A Reliable Message Oriented Middleware based on “Publish and Subscribe” Paradigm

Mirko Matoffi 171419 CdLS Ingegneria Informatica (0234)
Reti di Calcolatori LS
a.a. 2003/2004

Abstract
Le applicazioni distribuite sono sempre più diffuse nel panorama tecnologico attuale e sempre più ad esse vengono richieste proprietà quali scalabilità, robustezza, flessibilità e sicurezza. In questo scenario si colloca come attore protagonista il middleware basato sullo scambio di messaggi. Il sistema qui proposto vuole mettere in luce le caratteristiche di questo tipo di middleware. A quanto finora citato va aggiunto il contributo di forte disaccoppiamento dato dal paradigma di comunicazione “Publish and Subscribe” scelto per la nostra soluzione e la struttura della piattaforma di servizio di tipo “cluster per high availability” che garantisce la proprietà di affidabilità. Pertanto potremmo riassumere il sistema qui presentato come un’infrastruttura affidabile per la comunicazione asincrona business-to-business con supporto per lo scambio di documenti XML.

Keywords: Publish/Subscribe, Message Oriented Middleware, Cluster per high availability, Heartbeat protocol, XML data exchanging.

1 Introduzione

Un *Message Oriented Middleware* è un’architettura che permette la comunicazione basata su scambio di messaggi tra due o più applicazioni. La sua caratteristica più rilevante risiede nella asincronicità della comunicazione che conferisce flessibilità e possibilità di uso in ambienti dinamici. Tipicamente in questo contesto sono possibili due diverse tipologie di comunicazione, il *”Message Queuing”* e il *“Publish and Subscribe”*; la prima fornisce un modello “point-to-point” in cui ogni messaggio viene recapitato dal sistema verso il suo destinatario, mentre la seconda fornisce un modello “many-to-many” facilitando la comunicazione in sistemi distribuiti su larga scala. Quest’ultimo paradigma sta prendendo sempre più piede negli ultimi anni in quanto permette un maggiore disaccoppiamento tra le applicazioni coinvolte derivante dal fatto che non c’è alcun obbligo che il mittente conosca i destinatari del messaggio. Un sistema publish/subscribe può essere classificato in due categorie: “topic-based” e “content-based”. Nel primo caso il mittente specifica per ogni messaggio a quale topic appartiene, scegliendolo tra un insieme prefissato; nel secondo invece ogni messaggio è creato secondo un preciso schema dal mittente, mentre la sottoscrizione altro non è che una query sullo schema dei messaggi. Il sistema che andremo qui a proporre usa il paradigma publish/subscribe topic-based; estensioni future possono essere effettuate per incapsulare anche le altre tipologie di comunicazione.

2 Architettura logica del sistema

Il sistema sviluppato segue le indicazioni contenute nelle specifiche JMS, considerando

però solo la parte relativa al paradigma di comunicazione “Publish and Subscribe”.

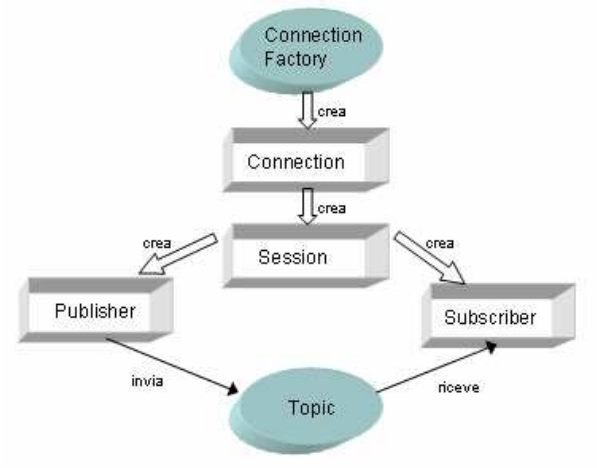

Figura 1 : Architettura logica del sistema

- *Connection Factory*: è un “administered object” responsabile della creazione della connessione; il client dovrà rivolgersi ad esso per ottenere la connessione al servizio.
- *Connection*: rappresenta una connessione attiva tra l’applicazione ed il middleware.
- *Session*: canale di comunicazione con uno specifico Topic.
- *Publisher*: responsabile dell’invio dei messaggi al Topic.
- *Subscriber*: responsabile della ricezione dei messaggi provenienti dal Topic
- *Topic*: è l’altro “administered object” e rappresenta una possibile destinazione per i messaggi; è responsabile anche del loro smistamento.

