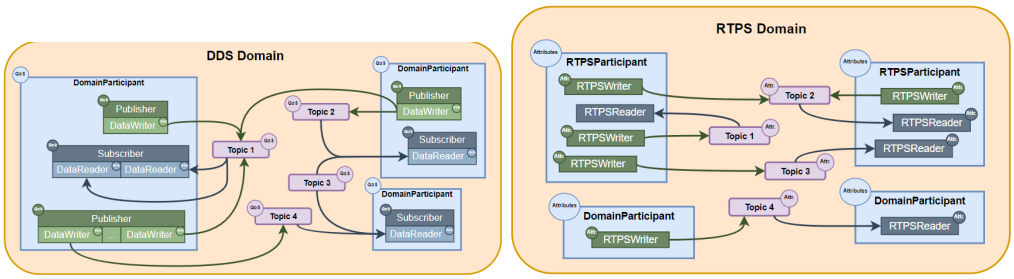
1. il *change* viene aggiunto nella cache del RTPSWriter;

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

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*.

Di seguito, dalla documentazione di eProsima FastDDS[15] gli schemi di funzionamento dei due layer di comunicazione.




**(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**[16] 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, permettendo 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.

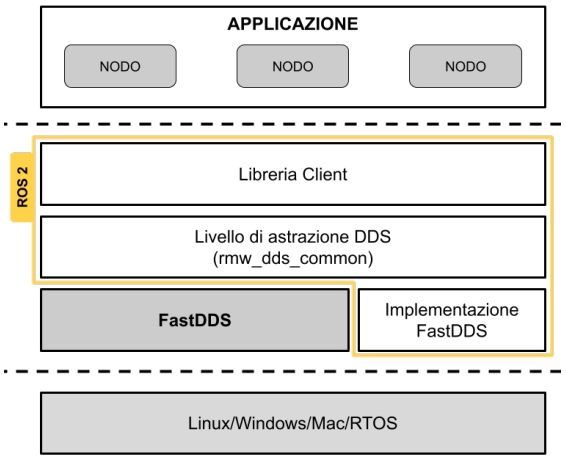



**Figura 3.2.** Ros Middleware per DDS

[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: in ROS sono predisposti dei pacchetti preconfigurati di entità, che risulta sconveniente per andare a caratterizzare questo strumento su architetture diverse, limitando la possibilità di 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;
- 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.

# REGALE

REGALE[17] è un progetto finanziato dall'UE[18] 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 quello di aprire la strada alla prossima generazione di applicazioni per HPC, riunendo 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[18]. Per farlo sono inoltre state definite delle politiche da seguire durante lo sviluppo di tutto il progetto:

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

## 4.2 Power Stack

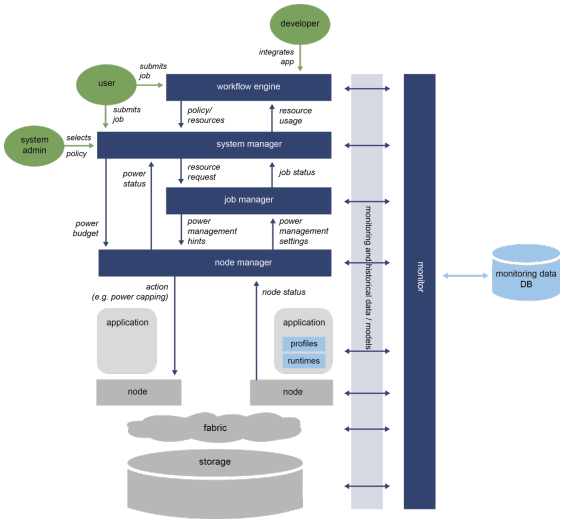
L'intero progetto di Power Stack, durante il suo sviluppo, si è basato su strumenti come MPI library[19], SLURM[20], o DCDB[21]. Inoltre, è stato deciso di introdurre software open-source che potessero soddisfare le esigenze del modello di Power Stack 4.1a.

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

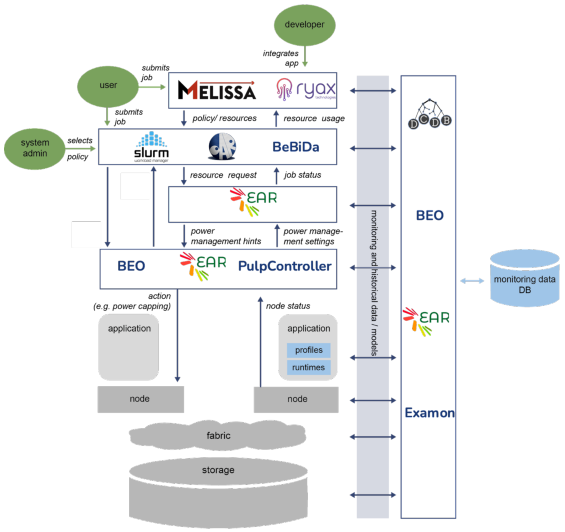
Infatti sono stati valutati e selezionati diversi applicativi (molti dei quali prodotti dai partner, come mostrati in tabella 4.1) anche con ruoli analoghi, per soddisfare diverse esigenze.

| **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




**(a)** Modello di Power Stack Regale




**(b)** Copertura componenti

**Figura 4.1.** Implementazione dei componenti secondo il modello del Power stack

## 4.3 Integrazione

Vista la natura dei software introdotti nel progetto, non era previsto che questi potessero scambiare informazioni tra di loro, in quanto nati per essere usati singolarmente. Inoltre, data la difficoltà di creare interfacce di comunicazione, specifiche per ogni coppia di componente, si è scelto di procedere con un **middleware DDS** unificato.

# Caratterizzazione DDS

Uno dei temi di questa tesi, è stato quello di generare e analizzare un modello utile per l'implementazione dell'infrastruttura sulla quale tutti gli attori di un Power Stack possano comunicare in modo distribuito. Per poter caratterizzare l'infrastruttura necessaria, sono stati utilizzati sistemi di High-Performance Computing sui quali andare a testare i vari esperimenti. A supporto di questo lavoro, sono stati resi disponibili due supercalcolatori uno da Cineca[5] e uno da E4[7] con le specifiche in tabella 5.1.

| **Parameter** | **Cineca** | **E4** |
|---|---|---|
| Processore | Intel CascadeLake 8260 S | Intel Xeon Silver 4216 |
| [#] sockets per nodo | 2 | 2 |
| [#] core per socket | 24 | 16 |
| Memoria per nodo | 384 GB | 100 GB |
| Connessione | Mellanox Infiniband 100GbE | Eth 100Gb |
| OS | CentOS Linux | Red Hat Enterprise 8.7 |
| MPI | Open MPI 4.1.1 | Open MPI 4.1.4 |

