Università degli Studi di Roma “Tor Vergata”
Facoltà di Ingegneria

Comunicazione nei Sistemi Distribuiti
(parte 2)

Corso di Sistemi Distribuiti

Valeria Cardellini
Anno accademico 2009/10

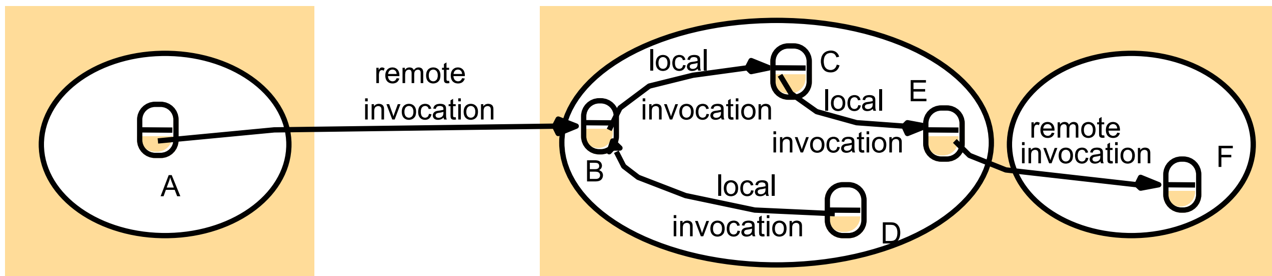
Java RMI: motivazioni

- Il paradigma RPC può essere facilmente esteso al modello a oggetti distribuiti
  – Java RMI (**Remote Method Invocation**): RPC in Java
  – RMI fornisce la possibilità di richiedere **l’esecuzione di metodi remoti** in Java, integrando il tutto con il paradigma OO

- Java RMI è un insieme di **strumenti**, **politiche** e **meccanismi** che permettono ad un’applicazione Java in esecuzione su una macchina di invocare i metodi di un oggetto di un’applicazione Java in esecuzione su una macchina remota

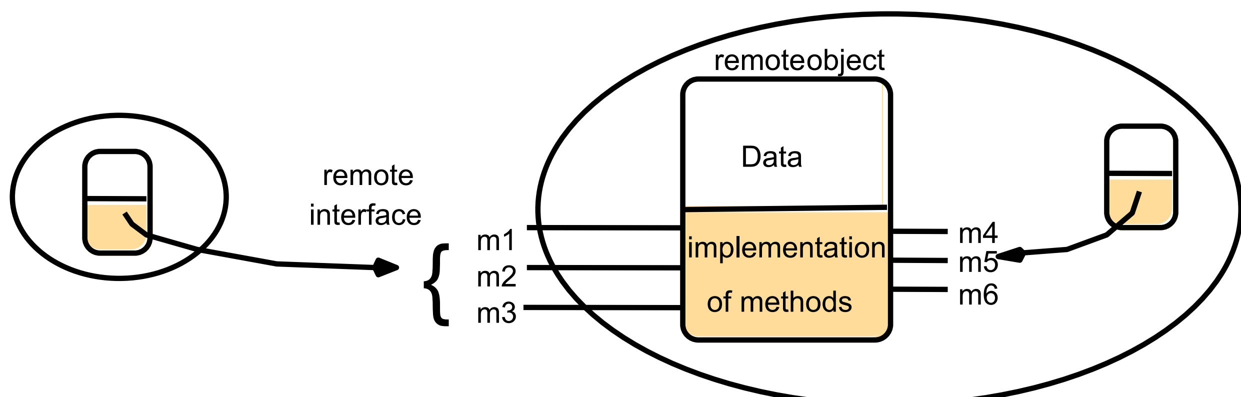
Java RMI: generalità

- Localmente viene creato solo il riferimento ad un oggetto remoto, che è invece effettivamente attivo su una macchina remota
- Un programma client invoca i metodi remoti attraverso questo riferimento locale
- Quali sono le differenza rispetto all'invocazione di un oggetto locale?
  – Affidabilità, semantica della comunicazione, durata, …



Java RMI: generalità (2)

- Principio base di Java RMI: la definizione del comportamento (interfaccia) e l'implementazione di quel comportamento (classe) sono concetti separati
- La separazione logica tra interfaccia ed oggetto consente anche la loro separazione fisica
  – **Interfaccia remota**: specifica quali metodi dell'oggetto possono essere invocati da remoto
  – Lo stato interno di un oggetto non viene distribuito!

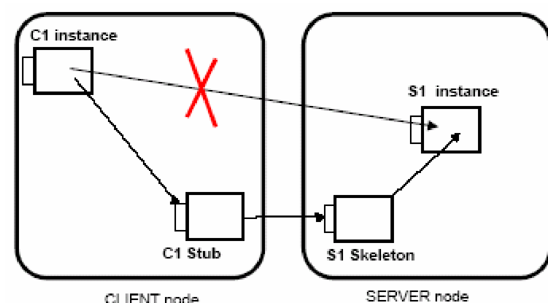


Java RMI: generalità (3)

- Al binding di un client con un oggetto server distribuito, una copia dell'interfaccia del server (stub) viene caricata nello spazio del client
  - Ruolo del tutto analogo a quello del client stub in ambiente RPC
- La richiesta in arrivo all'oggetto remoto viene trattata da un "agente" del client, locale al server (skeleton)
  - Ruolo del tutto analogo a quello del server stub in ambiente RPC
- Unico ambiente di lavoro come conseguenza del linguaggio Java, ma eterogeneità dei sistemi
  - Grazie alla portabilità del codice Java

Stub e skeleton

- Come in RPC, anche in RMI si usa il **proxy pattern**: i due proxy (*stub* dalla parte client e *skeleton* dalla parte server) nascondono al livello applicativo la natura distribuita dell'applicazione
  - Stub: proxy locale sul nodo client su cui vengono fatte le invocazioni destinate all'oggetto remoto
  - Skeleton: proxy remoto sul nodo server che riceve le invocazioni fatte sullo stub e le realizza effettuando le corrispondenti chiamate sull'oggetto server



Stub e skeleton (2)

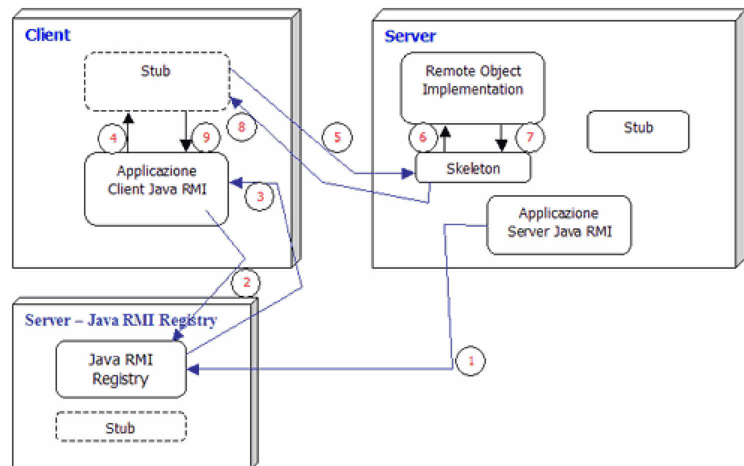
- Stub e skeleton rendono possibile la chiamata di un servizio remoto come se fosse locale, agendo da proxy
  - Generati dal compilatore RMI
- (De)serializzazione supportata direttamente da Java
  - Grazie all'uso del bytecode, non c'è bisogno di (un)marshaling come in RPC, ma i dati vengono (de)serializzati utilizzando le funzionalità offerte direttamente a livello di linguaggio
  - Serializzazione: trasformazione di oggetti complessi in sequenze di byte
    - metodo `writeObject()` su uno stream di output
  - Deserializzazione: decodifica di una sequenza di byte e costruzione di una copia dell'oggetto originale
    - metodo `readObject()` su uno stream di input