Gli “*administered object*” sono oggetti creati dall’amministratore e facenti parte del middleware lato server; sono quindi accedibili dal client solo da remoto. Gli altri oggetti fanno parte del middleware lato client e da quest ultimo sono utilizzati per comunicare con il topic; in particolare la comunicazione remota avviene tramite l’oggetto *Connection* che è l’unico oggetto lato client dipendente dal tipo di connessione remota utilizzata. Questo accentramento di responsabilità in un unico oggetto è molto importante perché permette di cambiare modalità di connessione senza intervenire nel resto dell’architettura; inoltre è anche possibile fornire più metodi di connessione e lasciare all’utente la facoltà di scegliere quale utilizzare. Quest’ultimo aspetto è importante anche per quanto riguarda la robustezza del sistema, in quanto, disponendo di più tipologie di connessione, se fallisce una, è sempre possibile provare con altre. Proprio per questa indipendenza dell’architettura dal tipo di connessione effettuata, nei paragrafi seguenti ci disintiresseremo del mezzo e delle modalità attraverso cui avviene la comunicazione, per tornare poi a parlarne verso la fine dell’articolo.
Nella figura 2 viene fornita una visione più specifica dell’architettura del sistema lato server.

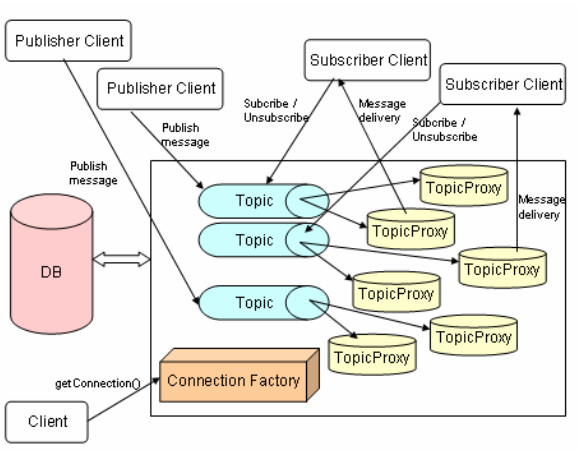

Figura 2 : Dettagli architettura lato Server

Oltre a Topic e Connection Factory già visti in precedenza ora c’è un nuovo oggetto, il *TopicProxy* che rappresenta il ClientSubscriber nell’ambiente Server. Esso è responsabile della consegna dei nuovi messaggi del topic al client e del mantenimento di essi in memoria qualora il Client non sia connesso al sistema.

Qualunque sia il tipo di connessione utilizzato, il sistema avrà bisogno di un **Sistema di Nomi** che permetta il dovuto disaccoppiamento tra Client e Server; l’amministratore dovrà quindi registrare gli “administered object” per poi permettere il loro recupero al Client che ne avesse bisogno.

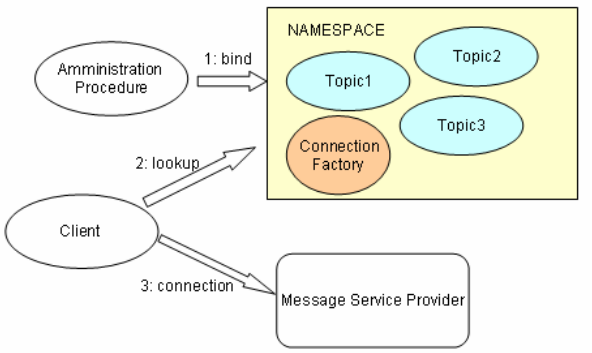

Figura 3: Sistema di nomi

3 Struttura dei messaggi

La struttura dei messaggi usati dall'applicazione è formata da un *Header*, un insieme di *MessageProperty* e dal *Body*.
L' ***Header*** consiste dei seguenti campi:

- *Destinazione*: nome del topic cui appartiene il messaggio.
- *MessageID*: identificativo univoco del messaggio generato automaticamente dal sistema.
- *CorrelationID*: usato per collegare una risposta alla sua richiesta; contiene il *MessageID* del messaggio a cui si vuole rispondere.
- *DeliveryMode*: tipologia di consegna; può essere valorizzato come *"Persistent"* o come *"NonPersistent"* (vedi più avanti per ulteriori dettagli).
- *Timestamp:*contiene l'ora in cui il messaggio viene spedito.
- *Expiration:* contiente l'istante temporale in cui il messaggio perde la sua validità; di solito viene settato dal publisher come la somme tra il timestamp attuale ed il *timeToLive* impostato nel publisher stesso.
- *Priority:*rappresenta la priorità del messaggio; è un valore intero compreso tra 0 e 9, con 0 lievllo minimo e 9 massimo; il valore di dafault è 4.

Qualora alcune funzionalità non interessassero al Client, è possibile impostare il *Publisher* in modo che non tenga conto di alcune di esse, settando opportunamente i seguenti attributi:

- *disableMsgID*
- *disableMsgTimestamp*

Altri attributi del publisher che vanno ad influire sulle proprietà dei messaggi da esso creati e inviati sono:

- *deliveryMode*
- *priority*
- *timeToLive*

Le ***MessageProperties*** altro non sono che un meccanismo che permette di estendere l'header di un messaggio; una proprietà è una coppia "nome-valore" che viene associata al messaggio e può essere usata dalle applicazioni come selettore di messaggi o per comunicare qualche proprietà relativamente al messaggio stesso.
Il ***Body*** contiene il corpo del messaggio rappresentato da un oggetto *java.lang.String*; la scelta di usare questo specifico formato è legata al fatto che in un oggetto della suddetta classe è facile introdurre un intero file **XML** che, se già non lo è, nel prossimo futuro sarà lo standard più usato per lo scambio di informazioni tra applicazioni. Quindi, anche se nascosto dentro una stringa, possiamo affermare che il sistema in questione supporta lo scambio di dati basato su XML.

