RE.VE.N.GE.: implementazione tramite RTI-DDS e replicazione del sistema di consegna

progetto di Marco Altini, Stefano Bonetti, Giuseppe Cardone

Reti di Calcolatori LS - relazione di Stefano Bonetti

Prof. Antonio Corradi Ing. Luca Foschini

Abstract

L'evoluzione delle applicazioni distribuite da un contesto di tipo "best-effort" come Internet verso un mondo differenziato dal punto di vista della qualità del servizio rappresenta ormai una tendenza consolidata. Offrire diversi livelli di qualità di servizio, nonchè monitorarne l'effettivo rispetto, consente di modellare un generico servizio su misura di chi ne fruisce, adeguando risorse impegnate (e costi) al reale bisogno dell'utenza.

Nell'ambito di un contesto di distribuzione di notizie tra generiche fonti e generici fruitori, il presente progetto si prefigge l'obiettivo di offrire una qualità di servizio completamente configurabile sia ai primi che ai secondi, garantendo accanto ad essa un buon livello di availability tramite un'architettura replicata.

1. Introduzione

Scopo del progetto è la realizzazione ed il deployment di un sistema per la distribuzione di notizie su larga scala tra fonti e fruitori. Gli obiettivi principali nella realizzazione del middleware sono i seguenti:

- *disaccoppiamento* delle parti in gioco per garantire la massima *eterogeneità* possibile tra fonti e fruitori
- garanzia di supporto a diversi livelli di *qualità del servizio* sia per le fonti che per i fruitori, direttamente configurabile da questi ultimi
- *replicazione* del servizio di consegna dei messaggi, con lo scopo di incrementare l'affidabilità del sistema.

Un vincolo tecnologico molto importante riguarda l'utilizzo di RTI-DDS, implementazione proprietaria dello standard OMG Data Distribution Service for Real-time Systems. Risulta cruciale in questo senso un'analisi della tecnologia DDS volta ad individuare a quali funzionalità può provvedere interamente l'infrastruttura e dove invece occorre intervenire con uno strato di protocollo ulteriore: in quest'ultimo caso occorrerà valutare attentamente l'intrusione a fronte dei vantaggi.

A seguito di una descrizione generale dell'applicazione, la seguente relazione si occuperà nello specifico delle problematiche relative alla fault tolerance del sistema progettato. Il resto della relazione è organizzata come segue: nel secondo capitolo viene brevemente descritto il middleware RTI-DDS, in particolar modo le caratteristiche interessanti per lo sviluppo dell'applicazione in oggetto. Nel terzo capitolo è presentata l'architettura generale del sistema e una prima serie di scelte progettuali. Il quarto, il quinto ed il sesto capitolo affrontano nel dettaglio le ipotesi di guasto formulate, i meccanismi di identificazione dei crash e i protocolli di recovery. I capitoli sette e otto presentano

alcuni test effettuati sul sistema e le relative conclusioni.

2. Tecnologie utilizzate

RTI-DDS è un middleware sviluppato da Real-Time Innovations con lo scopo di assistere lo sviluppo e il deployment di applicazioni distribuite real-time. DDS si fonda su un modello di comunicazione publish-subscribe, come previsto dallo standard OMG, fornendo alle applicazioni una serie di API per un'interazione di tipo data-centric, ovvero basata sullo scambio di dati in tempo reale da più sorgenti a più destinazioni. Segue una breve descrizione delle entità fondanti la piattaforma:

- *domain*: aree di comunicazione completamente isolate le une dalle altre
- *domainParticipant*: entità radice appartenente ad un solo domain, responsabile della creazione delle altre entità
- *topic*: partizioni tipate dello spazio globale di comunicazione. Sono identificati dal nome e dal tipo di dato associato (definito mediante OMG IDL)
- *publisher/subscriber*: entità che si occupano di inviare/ricevere effettivamente i dati. Possono creare e gestire diversi dataWriter/dataReader
- *dataWriter/dataReader*: endPoint di comunicazione associati ad uno specifico topic. Sono utilizzati dalle applicazioni per le operazioni di write/read sul topic stesso

