Università degli Studi di Padova

FACOLTÀ DI INGEGNERIA

Dipartimento di Ingegneria dell'Informazione

CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

Progetto e gestione di un sistema P2P basato su agenti mobili per il content delivery in WAN

Relatore: Chiar.mo Prof. Carlo Ferrari
Correlatore: Ing. Nicola Pagliarulo

Laureando: Antonio Quercia

Anno Accademico 2009-2010

Università degli Studi di Padova

FACOLTÀ DI INGEGNERIA

Dipartimento di Ingegneria dell'Informazione

Progetto e gestione di un sistema P2P basato su agenti mobili per il content delivery in WAN

Tesi di laurea

Laureando: *Antonio Quercia*

Relatore: Chiar.mo Prof. Carlo Ferrari

Correlatore: Ing. Nicola Pagliarulo

Corso di laurea Specialistica in Ingegneria Informatica

Anno Accademico 2009-2010

Indice

Introduzione

Con questa tesi si vuole documentare il percorso che ha portato alla realizzazione di nuove funzionalità per un sistema che si occupa della trasmissione di contenuti multimediali su rete P2P attraverso l'utilizzo di agenti mobili.

Progetto complesso e multiforme, quello presentato costituisce il terzo tassello di un mosaico foggiato in due fasi elaborate in sequenza: la prima si è conclusa con la creazione di un solido nucleo funzionante, mentre la seconda si è concentrata sulla selezione delle fonti per i contenuti da trasmettere.

L'attività svolta, su cui è imperniata tutta la trattazione, ha pervaso gran parte dell'impianto strutturale del sistema esistente, consentendone il funzionamento attraverso Internet, permettendone l'amministrazione remota a livello dei singoli nodi e prevedendo i primi accorgimenti per la rilevazione e la gestione di alcuni tipi di guasti.

Il lavoro che ci si appresta a descrivere è stato sostenuto dal desiderio di approfondire la conoscenza sia sulle reti P2P che sull'utilizzo degli agenti mobili e dalla volontà di ottenere un prodotto progettualmente coerente ed effettivamente funzionante.

All'interno dei diversi capitoli componenti l'elaborato, si potrà osservare da un'ampia prospettiva l'analisi del progetto complessivo, la descrizione del contesto in cui si colloca e l'esame dei requisiti (Capitolo 1), per poi passare ad una discussione molto più approfondita delle tematiche strettamente connesse alle attività di progettazione e di realizzazione: dalla comunicazione inter-regione attraverso Internet fra i vari nodi del sistema (Capitolo 2), all'elaborazione ed alla gestione delle primitive di stato e di amministrazione (Capitolo 3); dalla definizione delle modalità atte a rilevare ed a gestire alcune categorie di guasti (Capitolo 4), fino alla sintesi delle scelte realizzative che hanno caratterizzato l'implementazione definitiva del sistema, il tutto corredato da alcune considerazioni sul collaudo e sugli sviluppi futuri (Capitolo 5).

Il prodotto di questo lavoro si è rivelato molto fecondo, sia in termini di risultati, sia in termini di soddisfazione personale.

CAPITOLO 1

Analisi del sistema e delle esigenze

In questo capitolo viene introdotto lo scenario in cui si colloca il lavoro di tesi che verrà documentato nel corso di questa trattazione. Ad una preventiva panoramica sulla situazione attuale segue l'analisi delle esigenze del nuovo sistema. Il capitolo termina delineando alcune linee guida per l'attività di progettazione.

1.1 Introduzione

Il fine che si intende perseguire mentre ci si appresta a redigere questo elaborato è quello di documentare le attività di analisi, progettazione e realizzazione necessarie per perfezionare un applicativo aziendale esistente.

Risultato di uno sviluppo articolato in due fasi distinte, l'attuale sistema nasce dalla necessità di sostituire MediaSpreader (Sysdata Italia S.p.a), un prodotto già funzionante che presenta alcune limitazioni in ambito architetturale. Questo software si occupa della trasmissione di contenuti multimediali e della relativa visualizzazione su varie postazioni video. Un operatore, previa creazione di un palinsesto, è in grado di controllare la tipologia dei contenuti e la scadenza temporale per la riproduzione degli stessi, mentre la trasmissione vera e propria dei contenuti avviene in modo trasparente; tuttavia si basa su un'architettura

client-server che prevede la formulazione delle richieste da parte di ogni client direttamente al server. Questo evidentemente comporta, all'aumentare del numero di client connessi e delle richieste inviate, un incremento dei tempi di risposta del server ed una progressiva riduzione delle prestazioni complessive.

Nonostante la presenza di vari accorgimenti con cui si possono tamponare le criticità introdotte da questo tipo di architettura[1], si è preferito concentrare l'attenzione sull'utilizzo del paradigma P2P e, in particolar modo, sull'utilizzo della programmazione ad agenti software mobili.

Dopo una prima fase di studio e progettazione[2] che ha sondato lo stato dell'arte per concretizzare una struttura versatile e funzionale, ecco nascere la prima versione del prototipo che unisce i vantaggi offerti da un'architettura P2P (tra cui scalabilità e flessibilità) a quelli propri degli agenti mobili[3] (come la presenza di un delineato impianto strutturale, la possibilità di avvalersi di una programmazione modulare affinchè gli agenti cooperino per raggiungere uno scopo pur mantenendo un certo grado di indipendenza e la possibilità di usufruire di una comunicazione basata sullo scambio di messaggi).

La seconda fase ha visto un'analisi più approfondita degli algoritmi per la selezione delle fonti[4] ed ha portato ad una prima struttura multi-regione del sistema, in cui più peer possono essere raggruppati all'interno di una stessa regione ed interagire con peer di altre regioni per la trasmissione dei contenuti.

Questo lavoro di tesi intende proseguire lo studio sulla programmazione ad agenti software mobili su rete P2P e, alla luce delle criticità rilevate nel prototipo così com'è stato definito fino a questo momento, introdurre nuove funzionalità sulla base delle considerazioni che verranno proposte durante la trattazione.

---

[1] Si potrebbero citare il potenziamento della connessione del server o l'utilizzo di un cluster di server; tuttavia l'efficacia della soluzione diminuirebbe con un nuovo incremento delle dimensioni del sistema.

[2] Si veda (Bobice, 2009).

[3] Si veda (Bobice, 2009 p. 45-59)

[4] Si veda (Biscuola, 2009 p. 7-66)

1.2 Situazione attuale

Lo stato attuale del prototipo si fonda su di una struttura gerarchica a due livelli: un numero arbitrario di peer interconnessi fra loro compone una regione coordinata da un superpeer; più superpeer fanno parte di una overlay network e realizzano la connessione fra le regioni.

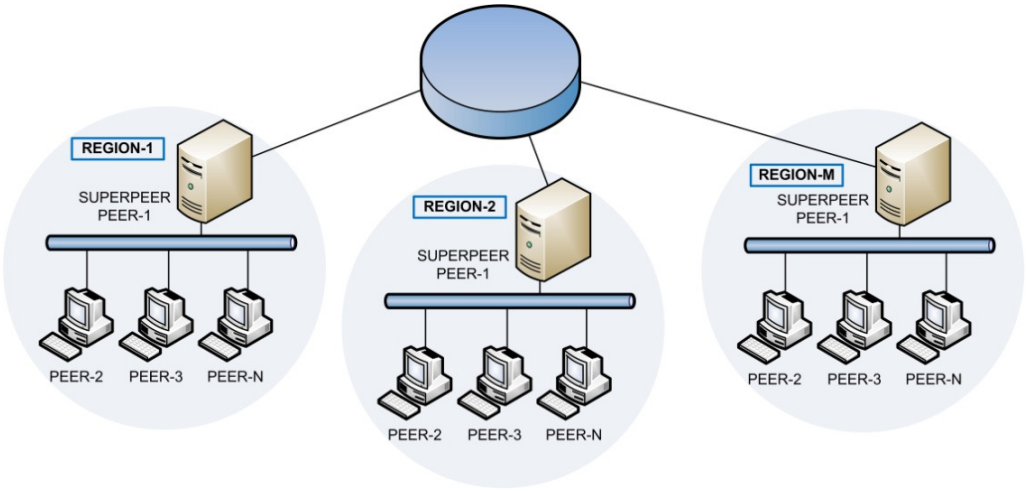


Figura 1-1 – Struttura del prototipo attuale.

Ogni peer possiede un proprio palinsesto che definisce la natura dei contenuti da pubblicare e la scadenza temporale per la visualizzazione. I contenuti sono partizionati in tranche (unità di trasferimento); solo quando il peer ottiene tutte le tranche che costituiscono un contenuto, tale contenuto può essere riprodotto.

Per eseguire la ricerca di una tranche, il peer formula una richiesta indirizzata al superpeer della regione di cui fa parte. Il superpeer, che tiene traccia di tutte le tranche possedute da ogni peer della regione, restituisce una lista di fonti per la particolare tranche richiesta: se la tranche è già presente all'interno della regione, la lista sarà composta da peer facenti parte della regione locale; al contrario, se la tranche non è presente in regione, il superpeer darà inizio ad una ricerca extra-regione che interesserà gli altri superpeer e restituirà, se presenti, una lista di peer di altre regioni.

Quando il peer richiedente avrà effettuato la selezione della fonte da cui prelevare la tranche, avrà luogo il trasferimento.

1.2.1 Agenti software

Le operazioni appena descritte in maniera molto sommaria, vengono svolte dagli agenti software che possono, all'occorrenza, migrare da un peer all'altro.

Un agente software può essere definito come un entità autonoma che esegue determinati compiti - specificati dal programmatore - all'interno di un particolare ambiente di esecuzione; è costituito da:

- *codice*: contiene la logica del programma;
- *dati*: tutte le variabili utilizzate dell'agente
- *stato*: i dati relativi all'esecuzione dell'agente (valore dei registri e delle variabili di istanza, stack, etc).

La piattaforma scelta per la gestione della programmazione ad agenti è JADE (Java Agent DEvelopment Framework)[5]. Si rimanda al paragrafo *1.2.2* per una panoramica sulla struttura e sulle principali funzionalità.

Di seguito si riporta l'elenco di tutti gli agenti utilizzati che possono essere suddivisi nei seguenti macro-componenti (*Figura 1-2*):

- gestione delle risorse (RSMA);
- gestione delle richieste (RQMA);
- ricerche intra/extra-regione (SA, RRA, CA, SearchAgent);
- gestione del trasferimento (DMA, UMA, MA);
- trasferimento (DA, UA);
- statistiche (SMA).

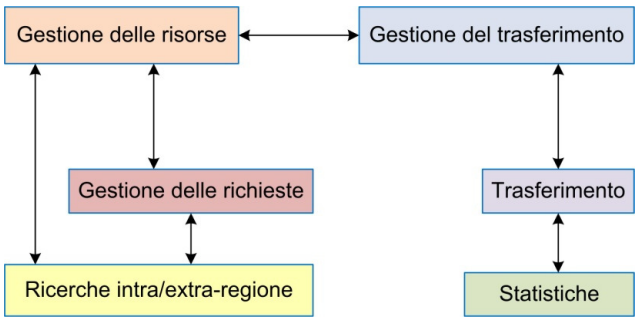


Figura 1-2 – Macro-componenti.

[5] Per maggiori informazioni si veda il sito web (), mentre per alcune considerazioni sul paradigma ad agenti si veda (Bobice, 2009 p. 45-59).

Per ogni agente si riporta una concisa descrizione delle principali funzioni (nella *Figura 1-3*, p.6 viene rappresentata la dislocazione di tali componenti sui nodi del sistema e le interazioni fra di essi):

- *RSMA* (ReSourceManagementAgent): gestisce i contenuti e le relative tranche che il peer deve scaricare, eseguendone l'aggiornamento dello stato;
- *RQMA* (ReQuestManagementAgent): gestisce le richieste (associate ad una tranche) inoltrate al superpeer della regione che ha il compito di soddisfarle;
- *DMA* (DownloaderManagementAgent): gestisce la selezione delle fonti creando diversi MA. Al termine di una fase di ricerca, deve adottare una strategia di selezione del peer da cui scaricare la tranche[6];
- *DA* (DownloaderAgent): gestisce i download delle tranche interfacciandosi con l'UA;
- *UMA* (UploaderManagementAgent): agente che gestisce le richieste di upload provenienti dagli altri peer presenti in regione.
- *UA* (UploaderAgent) agente che esegue l'upload di una tranche verso un particolare peer;
- *MA* (MigrationAgent): creato dal DMA (che ne lancia uno per ogni fonte disponibile) si occupa di migrare verso una delle fonti e di comunicare con il DMA del peer richiedente affinchè possa selezionare la fonte da cui ricevere la tranche;
- *SMA* (StatisticManagementAgent): gestisce le statistiche dei tempi di upload di un peer;
- *SA* (SchedulerAgent): gestisce l'invio delle richieste formulate dal RQMA al superpeer della regione;
- *RRA* (ResourceRegionAgent): risiede nel superpeer di ogni regione e svolge la funzione di indexing server. Mantiene aggiornate le informazioni su chi possiede le tranche all'interno della regione e risponde alle richieste provenienti dai peer fornendo una lista di fonti per la tranche ricercata. Se la tranche non è presente

[6] Si può trovare un'analisi più approfondita del processo di gestione del download al paragrafo 2.2

all'interno della regione, invia una richiesta al CA affinchè svolga una ricerca extra-regione;

- *CA* (CreatorAgent): risiede nel superpeer di ogni regione e gestisce le ricerche extra-regione tramite la crazione degli agenti di ricerca e comunica con gli agenti di ricerca generati dai superpeer di altre regioni (dai rispettivi CA);
- *SearchAgent*: agente di ricerca creato dal CA che percorre l'overlay network alla ricerca di fonti extra-regione[7].

Di seguito (*Figura 1-3*) viene schematizzata la dislocazione degli agenti sulle due tipologie di nodi del sistema: peer e superpeer.

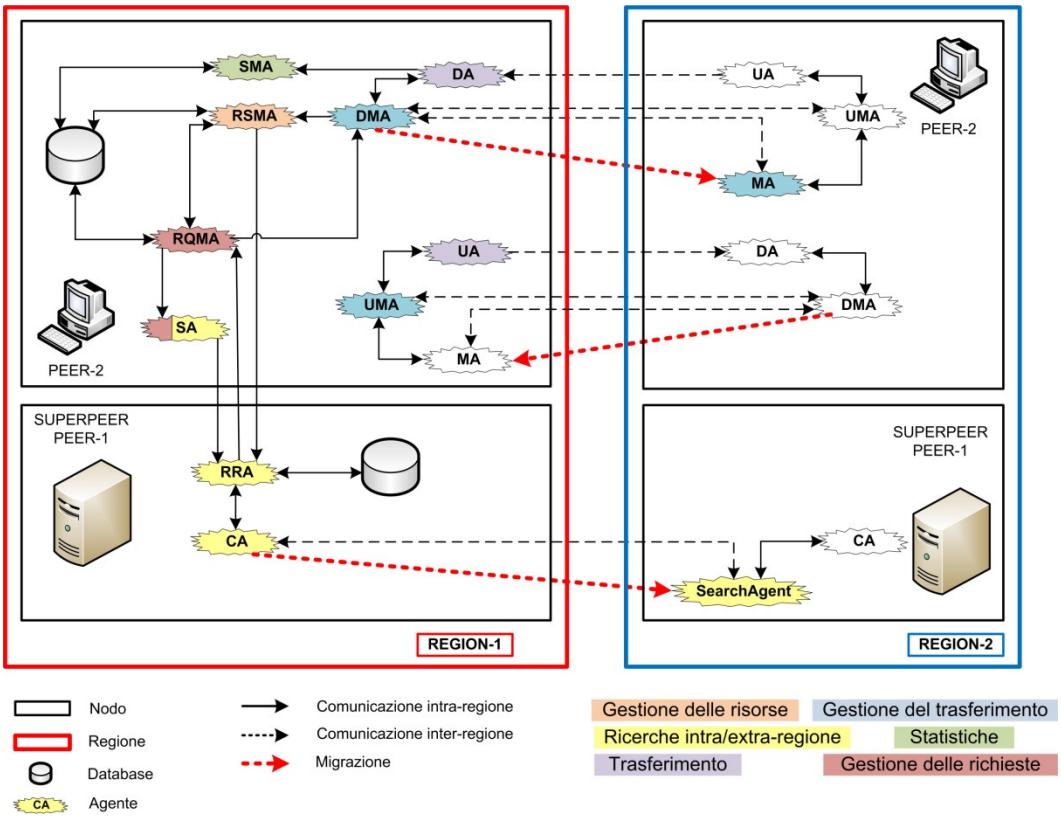


Figura 1-3 – Dislocazione degli agenti sui nodi del sistema attuale.

[7] Per una descrizione più accurata del processo di ricerca extra-regione si faccia riferimento al paragrafo 2.2.

1.2.2 Jade

JADE (Java Agent DEvelopment Framework) è una piattaforma per la realizzazione di sistemi distribuiti multi-agente (MAS - Multi Agent System). Implementato in Java da Telecom Italia Lab, è distribuito sotto licenza LGPL (Library Gnu Public Licence) e offre piena compatibilità con gli standard FIPA[8].

La piattaforma

Una *AgentPlatform* (AP)[9] JADE è composta da diversi container distribuiti sulla rete che forniscono l'ambiente all'interno del quale gli agenti vivono e svolgono la loro attività. Un container speciale, chiamato *main container*, rappresenta il punto di inizializzazione della piattaforma (dove viene creato l'RMI registry utilizzato internamente da JADE) e tutti gli altri container che vogliono entrare a far parte della piattaforma devono effettuare una procedura di registrazione con il main container.

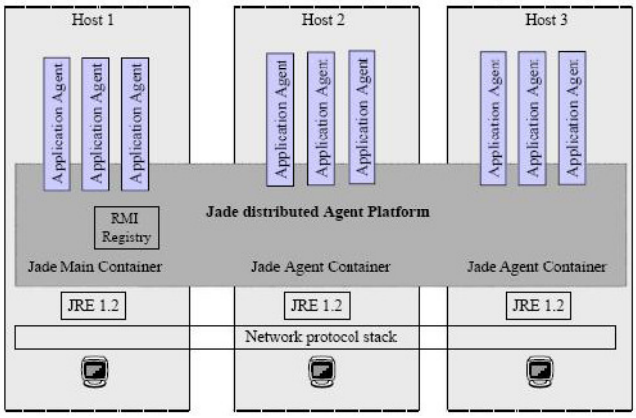


Figura 1-4 - Architettura di JADE

Il main container gestisce la *Container Table* (CT) - il registro degli indirizzi di tutti i container presenti nella piattaforma - e la *Global Agent*

[8] FIPA è un'associazione di compagnie molto prestigiose nel campo dell'informatica che, a partire dall'Ottobre del 1997, ha pubblicato le FIPASpecification, una serie di specifiche il cui obiettivo è la ricerca di soluzioni per la realizzazione di sistemi basati sull'utilizzo di agenti con alto grado di interoperabilità. Queste specifiche forniscono la definizione dell'architettura astratta di una Agent Platform (AP) per l'amministrazione degli agenti e definiscono alcuni parametri fondamentali come quelli relativi alla comunicazione tra agenti.

[9] Nello specifico una *AgentPlatform* (AP) è formata dai computer, dai sistemi operativi, dai componenti FIPA per la gestione degli agenti, dagli agenti stessi e da qualsiasi altro software di supporto. Una singola AP può essere realizzata da più computer e gli agenti non devono necessariamente essere presenti su un unico host.

Description Table (GADT) - il registro di tutti gli agenti presenti nella piattaforma, incluso il loro stato e la loro posizione. Inoltre ospita l'AMS e il DF, due agenti speciali che permettono la gestione degli agenti fornendo rispettivamente un servizio di "pagine bianche" e "pagine gialle"[10].

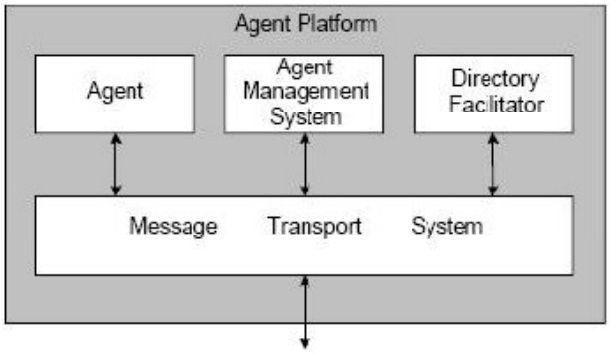


Figura 1-5 – Architettura di una agent platform

Agenti e comunicazioni

All'interno di una piattaforma gli agenti[11] sono identificati univocamente da un *AgentIDentifier (AID)* e svolgono particolari attività definite dagli utenti come particolari oggetti di tipo *Behaviour*. Ad ogni agente possono essere associati diversi behaviour che vengono schedulati in modo trasparente al programmatore secondo una politica cooperativa e non preemptive.

Il paradigma utilizzato per la comunicazione tra agenti è l'*asyncronous message passing* (ogni agente ha una "mailbox" in cui vengono memorizzati i messaggi spediti dagli altri agenti) e i messaggi trasmessi tra gli agenti rispettano le specifiche FIPA ACL[12].

[10] L'AgentManagementSystem (AMS) controlla la creazione, la cancellazione, la sospensione, la riesumazione, l'autenticazione e la migrazione di agenti su altre AgentPlatform e garantisce un servizio di "pagine bianche" per tutti gli agenti residenti sull'AP: ogni agente al momento della sua creazione deve registrarsi all'AMS per poter ottenere l'AID che lo identifica univocamente all'interno della piattaforma.

Il Directory Facilitator (DF) è un agente opzionale di una Agent Platform che gestisce le informazioni sui servizi forniti dai vari agenti della piattaforma ("pagine gialle"). In una AP possono essere presenti più DF organizzati in domini.

[11] L'agente è il processo computazionale identificato univocamente da un AgentIDentifier (AID) che vive nell'Agent Platform e offre uno o più servizi.

