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

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 che ha 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 software 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 ad 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.

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,che si può a sua volta suddivisa 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 pad dei chip;
- 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 GPU e TPU. Tutti questi questi hanno a disposizione memorie di capienze elevate, e ad alta banda. Andando a unire tutti cluster insieme si ottengo capacità computazioni che nei giorni nostri hanno raggiunto ordini del ExaFlops (10^{18} operazioni di Floating Point per secondo). 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[1]. 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, rendendo sempre più

di vitale importanza i software in grado di automatizzarne la gestione. Dall'arrivo degli exa-computer, la potenza necessaria per alimentare questi sistemi ha superato la precedente soglia dei 20MWatt[1]. Se poi si considera che la maggior parte della potenza fornita, viene convertita in calore, si deve prendere in considerazione anche i consumi necessaria per tenere raffreddati i sistemi. Se non adeguati, comporterebbero grandi inefficienze in termini di energia, che si traducono anche in degradazioni di prestazioni computazionali. Considerando tutto, i centri che ospitano queste macchine necessitano di decine di MWatt di potenza per ogni exa-supercomputer 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 situati 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 in grado di gestire la potenza assorbita di macchine HPC.

Mentre sono state proposte diverse tecniche per colmare questo bisogno, la maggior parte di esse si è rivelata essere una soluzione 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.

Stato dell'arte

Un Power-Stack deve gestire e monitorare la potenza assorbita, le frequenze e le temperature di processori all'interno dei sistemi. 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 sono rese disponibili due tipi di interfacce: Nella figura 2.1 viene schematizzata la differenza tra le due interfacce disponibili.

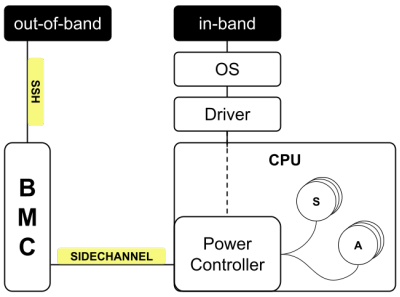


Figura 2.1. Differenza tra le due interfacce

- in-band
- 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, tuttavia a causa della loro natura altamente specializzata, tali approcci non saranno inclusi 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 tramite sistema operativo, le manopole per gestire e monitorare frequenze e informazioni della cpu. Queste ultime possono essere gestite in modo manuale, in automatico in base al carico di sistema oppure in risposta ad eventi ACPI. Una volta scelti i driver come *ACPI CPUfreq driver* e *Intel P-state* è possibile

scegliere tra diversi governors (o governatori) ognuno che agisce con delle policy specifiche. Per esempio in *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;

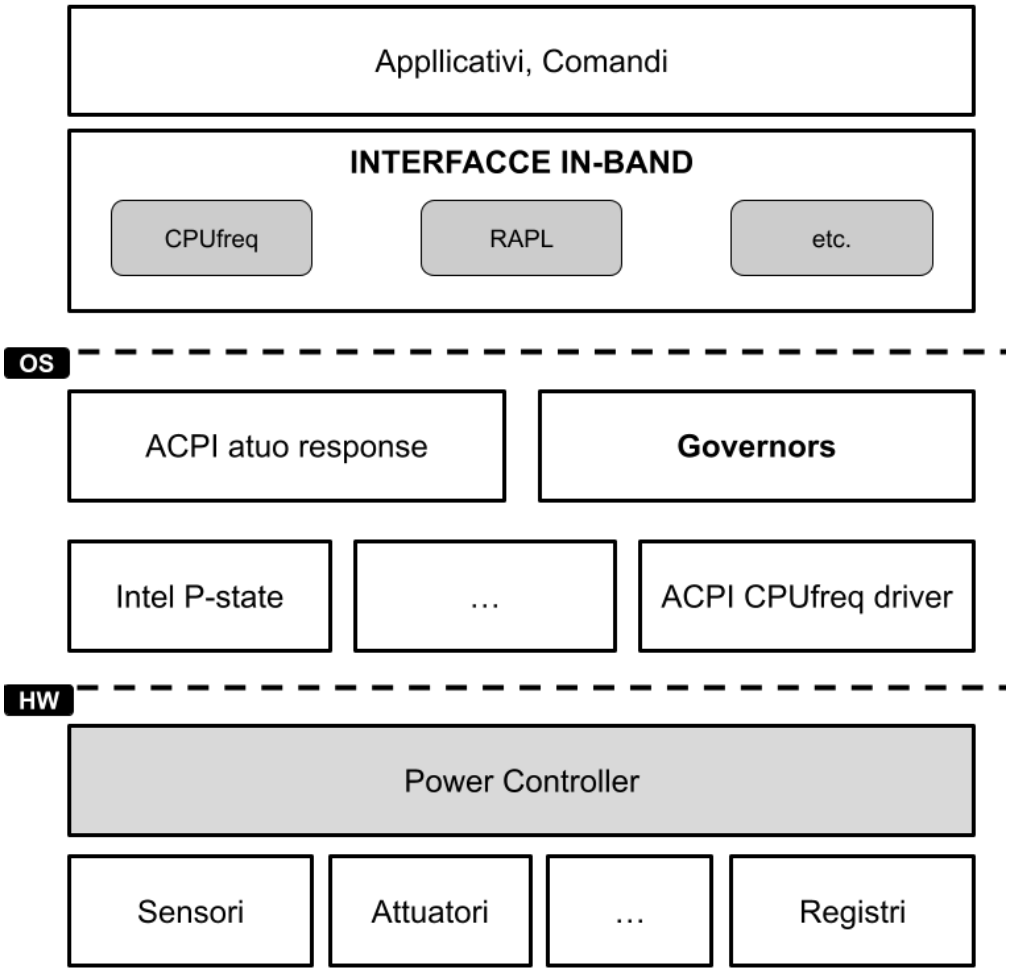


Figura 2.2. Struttura interfacce in-band: divise su più livelli tra cui Sistema Operativo (SO) e Hardware (HW)

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 solo le informazioni dei core sui quali i richiedenti vengono eseguiti.

2.0.2 Servizi out-of-band

Contrariamente alle interfacce in-band, le out-of-band fanno utlizzo di *sidechannels* ovvero canali di accesso alternativi per ottenere le informazioni richieste. Questo meccanismo permette di accedere ad informazioni da processi esterni dal processore del quale si vuole impostare o reperire dei dati. Per di più questo permette di monitorare i componenti anche quando ci sono errori ed eccezioni che normalmente bloccherebbe il workflow. Un componente tra i più famosi che svolge questa funzione è il Baseboard Management Controllers (BMC), solitamente un microcontrollore 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.

