Giuseppe Cardone - 0000276059

Progetto RE.VE.N.GE. DDS con Replicazione Sistema di Consegna

Marco Altini Stefano Bonetti Giuseppe Cardone

Abstract

Lo scopo del progetto RE.VE.N.GE. è stato quello di realizzare un middleware di supporto per la distribuzione di notizie su larga scala da parte di agenzie di stampa utilizzando l'implementazione di RTI dello standard DDS dell'OMG. Tale middleware disaccoppia nel tempo e nello spazio le interazioni tra fornitori e fruitori: non implica né la reciproca conoscenza né la compresenza durante l'invio delle informazioni.

Il middleware permette sia a fonti sia a fruitori di specificare la Quality of Service con cui il servizio viene fornito, personalizzato per ogni client.

Per garantire una elevata affidabilità per la consegna dei messaggi il sistema di consegna è replicato e prevede una serie di protocolli per il coordinamento dei peer, il monitoraggio degli errori e eventuale recovery del sistema.

Introduzione a RTI DDS

Il middleware utilizzato per lo sviluppo del progetto è RTI Data Distribution Service (RTI DDS), progettato per applicazioni real-time e che utilizza il modello di comunicazione publish-subscribe. RTI DDS è conforme a diversi standard: le API seguono lo standard DCPS delle specifiche DDS definite da OMG, i tipi di dati vengono definiti tramite l'OMG Interface Definition Language, mentre i pacchetti di dati usano il protocollo RTPS specificato dallo IEC.

RTI DDS implementa le API per il Data-Centric Publish-Subscribe (DCPS) dello standard DDS pubblicato da OMG. Il concetto fondamentale che ha portato alla progettazione delle API di DCPS è la distribuzione di dati tra applicazioni che comunicano fra di loro, al contrario di sistemi orientati agli oggetti dove il concetto fondamentale è l'interfaccia e la comunicazione è basata su invocazione di metodi su tali interfacce. I paradigmi data-centric e orientato agli oggetti sono complementari e si prestano a modellare in modo più naturale architetture diverse; ad esempio DDS è inadatto per il trasferimento di file, l'invocazione di metodi remoti, architetture basate su connessioni, mentre permette una progettazione molto semplificata in tutti quei sistemi in cui lo scambio di "pacchetti di dati" è l'unità fondamentale (come il sistema oggetto di questo progetto, scambio di informazioni borsistiche, posizioni di aereomobili e così via).

Il vantaggio principale di RTI DDS è che le interazioni sono molto disaccoppiate nello spazio e nel tempo: le applicazioni non hanno bisogno di informazioni sui partecipanti alla comunicazione, inclusa la loro esistenza o locazione. RTI DDS si occupa automaticamente di gestire gli aspetti della consegna dei messaggi senza intervento da parte dell'applicazione utilizzatrice, incluso:

- determinare chi deve ricevere i messaggi
- dove si trovano i riceventi
- cosa deve accadere se i messaggi non possono essere consegnati

Questo è reso possibile dai parametri della QoS che possono essere impostati dall'applicazione cliente, che configurano i

meccanismi di discovery e il comportamento da adottare per la consegna e ricezione dei messaggi.

Il paradigma per la comunicazione implementato da DDS è quello del publish/subscribe: un'applicazione indica la propria volontà di ricevere aggiornamenti di una certa "classe" tramite il subscribing, mentre l'invio avviene tramite il publishing. Entrambe le azioni sono compiute da ambo le parti senza sapere se vi siano publisher o subscriber: questo disaccoppia nello spazio le interazioni, evitando la reciproca conoscenza tra gli agenti. DDS disaccoppia le interazioni anche in senso temporale: il suo meccanismo di caching e routing permette la consegna dei dati anche se publisher e subscriber non sono presenti allo stesso tempo.

Il protocollo che si occupa fisicamente di effettuare il routing dei messaggi è lo standard OMG RTPS, che viene incapsulato in pacchetti UDP. DDS permette inoltre la comunicazione tramite shared memory. Altri supporti non sono inclusi ma possono essere implementati dagli sviluppatori.

Da un punto di vista delle proprietà lo scambio di messaggi inviati da DDS è asincrono (privo di risultato), asimmetrico (non c'è conoscenza tra i pari, indiretto (RTPS può portare a routing complessi). Le policy di QoS di DDS permettono di impostare se le comunicazioni debbano essere bloccanti o non bloccanti, bufferizzate o meno, reliable o unreliable. Nel caso di messaggi reliable gli ACK, se possibile, vengono inviati in piggyback su messaggi transitanti in senso opposto.

Alla base delle astrazioni del meccanismo di comunicazione di DDS vi sono i *topic*. Un topic è associato in modo univoco con il tipo di dati che vi vengono pubblicati (e possono esserci più topic associati allo stesso tipo di dati). Un pacchetto di dati pubblicato sul topic è detto *sample*. I componenti responsabili della lettura e scrittura dei dati sono rispettivamente i *DataReader* e *DataWriter*. Sullo stesso topic possono insistere più DataReader e DataWriter, anche appartenenti alla stessa applicazione. Tali oggetti appartengono a un *Subscriber* o a un *Publisher*, che si occupano dell'invio effettivo dei dati (processo detto "disseminazione").