[12] Ogni messaggio contiene delle informazioni standardizzate come:

- il mittente (sender) del messaggio
- la lista dei destinatari (receiver)
- l'azione (performative) che indica l'obiettivo che il mittente intende

La comunicazione tra agenti di piattaforme diverse avviene tramite l'indirizzo *MTP (Message Transport Protocol)* di ogni piattaforma (basato di default su HTTP), mentre per lo scambio di messaggi tra agenti all'interno di una stessa piattaforma viene utilizzato un protocollo (non compatibile con lo standard FIPA) chiamato *IMTP (Internal Message Transport Protocol)* basato su Java RMI (di default) o su socket TCP.

L'attesa della ricezione dei messaggi può bloccare l'esecuzione di un behaviour; ogni volta che un messaggio viene ricevuto dall'agente i behaviour bloccati vengono riattivati in modo da poter eventualmente leggere e processare il messaggio, inoltre è possibile richiedere di leggere solo i messaggi che rispettano determinate caratteristiche definendo filtri opportuni.

Migrazione

JADE, essendo implementato in Java, fornisce solo la forma leggera di migrazione (weak migration)[13].

Si possono individuare:

- una migrazione *intra-piattaforma*: un agente migra da un container ad un altro all'interno della stessa piattaforma
- una migrazione *inter-piattaforma*: un agente migra da un container di una piattaforma a quello di una piattaforma diversa

raggiungere (request, inform, propose, etc..)
- il contenuto (content) ossia l'informazione contenuta nel messaggio
- il linguaggio (language) che indica la sintassi utilizzata per rappresentare il contenuto
- l'ontologia (ontology) che indica il vocabolario dei termini utilizzati per rappresentare il contenuto
- altre informazioni usate per controllare conversazioni concorrenti (conversation-id, reply-with, in-reply-to, reply-by)

[13] Per *weak migration* s'intende una migrazione in cui non viene trasferito lo stato di esecuzione dell'agente e quindi l'esecuzione dell'agente ricomincia dall'inizio della procedura che ha invocato la migrazione. Per preservare in parte lo stato è necessario progettare l'agente come una macchina a stati finiti.

La *strong migration*, al contrario, prevede che lo stato di un agente venga "congelato" e che dopo la migrazione la sua esecuzione riprenda esattamente dal punto in cui era stata interrotta (a partire dall'istruzione successiva). Questa tecnica è molto complessa e dipende dall'architettura del sistema dato che richiede il salvataggio dello stato di esecuzione un agente (possibile solo attraverso l'interazione con il sistema operativo).

La migrazione intra-piattaforma è supportata nativamente da JADE attraverso l'*Agent Mobility Service* che utilizza la serializzazione Java per il marshalling degli oggetti Java che rappresentano l'agente (non è possibile la migrazione di file utilizzati dagli agenti dato che i loro descrittori non sono serializzabili).

La migrazione inter-piattaforma può essere effettuata tramite un add-on che implementa l'Inter-Platform-Mobility-Service (IPMS) il quale sfrutta i messaggi ACL come mezzo di trasporto: questi messaggi sono spediti tra gli AMS delle due piattaforme.

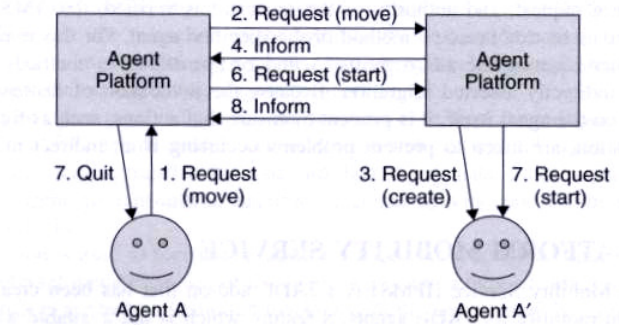


Figura 1-6 - Protocollo FIPA per la migrazione inter-piattaforma.

1.3 Analisi delle esigenze

Dopo aver effettuato una prima analisi in via preventiva della situazione attuale e del prototipo realizzato, la trattazione prosegue con l'indicazione dei limiti e con l'individuazione dei principali vincoli da rispettare; successivamente si stila una lista delle nuove funzionalità di cui si intende dotare il sistema e delle eventuali migliorie da apportare al comportamento dei componenti esistenti.

1.3.1 Limiti del prototipo

Le caratteristiche del prototipo così come è definito nella sua versione attuale lo rendono adatto ad un funzionamento multi-regione all'interno della stessa rete: i componenti presenti svolgono correttamente le loro funzioni di ricerca intra/extra-regione, gestione delle fonti e dei trasferimenti, trasmissione dei contenuti e gestione delle statistiche di download.

Tuttavia, a causa di ovvie scelte progettuali effettuate predecentemente, sono stati tralasciati quegli aspetti che riguardano:

1. la gestione dello stato dei peer e delle regioni e della loro connessione al sistema;
2. il funzionamento del sistema in un contesto più ampio, quale quello delle reti in aree geografiche (WAN);
3. il rilevamento dei guasti o, più nello specifico, degli errori di comunicazione;
4. l'efficienza dei protocolli relativi alla gestione delle richieste;
5. la sicurezza complessiva del sistema e, in particolare, della migrazione degli agenti fra piattaforme differenti.

Finora i limiti dell'ambiente di esecuzione per il sistema composto dalle varie piattaforme JADE hanno coinciso al più con quelli di una stessa Local Area Network.

Dal momento che l'intenzione è quella di estendere tali confini (è appunto in questa direzione che tende principalmente questo lavoro di tesi), è necessario far fronte ad una serie di problematiche che potrebbero comportare un notevole lavoro di upgrade e che porteranno all'elaborazione di un nuovo e più completo modello di riferimento.

Per questo motivo si decide di concentrare le energie sui primi tre punti dell'elenco appena proposto.

1.3.2 Vincoli

Vista la concretezza funzionale delle singole parti componenti il sistema e l'utile impianto strutturale offerto da JADE, si ritiene sensato proseguire il lavoro già cominciato cercando di salvaguardare i risultati ottenuti. Quindi è necessario individuare quali siano i vincoli da rispettare durante la progettazione e l'implementazione delle nuove funzionalità.

Al fine di integrare i nuovi componenti con quelli esistenti, bisogna far riferimento alla stessa base di dati utilizzata finora. Inoltre si deve individuare fino a che livello mantenere valida la struttura degli agenti JADE utilizzati dal prototipo: alcuni di essi sono costituiti da behaviour progettati come macchine a stati finiti, altri come thread autonomi da esegure una sola volta; è necessario, soprattutto se si intende introdurre un meccanismo per la gestione dello stato dei peer (e, di riflesso, degli agenti),

trovare un metodo per evitare di dover stravolgere la struttura dei vari behaviour che compongono gli agenti.

A livello di comunicazioni bisogna salvaguardare i protocolli già realizzati[14]: sia che si tratti di una comunicazione fra peer all'interno della stessa regione, sia che si tratti di una comunicazione fra peer di regioni differenti, è necessario che le modifiche apportate per gestire l'interazione attraverso Internet non vadano ad inficiare l'efficacia delle procedure esistenti.

Per quanto concerne la gestione dello stato dei peer e delle regioni e la gestione della loro connessione al sistema, bisogna far sì che i componenti che si occuperanno di tale gestione si integrino con quelli propri di JADE che creano i container e gli agenti.

1.3.3 Obiettivi

Tra le nuove funzionalità che il sistema dovrà essere in grado di offrire si enumerano:

1. il funzionamento, tra nodi di regioni distinte connesse tramite Internet, delle attività di:
   - gestione della overlay network;
   - ricerca extra-regione;
   - gestione delle fonti e dei trasferimenti;
   - trasmissione delle tranche;
2. l'amministrazione remota dello stato dei peer e delle regioni e della loro connessione al sistema tramite primitive di stato che realizzino, ad esempio, la connessione/disconnessione di un nodo in modo che lo stato degli altri nodi rimanga consistente e che il sistema continui a funzionare correttamente;
3. il rilevamento e la gestione di alcune tipologie di guasto: errori di comunicazione durante la gestione delle fonti e dei trasferimenti,

[14] Una descrizione dei principali protocolli di comunicazione verrà effettuata all'interno del paragrafo 2.2, in cui se ne analizzano i dettagli al fine di formulare un'ipotesi di soluzione al problema della comunicazione inter-regione attraverso Internet.

durante la trasmissione delle tranche e durante i cambiamenti di stato di peer e regioni.

1.3.4 Considerazioni

A questo punto è doveroso considerare, anche se solo in via preliminare, le difficoltà che potrebbero sorgere durante la progettazione e l'implementazione delle nuove funzionalità.

Prime fra tutte quelle legate alle comunicazioni attraverso Internet, che potrebbero essere dovute a problemi di connessione, a ritardi nella trasmissione o a problematiche configurazioni di rete: si dovrà dare particolare rilievo al problema della configurazione di NAT e firewall (se ne discuterà ampiamente nel corso del *CAPITOLO 2*).

In secondo luogo bisogna considerare che la ben definita strutturalità imposta da JADE potrebbe, in alcuni casi, costringere l'attività di progettazione entro limiti piuttosto stringenti: se da un lato, infatti, JADE offre un vantaggio semplificando l'implementazione di processi dal comportamento complesso o la realizzazione di protocolli di comunicazioni piuttosto elaborati, dall'altro potrebbe introdurre dei vincoli o rendere l'implementazione più artificiosa soprattutto nelle fasi di inizializzazione della piattaforma e di interazione con alcuni servizi.

Infine, per ciò che riguarda la gestione dei guasti, potrebbe rivelarsi più problematico del previsto realizzare una gestione il più possibile accurata: è possibile che ci si debba concentrare solo su alcuni tipi di condizioni di failure, soprattutto in relazione al carico di lavoro comportato dalle altre fasi del progetto.

1.4 Strategia di progettazione

Con l'intenzione di conferire un certo grado di coerenza all'attività di progettazione, si propone un piano di lavoro che guidi lo sviluppo e la seguente realizzazione.

In primo luogo sarà necessaria un'analisi approfondita della *struttura di JADE* per quanto riguarda i servizi di base e quelli aggiuntivi come, ad esempio, quelli per la mobilità e la persistenza degli agenti.

Seguirà uno *studio accurato del prototipo esistente* in tutte le sue parti: dalle fasi di boot dei container, a quelle della creazione degli agenti; dalla

struttura dei singoli behaviour che compongono gli agenti, alla definizione dei protocolli attualmente utilizzati per le comunicazioni ed, in particolare, per la gestione del trasferimento delle tranche.

Successivamente ci si occuperà di individuare delle soluzioni per il problema della *comunicazione fra gli agenti attraverso Internet*, soprattutto per quanto riguarda il trasferimento dei contenuti.

Quindi si concentrerà l'attenzione sulla *gestione dello stato di peer e regioni*: si studierà un modello per la gestione delle primitive di amministrazione (che dovranno essere definite accuratamente) e si analizzeranno le caratteristiche di eventuali componenti aggiuntivi che possano rivelarsi utili allo scopo.

Il corso della progettazione proseguirà analizzando quali accorgimenti possano rendere possibile la *rilevazione di alcuni tipi di guasti*, come gli errori di comunicazione che si possono verificare durante la gestione dei trasferimenti, durante i trasferimenti stessi e durante i cambiamenti di stato di peer e regioni.

Dopo si passerà all'*implementazione dei componenti*: questa fase potrà comportare modifiche al modello progettato in precedenza nel caso in cui si intraveda *in itinere* una soluzione migliore o si rilevino alcune incopatibilità di carattere tecnico. Parallelamente si svolgeranno le attività di test delle funzionalità sviluppate ed il collaudo dei vari componenti e delle interazioni fra di essi.

Infine si procederà alla *revisione del codice* e ad un attento refactoring al fine di migliorare le proprietà non funzionali del software, quali la leggibilità e la struttura del codice stesso, nonché il suo grado di manutenibilità e la sua estendibilità.

CAPITOLO 2

Comunicazioni inter-regione attraverso Internet

In questo capitolo vengono individuati i requisiti specifici del nuovo sistema per quanto riguarda le comunicazioni fra regioni differenti attraverso Internet.

All'analisi delle problematiche connesse a questo argomento, con particolare attenzione a quelle riguardanti l'infrastruttura di rete, seguirà la descrizione di tre ipotesi di soluzione. Infine si giungerà alla definizione di alcune linee guida per l'implementazione, che tengono in considerazione i vincoli imposti dal sistema attuale.

2.1 Introduzione

Il prototipo attualmente esistente del sistema prevede la comunicazione fra regioni distinte (a patto che coesistano all'interno di una stessa LAN) durante le seguenti operazioni:

- ricerca extra-regione;
- selezione delle fonti;
- trasferimento di una tranche.

Per quanto JADE offra, per lo scambio di messaggi fra peer, un sistema di indirizzamento basato su identificatori globali, si palesa la necessità di

apportare alcune modifiche ai componenti esistenti e di introdurne di nuovi che tengano in considerazione i vincoli legati alle configurazioni dell'infrastruttra di rete su cui poggia l'intero sistema.

Inoltre è necessario considerare le comunicazioni connesse allo sviluppo delle nuove funzionalità del sistema: prima fra tutte la gestione delle primitive di amministrazione, che richiedendo un valido modello di riferimento, verrà discussa nel corso del *CAPITOLO 3*.

Prima di proseguire, si precisa che le considerazioni relative alla gestione dei guasti verranno presentate più avanti e che si tralasciano, almeno per il momento, le questioni sulla sicurezza delle comunicazioni e del sistema nel suo complesso.

2.2 Analisi dei requisiti

Di seguito si descrivono le procedure che richiedono una comunicazione inter-regione e che individuano, quindi, i requisiti richiesti dal nuovo sistema per quanto concerne l'interazione attraverso Internet. Le procedure riportate sono le seguenti:

1. inizializzazione della piattaforma e mobilità degli agenti;
2. ricerca extra-regione;
3. selezione delle fonti;
4. trasferimento di una tranche;
5. gestione delle primitive di amministrazione.

Si ritiene necessaria una descrizione di tali procedure affinchè si possano individuare i punti nevralgici su cui porre maggiore attenzione.

Dal momento che si utilizzano gli acronimi definiti per gli agenti, è utile far riferimento al paragrafo introduttivo *1.2.1* per comprenderne il ruolo o la dislocazione.

2.2.1 Inizializzazione della piattaforma e mobilità degli agenti

Ogni regione (che coincide con una AgentPlatform[1]) è composta da un superpeer e da un numero arbitrario di peer. Su ogni peer risiede un containter, mentre sul superpeer risiedono sia un container (in quanto il

[1] Cfr. 1.2.2, pag. 7.

superpeer svolge anche le attività di un comune peer), sia un main container.

Ogni containter viene inizializzato con alcuni parameteri, fra cui ci sono uno o più indirizzi del protocollo MTP[2] che permettono agli agenti presenti nel container di comunicare sia con gli altri container della piattaforma, sia con le altre piattaforme (ovviamente anche la migrazione degli agenti verso altre piattaforme -leggasi "regioni"- si basa sulla visibilità dei container).

Di coseguenza si rileva la necessità di rendere in qualche modo raggiungibile - direttamente o indirettamente - ciascun container per tutti gli altri container sia all'interno che all'esterno della regione in cui si trova.

2.2.2 Ricerca extra-regione

RRA - CA - SearchAgent

Quando l'agente RSMA di un peer, analizzando il proprio palinsesto, rileva l'assenza di un contenuto, formula una richiesta per ogni tranche che compone il contenuto stesso; queste richieste vengono inoltrate al SA e raggiungono (*Figura 2-1*, p.18) [1] l'RRA[3] . Questo agente, per ogni richiesta, restituisce una risposta contenente una lista di fonti per quella determinata tranche.

Se non sono presenti fonti all'interno della regione, dà inizio ad una ricerca extra-regione: chiede [2] all'agente CA di creare ([3] e [4]) due SearchAgent affinchè esplorino l'overlayNetwork visitando tutti i superpeer che la compongono e rimane in attesa dell'esito di tale ricerca.

I due SearchAgent - come viene accuratamente descritto nell'elaborato (Biscuola, 2009 p. 124 - 134) - percorrono l'anello nei due sensi ([5] – [10], in *Figura 2-1*, p.18 e *Figura 2-2*, p.18) e, una volta visitati tutti i superpeer, restituiscono ([11] e [12]) i risultati al CA che li ha creati. Il CA fa la sintesi di tali risultati e li comunica [13] al RRA che li inoltra [14] all'RQMA del peer richiedente.

[2] Un esempio di indirizzo MTP, implementato su HTTP, è: http://151.16.149.126:6668/acc

[3] L'agente RRA tiene traccia di tutte le tranche possedute dai peer della propria regione.

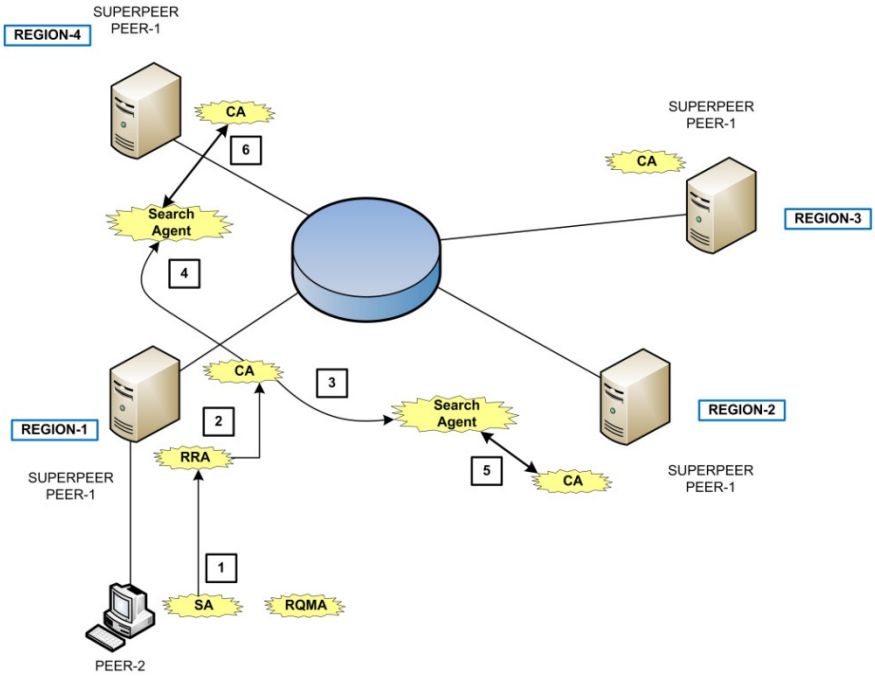


Figura 2-1 – Ricerca extra-regione (a).

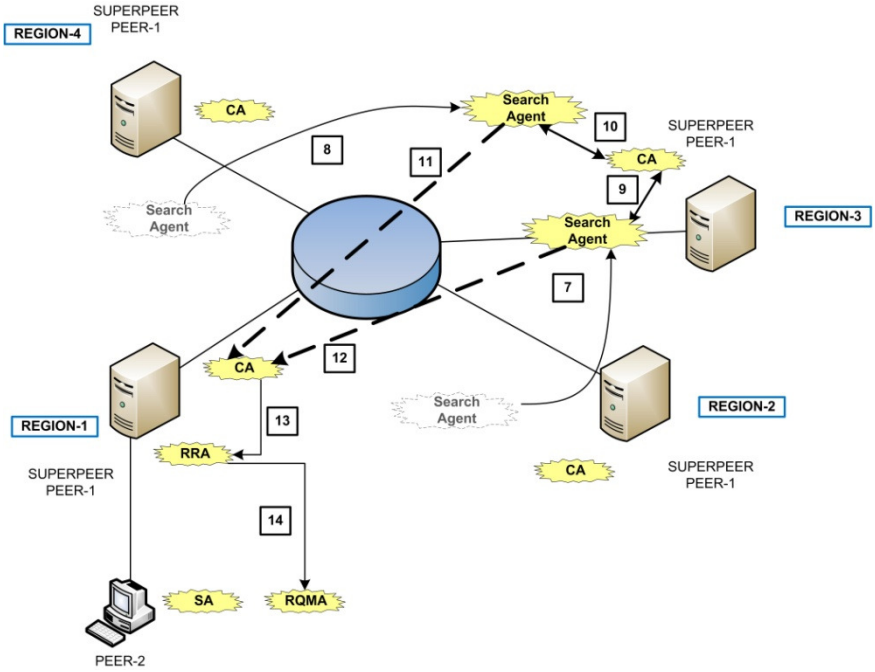


Figura 2-2 – Ricerca extra-regione (b).

Requisiti

E' necessario, quindi, che i SearchAgent possano visitare ciascun superpeer della overlayNetwork e restituire i risultati al CA che li ha originati, fornendo dei riferimenti utilizzabili per le comunicazioni successive.

2.2.3 Selezione delle fonti

DMA - MA - UMA

Quando il peer che ha richiesto le fonti per una determinata tranche entra in possesso della lista di tali fonti, l'agente DMA del peer in questione dà inizio alla fase della selezione delle fonti *Figura 2-3*: genera un agente MA per ogni peer che possiede la tranche e attende che questo migri o verso il container del peer (nel caso di fonte intra-regione) o verso il main container del superpeer (nel caso di fonte extra-regione). Ogni agente MA comunica con l'UMA della specifica fonte per cui è stato creato e si mette in coda prelevando un ticket. Quando l'UMA chiama il ticket, l'MA avvisa il DMA che, sulla base delle risposte inviate dagli altri MA migrati verso le altre fonti, accetta o declina la proposta di download (*Figura 2-4*).