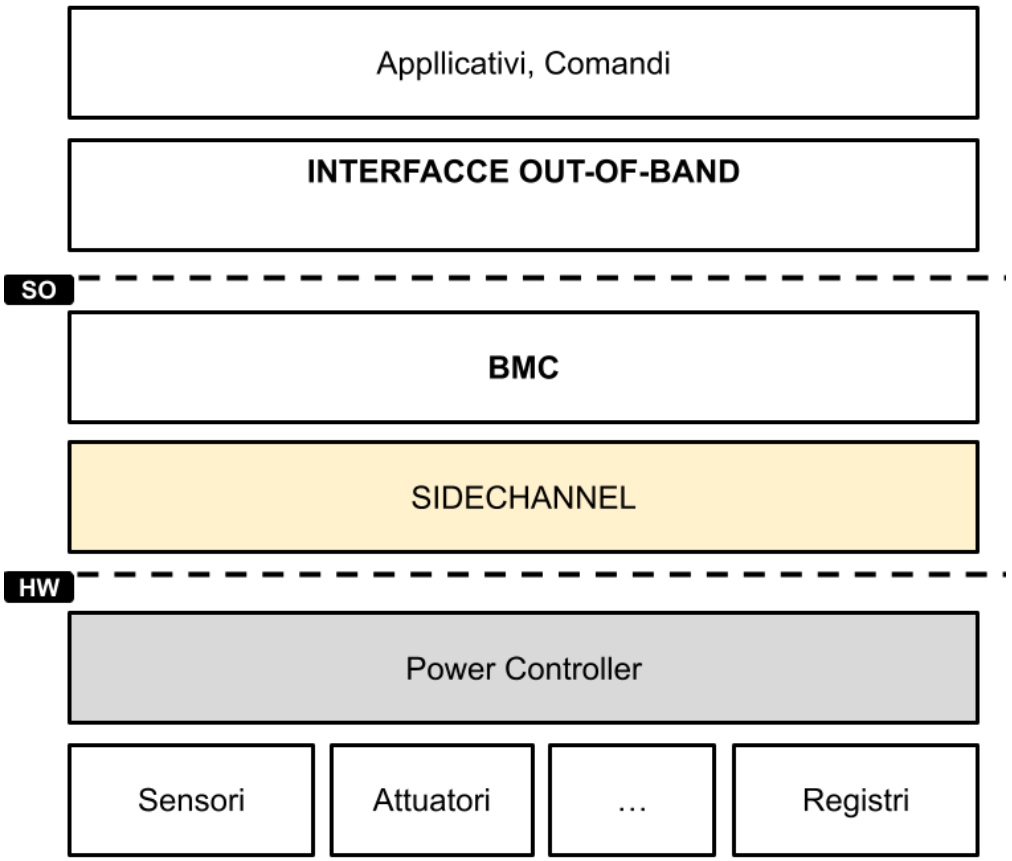


Figura 2.3. Struttura interfacce out-of-band

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)[2], e *HDEEM* (Atos)[3]. 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 applicazioni software che cooperando riescono a fornire ad applicazioni, utenti e amministratori gli strumenti per un servizio 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 Manager (SM)
- Job Manager (JM)
- Resource Manager (RM)
- Node Manager (NM)
- Monitor (M)

2.2.1 Workflow engine

Il workflow rappresenta un flusso di lavoro composto da diversi task che deve essere svolto 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 negli specifici jobs che verranno assegnati al system-manager.

2.2.2 System Manager

Il System Manager 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 contemporanea una parte chiamata 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 gli 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 schedulers

Il job scheduler ha il compito di assegnare e condividere le risorse computazionali e fisiche del sistema HPC, ai vari utenti che lo utilizzano. 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 dallo scheduler.
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 privilegio di gestire le risorse di un centro di calcolo. 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 linea con quanto il job ha necessita di essere eseguito.

Infine in alcuni casi il RM è anche responsabile di gestire elettricità e raffreddamento dei centri di calcolo.

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 2.4 che vuole mostrare la gerarchia e le possibili interazioni tra i vari attori.

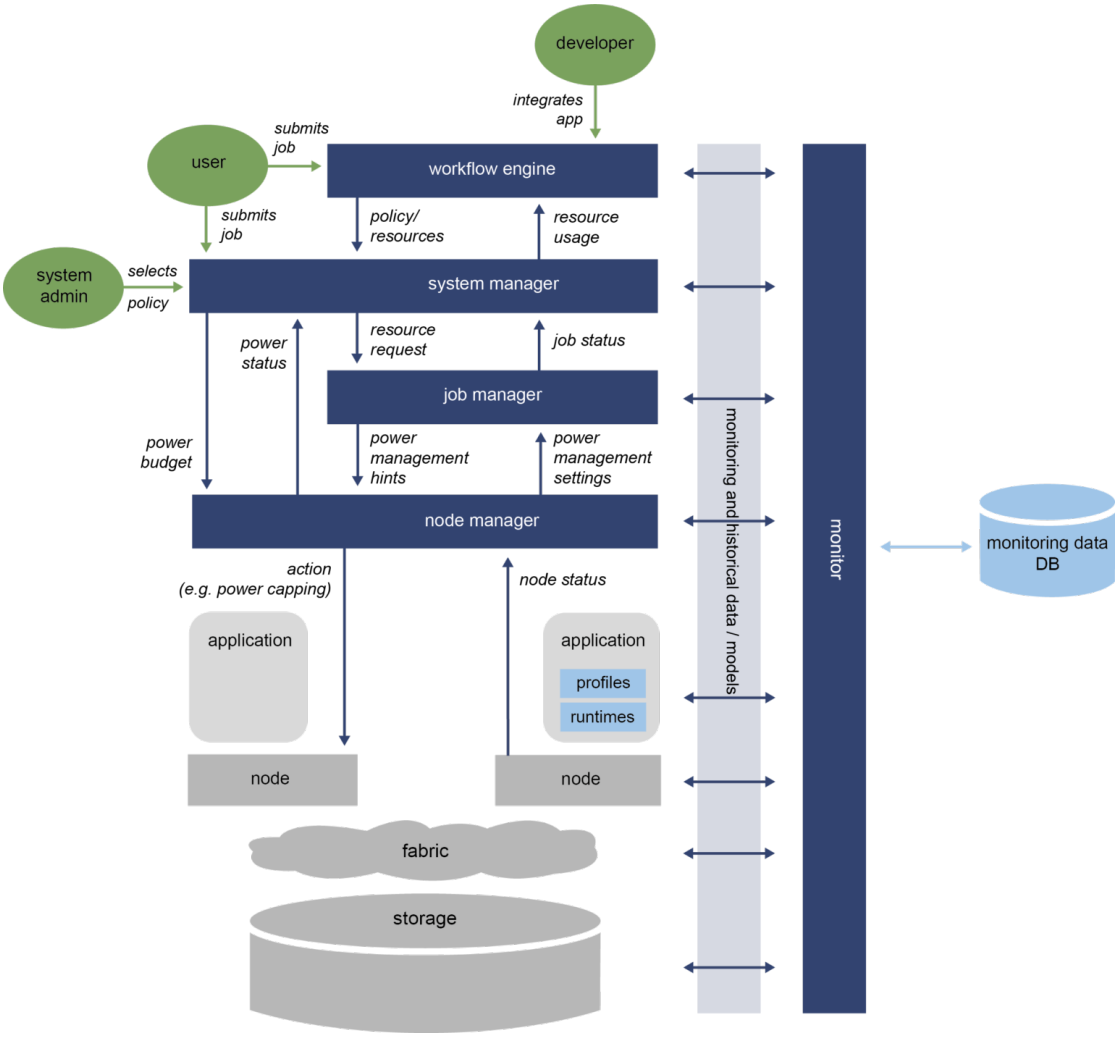


Figura 2.4. Modello di power stack

DDS & RTPS

DDS (Data Distribution Service)[4] e RTPS (Real-Time Publish-Subscribe)[5] costituiscono due soluzioni fondamentali nel campo delle comunicazioni distribuite e real-time. Queste tecnologie svolgono un ruolo importante nella la trasmissione di dati tra dispositivi e applicazioni interconnesse, rivestendo particolare importanza in scenari complessi come i sistemi embedded, in IoT e applicazioni ad alte prestazioni come l'HPC (High-Performance Computing).