Seguendo lo standard OMG ed estendendolo, RTI-DDS rende possibile la configurazione di numerosi parametri di qualità del servizio, associati alle diverse entità. Ai fini dell'applicazione realizzata, sono state individuati alcuni servizi utili, per esempio:

- *heartbeat* per il monitoraggio della liveness delle entità, con intervallo di invio e soglia di alert configurabili. Si può quindi delegare a DDS il compito di rilevare possibili situazioni di guasto di un nodo, agganciandovi il relativo codice di gestione
- *affidabilità* della comunicazione, realizzata mediante ritrasmissione dei messaggi fino alla conferma di avvenuta ricezione. Parametri come il tempo d'attesa dell'ack, la lunghezza della coda di ritrasmissione sono configurabili
- suddivisione dei topic in *partizioni* identificate da un nome, allo scopo di mettere in comunicazione solo un sottoinsieme di dataReader/dataWriter all'interno dello stesso topic

Ovviamente, ove possibile, funzionalità e servizi sono stati delegati al middleware in modo da alleggerire il carico di progettazione.

3. Architettura generale

Ad un livello di astrazione elevato, le tre entità presenti nel sistema sono fonte, fruitore e sistema di consegna, che nel seguito verranno denominate rispettivamente *source*, *sink* e *system*. Il system è l'entità centrale ed incorporerà tutte le funzionalità chiave dell'applicazione, quali dispatching delle news e controllo della qualità di servizio. Il progetto è stato svolto in modo da ridurre il più possibile le responsabilità di source e sink. È auspicabile infatti che questi ultimi possano essere utilizzati su dispositivi portatili con limitate capacità computazionali, come ad esempio un PDA.

Ad una prima analisi, i servizi che il system dovrà fornire sono due (fig.1):

- registrazione dei source e dei sink, nonchè della qualità di servizio da loro richiesta

- ricezione e inoltro delle notizie secondo la qos stabilita

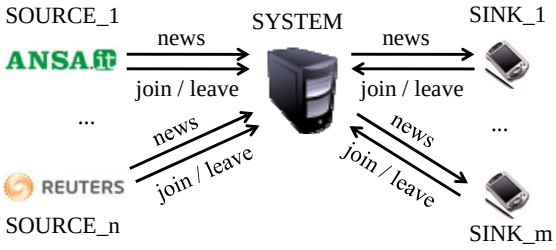

Figura 1. servizi forniti dal sistema

Come detto sopra, verrà utilizzato RTI-DDS come supporto alla comunicazione tra le entità, che dovrà pertanto essere basata su flussi di dati tipati tra *dataReader* e *dataWriter*. Per quanto riguarda la ricezione/dispatching delle notizie occorre stabilire quanto si vuole rendere personalizzabile la qualità di servizio per gli utenti. Stabilire staticamente la QoS (uno o più livelli) porterebbe all'utilizzo di un numero limitato di endpoint di comunicazione per lo smistamento delle notizie, diminuendo significativamente la configurabilità della comunicazione stessa. Per ottenere massima flessibilità si è quindi optato per una qualità del servizio differenziata e completamente personalizzabile per ogni source/sink: quello che ne consegue sono N+M canali (monodirezionali nel caso best-effort, bidirezionali nel caso reliable) dove N e M sono il numero dei source e il numero dei sink registrati (a livello implementativo vengono sfruttati due topic partizionati in N ed M settori).

Il sistema impone la QoS al servizio utilizzando delle *code interne*: in esse le notizie vengono bufferizzate per imporre una rate di consegna, vengono ordinate secondo la priorità delle fonti ed eventualmente eliminate nel caso la loro deadline di consegna non sia stata rispettata. Per ulteriori approfondimenti sul tema della QoS e sulla comunicazione source-system-sink si rimanda alla relazione di Altini.