Esistono due tipologie di consegna dei messaggi, quella persistente (*"Persistent Delivery Mode"*) e quella non persistente (*"Non Persistent Delivery Mode"*). La modalità non persistente è quella che introduce il minor overhead in quanto non richiede la memorizzazione dei messaggi ma, se il Client non fosse disponibile non riceverebbe il messaggio (**semantica *at-most-once***). Per contro la modalità persistente prevede la memorizzazione dei messaggi qualora essi il Client non fosse collegato al sistema (**semantica *once-and-only-once***).

4 Architettura Cluster

Le entità finora introdotte sono quelle definite nelle specifiche JMS più qualche entità creata ad hoc (vedi TopicProxy) e rappresentano la base che permette al middleware di far comunicare tra loro due applicazioni che non si conoscono. Esse non sono però sufficienti qualora al sistema venga richiesta anche la caratteristica di affidabilità ed alta disponibilità.

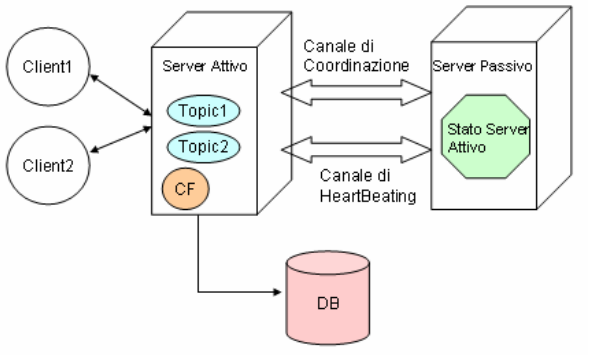

Figura 4: Configurazione cluster

Per soddisfare questi requisiti si è deciso di implementare un cluster di tipo *"Attivo-Passivo"* con una replicazione di tipo passivo *"Hot Stand by"*come mostrato in figura 4. Lo scopo del cluster è quello di garantire il servizio anche a fronte di un guasto del server primario in modo

del tutto trasparente per le applicazioni Client (Fig. 5).

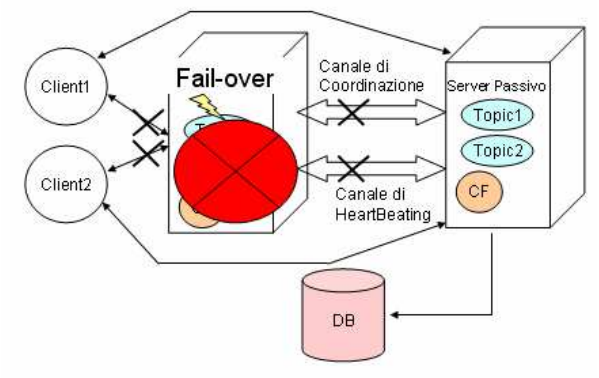

Figura 5: Configurazione cluster dopo un fail-over

In pratica finora abbiamo descritto il ServerAttivo; ora però emergono nuovi concetti ed entità che necessitano di essere approfonditi. Dalla immagine di Fig.4 si possono già notare la presenza di un *"canale di coordinazione"*, di un *"canale di heartbeating"* e di uno *"stato del server attivo"* memorizzato nel server in stand by.

Concentriamo innanzitutto la nostra attenzione sullo ***stato del server attivo***; quali sono le informazioni che lo caratterizzano? Sicuramente, se si vuole garantire la continuità del servizio, dovranno essere memorizzati i topic attivi , la connection factory ed i proxy utilizzati per inviare i messaggi ai destinatari.
Il ***canale di coordinamento***, invece, è una connessione attiva tra il server primario e il server in stand by attraverso la quale quest ultimo riesce a raccogliere informazioni sullo stato del primo in modo da essere aggiornato nel momento in cui dovesse verificarsi un guasto che impedisca l'operato del server primario. Ci sono diversi modi e tecnologie per realizzare una connessione, ma perché non utlizzare gli strumenti che il middleware oggetto di replicazione ci mette a disposizione? Perché non realizzare un **topic che funga da canale di coordinamento?** D'altronde le caratteristiche del sistema fin qui descritte ben si addicono a questa funzione. L'applicazione server attiva dovrà quindi:

1. creare un topic con un nome prefissato, che chiameremo "CoordinationTopic";
2. registrare il CoordinationTopic nel Name registry;
3. creare un publisher associato a quel topic e configurarlo in modo da inviare messaggi con un formato prestabilito;
4. inviare un messaggio tramite il publisher ogni qual volta si verifichi un cambiamento delle informazioni di stato inviando le infromazioni rilevanti.