Per garantire il disaccoppiamento nel tempo e la reliability DDS utilizza delle code non visibili all'applicazione finale. La reliability di DDS si riferisce solo ai dati che trovano spazio in tali code, sia in invio che in ricezione. Questo comporta che i dati non memorizzabili nei buffer vengano scartati senza alcun avviso. Si può aggirare il problema indicando a DDS di far crescere indefinitamente le code, ma questo implica che alla caduta irreversibile di un nodo le code di buffer delle fonti finirebbero per allocare tutta la memoria disponibile portando al crash dell'applicazione.

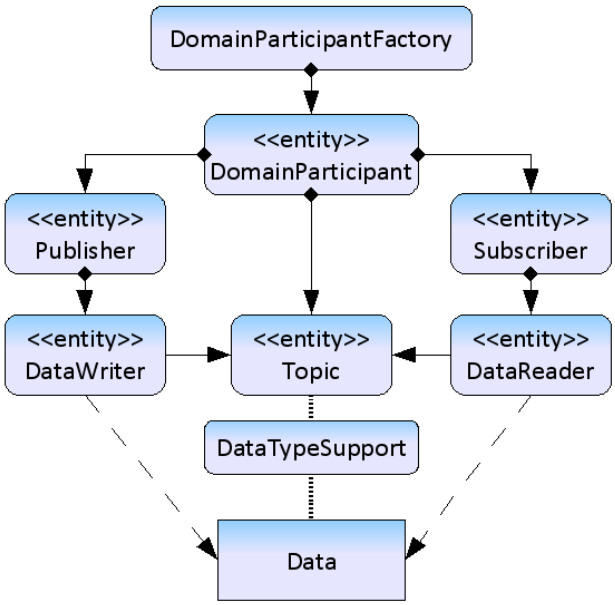


Immagine 1: diagramma UML delle classi fondamentali di RTI DDS. Tutte quelle marcate con «entity» possono specificare una propria QoS.

È importante notare come DDS metta in comunicazione un publisher con un subscriber solo se questi hanno dichiarato QoS compatibili: un fruitore che ha richiesto sample reliable non riceverà mai dati da un fornitore che ha dichiarato una QoS unreliable. Più in generale i DataReader richiedono una QoS e i DataWriter la

dichiarano: DDS metterà in comunicazione tra di loro solo le coppie per cui la QoS della sorgente è migliore o uguale, per tutti i parametri, rispetto a quella richiesta dal fruitore. Inoltre, a meno che non venga esplicitamente richiesto, DDS non mantiene l'ordine dei messaggi tra coppie che comunicano, e in ogni caso è incapace di mantenere un ordinamento globale tra tutti i sample in circolazione nel sistema. Questa importante caratteristica ha avuto un pesante impatto sulla progettazione dei protocolli coinvolti in RE.VE.N.GE.

DDS mette a disposizione delle API che permettono a un DataReader (o DataWriter) se è effettivamente in comunicazione con una controparte funzionante, e notifica eventuali variazioni del numero di entità con cui è connesso. Questo comportamento è ottenuto grazie all'invio di *heartbeat* tra i componenti coinvolti, la cui frequenza e tempo di validità può essere configurato opportunamente.

I fruitori possono ottenere facilmente un filtraggio di tipo type-based creando dei reader su uno specifico topic. Inoltre è possibile ottenere un filtro content based usando i *filtered topic,* che vengono configurati utilizzando un linguaggio simile a SQL.

Publisher e subscriber all'atto della creazione vengono assegnati a un *dominio*: entità appartenenti a domini diversi non possono comunicare tra di loro in alcun modo. Un dominio a sua volta può essere suddiviso in *partizioni*. Anche se da un punto di vista logico le partizioni non comunicano tra di loro, ciò non vieta a RTI DDS di utilizzare comunque nodi di partizioni differenti per un routing più efficiente.

Architettura generale del progetto RE.VE.N.GE.

A alto livello il sistema è diviso in tre blocchi: le fonti delle notizie (source), i fruitori delle notizie (sink) e il sistema che le gestisce. I componenti del sistema di smistamento sono chiamati peer.

Ogni componente è identificato univocamente da un ID di 128 bit, ricavato calcolando l'hash MD5 di alcuni dati casuali.

Quando il sistema riceve una notizia, appartenente a uno o più argomenti, si incarica di memorizzarla e consegnarla ai sink che si sono registrati per tale argomento. Anche nel caso in cui la relativa source cada, il sistema garantisce comunque la consegna. Il completo disaccoppiamento inoltre permette permette di garantire la massima eterogeneità di fonti e fruitori.

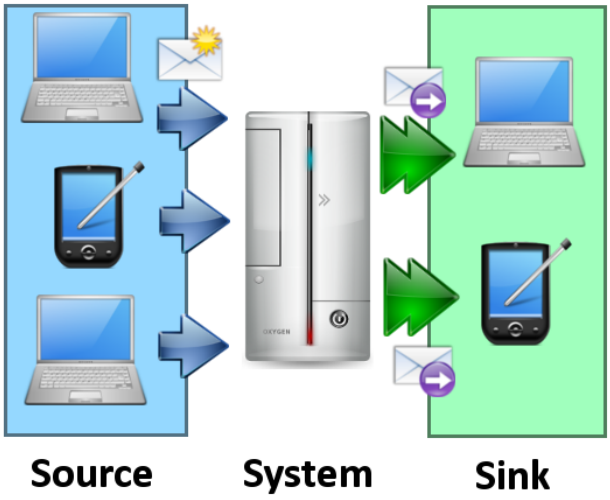


Immagine 2: architettura generale del sistema

Quando una nuova source desidera registrarsi presso il sistema dichiara la propria intenzione pubblicando un messaggio sul topic `SOURCEMNGTOPIC`, indicando il proprio identificativo, la QoS con cui intende fornire il servizio secondo specifiche (rate minimo e massimo di notizie per unità di tempo, consegna reliable/unreliable e così via). Tale registrazione avviene utilizzando una comunicazione reliable: è ragionevole infatti che anche se la fonte utilizza una connessione unreliable, la sua registrazione debba essere invece un evento certo per entrambe le parti. Inoltre a un source è associata una priorità che viene ereditata da tutte le notizie da esso inviate: il sistema farà in modo che le notizie inviate da source con priorità