3.1 Implementazione usasta

DDS e RTPS sono dei protocolli di comunicazione per specifici casi di utilizzo. Ci sono state diverse implementazioni di questi protocolli da diversi società e organizzazioni, come:

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

e tante altre. In tutti i successivi capitoli verrà preso come riferimento FastDDS ed in particolare la sua versione 2.11.2 [6]. E' stato scelto di utilizzare questa implementazione dato il supporto per le comunicazione Real-Time, e le impostazioni delle Qualità del servizio(QoS) che la rendevano perfetta per un utilizzo su sistemi di HPC.

3.2 DDS

Data Distribution Service è un protocollo di comunicazione 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 cosidette Qualità di Servizio (QoS policy) che servono configurare il comportamento di ognuno di questi attori.

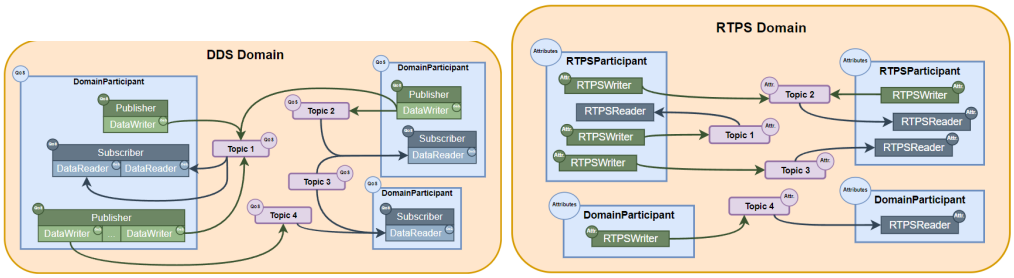
3.3 RTPS

Real-Time Publisher Subscribe protocol è un protocollo-middleware 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 communcazioni 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;
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 la distribuzione dei dati tra vari attori utilizzando un middlware basato su DDS non è idea inedita.Infatti dalla sua seconda versione il software open-source **Robot Operating System** meglio conosciuto come ROS ha deciso di usare questi strumenti introducendo un ulteriore livello che permette di cambiare varie 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[1] sopra il quale la community ha creato le implementazioni di DDS desiderate, andando a creare così diverse possibilità di implementazione dello stesso servizio di distribuzione dati. Inoltre per cambiare tra le diverse versioni di DDS si deve semplicemente impostare una variabile di ambiente, rendendo estremamente facile per tutti i possibili fruitori di ROS.

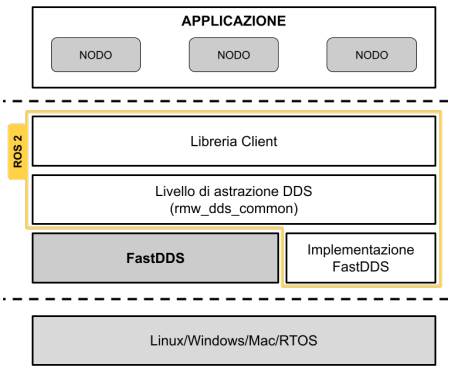


Figura 3.2. Ros Middleware per DDS

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

REGALE

REGALE[7] è un progetto Europeo 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 progettare ed implementare uno stack software open-source di Power Management olistico in grado di operare in sistemi ad alte prestazioni. 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, SLURM, or DCDB. Ulteriormente, Regale, ha deciso di considerare l'introduzione di molti software open-source che potessero soddisfare le esigenze modello di Power Stack 2.4. Infatti sono stati valutati e selezionati i software (molti dei quali prodotti dai partner) con i seguenti ruoli mostrati in tabella 4.1. A questi mancano solo il *Workflow*

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

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. Ruoli dei partner all'interno di REGALE e architettura di base

engine.