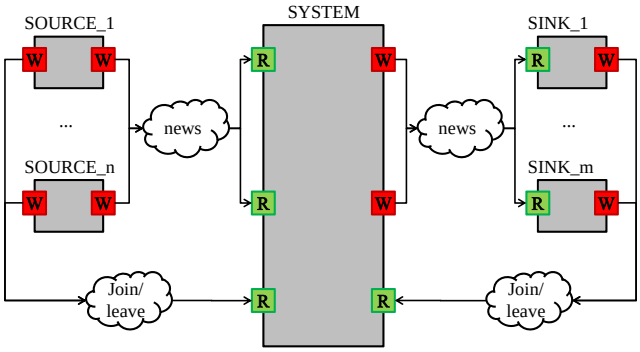

Figura 2. schema generale dell'applicazione

Con lo scopo di incrementare la fault tolerance, è stata supportata la replicazione spaziale del sistema, in modo completamente trasparente ai client di produzione e consumo delle news. La scelta del modello di replicazione da applicare è stata influenzata dall'utilizzo dell'infrastruttura DDS: il naturale supporto al multicast e la reliability sulla consegna dei messaggi rendono infatti il modello a copie attive molto meno complicato da implementare. Un modello di tipo master/slave, per contro, andrebbe a snaturare il modello di DDS, fortemente orientato alla comunicazione tra pari.

La scelta è quindi ricaduta su un *sistema replicato a copie attive indipendenti*, come illustrato in figura 3. Il percorso interno che il sistema segue all'arrivo di una notizia è il seguente:

1. tutte i sistemi copia ricevono la notizia prodotta dal source. Il multicast è realizzato da DDS. È importante notare come, grazie alla comunicazione totalmente disaccoppiata di DDS, *l'astrazione di sistema unico* e la conseguente *trasparenza alla replicazione* siano realizzate in modo molto semplice, e soprattutto senza l'introduzione di punti di centralizzazione (frontend);

2. ogni sistema si occupa di un certo numero di sink. I sistemi lavorano quindi indipendentemente l'uno dall'altro in load balancing di tipo statistico. Ogni sistema tuttavia possiede tutti i dati necessari per poter sostituire una copia guasta nel supporto ai suoi sink, ovvero mantiene lo stato delle code interne anche per i sink gestiti da altri sistemi-copia. Per l'analisi della fault tolerance si vedano i capitoli successivi

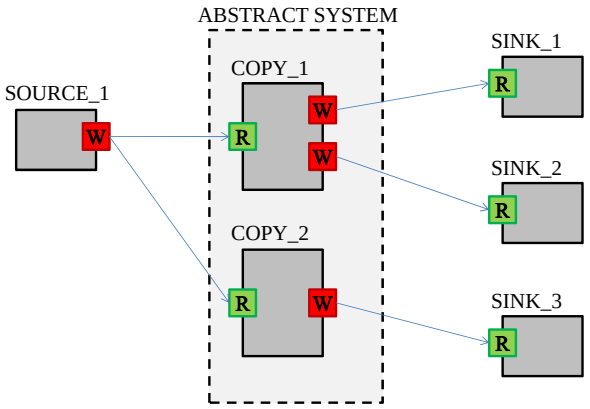

Figura 3. sistema replicato

3. ogni volta che il sistema invia una notizia ad uno dei suoi sink, è necessaria una fase di sincronizzazione con le altre copie. Dal momento che ogni entità del sistema così con ogni notizia è identificata da un guid univoco, i messaggi di sincronizzazione saranno quadruple univoche di update del tipo:
    - guid della notizia inviata
    - guid del sistema-copia che ha effettuato l'invio
    - guid del source origine della notizia
    - guid del sink di destinazione.

   Le tempistiche dell'aggiornamento variano soprattutto in base alla bufferizzazione operata da DDS nell'invio dei messaggi dai dataWriter ai dataReader.

