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

Indice

1 Introduzione **4**
1.1 Contributi 5

2 Power management **6**
2.0.1 High-Performance Computing 6
2.1 Stato dell'arte 7
2.1.1 Servizi in-band 7
2.1.2 Servizi out-of-band 8
2.2 Interfacce di alto livello 8
2.3 Modello di power stack 8
2.3.1 Workflow engine 9
2.3.2 Job schedulers 10
2.3.3 Resource Manager 10
2.3.4 System Manager 10
2.3.5 Job Manager 11
2.3.6 Node Manager 11
2.3.7 Monitor 11

3 REGALE **12**
3.1 Obbiettivi 12
3.2 Power Stack 12
3.3 Problematica 13

4 DDS & RTPS **14**
4.1 Implementazione usasta 14
4.2 DDS 14
4.3 RTPS 15
4.4 ROS 16

5 Casi di studio e valutazioni **17**
5.1 Struttura dei test 18
5.1.1 Shell 18
5.1.2 C++ 19
5.1.3 Ottenimento dei tempi 20
5.2 DataMiners 23
5.3 Test 23
5.3.1 Impatto del numero di sub in un dominio 23
5.3.2 Test-1 25
5.3.3 Test-2 26
5.3.4 Test-3 27
5.4 Risultati 28
5.5 Modello 28
5.6 Scheletro componenti 28
5.6.1 MQTT Bridge 29

6 Conclusioni **30**

Introduzione

Nel panorama scientifico e industriale contemporaneo, assistiamo ad una sempre più crescente domanda di capacità computazionali, alimentata anche dalla necessità di gestire, monitorare e analizzare ingenti quantità dati. Infatti ambiti come simulazioni complesse, meteorologia, calcolo in tempo reale e le nuove realtà di intelligenza artificiale sono rese possibili solo grazie a tecnologie il cui unico scopo è quello di risolvere problemi avanzati in tempi brevi. In particolare si parla di High Performance Computing (HPC), o calcolo ad elevate prestazioni, di quelle tecnologie che utilizzano cluster di processori e componenti hardware ad alte prestazioni in grado di processare dati multidimensionali in modo simultaneo. I sistemi di *High Performance Computing*, per poter sopperire a queste richieste, necessitano di diversi nodi di calcolo, ognuno dei quali è composto da molteplici processori come CPU, GPU o TPU e memorie ad alte prestazioni. Questi nodi solitamente sono collegati tra loro da reti ad alta velocità che permettono sia ai software di schedulazione di suddividere i workflow in diversi nodi, che ai diversi job di comunicare durante la loro esecuzione. Inoltre per poter funzionare al meglio questi cluster solitamente sono supportati da sistemi di raffreddamento in grado gestire le notevoli quantità di calore che prodotte dalle attività di calcolo.

La gestione energetica di sistemi HPC è diventata per questo motivo, una delle principali preoccupazioni, non solo a causa dei costi monetari, ma anche per la sostenibilità ambientale e per la progettazione di nuove generazioni[1] di supercomputer. Infatti perpendicolarmente all'aumento della potenza computazionale richiesta, le tecnologie associate allo sviluppo dei componenti primari dei processori, i componenti più energivori del sistema, si sono avvicinati ai loro limiti fisici. Questi ultimi hanno portato a delle difficoltà sempre più grandi nel ridimensionamento dei transistor, conseguito nella 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 questi sono venuti a mancare, il mantenimento e ancora di più lo sviluppo di nuove generazioni di sistemi è diventato un compito tutt'altro che banale, e con questi si sono resi necessari sempre più software in grado di automatizzarne la gestione. Infatti utilizzare efficientemente l'energia disponibile e ottimizzare le prestazioni delle applicazioni sotto un limite di potenza è diventata

una sfida, che ha richiesto soluzioni specifiche. Il concetto di Power Management è nato sotto questo contesto, definendo un modello software che ha il compito di gestire la potenza di sistemi di HPC. Per farlo sono stati definiti diversi attori ognuno con un compito specifico, e cercando di standardizzare le interazioni che questi componenti dovevano avere.

Questo Power-Stack ha però la necessità di avere una visione globale per la gestione energetica, al fine di permettere dove necessario di definire degli obbiettivi e limiti di prestazioni e consumi. Inoltre deve essere definita un'interfaccia standard per poter interagire con i controlli hardware e software di sistemi HPC di diversi produttori. 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 a granularità diverse 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 ai gestori dei sistemi di HPC, l'onere di farlo. L'obiettivo finale sarebbe infatti quello di fornire un middleware documentato e facilmente integrabile, nei vari strumenti ad oggi presenti, per collegarli tra loro utilizzando un approccio distribuito e sfruttando il potenziale del Data Distribution Service (DDS) nonché quello di Real-Time Publish-Subscribe (RTPS).

