JGroups

COMUNICAZIONE DI GRUPPO

Panoramica

JGroups è un framework che offre servizi di comunicazione di gruppo (jgroups.org) fondamenta dell'application server JBoss (jboss.org)

Servizi di comunicazione:

- Consegna garantita
- Gestione dei frammenti
- Ordinamento
- Consegna atomica

	Inaffidabile	Affidabile
unicast	UDP	TCP
multicast	IP-Multicast	**JGroups**

Servizi di gruppo:

- Stato condiviso
- Gestione membri
- Notifica di ingresso, uscita, fallimento dei membri

Lo stack JGroups

JGroups ha un'architettura a stack

È possibile aggiungere strati (protocolli) secondo necessità

Molti sono già presenti nella libreria

- Diversi protocolli di trasporto (UDP IP-Multicast, TCP, …)
- Affidabilità
- Ordinamento
- Compressione
- Crittografia
- Gestione del flusso (per bilanciare consumatori lenti e produttori veloci)
- Gestione dei fallimenti/partizionamenti

Ma è possibile utilizzarne di terze parti (o propri)

Architettura di gruppo

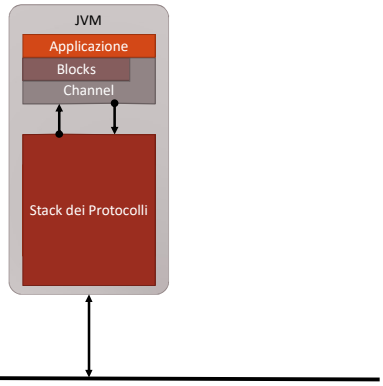
Membri di un gruppo (cluster) sono pari (ma con ruoli differenti)

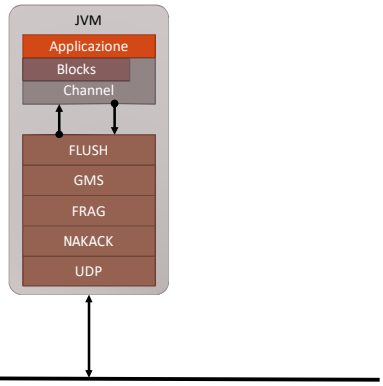
Eleggono un coordinatore (solitamente il più anziano)

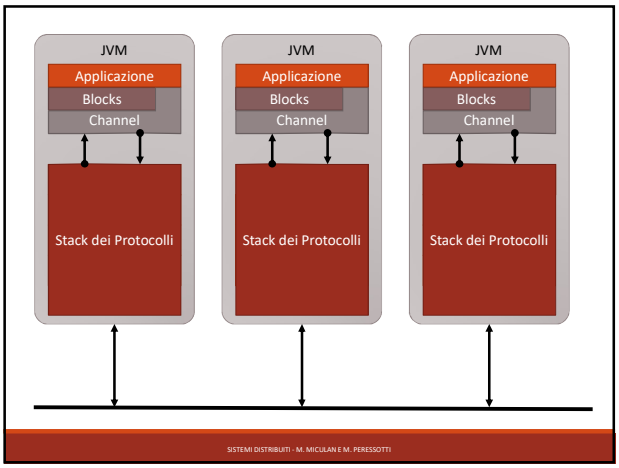
Il coordinatore
- Tiene traccia dei membri del gruppo
- Mantiene lo stato condiviso (e gestisce la replicazione)
- Gestisce la comunicazione di gruppo
  - Ordinamento totale (un nodo chiede al coordinatore di distribuire un messaggio)
  - Garanzia di consegna (ack/retry…)