Per approfondimenti sulla replicazione del sistema si veda la relazione di Cardone.

4. Fault tolerance e RTPS

Nel formulare le ipotesi di guasto è fondamentale capire quali sono le potenzialità di DDS, che garanzie può offrire e dove invece occorre intervenire a livello applicativo. Il protocollo RTPS di OMG utilizzato da DDS si appoggia su un generico livello di trasporto best effort offrendo un servizio di tipo "guaranteed load" particolarmente adatto ad applicazioni realtime non tolleranti. La garanzia che i messaggi arrivino tutti a tutti i destinatari e nel giusto ordine di pubblicazione è dovuta al reinvio dei messaggi sulla base di ack sia positivi che negativi, nonchè alla presenza di *code di send* e *di receive* rispettivamente presso i dataWriter e i dataReader. Tramite le prime si possono mantenere i messaggi per i quali non sono ancora pervenuti tutti gli ack relativi ai dataReader agganciati al topic. Le seconde servono in caso vengano ricevuti messaggi non in ordine, nel qual caso i messaggi successivi a quello non pervenuto vengono salvati in coda, mentre viene inviato un ack negativo per chiedere la ritrasmissione del messaggio mancante. Il caso delle code piene rappresenta una situazione critica: l'invio di un messaggio o la sua ricezione portano all'eliminazione del più vecchio messaggio tra quelli presenti in coda.

La reliability del protocollo è configurabile tramite il settaggio di svariati parametri, come ad esempio la lunghezza delle code e il massimo tempo di blocco in scrittura in caso di coda di send piena. La reliability effettiva è però raggiunta solo permettendo a DDS di allocare code di lunghezza illimitata. Questa

politica è sconsigliabile: in una situazione di congestione, infatti, DDS potrebbe occupare memoria senza controllo.

Ad un'ipotesi di crash di un nodo occorre quindi aggiungere quella di perdita di messaggi, benchè la probabilità di un comportamento simile venga significativamente ridotta dall'utilizzo di RTPS.

5. Read/write omission e recovery: protocollo di *whois*

L'ipotesi di *general omission* formulata nel capitolo precedente ha portato ad evidenziare alcuni settori dell'applicazione in cui l'omissione di un messaggio può causare problemi di consistenza abbastanza gravi.

In particolare, la registrazione di source e sink, così come il protocollo di join di un nuovo nodo sistema, costituiscono fasi di sincronizzazione che portano ognuno dei sistemi-copia a conoscere quali altre entità siano presenti nel dominio, siano queste altri sistemi-copia, source o sink. La piena situazione di consistenza garantisce che ogni sistema riceva correttamente da ogni source e inoltri le notizie ai sink di sua competenza. Si supponga invece che all'atto della registrazione di un source, un sistema-copia non riceva il messaggio e pertanto ignori la presenza di una fonte (si veda fig. 4). Come conseguenza non verrà istanziato il relativo dataReader ed eventuali sink agganciati al sistema-copia in questione non riceveranno notizie.

Nello specifico, ogni sistema dovrebbe avere piena conoscenza di:

- tutti i source registrati e la qos richiesta
- tutti i sistemi-copia esistenti
- tutti i sink registrati e la qos richiesta
- lo stato corrente delle associazioni copia-sink, ovvero la copia che si occupa del dispatching verso un sink

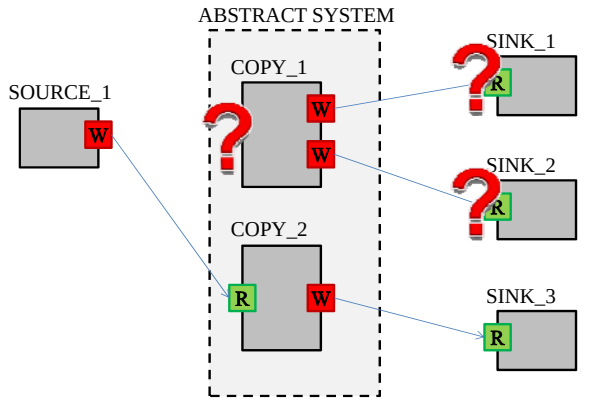

Figura 4. send/receive omission alla registrazione di un source

A fronte dei ragionamenti appena svolti, si rende necessaria la presenza di una meccanismo di recovery che si appoggi DDS e che permetta di ripristinare la consistenza.