1.1 Contributi

I contributi di questa tesi sono stati lo studio e caratterizzazione di una specifica implementazione di DDS all'interno di sistemi HPC al fine di fornire una visione più completa della possibilità di integrare questo strumento come base delle comunicazioni di un middlware per i componenti del power management. Successivamente sono stati valutati dei modelli basati su questi risultati come modo d'uso. Infine sono stati creati per completare il quadro due di questi attori, mancanti nelle implementazioni attualmente prodotte, utilizzando la libreria REGALE.

Power management

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;

Nel contesto di questa ricerca, si farà riferimento a questa parola per abbracciare tutti e tre i concetti che essa può rappresentare, riflettendo così una visione olistica per la comprensione del problema.

2.0.1 High-Performance Computing

Il contesto nel quale viene definito solitamente un Power Management è un sistema di computazione ad alte prestazioni, detto anche sistema HPC. Questi ultimi sono macchine computazionali composte da decine di centinaia di nodi interconnessi tra di loro da reti a bassa latenza e alta banda. Tutti i nodi uniti insieme, visto l'elevato numero di processori e agli acceleratori che hanno montato al loro interno offrono capacità computazioni che nei giorni nostri hanno raggiunto ordini del ExaFlops (10^{18} operazioni di Floating Point per secondo). Visto l'aumento della densità energetica e alle magnifiche prestazioni raggiunte, la potenza necessaria per alimentare questi sistemi sta superando i 20MWatt. 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 questi sistemi. Infatti se non adeguati, le difficili condizioni in cui lavorano comportano grandi inefficienze in termini di energia, che si traducono anche in degradazioni di prestazioni computazionali. Considerando tutto, ai centri che ospitano queste macchine si devono fornire decine di MWatt di potenza per ogni exa-supercomputer che hanno installato. Ordini di grandezza di questo tipo non sono facilmente ottenibili e 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. Power Management può essere visto come un sistema composto sia da parti software che Hardware. L'insieme di questi componenti va a formare un power-stack in grado di gestire la potenza assorbita di macchine HPC.

2.1 Stato dell'arte

Per riuscire nel suo scopo, il Power-Stack deve gestire la potenza dei sistemi a diversi livelli di granularità come sistemi, singoli nodi, e singoli elementi all'interno dei nodi. Per poterlo fare, è necessario utilizzare componenti hardware accessibili principalmente tramite servizi in-band e out-of-band.

2.1.1 Servizi in-band

I servizi in-band accedono alle risorse hardware utilizzando i medesimi canali attraverso cui transitano altre forme di informazioni e comandi. In questa specifica situazione, si tratta di interfacce che richiedono al processori stessi le informazioni necessarie. Ognuno di essi è sviluppato su più livelli tra cui i cosiddetti *Governors* (o governatori), i *Driver* che servono a veicolare queste informazioni ed infine componenti Hardware presenti all'interno dei chip. Fondamentalmente i governors permettono monitorare e in alcuni casi di gestire alcune metriche dei processori. Tra i più famosi Governors troviamo CPUfreq[1] (Linux)e *Intel Power Governors*[1]. **CPUFreq** è un insieme di strumenti del kernel Linux che regola e monitora la frequenza dei processori [1] . Può effettuarlo sia in autonomia, in risposta al carico di sistema, sia adattarsi in risposta agli eventi ACPI oppure essere modificata manualmente dai programmi nello spazio utente. E' reso possibili tramite degli specifici driver (intel_pstate) che comunicano con i componenti hardware sul chip. Diversamente **Intel Power Governors** utilizza le interfacce proprietarie (RAPL) con cui monitora potenza ed energia a livello di sistema. Entrambi questi strumenti hanno necessità di componenti Hardware dedicati come i power knob (manopole per la gestione della potenza) e sensori che permettono di monitorare temperature e tensioni. Usare i servizi in-band sono molto flessibili e permetto di operare in

real-time ed in modo dinamico. Infatti interagendo con l' Hardware e passando tramite il sistema operativo, sono necessari semplici chiamate e comandi per controllare questi strumenti.

2.1.2 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. Questi oltre a non andare ad influenzare il consumo e l'utilizzo del sistema, permettono di monitorare i componenti anche quando ci sono errori ed eccezioni. Un componente tra i più famosi nell'ambito del power management è il Board Management Controller, solitamente un microcontrollore animato da sistemi embedded linux, e accessibile tramite ssh con 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 che di singoli processori.

2.2 Interfacce di alto livello

Nel corso degli anni con l'obbiettivo di ottimizzare e automatizzare l' *ottenimento* di queste informazioni sono stati sviluppati diversi software di più alto livello che utilizzano sia strumenti in band che out of band per gestire il power management. Tra questi rivediamo i punto famosi: *Variorum* (LLNL), *GEOPM* (Intel)[2], e *HDEEM* (Atos)[3]. Tutti questi vogliono fornire una soluzione ad un sottoinsieme di problemi gestione dell'energia piuttosto che ad un software globale di Power Management di sistemi di calcolo ad alte prestazioni.

2.3 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 ad HPC dall'esecuzione di un applicativo, fino alla gestione delle tensioni.

- Workflow engine
- System Manager
- Job Manager

- Resource Manager
- Node Manager
- Monitor

Di seguito viene riportato uno schema2.1 che mostra le interazioni tra i vari attori.