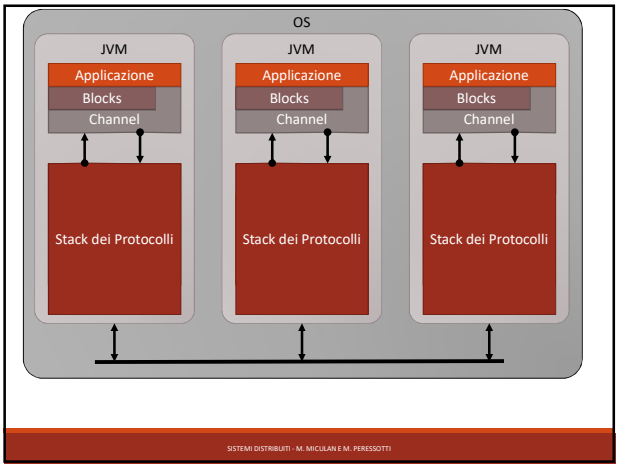
Fallback secondo «anzianità» nel gruppo

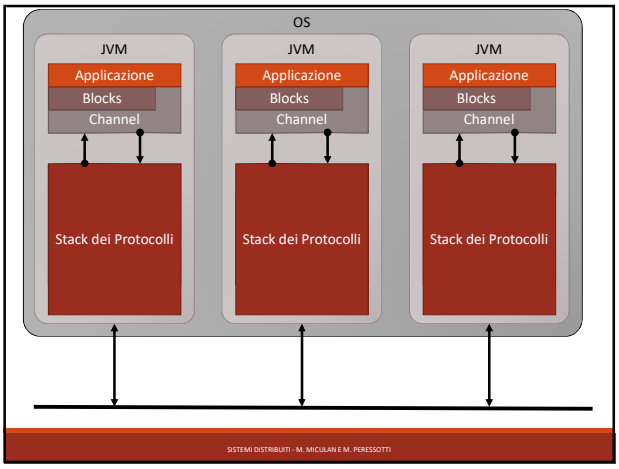
(almeno nella configurazione «base»…vedremo come rendere le cose più interessanti)