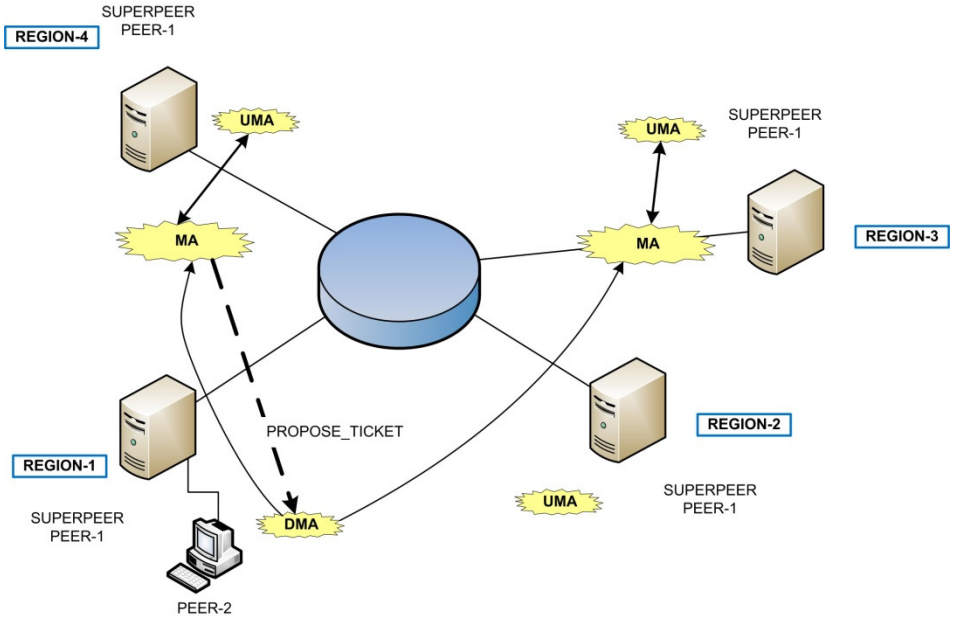


Figura 2-3 – Selezione delle fonti [extra-regione] (a).

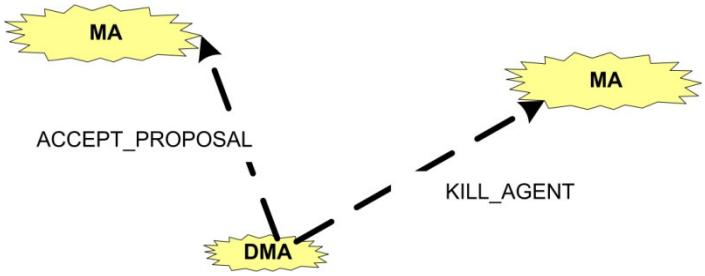


Figura 2-4 – Selezione delle fonti [extra-regione] (b).

Requisiti

E' necessario che ogni agente MA riesca e migrare verso il container del peer fonte o verso il main container della regione in cui tale peer si trova.

E' inoltre necessario che le comunicazioni fra MA e DMA, che possono avvenire all'interno della stessa regione o tra due regioni distinte, vengano garantite in entrambi i casi.

2.2.4 Trasferimento di una tranche

DA - UA

Una volta selezionata la fonte, l'UMA avvisa l'UA che contatta l'agente DA del peer richiedente chiedendo l'indirizzo (IP:port) per la connessione TCP necessaria al trasferimento. Il DA risponde alla richiesta e si mette in ascolto sulla porta comunicata al UA. Quest'ultimo effettua la connessione e trasferisce la tranche. Al termine, l'agente DA sblocca l'agente UMA del peer uploader affinchè possa servire altri MA.

Requisiti

Dal momento che, nella sua nuova versione, il sistema vedrà regioni appartenenti a LAN distinte, è necessario comprendere come gestire la visibilità diretta fra peer in regioni diverse: a meno che non si intenda rendere visibile ciascun peer dall'esterno della LAN in cui si trova[4], bisognerà stabilire come gestire le comunicazioni fra DA e UA. A tal proposito, si rimanda al paragrafo *2.4*.

[4] Ad esempio con il port forwarding di TUTTE le porte utilizzate dal peer stesso (da JADE, dai componenti per la gestione delle primitive, dai componenti per il trasferimento delle tranche).

2.2.5 Gestione delle primitive di amministrazione

Questa funzionalità non è prevista dal sistema nella sua versione attuale. Inoltre non sono ancora state definite le primitive di amministrazione; né è stato definito quale comportamento i nodi del sistema (peer e superpeer) debbano avere durante l'esecuzione di una primitiva.

Nonostante risulti prematuro definire dettagliatamente i requisiti del nuovo sistema relativamente a questo aspetto, si crede necessario fornire comunque alcune linee guida.

Si dovranno progettare, per ogni nodo (peer o superpeer), uno o più componenti in grado di inviare, inoltrare ed eseguire le primitive di amministrazione. Questi componenti dovranno anche gestire gli esiti dell'esecuzione delle varie primitive e darne comunicazione a chi le ha lanciate. Siccome le primitive di stato riguarderanno anche l'avvio e l'interruzione dell'esecuzione di un peer o di un superpeer, si dovrà prevedere l'eventuale funzionamento di uno o più componenti al di fuori della piattaforma JADE e, quindi, senza l'ausilio degli strumenti offerti da JADE stesso (come, ad esempio, lo scambio di messaggi o il sistema di indirizzamento globale).

2.3 NAT traversal

In ambito P2P quello della comunicazione tra i peer in presenza di dispositivi NAT è un argomento che ha comportato l'elaborazione di numerose tecniche volte a rendere possibile il funzionamento delle applicazioni in queste condizioni.

Dal momento che il sistema che verrà realizzato incentrerà le proprie funzioni sulla comunicazione attraverso Internet, con questo paragrafo si vuole definire il problema e descrivere alcune delle tecniche sperimentali attualmente utilizzate per aggirarlo.

2.3.1 NetworkAddressTranslation

I dispositivi NAT sono nati per far fronte al limitato numero di indirizzi IP a 32 bit disponibili per l'accesso ad Internet, la più estesa WAN attualmente esistente. Consentono, infatti, a più host con indirizzi IP privati

di condividere uno o più indirizzi pubblici. Inoltre nascondono la topologia della rete interna definendo una netta separazione da quella pubblica.

Gli indirizzi privati vengono messi in relazione con quelli pubblici attraverso l'associazione delle coppie di tipo IP_privato:porta_privata con quelle di tipo IP_pubblico:porta_pubblica. La traduzione avviene in maniera automatica a cura del NAT, che la rende trasparente a livello di applicazione. Anche l'assegnazione degli indirizzi pubblici IP_pubblico:porta_pubblica agli host (che può essere statica o dinamica) avviene in modo trasparente; così come la traduzione dell'indirizzo di destinazione per i pacchetti del protocollo ICMP da inoltrare agli host della rete privata[5].

Generalmente un dispositivo NAT[6] permette tutte le comunicazioni in uscita verso la rete pubblica; quelle in entrata vengono di norma consentite solo se in risposta a connessioni precedentemente iniziate da host interni[7]: in caso contrario, tali comunicazioni vengono rifiutate.

---

[5] Una descrizione accurata delle funzioni di un dispositivo NAT ed una più dettagliata tassonomia vengono fornite in (Shriushek, et al., August 1999).

[6] Esistono diverse tipologie di NAT. La più diffusa è quella dei Traditional NAT, suddivisi in Basic NAT e NAPT. I Basic NAT, usati quando il numero di indirizzi pubblici è maggiore di quelli privati, eseguono una semplice mappatura che si limita al solo indirizzo IP; i NAPT, invece, svolgono una mappatura che coinvolge anche le porte.

[7] Una ulteriore classificazione dei NAT riportata in (Biggadike, et al., 2005) descrive il comportamento assunto relativamente alla mappatura tra indirizzi privati (di tipo A:X, dove A rappresenta l'IP_privato e X la porta_privata dell'host interno) e indirizzi pubblici (di tipo NATa:Y, dove NATa rappresenta l'IP_pubblico assegnato al NAT dell'host A e Y la porta pubblica). Si distinguono:

- una mappatura Full Cone, che associa sempre la stessa porta pubblica Y per qualsiasi comunicazione eseguita dall'host interno attraverso la porta privata X;
- una mappatura Restricted Cone, che indirizza verso A:X le comunicazioni inziate da un host esterno C e dirette verso la porta pubblica NATa:Y solo se l'host A ha iniziato in precedenza una comunicazione verso l'host esterno C;
- una mappatura Port Restricted Cone, che indirizza verso A:X le comunicazioni iniziate da un host esterno C dalla porta C:M verso la porta pubblica NATa:Y solo se l'host A ha iniziato in precedenza una comunicazione verso C:M utilizzando la porta A:X;
- una mappatura Symmetric, che associa una porta pubblica diversa ad ogni comunicazione instaurata dall'host interno A con un diverso host esterno: ad esempio alla comunicazione instaurata da A:X verso C:M verrà associata la porta NATa:Y, mentre alla comunicazione da A:X verso B:Q verrà associata la porta B:Q.

E' proprio questo il motivo per cui sono state proposte diverse tecniche per consentire la comunicazione UDP o TCP fra host posti dietro dispositivi NAT.

2.3.2 Connection Reversal

Questa tecnica generica può essere applicata a comunicazioni basate sia su TCP che su UDP a patto che solo uno dei due host che vogliono instaurare una comunicazione acceda ad Internet attraverso un dispositivo NAT (di seguito è l'host A ad essere in questa condizione). Per consentire la comunicazione tra i due host A e B, è previsto che entrambi siano in grado di connettersi ad un terzo host noto e visibile ad entrambi: il rendezvous server S.

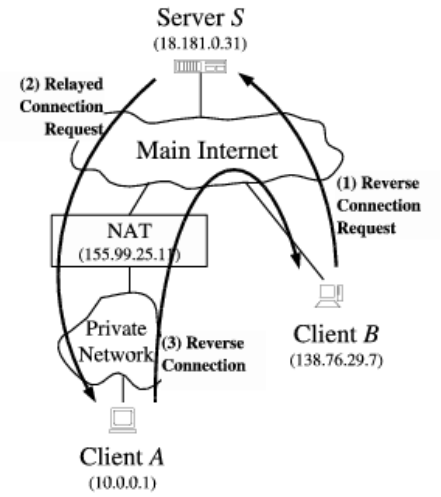


Figura 2-5 - Connection Reversal

Se A volesse instaurare una connessione verso B, il NAT di A consentirebbe la connessione (in uscita) verso B e la comunicazione avrebbe luogo senza problemi.

Se invece fosse B a voler iniziare la comunicazione verso A, un tentativo diretto verso il NAT di A fallirebbe in ogni caso, dal momento che il NAT di A non permetterebbe tale comunicazione; d'altra parte se B inoltrasse ad A tramite il rendezvous server S una richiesta di connessione, sarebbe A ad effettuare una connessione diretta verso B, "rovesciando" i ruoli.

Si noti che le mappature di tipo cone-mapping (che associano una sola porta pubblica a ciascun endpoint privato) mantengono valida l'associazione porta_pubblica – porta_privata finchè non scade un determinato periodo di inattività.

2.3.3 Relaying

Questa tecnica è quella con maggiori possibilità di applicazione. Come la precedente, si basa sulla presenza di un server S verso cui gli host A e B, che appartengono a reti private distinte controllate da due NAT differenti, siano in grado di instaurare una connessione.

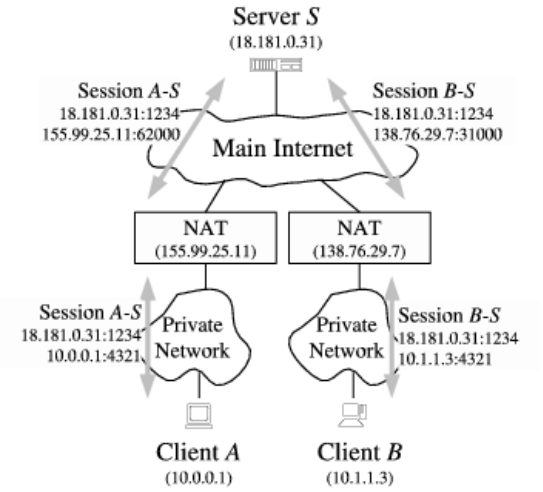


Figura 2-6 – Relaying.

Dal momento che i due NAT impediscono ai rispettivi host una comunicazione diretta, l'host A invia i propri messaggi indirizzati a B verso il server S, affinchè quest'ultimo li inoltri a B. B procede allo stesso modo, rendendo possibile una effettiva comunicazione tra due host.

Il principale svantaggio consiste nel fatto che la comunicazione fra A e B utilizza le risorse del server S, sia in termini di potenza di calcolo che in termini di banda.

Nella *Figura 2-6* si notano le connessioni instaurate fra gli endpoint pubblici di A (155.99.25.11:62000) e di B (138.76.29.7.31000) (associati dai rispettivi NAT agli endpoint locali) con l'endpoint pubblico di S(18.181.0.31:1234) che consentono, per l'appunto, l'inoltro dei messaggi in entrambi i sensi.

2.3.4 UDP Hole Punching

L'UDP Hole Punching è una tecnica che consente di instaurare una sessione UDP diretta tra due host posti dietro NAT sfruttando la presenza di

un rendezvous server solamente durante le fasi iniziali della comunicazione[8].

Rendezvous Server

Il principale compito del rendezvous server S è quello di mantenere in memoria gli endpoint privato e pubblico di ogni host che effettua una registrazione col server: l'endpoint privato è quello a partire dal quale l'host effettua la connessione al server (può essere comunicato a quest'ultimo come contenuto di un messaggio); l'endpoint pubblico è quello associato dal NAT all'endpoint locale dell'host (il server lo ricava dagli header dei pacchetti IP e UDP: nel primo viene letto il campo Source Address che fornisce l'IP, mentre nel secondo viene letto il campo Source Port che fornsce la porta UDP).

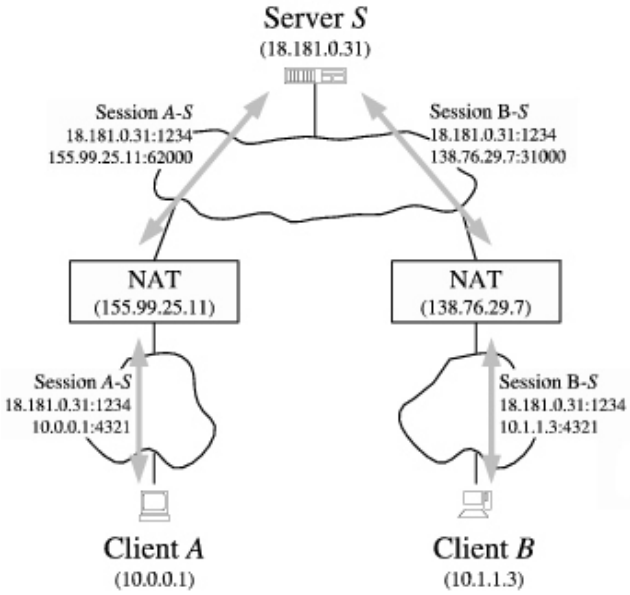


Figura 2-7 – Before UDP Hole Punching

[8] Riferimenti sull'argomento si trovano nella sezione 5.1 di (Holdrege, et al., January 2001) e in (Ford, et al., 2005). Questo paragrafo, così come il successivo, trae spunto proprio da quest'ultima fonte.

Instaurazione della sessione UDP

Il processo (*Figura 2-8*) che porta A a stabilire una sessione UDP diretta con B è il seguente:

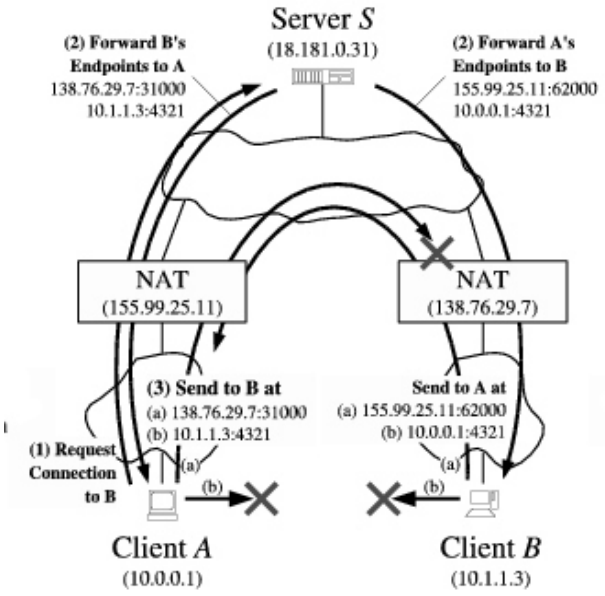


Figura 2-8 – The UDP Hole Punching process

1. A stabilisce una sessione UDP con S chiedendogli, eventualmente, di stabilire una sessione UDP con B;
2. S stabilisce la sessione UDP con B, comunica a B gli endpoint pubblico e privato di A e comunica ad A gli endpoint pubblico e privato di B;
3. appena A riceve gli endpoint di B, tenta di stabilire una sessione UDP diretta con B verso entrambi gli endpoint (di B) comunicatigli dal server S. Lo stesso fa B, che tenta di stabilire una sessione UDP verso i due endpoint di A. Se A e B si trovano nella stessa rete privata, la sessione verrà creata attraverso gli endpoint privati dei due host; altrimenti (*Figura 2-9*, p.27) verrà creata tra i rispettivi endpoint pubblici[9].

[9] In (Ford, et al., 2005 p. par. 3.5, fig. 8) si analizza nello specifico anche il caso che vede l'instaurazione di una sessione UDP tra peer dietro livelli multipli di NAT. Esistono protocolli come STUN (Rosenberg, et al., March 2003) che, tramite un server intermedio, tentano di prevedere il comportamento dei nat attraversati da una comunicazione UDP; tuttavia prevedere la porta pubblica associata da un NAT ad un

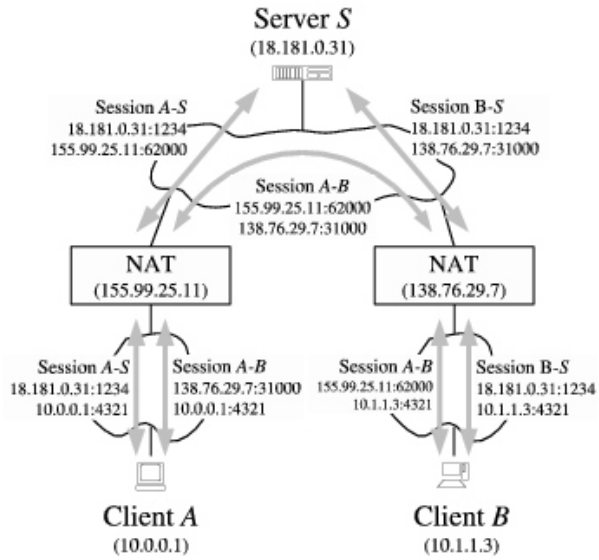


Figura 2-9 – After UDP Hole Punching

2.3.5 TCP Hole Punching

Stabilire una connessione TCP tra due host posti dietro NAT rappresenta un'operazione più complessa rispetto al caso del protocollo UDP, essenzialmente perché il TCP richiede una fase preliminare di connessione (three-way-handshake) che prevede lo scambio di tre messaggi (SYN, SYN+ACK, ACK) tra i due host.

Inoltre le tecniche sviluppate per implementare il TCP Hole Punching richiedono che un socket TCP possa essere utilizzato contemporaneamente per una connessione in ingresso ed una connessione in uscita: nonostante molti sistemi operativi prevedano le opzioni `SO_REUSEADDR` e `SO_REUSEPORT`, può capitare che questo requisito comporti il fallimento della procedura[10]. Tuttavia questo protocollo possiede determinate caratteristiche che lo rendono molto più efficace del UDP in molte applicazioni.

ednpoint privato non è un'attività che va sempre a buon fine, soprattutto in presenza di livelli multipli di NAT. Inoltre, se i dispositivi NAT in questione soddisfano le specifiche espresse nel paragrafo 2.3.6 (P2P-Friendly NAT), la tecnica dell'UDP Hole Punching dovrebbe funzionare correttamente senza complicazioni.

[10] Un approccio alternativo che può risolvere il problema è rappresentato dal Sequential Hole Punching di cui si scrive al paragrafo *2.3.5.2*.

Come descritto precedentemente, un host dietro NAT non può essere raggiunto da nessun messaggio se prima non è stata instaurata una connessione verso l'host che invia tale messaggio.

Sono state elaborate alcune soluzioni che operano a "basso livello" sui singoli pacchetti SYN – calibrando a dovere il TTL (TimeToLive) dei pacchetti IP su cui viaggia il TCP ed operando sui numeri di sequenza dei pacchetti TCP[11] -, ed altre di maggior versatilità che eseguono anche una diagnostica pre-connessione[12].

Nei paragrafi che seguono si esaminano attentamente le soluzioni proposte da NatTrav e P2PNAT, che si ritiene essere più idonee al progetto da svolgere, e si riportano alcune considerazioni sui NAT P2P-fiendly.

2.3.5.1 NatTrav

Questa tecnica, descritta da Eppinger in (Eppinger, 2005), è pensata per dispositivi NAT di tipo cone-mapping (cfr. nota 7, pag. 22) e prevede la presenza di un Connection Broker presso cui si debbano registrare (fornendo un identificativo universale URI) i peer intenzionati ad instaurare una connessione TCP diretta.

Come rappresentato in *Figura 2-10* (p.29), il peer Initiator I formula una prima richiesta (step 3) al connection broker B, il quale (step 4) chiede via UDP al peer Recipient R di instaurare una connessione TCP verso B (step 5) e comunica a R le informazioni sull'endpoint pubblico di I, memorizzando quelle sull'endpoint pubblico di R e chiudendo la connessione TCP con R (step6).

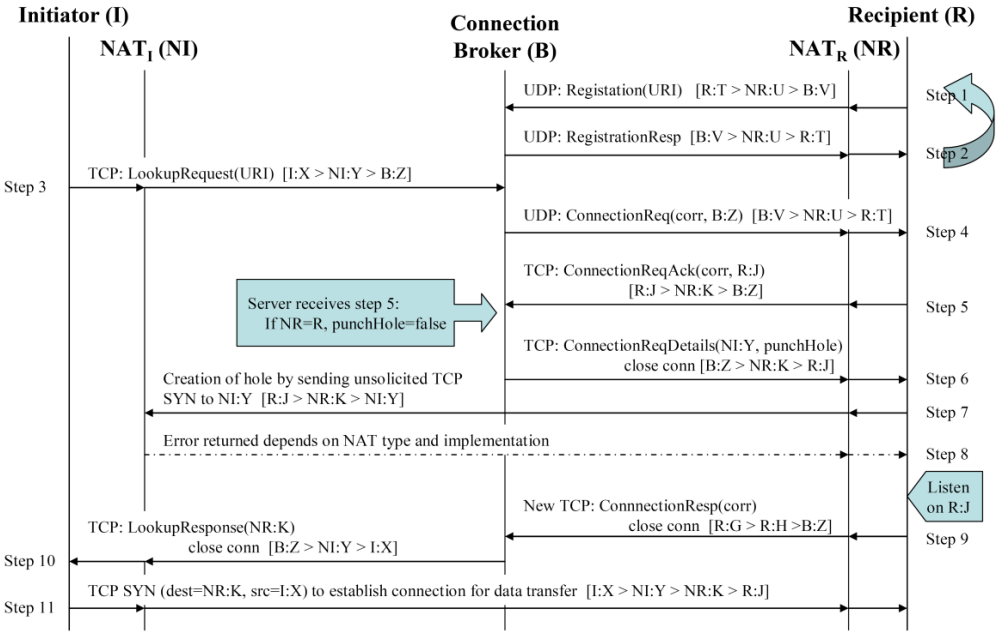
Successivamente R tenta una connessione diretta verso l'endpoint pubblico di I (step 7) e rimane in attesa di rilevare l'errore causato dal NAT di I che non consente la comunicazione in entrata (step 8).