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

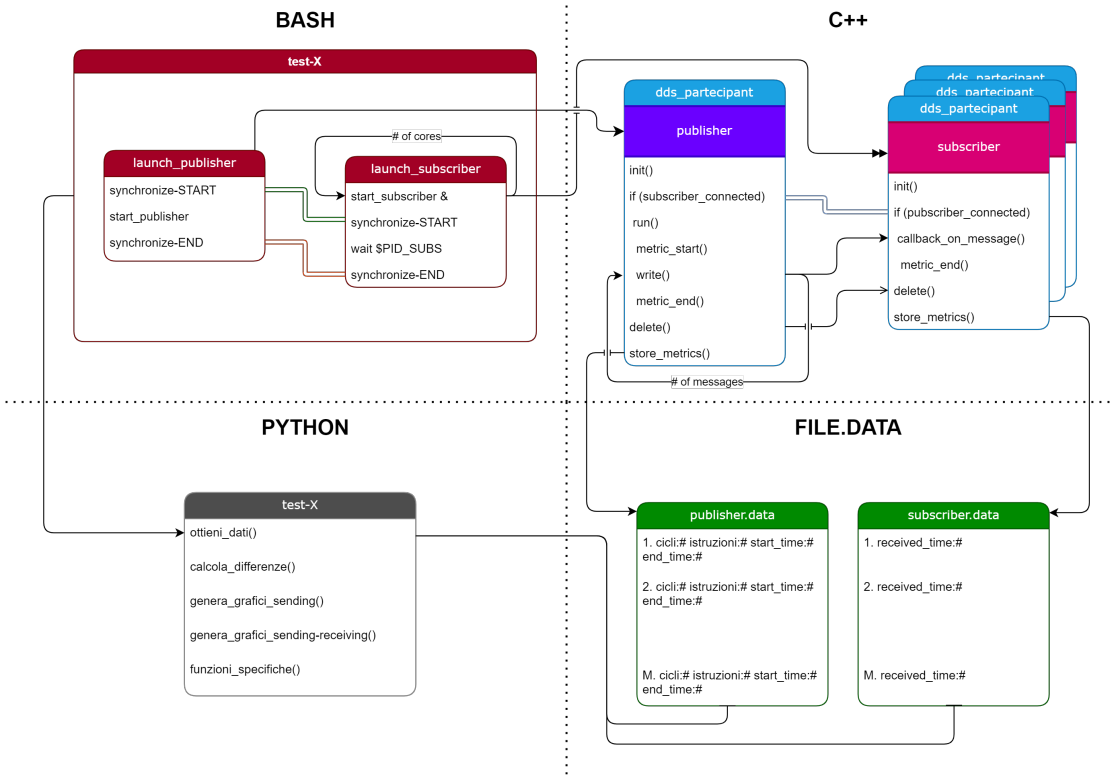



**Figura 5.1.** Struttura test

### 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 avevano il compito di (i) gestire i diversi parametri da passare agli attori, (ii) inizializzare le variabili d'ambiente, (iii) prendere decisioni di quali core dovevano essere utilizzati da ogni partecipante (task-setaffinity) e (iiii) mantenere sincronizzati i test per evitare che alcuni attori venissero inizializzati troppo presto. Infine ripulivano e ordinavano i dati una volta terminato i test andando ad eseguire gli script python che processavano i dati, nelle cartelle corrette.

### 5.1.2 C++

E' stato scelto di realizzare una unica versione di publisher e subscriber in cui cambiavano i parametri con cui venivano lanciati. In questo modo è stato 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: un topic,un publisher ed 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). Per quanto concerne al publisher ed al subscriber, sono state utilizzate delle versioni modificate da quelle proposte nella documentazione ufficiale [15] che andassero sia ad integrare tutte le principali configurazioni possibili, sia integrare alcuni strumenti per l'ottenimento di metriche precedentemente concordate con i collaboratori. Nello specifico sono state scelte per il **Publisher**:

- Tempo di invio;
- Istruzioni Perf-Event;
- Cicli TSC (read_tsc);

mentre per il **Subscriber**, solo il tempo di ricezione con un meccanismo di controllo degli indici del messaggio ricevuto.

### 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 questo tipo. Il valore, viene letto prima e dopo l'istruzione studiata, tramite l'esecuzione della 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 riportato 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 a bassa granularità. In questo caso è stata usata una chiamata alla libreria **perf_event** utile all'ottenimento del *PERF_COUNT_HW_INSTRUCTIONS*.

### 5.1.5 UML

Lo schema UML di funzionamento degli attori DDS, con questo approccio, è riassunto e schematizzato dalla figura 5.2

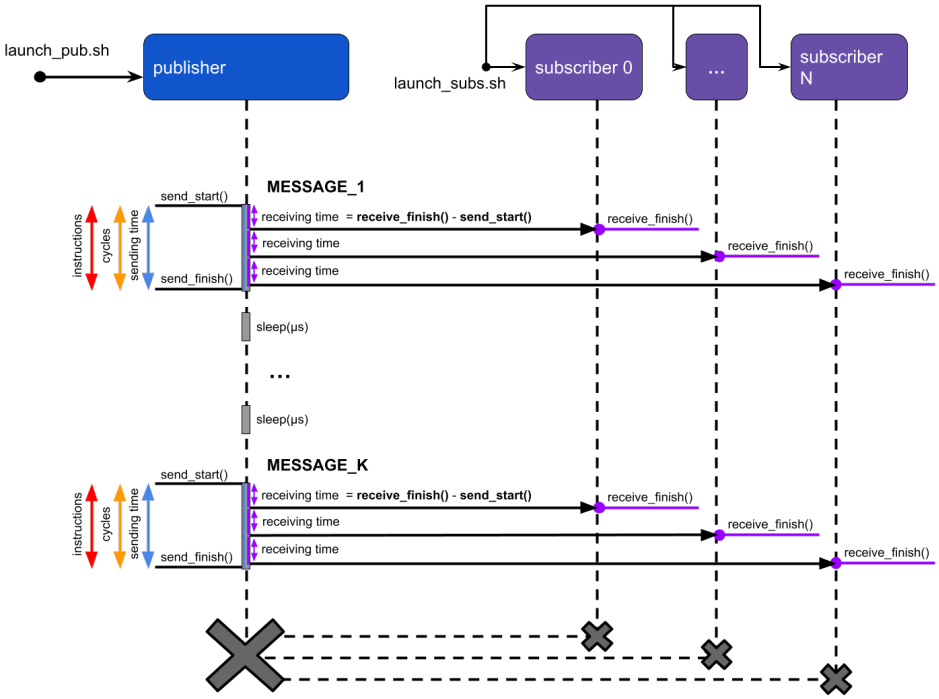



**Figura 5.2.** Schema UML

Come si può vedere, il Publisher 3.2 prima e dopo la write() salva i valori di tempo di invio, istruzioni, e TSC, mentre al lato ricevente, il Subscriber 3.2 salva il tempo di ricezione del messaggio.

### 5.1.6 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 e GPS, è necessario continuare a tenere questi orologi aggiornati. Nei sistemi utilizzati in questo progetto, ed in particolare nei supercalcolatori mostrati in tabella 5.1, lo strumento adottato è Network Time Protocol (fornito da ntpd). Quest'ultimo va a fornire periodicamente agli orologi dei nodi, un tempo di riferimento. Questo intervallo è fissato di default ogni 1024s, ma non disponendo dei diritti necessari, non è stato possibile recuperare questa informazione.

Per ottenere invece le differenze di tempi su sistemi Linux, è ricorrente utilizzare una funzione chiamata **clock_gettime()** che restituisce il tempo all'istante della chiamata. Perciò calcolando la differenza tra due diverse clock_gettime(), si ottiene il tempo trascorso. Il tempo ottenuto, deve avere però un riferimento, non essendo assoluto. Infatti questa funzione mette a disposizione diverse metriche tra cui:

- CLOCK_REALTIME: ottiene il tempo "reale" (ore, minuti e secondi del giorno corrente), sincronizzato nei vari sistemi da ntpd;
- CLOCK_MONOTONIC: ottiene un tempo relativo, da un punto non preciso dall'avvio del sistema;
- CLOCK_MONOTONIC_RAW: ottiene un tempo relativo, da un punto non preciso dall'avvio del sistema, 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 tutte queste è stato deciso di utilizzare MONOTONIC, visto che REALTIME con frequenze di aggiornamento di ntpd ogni 1024s fa in tempo ad accumulare sfasamenti di alcuni millisecondi[22], 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 (con più nodi) per volta, la precisione di MONOTONIC_RAW non è risultata necessaria.

Il problema riscontrato nell'utilizzare MONOTONIC, è che su sistemi con orologi diversi, questi possono essere sfasati anche di diverse migliaia di secondi. Per aggirare questo problema, sono stati usati 2 approcci completamente diversi:

- Sincronizzazione dei nodi
- RTT

### 5.1.7 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 sincronizzare i diversi processi tra di loro. Nello specifico è stata utilizzata una funzionalità chiamata MPI_barrier, che permette di bloccare i processi che ne fanno uso, fino a quando sono tutti sincronizzati, procedendo di conseguenza tutti insieme. Nonostante questo meccanismo fosse particolarmente utile alla sincronizzazione degli orologi non è nato per questo motivo, e racchiudeva per questo alcuni ritardi. Nella figura 5.3 sono stati riportati sull'asse delle Y gli sfasamenti dei tempi di due nodi diversi e sull'asse X il numero dei test effettuati.