maggiore attraversino più velocemente le code di inoltro interne di smistamento e, se il supporto di trasporto lo permette, assegna anche una QoS maggiore sulla rete.

Dopo la registrazione il source crea un DataWriter e si collega al dominio dei source su una partizione avente identificativo uguale al proprio ID: è lì infatti che il sistema creerà i DataReader in attesa di ricevere notizie.

Simmetricamente quando un sink vuole ricevere notizie, invia un messaggio di join sul topic `SOURCEMNGTOPIC`, indicando il proprio ID, la QoS desiderata, gli argomenti a cui intende sottoscriversi e se intende avere l'esclusiva su alcuni argomenti. Come nel caso precedente, anche questa registrazione è effettuata utilizzando un connessione reliable. Dopo la registrazione il sink crea un DataReader sul dominio dei sink nella partizione avente per identificativo uguale al proprio source, in attesa che il sistema fornisca le notizie attinenti agli argomenti richiesti. Si è deciso di effettuare il content filtering a monte, sul sistema, invece di utilizzare le API di DDS in locale sul sink per due motivi:

1. Filtrare a valle significherebbe sprecare della banda verso i sink, risorsa che è stata ritenuta più preziosa e da tutelare rispetto alle trascurabile carico di CPU necessario per filtrare le notizie nel sistema
2. Poiché il progetto prevede replicazione del sistema, sincronizzare delle informazioni non effettivamente utilizzate costituirebbe un overhead inutile

Quello che accade quindi è che i source si registrano con una certa QoS e inviano news relative a un certo argomento aventi una determinata deadline e priorità, il sistema ordina per priorità nelle code associate ai sink le notizie, facendo attenzione nel caso in cui alcuni sink abbiano chiesto l'esclusiva su un argomento, e le consegna (se è stato in grado di rispettare la deadline) ai vari sink. Per i dettagli su come venga gestita la QoS si faccia riferimento alla relazione di Altini.

Architettura del sistema di consegna

Le specifiche di progetto hanno imposto la replicazione del sistema di consegna per garantire tolleranza ai guasti. Si noti come in tutta la procedura appena descritta è invisibile ai fruitori del servizio che RE.VE.N.GE. è formato in realtà da più server che cooperano: il protocollo sviluppato è stato progettato in modo da far sì che il sistema sia trasparente alla replicazione.

RTI DDS fornisce delle API che permettono di creare in modo molto naturale una rete P2P. Si è scelto quindi di non snaturare il middleware, preferendo quindi un'architettura a copie attive indipendenti priva di punti di centralizzazione (frontend) a un più comune sistema master-replica.

Il sistema di smistamento è composto da un certo numero di nodi indipendenti (peer) che si conoscono tutti tra di loro. Questa decisione, aliena rispetto all'orientamento ai dati di DCPS, è dettata dal fatto che è necessario che i peer replichino lo stato dei loro pari, e devono quindi necessariamente conoscerne l'identità.

Quando un source si registra presso il sistema tutti i peer creano un DataReader per ricevere le sue news.

Quando si registra un un sink uno e un solo peer è responsabile dell'inoltro delle notizie verso di esso. Questa scelta è dovuta a più fattori:

- evitare che arrivino più copie della notizia al sink, che dovrebbe filtrarle a valle
- evitare di sprecare banda inviando più copie dello stesso dato
- permettere load sharing

DDS mette a disposizione un meccanismo che permette di individuare duplicati dello stesso sample, permettendo di avere più DataWriter che scrivono lo stesso dato ma lo spazio delle chiavi è piuttosto piccolo, e in ambienti in cui il numero di notizie inviate per unità di tempo è molto elevato potrebbe portare a collisioni indesiderate.

La possibilità di eseguire un load sharing statistico è una conseguenza del fatto che a ogni peer e a ogni sink è associato un ID univoco, usando una metodo simile a quella di altre tecniche per mantenere il carico bilanciato in reti P2P (ad esempio P-ring).

Come detto precedentemente i peer si conoscono tutti tra loro e ogni peer sa qual è l'ID dei suoi pari. Lo spazio degli identificativi dei sink va da 0x00…00 (per 16 byte) a 0xFF…FF. Tale intervallo è diviso in un numero di partizioni pari al numero di peer attivi in quel momento nel sistema. Vengono presi gli ID di tutti i peer e ordinati per valore crescente. A ogni peer viene assegnata la partizione nella posizione corrispondente alla posizione dell'ID del peer nella lista ordinata.