Il protocollo di *WHOIS* si colloca precisamente in quest'ottica e sfrutta i messaggi di update che circolano sul topic di sincronizzazione delle copie. Quando un sistema-copia riceve una quadrupla di aggiornamento (source, sink, sistema e notizia) verifica se è a conoscenza delle prime tre entità. In caso contrario utilizza un topic apposito per chiedere alle altre copie informazioni sulle entità presenti nel dominio ed eventuali informazioni su di esse (qos richiesta). Il protocollo è schematizzato in figura 5. Il messaggio di whois conterranno:

- tipo dell'entità (source, sink, system)
- guid dell'entità

e verranno ricevuti da tutte le copie presenti.

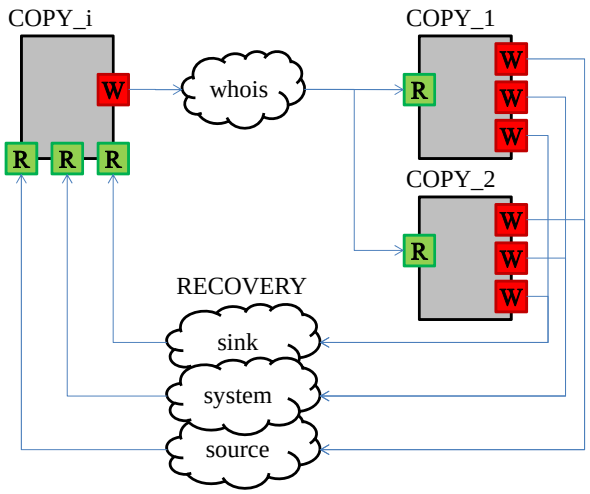

Figura 5. protocollo di whois

Tenendo conto che il numero di copie sarà limitato e che si considera l'inconsistenza una situazione poco frequente, è ammissibile che ognuno dei sistemi-copia risponda alle richieste di *whois*. Alcune ottimizzazione in questo senso possono essere fatte per quanto riguarda i sink (risposta solo da parte della copia che se ne occupa), e i sistemi (risposta solo da parte della copia stessa). Successivamente la copia richiedente potrà leggere su uno dei topic di recovery i dati relativi all'entità richiesta e aggiornare il proprio stato interno.

Benchè il protocollo descritto agisca con un certo ritardo la possibilità che una notizia vada persa è ragionevolmente limitata.

6. Caduta dei nodi e recovery

L'introduzione di replicazione spaziale migliora la fault tolerance dell'applicazione, evitando le vulnerabilità di un servizio centralizzato su un unico nodo. L'architettura a copie attive proposta consente di tollerare guasti sulle copie attive riducendo significativamente il numero di notizie perse.

La vulnerabilità principale cui il sistema risulta esposto riguarda i dati presenti nelle code interne in attesa di essere inviate ai sink. In caso di caduta di un nodo sistema, le notizie nelle code interne andrebbero perse. Come detto nell'introduzione, ogni copia mantiene lo stato delle code ed è quindi in grado di sostituire una copia guasta.