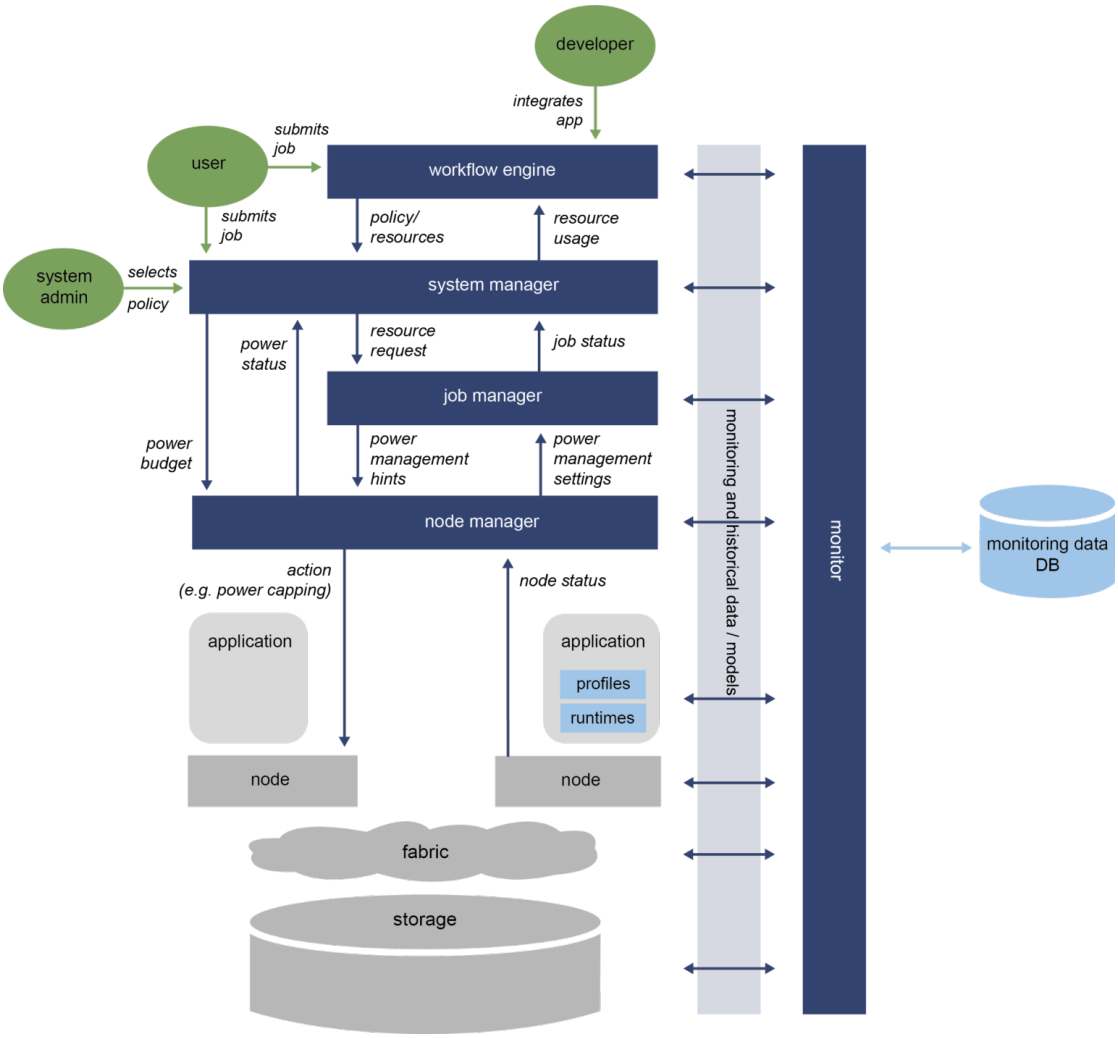


Figura 2.1. Modello di power stack

2.3.1 Workflow engine

Il workflow engine analizza le dipendenze e le richieste di risorse di ogni workflow e decide dinamicamente come dividerlo negli specifici jobs che verranno assegnati al system-manager.

2.3.2 Job schedulers

Il job scheduler ha il compito di assegnare e condividere le risorse computazionali e fisice del sistema HPC, ai vari utenti che lo utilizzano. In particolare la serie di compiti che si trova a svolgere è il seguente L'utente schedula i jobs da svolgere in una o più code, definite dallo scheduler. 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.

2.3.3 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.

2.3.4 System Manager

Il System Manager è un componente che riceve come input un insieme di jobs che devono essere schedulati all'interno del sistema, e in modo indicativo decide quando

schedulare ogni job, su quale nodo, e con quale power budget. Successivamente vengono monitorati i dati relativi a potenza ed energia, e controlla di conseguenza i budget di potenza, e la *user-fairness*

2.3.5 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.

2.3.6 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.

2.3.7 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.