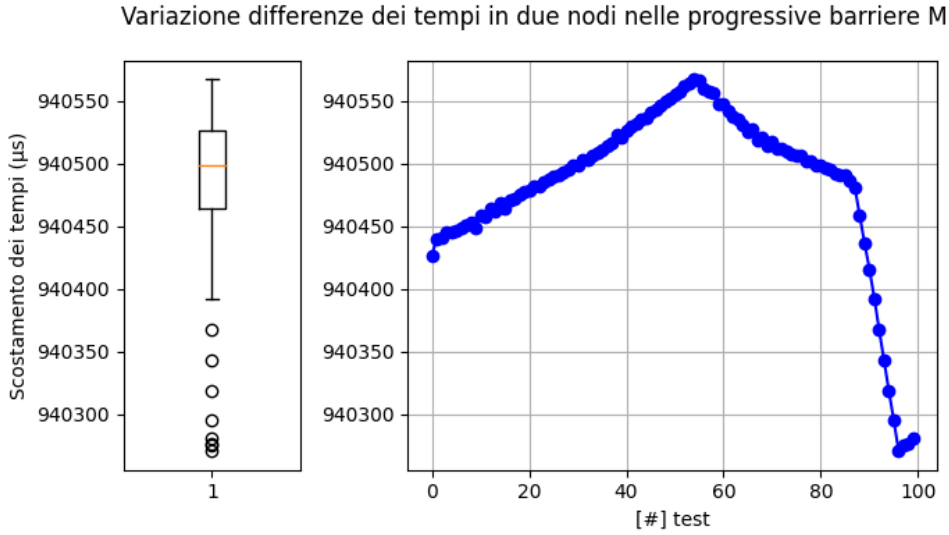



**Figura 5.3.** Scostamento del tempo su nodi diversi

Come possibile notare, nonostante le mpi_barrier, gli scostamenti di tempo tra 2 nodi durante i diversi tentativi effettuati (100), cambia notevolmente, arrivando a differenze fino ad un massimo di 296.659 microsecondi. Questo rende il metodo appena mostrato altamente inaffidabile nel misurare variazioni di tempi dell'ordine di decine di microsecondi.

### 5.1.8 RTT

Nonostante la sincronizzazione fosse idealmente il metodo più preciso per ottenere i tempi di invio-ricezione, essendo l'errore riscontrato, dello stesso ordine di grandezza dei tempi di ricezione, per alcuni test si è utilizzato un approccio che non richiedesse sincronizzazione. Si è scelto quindi di utilizzare il Round Trip Time (RTT). Questa 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 l'invio 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, dei meccanismi di ascolto e risposta. Per semplificarne la comprensione viene riportato lo schema modificato:

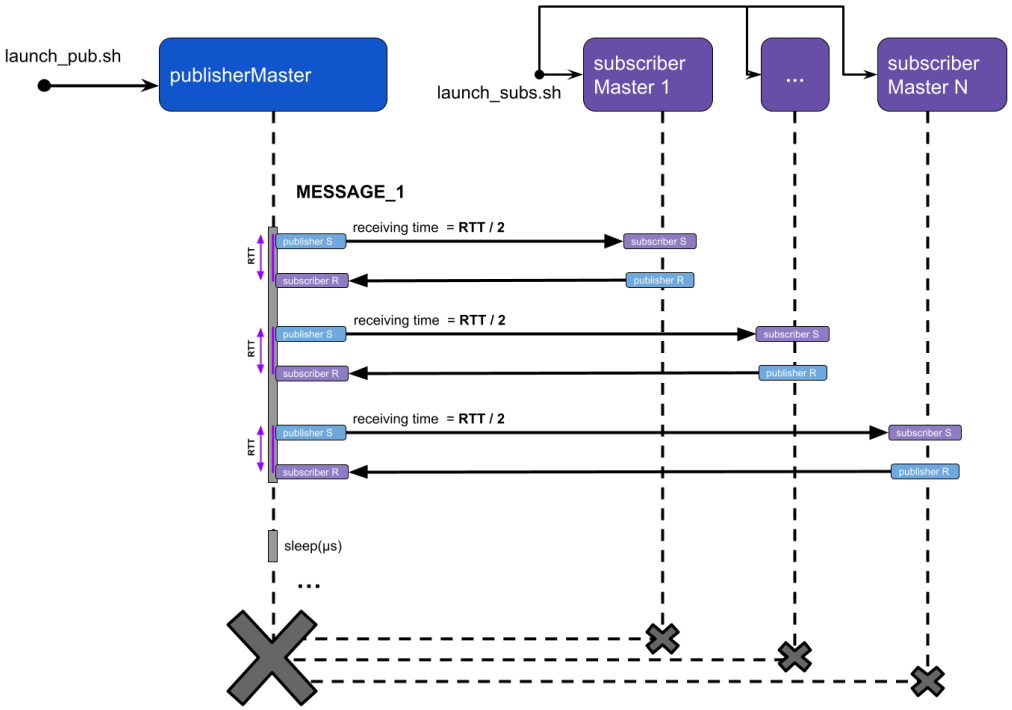



**Figura 5.4.** Schema UML RTT

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).

## 5.2 Python

Gli script python sono stati utili a organizzare e processare tutti i dati oltre a generare tutti i grafici prodotti in questa tesi, considerando che ogni publisher ha generato in ogni test circa 10 mila messaggi, da inviare a tutti e 48 i subscriber, e per ogni protocollo di trasporto, arrivando ad avere per ogni test fino a $1'960'000$ messaggi.

## 5.3 Schema test

I test svolti per il modello di utilizzo e per la caratterizzazione di DDS in sistemi HPC sono stati diversi. Nello specifico sono state pensate 4 categorie di test che comprendessero diverse famiglie di problemi:

- test-0: discovery
- test-1: protocollo di comunicazione
- test-2: partizioni e wildcards
- test-3: throughput

Al fine di condurli nel modo più trasparente possibile sono stati resi disponibili [23] tutti i codici utilizzati durante lo svolgimento di questi test.

### 5.3.1 Discovery

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 (se non esplicitamente configurato). Il problema sorge, quando si hanno numeri elevati di attori, che per eseguire la discovery (comunicazione due a due), generano migliaia di pacchetti con la conseguente crescita di computazioni necessarie. Per questo motivo, in DDS sono stati creati dei meccanismi di discovery centralizzati. In questo esperimento si andranno a comparare le due diverse implementazioni e per farlo si useranno gli strumenti **perf** e **TCPdump**.

### 5.3.2 Protocolli di comunicazione

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

- udp: fornisce la versione v4 e v6 dell'omonimo protocollo di trasporto;
- tcp: fornisce la versione v4 e v6 dell'omonimo protocollo di trasporto;
- udp-multicast: una versione modificata del semplice udp, dove 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;
- 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 sui diversi nodi del supercalcolatore.

### 5.3.3 Wildcards e metodi di instradamento

Un concetto fondamentale nelle comunicazioni tra attori con gerarchie diverse, in sistemi con diverse centinaia di migliaia di entita, come cluster, nodi, processori, 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 costituiscono la scelta predefinita 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 durante il loro utilizzo. Inoltre il topic non permette gerarchie ed è difficilmente modificabile a run-time.

### Partizione

Questo strumento risulta molto interessante, in quanto permette di definire gerarchie e wildcards all'interno di un dominio creando una segmentazione virtuale. Inoltre è facilmente modificabile a run-time.

### Wildcards

Le wildcards sono un costrutto appartenente alle partizioni, che definisce 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.

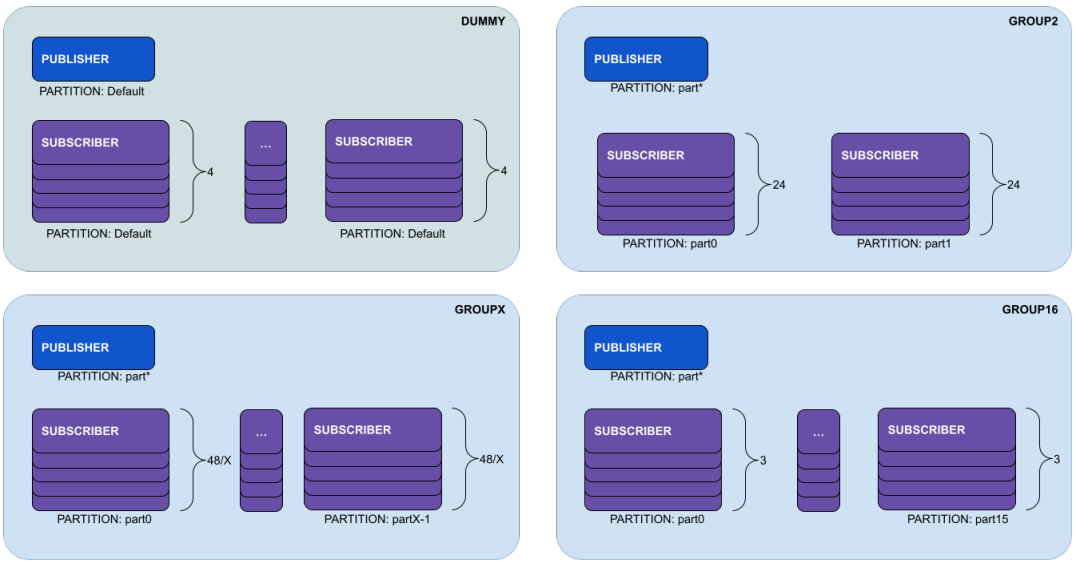



**Figura 5.5.** Schema test wildcards

In questo test si è valutata la differenza in termini di performance dei diversi strumenti, con un particolare focus sulle partizioni e le wildcards, con schema in figura 5.5.