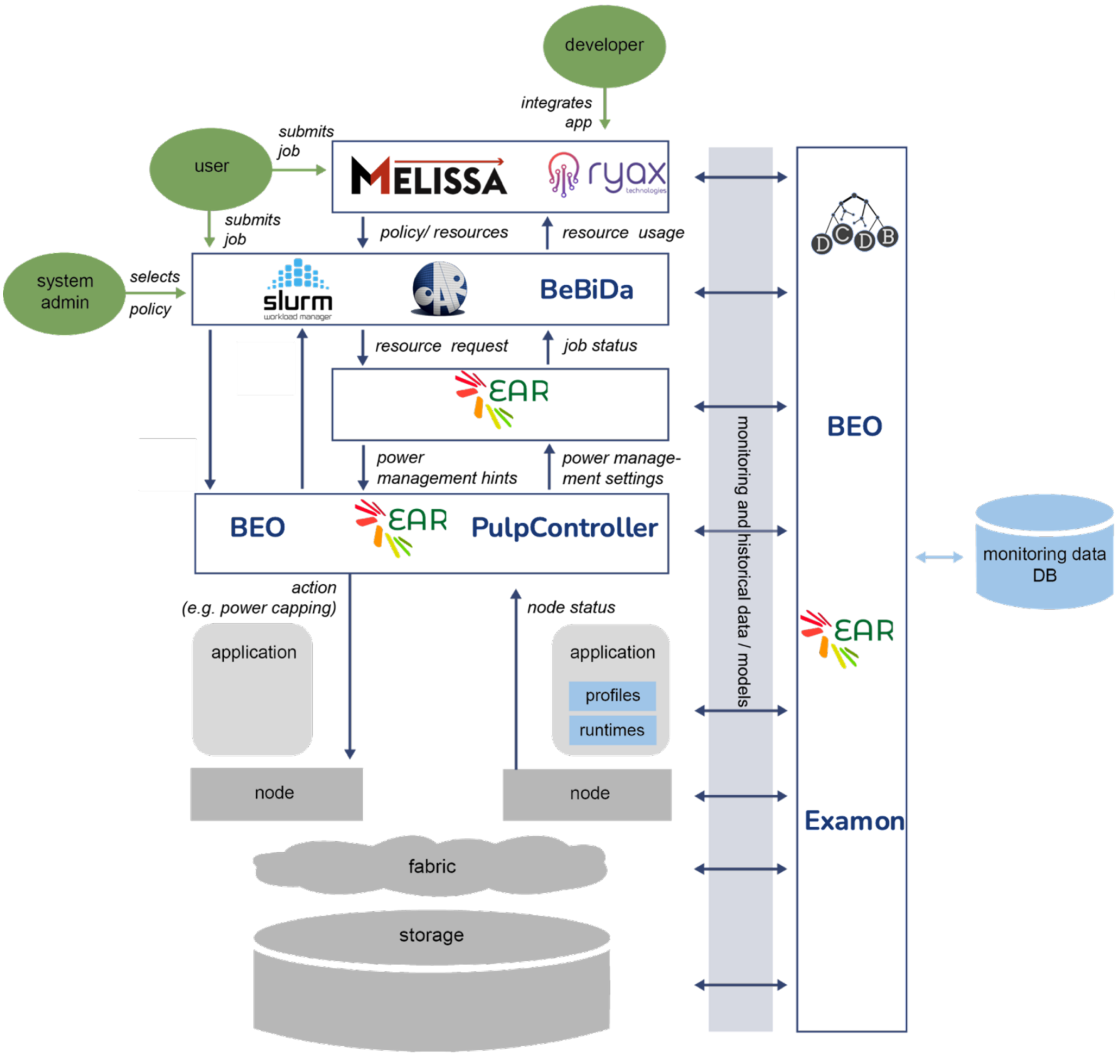


Figura 4.1. Copertura componenti REGALE

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 isolati. 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**. Questa tesi è nata in collaborazione con REGALE, ed è stata stilata anche per riportare test utili alla finalità di REGALE.

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[1] e uno da E4[1] nel tentativo di ottenere risultati affidabili. Di seguito le specifiche dei sistemi utilizzati:

Parameter	**Cineca**	**E4**
Number of nodes used	3	3
Processor	Intel CascadeLake 8260	Intel CascadeLake 8260
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	

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.

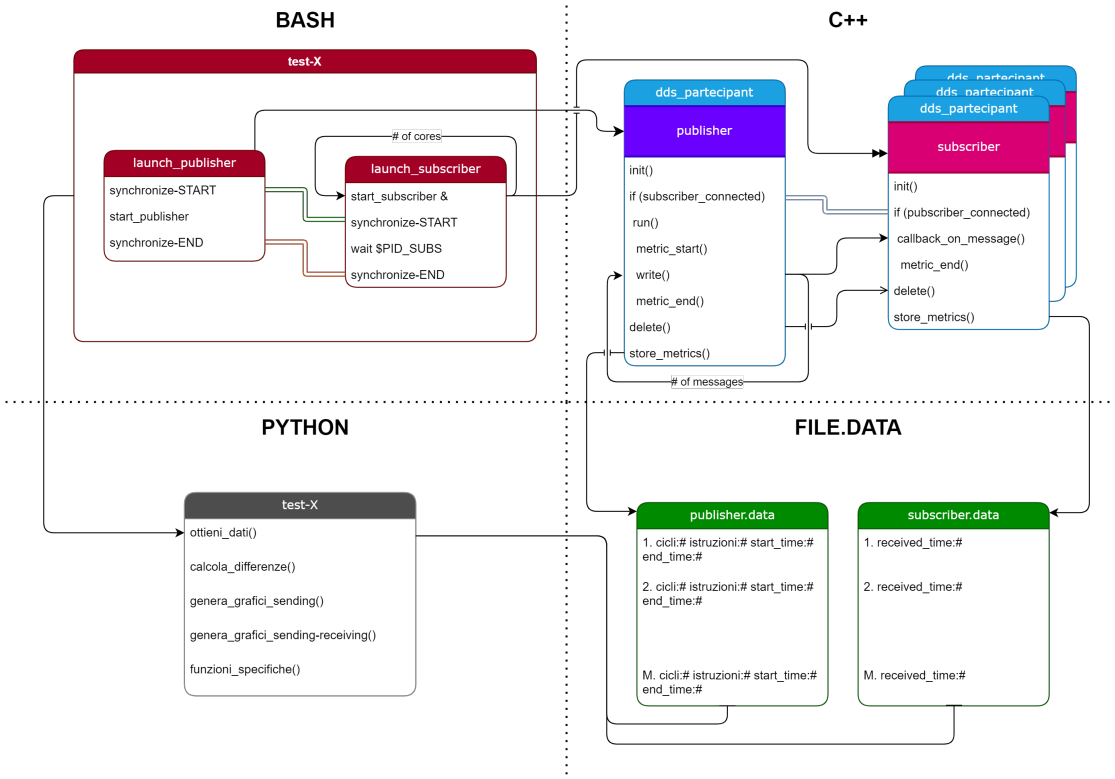


Figura 5.1. Struttura test

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 parametri variabili da passare agli attori, inizializzavano le variabili d'ambiente, decidevano quali core dovevano essere utilizzati da ogni partecipante (task-setaffinity) 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 utilizzare direttamente l'implementazione DDS invece che il già citato *Ros middlware* [3.4] 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;
- 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;
- Comodità: Implementare completamente *rmw_dds_common* richiedeva un impegno e uno studio non indifferente della architettura sottostante a ROS, che seppur ben documentata, sarebbe costata molto tempo in più.

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, rendendo 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[1]. 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 5.2, 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[8], 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

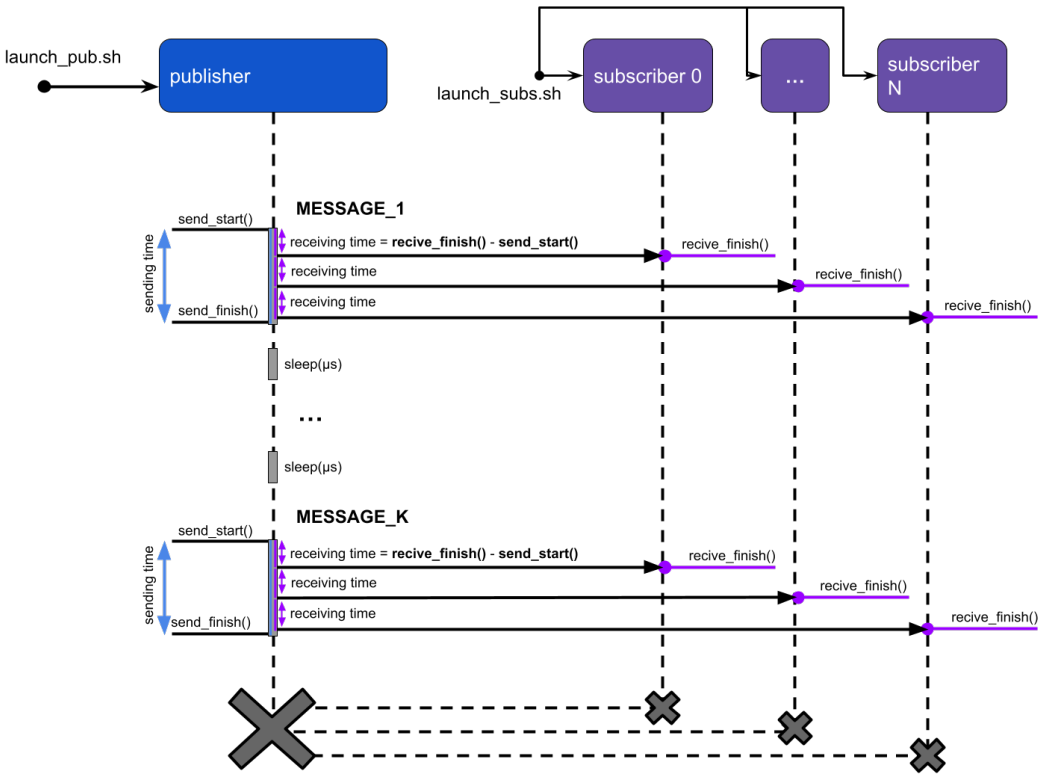


Figura 5.2. Schema UML

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 10K messaggi da inviare a 48 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.