REGALE

REGALE[4] è un progetto open source nato ad Aprile 2021 che ha come finalità quello di costruire e assemblare uno stack software scalabile per migliorare l'efficienza di sistemi Exascale[1] HPC. Questo progetto di tesi è nato in collaborazione con REGALE visto lo stesso interesse nell'ambito del Power Management in quanto si è proposta di implementare la quasi totalità degli attori proposti nello dello stato dell'arte per quanto riguarda il Power Management di sistemi di HPC.

3.1 Obbiettivi

Gli obbiettivi che si è posto di seguire il progetto, sono suddivisibili in 3 categorie:

- Effettivo utilizzo delle risorse messe, 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, durante lo sviluppo dei software;
- Facilità di implementazione ottenibile tramite la creazione di una infrastruttura flessibile, e che gestisca in automatico le risorse.

3.2 Power Stack

Molti dei componenti mostrati nel modello del Power Stack 2.1 sono stati sviluppati e resi disponibili nel progetto di REGALE. A questi mancano solo il *Workflow engine* ed un *Resource manager* specifico.

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

	Monitor				Node Manager	Job Manager	System Manager		Workflow Engine
	DB	Dashboard	Estimator	Sig Hndlr			RJMS	SPM	
SLURM							X		
OAR							X		
DCDB	X	X	/	/					
BEO	X	X			X			X	
BDPO						X			
EAR	X	X	?	?	X	X		X	
Melissa									X
RYAX									X
Examon	X	X	/	/					
COUNTDOWN						X			
PULPcontroller					X				
BeBiDa							X		
EPCM			/					/	

Figura 3.1. Copertura componenti REGALE

3.3 Problematica

Vista l'implementazione proposta, il problema con il quale si sta interfacciando il progetto può essere descritto con mancanza di interoperabilità. Ciò che manca in REGALE, è un layer di comunicazione che permetta ai vari attori di comunicare tra di loro. Infatti sono stati prima proposti diversi modelli di interazione tra i vari componenti a due a due, fino a quando si è deciso di provare a sviluppare un ulteriore middleware che standardizzasse le comunicazioni tra i vari attori, utilizzando un singolo strumento per tutti i componenti.

DDS & RTPS

DDS (Data Distribution Service)[5] e RTPS (Real-Time Publish-Subscribe)[6] 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).

4.1 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 [7]. 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.

4.2 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.

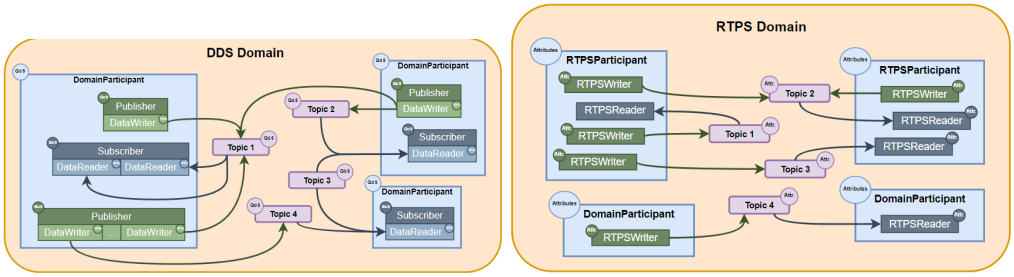
4.3 RTPS

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 4.1. Confronto tra architettura DDS e RTPS

4.4 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 4.2[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.

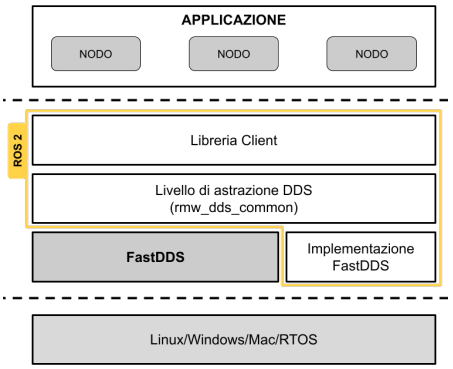


Figura 4.2. Ros Middleware per DDS

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

Casi di studio e valutazioni

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 e testare questa infrastruttura sono stati necessari veri sistemi di High-Performance Computing sui quali andare a provare quello che sequenzialmente è stato prodotto. Infatti per questo lavoro sono stati resi disponibili due supercomputer diversi nel tentativo di ottenere risultati altamente affidabili.

In questo capitolo verranno riportati i casi studio e i test effettuati sul framework. Tutti questi sono stati eseguiti su un sistema HPC Galielo-100 Cineca con le specifiche riportate nella tabella seguente

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	

5.1 Struttura dei test

I test effettuati in questa sezione sono composti da tre componenti ognuno di essi fondamentale all'ottenimento dei risultati nel modo più veloce e preciso possibile.