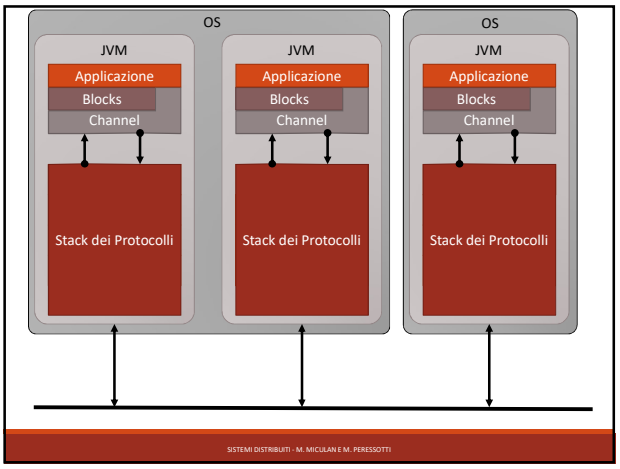


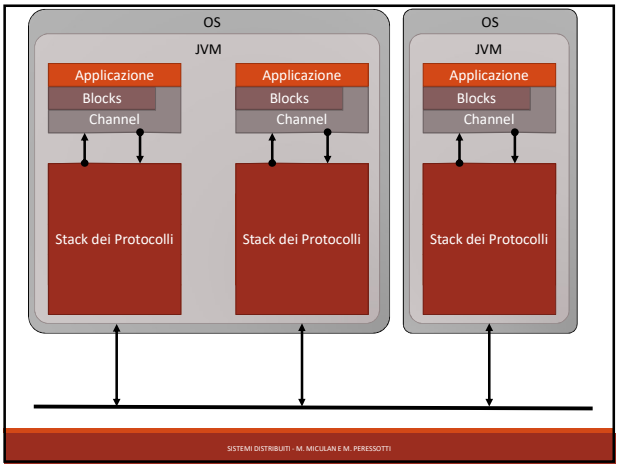


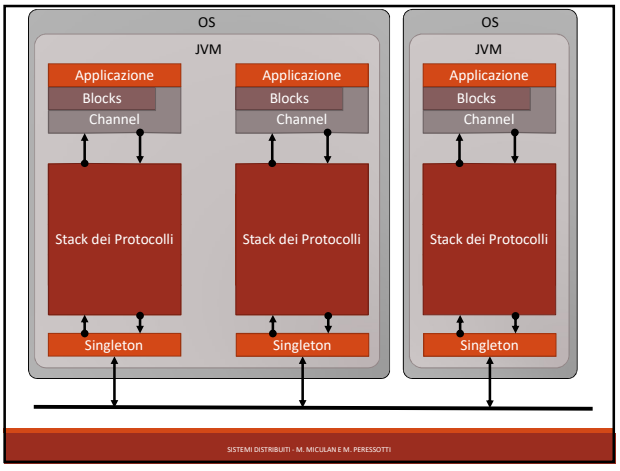

JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
SISTEMI DISTRIBUITI - M. MICULAN E M. PERESSOTTI

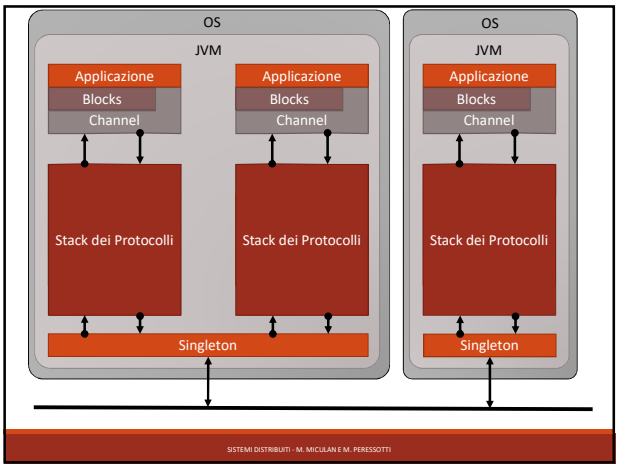

OS
JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
SISTEMI DISTRIBUITI - M. MICULAN E M. PERESSOTTI

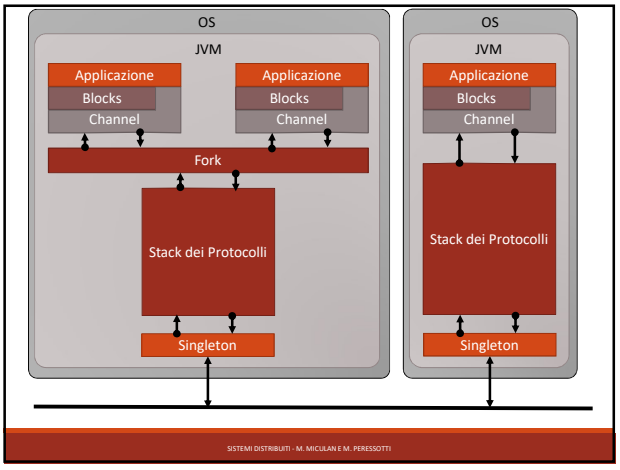

OS
JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
SISTEMI DISTRIBUITI - M. MICULAN E M. PERESSOTTI


OS
JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
OS
JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
SISTEMI DISTRIBUITI - M. MICULAN E M. PERESSOTTI


OS
JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
Applicazione
Blocks
Channel
Stack dei Protocolli
OS
JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
SISTEMI DISTRIBUITI - M. MICULAN E M. PERESSOTTI


OS
JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
Singleton
Applicazione
Blocks
Channel
Stack dei Protocolli
Singleton
OS
JVM
Applicazione
Blocks
Channel
Stack dei Protocolli
Singleton
SISTEMI DISTRIBUITI - M. MICULAN E M. PERESSOTTI





JChannel

La classe JChannel costituisce l'API principale della libreria
- Connessione a un gruppo (cluster)
- Invio/ricezione dei messaggi
- Eventi ed ascoltatori (e.g. ingresso uscita di nodi dal gruppo)

La classe Message definisce i messaggi
- Il payload deve essere serializzabile
- Manca polimorfismo parametrico (generics)

I messaggi possono essere inviati e ricevuti solamente da membri dello stesso gruppo

I membri del gruppo sono noti e reperibili mediante viste (View)