Quindi R comunica a B (step 9) l'avvenuta esecuzione della prima fase della creazione dell'hole (step 7 e 8); infine B comunica ad I le informazioni sull'endpoint pubblico di R (step 10) ed I instaura con successo la

[11] Un esempio è STUNT -(Francis, et al.), (Francis, 2003) - che consiste in un'estensione di STUN (Rosenberg, et al., March 2003) al fine di comprendere le caratteristiche del TCP (sequenceNumber).

[12] Come NATBlaster (Biggadike, et al., 2005), che tenta di determinare in che modo il NAT associa porta pubblica e porta privata (sequenziale, consistente, etc..) e prevede tecniche diverse utilizzabili a seconda del comportamento dei NAT.

connessione diretta con R (step 11), dal momento che il NAT di R non blocca il pacchetto contenente il SYN proveniente da I interpretandolo come appartenente alla precedente comunicazione iniziata da R.



Note: The IP addresses and ports used for communication are shown in square brackets. For example: [I:X > NI:Y > B:Z] denotes communication from the Initator's IP address I and port X to the connection broker's IP address B on port Z via NAT$_I$'s public address NI and public port Y.

Figura 2-10 - NatTrav

Si noti che il corretto funzionamento di questa tecnica dipende sicuramente dal comportamento dei NAT: infatti si verificherebbe il fallimento dell'intera procedura se durante lo step 8, il NAT di R eseguisse il forwarding verso R del pacchetto di errore (TCP RST o ICMP TTL exceeded) relativo al tentativo di connessione dello step 7 o se cancellasse la mappatura tra l'endpoint pubblico e privato di R.

2.3.5.2 P2PNat

Questa tecnica, proposta da Kegel (Ford, et al., 2005), risulta molto simile a quella proposta per l'UDP nel paragrafo *2.3.4*. Tuttavia, mentre per l'UDP Hole Punching ogni host è in grado di comunicare con il rendezvous server S e con un numero arbitrario di altri host attraverso lo stesso socket, per il TCP Hole Punching è necessario che ogni host associ alla stessa porta più socket TCP (*Figura 2-11*): uno per la connessione verso S, due per i tentativi di connessione verso gli endpoint pubblico e privato dell'altro peer ed uno su cui rimanere in ascolto in attesa della connessione proveniente dall'altro peer.

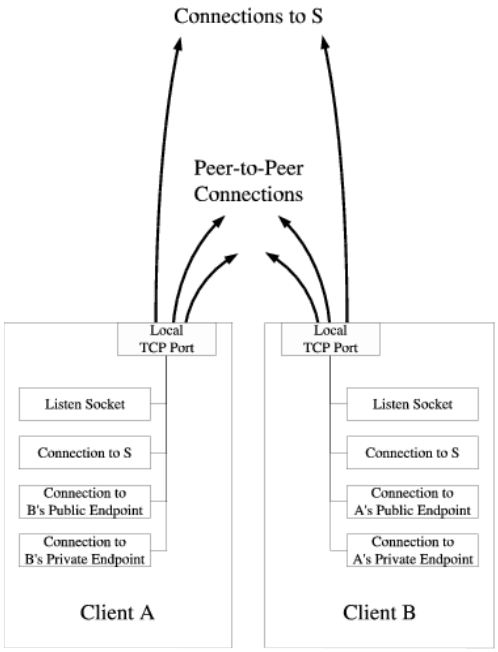


Figura 2-11 – P2PNat – socket e connessioni

Gli host A e B che vogliano instaurare una connessione TCP diretta, sfruttano la presenza di un rendezvous server S, che memorizza le informazioni sugli endpoint pubblico e privato di entrambi, proprio come descritto per l'UDP Hole Punching.

I passi della procedura sono i seguenti (si considerino la *Figura 2-7*, la *Figura 2-8* e la *Figura 2-9*):

1. A stabilisce una sessione TCP con S chiedendogli gli endpoint di B;
2. S stabilisce la sessione TCP con B, comunica a B gli endpoint pubblico e privato di A e comunica ad A gli endpoint pubblico e privato di B;
3. appena A riceve gli endpoint di B, tenta di stabilire una sessione TCP diretta con B verso entrambi gli endpoint (di B) comunicatigli dal server S.

   Lo stesso fa B, che tenta di stabilire una sessione TCP verso i due endpoint di A. Se A e B si trovano nella stessa rete privata, la sessione verrà creata attraverso gli endpoint privati dei due host; altrimenti verrà creata tra i rispettivi endpoint pubblici.

Se i NAT dei due peer assumono un comportamento P2P-friendly (paragrafo *2.3.6*), il canale si instaura sia nel caso (a) in cui il SYN di A raggiunga il NATb di B prima che il SYN di B passi attraverso il NATb di B, sia nel caso speculare (b).

Infatti (caso a) se il SYN di A raggiunge il NATb di B ma il SYN di B non ha ancora raggiunto il NATb di B, il NATb interpreterà il SYN di A come una comunicazione non richiesta e non la inoltrerà a B; d'altronde quando il SYN di B raggiungerà il NATa di A, quest'ultimo lo interpreterà come appartenente alla comunicazione precedentemente iniziata da A verso l'endpoint pubblico di B e inoltrerà tale messaggio ad A che potrà quindi inviare un SYN-ACK a B, il quale potrà concludere la terza fase di negoziazione della connessione TCP.

Lo stesso dicasi per il caso (b).

Considerazioni

Si ritengono necessarie alcune considerazioni sull'applicazione che si trova ad eseguire la procedura di TCP Hole Punching.

Nell'eventualità in cui ci si trovi nel primo dei due casi appena descritti, è possibile che si verifichi una delle seguenti eventualità[13]:

1. il SYN di B che raggiunge A viene associato allo stesso socket che A stava utilizzando per connettersi a B e quindi A risponde con un SYN+ACK;
2. il SYN di B, non contenendo anche l'ACK relativo al SYN di A, viene interpretato come un nuovo tentativo di connessione e di conseguenza viene inizializzato un nuovo socket (in A) sulla stessa porta (è attraverso questo nuovo socket che prosegue la procedura di negoziazione con B -invio del SYN+ACK e ricezione del ACK-). A questo punto l'host A riceverà, relativamente al socket attraverso cui ha effettuato il primo invio del SYN verso B, un errore di tipo "address already in use" che dovrà essere ignorato dall'applicazione che si occupa di effettuare il TCP Hole Punching.

[13] Il verificarsi dell'una o dell'altra può dipendere dal sistema operativo utilizzato: la prima appare più comune su sistemi BSD, mentre la seconda su Linux e Windows.

Un'altra considerazione riguarda la possibilità di un'apertura simultanea di due connessioni TCP, che si verificherebbe nel (raro) caso in cui entrambi i NAT venissero attraversati prima dal pacchetto SYN "uscente" e poi da quello "entrante", che non verrebbe quindi bloccato. Le applicazioni su entrambi gli host dovranno tener conto di questa eventualità e agire di conseguenza.

Un'ultima considerazione riguarda l'eventualità, da parte di un host, di non riuscire a mettersi in ascolto ed effettuare contemporaneamente una connessione sulla stessa porta, nonostante la corretta impostazione dei parametri `SO_REUSEADDR` e `SO_REUSEPORT`.

In questo caso una soluzione potrebbe consistere nel ricorrere ad un *Sequential Hole Punching*: la procedura sarebbe molto simile a quella per l'Hole Punching definito da P2PNat eccetto per il fatto che ciascun host, mentre tenta di iniziare la connessione verso l'altro, non rimane anche in ascolto sulla stessa porta.

In breve A, effettuata la fase di rendezvous con S, non si mette subito in ascolto sulla stessa porta; B, su richiesta di S, invia un SYN verso A, creando un "hole" nel NATb di B (il SYN non viene inoltrato ad A dal NATa di A); B chiude la propria connessione con S e si mette in ascolto; S chiude la sua connessione con A che, a questo punto, invia il SYN a B.

2.3.6 P2P-Friendly NAT

In questa breve sezione si riassumono le due principali caratteristiche che dovrebbero appartenere ad un NAT in grado di creare meno problemi possibile alle procedure di hole punching descritte in questi paragrafi.

Associazione consistente degli endpoint

Le tecniche di hole punching descritte funzionano se i NAT associano porta pubblica ed endpoint privato in modo consistente: tutte le comunicazioni originate dallo stesso endpoint privato devono essere associate alla stessa porta pubblica (i NAT con un comportamento simile sono definiti *cone NAT* - cfr. nota 7, pag.22 -). Se ciò non accadesse, si dovrebbe prevedere il metodo di associazione tra endpoint privato e pubblico utilizzato dal NAT e questa operazione costituirebbe una non semplice difficoltà da superare.

Gestione delle comunicazioni "indesiderate"

Quando un NAT riceve una richiesta di connessione dall'esterno, dovrebbe scartare il pacchetto ricevuto senza rispondere con un TCP Reset o un messaggio di errore ICMP.

Si noti, tuttavia, che questo comportamento non compromette necessariamente l'intera procedura di Hole Punching, ma per lo meno dilata i tempi richiesti per una completa riuscita del processo.

2.4 Ipotesi di soluzione

Alla luce dell'analisi dei requisiti richiesti dal nuovo sistema e delle considerazioni relative alle tecniche di Hole Punching presentate nel precedente paragrafo, di seguito si presentano tre ipotesi di soluzione per il problema della comunicazione fra regioni attraverso Internet che, si ricorda, riguarda:

1. l'inizializzazione della piattaforma e la mobilità degli agenti;
2. la ricerca extra-regione;
3. la selezione delle fonti;
4. il trasferimento di una tranche;
5. la gestione delle primitive di amministrazione.

2.4.1 Ipotesi 1: "VPN"

La prima ipotesi di soluzione schematizzata in *Figura 2-12* (p.34) prevede la configurazione di una Virtual Private Network[14] tra le regioni che costituiscono il sistema.

In questo modo è possibile, con qualche accorgimento, ricondurre il nuovo scenario - che vede le regioni interconnesse attraverso Internet - all'interno dei limiti imposti dal prototipo attualmente esistente, che si limita ad eseguire una comunicazione inter-regione solo fra regioni appartenenti allo stesso dominio di collisione.

Per poter realizzare questo tipo di soluzione, si deve configurare una VPN di tipo LANtoLAN, che realizza un tunnel crittografato[15] che unisce due

[14] Una rete VPN (Virtual Private Network) permette di stabilire un collegamento crittografato tramite Internet tra sedi fisiche diverse (HOSTtoLAN o LANtoLAN).

LAN (geograficamente distanti) attraverso Internet, in modo da farle apparire separate da un unico segmento di rete[16].

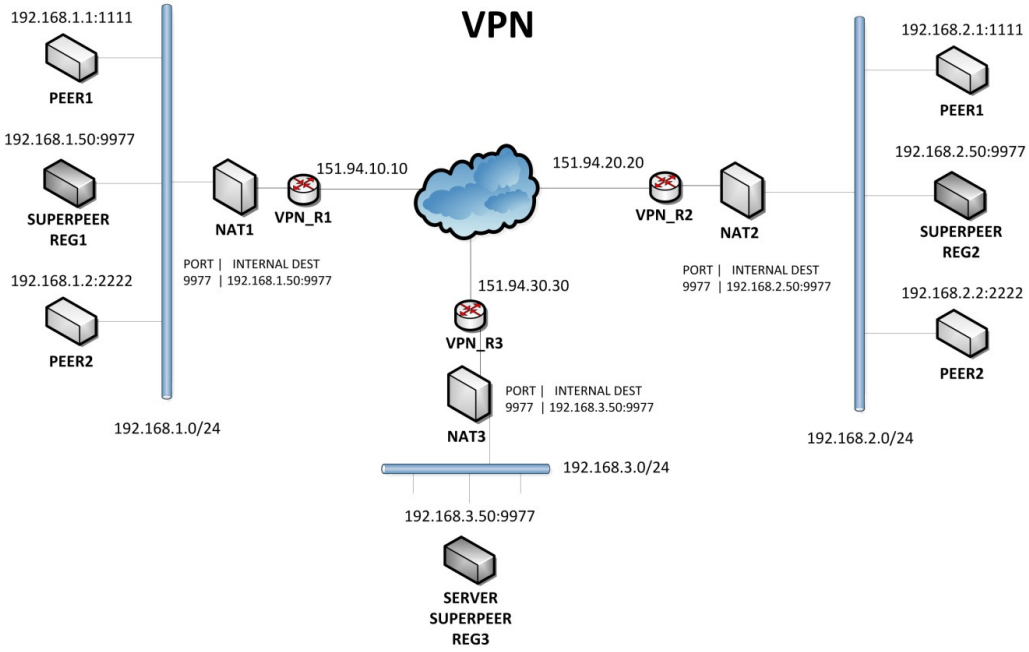


Figura 2-12 – Ipotesi 1: VPN

Esistono diversi protocolli standard per costruire VPN[17]. IPSEC, ad esempio, è quello sicuramente più conosciuto per VPN di tipo IP. Questo

[15] I tunnel VPN si possono classificare a seconda che al loro interno siano incapsulati pacchetti di Layer 2 (Data Link) o pacchetti di Layer 3 (Network). Nel primo caso le due LAN sono generalmente in bridge, per cui qualsiasi protocollo di livello 3 (es.IP) può passarvi (naturalmente anche il broadcast di livello 2 si propaga tra le due LAN). Nel secondo tipo di VPN invece, un solo protocollo di Layer 3 puo transitare (generalmente IP) e il traffico viene instradato mediante route statiche; si deduce, pertanto, che le due LAN devono appartenere a subnet distinte.

[16] Un altro tipo di VPN (HOSTtoLAN) consiste in un tunnel criptato che, attraverso Internet, connette l'host esterno ad un VPN server interno alla LAN, facendo in modo che l'host esterno risulti come se fosse fisicamente all'interno della LAN.

[17]Tra i vari protocolli esistenti si citano: PPTP (Point-to-Point Tunneling Protocol), L2F (Layer Two Forwarding), L2TP (Layer Two Tunneling Protocol) e IPSec. I protocolli PPTP e IPSec offrono il livello di protezione più elevato. Il protocollo PPTP permette di incapsulare i pacchetti dati in un datagramma IP al fine di creare una connessione punto-a-punto. In questo caso, i dati vengono protetti a due livelli poiché i dati sulla rete locale (come gli indirizzi dei PC) vengono incapsulati in un messaggio PPP che è a sua volta incapsulato in un messaggio IP. IPSec offre tre moduli (Authentication Header, Encapsulating Security Payload e

protocollo prevede che, per ogni singolo pacchetto, l'header venga autenticato (dal protocollo AH, Authentication Header) e il payload venga criptato (dal protocollo ESP, Encapsulating Security Payload). Tuttavia, il fatto di criptare il payload, contenente anche il numero di porta sorgente e destinazione del livello trasporto TCP/UDP, crea problemi ai router intermedi che effettuano NAT. Questi ultimi inoltre, modificando l'IP sorgente dei pacchetti, generano errori di checksum per il protocollo di autenticazione AH. Una soluzione generalmente utilizzata, che prende il nome di NAT Traversal (NAT-T), è quella di incapsulare IPSEC in datagrammi UDP.

I principali vantaggi che derivano dalla scelta di realizzare una VPN consistono nella sicurezza delle comunicazioni (garantita dai protocolli utilizzati) e nell'elevato grado di scalabilità offerto. A livello di applicazione, le funzionalità degli agenti che compongono ciascun peer (o superpeer) vengono preservate dal momento che ogni peer è in grado di comunicare direttamente con tutti i peer di tutte le regioni; anche l'introduzione di nuove funzionalità trarrebbe vantaggio dalla presenza della VPN in quanto, a fronte di una configurazione iniziale, renderebbe più semplici i protocolli di comunicazione.

Tuttavia l'adozione di questo tipo di soluzione sottintende la necessità di configurare adeguatamente l'infrastruttura di rete in cui si colloca ciascuna regione; inoltre bisogna considerare l'infrastruttura di rete stessa perché deve permettere la creazione di una VPN.

2.4.2 Ipotesi 2: "Port forwarding" (all nodes)

La seconda soluzione proposta consiste nel configurare sul NAT il port fowarding di tutte le porte di comunicazione necessarie ad ogni peer della regione. In altre parole si richiede che il NAT crei e mantenga valida nel tempo l'associazione tra endpoint privato ed endpoint pubblico per tutte le comunicazioni da/verso ogni peer della regione, a prescindere dal fatto che siano già state instaurate o che lo saranno entro breve.

In questo modo ogni peer è direttamente raggiungibile dall'esterno e possono svolgersi tutte le operazioni legate alla ricerca extra-regione, alla selezione delle fonti, al trasferimento delle tranche, alla gestione delle

Security Association) che ottimizzano la protezione, garantendo la riservatezza, l'integrità e l'autenticazione dei dati.

primitive di amministrazione (comprese le primitive di stato per peer e regioni) e al funzionamento del sistema nel suo complesso.

In *Figura 2-13* è rappresentata la procedura di selezione delle fonti: il DMA del peer richiedente PEER1@REGION1 crea l'agente MA affinchè interagisca col peer fonte PEER1@REGION2; l'MA migra verso l'AMS della REGION2 (le porte usate da JADE per i messaggi MTP indirizzati ai superpeer delle regione devono essere aperte sui rispettivi NAT e le regole di port forwarding devono essere configurate correttamente); l'MA interagisce con l'UMA e gli comunica l'endpoint pubblico del peer richiedente su cui avverrà la ricezione della tranche (151.94.10.10:4545); l'MA informa il DMA del peer richiedente della chiamata del ticket e il DMA accetta la proposta (è già a conoscenza dell'endpoint pubblico del peer fonte della REGION2); a questo punto i due peer comuncano direttamente attraverso le porte pubbliche 4545 dei rispettivi NAT.

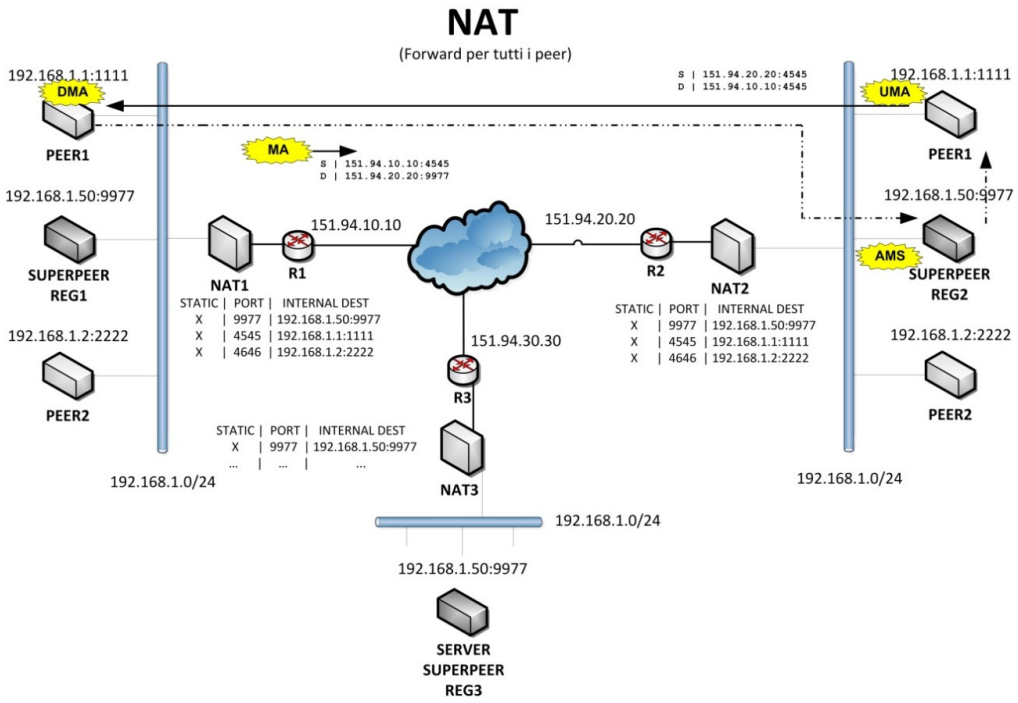


Figura 2-13 – Ipotesi 2: Port forwarding per il superpeer e per tutti i peer

Ovviamente una simile configurazione del NAT risulta molto laboriosa e rende poco pratica l'installazione del sistema: infatti sono molti i componenti che richiedono di interagire con le altre regioni e, anche se si cercasse di elaborare un modello che concentrasse l'elaborazione inter regione solo sui superpeer, l'introduzione di nuove funzionalità potrebbe gravare sulle prestazioni di questi ultimi.

Inoltre questo tipo di soluzione evidenzia un bassissimo grado di flessibilità: un peer (o un superpeer) potrebbe non essere più in grado di comunicare con le altre regioni sia nel caso in cui (ad esempio in seguito ad un ripristino dopo un guasto) gli fosse assegnato un IP diverso da quello inserito nelle regole di port forwarding del NAT, sia nel caso più comune in cui offra un certo servizio o esegua una certa comunicazione da una porta locale diversa da quella prevista dal NAT.

Un'altra considerazione riguarda la sicurezza: per quanto le comunicazioni possano essere rese sicure da meccanismi di crittografia, la configurazione delle molte regole di port forwarding sul NAT riduce il grado di separazione tra la rete interna e l'esterno, una delle principali peculiarità di questo tipo di dispositivi.