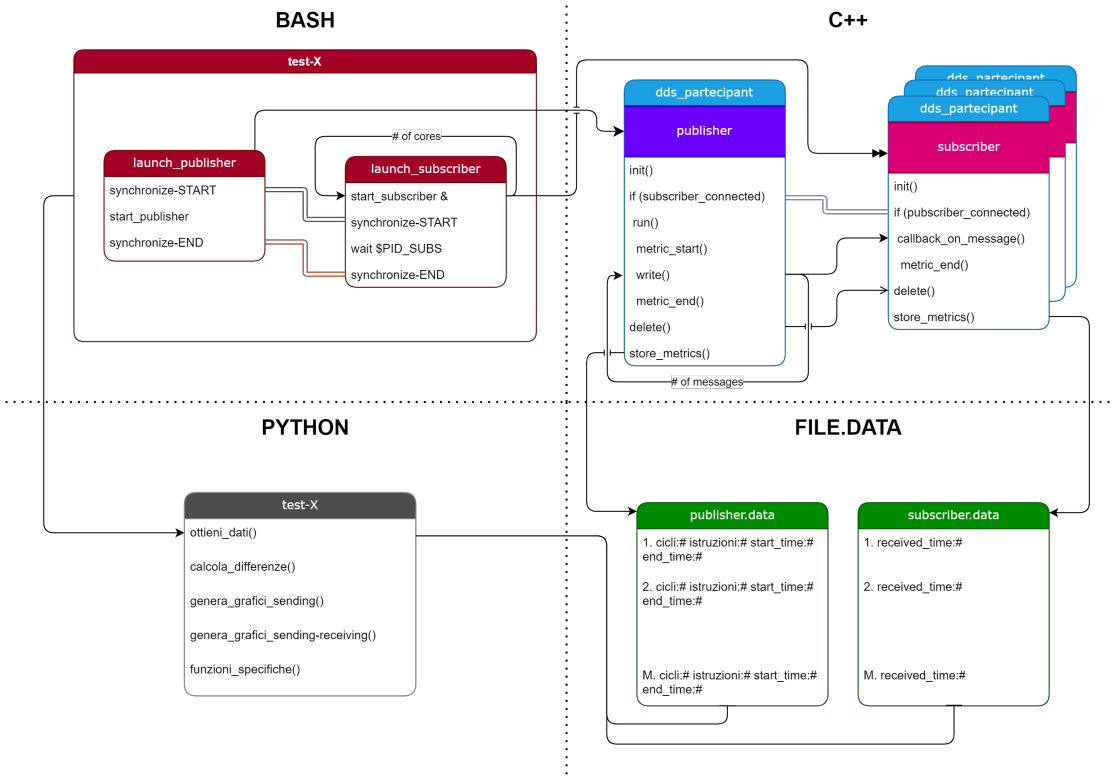


Figura 5.1. Struttura test

5.1.1 Shell

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* [4.4] 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 Calcolo TSC

5.1.4 Conteggio istruzioni

5.1.5 Ottenimento dei tempi

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 dovrebbe ottenere il tempo trascorso tra queste due.

Sincronizzazione orologi su nodi diversi

Per ottenere risultati attendibili sulla metrica del tempo è stato necessario sincronizzare i nodi utilizzati prima di poter far partire i test. Per farlo è stato usata una funzione *CLOCK_MONOTONIC* che rappresenta un *un orologio non impostabile a livello di sistema che rappresenta il tempo monotono da un punto non specificato nel passato. Su Linux, quel punto corrisponde al numero di secondi di esecuzione del sistema da quando è stato avviato. L'orologio CLOCK_MONOTONIC non è influenzato da salti discontinui nell'ora del sistema, ma è influenzato dalle regolazioni incrementali eseguite da NTP.* Il problema che si è presentato, è che avendo nodi diversi su cui far eseguire i test, per provare ad esempio nel modo più affidabile i protocolli di trasporto, è stato necessario implementare delle MPI_Barrier prima di diverse esecuzioni di *clock_gettime(CLOCK_MONOTONIC)*. Di seguito sono stati riportati i grafici del risultato ottenuto.