Interazione tra stub e skeleton

- Passi per la comunicazione
  1. Il client ottiene un'istanza dello stub (come?)
  2. Il client invoca i metodi sullo stub
  3. Lo stub effettua la serializzazione delle informazioni per l'invocazione (ID del metodo e argomenti) ed invia un messaggio allo skeleton contenente le informazioni
  4. Lo skeleton effettua la deserializzazione dei dati ricevuti, invoca la chiamata sull'oggetto che implementa il server (dispatching), effettua la serializzazione del valore di ritorno e lo invia allo stub in un messaggio
  5. Lo stub effettua la deserializzazione del valore di ritorno e restituisce il risultato al client

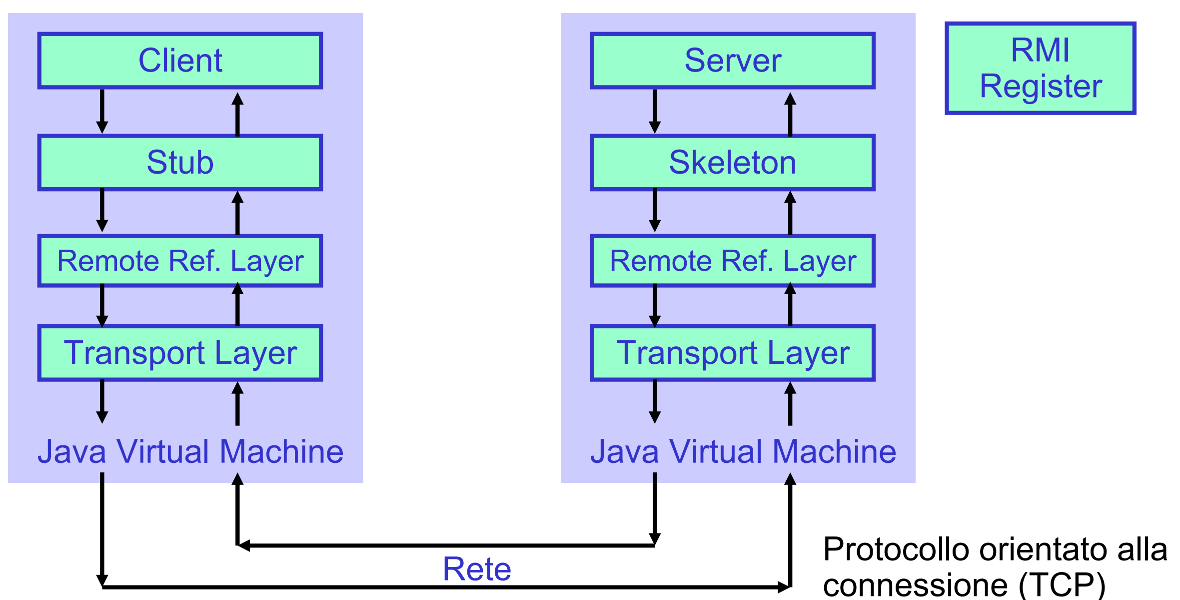
Interazione tra componenti in Java RMI

- Java RMI Registry: è il binder per Java RMI
- Offre un servizio di naming che consente al server di pubblicare un servizio e al client di recuperarne il proxy
- Funziona come punto di indirezione



Architettura Java RMI

- Architettura a strati
- Solo interazioni sincrone e bloccanti



Architettura Java RMI (2)

- Remote Reference Layer
  - Responsabile della gestione dei riferimento agli oggetti remoti e dei parametri e delle astrazioni di connessione orientata agli stream
  - Scambio di dati con il Transport Layer
- Transport Layer
  - Responsabile della gestione delle connessioni fra i diversi spazi di indirizzamento (JVM diverse)
  - Gestisce il ciclo di vita delle connessioni
  - Utilizzo di un protocollo proprietario
  - Possibilità di utilizzare protocolli applicativi diversi, purché siano orientati alla connessione
    - Ad es. HTTP per poter invocare oggetti remoti che sono in esecuzione su macchine protette da firewall

Passi per l'utilizzo di Java RMI

- Realizzare componenti remoti
  1. Definizione del comportamento: interfaccia che
     - Estende java.rmi.Remote
     - Propaga java.rmi.RemoteException
  2. Implementazione del comportamento: classe che
     - Implementa l'interfaccia definita
     - Estende java.rmi.UnicastRemoteObject
- È necessario:
  1. Definire interfacce ed implementazioni dei componenti utilizzabili in remoto
  2. Compilare le classi (con javac) e generare stub e skeleton (con rmic) delle classi utilizzabili in remoto
  3. Pubblicare il servizio
     - Attivare RMI Registry
     - Registrare il servizio (il server esegue bind/rebind su RMI Registry)
  4. Ottenere (lato client) il riferimento all'oggetto remoto tramite il name service, facendo un lookup su RMI Registry

A questo punto l'interazione tra il cliente e il server può procedere

Esempio echo: interfaccia

- L'interfaccia estende l'interfaccia Remote
- Ciascun metodo remoto
  - Deve lanciare una RemoteException
    - L'invocazione dei metodi remoti non è completamente trasparente
  - Ha un solo parametro di uscita; nessuno, uno o più parametri di ingresso
  - I parametri di ingresso devono essere passati per valore (dati primitivi o oggetti Serializable) o per riferimento (oggetti Remote)

```
public interface EchoInterface
    extends java.rmi.Remote{
  String getEcho(String echo)
    throws java.rmi.RemoteException;
}
```


Esempio echo: server

- La classe che implementa il server
  - Estende la classe UnicastRemoteObject
  - Implementa tutti i metodi definiti nell'interfaccia
- Un processo in esecuzione sulla macchina del server registra tutti i servizi
  - Invoca tante bind/rebind quanti sono gli oggetti server da registrare, ognuno con un nome logico
- Registrazione del servizio nel servizio di naming offerto dall'RMI registry
  - Bind e rebind possibili solo su registry locale

```
public class EchoRMIServer
  extends java.rmi.server.UnicastRemoteObject
  implements EchoInterface{
  //Costruttore
  public EchoRMIServer()
   throws java.rmi.RemoteException
  { super(); }
  //Implementazione del metodo remoto
         dichiarato nell'interfaccia
  public string getEcho(String echo)
    throws java.rmi.RemoteException
   { return echo; }
  public static void main(String[] args) {
    // Registrazione del servizio
  try
  { EchoRMIServer serverRMI =
         new EchoRMIServer();
    Naming.rebind("Echoservice", serverRMI); }
   catch (Exception e)
  {e.printStackTrace(); System.exit(1); }
 }
}
```