## 5.3.4 Throughput

Nell'ultimo esperimento ci si è focalizzati sulla caratterizzazione di questo middleware ed in particolare è stato misurato il throughput e la frequenza di scambio dei messaggi nei sistemi target. Per farlo, sono stati eseguiti i test su tutti e 4 i protocolli messi a disposizione e con volumi di scambio più elevati, al fine di fornire una panoramica completa.

# Risultati

Nella sezione corrente, vengono riportati tutti i risultati rilevanti ottenuti durante la fase di testing stilando, dove possibile, un modello di utilizzo utile alle finalità di Power Management.

## 6.1 Discovery: centralizzata vs distribuita

Come previsto, nella fase di discovery un numero elevato di entità genera una quantità di di pacchetti scambiati 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 6.1, con il numero di pacchetti sull'asse Y e il numero di entita sull'asse X, è possibile notare un distacco tra i due approcci, con sole 30 unità. Nella figura 6.2 con i cicli sulle Y, numero entità sulle X, e nella figura 6.3 con istruzioni sulle Y, viene confermato l'andamento non lineare.

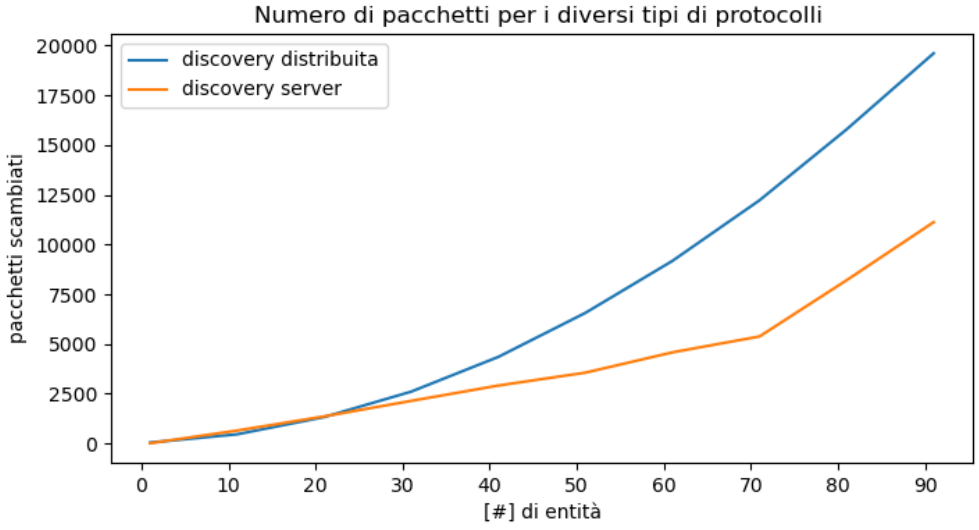



**Figura 6.1.** Numero di pacchetti scambiati durante la discovery all'aumentare di entità

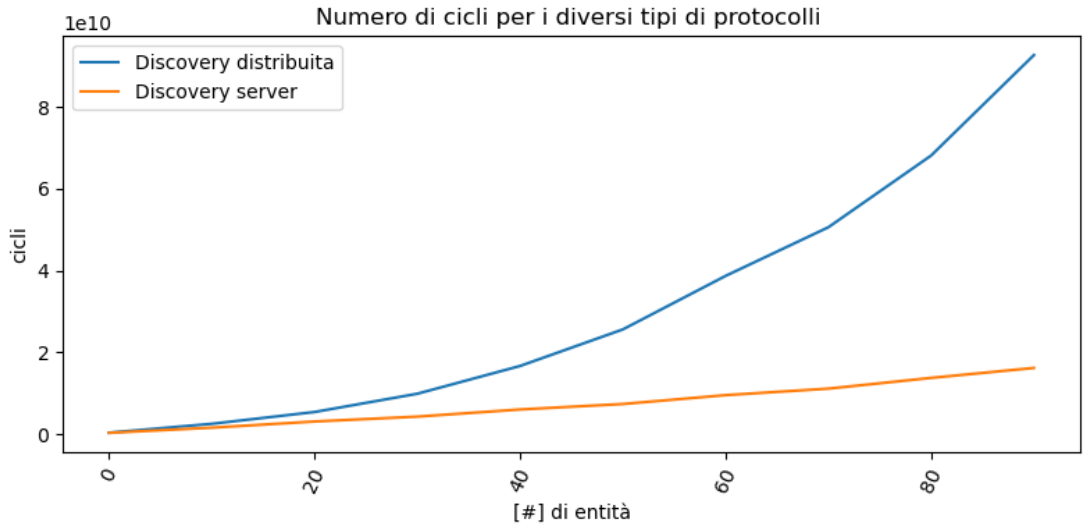



**Figura 6.2.** Numero di cicli durante la discovery all'aumentare di entità nelle diverse opzioni

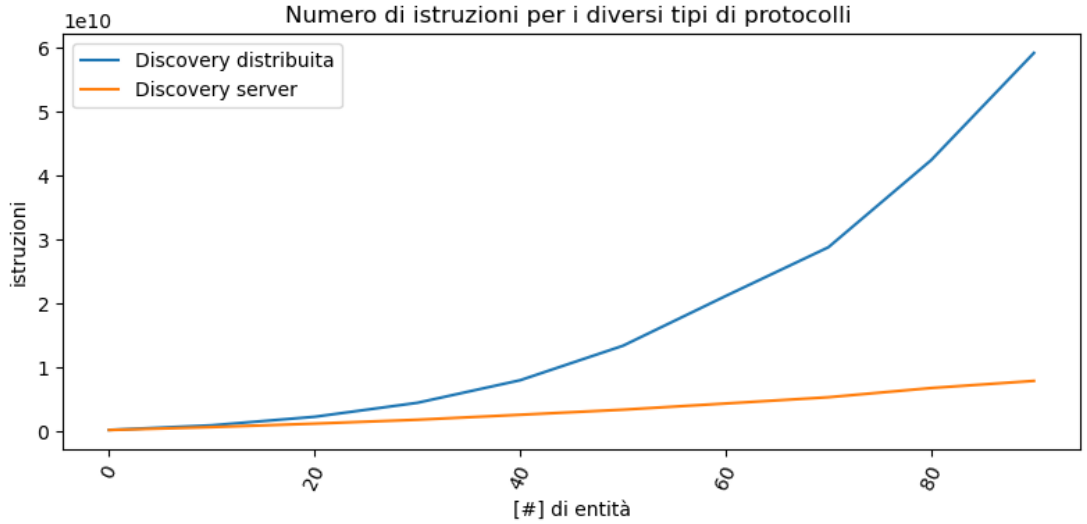



**Figura 6.3.** Numero di cicli durante la discovery all'aumentare di entità nelle diverse opzioni

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 Scalabilità del numero di subscriber 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*. Questo viene dimostrato nella figura 6.4 (con i tempi medi di invio sulle Y) e 6.5 (con latenza media di ricezione in asse Y). In tutte le figure sull'asse X viene riportato il numero di subscriber che cresce fino a 40. L'effetto visto è particolarmente pronunciato anche in protocolli come *UDP* che non possiedono concetti di connessione. Questo potrebbe essere dovuto al costo di inizializzazione ed invio dei messaggi, ad indirizzi di rete diversi.