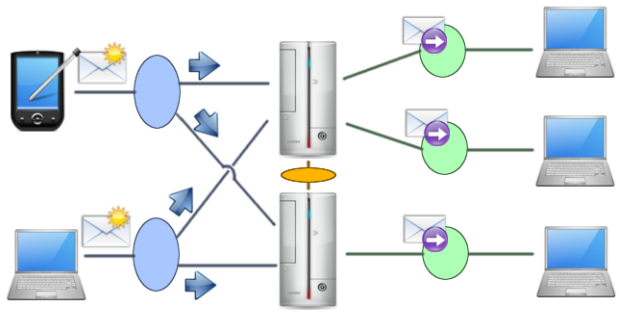


Immagine 3: ogni peer del sistema riceve dai topic le news di tutti i source, e le inoltra ai sink di cui è responsabile che si sono sottoscritti all'argomento della news inviata.

Poiché gli identificativi dei sink sono generati tramite hash MD5 di valori casuali si distribuiranno in modo grossomodo uniforme nello spazio delle chiavi possibili, per cui i peer tenderanno a gestire un numero equamente diviso di sink.

Gestione della replicazione

Quando un source si registra tutti i peer del sistema lo sanno, per cui questa parte dello stato di un peer non ha bisogno di essere replicata, essendo già intrinsecamente comune a tutti i componenti.

Ogni peer ha associato per ogni sink che serve una coda particolarmente complessa, detta SmartQueue (o PriorityFixedRateQueue), che si occupa di garantire parte della QoS, come ad esempio il rate massimo di notizie per unità di tempo o che vengano consegnate solo le notizie la cui deadline non è scaduta. Tale oggetto è quello che mantiene lo stato del servizio verso il sink e è quello che deve essere ripristinato in un altro peer nel caso in cui il servitore originale vada in crash.

Quando un sink si registra presso il sistema il peer che lo deve servire crea una SmartQueue per esso, mentre tutti gli altri una struttura dati leggermente più complessa di una normale coda, che consente accessi veloci ai suoi elementi tramite degli HashMap, che viene marcata come di proprietà di quel particolare sink.

Appena una notizia arriva e si determina che deve essere inoltrata a un certo sink, essa viene inserita con un push in tutte le code di tutti i peer. Quando un peer inoltra una notizia a un sink, notifica tutti i pari con un messaggio di update (linguaggio OMG IDL):

```
struct QueueUpdate{
    string<32> newsguid;
    string<32> peerguid;
    string<32> sourceguid;
    string<32> sinkguid;
};
```

che possono quindi rimuovere quella notizia dalle loro code. L'invio dell'aggiornamento è deciso esclusivamente dalla copia primaria (quella che ha appena servito il client) e segue una politica lazy: l'update è inviato dopo che la notizia è stata inoltrata, questo per migliorare l'availability del sistema. Ricordando che non abbiamo garanzie sull'ordine globale di arrivo dei

sample, è possibile che arrivi un update relativo a una notizia che il peer ancora non ha visto: in tal caso l'ordine di rimozione dalla coda viene rinviato, e un thread apposito di occupa di eliminare tali news appena giungono. Dopo un certo timeout, se la notizia non viene ancora ricevuta, la si considera persa e il comando di rimozione deferred viene scartato (`NewsQueuesManager.java`).

DDS prevede nativamente l'invio di heartbeat che servono per individuare il numero di DataReader e DataWriter presenti e accertarne il funzionamento. Tale meccanismo però non è in grado di determinare "l'identità" degli attori in gioco. Per questo motivo è stato necessario far sì che periodicamente i peer dichiarassero esplicitamente di essere ancora in funzione (`ImfineThrower.java`).

Quando l'heartbeat di un peer non perviene per più di un certo intervallo di tempo, questi viene dichiarato irraggiungibile dai restanti che provvedono a ripartire nuovamente lo spazio degli ID dei sink, e i peer il cui range di responsabilità si sovrappone a quello prima gestito dal pari non più raggiungibile prendono in carico i sink, convertendo le code di copia in SmartQueue (metodo `removePeer` in `PeerList.java`). Bisogna notare che in questo modo il carico aggiuntivo viene assunto da al più due peer, per cui non si garantisce più un carico bilanciato su tutto il sistema: i sink assegnati a un peer saranno sempre gestiti da questo, indipendentemente dal fatto che a causa di modifiche nel numero di nodi del sistema questo abbia un ID che non è più responsabilità del peer stesso. In altre parole per evitare ulteriori complicazioni al protocollo si è deciso di non implementare la possibilità per i sink di cambiare al volo il nodo che gli fornisce il servizio.

Il tempo necessario per individuare il crash di un peer può essere, a seconda del deployment, non trascurabile. In questo intervallo di tempo alcuni sink rimangono scoperti e il tempo di recovery potrebbe essere tale per cui alcune notizie potrebbero non essere consegnate prima della loro deadline: il sistema garantisce *liveness* ma non *safety*.

I test effettuati durante l'elaborazione del progetto hanno tutti previsto i nodi in una rete locale da 10Mbps con tempi di RTT per il ping di circa 0.5 ms. Gli heartbeat quindi in questo caso non ponevano problemi né per il tempo necessario per arrivare a destinazione, né per la banda occupata. Con test empirici si è deciso di inviare un heartbeat ogni 2000 ms e di considerare non più raggiungibile un peer dopo che non sono stati ricevuti per il triplo di questo tempo, 6000 ms. In un deployment realistico sarebbe probabilmente ragionevole aumentare quest'ultimo tempo in modo proporzionale alla difficoltà di connessione tra i peer, per evitare che banali problemi di rete che si risolvono in pochi secondi siano trattati come fallimento di un elemento del sistema. Una opportuna scelta dei tempi permette di reagire in tempi ragionevoli a errori di rete di durata consistente (trascurando quelli transitori), i casi di *send & receive omission* (ad esempio causati dai buffer di DDS pieni) e crash dei pari.