Esempio echo: client

- Servizi acceduti tramite la variabile interfaccia, ottenuta con una richiesta al registry
- Reperimento di un riferimento remoto
  – Ossia un'istanza di stub dell'oggetto remoto (non della classe dello stub, che di solito si assume già presente sul client)
- Invocazione del metodo remoto
  – Chiamata sincrona bloccante con i parametri specificati nell'interfaccia

```
public class EchoRMIClient
{
  //Avvio del client RMI
  public static void main(String[] args)
  {
    bufferedReader stdIn =
      new BufferedReader(
        new InputStreamReader(System.in));
    try
    {
      //Connessione al servizio RMI remoto
      EchoInterface serverRMI = (EchoInterface)
      java.rmi.Naming.lookup("EchoService");
      //Interazione con l'utente
      String message, echo;
      System.out.print("Messaggio? ");
      message = stdIn.readLine();
      //Richiesta del servizio remoto
      echo = serverRMI.getEcho(message);
      System.out.println("Echo: "+echo+"\n");
    }
    catch (Exception e)
    {e.printStackTrace(); System.exit(1); }
  }
}
```


Esempio echo: compilazione ed esecuzione

- Lato server
  1. Compilazione
     `javac EchoInterface.java`
     `javac EchoRMIServer.java`
  2. Generazione stub e skeleton
     `rmic [-vcompat] EchoRMIServer` → `EchoRMIServer_Skel.class` `EchoRMIServer_Stub.class`
  3. Esecuzione lato server
     Avvio del registry: `rmiregistry`
     Avvio del server: `java EchoRMIServer`
- Lato client
  1. Compilazione
     `javac EchoRMIClient.java`
  2. Esecuzione
     `java EchoRMIClient`

Confronto tra SUN RPC e Java RMI

- SUN RPC: visione a processi e non completa trasparenza all'accesso; operazioni richieste all'host del server
- Entità che si possono richiedere: solo operazioni o funzioni
- Semantica di comunicazione (default): at-least-once
- Modi di comunicazione: sincroni e asincroni
- Durata massima: timeout
- Ricerca del server: port mapper sul server
- Presentazione dei dati: linguaggio IDL ad hoc (XDR) e generazione stub con rpcgen
- Passaggio dei parametri: per copia-ripristino
  - Le strutture complesse e definite dal client sono linearizzate e ricostruite dal server stub per essere distrutte al termine dell'operazione
- Varie estensioni: broadcast, sicurezza…

Confronto tra SUN RPC e Java RMI (2)

- Java RMI: visione per sistemi ad oggetti con trasparenza all'accesso, con scelte che non privilegiano l'efficienza
- Entità che si possono richiedere: metodi di oggetti via interfacce
- Semantica di comunicazione: at-most-once
- Modi di comunicazione: solo sincroni
- Durata massima ed eccezioni: trattamento di casi di errore
- Ricerca del server: uso di registry con broker unico centrale
- Presentazione dei dati: generazione stub e skeleton con rmic
- Passaggio dei parametri: di default per valore, per riferimento di oggetti con interfacce remotizzabili

Comunicazione orientata ai messaggi

- RPC e RMI migliorano la trasparenza all'accesso
- Ma non sono sempre meccanismi adatti a supportare la comunicazione in un SD
  - Ad es. quando non si può essere certi che il destinatario sia in esecuzione
  - La natura essenzialmente sincrona delle RPC comporta una perdita di efficacia nella comunicazione
- Alternativa: comunicazione orientata ai messaggi
  - Di tipo transiente
    - Berkeley socket
    - **Message Passing Interface (MPI)**
  - Di tipo persistente
    - **Sistemi a code di messaggi** o **Message Oriented Middleware (MOM)**

Message Passing Interface (MPI)

- Libreria standard de-facto per lo scambio di messaggi tra processi diversi
  - E' solamente la specifica di una interfaccia, non una implementazione
    - Numerose implementazioni di MPI, tra cui Open MPI e MPICH
  - L'utilizzo principale di MPI è nel campo della computazione parallela e distribuita
- MPI prevede nelle proprie specifiche la definizione di una serie di primitive per la comunicazione tra processi
  - Primitive per la comunicazione punto-punto
  - Primitive per la comunicazione collettiva

Comunicazione punto-punto in MPI

- Per l'invio e la ricezione di un messaggio tra due processi diversi
- MPI_Send e MPI_Recv: comunicazione bloccante
  - MPI_Send con modalità sincrona o bufferizzata a seconda dell'implementazione
- MPI_Bsend: invio bloccante bufferizzato
- MPI_Ssend: invio sincrono bloccante
- MPI_Isend e MPI_Irecv: comunicazione non bloccante

Primitive MPI	Significato
MPI_Bsend	Aggiunge un messaggio in uscita ad un buffer per l'invio
MPI_Send	Invia un messaggio e aspetta finché non viene copiato in un buffer locale o remoto
MPI_Ssend	Invia un messaggio e aspetta finché non inizia la ricezione
MPI_Isend	Invia il riferimento ad un messaggio in uscita e continua
MPI_Recv	Riceve un messaggio; si blocca se non ce ne sono

Esempio di comunicazione in MPI

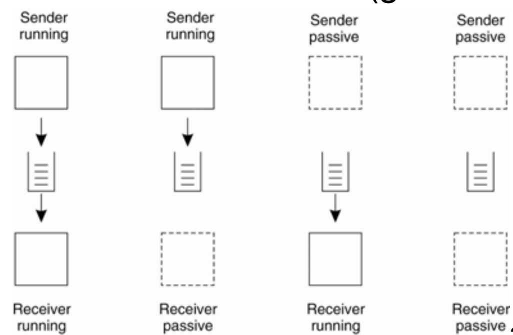
```
#include <stdio.h>
#include <string.h>
#include <mpi.h>
int main (int argc, char **argv) {
  int myrank;
  char message[20];
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  printf("Il mio rank e' : %d\n", myrank);
  if (myrank == 0) {
    //Invia un messaggio al processo 1
    strcpy(message, "PROVA");
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    printf("%d) Ho inviato: '%s'\n", myrank, message);
  } else if (myrank==1) {
    //Riceve il messaggio dal processo 0
    MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
    printf("%d) Ho ricevuto: '%s'\n", myrank, message);
  }
  MPI_Finalize();
  return 0;
}
```

MPI_Send(buf, count, datatype, dest, tag, comm)

MPI_Recv(buf, count, datatype, source, tag, comm, status)

Middleware orientato ai messaggi (MOM)

- Idea base di un **MOM** (o *sistema a code di messaggi*): le applicazioni distribuite comunicano inserendo **messaggi** in apposite **code**
  - Di solito ogni applicazione ha la sua coda privata (esistono anche code condivise da più applicazioni)
  - Eccellente supporto per comunicazioni persistenti e asincrone
- Il mittente ha soltanto la garanzia che il suo messaggio venga inserito nella coda del destinatario
  - Nessuna garanzia che il destinatario prelevi il messaggio dalla sua coda
  - Disaccoppiamento temporale tra mittente e destinatario (già esaminato per publish/subscribe)
- Un messaggio può contenere qualunque dato e deve essere indirizzato opportunamente
  - Nome univoco a livello del sistema