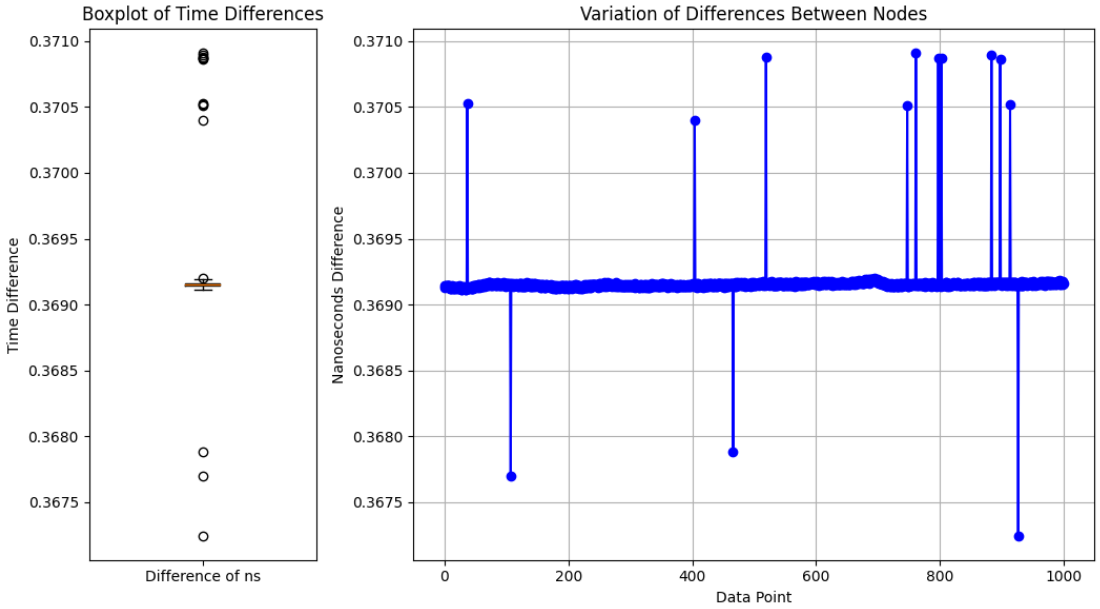


Figura 5.2. Scostamento del tempo su nodi diversi

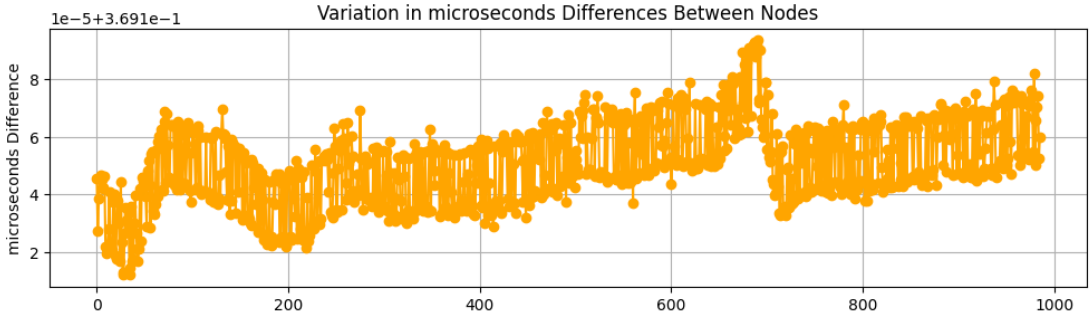


Figura 5.3. Scostamento senza outliers

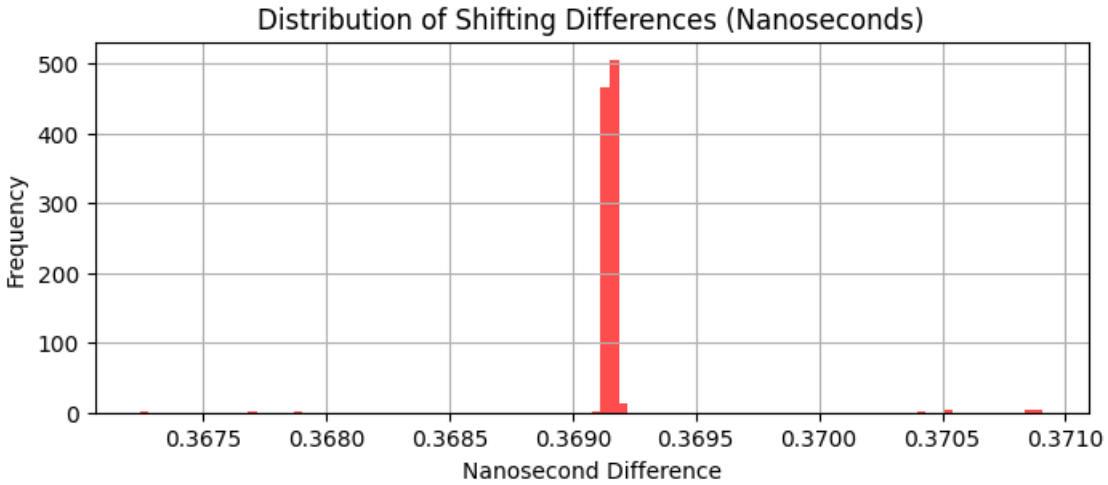


Figura 5.4. Distribuzione delle differenze

Come possibile vedere nella figura 5.2 nonostante le mpi_barrier, sono presenti degli scostamenti di tempo tra 2 nodi durante diversi test effettuati (in particolare 1000), e si è scelto di utilizzare il valore modale di questa differenza,sulla base del quale, si sono elaborati tutti i dati successivi.

Nonostante fosse idealmente il metodo più preciso per ottenere i tempi di send-receiving, essendo l'ordine di grandezza dell'errore dello stesso ordine di grandezza dei tempi di ricezione, si è preferito procedere con un approccio di ottenimento dei tempi basato sul **RTT**.