2.4.3 Ipotesi 3: "TCP Hole Punching and port forwarding"

Questa ipotesi di soluzione prevede:

a) la configurazione delle regole per il port forwarding per i soli superpeer delle regioni e non per tutti i peer di ogni regione;
b) l'implementazione di un servizio di rendezvous-proxy;
c) l'utilizzo delle tecniche di TCP Hole Punching per il trasferimento delle tranche.

I requisiti relativi all'inizializzazione della piattaforma, alla migrazione degli agenti, alla ricerca extra-regione e, in parte, alla gestione delle primitive di amministrazione vengono soddisfatti da (a): i superpeer sono raggiungibili direttamente dell'esterno attraverso tutte le porte utilizzate da JADE.

Il servizio di rendezvous-proxy (b) avviato sul superpeer di ogni regione permette, attraverso i superpeer, la comunicazione (scambio di messaggi) di ogni peer della regione con qualsiasi altro peer di altre regioni e quindi risolve le problematiche relative alla selezione delle fonti e supporta la gestione delle primitive di amministrazione.

L'utilizzo del TCP Hole Punching (c) permette il trasferimento diretto delle tranche tra peer di regioni diverse, senza che siano stati configurati i rispettivi NAT.

Il servizio di rendezvous-proxy (b) permette l'inoltro, da parte del superpeer su cui è avviato, dei messaggi indirizzati ad altri peer della regione e provenienti dall'esterno (proxy). Inoltre è fondamentale per la

corretta esecuzione del TCP Hole Punching, come si approfondirà di seguito (rendezvous).

Per quanto riguarda il trasferimento extra-regione, si ritiene utile presentare un prototipo basato sull'utilizzo di thread Java che è stato appositamente realizzato al fine di testare l'efficacia del TCP Hole Punching prima di ragionare in termini di agenti.

A differenza dalla soluzione proposta nel paragrafo *2.3.5.2* (P2PNat), questo prototipo sfrutta solo le regioni dei peer interessati al trasferimento della tranche e non necessita di una terza entità (Per una versione ad agenti si vedano i paragrafi *2.5* e *2.5.2*).

In *Figura 2-14* è stato schematizzato il funzionamento del prototipo.

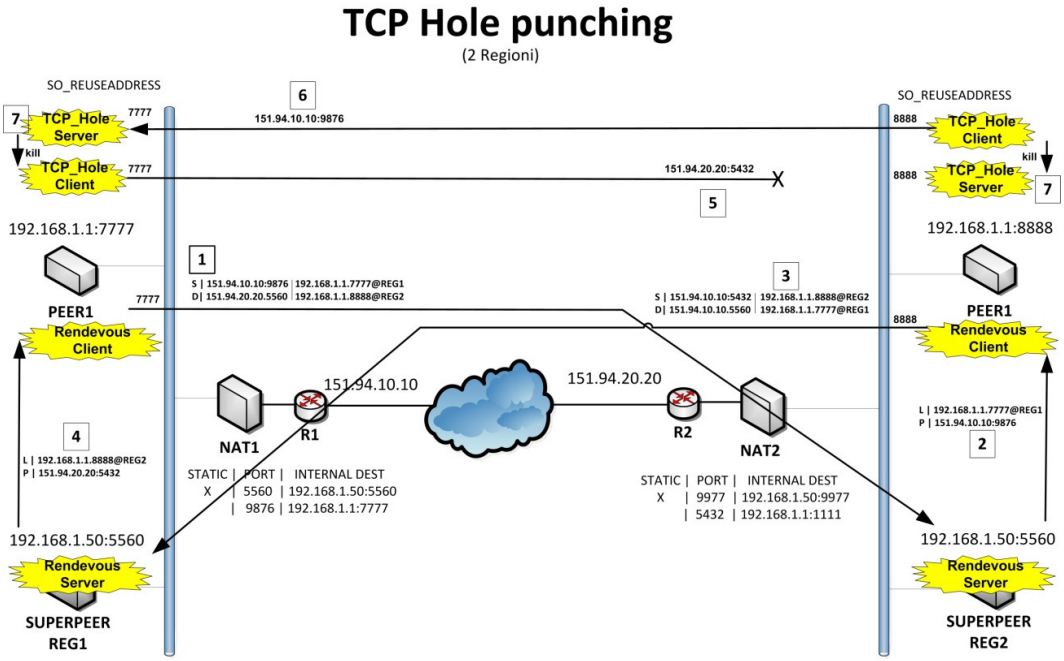


Figura 2-14 – Ipotesi 3: TCP Hole Punching e port forwarding. [thread]

Lo scenario vede il PEER1 della REGION1 e il PEER1 della REGION2 intenzionati ad instaurare una connessione TCP diretta.

I due superpeer offrono il servizio di rendezvous-proxy sulla porta 5560 mentre i due peer utilizzano rispettivamente la porta locale 7777 e 8888 per il trasferimento extra-regione delle tranche.

Fase di rendezvous

Inizialmente il thread rendezvous-client del PEER1@REGION1 contatta [1] il thread rendezvous-server del SUPERPEER@REGION2 dalla stessa porta 7777 che verrà utilizzata successivamente per la connessione all'altro peer.

Il SUPERPEER@REGION2 comunica [2] al rendezvous-client del PEER1@REGION2 l'endpoint pubblico del PEER1@REGION1 (151.94.10.10:9876) associato dal NAT1 all'endpoint privato (192.168.0.1:7777).

Il PEER1@REGION2 contatta [3] il rendezvous-server del SUPERPEER@REGION1 che comunica [4] al PEER1@REGION1 l'endpoint pubblico del PEER1@REGION2 (151.94.20.20:5432) associato dal NAT2 all'endpoint privato (192.168.0.1:8888).

Hole Punching

I due thread TCP_Hole_Server e TCP_Hole_Client avviati su ciascun peer svolgono l'ultima fase della procedura: il TCP_Hole_Server rimane in ascolto sulla porta utilizzata precedentemente per la connessione al rendezvous-server e il TCP_Hole_Client inizia la connessione verso l'endpoint pubblico dell'altro peer. In figura si suppone che sia il PEER1@REGION1 ad tentare per primo la connessione [5] che ovviamente viene rifiutata dal NAT2.

Quando il TCP_Hole_Client del PEER1@REGION2 inizia la connessione verso l'endpont pubblico dell'altro peer [6], il NAT1 la interpreta come risposta a quella precedente e la comunicazione viene instaurata con il TCP_Hole_Server[18] del PEER1@REGION1, comportando l'interruzione [7] degli altri due thread.

Questa ipotesi di soluzione prevede l'utilizzo delle tecniche di TCP Hole Punching, che possono comportare alcune problematiche soprattutto nel caso in cui si utilizzino primitive di stato per peer o regioni[19].

[18] E' possibile che la connessione venga stabilita tra i due thread client; in questo caso verranno interrotti i due thread server non interessati dalla comunicazione.

[19] Nello specifico è possibile che si verifichino problemi sul rilascio delle risorse (socket) da parte del sistema operativo quando vengono sfruttate le opzioni SO_REUSEADDRESS e SO_REUSEPORT.

Tuttavia rappresenta una valida alternativa alle prime due in quanto offre un buon grado di scalabilità e svincola dalla necessità di configurare il port forwarding per tutte le porte utilizzate da tutti i peer di ogni regione.

2.4.4 Considerazioni

Dall'analisi delle tre ipotesi di soluzione presentate nei paragrafi precedenti ed alla luce delle considerazioni riportate per ognuna di esse, ci si appresta ora a definire quale sia la strategia da seguire.

Scegliere di realizzare una VPN potrebbe risultare l'alternativa più vantaggiosa, sia in termini di semplicità di sviluppo dell'applicazione, sia in termini di sicurezza. Tuttavia le risorse attualmente a disposizione per le attività di progettazione e realizzazione non permettono una semplice configurazione della VPN o, comunque, non ne consentono la configurazione diretta: per eseguire l'implementazione del nuovo sistema e i test sul funzionamento dei componenti è necessaria la presenza di almeno tre regioni localizzate in luoghi geograficamente distinti; dal momento che si presume che due di essi siano la sede di Mestre della Sysdata S.p.a e il Dipartimento di Ingegneria dell'Informazione, si ritiene difficile avere libero accesso ai parametri di configurazione dei NAT e/o della VPN per entrambe le sedi.

Inoltre si intende progettare il nuovo sistema affinchè sia possibile utilizzarlo a prescindere dalla possibilità di creare una VPN attraverso l'infrastruttura di rete delle varie regioni: in una dimensione più ampia, si vuole realizzare un prodotto dotato di maggiore versatilità, in grado di poter essere impiegato in contesti di diversa natura.

La seconda ipotesi di soluzione rivela un'elevata semplicità di realizzazione e non impone ostacoli alla connettività tra tutti i peer di tutte le regioni[20]. Nonostante ciò, comporta una laboriosa fase di installazione del sistema una volta ultimato (configurazione dei NAT) e, soprattutto, denota una grave carenza di flessibilità: si richiede, infatti, che la struttura delle regioni rimanga stabile ed invariata nel tempo, precludendo la possibilità di introdurre eventuali miglioramenti che sfruttino, ad esempio, la mobilità dei peer.

[20] Anche eventuali tecniche per la gestione dei guasti (come la replicazione dei superpeer) potrebbero essere realizzate in maniera più semplice se ogni peer fosse direttamente raggiungibile dall'esterno della regione in cui si trova.

La terza ipotesi è quella che si ritiene essere il giusto compromesso tra staticità e dinamicità. E' adatta agli scopi previsti e si presta ad essere impiegata anche nel caso in cui si debba ripiegare verso la seconda ipotesi a causa di problemi nel TCP Hole Punching.

Il fatto che solo i superpeer siano direttamente raggiungibili dall'esterno della propria regione può rappresentare un limite in termini di efficienza, soprattutto per quanto riguarda il trasferimento di una tranche che richiede una certa quantità di tempo prima di poter iniziare. Tuttavia si sceglie di implementare questa soluzione anche in quanto offre un elevato grado di versatilità.

2.5 Linee guida per l'implementazione

Si rivela necessario, a questo punto, presentare una traduzione dell'ipotesi di soluzione basata su thread (paragrafo *2.4.3*) in un primo modello basato su agenti. Lo scopo non è quello di descrivere esaustivamente i meccanismi di funzionamento dei vari agenti[21], bensì è quello di focalizzare l'attenzione sul loro comportamento relativamente alle comunicazioni inter-regione e, nello specifico, durante la fase di selezione delle fonti ed il trasferimento effettivo di una tranche.

Prima di proseguire si elencano i requisiti di comunicazione per ogni nodo di tipo peer o superpeer; per completezza si comprendono anche i requisiti che derivano dall'analisi eseguita nei capitoli successivi.

Descrizione	Visibilità
JADE Main port	locale
Container Local Port	locale
MainBoot Port	locale
Tranche Transfer Local Port	locale
Extra-region Tranche Transfer Local Port	locale

Tabella 2-1 – Peer: requisiti di comunicazione.

[21] A tal proposito si rimanda al CAPITOLO 5.

Descrizione	Visibilità
JADE Main port	locale
Container Local Port	locale
Container Public Port	pubblica
Main Container Local Port	locale
Main Container Public Port	pubblica
Rendezvous Port	pubblica
MainBoot Port	pubblica
Tranche Transfer Local Port	locale
Extra-region Tranche Transfer Local Port	locale

Tabella 2-2 – Superpeer: requisiti di comunicazione.

I due agenti che si occuperanno di rendere possibile le comunicazioni inter-regione sono l'RPA (RendezvousProxyAgent) e l'HA (HoleAgent).

2.5.1 Agente RPA (RendezvousProxyAgent)

L'agente RPA (RendezvousProxyAgent), eseguito da ogni superpeer, svolge essenzialmente due funzioni:

1. effettua un servizio di tipo proxy;
2. funge da rendezvous server per la procedura di TCP Hole Punching.

Ogni peer della regione, anche se il relativo container JADE non è visibile dall'esterno della regione stessa, è in grado di ricevere i messaggi inviati da altre regioni: tali messaggi, infatti, una volta inviati all'agente RPA del superpeer, verranno da esso inoltrati al peer destinatario. Allo stesso modo l'eventuale risposta sarà inviata all'agente RPA della regione in cui si trova il peer mittente e da questo agente verrà inoltrata verso il peer mittente.

L'agente RPA offre anche il servizio di rendezvous per la procedura di TCP Hole Punching: riceve la richiesta dall'agente HA del peer esterno alla regione, rileva l'endpoint pubblico del mittente ed inoltra tale informazione all'agente HA del peer interno alla regione. Per una esemplificazione del processo, si veda il paragrafo *2.5.3*.

2.5.2 Agente HA (HoleAgent)

L'agente HA (HoleAgent) si occupa di eseguire il TCP Hole Punching creando una connessione TCP diretta fra due peer di regioni differenti e interagendo con gli agenti DA e UA che occupano del trasferimento effettivo.

Poiché la procedura in oggetto potrebbe incontrare alcune difficoltà a livello di comunicazione, è necessario prevedere alcuni accorgimenti per gestire gli eventuali casi di guasto e non inficiare il corretto funzionamento del sistema.

2.5.3 Esempi di applicazione

Di seguito vengono presentate due immagini al fine di chiarire quale debba essere il comportamento dei due agenti appena presentati: la *Figura 2-15* (p.45) schematizza la fase di selezione delle fonti, mentre la *Figura 2-16* (p.46) rappresenta la fase di rendezvous seguita dal trasferimento effettivo della tranche.

E' evidente che le due figure non vogliono rappresentare un vincolo per la fase di implementazione: lo scopo è solo quello di delinare i ruoli dei singoli agenti e di fissare le base per i protocolli di comunicazione che verranno studiati, definiti e documentati accuratamente nel seguito della trattazione e precisamente nel *CAPITOLO 5*.

In entrambe le figure lo scenario rappresentato vede due regioni composte da un superpeer e da un peer. Il PEER2 della REGION2 è la fonte della tranche mentre il PEER2 della REGION1 è il peer richiedente.

Selezione delle fonti

Nella prima figura - *Figura 2-15* (p.45) - il DMA del PEER2@REGION1 crea l'agente MA che migra [1] verso il superpeer dell'altra regione[22] e interagisce con l'UMA attendendo la chiamata del ticket.

Quando il ticket viene chiamato, l'MA informa il DMA creatore inviando un messaggio [2] all'agente RPA del superpeer della REGION1 che lo inoltra [3] al PEER2 (L'RPA offre il servizio di proxy tramite il suo behaviour RPAProxyBhv).

[22] Si ricorda che la migrazione inter-regione avviene tramite lo scambio di messaggi (tramite piattaforma JADE) tra il peer di origine e l'agente AMS del superpeer di destinazione. Tale comunicazione è garantita dal fatto che il supereer è raggiungibile direttamente dall'esterno della regione attrraverso le porte utilizzate da JADE.

Il DMA risponde al MA comunicando direttamente con il superpeer della REGION2[23] [4]. L'MA ne dà conferma all'UMA [5] che interrompe l'esecuzione del MA [6] e informa l'agente UA [7].

A questo punto l'agente UA del peer fonte esegue due operazioni [8]:

a) comunica con il DA del peer richiedente sempre tramite l'agente RPA della REGION1 che offre il servizio proxy;
b) invia una richiesta all'agente HA (sempre del peer fonte) affinchè sia pronto a cominciare la procedura di TCP Hole Punching.

Dopo che l'agente RPA inoltra il messaggio al DA del richiedente [9] e dopo che questo risponde al UA della fonte tramite l'RPA della REGION2, ha inizio la fase di rendezvous (*Figura 2-16*, p.46) eseguita dagli agenti HA dei due peer.

Rendezvous & tranche transfer

Nella *Figura 2-16* (p.46) si vede l'agente HA del peer fonte che contatta [1] l'agente RPA (che offre il servizio di rendezvous tramite il suo behaviour RPARendezvousBhv); l'RPA rileva l'endpoint pubblico del peer fonte e lo comunica [2] al HA del peer richiedente nella REGION1; quest'ultimo agente contatta [3] l'RPA del peer fonte che rileva l'endpoint pubblico e lo comunica [4] all'HA del PEER2@REGION2.

A questo punto ciascun peer conosce l'indirizzo e la porta pubblica dell'altro. In modo indipendente i due agenti HA del peer fonte PEER2@REGION2 e del peer richiedente PEER2@REGION1 informano rispettivamenti gli agenti UA e DA che danno il via alla vera e propria procedura di Hole Punching eseguendo i behaviour di tipo client e di tipo server [4], [5], e [6].

Nell'immagine si ipotizza che sia il DA del peer richiedente (tramite il behaviour di tipo client) a tentare per primo la connessione verso l'UA [7]; il tentativo non va a buon fine, ma crea l'associazione tra i due endpoint nel NAT della REGION1.

[23] L'agente MA, una volta migrato verso il superpeer della REGION2, non migra nuovamente verso il PEER2@REGION2 ma resta sul superpeer, rimanendo in grado di ricevere direttamente i messaggi del DMA che lo ha creato.

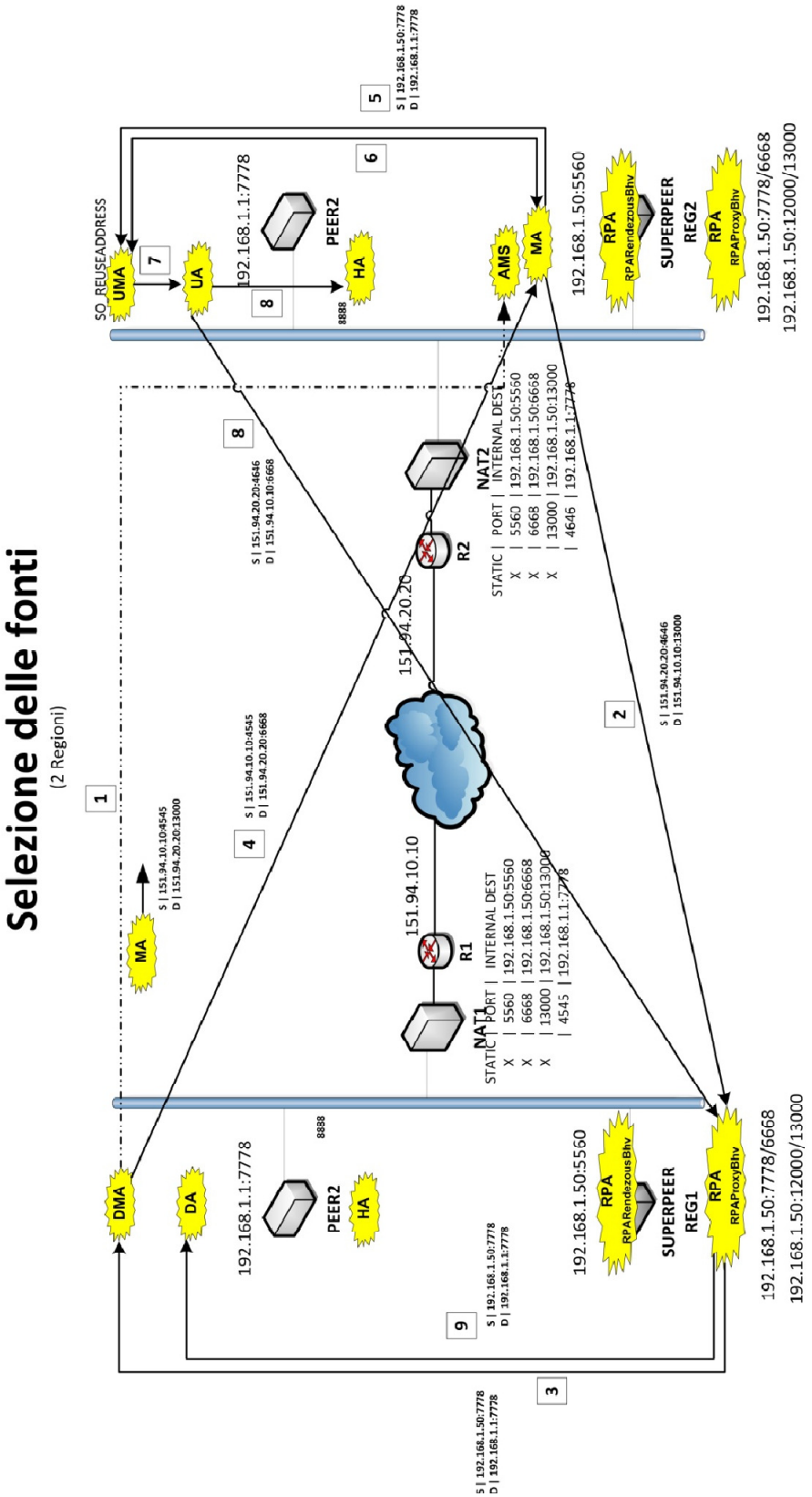


Figura 2-15 – Ipotesi 3: selezione delle fonti. [agenti]

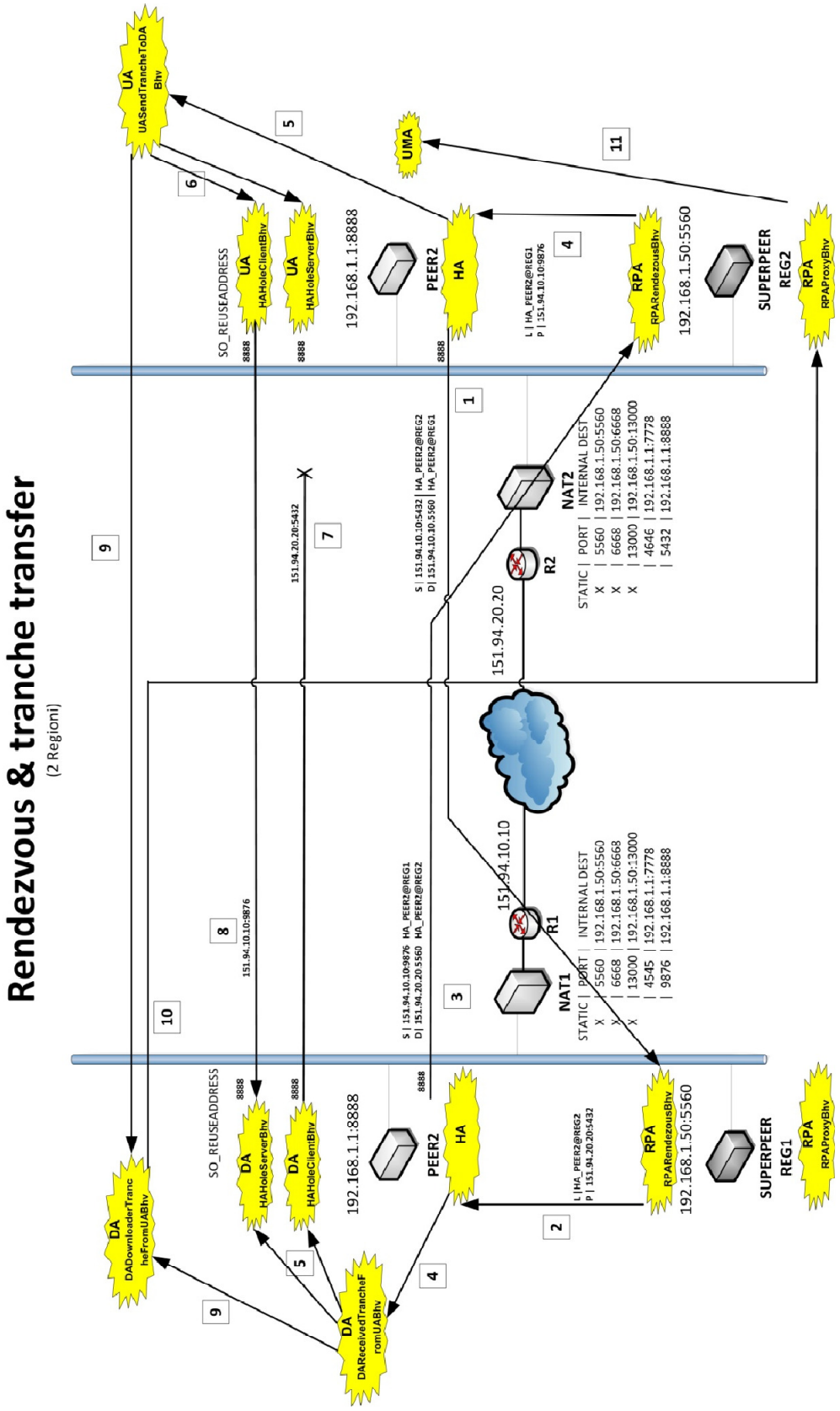


Figura 2-16 – Ipotesi 3: rendezvous e trasferimento delle tranche. [agenti]

Quando l'agente UA della fonte (tramite il behaviour di tipo client) tenta la connessione verso il DA [8], la connessione viene instaurata grazie al behaviour di tipo server in ascolto sul DA ed il trasferimento diretto può aver luogo [9][24].