Il sistema non prevede alcuna forma di autenticazione, gestione della riservatezza o simili, e è quindi completamente vulnerabile a *failure bizantine* o comportamenti maliziosi.

Decidere che qualunque sistema possa sostituire qualunque altro in qualsiasi momento è una scelta progettuale forte, dato che per continuare a garantire il servizio è necessario essere certi che ogni singolo elemento del sistema di smistamento abbia una quantità di memoria sufficiente a mantenere lo stato di tutto il sistema, e sicuramente ci sono valide alternative. Ad esempio assumendo un'ipotesi di *single failure* è sufficiente che ogni peer abbia una singola copia attiva che ne duplichi lo stato: questo permetterebbe ad esempio di far scalare il sistema aggiungendo nuovi nodi, in un modo che non sarebbe possibile utilizzando un'architettura master-replica. La scelta seguita

nel nostro progetto è stata guidata da diversi fattori:

- banalmente ci sarebbero problemi se il numero di peer fosse dispari, anche se questo potrebbe essere risolto definendo un comportamento standard in questo caso (ad esempio il peer con identificativo più basso potrebbe comportarsi da copia per due peer invece che per uno solo), in ogni caso bisognerebbe complicare i protocolli con cui gli elementi del sistema si sincronizzano
- finché si rimane in uno scenario per cui il progetto potrebbe essere affrontato con un'architettura di tipo master-replica, è necessario supporre che anche un singolo nodo dovrebbe essere in grado di sostenere tutto il traffico del sistema, per cui le richieste di memoria per un singolo peer non sono irragionevoli
- la comunicazione tra coppie di peer transiterebbe in ogni caso sullo stesso topic comune per tutti (dato che non è possibile creare topic al volo), per cui le informazioni per la replicazione completa su ogni peer sono comunque disponibili senza costi aggiuntivi di rete. Si potrebbe ovviare al problema creando una partizione sullo stesso topic per ogni coppia, ma sarebbe comunque una complicazione non trascurabile al protocollo che non necessariamente porterebbe a dei vantaggi.

Inoltre i test hanno mostrato come il consumo di memoria per un peer siano molto bassi e di sicuro non il bottleneck del sistema: la gestione di 10 sink e di 10 source, ognuno dei quali invia 100 notizie al secondo richiede circa 4MiB di memoria heap. Questo è dovuto al fatto che effettivamente le code associate ai sink sono quasi sempre completamente vuote.

C'è anche da notare che è possibile isolare completamente un cluster di peer da un altro: tutte le comunicazioni di sistema infatti avvengono all'interno di un dominio apposito, e assegnando a peer diversi domini di sistema diversi si isolerebbero completamente tra loro, e potrebbero quindi gestire in modo completamente indipendente diversi sink. Questa scelta comporterebbe piccole modifiche al protocollo con cui un sink si registra presso il sistema e causerebbe in parte la perdita della trasparenza alla replicazione, dovendo i fruitori ricordare l'identificativo del cluster che utilizzano.

Nonostante la completa replicazione, è stato comunque necessario fare un'assunzione di single *point of failure* durante la progettazione del protocollo: ad esempio si considera che se un nodo non è a conoscenza di un source in conseguenza di un *send & receive omission*, gli altri peer siano in grado di riparare l'errore tramite il protocollo di whois.

Protocollo di gestione dei peer

Si è deciso di rendere la rete dei peer del sistema completamente dinamica, per cui è possibile aggiungere e rimuovere nodi durante il funzionamento del sistema senza causare malfunzionamenti.

Tutti i messaggi sono scambiati su un topic globale utilizzando questa struttura per i sample:

```
struct SystemMsg {
    string<32> senderguid;
    string<32> receiverguid;
    string<16> msg;
};
```

Una costante particolare al posto di `receiverguid` indica quali messaggi sono destinati a tutti i peer. In questo modo ognuno di essi può filtrare a valle i messaggi e considerare solo quelli destinati a se stesso o a tutto il sistema. Si potrebbe modificare il protocollo in modo che ogni vi sia un topic privato per qualunque coppia di peer, ma ciò complicherebbe molto l'implementazione senza fornire vantaggi sostanziali, dato che i messaggi

uno a uno sono solo una piccola percentuale di tutti i messaggi inviati.

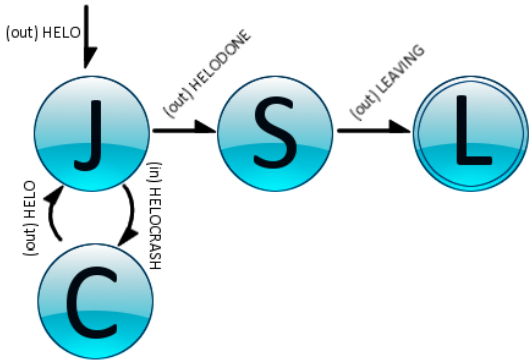


Immagine 4: gli stati di un peer e i messaggi che vengono emessi durante la transizione (out) o ne causano la transizione (in)

Ogni nodo è una automa con 4 stati: `joining`, `crashed`, `stable` e `leaving`.