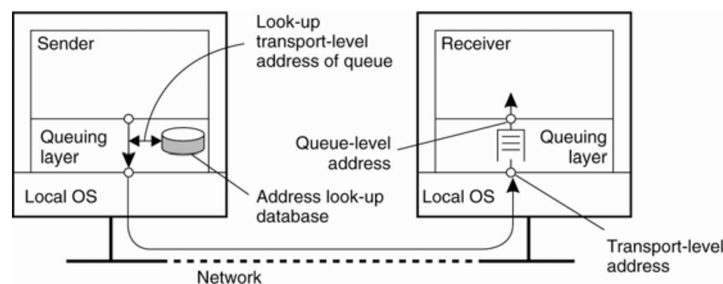


Middleware orientato ai messaggi (2)

- Principali primitive offerte dall'API di un MOM:
  - **Put**: invio non bloccante di un messaggio
    - Il messaggio viene aggiunto ad una coda
    - Operazione asincrona → come trattare il caso di coda piena?
  - **Get**: ricezione bloccante di un messaggio
    - Si blocca finché la coda non è più vuota e preleva il primo messaggio (oppure si blocca in attesa di un messaggio specifico)
    - Operazione sincrona rispetto alla presenza di messaggi in coda
  - **Poll**: ricezione non bloccante di un messaggio
    - Controlla se ci sono messaggi in una coda e rimuove il primo (oppure un messaggio specifico), senza bloccarsi
  - **Notify**: ricezione non bloccante di un messaggio
    - Installa un handler (funzione di callback) che avvisa il destinatario quando viene inserito un messaggio in una coda
    - Il meccanismo di callback separa la coda dall'attivazione del destinatario

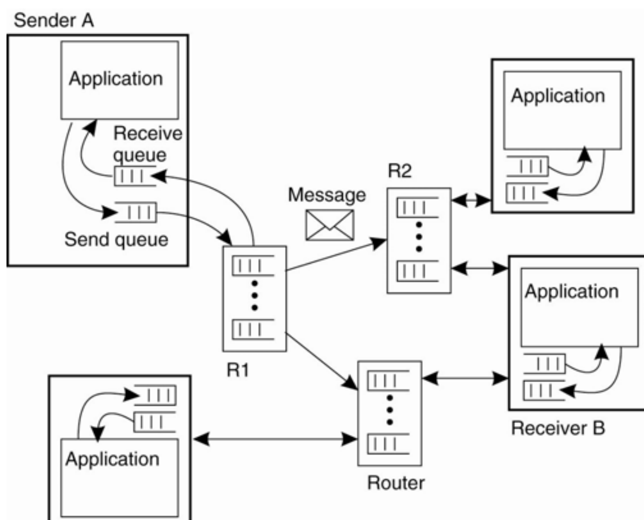
Architettura di MOM

- I messaggi possono essere inseriti/letti solo in/da code locali al mittente/destinatario
  - Inserimento in coda sorgente (o di invio)
  - Lettura da coda di destinazione (o di ricezione)
  - La coda appare locale sia al mittente che al destinatario (trasparenza della distribuzione)
  - E' il sistema ad occuparsi del trasferimento dei messaggi
- Il MOM deve mantenere la corrispondenza tra ogni coda ed la sua posizione sulla rete (servizio di naming)



Architettura di MOM (2)

- L'architettura generale di un MOM *scalabile* richiede un insieme di gestori (router o relay) specializzati nel servizio di routing dei messaggi da un gestore all'altro



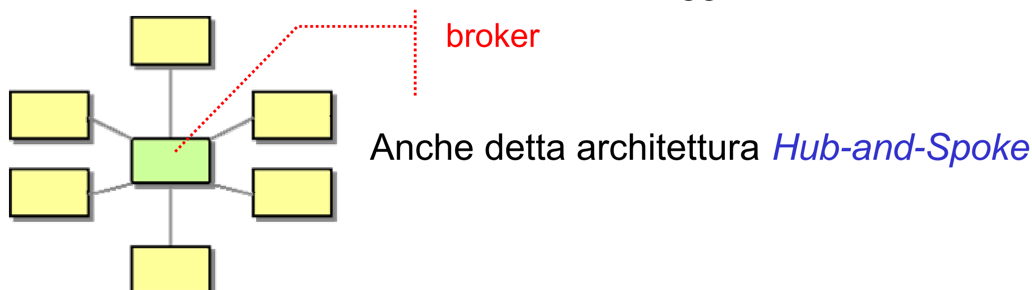
- Il MOM realizza un overlay network con topologia propria e distinta
  - Occorre un servizio di routing
    - Una sottorete connessa di router conosce la topologia della rete logica (statica) e si occupa di far pervenire il messaggio dalla coda del mittente a quella del destinatario
    - Topologie complesse e variabili (scalabili) richiedono la gestione dinamica delle corrispondenze coda-posizione di rete (analogia con routing in reti IP)

Message broker

- L'area di applicazione più importante dei MOM è l'integrazione di applicazioni in ambito aziendale (Enterprise Application Integration, EAI)
  - Le applicazioni devono essere in grado di interpretare il formato dei messaggi dei messaggi (struttura e rappresentazione dei dati)
- Soluzioni possibili per gestire l'eterogeneità dei messaggi (2 su 3 già esaminate):
  - Ogni destinatario è in grado di comprendere ogni formato
  - Formato comune dei messaggi
  - Gateway di livello applicativo (*broker*) in grado di effettuare le conversioni tra formati diversi
- La soluzione generalmente adottata nei MOM si basa sulla presenza di messagge broker

Message broker (2)

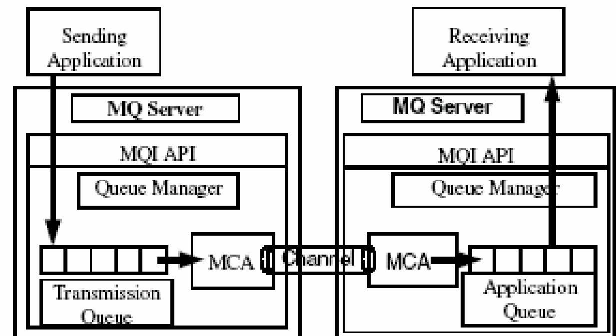
- Message broker: componente (centralizzato) che si prende cura dell'eterogeneità delle applicazioni
  - Trasforma i messaggi in ingresso nel formato adatto all'applicazione destinataria, fornendo trasparenza di accesso ai messaggi
  - Gestisce un repository delle regole e dei programmi che consentono la conversione dei messaggi



- Esempi di MOM con broker
  - IBM WebSphere MQ
  - Microsoft Message Queueing (MSMQ)
  - Sun Java Message Service (JMS)