Infine, a trasferimento ultimato o in caso di errore di trasmissione, il DA sblocca l'agente UMA della fonte inviando un messaggio [10] al RPA della REGION2 che lo inoltra [11] al UMA.

[24] Si ricorda che, per ogni richiesta di trasferimento di una tranche, le comunicazioni a livello di socket del HA verso il RPARendezvousBhv e del DA (UA) verso l'UA (DA) (per il trasferimento effettivo della tranche) devono avvenire attraverso la stessa porta locale, affinchè sia possibile utilizzare la stessa porta pubblica associata dal NAT all'endpoint locale del peer. Ovviamente si sottintende la bontà del NAT che si presume si comporti in modo friendly (cfr. par. *2.3.6*)

CAPITOLO 3

Funzioni di amministrazione

In questo capitolo si individuano i requisiti relativi alle funzioni di amministrazione che il sistema deve essere in grado di offrire e, tramite la formulazione di alcune ipotesi di soluzione, si delineano alcune linee guida per l'implementazione.

3.1 Introduzione

Affinchè il sistema consenta di essere amministrato agevolmente, è necessario che vengano previste alcune funzionalità e che si definiscano alcune primitive relative ai peer ad alle regioni.

Un amministratore deve essere principalmente in grado di:

1. gestire lo stato di un peer;
2. gestire lo stato di una regione;
3. ricevere informazioni sulla struttura della overlay network;
4. gestire i parametri di ogni nodo del sistema (peer, superpeer, server)

Queste funzionalità possono essere offerte tramite la modifica diretta dei parametri su ogni nodo del sistema (4) o attraverso un'interfaccia (testuale o grafica) che permetta di formulare delle richieste e di ricevere delle risposte (1, 2 e 3): in quest'ultimo caso sono da decidere le modalità con cui l'amministratore debba interagire col sistema.

Nel corso della trattazione si valuterà l'eventualità di aggiungere (oltre al peer ed al superpeer) un nuovo tipo di entità al sistema, ossia il server. Si procederà prima ad una breve descrizione della situazione attuale del sistema, poi ad un'analisi più approfondita del problema e delle primitive citate, introducendo anche alcune ipotesi di soluzione ma lasciando i dettagli implementativi ad una successiva trattazione.

3.2 Situazione attuale

Nel momento in cui un peer o un superpeer vengono avviati, sono già a disposizione tutte le informazioni necessarie; tra le altre un peer è a conoscenza:

- del proprio nome identificativo (univoco a livello di regione);
- del nome (univoco a livello globale) della regione in cui si dovrà inserire;
- del nome del superpeer della regione;
- degli indirizzi (ip:porta) pubblici e locali presso cui è possibile raggiungere il superpeer e il peer stesso per comunicazioni relative alla piattaforma Jade (messaggi ACL) e per i trasferimenti delle tranche.

Ogni superpeer, oltre alle informazioni disponibili per ogni peer, conosce l'identificativo globale (GUID) e l'indirizzo pubblico dell'agente AMS presente in ciascuno dei due superpeer ad esso vicini nell'overlay network che attualmente è quindi definita in modo statico a priori.

Ogniqualvolta un peer viene avviato, vengono eseguite le seguenti operazioni:

- creazione del container Jade;
- connessione al main container del superpeer della regione in cui il peer si inserisce;
- creazione e avvio degli agenti.

Un superpeer che viene avviato procede automaticamente alla creazione del main container, degli agenti tipici di un peer e di quelli riservati ai superpeer.

Attualmente un peer non è direttamente raggiungibile dall'esterno della regione di appartenenza, quindi i messaggi provenienti da altre regioni devono essergli inoltrati dal superpeer.

Il vincolo imposto da una struttura come quella attualmente implementata consiste nella limitata flessibilità delle operazioni legate all'avvio/interruzione di un peer, di quelle connesse all'amministrazione di una regione e di quelle legate alla gestione dell'overlay network: non è prevista la possibilità di interrompere l'esecuzione di tutti gli agenti di un peer consentendo al sistema di continuare a funzionare correttamente, né tantomeno è prevista la possibilità di aggiungere/disconnettere un'intera regione.

3.3 Gestire lo stato di un peer

L'amministratore deve poter modificare lo stato di un peer in modo da controllarne l'attività[1]. Lo stato di un peer è strettamente connesso a quello dei suoi agenti. Per ogni singolo agente, Jade prevede i seguenti stati:

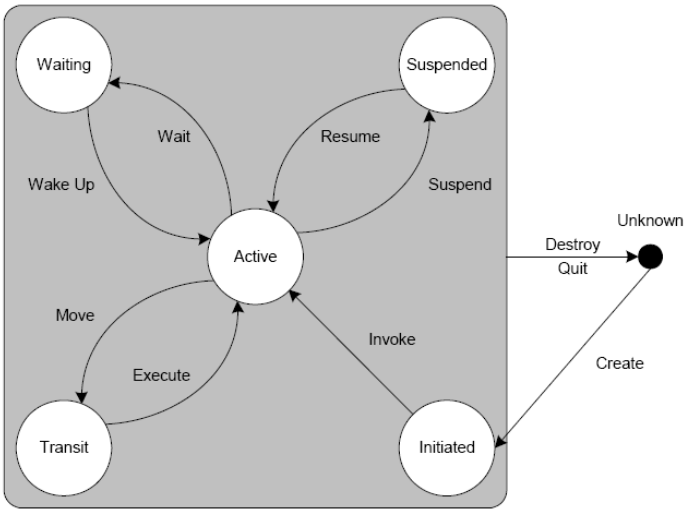


Figura 3-1 – Stati di un Agente

[1] Si noti che qualsiasi operazione effettuata su un peer presuppone la creazione e la connessione al sistema della regione in cui il peer si trova: quindi il superpeer della regione deve essere attivo e funzionante.

- INITIATED : l'agente è stato creato ma non si è ancora registrato presso l'agente AMS; non possiede un nome, né un indirizzo e non ha possibilità di comunicare con gli altri agenti;
- ACTIVE : l'agente si è registrato presso l'AMS e può accedere a tutte le funzionalità di Jade;
- SUSPENDED : l'agente non sta eseguendo nessuna operazione. Il thread su cui era in esecuzione è interrotto e quindi non viene eseguito nessun behaviour ;
- WAITING : l'agente è bloccato e nessun behaviour può essere eseguito; il thread su cui era in esecuzione è bloccato in attesa di essere svegliato dal verificarsi di una condizione (in genere l'arrivo di un messaggio nella coda dell'agente) ;
- DELETED : l'agente è terminato definitivamente. Il thread su cui era in esecuzione ha terminato la sua esecuzione e l'agente non è più registrato presso l'AMS;
- TRANSIT: un agente mobile entra in questo stato quando sta per migrare verso un altro container. Il sistema continua ad accodare i messaggi in arrivo che saranno inviati alla nuova destinazione.

E' indispensabile individuare in quali stati è possibile che un peer si trovi e ciò deve essere fatto alla luce di quali caratteristiche comportamentali un peer deve possedere.

Ad esempio si richiede che un peer possa essere avviato per la prima volta quando non ha a disposizione nessuna tranche di nessun contenuto; oppure che possa essere ripristinato da uno stato di ibernazione, facendo uso delle tranche che è riuscito ad ottenere durante le connessioni avvenute in precedenza: infatti potrebbe essere necessario effettuare delle operazioni di aggiornamento sulla macchina su cui è in esecuzione un peer senza, per questo motivo, voler perdere i progressi fatti fino quel momento.

Un peer deve poter essere interrotto definitivamente, facendo sì che il prossimo avvio venga considerato come un primo avvio del peer, senza nessuna informazione sulla sua vita precedente; infine si richiede che un peer possa essere sospeso temporaneamente, interrompendo lo svolgersi delle sue attività ma mantenendolo ancora connesso al sistema.

3.3.1 Stati

Di seguito si elencano gli stati in cui è possibile che si trovi un peer:

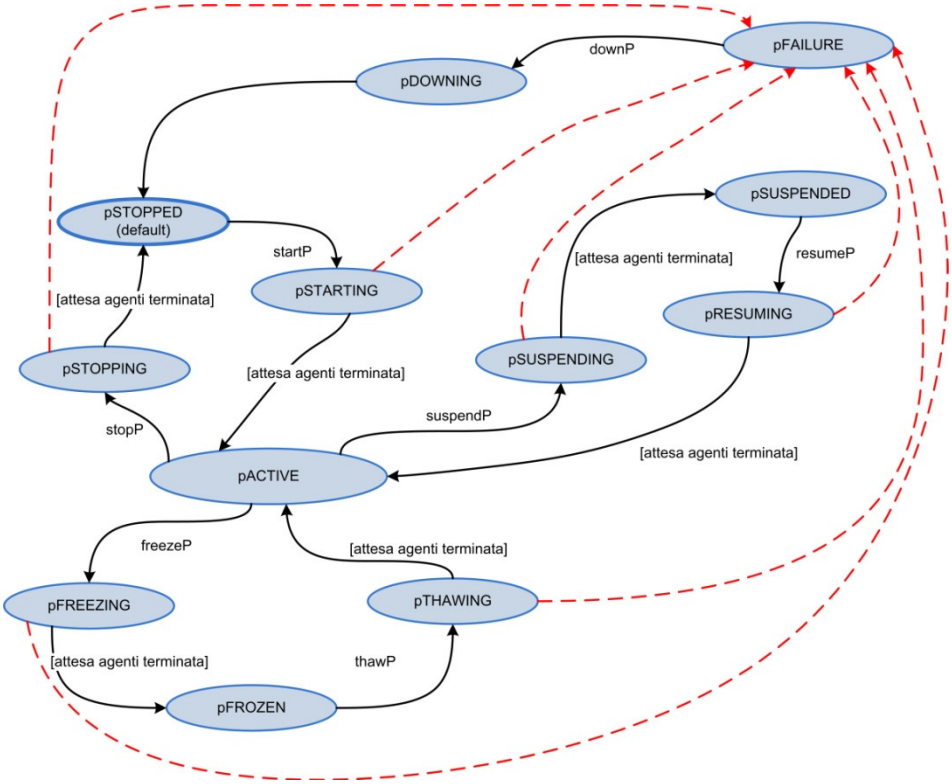


Figura 3-2 – Stati di un peer.

pACTIVE

Il peer è avviato e connesso al sistema; gli agenti stanno svolgendo le loro attività.

pSUSPENDING

Il peer sta passando dallo stato pACTIVE allo stato pSUSPENDED. I singoli agenti stanno attendendo di essere in uno stato appropriato per sospendere la propria esecuzione.

pSUSPENDED

Le attività del peer sono sospese: gli agenti (come il container Jade che li contiene) sono in esecuzione ma non stanno svolgendo nessuna attività; la coda dei messaggi in arrivo verso ogni agente continua ad essere popolata dai nuovi messaggi ma nessun agente li processa. Il peer è connesso al sistema, viene riconosciuto come

possibile fonte per le tranche, è raggiungibile come se fosse nello stato pACTIVE ma non esegue attività di alcun genere eccetto l'attesa del messaggio resumeP.

pRESUMING

Il peer sta passando dallo stato pSUSPENDED allo stato pACTIVE.

pFREEZING

Stato di passaggio dallo stato pACTIVE verso lo stato pFROZEN. In questo stato devono essere liberate le risorse impegnate dai vari agenti, come la connessione al database o le connessioni TCP instaurate verso altri peer.

pFROZEN

Le attività del peer sono interrotte: gli agenti sono nello stato cAGENT_STATE_DELETED, il container del peer è ancora in esecuzione ed è connesso al mainContainer del superpeer, ma i messaggi indirizzati agli agenti del peer non vengono recapitati, le informazioni sulle tranche possedute vengono rimosse dal database del superpeer (quindi il peer non viene riconosciuto come fonte per le tranche che possiede). L'unica attività del peer consiste nell'attesa del messaggio thawP.

Quando il peer è in questo stato, le informazioni relative alle tranche già possedute sono memorizzate in modo persistente per poter essere recuperate in un secondo momento così che, quando il peer tornerà nello stato pACTIVE, il superpeer potrà nuovamente considerarlo una fonte valida per le tranche in suo possesso ed il peer potrà riprendere la ricerca delle sole tranche mancanti.

Se possibile lo stato di ogni singolo agente deve essere mantenuto in memoria e anche le code dei messaggi devono essere salvate per essere poi ripristinate al successivo avvio del peer, in modo tale che le attività del peer possano riprendere esattamente da dove erano state sospese[2].

[2] Naturalmente questo requisito deve essere valutato in relazione alle caratteristiche di JADE e degli eventuali componenti aggiuntivi.

pTHAWING

Il peer sta passando dallo stato pFROZEN allo stato pACTIVE.

pSTOPPING

Stato che precede immediatamente lo stato pSTOPPED a partire dallo stato pACTIVE. In questo stato devono essere liberate le risorse impegnate dai vari agenti, come la connessione al database o le connessioni TCP instaurate verso altri peer.

pSTOPPED

Valgono le stesse considerazioni fatte per lo stato pFROZEN, ma non viene conservata nessuna informazione sulle tranche già in possesso del peer né sullo stato dei vari agenti, né sulle code dei messaggi di ogni agente. Quindi al successivo avvio, il peer comincerà a cercare tutte le tranche dei contenuti da visualizzare e il superpeer non ripristinerà le informazioni sulle tranche ottenute precedentemente dal peer.

pSTARTING

Il peer passa dallo stato pSTOPPED allo stato pACTIVE. In questo stato gli agenti vengono creati ed avviati. Se il peer è anche superpeer devono essere eseguite le operazioni previste per questo tipo di nodo.

pFAILURE

Un peer che si trova in questo stato ha rilevato una condizione di guasto durante l'esecuzione di una primitiva di amministrazione e non è più in grado di svolgere le sua attività in modo consueto. E' necessario che riceva una primitiva che lo riporti nello stato pSTOPPED ripristinando lo stato di tutti gli altri agenti ed aggiornando le informazioni mantenute dal superpeer.

pDOWNING

Stato di passaggio dallo stato pFAILURE allo stato pSTOPPED.

Si illustrano di seguito le primitive che possono essere utilizzate per un peer, analizzando alcuni requisiti di modello e rimandando ad un secondo momento l'analisi dei particolari legati all'implementazione.

3.3.2 Primitive

Nel descrivere le primitive di stato dei peer, per il momento si presuppone che ciascun peer a cui è destinata una primitiva sia sempre raggiungibile e riesca ad elaborare la primitiva stessa.

Inizialmente si era pensato di associare all'arresto di un peer anche l'interruzione di tutti i suoi agenti e la disconnessione del container dal main container del superpeer. Questa ipotesi si è rivelata poco praticabile in quanto molte operazioni legate alle fasi di sospensione/ibernazione/arresto e ripristino/"scongelamento"/avvio presuppongono che la piattaforma JADE sia in esecuzione.

Inoltre veicolare i messaggi per l'avvio dei servizi di JADE verso peer non raggiungibili direttamente via TCP (a causa dei NAT) e gestire un protocollo adeguato sarebbero stati solo elementi che avrebbero appesantito la struttura del nuovo sistema. Per di più non era richiesto in alcun modo che il container di un peer non in esecuzione dovesse necessariamente essere disconnesso dal sistema.

Dunque si è giunti alle seguenti conclusioni:

- nel momento in cui un peer viene attivato per la prima volta, il suo container viene connesso al main container del superpeer, gli agenti di supporto vengono avviati e il peer rimane nello stato pSTOPPED; solo a partire da questo momento il peer potrà ricevere la primitiva startP;
- eventuali agenti di supporto possono essere eseguiti all'interno del container a prescindere dallo stato del peer;
- deve essere permesso l'utilizzo di thread esterni alla piattaforma JADE per l'avvio della piattaforma stessa e la relativa gestione (anche a livello degli agenti in essa presenti);
- la connessione del container al maincontainer rimane valida finchè non viene interrotta da fattori esterni, ad esempio in seguito ad un problema di comunicazione o all'interruzione della JVM su cui il peer è in esecuzione.

Si noti che l'esecuzione di una primitiva, a prescindere che sia di tipo peer o di tipo regione, richiede che vengano restituite delle informazioni al nodo "creatore" da cui la primitiva è stata lanciata. Dal momento che i peer non sono direttamente raggiungibili dall'esterno, nel caso in cui la primitiva

sia lanciata da uno di tali peer, i messaggi di risposta dovrebbero essere inoltrati dall'agente RPA della regione affinchè raggiungano il peer creatore.

Di seguito vengono presentate tutte le primitive che possono essere eseguite su un peer.

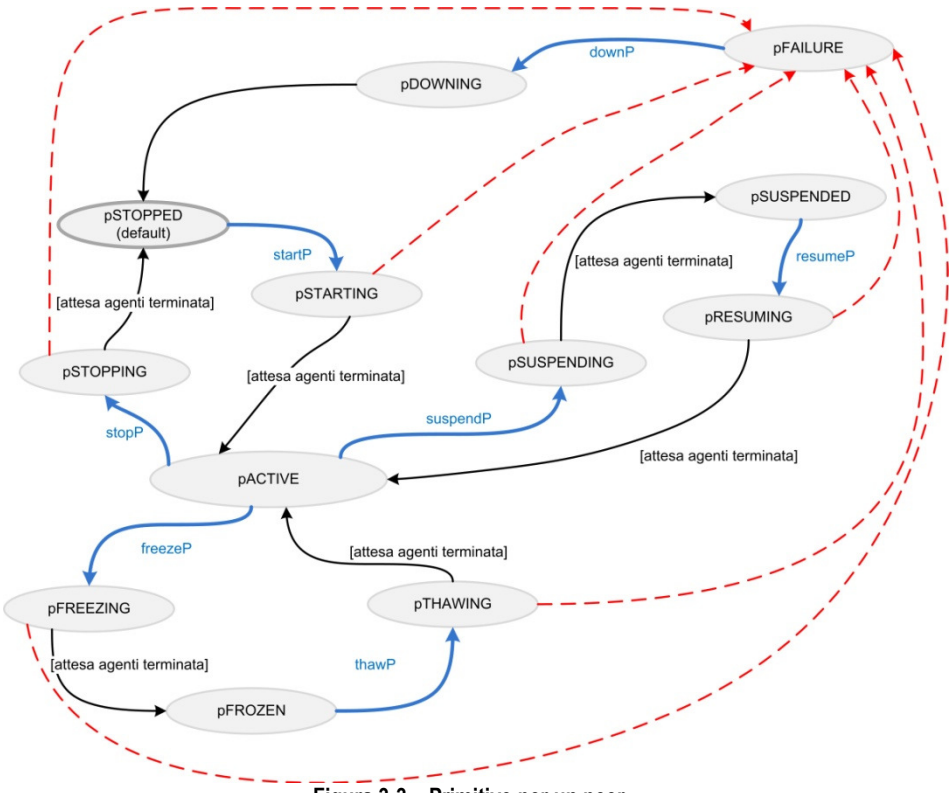


Figura 3-3 – Primitive per un peer.

startP

La primitiva avvia un peer che si trovava nello stato pSTOPPED.

Per avviare un peer all'interno di una regione potrebbe essere sufficiente far sì che per ogni peer potenzialmente avviabile venga lasciato in esecuzione sempre almeno un processo (eventualmente rappresentato da un agente)[3] che si occupi di ricevere il messaggio di avvio (startP) e di eseguire le operazioni necessarie (come la creazione e l'avvio degli agenti).