La ripartizione dei sink gestiti dal copia caduta dovrà essere tempestiva per mantenere un buon livello di *liveness*. Il dominio dei guid è suddiviso in un numero di intervalli pari al numero delle copie: alla registrazione di un sink, il sistema-copia che se ne occupa è scelto sulla base del guid del sink. All'arrivo di una nuova copia, così come alla sua rimozione, gli intervalli vengono ricalcolati e i sink rimasti senza copia vengono distribuiti sui sistemi-copia presenti. Disponendo di un'infrastruttura publish-subscribe come DDS, è possibile realizzare la migrazione di un sink tra una copia e l'altra del sistema di consegna in modo completamente trasparente al sink: mentre quest'ultimo può mantenere lo stesso dataReader in ascolto sulla partizione di topic, mentre lato sistema viene istanziato un nuovo dataWriter sulla copia che sostituisce quella guasta.

Per le operazioni di recovery descritte sopra viene assunto un *single point of failure* sulle copie, per semplificare la procedura di migrazione dei sink e di sincronizzazione globale delle entità.

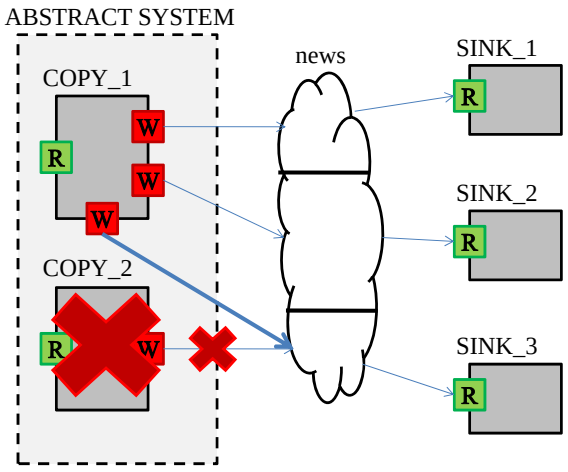

Figura 6. crash di una copia

Vediamo ora come individuare le situazioni di guasto delle diverse entità. Escludendo a priori comportamenti di guasto malizioso (failure bizantine), il metodo più efficace per rilevare un guasto su un nodo è utilizzare un heartbeat. Il layer di discovery automatica di DDS offre una serie di callback implementabili per reagire ad eventi di registrazione di un nuovo dataReader o dataWriter su un topic. L'evento scatta:

- presso il dataWriter ogniqualvolta si registra o si cancella un dataReader che può leggere i suoi messaggi
- presso il dataReader ogniqualvolta si registra o si cancella un dataWriter che può inviargli messaggi

All'interno di ogni domainParticipant sono presenti 3 dataReader e 3 dataWriter "built-in", che servono a inviare e ricevere informazioni sullo stato dei participant, dei dataReader e dei dataWriter presenti nel dominio. Il meccanismo di discovery è configurabile tramite due parametri:

- *tempo di assert*: intervallo di tempo ogni quanto viene inviato il messaggio di heartbeat
- *tempo di lease*: intervallo di tempo trascorso il quale, se non sono giunti heartbeat, scatta un evento e l'eventuale callback. L'evento contiene informazioni su quanti dataReader/dataWriter si affacciano sul topic e la differenza con lo stato precedente

6.1. Caduta di un nodo source

Il caso più semplice da gestire risulta la caduta di una fonte. Sia N il numero dei source registrati. Ogni sistema possiede N dataReader e il topic "news" utilizzato è suddiviso in N partizioni. Alla caduta di un source verrà a mancare l'heartbeat: sapendo presso quale dataReader scatta l'evento, il sistema saprà quale fonte è caduta e provvederà a rimuovere le relative entità. È fondamentale, specialmente quando si utilizza un'infrastruttura come RTI-DDS, eliminare le entità non più in uso per evitare lo spreco di banda dovuto al protocollo di discovery

6.2. Caduta di un nodo sink

Il caso è leggermente diverso dal precedente in quanto un sink riceve notizie da un solo sistema. Possedendo quest'ultimo un dataWriter che scrive su un topic partizionato. Sulla partizione relativa ad un sink insistono quindi solo un dataWriter ed un dataReader, pertanto il sistema è in grado di capire quale sink è caduto e rimuovere le entità relative. La copia che rileva il guasto dovrà avvisare gli altri sistemi con un messaggio apposito.