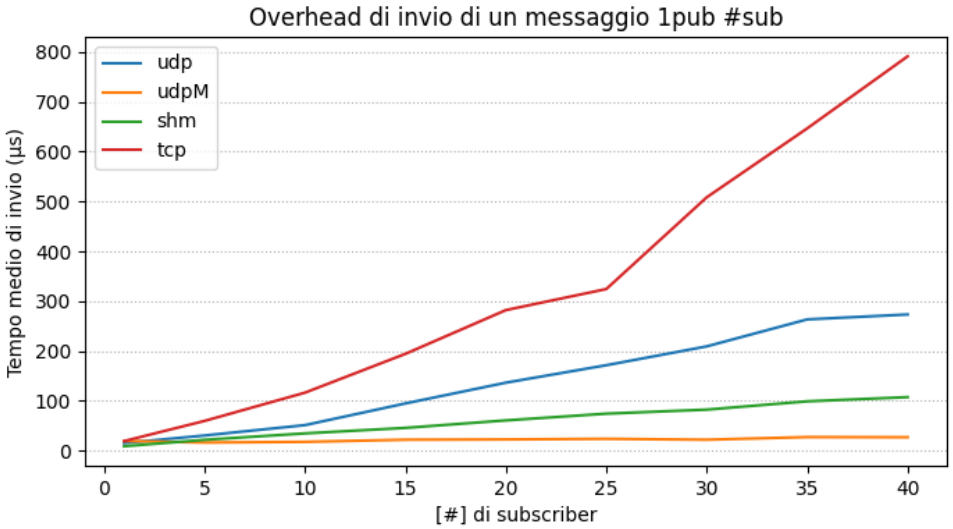



**Figura 6.4.** Overhead sulla publish all'aumentare dei subscriber

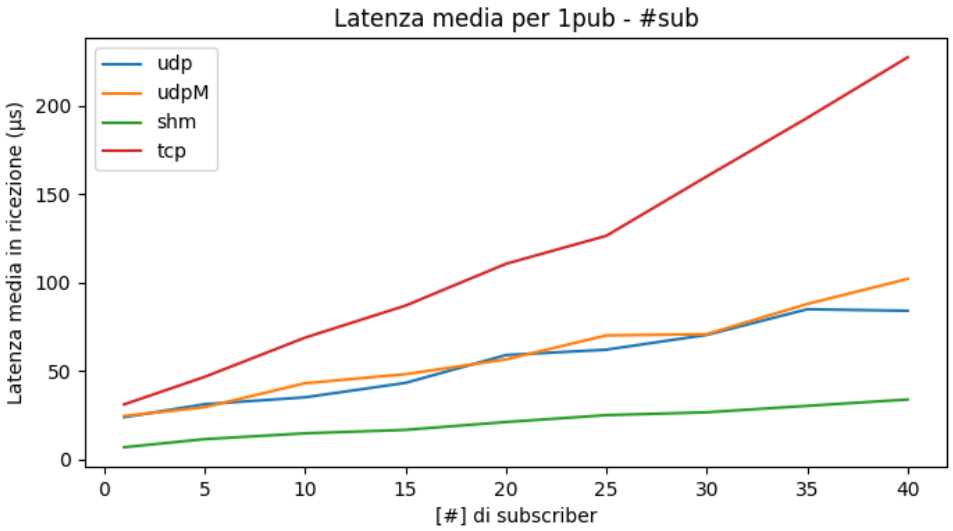



**Figura 6.5.** Latenza di ricezione all'aumentare dei subscriber

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

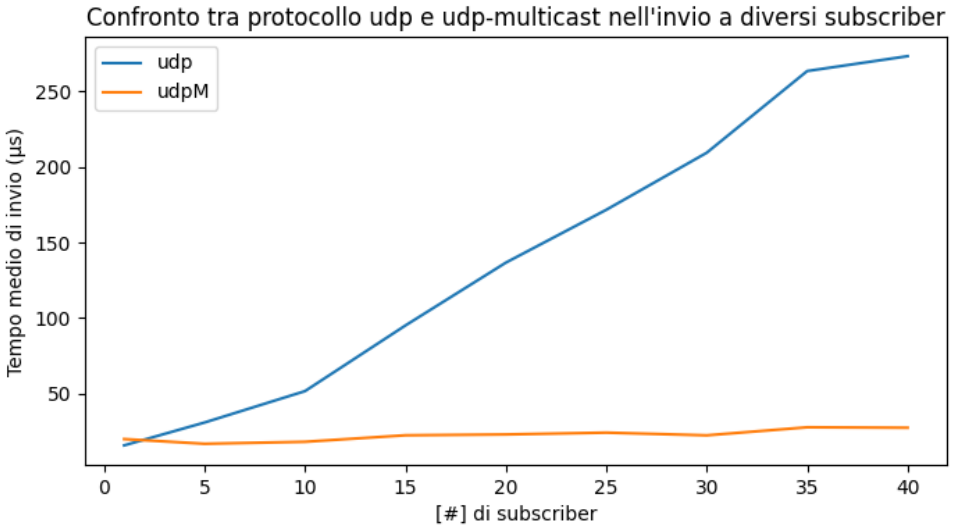



**Figura 6.6.** Tempo di invio di un messaggio a [#] subscriber nei protocolli UDP e UDP Multicast

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 Overhead sul 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 5.4). In figura 6.7 dove sugli assi Y viene mostrato il tempo di ricezione in microsecondi, e sull'asse X la sequenza dei messaggi, viene mostrato il problema citato.

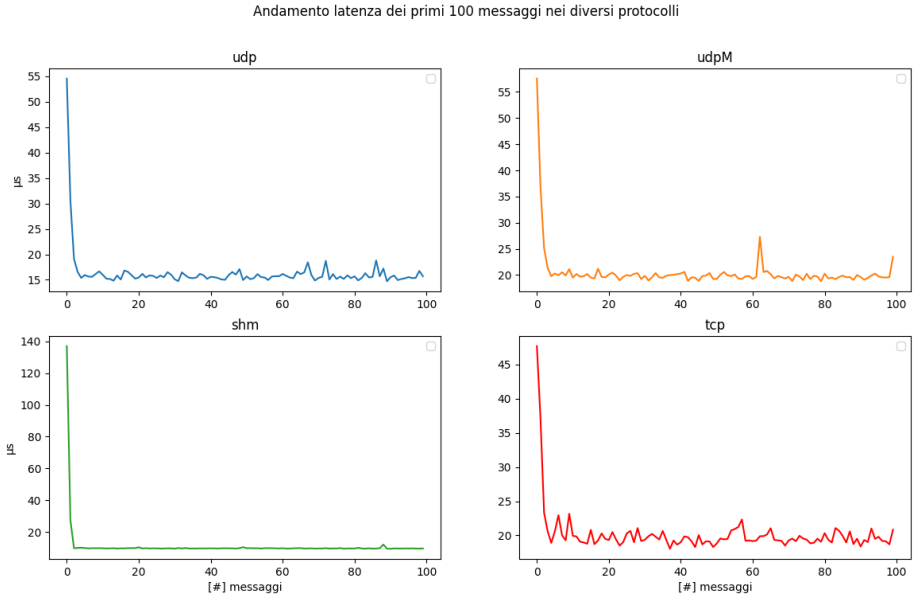



**Figura 6.7.** Overhead del primo messaggio nei vari protocolli

Infine, vista la mancanza di approfondimento verso tale problema, non si può concludere che l'invio di pochi messaggi sia sfavorito in questa implementazione.

## 6.4 Protocolli di comunicazione

Visti gli schemi 2.4 è risulta probabile che in un 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*. In figura 6.8 sugli assi X il numero di subscriber, e sulle Y le latenza di ricezione in microsecondi.