Alcune operazioni (attualmente racchiuse nel Main.java) devono essere eseguite prima di ogni altra per permettere la ricezione della primitiva startP (ad esempio: la creazione del container, la connessione al main

[3] Un peer che si trova nello stato pSTOPPED mantiene in esecuzione il container.

container del superpeer, l'avvio dei servizi del core di JADE) e quindi è indispensabile che vengano eseguite da un altro processo (non agente) che si occupi di avviare la piattaforma JADE sul singolo peer. Il messaggio di startP, nel caso la configurazione di rete non permetta al peer di essere raggiunto direttamente dall'esterno della sua regione, dovrà essere inviato al superpeer il quale lo inoltrerà al peer da avviare.

Ovviamente il presupposto per l'avvio di un peer è che la regione all'interno della quale verrà avviato sia già connessa al sistema e che quindi il superpeer sia già in funzione.

Al termine della fase di avvio, il superpeer dovrà essere informato del cambiamento di stato del peer.

suspendP

Attraverso questa primitiva il peer viene sospeso e passa dallo stato pACTIVE allo stato pSUSPENDED.

Dal momento che il peer era attivo prima della ricezione del messaggio, le operazioni associate alla sospensione possono essere svolte avvalendosi delle funzionalità offerte da Jade. Gli agenti del peer dovranno sospendere le loro attività, pronti a poterle riprendere esattamente da dove le avevano interrotte e il superpeer dovrà essere informato del cambiamento di stato.

Il peer rimane in esecuzione ma non svolge nessuna attività eccetto quella di attesa del messaggio resumeP.

freezeP

La primitiva comporta un cambiamento di stato da pACTIVE a pFROZEN.

Il peer che viene congelato salva le seguenti informazioni in modo da poter riprendere la propria esecuzione da dove l'aveva interrotta quando sarà avviato nuovamente:

- tranche già ottenute;
- stato di ogni agente;
- contenuto delle code dei messaggi di ogni agente (se possibile[4]).

[4] Compatibilmente con le funzionalità offerte da JADE.

Per ogni peer si deve introdurre un agente che si occupi di:

1. ricevere il messaggio di freezeP;
2. informare tutti i restanti agenti del peer in modo che terminino la loro esecuzione appena possibile (il comportamento di ogni agente dovrà essere definito in base alla tipologia dell'agente stesso, allo stato in cui si trova il behaviour attualmente in esecuzione, allo stato delle comunicazioni instaurate con altri agenti e alla presenza di eventuali agenti mobili creati dall'agente e migrati presso altri peer);
3. salvare le informazioni necessarie per un futuro ripristino del peer;
4. informare il superpeer affinchè rimuova il peer dalla lista dei tranche_owner per ogni tranche posseduta;
5. informare il superpeer del cambiamento di stato.

Nel caso in cui non sia possibile salvare il contenuto delle code dei messaggi di ogni agente, al risveglio del peer ogni agente riprenderà le propria attività a partire dallo stato di default. In questo modo la primitiva freezeP diventa molto simile a stopP, con l'unica differenza che il peer tiene traccia delle tranche ottenute fino al momento del congelamento e, al successivo avvio, non ne effettuerà nuovamente il download.

resumeP

Questa primitiva riattiva un peer che era nello stato pSUSPENDED portandolo nello stato pACTIVE.

Gli agenti del peer vengono risvegliati: le code dei messaggi sono state aggiornate durante lo stato pSUSPENDED e lo stato degli agenti è rimasto invariato. Dopo che tutti gli agenti hanno ripreso le loro attività, il superpeer viene informato del cambiamento di stato.

thawP

Questa primitiva riporta un peer nello stato pACTIVE dal precedente stato pFROZEN.

Ogni peer potenzialmente riattivabile dopo un'ibernazione ha in esecuzione, come nel caso della primitiva startP, un processo (agente) che si occupa di ricevere questo messaggio.

Al momento del risveglio del peer, il container è già creato e connesso al mainContainer del superpeer; tutti gli agenti vengono creati e avviati a partire dalle informazioni memorizzate in modo persistente durante la fase di freeze. In questo modo gli agenti riprendono la loro esecuzione

a) esattamente dal punto in cui l'avevano interrotta (se è stato possibile salvare le code dei messaggi);
b) dallo stato di default (se non è stato possibile salvare le code dei messaggi), tenendo però comunque traccia delle tranche già in possesso del peer e, più in generale, considerando che l'avvio in corso segue lo stato di ibernazione FROZEN.

stopP

Questa primitiva porta il peer dallo stato pACTIVE o allo stato pSTOPPED

Per ogni peer è sensato introdurre un agente che si occupi di:

1. ricevere il messaggio di stopP;
2. informare tutti i restanti agenti del peer in modo che terminino la loro esecuzione appena possibile (il comportamento di ogni agente dovrà essere definito in base alla tipologia dell'agente stesso, allo stato in cui si trova il behaviour attualmente in esecuzione, allo stato delle comunicazioni instaurate con altri agenti e alla presenza di eventuali agenti mobili creati dall'agente e migrati presso altri peer);
3. informare il superpeer affinchè rimuova il peer dalla lista dei tranche_owner per ogni tranche posseduta;
4. informare il superpeer del cambiamento di stato.

La primitiva stopP richiede, così come freezeP, che gli agenti vengano completamente rimossi dal container del peer. Questa operazione può essere eseguita da un agente di supporto che rimane in esecuzione all'interno del container o da un thread esterno alla piattaforma JADE che interagisce con essa.

Nel caso il peer da interrompere sia in realtà il superpeer, si potrebbe interpretare questa richiesta come una richiesta di disconnessione dell'intera regione e di non procedere quindi all'elezione di un nuovo superpeer (procedura alquanto laboriosa poiché sarebbe necessario creare un nuovo main container sul nuovo superpeer, informare tutti i peer della regione affinchè connettano il proprio container al nuovo main container,

trasferire tutte le informazioni della regione verso il nuovo superpeer, informare i superpeer vicini nell'overlay network, garantire che il nuovo superpeer sia visibile dall'esterno così come il precedente superpeer, etc..).

downP

Questa primitiva porta il peer dallo stato pFAILURE o allo stato pSTOPPED.

Le operazioni svolte dalla primitiva downP sono simili a quelle previste per stopP; tuttavia, siccome il peer si trova in uno stato che identifica il palesarsi di un guasto, è necessario prevedere di poter interrompere bruscamente l'esecuzione di tutti gli agenti del container (ad eccezione di quelli di supporto) senza attendere che si trovino in uno stato appropriato.

3.4 Gestire lo stato di una regione

Poter gestire lo stato di una regione significa poter controllare le attività di tutti i peer al suo interno e poter modificare la struttura della rete di overlay in cui si trovano tutti i superpeer connessi al sistema: ogni regione viene inserita nella rete al momento della sua creazione e rimuoverla significa modificare la overlay network e privare il sistema di eventuali fonti.

Assume un'importanza sostanziale decidere come debba avvenire la gestione dello stato delle regioni del sistema: una soluzione potrebbe essere quella di far riferimento ad un server che abbia a disposizione le informazioni aggiornate sulla struttura complessiva della rete logica su cui è basato il sistema.

3.4.1 Definizione del server

Per poter aggiungere una regione al sistema si deve innanzitutto modificare l'overlay network in modo che il superpeer della nuova regione sia in grado di interagire con gli altri superpeer.

Affinché ciò sia possibile, è necessario che sia presente un'entità la quale abbia informazioni sulla struttura di tutta l'overlay network. Si potrebbe definire un server che ogni superpeer, durante le fasi di avvio, dovrebbe contattare per ottenere i riferimenti ai superpeer vicini nella overlay network (l'indirizzo e il GUID dell'agente AMS).

Nel caso in cui un superpeer non sia più raggiungibile (verifica effettuabile ad esempio tramite dei probe message) o abbia effettuato la disconnessione delle propria regione dal sistema, sarà sempre compito del server mantenere coerente la struttura della rete, notificando ad ogni superpeer eventuali aggiornamenti per esso rilevanti (riguardanti, cioè, i superpeer adiacenti).

Il server potrebbe essere rappresentato (a) dal superpeer di una regione (b) dal superpeer di una regione priva di altri peer (costituita dal solo superpeer) (c) dal superpeer di una regione priva di altri peer, raggiungibile da ogni peer del sistema ma esterna alla overlay network.

a) in questo caso l'efficienza delle funzioni del server (che tra l'altro detiene tutte le tranche di tutti i contenuti che ogni peer dovrà ottenere) potrebbe diminuire a causa delle normali richieste formulate dai peer della regione di cui il server è anche superpeer;
b) in questo caso il server dovrebbe rispondere solo alle richieste provenienti da altre regioni ma, come nel caso precedente, in caso di guasto del server verrebbe danneggiata direttamente la struttura della overlay network;
c) in quest'ultimo caso un guasto del server non inficerebbe la struttura della overlay network (almeno fino al successivo guasto di uno dei superpeer) e sarebbe possibile evitare che il server venga considerato come fonte plausibile per ogni ricerca effettuata dai superpeer (così come avviene nei due scenari precedenti).

Si ritiene che la soluzione (c) sia quella più vantaggiosa[5].

Per limitare le conseguenze di un guasto del server sulla struttura della overlay network, potrebbe essere sensato replicare su alcuni superpeer le informazioni sulla struttura della rete. Una strategia più efficace potrebbe consistere nel mantenere una conoscenza distribuita della struttura complessiva o più semplicemente, per ogni superpeer, i riferimenti a superpeer vicini fino ad un livello predeterminato o stabilito dinamicamente.

Qualunque sia la scelta effettuata, dovrebbe essere implementata una strategia per la diffusione iniziale delle prime tranche ai superpeer in base

[5] In fase di implementazione potrebbe essere sensato per ragioni di semplicità implementare il server in modo che faccia comunque parte della overlay network. Ottenere una overlay network priva del server non richiederebbe, a quel punto, eccessivo sforzo realizzativo.

alle esigenze dei peer della regione[6]: in caso contrario il server, nel primo periodo di attività del sistema, verrebbe inondato dalle richieste di tutti i peer che, a seguito delle ricerche extra-regione, individuerebbero nel server stesso l'unica fonte per le tranche richieste.

La presenza di un server non deve essere fuorviante: l'architettura del sistema rimane peer-to-peer e la struttura gerarchica che si viene a creare ha il solo scopo di gertire al meglio quelle attività che richiedono l'interazione fra regioni distinte come la ricerca o l'inoltro delle primitive di amministrazione per peer e regioni.

Il server funge da contenitore per tutti i contenuti potenzialmente fruibili da ciascun peer che ne faccia richiesta e svolge l'attività di coordinatore per quanto riguarda la gestione della struttura della overlay network e l'inoltro delle primitive di amministrazione originate dall'interno di una regione e indirizzate verso altre regioni o peer di altre regioni.

3.4.2 Stati

rADDED

La regione è attiva ed inserita all'interno del sistema. Il superpeer svolge le proprie attività e i peer della regione che sono stati attivati possono eseguire i loro compiti.

rREMOVING

Stato che precede immediatamente lo stato rREMOVED a partire dallo stato rACTIVE.

Il superpeer deve prima inoltrare la primitiva stopP a tutti i peer della regione che attualmente non si trovano già nello stato pSTOPPED e, solo successivamente, interrompere la propria esecuzione passando nello stato rREMOVED.

rREMOVED

In questo stato la regione non è connessa al sistema, quindi non è presente all'inerno della overlay network. Il mainContainer ed il container del superpeer sono in esecuzione; tutti i peer sono nello stato pSTOPPED e

[6] Il server conosce il palinsesto di ogni regione.

tutti gli agenti dei peer e del superpeer della regione non sono in esecuzione (ad eccezione di quelli di supporto).

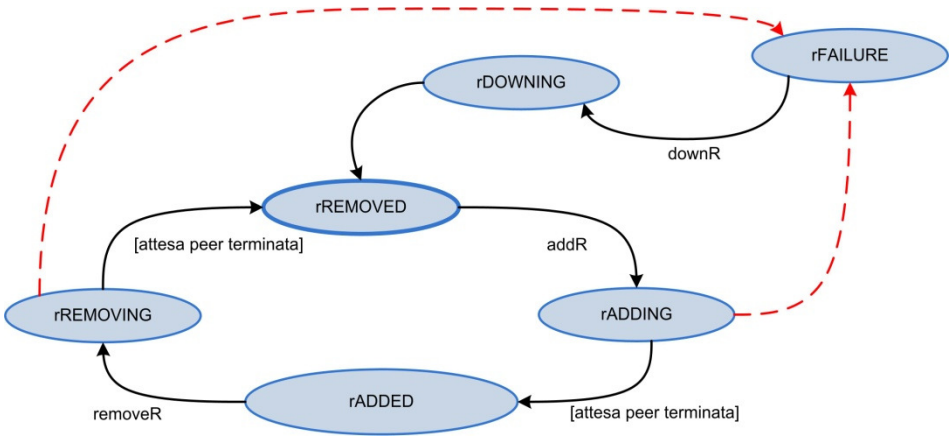


Figura 3-4 - Stati di una regione.

rADDING

Stato di transizione dallo stato rREMOVED allo stato rADDED.

rFAILURE

Una regione si trova in questo stato quando il superpeer ha rilavato un errore durante l'esecuzione di una primitiva di amministrazione a livello di regione. Dal momento che non è detto che sia possibile conoscere con certezza lo stato di ogni peer della regione, l'unica operazione da compiere è il ripristino tramite la primitiva downR.

rDOWNING

La regione sta tornando allo stato rREMOVED dallo stato rFAILURE.

3.4.3 Primitive

Analogamente al caso dei peer (cfr. par. *3.3.2*), anche per le regioni si suppone che:

- nel momento in cui un superpeer viene attivato per la prima volta, verranno avviati il maincontainer ed il container, verranno avviati gli agenti di supporto e la regione rimarrà nello stato rREMOVED; solo a partire da questo momento il superpeer potrà ricevere la primitiva addR;

- eventuali agenti di supporto potranno essere eseguiti all'interno del container a prescindere dallo stato del peer e della regione;
- l'utilizzo di thread esterni alla piattaforma JADE dovrà essere permesso per l'avvio della piattaforma stessa e la relativa gestione (anche a livello di stato degli agenti in essa presenti).

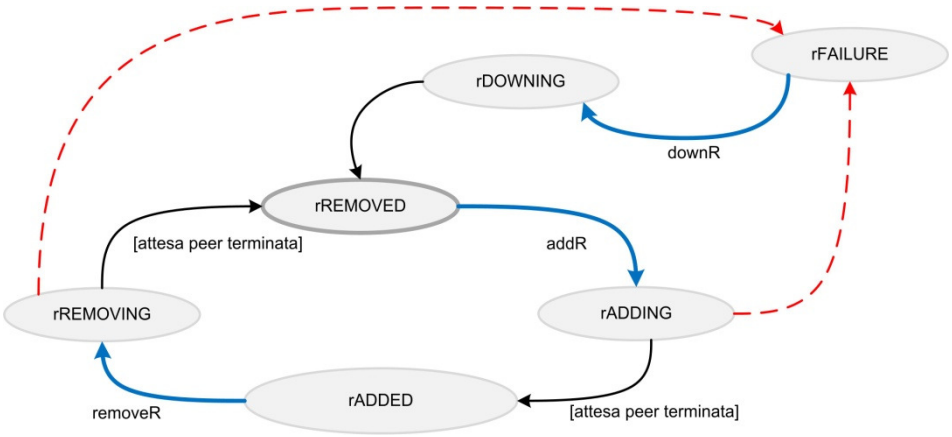


Figura 3-5 – Primitive per una regione.

addR

L'aggiunta di una regione al sistema consiste nell'avvio di un nuovo superpeer che si inserisca tra altri due superpeer all'interno della overlay network. I futuri vicini del superpeer appena aggiunto vengono informati dal server del cambiamento della overlay network; sospendono momentaneamente le operazioni che coinvolgono i vicini e le riprendono considerando il nuovo superpeer.

Come nel caso di un peer, anche per l'avvio di un superpeer potrebbe rivelarsi sufficiente mantenere in esecuzione un processo che si occupi di ricevere l'input di avvio e la posizione all'interno della overlay network (ossia dei GUID e degli indirizzi dei superpeer adiacenti): una volta ricevuto il messaggio, il superpeer informa il server del cambiamento di stato e procede con le normali attività.

Nessun vincolo è imposto sui peer della regione: quando il superpeer riceverà il comando startP dal server per un determinato peer, allora quel peer sarà messo in esecuzione.

removeR

Disconnettere una regione dal sistema è un'operazione riconducibile ad interropere tutti i peer di una regione e, per ultimo, anche il superpeer; deve, inoltre, essere informato il server che provvederà a ristrutturare l'overlay network informando i superpeer adiacenti alla regione che verrà rimossa. Le operazioni necessarie vengono svolte dal superpeer: lo stesso agente citato in precedenza e che si occupa dell'interruzione di un peer, si occupa di far interrompere le attività di tutti i peer della regione.

downR

La primitiva riporta nello stato rREMOVED una regione che si trova nello stato rFAILURE, ripristinando lo stato di ogni peer della regione (downP). Con questa primitiva deve essere anche aggiornata la overlay network, da cui dve essere rimossa la regione in questione.

3.4.4 Considerazioni

Come accennato in precedenza, l'esecuzione di una primitiva, che sia indirizzata ad un peer o sia rivolta ad una regione, richiede che vengano restituite delle informazioni al nodo "creatore" da cui la primitiva stessa è stata lanciata.

		DESTINATARIO					
		PEER (locale)	PEER (stessa regione del mittente)	PEER (altra regione)	REGIONE (stessa regione del mittente)	REGIONE (altra regione)	SERVER
MITTENTE	PEER	Sì	Sì	No	No	No	No
	SUPERPEER	Sì	Sì	Sì	Sì	Sì	No
	SERVER	Sì	- ([7])	Sì	Sì	Sì	Sì

Tabella 3-1 – Vincoli per l'esecuzione delle primitive di amministrazione.

[7] Come stabilito nel paragrafo 3.4.1, non si prevede la presenza di altri peer oltre al server nella regione del server.

Dal momento che si ritiene poco sensato poter eseguire le primitive di amministrazione solo dal server, si decide di dotare il sistema della possibilità di eseguirle da più punti.

Quindi si prevede di realizzare un insieme di componenti che permettano, nello specifico, di lanciare le primitive indirizzate a peer e regioni secondo la Tabella 3-1 (p.66).

3.5 L'add-on Persistence

JADE dispone di un add-on chiamato *Persistence* liberamente ottenibile dal sito ufficiale.

Questo add-on offre un sistema di memorizzazione persistente degli agenti e dei container e garantisce piena compatibilità con diversi sistemi DBMS relazionali[8].

3.5.1 La libreria Hibernate

Viene utilizzata principalmente la libreria *Hibernate*[9] che si occupa di garantire la portabilità tra i sistemi DBMS, gestire i problemi di efficienza, definire un formato basato su XML per la persistenza delle strutture dati (e la relativa mappatura nelle tabelle del database) e fornire un'ApplicationProgrammigInterface (API).

Hibernate, quindi, permette di definire, memorizzare in modo persistente e gestire qualsiasi struttura dati che può essere definita in Java con il minimo impatto sul codice sorgente.

3.5.2 La struttura

Grazie alle funzionalità di Hibernate, l'add-on Persistence consente agli sviluppatori di salvare (e ripristinare) non solo lo stato dei container e degli agenti JADE - comprese le code dei messaggi -, ma anche eventuali classi

[8] Purtroppo quello utilizzato dal prototipo attuale - SQLite (http://www.sqlite.org/) - non è supportato. In alternativa, come consigliato dagli stessi creatori di JADE, è possibile utilizzare HyperSQL Database (http://hsqldb.sourceforge.net), una implementazione open-source basata unicamente su java di un DBMS relazionale che può funzionare semplicemente come libreria o come server stand-alone.

[9] http://www.hibernate.org

specifiche (la mappatura tra classi e dabatase viene stabilita da un file XML).

Gli agenti ed i container vengono salvati all'interno di uno o più repository[10] che possono essere dislocati in un punto qualsiasi della piattaforma JADE (questa caratteristica, come si analizzerà più avanti, si rivela molto utile per il raggiungimento degli scopi del sistema da realizzare).

Le descrizioni dei repository disponibili sono mantenute in un repository particolare chiamato *meta-repository* che viene creato automaticamente da Persistence; se non viene definito alcun repository, ne viene creato sempre automaticamente uno chiamato JADE-DB adatto alla memorizzazione di agenti e container.

In *Figura 3-6* è possibile osservare una configurazione tipica di Persistence in cui i componenti presenti sui vari container della piattaforma puntano allo stesso meta-repository (ed allo stesso file di configurazione per Hibernate) ed è presente solo il repository di default JADE-DB per la memorizzazione degli agenti.

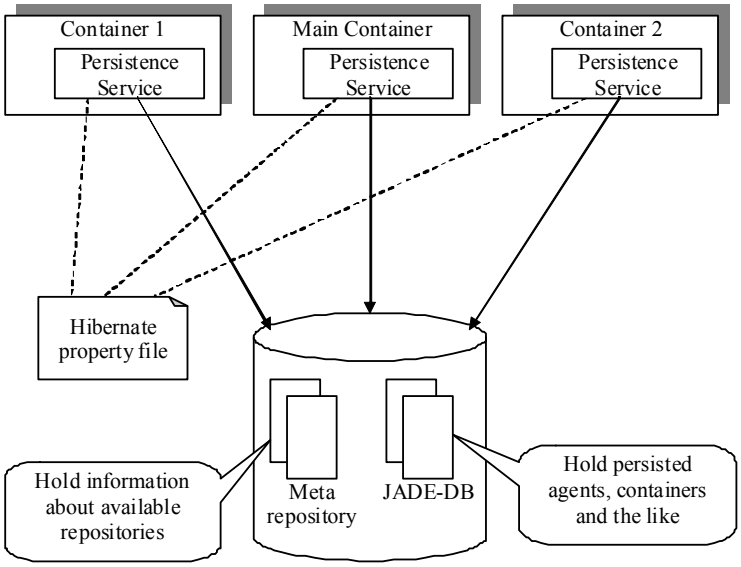


Figura 3-6 - add-on Persistence: dislocazione dei repository.

[10] Un repository è un insieme di tabelle all'interno di un database.