Per contro l'applicazione server passiva dovrà:

1. creare una connessione all'applicazione nel server primario tramite la Connection Factory;
2. creare un subscriber per il CoordinationTopic;
3. interpretare correttamente i messaggi ricevuti dal subscriber, aggiornando opportunamente lo stato;

In pratica per realizzare un cluster non abbiamo fatto altro che costruire una applicazione client che usa l'infrastruttura che stiamo progettando; dato che, come più volte ripetuto, il modello "publish/subscribe" utilizzato dal middleware, è una tipologia di comunicazione "molti a molti", possiamo aumentare il numero di server passivi a costo zero (dal punto di vista del software e del carico computazionale, non certo da quello hardware). Il fatto che dei server possano essere aggiunti o rimossi dal cluster secondo necessità rende la soluzione adottata estremamente scalabile.

Lo schema logico utilizzato per la realizzazione del cluster ("logico" perché abbiamo solo descritto la modalità di funzionamento ma non abbiamo finora mai parlato di come connettere fisicamente le parti in gioco), può benissimo essere utilizzato anche in locale, ad esempio sul server primario. In questo caso realizziamo una replicazione delle risorse secondo un modello passivo di tipo "hot stand by", aumentando ancora di più l'affidabilità del sistema e quindi il suo livello di disponibilità.

Come si può ben notare dalla figura 5, non è solo la parte server del middleware in questa operazione di aumentare l'affidabilità del sistema tramite un cluster, ma anche la parte client. Essa infatti dovrà essere in grado, a fronte di un guasto sul server primario, di riuscire a localizzare il server secondario che si è attivato per garantire la continuità di servizio. Dato che la modalità di reindirizzamento è fortemente influenzata dal tipo di connessione utilizzata dal middleware, essa verrà descritta unitamente alla tipologia di connessione che si è scelto di utilizzare per l'implementazione di questa infrastruttura.

4.1 Heartbeat

Manca da descrivere il ***canale di heartbeating***; il sistema di Heartbeat è un'infrastruttura di supporto essenziale per il corretto funzionamento del cluster. Essa è responsabile del monitoraggio

sulla effettiva presenza delle risorse critiche per il funzionamento del sistema. Si basa tipicamente su messaggi di "*I'm alive*" che la risorsa monitorata deve periodicamente inviare in broadcast. In caso di mancata ricezione da un tot di tempo di tale messaggio, il responsabile del monitoraggio considera la risorsa come non più funzionante e informa chi di dovere, in modo che vengano eseguire le adeguate operazioni atte a garantire la continuità del servizio.
Dalle caratteristiche evidenziate emerge la possibilità di costruire questa infrastruttura come un'applicazione che usa il middleware che si sta progettando. Infatti la necessità di supportare il broadcast e di mettere in comunicazione applicazioni che non necessitano di conoscersi sono alcune delle problematiche che il message oriented middleware risolve. Così, come avvenuto per il canale di coordinamento, **anche il canale di heartbeating può essere implementato tramite un topic**.

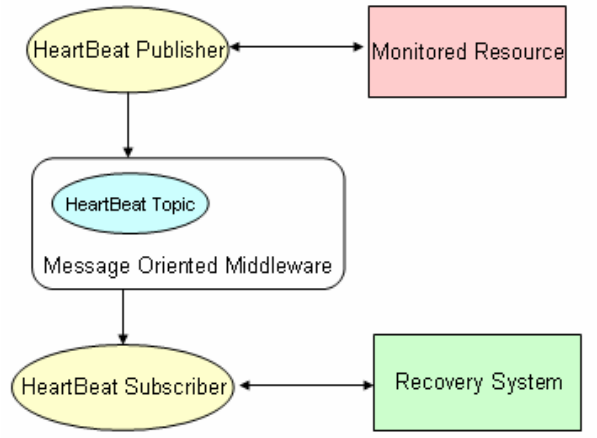

Figura 6: Infrastruttura per l'Heartbeat

Quella mostrata nella figura 6 è una infrastruttura del tutto generale e può essere usata in qualsiasi applicazione distribuita che abbia bisogno di questa funzionalità; la presenza del collegamento tra l'"HeartBeat Publisher" e la "Monitored Resource" è dovuta al fatto che la generazione del messaggio di *Alive* potrebbe anche dipendere da un controllo che il publisher effettua sullo stato della risorsa monitorata. Contestualizzando l'infrastruttura proposta al nostro caso, avremmo che la "Monitored resource" non sarebbe altro che l'applicazione server attiva, mentre il "Recovery System" è l'applicazione server in standby (Fig. 7). In caso di mancata ricezione del messaggio di *Alive*, il subscriber informerà l'applicazione server in standby che provvederà ad attivarsi caricando lo stato del server attivo precedentemente salvato e garantendo la continuità di servizio.