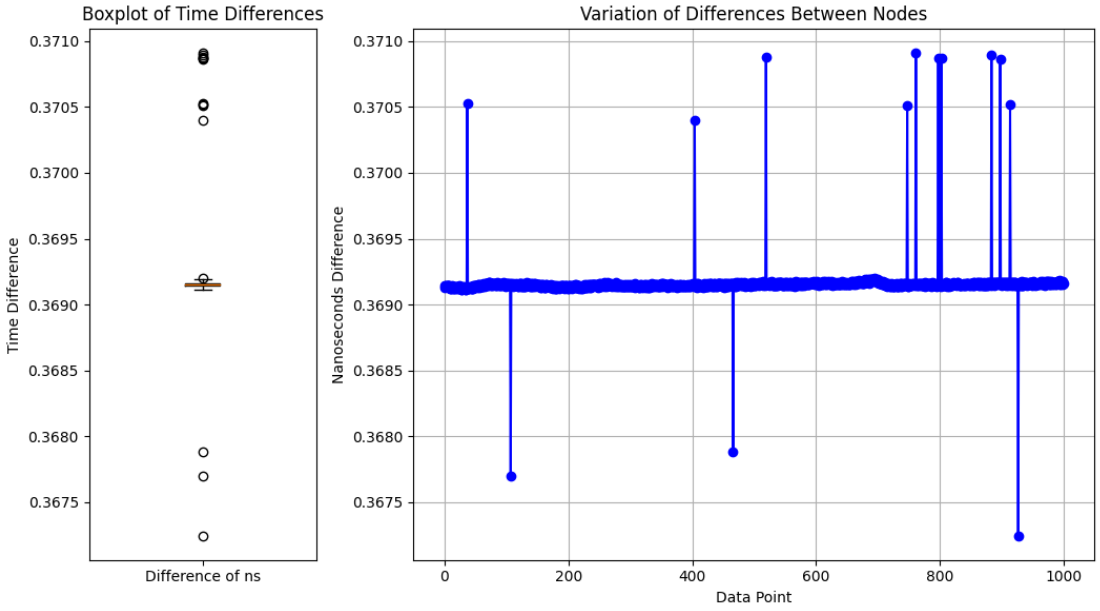


Figura 5.3. Scostamento del tempo su nodi diversi

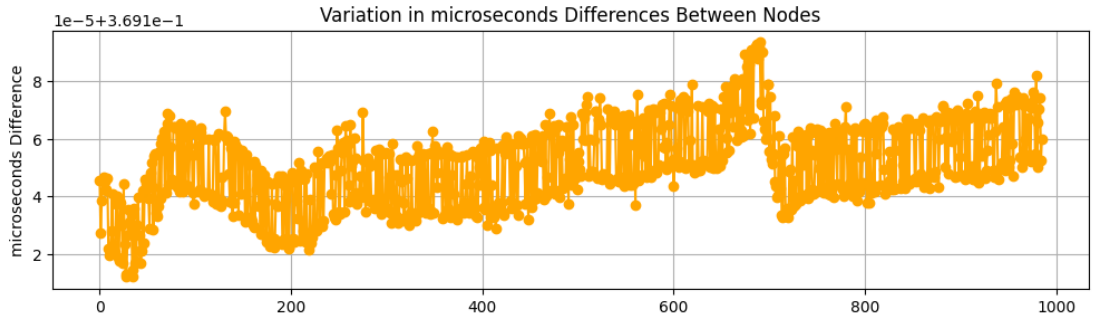


Figura 5.4. Scostamento senza outliers

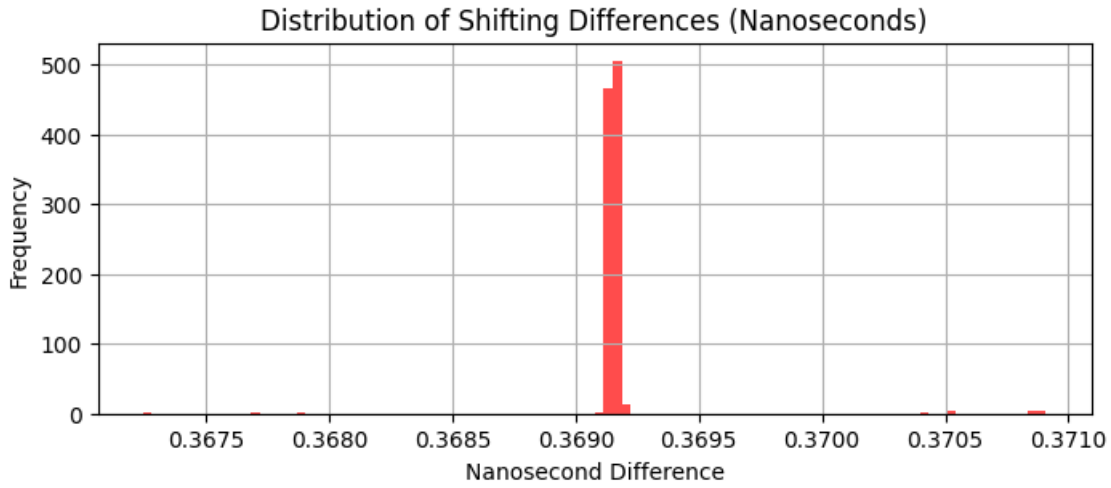


Figura 5.5. Distribuzione delle differenze

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:

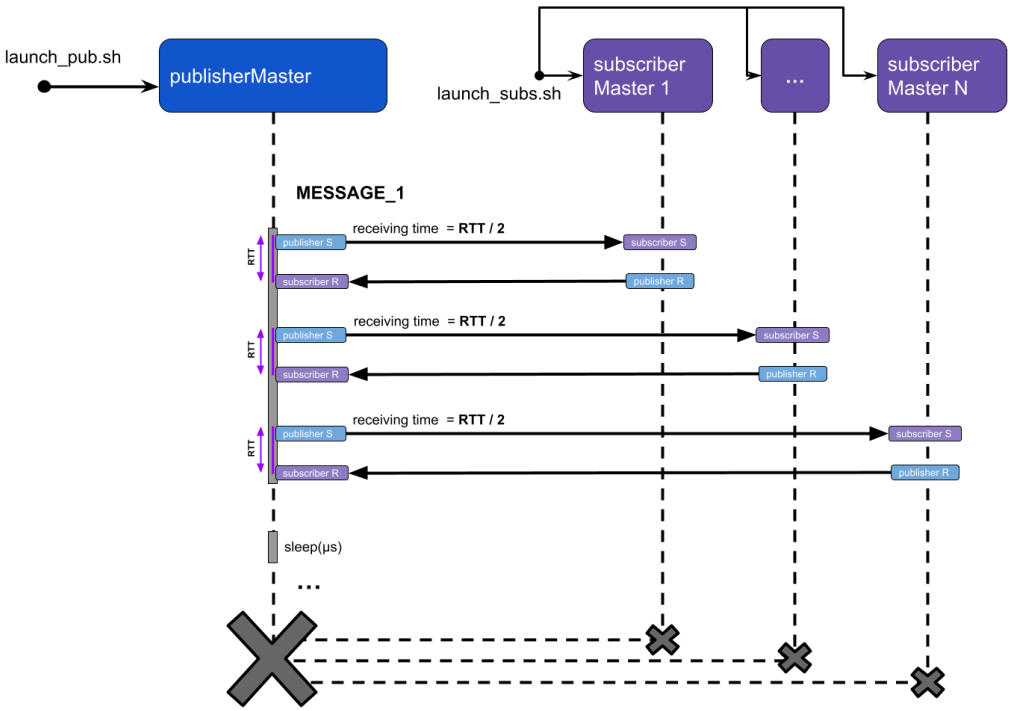


Figura 5.6. Schema UML RTT

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-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 [9] tutti i codici utilizzati durante lo svolgimento di questi test.