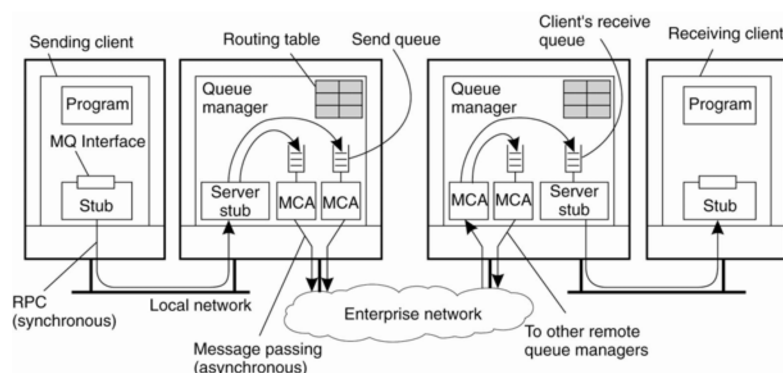
IBM WebSphere MQ

- MOM molto diffuso (~40% del mercato) e supportato
- I messaggi sono gestiti da queue manager (QM)
  - Le applicazioni possono inserire/estrarre messaggi solo nelle/dalle code locali o attraverso un meccanismo RPC
- Il trasferimento dei messaggi da una coda all'altra di QM diversi avviene tramite canali di comunicazione unidirezionali ed affidabili gestiti da message channel agent (MCA) che si occupano di tutti i i dettagli
  - Stabilire canali usando gli strumenti messi a disposizione dai protocolli di rete
  - Tipo di messaggi
  - Invio/ricezione dei pacchetti
- Ogni canale ha una coda di invio



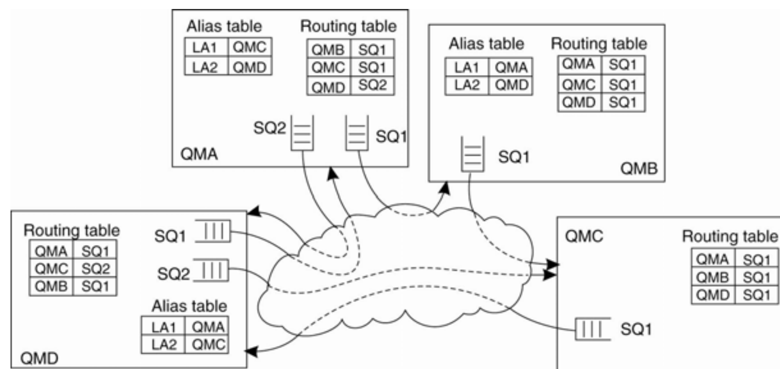
IBM WebSphere MQ (2)

- Il trasferimento sul canale può avvenire solo se entrambi gli MCA sono attivi
  - Il sistema MQ fornisce meccanismi per avviare automaticamente un MCA
- Il routing è sotto il controllo di una gestione di sistema sempre statica e non flessibile
  - L'amministratore stabilisce le opportune interconnessioni tra QM con tabelle di routing all'atto della configurazione



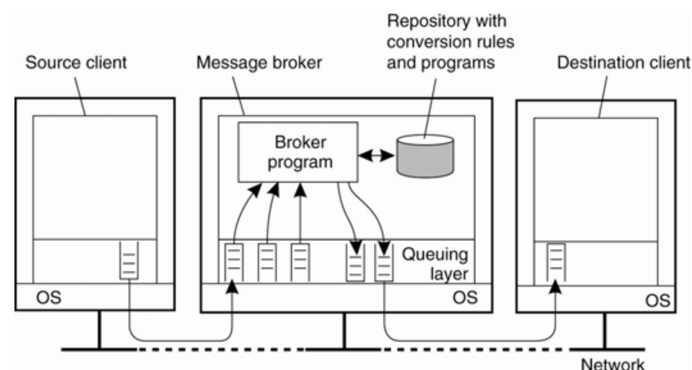
IBM WebSphere MQ (3)

- Indirizzo di destinazione: nome del gestore di destinazione + nome della coda di destinazione
- Elemento della tabella di routing: <destQM, sendQ>
  - destQM: nome del gestore di destinazione
  - sendQ: nome della coda d’invio locale
- Quali sono i problemi del routing nel sistema MQ?

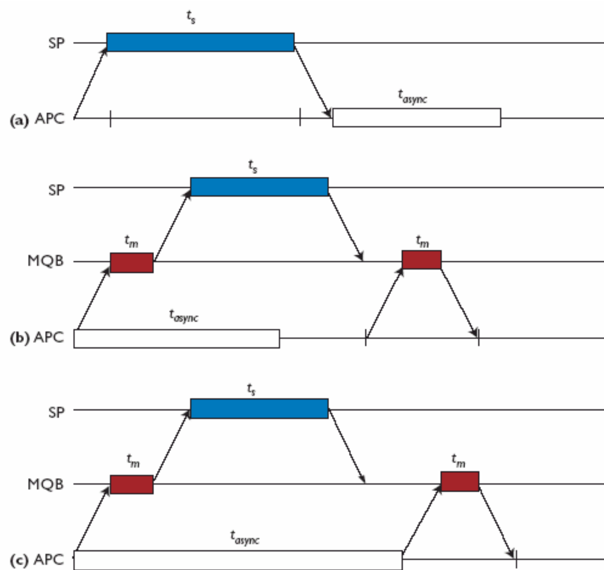


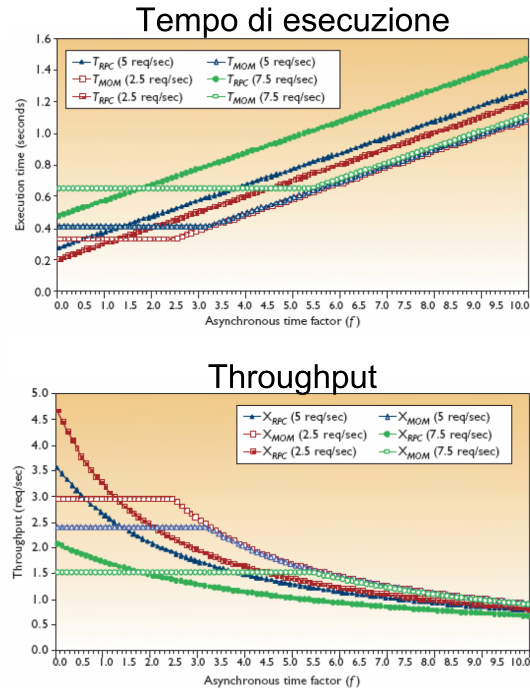
IBM WebSphere MQ (4)

- Per favorire l’integrazione di applicazioni, MQ Broker può intervenire sui messaggi
  - Trasformando formati
  - Organizzando routing in base al contenuto
  - Lavorando su informazioni di applicazione, per specificare sequenze di azioni



Confronto quantitativo tra RPC e MOM





Riferimento: D. A. Menascé, "MOM vs. RPC: Communication models for distributed applications", *IEEE Internet Computing*, Vol. 9, No. 2, pp. 90-93, 2005.