[1] comunicazioni effettuati in localhost o in shared memory

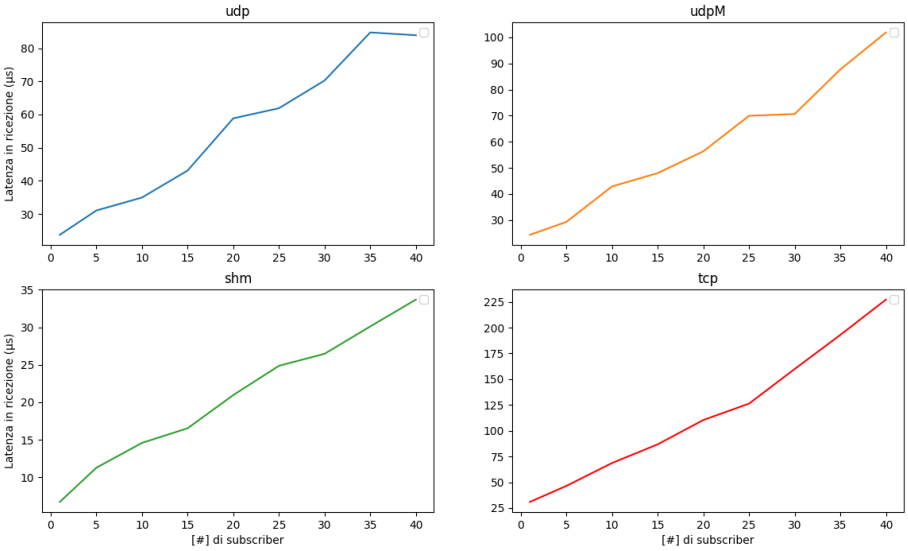



**Figura 6.8.** Differenza della latenza di ricezione dei vari protocolli con

Invece, in figura 6.9 e figura 6.10 sugli assi X i protocolli, mentre sulle Y per il primo solo le latenza di ricezione in microsecondi, e nel secondo sia latenza di ricezione che overhead di invio.

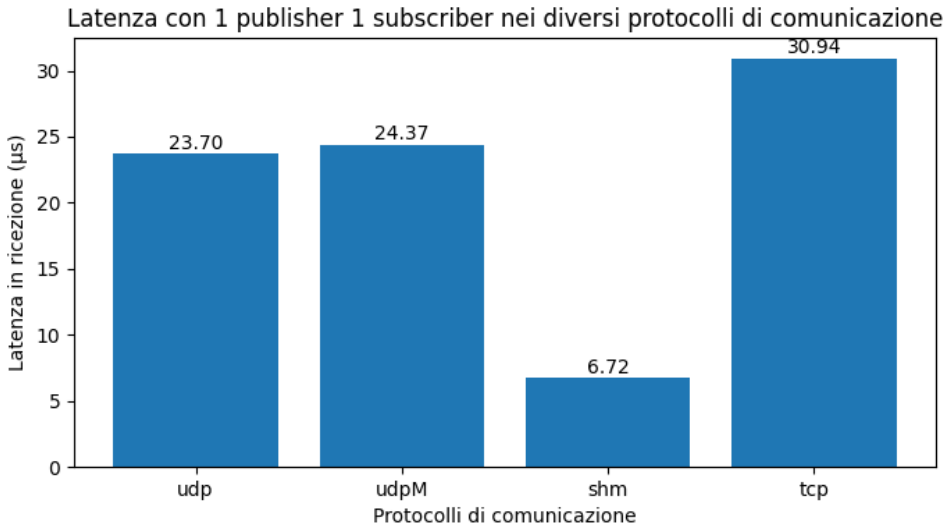



**Figura 6.9.** Latenza media di ricezione di un singolo messaggio nei vari protocolli

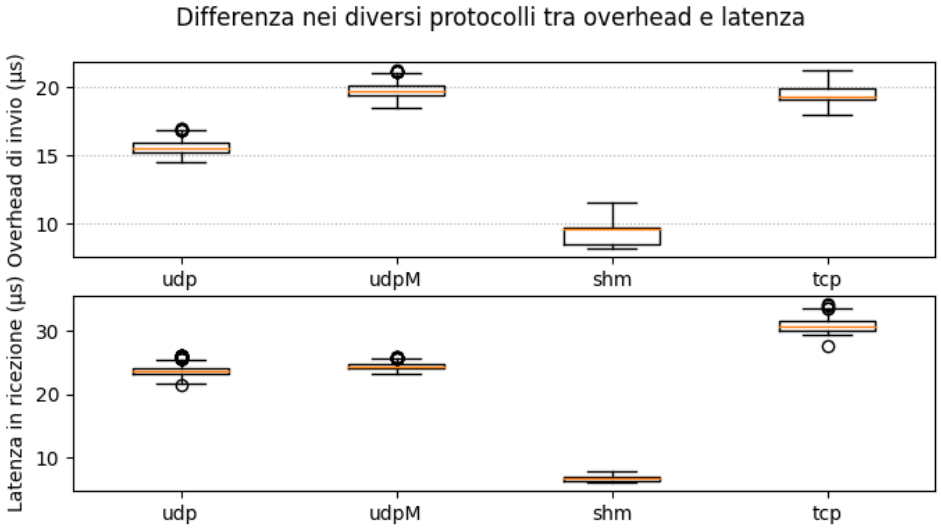



**Figura 6.10.** Diagramma a scatola nei vari protocolli di comunicazione con un publisher e un subscriber

Infine, in figura 6.11 sugli assi X i protocolli, mentre sulle Y il conteggio di cicli e istruzioni ($10^6$).

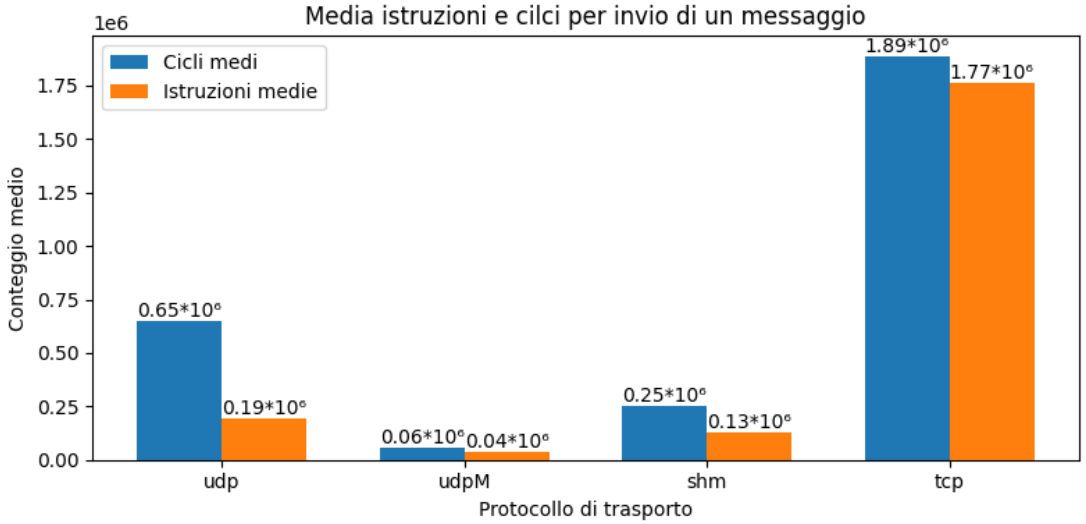



**Figura 6.11.** Conteggio cicli e istruzioni per ogni protocollo

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 (6.12). 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. In figura 6.12 sugli assi X il numero crescente di subscriber, e sulle Y conteggio medio di cicli ($10^6$).

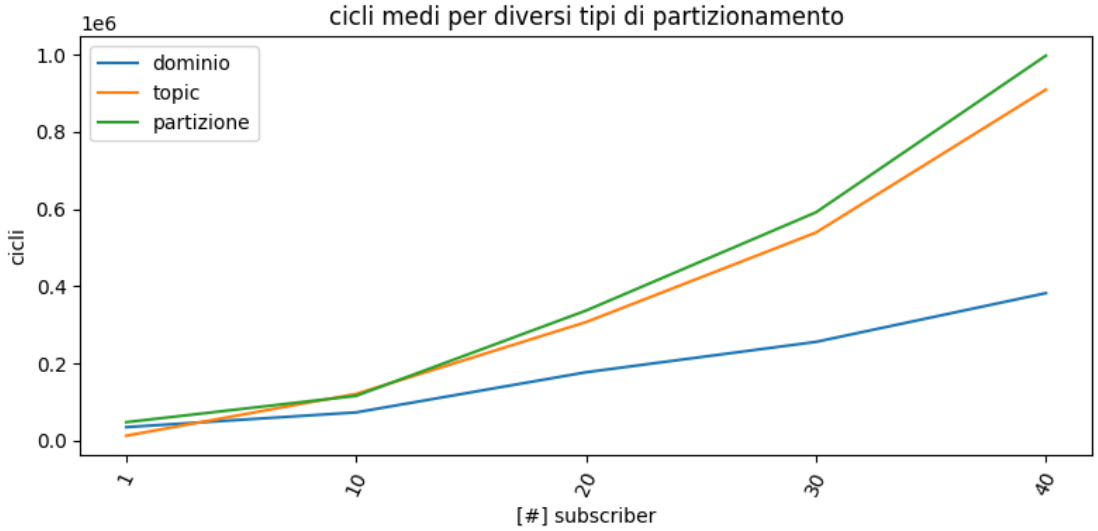



**Figura 6.12.** Peso in termini di cicli nell'usare partizioni topic e domini

Per quanto riguarda entità che necessitano di gerarchie, è conveniente usare le partizioni con le Wildcards visto che non hanno un peso significativo come notabile in 6.13 dove sull'asse X vi è il numero di partizioni usate, mentre sulle Y il tempo di invio. Infine lasciando i topic come mezzo di partizionamento, per il tipo di dati, e il tipo di istruzioni da utilizzare.