5.5.1 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 **??** su diversi nodi di un supercalcolatore.

5.5.2 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.

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

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.

5.5.3 Test-3

Nel test-3 si è voluto misurare il throughput e la bandwidth massima per ciascun protocollo. Per calcolarlo sono stati usati i seguenti dati:

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 e verrà stilato un modello di use-case utile alle finalità di Power Management.

6.1 Impatto del numero di sub in un dominio

Visto lo schema 5.2 risulta facile capire, che il numero di subscriber presenti in un dominio comporta un overhead di comunicazione che va ad influenzare sia i tempi, che i cicli, che le istruzioni impiegate nella singola *publish* su un topic come viene facilmente dimostrato nella figura6.2.

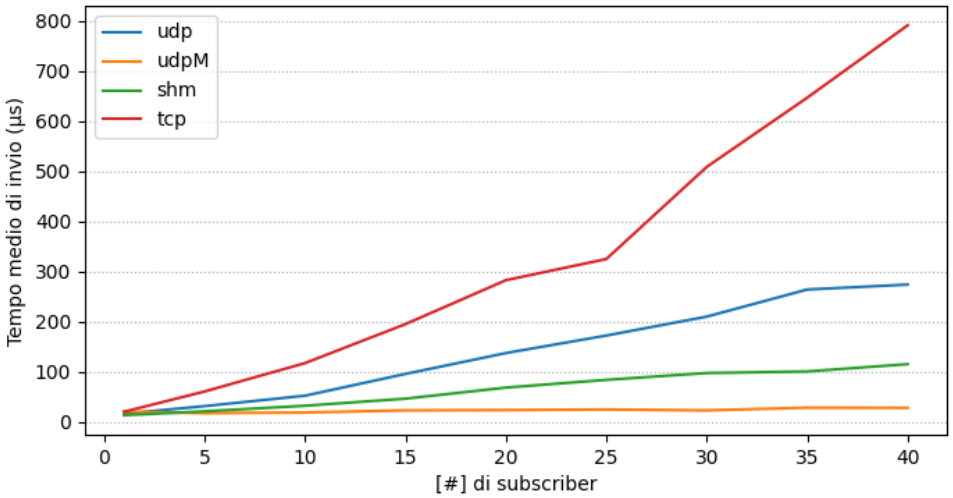


Figura 6.1. overhead sulla publish all'aumentare dei subscriber

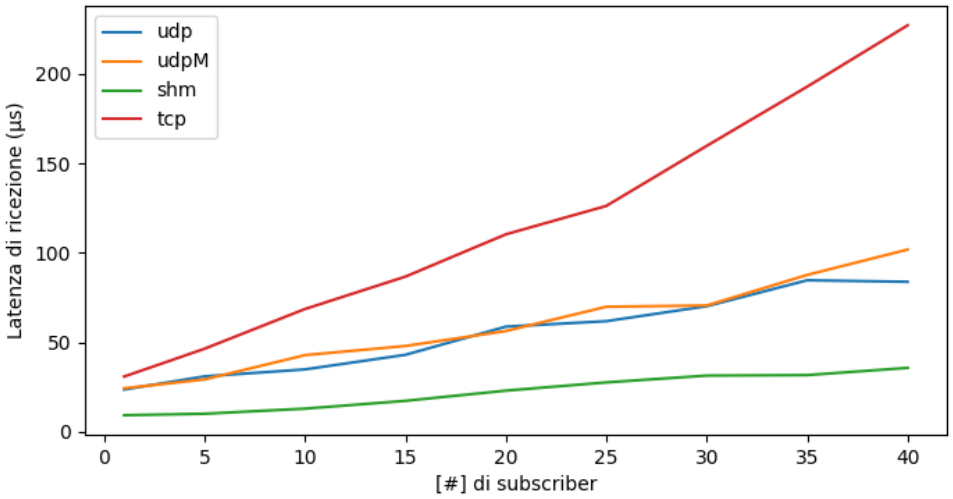


Figura 6.2. latenza di ricezione all'aumentare dei subscriber

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

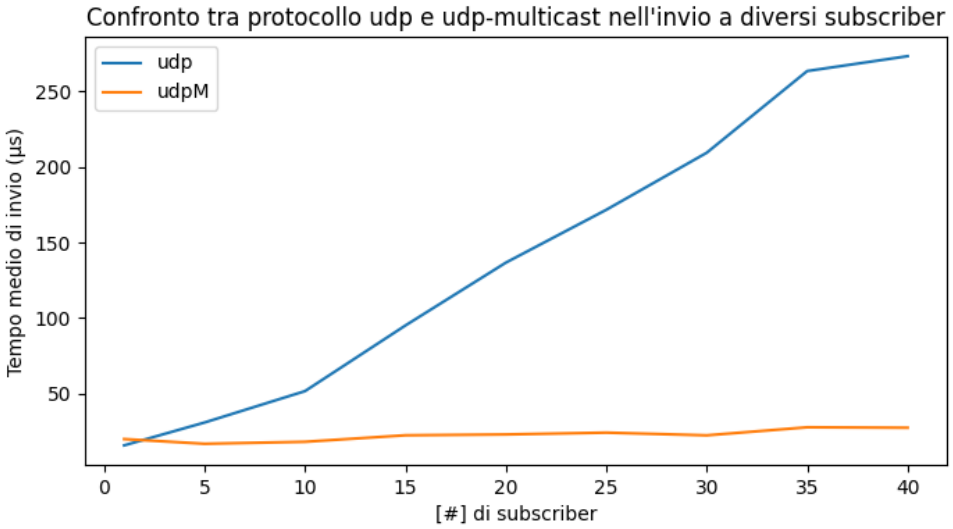


Figura 6.3

Da questo si può concludere che sia il publisher che subscriber risentono della presenza di molteplici ascoltatori su un topic. Questo problema è facilmente risolvibile lato publisher utilizzando protocolli che si basano su multicast.

6.2 Primo messaggio

E' stato notato in tutte le comunicazioni effettuate 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 5.6).