UML

Lo schema UML di funzionamento degli attori DDS è riassunto e schematizzato dalla figura 5.5

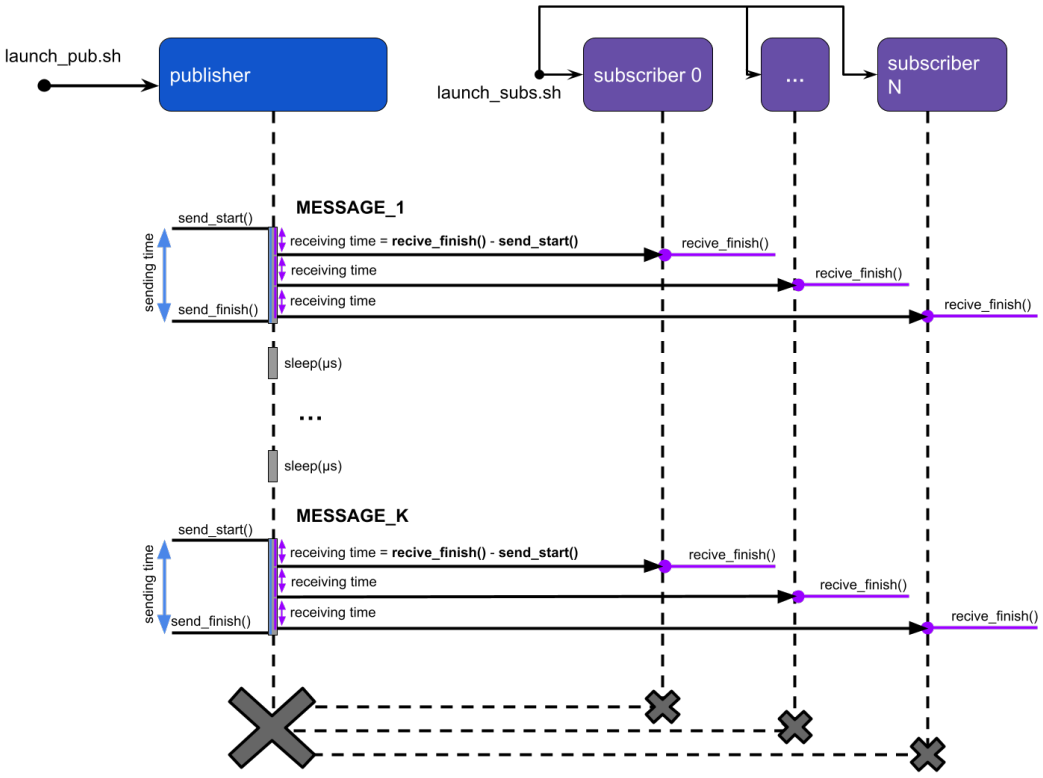


Figura 5.5. Schema UML

In particolare nel Publisher 4.2 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 4.2 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 generasse 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 Test

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

5.3.1 Impatto del numero di sub in un dominio

Visto lo schema 5.5 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 che viene facilmente dimostrato nella figura5.6.