Quando il peer cerca di unirsi al sistema è in stato `joining`. L'operazione consta di diversi passi:

1. Invio del messaggio `HELO` a tutti i peer
2. Si raccolgono le risposte `HELOREPLY` inviate dai peer già in funzione. Attesa di un timeout costante `JOINTIMEOUT`, che nei nostri test è stato impostato a 2000 ms. Se ci sono state risposte vai al punto 3, altrimenti termina.
3. Invia un messaggio `LISTSOURCE` al peer con identificativo più basso, questi risponde inviando l'elenco dei source tramite una serie di sample SourceElement (riga 40 di `News.idl`) contenenti gli ID dei source. Utilizzando il meccanismo di whois descritto nella relazione di Bonetti, il peer è in grado di recuperare tutti i dati sulla QoS da associare a quel source. L'ultimo sample dell'elenco ha il campo `lastElement` a `true`: in questo modo il nodo sa di aver terminato la ricezione dell'elenco dei source. È possibile utilizzare questo metodo poiché la QoS di DDS associata ai DataWriter e DataReader coinvolti è reliable e mantiene l'ordinamento dei sample.
4. Invia un messaggio `LISTSINK` a ogni peer che ha precedentemente risposto con un `HELOREPLY`, il quale replica inviando l'elenco delle news dei sink di cui è responsabili e il loro identificativo (riga 25 di `News.idl`): il nodo può quindi replicare il loro stato ricostruendo le code, e può correttamente associare a ogni peer i sink a cui è associato. Anche per ottenere le informazioni sui sink si usa il whois.
5. Viene inviato a tutti i pari il messaggio `HELODONE`: il nodo passa nello stato `stable` e lo spazio degli ID possibili dei sink viene ripartizionato da ogni nodo, tenendo conto del nuovo partecipante.

Se durante il protocollo di join il peer riceve un messaggio di `HELO` emesso da un altro nodo gli invia un messaggio `HELOCRASH`, che fa abortire la procedura del secondo pari. Quando un nodo riceve tale messaggio scarta tutte le informazioni ricevuto fino a quel momento e passa nello stato `CRASHED` per un certo periodo di tempo (algoritmo di back-off). Tale accorgimento è necessario per evitare un'eccessiva complicazione del protocollo permettendo il join di più peer contemporaneamente. Di fatti in questo caso il nodo in stato ancora non consistente dovrebbe poter notificare la propria presenza specificando di non essere ancora pronto per il funzionamento, i sample che descrivono lo stato delle code dovrebbero contenere destinatario e mittente per evitare collisioni e così via. In breve, considerando che l'aggiunta di un peer al sistema dovrebbe essere un evento raro, si è preferito mantenere il protocollo di join più semplice aggiungendo il vincolo che i peer possono unirsi al sistema solo uno alla volta.

Il tempo che viene trascorso nello stato di `CRASHED` è pari a $k*i+r*i$, dove k è un valore

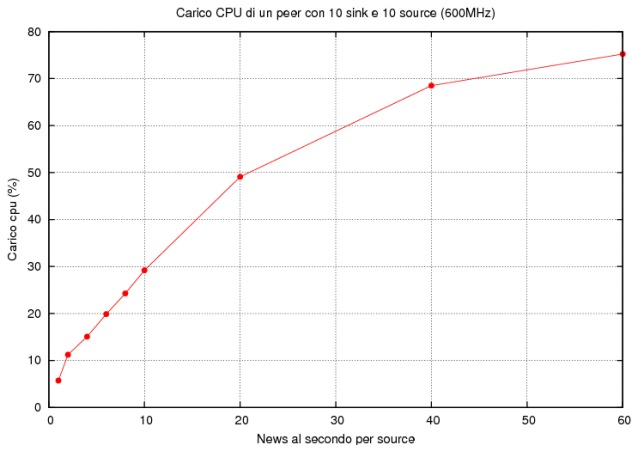
costante, impostato a 3000ms nelle nostre prove, *r* è un valore casuale tra 200 e 2000 ms e *i* è il numero di volte che si è verificata una colisione. La componente *k*i* è dovuta a due osservazioni:

- l'operazione di join, anche se fallisce, è costosa in termini di occupazione di banda, e la sua durata è presumibilmente trascurabile rispetto al tempo per cui il sistema sarà in funzione; per questo motivo è nel nostro interesse rinviare anche in modo consistente il join purché abbia successo.
- Se si verifica una collisione vuol dire che o ci sono molti peer che stanno cercando di unirsi al sistema o che la procedura di join richiede molto tempo a causa di un trasferimento di notizie consistente. In entrambi i casi è preferibile rinviare il momento in cui si effettuerà il join.