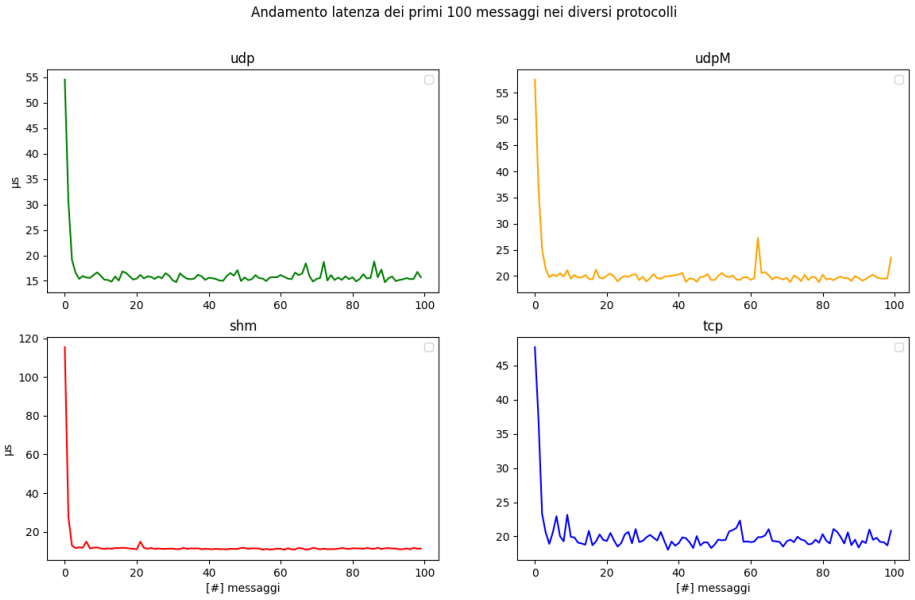


Figura 6.4

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

6.3 test-1

I risultati che sono stati trovati forniscono importanti informazioni,

[1] comunicazioni effettuati in localhost o in shared memory

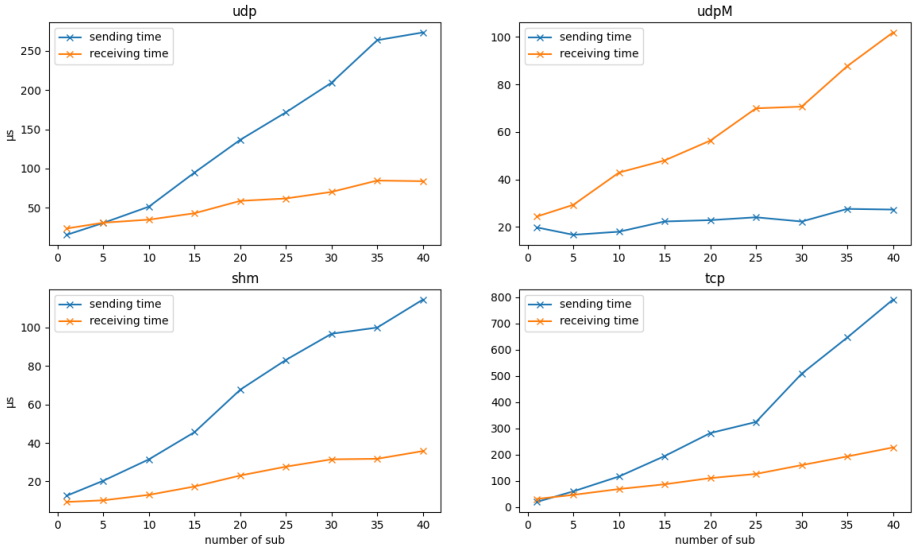


Figura 6.5. differenza tra solo publish e publish-subscribe per ogni protocollo

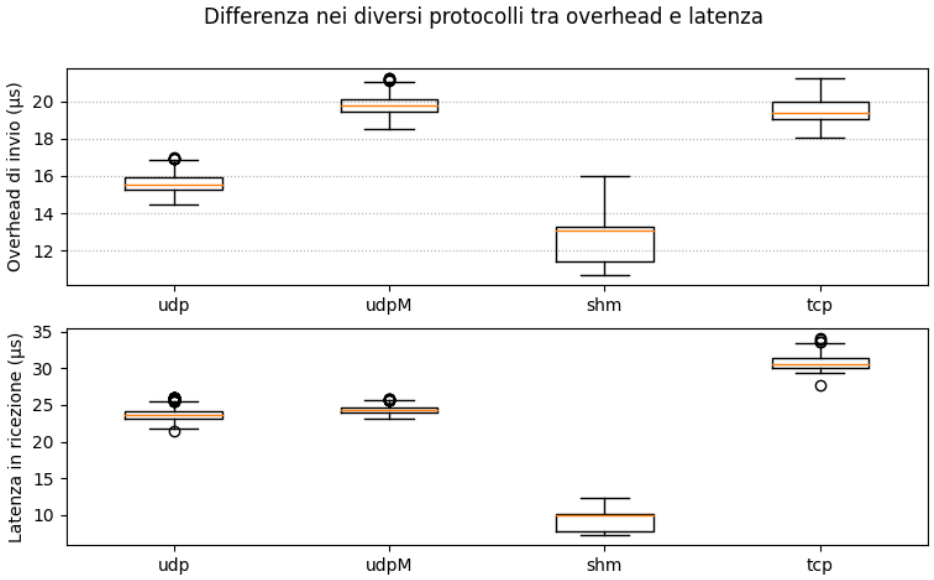


Figura 6.6. diagramma a scatola nei vari protocolli di comunicazione

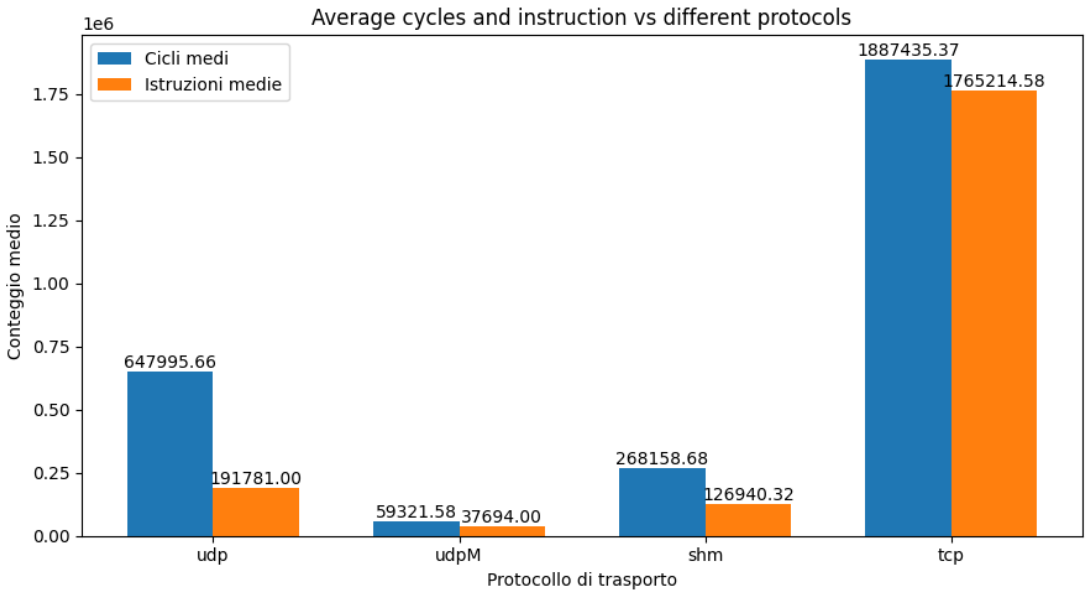


Figura 6.7. Conteggio cicli e istruzioni per ogni protocollo

6.4 test-2

Nei test effettuati con domini, topic e partizioni, non sono state notate differenze degne di nota in termini di performance (cicli e istruzioni) nell'usare uno strumento piuttosto che un altro.