Comunicazione orientata agli stream

- I tipi di comunicazione analizzati finora sono basati su uno scambio di unità di informazioni discreto, ovvero indipendente dal tempo
  - Le relazioni temporali tra i dati non sono fondamentali per l'interpretazione dei dati
- Nei media continui i valori sono dipendenti dal tempo
  - Audio, video, animazioni, dati di sensori (temperatura, pressione, …)
- Data stream: una sequenza (flusso) di unità di dati dipendenti dal tempo
  - Ad es. stream audio, stream video
  - Requisiti temporali espressi come requisiti di qualità del servizio (Quality of Service, QoS)

Comunicazione orientata agli stream (2)

- L'invio di data stream è facilitato dal mezzo trasmissivo a rappresentazione continua, ma non ne è dipendente
  - Ad es. le pipe UNIX e le connessioni TCP/IP forniscono un mezzo trasmissivo a rappresentazione discreta (orientata a [gruppi di] byte)
- Vincoli temporali sulla modalità trasmissiva dei dati
  - Asincrona
    - Preserva l'ordinamento, non la distanza temporale tra unità dati in uno stream
  - Sincrona
    - Preserva l'ordinamento tra unità dati e garantisce un tempo massimo di trasmissione per ogni unità dati
  - Isocrona
    - Aggiunge rispetto alla modalità sincrona la garanzia di un tempo minimo di trasmissione → bounded (delay) jitter

Stream

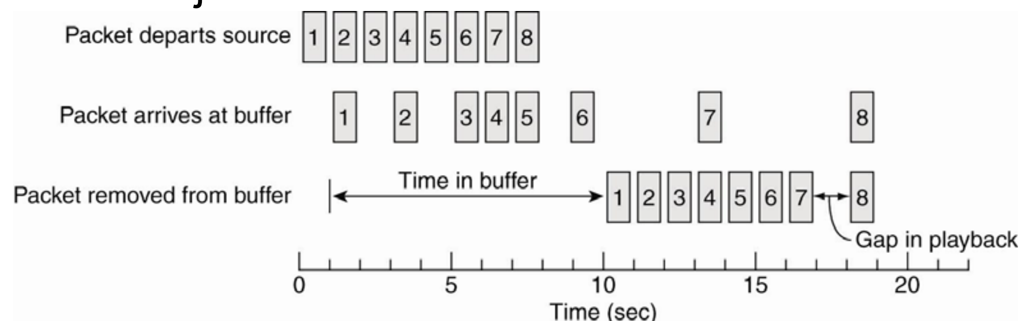
- Definizione: consideriamo solo stream di dati continui che usano la trasmissione isocrona
- Alcune caratteristiche comuni degli stream
  - Gli stream sono unidirezionali
  - Generalmente c'è un'unica sorgente ed una o più destinazioni
- Gli stream possono essere semplici o compositi
- Stream semplici
  - Una semplice sequenza di dati
- Stream compositi
  - Internamente strutturati e composti da stream semplici (*substream*), con requisiti temporali tra i substream che li compongono
  - Ad es. film con 1 substream video e 2 substream audio per effetto stereo
  - Problema di sincronizzazione

Stream e QoS

- In uno stream è essenziale che siano preservate le relazioni temporali
- Quali metriche usare per specificare la QoS a livello applicativo?
  - Il bit rate a cui devono essere trasportati i dati
  - Il tempo massimo di set up di una sessione (quando un'applicazione può iniziare ad inviare dati)
  - Il tempo massimo di trasporto (quanto impiega un'unità di dati ad arrivare al destinatario)
  - La varianza massima del tempo di trasmissione (jitter)
  - Il tempo massimo di round-trip

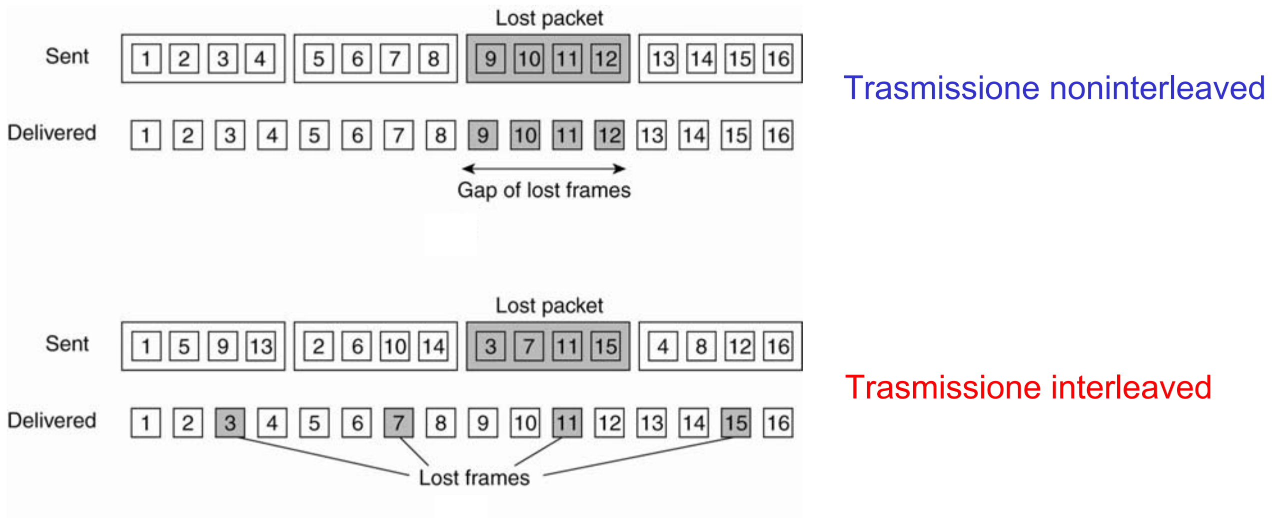
Garantire la QoS

- Come garantire la QoS considerando che lo stack di protocolli Internet si basa su un servizio a datagram e di tipo best-effort?
- Esistono vari meccanismi a livello di rete, come i servizi differenziati mediante i quali alcuni pacchetti possono essere trattati con diversa priorità
  - Ad es. classi di inoltro rapido ed inoltro assicurato
- A livello di SD, diverse tecniche tra cui uso di buffer per ridurre lo jitter



Garantire la QoS (2)

- Come ridurre gli effetti della perdita dei pacchetti (quando più frame sono nello stesso pacchetto)?
  – Usando la tecnica del frame interleaving

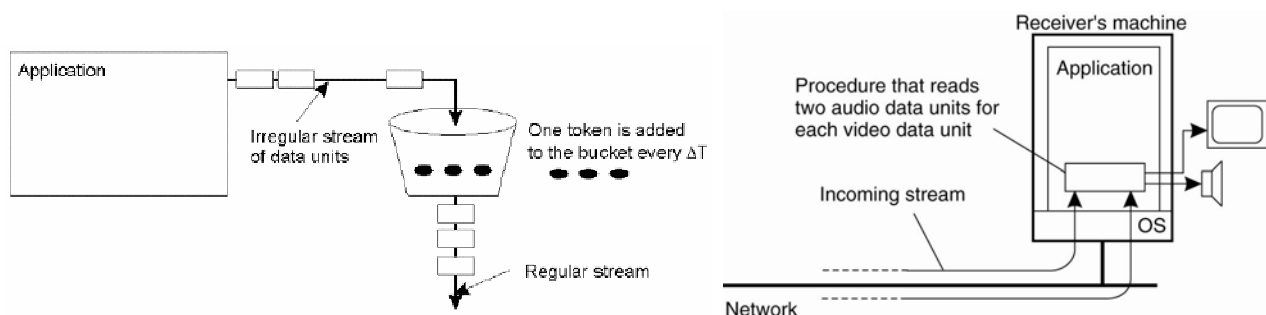


Trasmissione noninterleaved

Trasmissione interleaved

Sincronizzazione degli stream

- Problema: dato uno stream composito, come mantenere la sincronizzazione tra i diversi stream semplici componenti?
  – Ad es. effetto stereo con due canali audio: la differenza di sincronizzazione tra i due substream deve essere inferiore a 20-30 μsec!
- Soluzione possibile: sincronizzazione esplicita
  – Applicazione del principio dell'algoritmo token bucket usato per il filtraggio del traffico