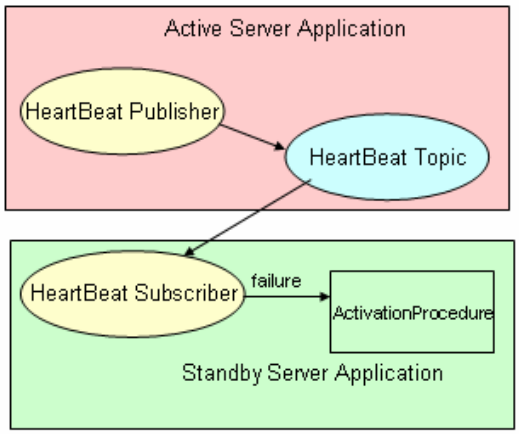

Figura 7: Implementazione dell'infrastruttura di Heartbeat

Come nel caso del canale di coordinamento, anche per quanto riguarda il canale di heartbat le due tipologie di applicazioni, quella attiva e quella in stand by, hanno diverse responsabilità. In particolare l'applicazione attiva dovrà:

1. creare l'Heartbeat topic e registrarlo sul registro di nomi con un nome prefissato;
2. creare un publisher relativo al topic di cui sopra e impostarlo correttamente per l'invio di messaggi di *Alive*;
3. creare un Heartbeat publisher che, usando il publisher appena creato, invii i messaggi con un determinata frequenza.

Mentre l'applicazione in standby dovrà:

1. creare una connessione con il server attivo;
2. creare una sessione con il topic predefinito;
3. creare un subscriber per ricevere i messaggi provenienti dal topic;
4. creare un Heartbeat subscriber che controlli con una determinata frequenza l'avvenuta ricezione di un messaggio di *Alive* e che, in caso negativo, invochi la procedura di attivazione.

4.2 Lo snapshoter

Precedentemente è stato affermato che la sincronizzazione tra server attivo e server in standby avviene tramite il *canale di comunicazione*; con il sistema descritto ciò è possibile, soltanto che richiederebbe che ogni topic fosse a conoscenza del canale di comunicazione. Per rispettare il "*Principio di minima intrusione*" si è scelto di non "sporcare" le entità server per far loro svolgere compiti che non dovrebbero competere loro e quindi si è mantenuto il canale di comunicazione soltanto per informare riguardo la creazione di nuovi topic, mentre, per quanto riguarda i proxy, il

coordinamento avviene tramite lo ***"snapshoter"***. Lo "snapshoter" altro non è che un processo che con una certa frequenza si rivolge al server in standby per avere l'elenco dei topic presenti sul server attivo per poi interrogare i suddetti topic in modo da recuperare l'elenco dei subscriber afferenti ad esso e restituire gli elenchi creati al server in standby. Del *TopicProxy* viene memorizzato quindi solo il riferimento remoto del subscriber collegato ad esso; questo riferimento verrà poi usato dal server in standby per ripristinare la comunicazione con i vari Client in caso di guasto del Server primario. La comunicazione tra server in standby e snapshoter avviene secondo il pattern "observer".

5 Scrivere una applicazione Client

Creare un'applicazione Client è molto semplice; per prima cosa è necessario importare i **package** *middleware.common* e *middleware.client*. Quindi si deve creare un'istanza della classe *ClientConnectionManager* passando come parametri l'indirizzo e la porta del server primario e quelli del server secondario. Tramite essa si può creare una connessione (classe *Connection*) e poi rivolgersi a quest'ultima per creare una o più sessioni (classe *Session*). A questo punto, per ogni topic a cui si è interessati, è necessario creare un *Publisher* e/o un *Subscriber* a seconda del ruolo che si vuole che il Client assuma per quel topic. È importante sottolineare che la sessione rappresenta un contesto "single-threaded", pertanto se si creano più publisher tramite una stessa sessione, le operazioni di invio effettuate da questi vengono sequenzializzate. Se si vuole avere la possibilità di inviare più messaggi contemporaneamente ai vari topic è necessario utilizzare più sessioni; invece non conviene, in generale, usare più connessioni visto che questi sono oggetti pesanti dal punto di vista delle risorse utilizzate. Per creare un *Publisher* o un *Subscriber* è necessario invocare l'apposito metodo dell'oggetto *Session* indicando il topic a cui deve far riferimento; per individuare un topic è sufficiente passare il suo nome, sarà poi l'oggetto *Connection* a mappare questo nome in un riferimento remoto all'oggetto *Topic* a secondo della tipologia di connessione utilizzata.

5.1 Usare l'oggetto *Publisher*

Prima di inviare messaggi al topic tramite il publisher associato è opportuno, a meno che non vadano bene i parametri di default, settare le modalità di invio dei messaggi ed i rispettivi parametri. È infatti possibile impostare la modalità di consegna, disabilitare il timestamp o il messageID, definire per ogni messaggio inviato la sua priorità ed il suo expirationTime.
Ci sono due modi differenti per inviare un messaggio: si può inviare una stringa che verrà poi incapsualta in un oggetto *TextMessage* dal publisher stesso, oppure si può inviare direttamente un *TextMessage* creato autonomamente; questa seconda modalità è da usare qualora si vogliano impostare delle "*message properties*".