6.5 test-3

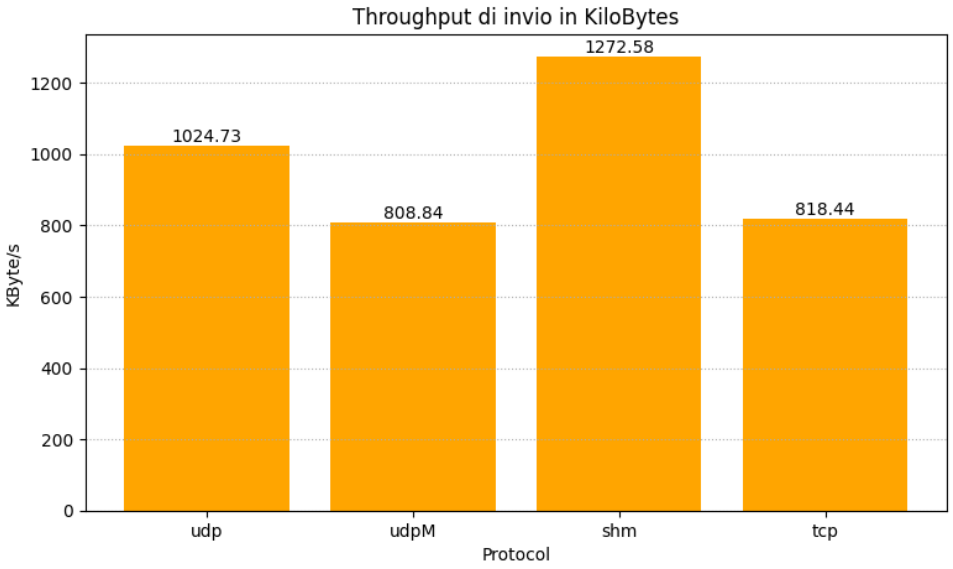


Figura 6.8

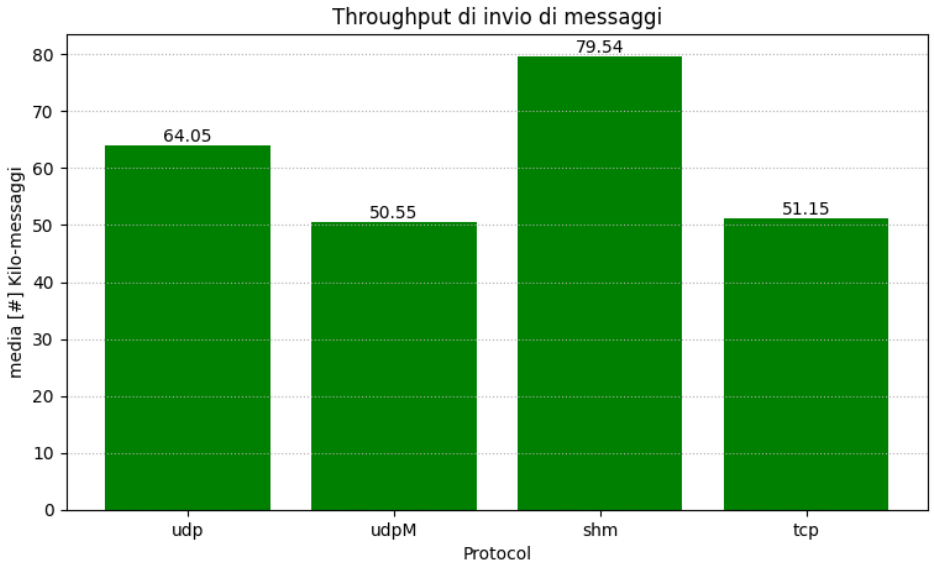


Figura 6.9

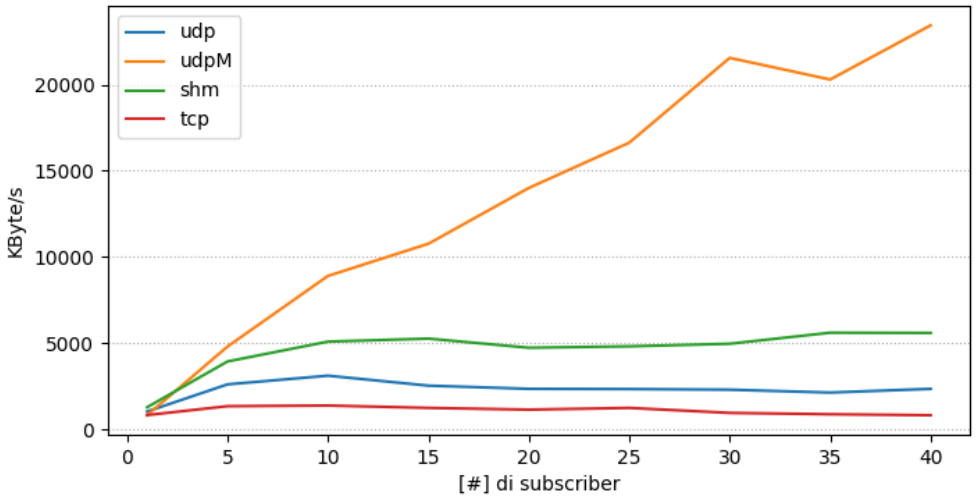


Figura 6.10

6.6 Modello

Componenti dummy

Nel corso di questa tesi con la collaborazione di alcuni membri del progetto REGALE, come Cineca[1] e BSC[1] 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 effettiva implementazione del middleware DDS all'interno di componenti già sviluppati nell'ambito del PM che vogliono essere introdotti nello stack.

seguente capitolo viene stilato uno scheletro dei componenti con i relativi topic usati al fine di dare una visione completa e aggiuntiva rispetto al modello precedentemente stilato

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

7.0.1 Job Manager

7.0.2 MQTT Bridge

Conclusioni

Nel corso di questi test è stato possibile dimostrare le potenzialità di questo strumento in un ambiente altamente prestante come i sistemi HPC. È stato dimostrato come un framework di comunicazione DDS può essere usato all'interno di un Power-Stack per la gestione di energia in sistemi HPC vincolati dalla potenza al fine di affrontare il problema della limitazione energetica.

Glossario

assembly 25

kernel 25

ntpd 25

35

overhead 36

Bash Bourne Angain Shell, linguaggio di scripting 23

Real-Time In tempo reale, con latenze molto basse 10

Bibliografia

[1] TODO. *TODO*. 2023. URL: https://wikipedia.it.

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

[3] 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.

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

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

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

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

[8] 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.

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