JChannel

Creazione e connessione ad un canale:

```
channel=new JChannel(); //configurazione predefinita udp.xml
channel.connect("NomeGruppo");
```

Invio di un messaggio:

```
channel.send(new Message(
  null,   // destinatario, null -> gruppo
  null,   // indirizzo risposte, null -> "questa" istanza
  "ciao"  // payload
));
```

Disconnessione e chiusura

```
channel.disconnect();
channel.close();
```


Ricezione

La ricezione di messaggi ed eventi è demandata a implementazioni di **Receiver** (e.g. `ReceiverAdapter`)

```
public class Chat extends ReceiverAdapter {
   …
   @Override
   public void receive(Message msg){
       Object payload = msg.getObject();
       …
   }
}
```

Infine, il ricevitore va registrato presso il canale:

```
channel.connect("NomeGruppo");
…
channel.setReceiver(this);
```


Viste e partizionamenti

La lista dei membri di un gruppo è reperibile mediante

```
View view = channel.getView();
```

Le viste sono cristallizzate i.e. non sono aggiornate automaticamente. Gli aggiornamenti sono notificati al receiver mediante la callback:

```
viewAccepted(View newView)
```

I gruppi possono essere partizionati a seguito di fallimenti della rete sottostante (non è possibile distinguere fallimenti di link e nodi)

Le parti evolvono indipendentemente rendendo necessario il merge (se e quando il link sarà ripristinato)

Il merge è demandato all'utente cui è fornita una **MergeView** (i.e. un albero di view)

(Esempio…)

Stato «condiviso»

I Receiver possono esportare/importare uno stato su stream
- getState
- setState

I membri del gruppo possono richiedere lo stato degli altri membri:
`channel.getState(member, timeout);`

Non possono modificarlo senza collaborazione

Lo stato **non è condiviso**, ogni istanza ha la propria «visione»

Uso tipico:
- stato replicato,
- aggiornato alla ricezione dei messaggi
- i nuovi membri copiano lo stato dal coordinatore

(esempio…)

Ascoltatori

JChannel consente di registrare ascoltatori per vari eventi:
- MessageListener
  - void **receive**(Message msg)
  - void **getState**(OutputStream output) throws Exception
  - void **setState**(InputStream input) throws Exception
- MembershipListener
  - void **viewAccepted**(View new_view) cambiamenti nella vista
  - void **suspect**(Object suspected_member) fallimenti sospettati
  - void **block**() notifica la sospensione dell'invio di messaggi (e.g. FLUSH e STATE_TRANSFER) breve!
  - void **unblock**() ripristino invio di messaggi

ReceiverAdapter implementa Receiver estende questi ascoltatori

Attenzione alle dipendenze tra questi eventi o si rischia di bloccare lo stack (e.g. invio/ricezione di messaggi durante viewAccepted in presenza di FLUSH)

Indirizzi

Gli indirizzi JGroups sono UUIDs

Nasconde gli indirizzi dello stack sottostante

È possibile personalizzare gli indirizzi:
1. Implementare `org.jgroups.stack.AddressGenerator`
2. Registrazione `Jchannel.setAddressGenerator(AddressGenerator)`

Alcuni accorgimenti:
- Gli indirizzi devono implementare Address
- Gli indirizzi devono estendere UUID
- Non sovrascrivere compare(), equals() e hashCode
- Registrarazione (della classe, non dell'istanze)
  - ClassConfigurator.getInstance().put(1234, AwesomeAddress.class)
  - jg-magic-map.xml

Indirizzi

Perché provvedere indirizzi diversi?
- (perché possiamo)
- UUID sono unici ma non riflettono e.g. strutture amministrative
- UUID sono poco «leggibili» (mnemonici, significativi, …)

Perché provvedere generatori diversi?
- Aggiunta di informazioni
- Controllo sull'assegnazione degli indirizzi

Alternative (o basi per *`TheChefsChoiceOfTheDayAddress`*) in JGroups:
- AdditionalDataUUID: campo binario
- PayloadUUID: campo testuale
- TopologyUUID: campi sito, rack, macchina

Stack dei protocolli

JGroups ha uno stack di protocolli altamente configurabile.