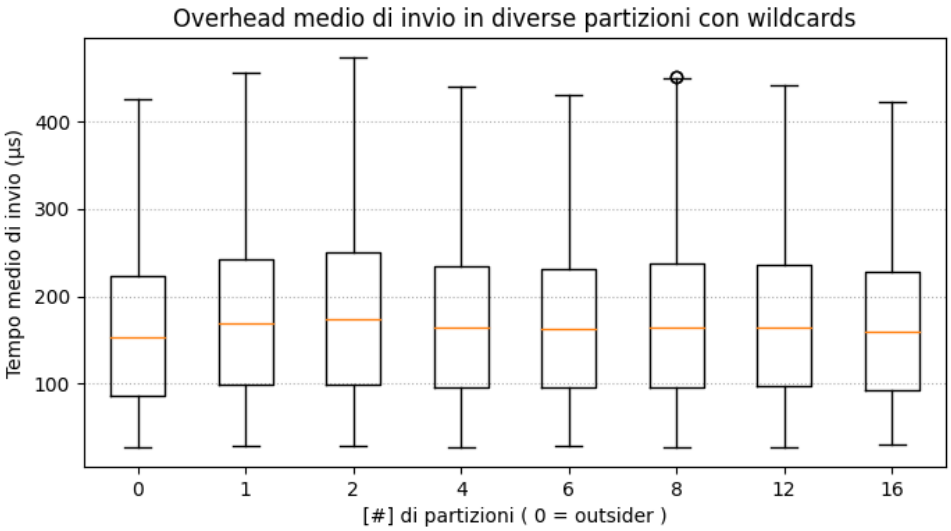



**Figura 6.13.** Differenza in tempi di invio con utilizzo di wildcards in diverse partizioni

Nella figura 6.14 sulle Y il conteggio di istruzioni e cicli ($10^6$) rispetto al numero di partizioni (sulle X).

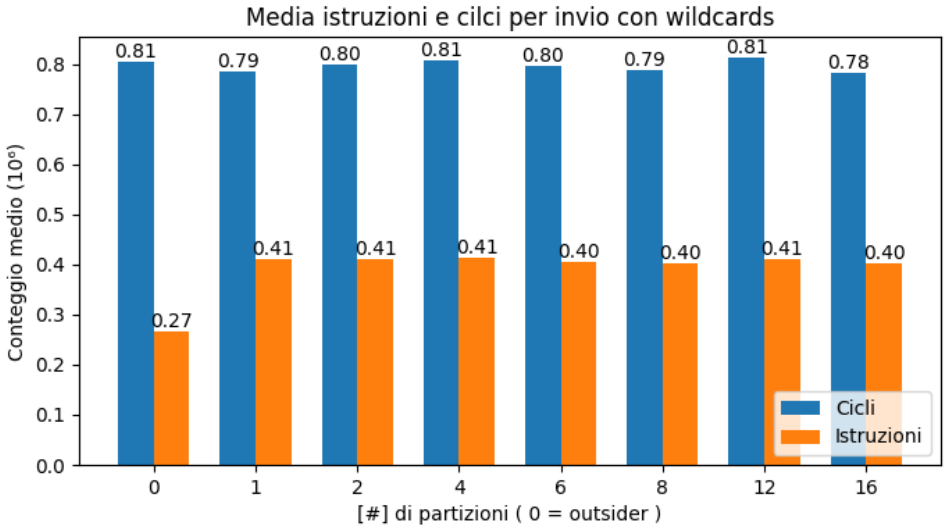



**Figura 6.14.** Cicli e istruzioni necessari per invio con wildcards in diverse partizioni

## 6.6 Throughput

L'ultimo test, rivolto alla caratterizzazione della capacità di comunicazione, del middleware DDS, ci permette di capire il massimo scambio di dati che può avvenire tra i vari

attori del Power Management. Nella figura 6.15 l'asse Y, a sinistra, mostra in verde i il valore di KMessaggi/s [2] raggiungibili dai vari protocolli di comunicazione (in asse X), mentre in grigio, a destra i KByte/s. Dalla figura si può vedere che è possibile inviare ogni secondo da un publisher ad un subscriber più di un MB/s di dati. Inoltre è possibile raggiungere frequenze di scambio di messaggi dell'ordine di decine di KHz (6.15). Inoltre, nel caso di invii a più subscriber, come si può vedere in figura 6.17, mentre i protocolli udp shm tcp saturano all'aumentare del numero di subscriber, il protocollo UDP Multicast, permette una crescita lineare del throughput.

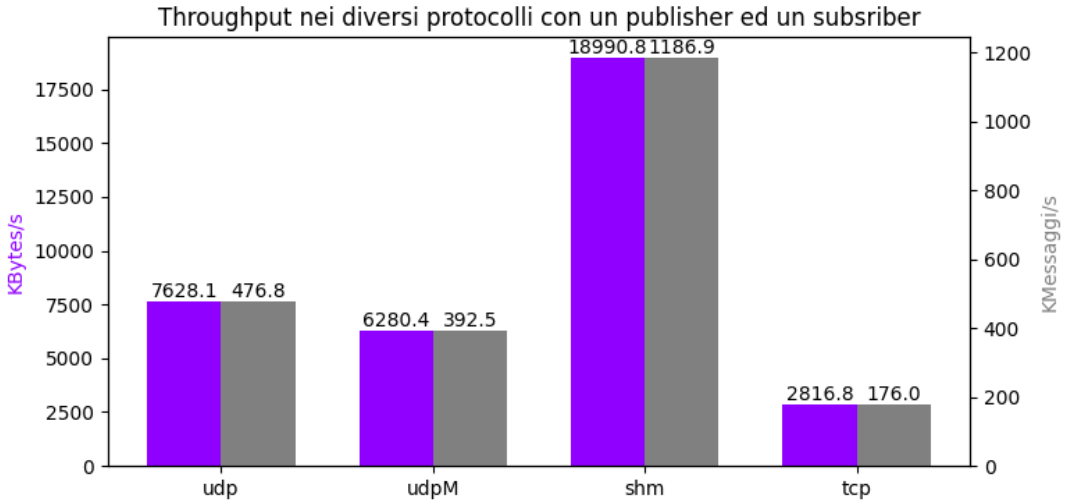



**Figura 6.15.** Throughput con un publisher e un subscriber

[2] 1000 Messaggi al secondo

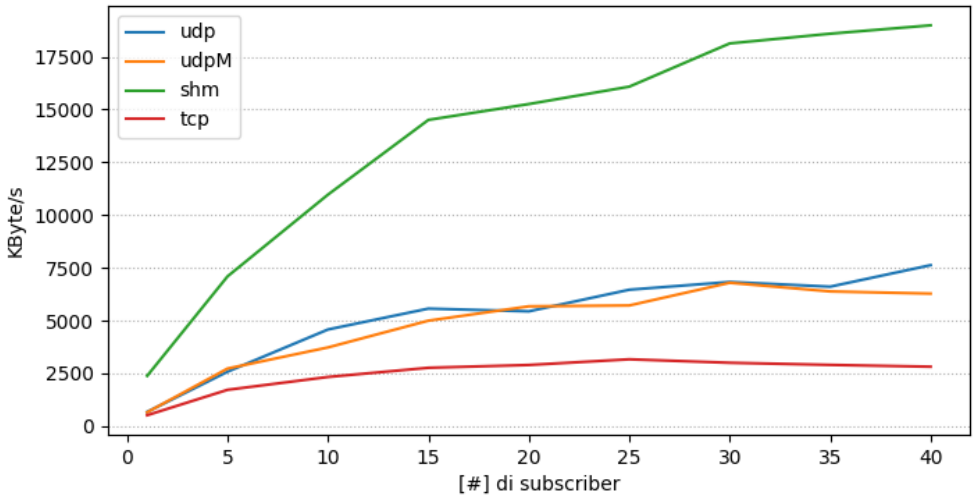



**Figura 6.16.** Throughput a crescenti numeri di subscribers

Il risultato ottenuto nelle figure 6.15 6.16 (KBytes/s sulla Y e numero di subscriber sulla X), per quanto riguarda il protocollo UDP Multicast, è influenzato dalla struttura del test mostrato in figura 5.4, che non permette ad UDP Multicast di mostrare le sue potenzialita, come invece si può vedere dalla figura 6.17.