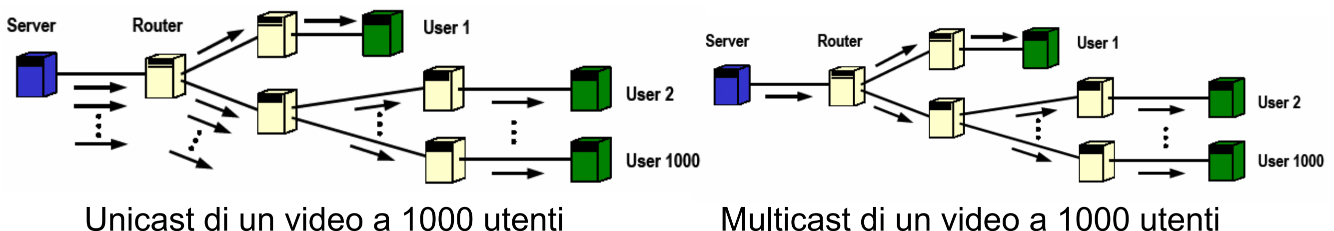


Sincronizzazione degli stream (2)

- Alternativa: multiplexing di tutti i substream in un singolo stream e demultiplexing al lato ricevente
  - Esempio di sincronizzazione di stream: MPEG (Motion Picture Expert Group)
- MPEG è un insieme di algoritmi standard per la compressione di video e audio
- Per combinare in un singolo stream un insieme illimitato di stream distinti sia discreti che continui
  - Ciascuno stream originario viene trasformato in un flusso di unità dati (frame) la cui sequenza è determinata da un'etichetta temporale (timestap) generata da un orologio unico con caratteristiche fissate (90 MHz)
  - I pacchetti di ciascuno stream vengono combinati mediante multiplexing in una sequenza composita di pacchetti a lunghezza variabile ma con propria etichetta temporale
  - A destinazione si ricompongono gli stream originari usando l'etichetta temporale per riprodurre la sincronizzazione tra ciascuna unità dati al suo interno

Comunicazione multicast

- Comunicazione multicast: schema di comunicazione in cui i dati sono inviati a ***molteplici*** destinatari
  - La comunicazione broadcast è un caso particolare della multicast, in cui i dati sono spediti a ***tutti*** i destinatari connessi in rete
  - Esempi di applicazioni multicast one-to-many: distribuzione di risorse audio/video, distribuzione di file
  - Esempi di applicazioni multicast many-to-many: servizi di conferenza, giochi multiplayer, simulazioni distribuite interattive



Unicast di un video a 1000 utenti

Multicast di un video a 1000 utenti

Tipologie di multicast

- Multicast a livello di protocolli di rete
  - Multicast a livello IP basato sui gruppi
    - Gruppo: host interessati alla stessa applicazione multicast
    - Indirizzo IP di classe D assegnato al gruppo
  - La replicazione dei pacchetti e il routing sono gestiti dai router
  - Uso limitato nelle applicazioni per la mancanza di uno sviluppo su larga scala (solo circa 5% degli AS supporta il multicast) e per il problema di tener traccia dell'appartenenza ad un gruppo
- Multicast a livello applicativo
  - La replicazione dei pacchetti e il routing sono gestiti dagli end host
  - *Strutturato*: creazione di percorsi di comunicazione espliciti
  - *Non strutturato*: basato su gossip

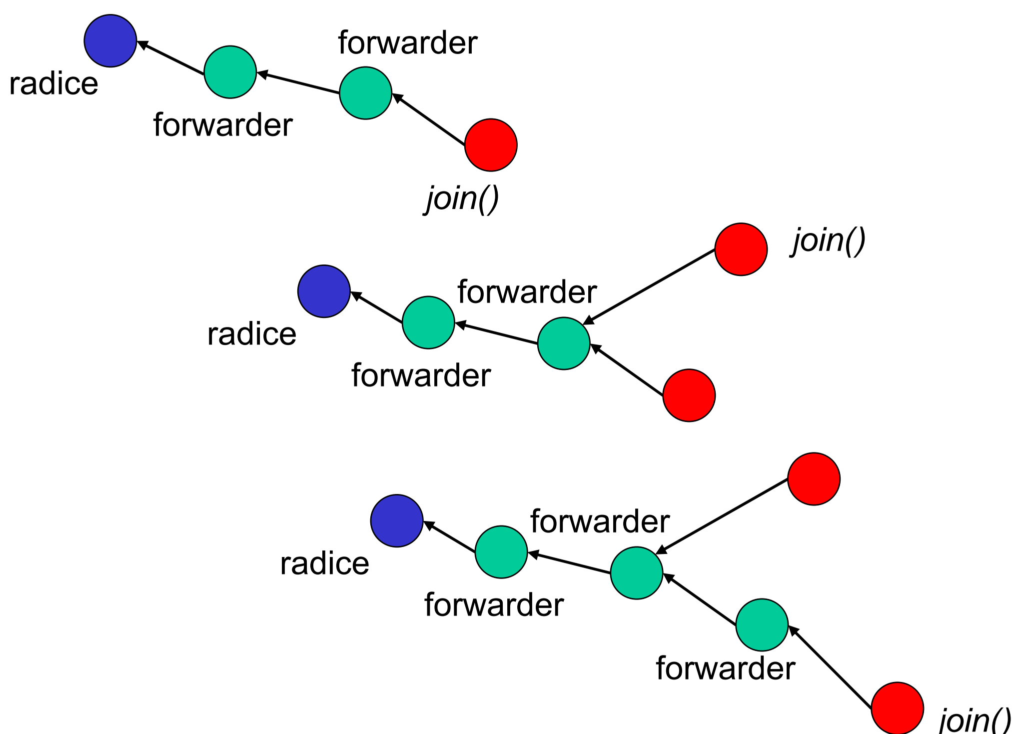
Multicasting applicativo

- Idea di base del multicasting applicativo
  - Organizzare i nodi in una rete overlay
  - Usare tale rete overlay per diffondere le informazioni
- Come costruire la rete overlay?
  - Nodi organizzati in un albero
    - Unico percorso tra ogni coppia di nodi
  - Nodi organizzati in una rete a maglia
    - Molti percorsi tra ogni coppia di nodi

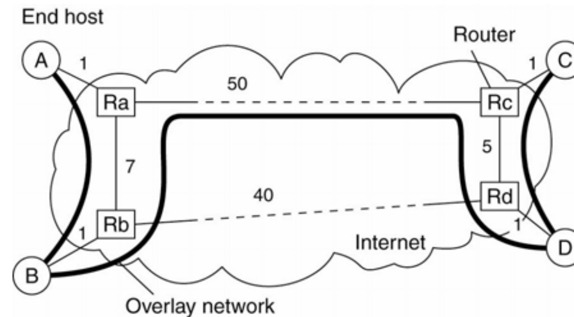
Multicasting applicativo (2)