6.3. Caduta di un nodo sistema

Il meccanismo di discovery offerto da DDS possiede un limite importante, dato dalla modalità di interazione data-centric: in un topic molti-a-molti non ci si può accorgere quale specifico dataReader o dataWriter si sia sganciato dal topic. Per questo motivo non c'è modo di capire quale sistema-copia sia caduto utilizzando solo gli heartbeat di discovery. Perciò è stato realizzato a livello applicativo un *heartbeat firmato* (un messaggio IMFINE), ovvero contenente il guid del sistema che lo genera. Ogni sistema-copia dovrà predisporre un numero di timer pari al numero delle altre copie presenti e resettarli alla ricezione dei diversi IMFINE. Esattamente come nel caso dell'heartbeat built-in di DDS, possiamo decidere ogni quanto inviare il messaggio di IMFINE e dopo quanto tempo dichiarare caduto un nodo sistema, cercando un ragionevole compromesso tra traffico utilizzato e prontezza di reazione dell'applicazione.

Tempi nell'ordine di grandezza del secondo sembrano costituire il miglior compromesso.

7. Test

L'applicazione è stata testata in modo da esaminare l'impiego di banda e il carico del sistema sul processore. I test sono stati effettuati facendo girare un sistema-copia su una macchina monitorata, e utilizzando altre due macchine per lanciare source, sink e altri sistemi.

In figura 7 è mostrato il grafico riguardante l'impiego di banda di un nodo copia al verificarsi di diversi eventi, indicati sotto.

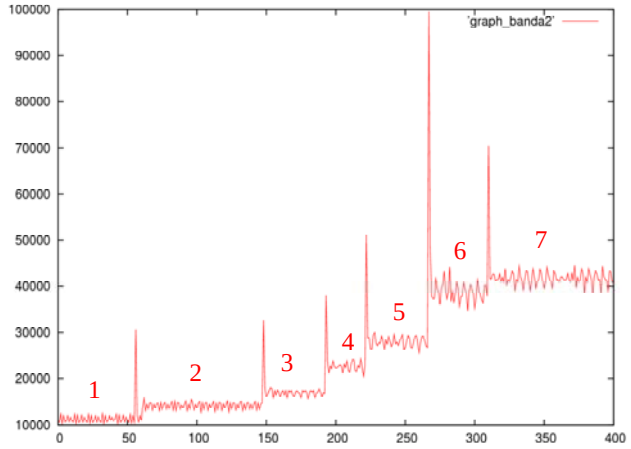

Figura 7. banda occupata su un nodo-sistema

1. **due copie attive, la copia #1 sulla macchina monitorata**. La banda occupata dalle due copie a riposo (12 KB/sec) ci permette di valutare l'intrusione di DDS e dei meccanismi di discovery, in aggiunta ai messaggi di IMFINE inviati e ricevuti
2. **un source attivo con rate costante da 10 news/sec**. Ulteriori 3 KB/sec dovuti alle notizie in entrata (notizie di test da circa 300 byte l'una)
3. **un sink attivo, gestito dalla copia #2**. Ulteriori 3 KB/sec dovuti agli update in arrivo dalla copia #2
4. **un sink attivo, gestito dalla copia #1**. Ulteriori 3 KB/sec per l'invio delle notizie al sink, e altri 3 KB/sec per l'invio degli update
5. **un sink attivo, gestito dalla copia #1**. Come al punto 4
6. **copia #3 attiva**. Ulteriori 10 KB/sec dovuti a 6 KB/sec di update, più overhead dovuto a DDS e ai messaggi di IMFINE
7. **un sink attivo, gestito dalla copia #3**. Come al punto 3.