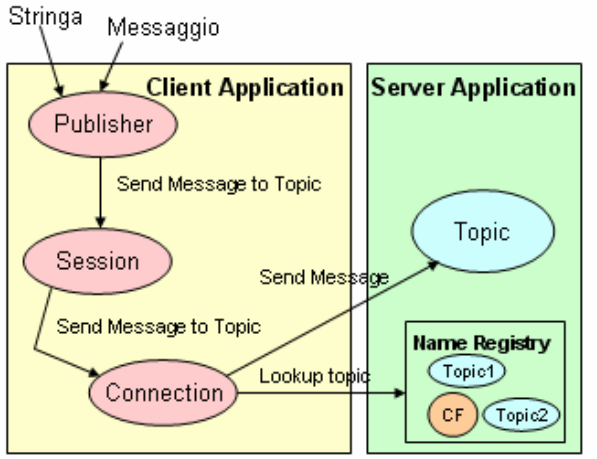

Figura 8: Invio di un messaggio

5.2 Usare l'oggetto *Subscriber*

Per ricevere messaggi da un topic, oltre a creare l'oggetto *Subscriber*, è necessario definire una classe che implementi l'interfaccia *ImessageObserver*. Una volta creato un oggetto di questa classe, esso deve essere passato al *Subscriber* (pattern Observer), il quale, ad ogni messaggio ricevuto, invocherà l'apposito metodo dell'observer fornendo il messaggio all'applicazione Client.

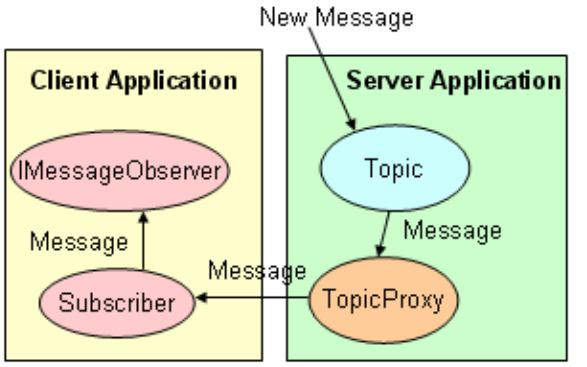

Figura 9: Ricezione di un messaggio

6 Connessione tramite Java RMI

Abbiamo esposto come è possibile costruire un'applicazione Client senza peraltro sapere quale è il tipo di connessione tra Client e Server: tutto quello che il client deve conoscere è l'indirizzo del server ed i nomi dei topic a cui è interessato. Questa considerazione mette bene in mostra la qualità di disaccoppiamento tra l'architettura del middleware e la connessione che essa usa; sarebbe quindi possibile cambiare il tipo di connessione andando a toccare solo le poche classi interessate da questo cambiamento o, perché no, offrire al client la possibilità di scegliere tra più tipologie di comunicazione.
Andiamo però ad analizzare la scelta fatta nell'implementazione attuale del sistema e le conseguenze che essa ha sulle sue caratteristiche: Java RMI.

6.1 Gli *"Administered object"*

Gli "Administered object" (di classe Topic e ConnectionFactory) rappresentano nel contesto RMI, dei fornitori di servizio che devono essere registrati sul *registry* e che devono implementare l'interfaccia che permette ai Client remoti di comunicare con loro. Per questo motivo la classe *Topic* implementa l'interfaccia *ITopicRMI* mentre la classe *ConnectionFactory* implementa *IConnectionFactoryRMI*.

6.2 *ClientConnectionManager* e *Connection*

Le classi responsabili della comunicazione con questi oggetti remoti sono *Connection* e *ClientConnectionManager* che usano rispettivamente l'interfaccia *ITopicRMI* e l'interfaccia *IConnectionFactoryRMI*. L'oggetto di classe *Connection* deve essere serializzabile in quanto viene fornito al Client dall'oggetto remoto di tipo *ConnectionFactory*; in Java RMI la connessione non è altro che la chiamata ad un metodo di un oggetto remoto ed infatti l'oggetto di tipo *Connection* usato dal Client per comunicare con il Server, non fa altro che recuperare dal *registry* attivo sul server, il riferimento remoto al topic il cui nome viene fornito dall'applicativo. Per ottimizzare gli accessi remoti al registry, non viene effettuato un lookup ad ogni richiesta di invio o ad ogni sottoscrizione per un topic, ma questa operazione viene svolta solo la prima volta, dopodichè la *Connection* memorizza la coppia "nome del topic – riferimento remoto" in una tabella che poi verrà usata per recuperare il riferimento nelle seguenti richieste verso il topic

6.3 Il *Subscriber*

Come mostrato in figura 9, la comunicazione non avviene soltanto dal Client verso il Server, ma anche nell'altro verso; è necessario quindi fare uso della ***RMI Callback***. Per questo la classe *Subscriber* implementa l'interfaccia *IClientRMI* che viene usata dal relativo proxy per inviare i nuovi messaggi; il riferimento remoto arriva al proxy tramite i seguenti passaggi:

1 l'applicazione Client richiede all'oggetto *Session* la creazione di un *Subscriber*;
2 l'oggetto *Session* crea il *Subscriber* e, prima di restituirlo all'applicazione Client richiede all'oggetto *Connection* di effettuare la sottoscrizione al topic associato;
3 l'oggetto *Connection* recupera il riferimento remoto al *Topic* e gli passa il riferimento remoto al *Subscriber* e l'identifcativo del Client che vuole iscriversi a quel topic;
4 il *Topic* riceve il riferimento remoto sottoforma di *IClientRMI*; se esiste già un *TopicProxy* per quel Client gli comunica il nuovo riferimento remoto a cui inviare i messaggi, altrimenti crea un nuovo *TopicProxy* e lo inizializza con quel riferimento;
5 il *TopicProxy* userà il riferimento remoto per fornire al Client i nuovi messaggi e, qualora il Client non fosse più disponibile, memorizzerà i messaggi in attesa il Client torni ad essere on line.