- Esempio: costruzione di un albero per il multicast applicativo in un sistema P2P basato su Chord
  - Infrastruttura Scribe
  1. Il nodo che inizia la sessione multicast genera un identificatore multicast *groupId*
  2. Cerca il nodo responsabile per la chiave *groupId*
  3. Tale nodo diventa la radice dell'albero di multicast
  4. Se *P* vuole unirsi all'albero, invia una richiesta di join verso la radice
  5. Quando la richiesta arriva a *Q*
     - *Q* non ha mai visto prima una richiesta di join ⇒ *Q* diventa *forwarder*; *P* diventa figlio di *Q*; *Q* continua ad inoltrare la richiesta verso la radice
     - *Q* è già un forwarder per *groupId* ⇒ *P* diventa figlio di *Q*. Non occorre inviare la richiesta di join alla radice
- Come costruire l'albero per il multicast applicativo in modo efficiente?
- Come gestire il guasto di un nodo dell'albero?

Multicasting applicativo (3)



Costi del multicasting applicativo



- Stress sui collegamenti: quante volte un messaggio di multicasting applicativo attraversa lo stesso collegamento fisico?
  - Esempio: il messaggio da *A* a *D* attraversa <*Ra*,*Rb*> due volte
- Stretch: rapporto tra il tempo di trasferimento nell'overlay network e quello nella rete sottostante
  - Esempio: i messaggi da *B* a *C* seguono un percorso con costo 71 a livello applicativo, ma 47 a livello di rete ⇒ stretch=71/47

Protocolli basati su gossip

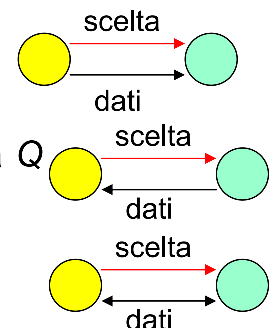
- Protocolli di tipo probabilistico, detti anche epidemici o di gossiping
  - Essendo basati sulla teoria delle epidemie
- Permettono la rapida diffusione delle informazioni attraverso la scelta casuale dei destinatari successivi tra quelli noti al mittente
  - Ogni nodo, per inviare un messaggio, recapita il messaggio ad un sottoinsieme, scelto casualmente, dei nodi nel sistema
  - Ogni nodo che lo riceve ne rinvierà una copia ad un altro sottoinsieme, anch'esso scelto casualmente, e così via
- Caratteristiche della diffusione delle informazioni basata su gossip
  - Scalabilità: ogni nodo invia soltanto un numero limitato di messaggi, indipendentemente dalla dimensione complessiva del sistema
  - Affidabilità: messaggi ridondanti

Principi dei protocolli epidemici

- Lavoro del 1987 di Demers *et al.* sulla garanzia di consistenza nei database replicati
- Idea di base: assumendo che non vi siano conflitti di scrittura
  - Le operazioni di aggiornamento sono eseguite inizialmente su una o alcune repliche
  - Una replica comunica il suo stato aggiornato ad un numero limitato di vicini
  - La propagazione dell'aggiornamento è *lazy* (non immediata)
  - Al termine, ogni aggiornamento dovrebbe raggiungere tutte le repliche
- Anti-entropia: modello di propagazione in cui ciascuna replica sceglie *a caso* un'altra replica e si scambiano gli aggiornamenti, giungendo al termine ad uno stato identico su entrambe
- Gossiping: una replica che è stata appena aggiornata (ossia *infettata*) contatta un certo numero di repliche scelte casualmente inviandogli il proprio aggiornamento (infettandole a loro volta)

Anti-entropia

- Un nodo *P* sceglie casualmente un altro nodo *Q* nel sistema
- Tre strategie di aggiornamento:
  - **Push**: *P* invia soltanto i suoi aggiornamenti a *Q*
  - **Pull**: *P* si prende soltanto i nuovi aggiornamenti da *Q*
  - **Push-Pull**: *P* e *Q* si scambiano reciprocamente gli aggiornamenti (dopodiché possiedono le stesse informazioni)



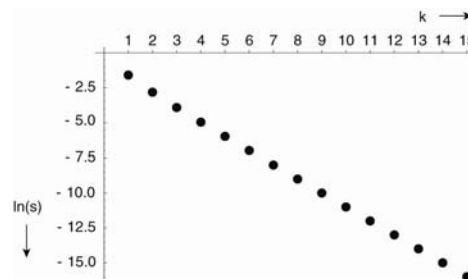
- Osservazione: la strategia push-pull impiega solo O(*log*(*N*)) round per propagare un singolo aggiornamento a tutti gli *N* nodi nel sistema
  - Round: intervallo di tempo in cui ogni nodo ha preso almeno una volta l'iniziativa di scambiare aggiornamenti

Gossiping

- Modello di base: un nodo *P* che è stato appena aggiornato, contatta un certo numero di nodi scelti a caso. Se un nodo contattato ha già ricevuto l'aggiornamento (è già *infetto*), *P* smette con probabilità $1/k$ di contattare altri nodi
- Se *s* è la frazione di nodi che rimangono ignoranti rispetto ad un aggiornamento) si dimostra che

$$s = e^{-(k+1)(1-s)}$$

Esempio: k = 4 → ln(s) = -4,97 → s < 0,7%



- Per garantire che tutti i nodi siano aggiornati, occorre combinare il gossiping puro con l'anti-entropia

Un protocollo di gossiping in generale

- Due peer P e Q, con P che ha scelto Q per lo scambio di dati

Active thread (peer P):		Passive thread (peer Q):
(1) selectPeer(&Q);		(1)
(2) selectToSend(&bufs);		(2)
(3) sendTo(Q, bufs);	----->	(3) receiveFromAny(&P, &bufr);
(4)		(4) selectToSend(&bufs);
(5) receiveFrom(Q, &bufr);	<-----	(5) sendTo(P, bufs);
(6) selectToKeep(cache, bufr);		(6) selectToKeep(cache, bufr);
(7) processData(cache);		(7) processData(cache)

- Quali sono gli aspetti cruciali?
  - La selezione dei peer
  - La selezione dei dati scambiati
  - Il processamento dei dati ricevuti

Riferimento: A.-M. Kermarrec, M. van Steen, "Gossiping in Distributed Systems", *ACM Operating System Review* 41(5), Oct. 2007.

Applicazioni dei protocolli di gossiping nei SD

- Diffusione dell'informazione
  - E' l'applicazione classica e più popolare
- Peer sampling
  - Per la creazione di un gruppo di peer
- Monitoring di nodi e risorse in sistemi distribuiti a larga scala
- Computazioni
  - Di valori aggregati (ad es. media, massimo, minimo)
  - Ad es. nel caso di media
    - v_i e v_j valori iniziali posseduti dai nodi *i* e *j*
    - Dopo il gossiping tra *i* e *j*: $v_i, v_j \leftarrow (v_i + v_j)/2$