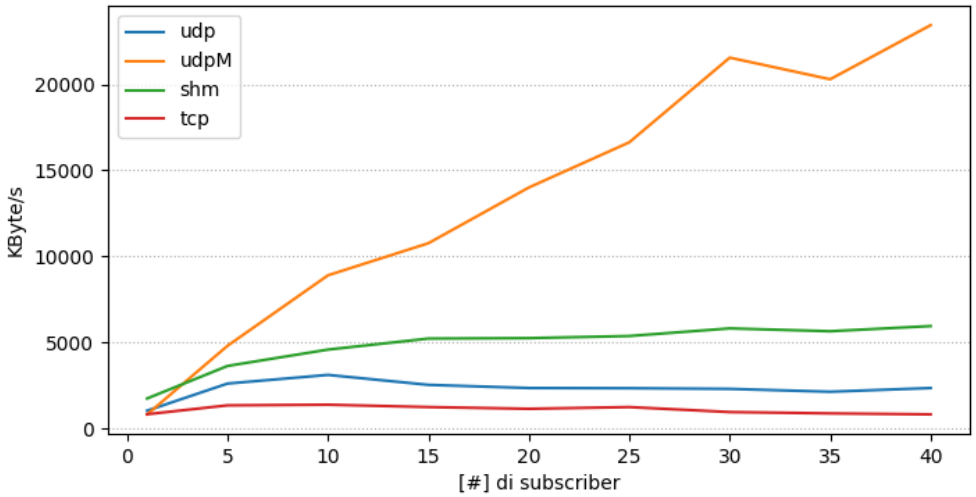



**Figura 6.17.** capacità di invio in KByte/s da un publisher a molteplici subscriber

# Realizzazione dei prototipi di Power Stack

Nel corso di questa tesi con la collaborazione dei membri del progetto REGALE, come Cineca[5], E4[7] e BSC[6] sono stati sviluppati i middleware ed i 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.

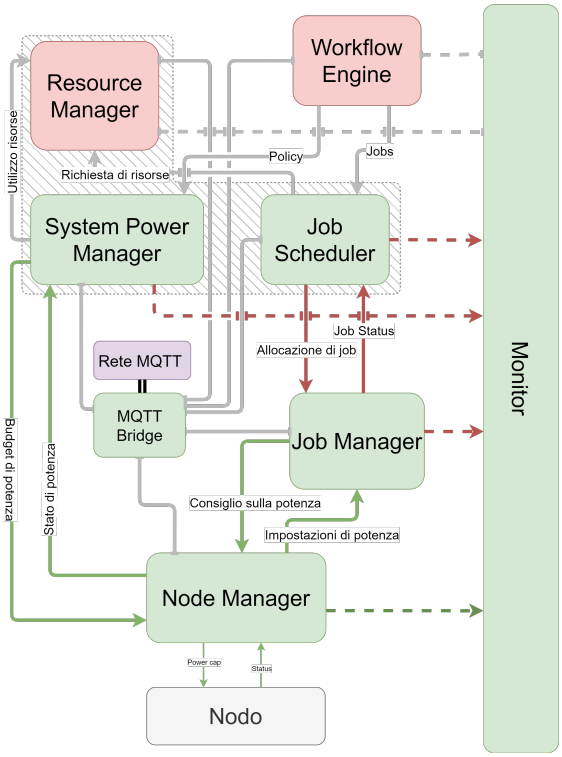



**Figura 7.1.** Schema componenti sviluppati: in verde completato, in rosso ancora da sviluppare, in grigio non previsto

L'infrastruttura DDS creata, chiamata *REGALE Library*[24] al momento permette di utilizzare i vari tipi di comunicazione, configurazioni di qos, e anche vari tipi di dati scambiati, tutti impostabile tramite file **XML**. I prototipi invece, sono dei programmi (c++) che vanno a simulare uno scambio di informazioni realistico. Al momento per motivi di semplicità utilizzano dati di tipo *uint32_t* ed il loro comportamento si può riassumere nel seguente modo.

## Implementazione Job Scheduler

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.

## Implementazione Job Manager

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.

## Implementazione System power manager

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

- GET_INFO
- GET_POWER
- SET_POWER

## Implementazione Node manager

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_FREQ_NODE
- SET_POWER_NODE

### Implementazione Monitor

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[25]. 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.

## 7.1 Struttura

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

● = PUBLISHER DOMAIN PARTITION TOPIC
● = SUBSCRIBER DOMAIN PARTITION TOPIC

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

| DUMMIES | |
|---|---|
| **NAME** | **USED** |
| NODE MANAGER DUMMY | ● P 0 default Monitor_report_job_telemetry<br>● P 0 default Monitor_report_node_telemetry<br>● P 0 default Monitor_report_cluster_telemetry |
| JOB SCHEDULER DUMMY | ● P 0 default SystemPowerManager_get<br>● P 0 default SystemPowerManager_set<br>● S 0 default SystemPowerManager_get_reply<br>● S 0 default SystemPowerManager_set_reply |
| JOB MANAGER DUMMY | ● S 0 default NodeManager_get<br>● P 0 default NodeManager_get_reply |

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

# Conclusioni

Per concludere, dopo aver presentato in modo esaustivo tutti gli strumenti utilizzati per eseguire i test, con obbiettivi e metodo di sperimentazione, sono stati mostrati i risultati degli esperimenti fatti. In primo luogo si è visto l'andamento esponenziale nella fase di discovery, in presenza di molteplici partecipanti, risolvibile tramite la sua implementazione di server. In secondo luogo, sono stati analizzati e caratterizzati i protocolli, di cui il più performante si è rilevato essere quello di Shared Memory con una latenza media di ricezione messaggi di appena 6.72 µs, utilizzabile però solo in presenza di memorie condivise. Al secondo posto troviamo UDP con 23.70 µs e UDP Multicast con 24.37 µs. Per ultimo TCP con 30.94 µs di media, che però offre garanzia di ricezione. Escludendo la shared memory, per le comunicazioni tramite rete si sono raggiunti throughput di 7.63 MB/s con frequenze di invio a 476 KHz.

Per quanto concerne al test sul partizionamento dello scambio di informazioni, si è visto come i domini, sono quelli con minore impatto sulle performance, seguito dai topic ed infine partizioni. Inoltre, le minime differenze sia in termini di performance, che di tempi, nell'usare le wildcards, permette un ampio uso di comunicazioni di tipo gerarchico. Infine, insieme a questi risultati è stato fornito anche un modello di utilizzo di questo middleware caratterizzato per HPC.

Come ultimo passo, sono stati mostrati tutti i prototipi di Power Stack, insieme alle sue strutture, che comunicano tramite il middleware DDS chiamato REGALE Library prodotto in collaborazione i membri del progetto.

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

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

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

[8] Linux. *CPUfreq*. URL: https://www.kernel.org/doc/Documentation/cpu-freq/.

[9] Kashif Khan et al. «RAPL in Action: Experiences in Using RAPL for Power Measurements». In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 3 (gen. 2018). DOI: 10.1145/3177754.

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

[11] 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.

[12] 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.

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

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

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

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

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

[18] *An open architecture to equip next generation HPC applications with exascale capabilities*. https://cordis.europa.eu/project/id/956560.

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

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

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

[22] 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.

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

[24] RegaleLibrary. *REGALE Library*. URL: https://gricad-gitlab.univ-grenoble-alpes.fr/regale/tools/regale/-/tree/fastdds-regale-thesis.

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

# Elenco delle figure