6.4 Procedura di Recovery

Nel capitolo 4 abbiamo descritto l'architettura cluster utilizzata in modo da rendere il Server affidabile; perché però tutto il middleware sia affidabile è necessario che anche il Client esegua opportune operazioni in caso di guasto del Server attivo. Abbiamo già detto che l'oggetto *Connection* effettua il lookup per reperire il riferimento remoto di *Topic* sconosciuti, dopodichè usa quel riferimento per comunicare. Se il server dove è presente fisicamente l'oggetto *Topic* dovesse avere un guasto, ogni tentativo di chiamata di metodi tramite riferimenti remoti produrrebbe un'eccezione lato Client. Possiamo interpretare questa eccezione come la mancata ricezione di un messaggio di *Alive* in un sistema di Heartbeat; pertanto, qualora si verificasse un'eccezione l'oggetto *Connection* deve intercettarla e chiedere al

ClientConnectionManager di ricercare un nuovo server attivo. Infatti esso possiede l'indirizzo non solo del server primario, ma anche di quello secondario. Trovato il nuovo server il *ClientConnectionManager* ritornerà poi alla *Connection* i nuovi parametri per effettuare il lookup.
Per quanto riguarda la ricezione dei messaggi, in caso di guasto del server primario, non c'è bisogno di alcun reindirizzamento in quanto il server in standby, tramite lo "snapshoter" memorizza per ogni topic i riferimenti remoti dei suoi subscribers.

7 Tool di amministrazione

Lato Server si ha a disposizione un tool di amministrazione con interfaccia grafica che permette di avviare l'applicazione server impostando tutti i parametri necessari. In figura 10 è riportato un esempio di creazione di un'applicazione server attiva.

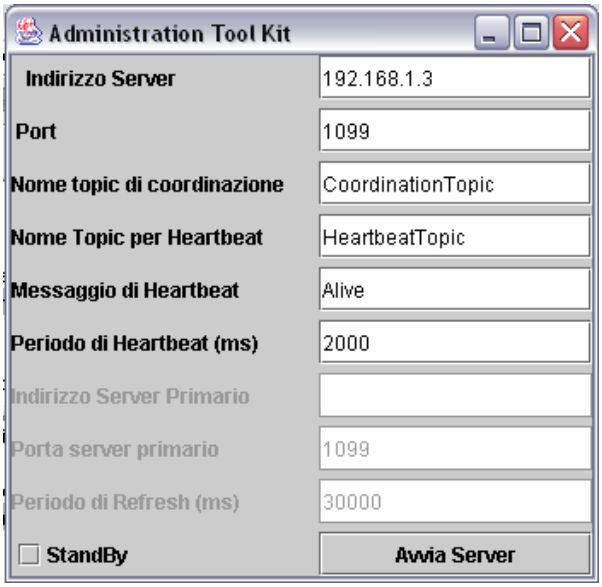

Figura 10: Interfaccia di amministrazione

Se si fosse voluto crearne una in standby sarebbe stato sufficiente selezionare l'apposita checkbox ed inserire gli ulteriori parametri richiesti (riguardanti il server primario). Il periodo di refresh indica ogni quanti millisecondi lo "snapshoter" deve andare in esecuzione. Una volta avviato il Server, compare un'ulteriore interfaccia grafica (Fig. 11) tramite la quale è possibile creare nuovi topic e vedere quelli già creati.

Figura 11: Interfaccia di gestione topic

Come si può notare dalla figura 11 c'è anche la possibilità di creare una **gerarchia di topic**; in questo caso un messaggio inviato ad un certo topic viene ricevuto dai Client iscritti a quel topic a anche da tutti quelli iscritti a topic posti ad un livello più alto nella gerarchia.

8 Risultati dei test

I test eseguiti hanno come obiettivo evidenziare come evolve il tempo impiegato dal Server in standby ad attivarsi, in base alle entità in gioco. Queste entità sono quelle che compongono lo stato del server: i Topic ed i Subscriber. Sono stati quindi fatti variare questi valori e registrati i tempi di attivazione. Come Server sono stati utilizzati due macchine distinte collegate tramite LAN a 100Mbps:
- *Macchina A*: CPU Pentium 4 a 2,4GH con 512KB di cache di secondo livello e 256MB di RAM.
- *Macchina B*: CPU AMD Athlon64 3200+ a 2GH con 512KB di cache di secondo livello e 512MB di RAM.

Sulla Macchina A sono in esecuzione il Server attivo, i subscribers ed i publishers; sulla Macchina B, che è poi quella dove verranno presi i tempi, soltanto il Server in standby.
I parametri di configurazione utilizzati sono:
- periodo di Heartbeat = 2 secondi;
- periodo dello snapshoter = 30 secondi.