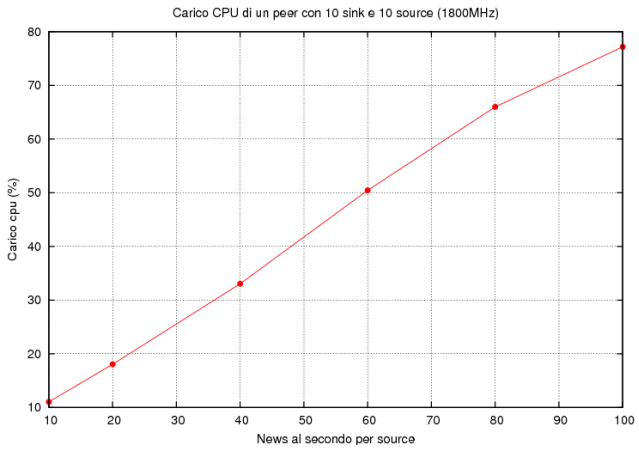
La componente *r*i* è stata introdotta per evitare il caso in cui due peer tentino di entrare a far parte del sistema in istanti molto prossimi: in tal caso è possibile che entrambi inviino il messaggio di join, entrambi ricevano il messaggio dell'altro e entrambi inviino un messaggio di crash. In tal caso entrambi seguiranno l'algoritmo di back-off, e al tentativo successivo hanno un'altra probabilità di collidere ancora. Questo comportamento è dovuto ai naturali ritardi di rete e al fatto che DDS garantisce, su richiesta, un ordinamento FIFO, ma non garantisce in alcun modo un ordinamento atomico. L'aggiunta di questa componente piccola casuale con l'aumentare delle collisioni compensa i ritardi di rete, fino a lasciar vincere uno dei peer.

Collaudo e test

Per controllare il carico sulla CPU indotto da un nodo del sistema, è stato avviato un peer su un computer con sistema operativo Linux e processore Intel Pentium M 1.8 Ghz, il cui clock è stato bloccato alla frequenza di 600MHz, per evitare che il gestore del risparmio energetico causasse misure inaccurate. Al peer sono stati collegati 10 source e 10 sink.



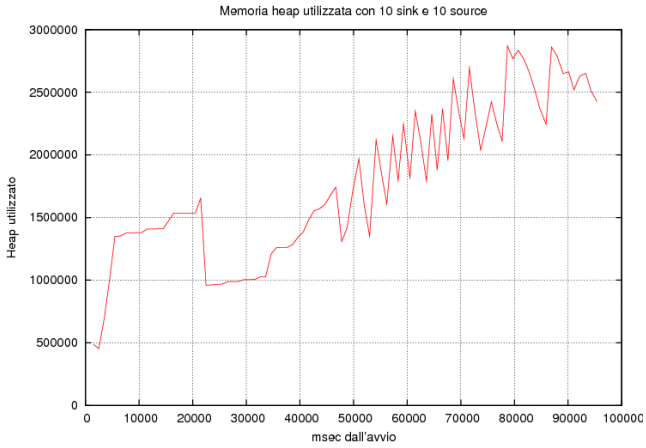
Ogni source generava il numero di notizie al secondo che si può leggere sull'asse delle ascisse. Un peer è quindi in grado di gestire in ingresso 100 notizie in un secondo e consegnarne 1000 (poiché dovevano essere inoltrate a tutti i sink) con un carico sulla cpu del 30%. Superate le 60 notizie al secondo per source il carico è tale per cui il sistema non è più in grado di soddisfare la QoS e ne conseguono pesanti malfunzionamenti. Il test è stato quindi ripetuto sulla stessa piattaforma imponendo un clock di 1800MHz.



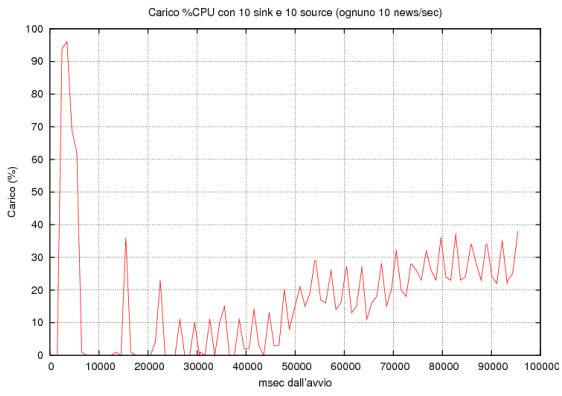
Il sistema si rivela in grado di ricevere al più 1000 notizie al secondo e inoltrarne 10.000 soddisfacendo la QoS. Superato questo limite non è più in grado di garantire il servizio.

Per quanto riguarda l'utilizzo di memoria si è

rivelato sufficientemente contenuto.



L'utilizzo di pool di thread per l'esecuzione e di strutture dati efficienti contiene il fabbisogno di memoria. In figura si può osservare come cresce l'utilizzo di memoria heap di un nodo mano a mano che si registrano 10 sink e 10 source, fino a arrivare al valore di regime di circa 2.5 MiB. L'utilizzo di CPU dello stesso test è il seguente:

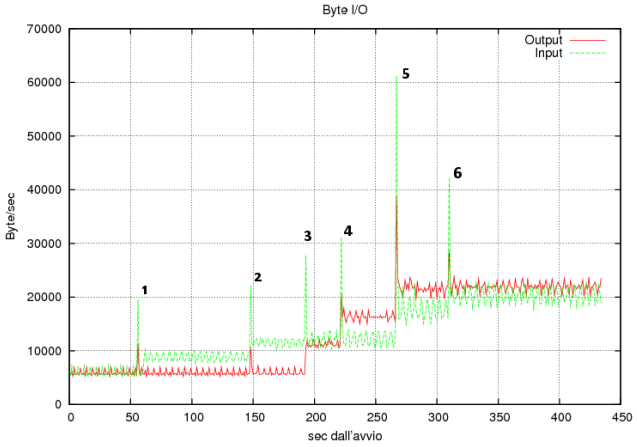


I picchi all'avvio sono dovuti all'allocazione delle strutture dati iniziali, all'avvio di DDS e all'avvio dei vari thread che si occupano del funzionamento del nodo.