La classe `ProtocolStack` consente di manipolare lo stack:
- `get/find/add/insert/remove/replaceProtocol`
- `printProtocolSpec`
- `dumpStats`
- `down/up(Event)`

Il metodo `channel.getProtocolStack()` restituisce lo stack del canale.

(esempi…)

Configurare lo stack: java

```
JChannel channel=new JChannel(false);
ProtocolStack stack=new ProtocolStack();
channel.setProtocolStack(stack);
stack.addProtocol(new UDP())
     .addProtocol(new PING())
     .addProtocol(new MERGE2())
     .addProtocol(new FD_SOCK())
     .addProtocol(new FD_ALL()
          .setValue("timeout", 12000)
          .setValue("interval", 3000))
     .addProtocol(new VERIFY_SUSPECT())
     .addProtocol(new BARRIER())
     .addProtocol(new NAKACK())
     .addProtocol(new UNICAST2())
     .addProtocol(new STABLE())
     .addProtocol(new GMS())
     .addProtocol(new UFC())
     .addProtocol(new MFC())
     .addProtocol(new FRAG2());
stack.init();
```

Configurare lo stack: xml

```
<config xmlns="urn:org:jgroups"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:org:jgroups
http://www.jgroups.org/schema/JGroups-3.4.xsd">
    <UDP … />
    <PING … />
    <MERGE2 … />
    <FD_SOCK/>
    <FD_ALL "timeout"=12000 "interval"=3000/>
    <VERIFY_SUSPECT … />
    <BARRIER />
    <pbcast.NAKACK … />
    <UNICAST2 … />
    <pbcast.STABLE …/>
    <pbcast.GMS …/>
    <UFC …/>
    <MFC …/>
    <FRAG2 … />
</config>
```


Configurazioni ed estensioni

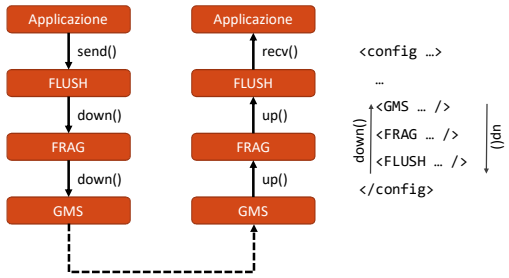
Molte configurazioni sono distribuite assieme al pacchetto JGroups
- E.g. udp.xml, tcp.xml

Molti (ma proprio molti) protocolli e servizi in `org.jgroups.protocols`
- (www.jgroups.org/manual/html/protlist.html)

Se questi non dovessero bastare, le terze parti sono supportate

Scrivere un protocollo:
1. Estendere `org.jgroups.protocols.Protocol` (o sottoclassi)
2. Fornire un nome univoco
3. Implementare i metodi up( ) e down( )
4. (Eventuali header devono avere un nome univoco)

Up(side)-down




```
<config …>
  …
  <GMS … />
  <FRAG … />
  <FLUSH … />
</config>
```

down() up()

Esempio: THROTTLER

```
public class THROTTLER extends Protocol{
    public Object down(final Event evt) {
       if (isMessage(evt) & coinFlip())
           return null;
       else
           return down_prot.down(evt);
    }

    public Object up(final Event evt) {
        if (isMessage(evt) & coinFlip())
           return null;
        else
           return up_prot.up(evt);
    }
}
```


Header

I messaggi possono essere decorati da header

Veicolano informazione usata dai protocolli

È buona educazione **non toccare** gli header altrui…

…ma definire i propri (salvo casi marginali)

1. Estendere **Header** (almeno un costruttore vuoto)
2. Implementare **Streamable** (marshaling)
3. Nome univoco (e.g. annidarlo nella classe del protocollo)
4. Implementare **size()** (byte prodotti durante il marshaling)
5. Registrarazione (java o xml)
   - ClassConfigurator.getInstance().put(1234, AwesomeHeader.class)
   - jg-magic-map.xml