Nel calcolo del tempo di attivazione non è stato considerato il periodo di Heartbeat in quanto esso è configurabile dall'esterno; pertanto il tempo calcolato non è altro che il tempo utilizzato dal Server in standby per caricare lo stato del Server attivo precedentemente salvato.

Risultati ottenuti:

Numero Topic	Subscribers per Topic	Totale Subscribers	Tempo di attivazione (ms)
6	300	1800	1188
6	500	3000	1718
8	300	2400	1250
8	500	4000	1953
10	300	3000	1922
10	500	5000	2250

Dai dati risultanti emerge che, anche a fronte di un numero discretamente elevato di subscriber, il sistema reagisce sempre con tempi accettabili. A fronte di una triplicazione dl numero di subscriber, si ha solo un raddoppio del tempo di caricamento degli oggetti. Come detto precedentemente, per avere il tempo effettivo di inattività a fronte di un guasto bisogna sommare al tempo riportato in tabella quello che intercorre tra l'avvenimento del guasto e il suo rilevamento. Un ottimo upperbound a questo tempo è fornito dal valore che si è scelto di assegnare al periodo di Heartbeat (nel nostro caso 2 secondi).

9 Conclusioni e sviluppi futuri

In questo articolo abbiamo mostrato come riesce il middleware basato su scambio di messaggi qui proposto a garantire le proprietà di affidabilità e disaccoppiamento promesse; nella attuale realizzazione ci sono però dei limiti derivanti dalla scelta di usare Java RMI come strumento di interconnessione tra Client e Server. Innanzitutto l'applicazione Client deve conoscere l'indirizzo fisico di dove si trova il Server attivo per poter inviare o ricevere messaggi con problemi nel caso si voglia ottenre indipendenza dalla locazione fisica del fornitore del servizio. Un possibile sviluppo futuro atto a rimuovere questo limite potrebbe essere **l'introduzione di un DNS** che permetta l'uso di nomi logici piuttosto che indirizzi fisici. L'altra limitazione è dovuta al fatto di essere legati alla tecnologia Java; per superarla ed aumentare così l'interoperabilità si potrebbe **usare SOAP** e portare tutto sul web.
Lasciando da parte la tipologia di connessione, una possibile funzionalità da aggiungere in modo da far aumentare la QoS è l'**implementazione di politiche di sicurezza**. Ciò permetterebbe all'amministratore di limitare il diritto di publishing e/o subscribing su di uno specifico topic solo a certi Client piuttosto che altri. Sempre nell'ottica di aumentare la QoS si potrebbe fornire all'utente anche la **modalità di comunicazione punto-a-punto** ed estendere la modalità publish-and-subscribe in modo che sia possibile affettuare anche **sottoscrizioni di tipo content-based**.

Riferimenti

- Ying Liu, Beth Plale, *"Survey of Publish Subscribe Event Systems"*.
- J. Pereira, F. Fabret, F. Llirbat, H.A. Jacobsen, D. Shasha, *"WebFilter: A High-throughput XML-based Publish and Subscribe System"*.
- Han Li, Guofei Jiang, *"Semantic Message Oriented Middleware For Publish/Subscribe Networks"*.
- Java Message Service: http://java.sun.com/products/jms/
- Nicholas Whitehead, *"Listen to heartbeats using JMS"*.
- Gordon Van Huizen, *"JMS: An infrastructure for XML-based business-to-business communication"*.
- Eric Bruno, *"Simplify JMS with the Facade Design Pattern"*.
- Eiko Yoneki, *"Many Aspects of Reliability in a Distributed Mobile Messaging Middleware"*.
- Java Remote Metod Invocation (RMI): http://java.sun.com/products/jdk/rmi/

Sommario

1 Introduzione 1
2 Architettura logica del sistema 1
3 Struttura dei messaggi 3
4 Architettura Cluster 3
4.1 Heartbeat 4
4.2 Lo snapshoter 5
5 Scrivere una applicazione Client 6
5.1 Usare l'oggetto *Publisher* 6
5.2 Usare l'oggetto *Subscriber* 6
6 Connessione tramite Java RMI 7
6.1 Gli *"Administered object"* 7
6.2 *ClientConnectionManager* e *Connection* 7
6.3 Il *Subscriber* 7
6.4 Procedura di Recovery 7
7 Tool di amministrazione 8
8 Risultati dei test 8
9 Conclusioni e sviluppi futuri 9

Indice delle figure

Figura 1 : Architettura logica del sistema 2
Figura 2 : Dettagli architettura lato Server 2
Figura 3: Sistema di nomi 2
Figura 4: Configurazione cluster 3
Figura 5: Configurazione cluster dopo un fail-over 4
Figura 6: Infrastruttura per l'Heartbeat 5
Figura 7: Implementazione dell'infrastruttura di Heartbeat 5
Figura 8: Invio di un messaggio 6
Figura 9: Ricezione di un messaggio 6
Figura 10: Interfaccia di amministrazione 8
Figura 11: Interfaccia di gestione topic 8