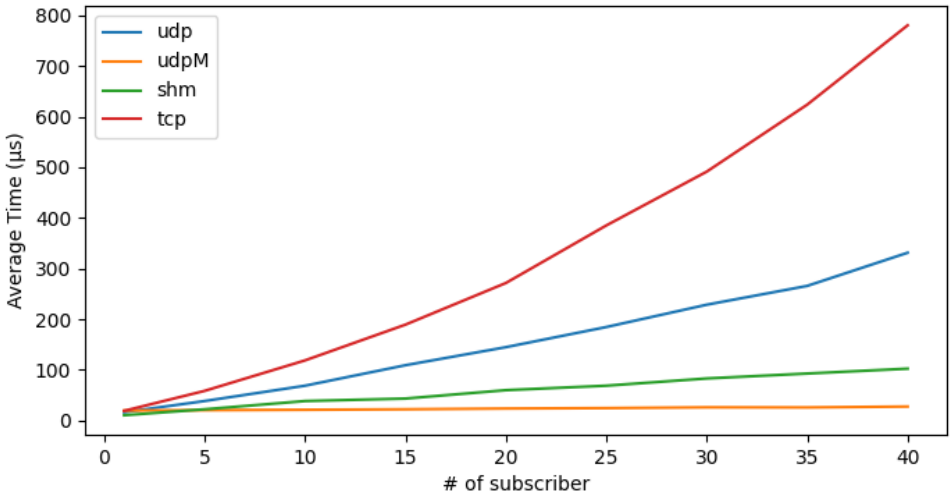


Figura 5.6. overhead sulla publish all'aumentare dei subscriber

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

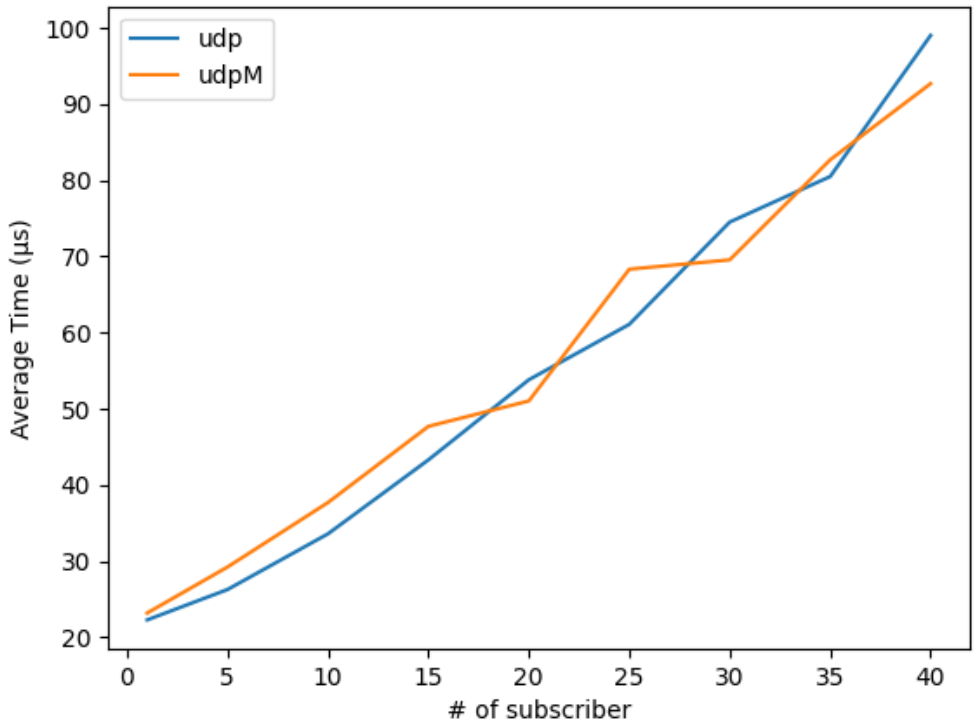


Figura 5.7

Questo può portare una singola publish a impiegare più cicli e più tempo della singola ricezione dei messaggi, come si vede nella figura5.8

In tutti i test successivi, ove non specificato diversamente sono stati usati 1 publisher e 48 subscriber su diversi nodi. Questo è stato fatto per provare la scalabilità, visto che nel testbed che è stato utilizzato, erano presenti 48 core (1 core per ogni subscriber).

5.3.2 Test-1

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

- udp: fornisce due versioni v4 e v6 e importa l'omonimo protocollo di trasporto
- tcp: fornisce due versioni v4 e v6 e importa l'omonimo protocollo di trasporto

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

Nel primo test si è valutata la differenza di queste implementazioni utilizzando la rete infiniband **??** su diversi nodi di un supercalcolatore. I risultati che sono stati trovati forniscono importanti informazioni,

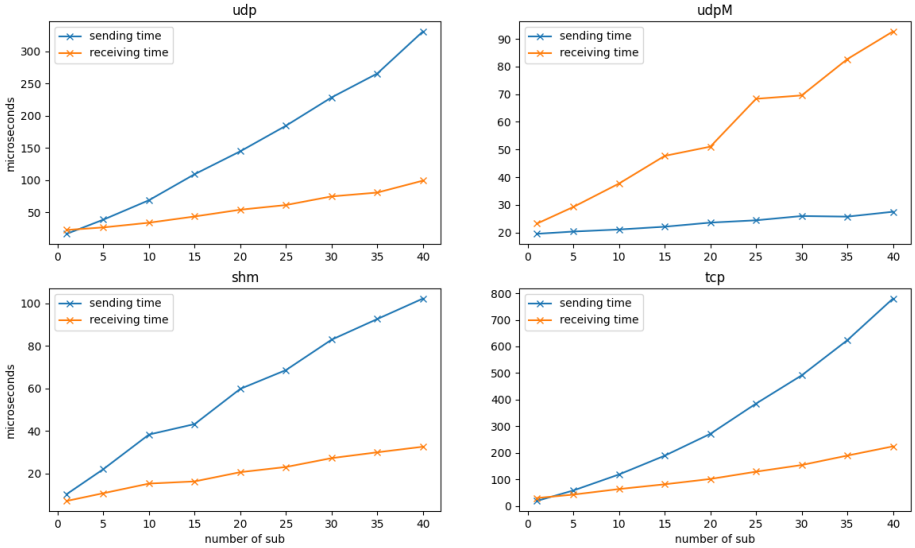


Figura 5.8. differenza tra solo publish e publish-subscribe per ogni protocollo

5.3.3 Test-2

Un concetto fondamentale nelle comunicazioni tra attori con gerarchie diverse, in sistemi con diverse centinaia di migliaia di entita, come cluster, nodi, processori, workflow, job (etc.), sono le possibilità di instradare, segmentare e rendere gerarchiche le comunicazioni. Come spiegatolo nel capitolo 4 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.

Comparazione strumenti

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.

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.

5.3.4 Test-3

Nel test-3 si è voluto vedere il throughput posssibile per ciascun protocollo, utilizzando un payload di X.

5.4 Risultati

Visti i diversi risultati si può dire che è meglio usare i topic per* le partizioni per * etc etc

5.5 Modello

5.6 Scheletro componenti

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

5.6.1 MQTT Bridge

Conclusioni

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

Real-Time In tempo reale, con latenze molto basse 5

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

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

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

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

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