Non è stato possibile eseguire test di carico della CPU al crescere del numero di client del sistema poiché la versione di RTI DDS dataci in licenza pone un limite al numero di DataWriter e DataReader presenti su un unico sistema operativo, per cui è stato impossibile simulare l'utilizzo di RE.VE.N.GE. da parte di molti client e misurare lo sforzo computazionale conseguente. Una prima stima eseguita con un piccolo numero di source indica che il costo di due source con lo stesso rate di emissione di news è grossomodo pari a quello di un unico source che abbia un rate di notizie doppio. A tale valore va aggiunto un piccolo overhead indotto dalle strutture dati aggiuntive necessarie, purtroppo non misurabile a causa delle limitazioni sopra esposte.

I picchi periodici visibili in figura sono dovuti all'overhead introdotto dal profiler utilizzato.

Per quanto riguarda l'utilizzo della rete è difficile distinguere esattamente la porzione di banda occupata da ogni singola attività (invio degli heartbeat, inoltro delle notizie, inoltro degli update e così via), poiché RTI DDS non mette a disposizione alcuna facility per effettuare tali misurazioni. È stato possibile tuttavia utilizzare uno sniffer di rete per misurare la banda in ingresso e in uscita. Il grafico ottenuto è il seguente:



La situazione iniziale è quella di due peer, che indicheremo con #1 e #2. Il peer monitorato è #1. L'invio e la ricezione degli heartbeat occupa circa 6 kbps in entrambe le direzioni. Si possono notare dei vistosi picchi in corrispondenza di diverse azioni compiute durante i test:

1. attivato un source costante con un rate di 10 news/sec. Vengono occupati circa altri 3 kbps in input, mentre la banda in output rimane invariata

2. attivato un sink gestito dal peer #2. Vengono consumati circa altri 3 kbps in input a causa degli update ricevuti. Le news inviate sono artificialmente piccole e di dimensione paragonabile agli update, se avessero dimensioni maggiore, come realisticamente accadrebbe, sarebbe un risparmio di banda

3. viene attivato un sink, gestito dalla copia #1. La banda in uscita passa da circa 6 kbps a circa 12 kbps: sono necessari quindi altri 6 kbps, che sono divisi in parti uguali per l'inoltro delle notizie e per l'aggiornamento della copia #1

4. viene attivato un altro sink che viene gestito dalla copia #1. Le conseguenze sono le stesse del punto 3

5. Viene attivato un ulteriore peer #3. Vengono occupati altri 6 kbps per la ricezione degli heartbeat in input, mentre si aggiungono 5 kbps in output per l'invio degli aggiornamenti e dell'heartbeat.

6. Viene aggiunto un sink che viene gestito dalla copia #3, per cui la ricezione degli update consuma altri 3 kbps.

Far gestire un sink a un nodo ha un certo costo in banda in uscita, approssimabile dal numero di news in uscita a cui va sommato il costo degli update moltiplicato per il numero di peer presenti nel sistema. Tale valore è valido però solo in prima approssimazione, poiché RTPS prevede un proprio protocollo di routing e di disseminazione, quindi un DataWriter che debba raggiungere N DataReader non necessariamente deve mandare N pacchetti, ma tale compito può essere delegato a altri elementi della rete.

Il costo del carico della CPU può essere stimato allo stesso modo, ma tale approssimazione risulta imprecisa per i motivi appena esposti. Inoltre è difficile effettuare prove di carico al variare del numero di peer senza utilizzare un numero di computer in numero paragonabile a un deployment reale. Da prove effettuate senza stressare molto la CPU a 600MHz risulta che l'aggiunta di un peer al sistema di smistamento comporti un carico aggiuntivo di circa il 4%, paragonabile a quello di un sink aggiuntivo. Tale costo è dovuto alla necessità di inviare e ricevere gli update, e dipende dal numero di sink serviti da ambo i nodi.

Conclusioni

Il middleware RTI DDS si è rivelato molto flessibile è permette di gestire agevolmente la QoS, sfruttando se possibile anche le facility messe a disposizione dal supporto di rete.

Un grande punto di forza di DDS è il meccanismo di discovery che gli consente di individuare i participant del dominio e creare la rete P2P senza bisogno di interventi particolari da parte dell'applicazione che utilizza il middleware, al più solo indicando, se necessario, gli indirizzi IP di alcuni participant.

La scelta di utilizzare copie attive indipendenti porta a una penalizzazione di performance non trascurabile, che nel peggiore dei casi causa un overhead che cresce linearmente con il numero di copie in funzione. È quindi necessario non aumentare in modo indiscriminato i peer del sistema di smistamento. Nel caso in cui si determini che non è importante replicare lo stato di ogni elemento sarebbe possibile eliminare del tutto il meccanismo di invio degli update e si renderebbe molto più semplice il join al sistema. Questo permetterebbe di scalare molto meglio, ma con limitazioni dovute al fatto che comunque le notizie generate da un unico source devono essere consegnate a più peer.

Se l'approccio data centric è molto comodo per la disseminazione delle informazioni, risulta poco adatto per tutte quelle azioni che si tenderebbe a modellare come chiamate a procedure remote. Questo handicap è visibile ad esempio nella macchinosità del protocollo

utilizzato per il join di un peer.

L'assenza in DDS del concetto di “identità” di un produttore, necessaria per il disaccoppiamento, ha fatto sì che fosse necessario implementare un nostro meccanismo di heartbeat per individuare crash dei nodi.