Il test permette di capire il livello di intrusione di un sistema a copie attive come quello presentato. Considerando la banda globale occupata, la registrazione di un nuovo sink occupa una fetta costante di banda per l'invio delle notizie (c'è sempre una sola copia che effettua l'invio), e una componente crescente con il numero delle copie, dovuta agli update. In figura 8 è mostrato l'andamento teorico della banda globale occupata per gestire un sink al variare del numero di nodi, supponendo un rate di 10 news/sec.

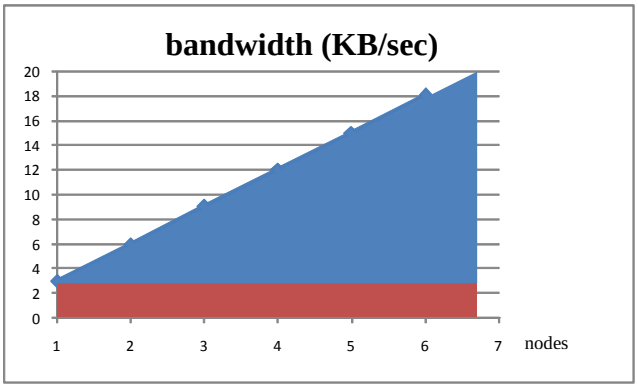

Figura 8. banda globale alla registrazione di un sink

Nell'ipotesi semplificativa in cui notizie e update abbiano la stessa dimensione, avere un sistema con un numero ragionevole di copie, per esempio 5, porta il costo di inserimento di un sink a 5 volte il costo di un inserimento nel caso di sistema unico.

Un ulteriore test è stato svolto per identificare il carico limite del processore

(600 MHz) su cui gira un sistema. Il test è stato condotto con un unico nodo sistema e 10 sink. Ad una frequenza di 600 news/sec si sono ottenuti malfunzionamenti dovuti al raggiungimento della soglia limite. I risultati sono riportati in figura 9, e sono soddisfacenti, tenendo conto che il limite individuato costituisce un rate di notizie estremamente alto.

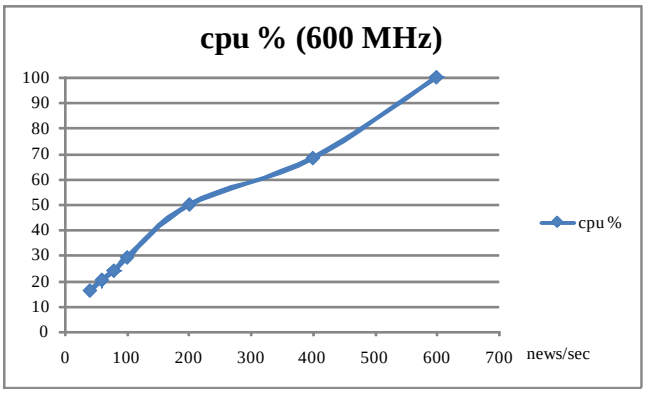

Figura 9. carico di cpu al variare del rate di notizie

8. Conclusioni

L'applicazione realizzata consente alle fonti e ai fruitori di personalizzare il canale di comunicazione configurando alcuni parametri. Una soluzione flessibile introduce dei costi che si è provato di minimizzare delegando il più possibile l'infrastruttura DDS dei compiti da svolgere.

Per ottenere un buon livello di disponibilità di servizio si è fatto ricorso alla replicazione del sistema di consegna, scegliendo un'architettura a copie attive per seguire la natura P2P-oriented di DDS. Le copie si spartiscono i sink in modo casuale, lavorando in bilanciamento di carico. Il costo di coordinamento e sincronizzazione tra le copie è risultato linearmente proporzionale al numero delle copie: un numero di copie limitato è quindi consigliabile per avere buona fault-tolerance e non far crescere troppo l'overhead.

In alcuni frangenti è stato necessario introdurre uno strato di gestione sopra a DDS, per realizzare funzionalità specifiche (protocollo di whois, heartbeat firmato, ...), operando scelte sulla base dell'intrusione aggiuntiva.