Message espone `putHeader()`, `getHeader()` ~~`removeHeader()`~~

Giusto per farsi un'idea

Transport
- UDP, TCP, TUNNEL

Membership discovery
- PING, TCPPING, TCPGOSSIP, MPING, FILE_PING, JDBC_PING, BPING, RACKSPACE_PING, S3_PING, GOOGLE_PING, AWS_PING, PDC

Partition recovery
- MERGE, MERGE2,MERGE3

Failure detection
- FD, FD_ALL, FD_ALL2, FD_SOCK, FD_PING, FD_HOST, VERIFY_SUSPECT

Reliable message transimission
- NAKACK, NAKNACK2, UNICAST, UNICAST2, UNICAST3, RSVP

Message stability
- STABLE

Group membership
- GMS

Flow control
- FC, MFC, UFC

Fragmentation
- FRAG, FRAG2

Group ordering
- SEQUENCER, TOA

State transfer
- STATE_TRANSFER, STATE, STATE_LOCK, BARRIER, FLUSH

Misc
- STAT, ENCRYPT, AUTH, COMPRESS, SCOPE, RELAY, RELAY2, DAISYCHAIN, RATE_LIMITER, CENTRAL_LOCK, PEER_LOCK, COUNTER, CENTRAL_EXECUTOR, SUPERVISOR, FORK

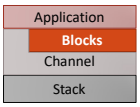
Esercizi (spunti)

1. Servizi replicati: client/server con server replicato
   1. Interazione client/server a scelta e.g. socket, RPC, RMI, REST etc.
   2. Diversificare i protocolli (server RMI, server REST etc.)
   3. Server «attivabili»
2. Cluster di computazione basato su task
   1. Task sono sottoposti al cluster
   2. I task sono distribuiti internamente ai membri del cluster
   3. Assunzione: i task terminano
   4. Ogni task sottoposto dovrà essere completato
3. Aggiungere priorità, tipologie, time-slot, dipendenze tra (sub)task
4. Gestione partizionamenti/merge in presenza di stato
5. Scope e bridge tra clusters (RELAY and SCOPE)
6. Giocare con lo stack (e.g. negoziare QoS)

Blocchi

Formano uno strato aggiuntivo tra applicazione e canale

Offrono
- astrazione sui canali (che sono costrutti **socket-like** asincroni)
- Interfacce più articolate
- Servizi aggiuntivi (e.g. Stato, RPC)



Alcuni esempi
- Message dispatcher
- Rpc dipatcher
- Replicated hashmap
- Replicated cache
- Cluster-wide lock/counter/task/transaction

MessageDispatcher

Correlazione bloccante (e non) tra richiesta e risposta

Diverse tipologie di richiesta:
- GET_ALL
- GET_NONE
- GET_FIRST
- GET_MAJORITY

Esempio: richiesta di voto, servizio

Ulteriori opzioni:
- Timeout
- Unicast
- Filtri (e.g. invalida la richiesta di voto se c'è un veto)
- Exclusion list
- Scope (nel caso sia abilitato l'invio concorrente, cf protocollo SCOPE)
- Flags (gli stessi dei messaggi)

MessageDispatcher

Creazione di un dipatecher

- `MessageDispatcher disp = new MessageDispatcher(channel);`

Ascoltatore per richieste al dispatcher:

-
```
disp.setRequestHandler(new RequestHandler(){
    public Object handle(Message msg) throws Exception {
      …
    }});
```

Invio di richieste:

- castMessage(…)
- castMessageWithFuture(…)
- sendMessage(…)
- sendMessage(…)

(Esempio: Poll)

Esercizi (spunti)

7. Utilizzare i blocchi nei casi precedenti
   - (Non tutti consentono di lavorare con il canale sottostante, FORK?)
   - Voto nella chat
   - Risoluzione merge per votazione (dei processi)
   - Repliche con consistenza debole (letture e scritture per voto)
   - (La votazione è uno spunto, ci sono tanti altri problemi interessanti…)
   - Gestione dei lock (e.g. scrittura di risorse replicate)