3.5.3 Le primitive

Le primitive messe a disposizione da Persistence sono le seguenti:

- **Save-Agent**: salva un'istantanea dello stato dell'agente e delle code dei messaggi all'interno di uno specifico repository appena il controllo è restituito allo scheduler di JADE dal behaviour attualmente in esecuzione. Ovviamente l'agente deve essere serializzabile o comunque reso tale (deve implementare, quindi, l'interfaccia java.io.Serializable).
- **Load-Agent**: crea un *nuovo* agente, leggendo lo stato iniziale dal repository.
- **Reload-Agent**: resetta lo stato attuale dell'agente caricando quello memorizzato all'interno del repository.
- **Delete-Agent**: rimuove dal repository lo stato dell'agente salvato in precedenza.
- **Freeze-Agent**: salva un'istantanea dello stato dell'agente e ne interrompe l'esecuzione senza eliminarlo. Viene attivato un meccanismo che raccoglie su un dato container i messaggi destinati all'agente mentre si trova nello stato frozen (il container può essere diverso da quello in cui si trovava l'agente ibernato); l'agente apparirà come se fosse in esecuzione su questo container ed i messaggi ricevuti verranno memorizzati all'interno del repository.
- **Thaw-Agent**: ripristina il normale funzionamento di un agente precedentemente congelato, leggendo il suo stato dal repository ed inoltrandogli tutti i messaggi ricevuti durante il periodo in cui era congelato.
- **Save-Container**: salva lo stato del container all'interno del repository compresi nome ed indirizzi MTP del container e tutti gli agenti in esso contenuti.
- **Load-Container**: carica un intero container dal repository: prima vengono interrotti tutti gli agenti attualemnte in esecuzione e vengono disabilitati tutti gli endpoint MTP; poi viene ripristinato tutto dal repository.
- **Delete-Container**: elimina il salvataggio del container dal repository.

Esecuzione delle primitive

E' possibile eseguire le primitive appena elencate in tre modi:

1. tramite le API fornite dall'add-on Persistence: un'interfaccia `PersistenceHelper`[11] fornisce agli agenti il modo di eseguire le primitive di tipo save, reload o freeze;

2. inoltrando una richiesta all'agente AMS della regione affinchè esegua una Action (`jade.content.onto.basic.Action`) secondo l'ontologia specifica di Persistence (definita nel package `jade.domain.persistence`);

3. attraverso l'interfaccia grafica fornita da Persistence.

3.5.4 Considerazioni

Le primitive offerte da questo add-on di JADE si rivelano molto utili per gli scopi del sistema che si andrà a realizzare. In particolare Save-Agent e Load-Agent assumono una rilevanza notevole in quanto gestiscono anche le code dei messaggi di ogni agente.

Le primitive Freeze-Agent e Thaw-Agent sono potenzialmente complete ma vanno oltre le specifiche necessità del nuovo sistema: utilizzarle per implementare le primitive suspendP e resumeP (paragrafo *3.3.2*) si rivelerebbe eccessivo, in quanto per la sospensione di un peer è sufficiente sospendere le attività degli agenti, in modo che i messaggi in arrivo vengano comunque recapitati ma non processati.

Utilizzare Freeze-Agent e Thaw-Agent dell'add-on Persistence per realizzare le primitive freezeP e thawP potrebbe essere una soluzione; ciò, tuttavia, siccome l'ibernazione (freezeP) di un peer comporta il congelamento (Freeze-Agent) di tutti i suoi agenti, potrebbe rappresentare un problema: infatti sarebbe necessario congelare (Freeze-Agent) molti agenti per l'ibernazione (freezeP) di un solo peer e il numero aumenterebbe molto se la richiesta fosse di ibernare più agenti, sovraccaricando in questo modo i container che andranno ad ospitare la versione congelata dei vari agenti, memorizzando all'interno del database i messaggi in arrivo.

Le primitive Save-Container e Load-Container rivelano una sostanziale efficacia ma memorizzano senza distinzione tutti gli agenti presenti in un dato momento all'interno del container (anche eventuali agenti migrati

[11] Un oggetto di tipo `PersistenceHelper` può essere ottenuto tramite un metodo della classe Agent con parametro `PersistenceService.NAME`.

temporaneamente) e lasciano poco margine per la gestione delle dipendenze fra gli agenti: ad esempio l'agente RSMA (che gestisce lo stato delle risorse) per poter interrompersi in seguito ad una primitiva stopP deve prim attendere che tutti gli agenti che lo contattano per aggiornare lo stato di una risorsa abbiano completato l'esecuzione della stessa primitiva stopP.

Il salvataggio individuale di ogni agente assicura un maggior controllo sia del momento in cui può avvenire, sia delle specifiche necessità di ogni agente.

3.6 Primitive di amministrazione: ipotesi di soluzione

Descrivendo le primitive introdotte di seguito si assume che la struttura interna di una regione e quindi di tutti i peer in essa presenti sia definita coerente tramite i parametri da impostare prima di avviare qualsiasi peer: ad esempio il nome e gli indirizzi pubblici e privati del superpeer e del server o lo stesso nome del peer devono essere validi nel momento in cui un peer viene avviato.

Per il momento si suppone, inoltre, che non si verifichino errori di comunicazione, perdita di messaggi o guasti.

Si predispongono su ogni elemento del sistema - peer, superpeer o server - un thread MainBoot e un agente PeerManagementAgent (PMA) con compiti diversi a seconda che vengano avviati da un peer, da un superpeer o dal server. Nei successivi paragrafi si analizzaranno questi due componenti e, successivamente, si descriverà una possibile struttura per ciascuna delle primitive precedentemente presentate.

3.6.1 Thread MainBoot

Il thread MainBoot è il primo componente del sistema che deve essere avviato. Si occupa di eseguire le operazioni iniziali necessarie alla creazione della piattaforma JADE e deve quindi essere eseguito da qualsiasi nodo del sistema che si intende rendere operativo all'interno del sistema stesso.

Inoltre è responsabile delle ultime operazioni richieste dalle primitive di amministrazione: infatti è il componente che si occupa di gestire, dall'esterno della piattaforma JADE, lo stato dei container e degli agenti in essi contenuti.

Il MainBoot assume comportamenti differenti a seconda della tipologia del nodo su cui viene avviato:

- PEER:
  - creazione del container JADE e configurazione dei relativi parametri;
  - avvio dei servizi di JADE e dei servizi di terzi previsti per un peer;
  - creazione degli agenti di supporto;
  - connessione del container JADE al main container della piattaforma (in esecuzione sul superpeer della regione);
  - attesa delle primitive di amministrazione indirizzate ad un peer;
- SUPERPEER:
  - creazione del main container JADE e configurazione dei relativi parametri;
  - avvio dei servizi di JADE e dei servizi di terzi previsti per un superpeer;
  - attesa delle primitive di amministrazione indirizzate ad un superpeer;
  - esecuzione di tutte le operazioni descritte precedentemente per un peer;
- SERVER:
  - esecuzione di tutte le operazioni descritte precedentemente per un superpeer;

E' importante notare che tramite questo componente è possibile gestire la piattaforma nel suo complesso; quindi è necessario che rimanga sempre in esecuzione.

Ad esempio il MainBoot deve poter ricevere, tra le altre, la primitiva startP in modo da avviare tutti e soli gli agenti relativi al nodo su cui è in esecuzione (peer, superpeer o server) [12]; deve poter ricevere la primitiva suspendP, in modo da completarne l'esecuzione portando tutti gli agenti del nodo (ad eccezione di quelli di supporto) nello stato SUSPENDED (cfr. *Figura 3-1*, p.51); deve poter ricevere le primitive stopP e freezeP, interrompendo definitivamente l'esecuzione di tutti gli agenti del nodo su cui è in esecuzione[13] (eccezion fatta per quelli di supporto).

[12] Deve quindi aver accesso agli oggetti di tipo AgentController usati dal Main

[13] Attraverso l'invocazione del metodo `kill()` dell'oggetto creato dal Main di tipo AgentController. Per ulteriori dettagli, si veda il paragrafo *5.2.2*.

Sempre a titolo esemplificativo, quando il MainBoot riceve una primitiva di tipo regione come removeR, deve essere in grado di rimuovere non solo gli agenti tipici di un peer ma anche quelli che caratterizzano un superpeer.

Inizialmente si era deciso di attribuire maggiori funzionalità al MainBoot, ma successivamente si è preferito concentrare la maggior parte dei compiti sull'agente PMA descritto nel prossimo paragrafo.

3.6.2 Agente PMA (PeerManagementAgent)

L'agente PMA (PeerManagementAgent) è in esecuzione su ogni nodo del sistema e costituisce il punto nodale di tutto l'impianto per la gestione delle primitive di amministrazione, compresi la validazione e l'inoltro delle primitive stesse.

Tutte le comunicazioni fra i nodi del sistema relative all'esecuzione delle primitive di amministrazione vengono gestite dai vari PMA. Dal momento che si intende rendere possibile la creazione di una primitiva da più punti del sistema e non solo, ad esempio, dal server, i PMA organizzano tali comunicazioni in modo gerarchico.

A titolo di esempio si osservi la *Figura 3-7* (p.74): se una primitiva stopP inviata dal superpeer della REGION2 è indirizzata ad un peer di un'altra regione (REGION3), tale primitiva, creata dal PMA del superpeer, viene inoltrata [1] al PMA del server il quale conosce la struttura dell'overlay network ed è in grado di inoltrarla a sua volta [2] al superpeer della REGION3; questo superpeer è in grado di raggiungere direttamente ([3] e [4]) il peer destintario della primitiva e di comunicare direttamente [5] con il suo MainBoot che andrà, in questo caso, a terminare l'esecuzione degli agenti del peer.

Da notare è la possibilità di far giungere [6] in modo diretto (senza passare dal server) al nodo creatore (nel caso specifico il superpeer della REGION2) le informazioni relative all'esecuzione delle primitiva, che potrebbero essere, ad esempio, gli aggiornamenti sull'interruzione dei signoli agenti o il verificarsi di condizioni di errore.

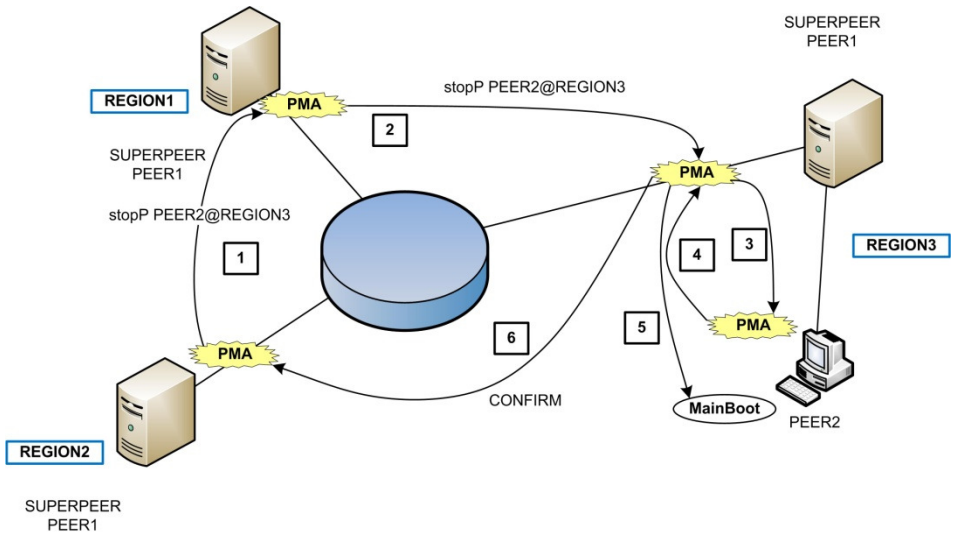


Figura 3-7 – Gestione gerarchica delle primitive di amministrazione.

Esecuzione sul server

Quando l'agente PMA è avviato sul server deve essere in grado di:

- offrire un interfaccia (grafica o testuale) che permetta all'amministratore di eseguire tutte le primitive;
- avviare un superpeer e quindi aggiungere una regione (stabilendo direttamente una connessione con il MainBoot del superpeer che riceverà dal PMA del server la propria posizione nella overlay network);
- interrompere un superpeer e quindi rimuovere una regione (il PMA del server contatta il PMA del superpeer che si occuperà di interrompere l'esecuzione di tutti i peer della regione e di notificare l'esito al PMA del server. Il PMA del server informa preventivamente anche i superpeer vicini in modo che effettuino un bypass del superpeer in capo alla regione da rimuovere);
- comunicare con il PMA dei superpeer per informarli circa gli aggiornamenti della loro posizione all'interno della overlay network;
- inviare una primitiva ad un peer di qualsiasi regione (inviando un messaggio al PMA del superpeer, chiedendogli di eseguire la primitiva sul peer, contattare il MainBoot del peer e darne comunicazione al server;

- restare in attesa di richieste da parte di altri superpeer da inoltrare verso altre regioni;
- restare in attesa di eventuali comandi di servizio;
- restare in attesa di messaggi provenienti da superpeer (regioni) che vengono avviati/ interrotti "in locale" e non dal PMA del server. In questo modo viene garantita l'integrità della overlay network.

Le informazioni complessive sulla struttura della overlay newtork vengono conservate all'interno del database del server o, in alternativa, in un'apposita struttura dati creata ad hoc.

Per ogni superpeer (regione) il server dovrà mantenere le seguenti informazioni:

Attributo	**Descrizione**
name	Nome univoco
state	Stato
AMSGUID	GUID dell'AMS
AMSMtpAddressPublic	Indirizzo MTP pubblico del main container in cui vive l'AMS (IP.IP.IP.IP:porta) [statico o reso simile da un servizio di DynamicDNS]
PMAGUID	GUID del PMA
PMAMtpAddressPublic	Indirizzo MTP pubblico del container in cui vive il PMA (IP.IP.IP.IP:porta) [statico o reso simile da un servizio di DynamicDNS]
clockwiseRegionName	Nome della regione vicina (clockwise) nella overlay network
clockwiseAMSGUID	GUID dell'AMS (clockwise)
clockwiseAMSMtpAddress Public	Indirizzo MTP pubblico del AMD (clockwise)

anticlockwiseRegionName	Nome della regione vicina (anticlockwise) nella overlay network
anticlockwiseAMSGUID	GUID dell'AMS (anticlockwise)
anticlockwiseAMSMtpAddress Public	Indirizzo MTP pubblico del AMD (anticlockwise)

Tabella 3-2 – Informazioni che identificano una regione (superpeer).

Esecuzione sul superpeer

L'agente PMA avviato sul superpeer ha il compito di:

- offrire un interfaccia (grafica o testuale) che permetta all'amministratore di eseguire le primitive di tipo peer (startP, stopP, suspendP, freezeP, etc..) sui peer della regione e le primitive di tipo regione(addR, downR, etc..) sulla regione locale e sulle altre regioni (tramite l'inoltro al server);
- mantenere aggiornata la posizione del superpeer nella overlay network (riferimenti ai superpeer adiacenti) ricevendo gli aggiornamenti inviati dal PMA del server;
- occuparsi della rimozione (removeR) della regione di cui è superpeer inviando il messaggio di stopP ai PMA di tutti i peer della regione, attendendo le risposte, contattando i MainBoot dei vari peer, informando il PMA del server, interrompendo la propria esecuzione e chiedendo al proprio MainBoot di rimuovere tutti gli agenti locali previsti per un peer e per un superpeer;
- restare in attesa di eventuali comandi di servizio;
- restare in attesa dei deguenti messaggi e inoltrarli, su richesta del server o di altri superpeer, al PMA del peer destinatario presente in regione (e, se necessario, anche al MainBoot):
  - il messaggio stopP, rimuovendo il peer dalla lista dei tranche owner di tutte le tranche possedute dal peer, attendendo la notifica e aggiornando lo stato del peer nel database (del superpeer);
  - i messaggi suspendP e resumeP, attendendo la notifica e aggiornando lo stato del peer nel database;

- il messaggio freezeP, rimuovendo il peer dalla lista dei tranche owner, attendendo la notifica e aggiornando lo stato del peer nel database;
- il messaggio startP, attendendo la notifica e aggiornando lo stato del peer nel database;
- il messaggio resumeP, attendendo la notifica e aggiornando lo stato del peer nel database;
- il messaggio thawP, reinserendo all'interno del db (o comunque rendendole nuovamente valide) le informazioni precedente rimosse in seguito ad un messaggio di freezeP (ibernazione), attendendo la notifica e aggiornando lo stato del peer nel database;
- il messaggio downR, rimuovendo il peer dalla lista dei tranche owner, attendendo la notifica e aggiornando lo stato del peer nel database.

All'interno del database del superpeer, per ogni peer della regione deve essere conservata (oltre a quelle già previste) l'informazione relativa allo stato del peer stesso.

Esecuzione sul peer

L'agente PMA avviato su un peer deve:

- offrire un interfaccia (grafica o testuale) che permetta all'amministratore di eseguire le primitive di tipo peer (startP, stopP, suspendP, thawP, etc..) sul peer locale o su qualsiasi altro peer della regione;
- ricevere tutte le primitive di tipo peer indirizzate al peer stesso. In particolare:
  - la primitiva stopP, per cui deve occuparsi dell'interruzione del peer, inoltrando il messaggio relativo alla primitiva stopP a tutti gli agenti del peer, rimanendo in attesa delle risposte (alcuni agenti non possono interrompere immediatamente la propria esecuzione: ad esempio il DMA deve prima inviare un messaggio di KILL a tutti gli MA creati e migrati sugli altri peer fonti di determinate tranche), notificando l'esito dell'operazione al superpeer, chiedendo al MainBoot di rimuovere tutti gli

agenti del peer senza però interrompere la propria esecuzione;

- la primitiva freezeP, procedendo come per la primitiva stopP, ma chiedendo anche al superpeer di rimuovere temporaneamente il peer dalla lista dei tranche owner per le tranche possedute dal peer stesso.
- la primitiva suspendP, inoltrandolo poi a tutti gli agenti che provvederanno a sospendersi, ricevendo le varie risposte, infomando il superpeer e il MainBoot (che rimarrà in attesa del messaggio resumeP).

3.6.3 Behaviour "coordinatore"

Ogni agente (a prescindere dalla tipologia di nodo in cui viene creato - peer, superpeer o server -) deve essere dotato di un componente in grado di elaborare le primitive ricevute: più precisamente questo componente aggiuntivo (ad esempio un behaviour "coordinatore") deve poter interagire col PMA e con tutti gli altri behaviour dell'agente, in modo da stabilire se e quando tutti siano o meno in grado di garantire il passaggio di stato (Figura 3-8).

Una volta ricevuta la notifica da tutti gli altri behaviour, il behaviour "coordinatore" conferma all'agente PMA del peer la corretta esecuzione della primitiva da parte di tutti gli agenti del peer di cui è coordinatore.

Si ricorda che ogni behaviour di un agente agisce autonomamente; quindi ciascun behaviour controlla indipendentemente dagli altri la presenza di una primitiva da eseguire e può richiedere tempi diversi dagli altri per portarla a termine. Di conseguenza è possibile che un agente debba attendere più del dovuto perché uno dei suoi behaviour ha, ad esempio, incontrato problemi durante la comunicazione con altri agenti.

Il behaviour "coordinatore" di ogni agente che processa il messaggio, potrebbe impostare una variabile statica dell'agente che può essere letta da tutti i behaviour nel momento che ciascuno di essi ritiene più adatto; quando un behaviour vede che è stata richiesta, ad esempio, l'ibernazione, svolge le ultime attività necessarie ed invia una notifica al behaviour coordinatore.

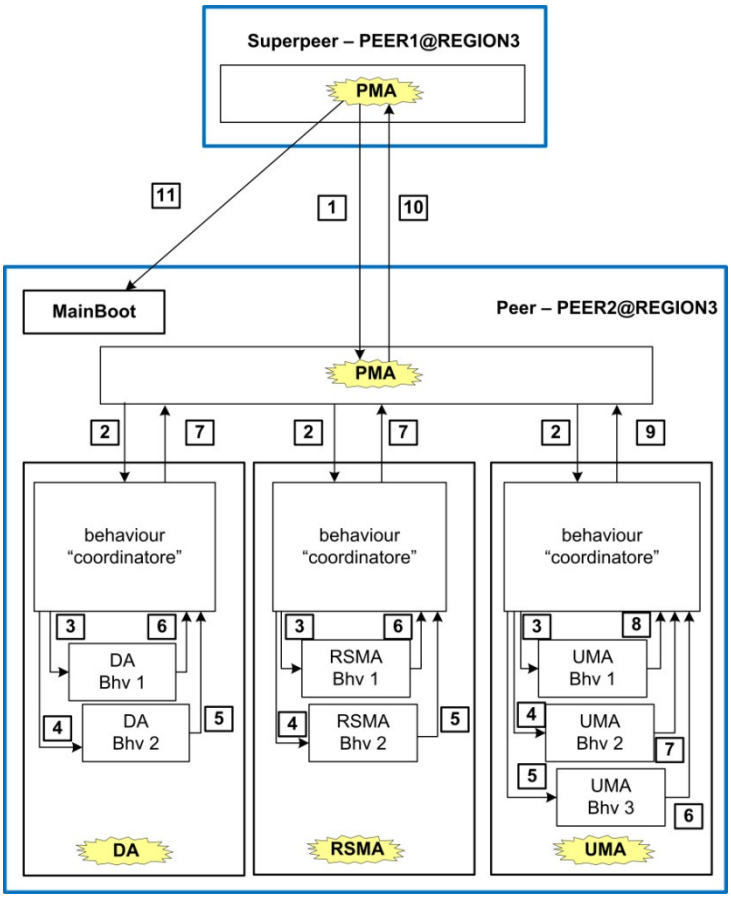


Figura 3-8 – Primitive di amministrazione: behaviour "coordinatore".

Quando tutti i behaviour hanno confermato, il behaviour "coordinatore" avvisa il PMA del peer. Il behaviour "coordinatore" che riceve freezeP potrebbe lanciare anche un threaded behaviour[14] di tipo Ticker (con un timeout) per controllare periodicamente le notifiche ricevute dagli altri behaviour. Anche il PMA del peer potrebbe dotarsi di un threaded behaviour Ticker per verificare le notifiche ricevute da tutti gli agenti del peer; e anche il PMA del superpeer, nel caso della rimozione della regione, ad esempio, per controllare le notifiche ricevute da tutti i peer della regione.

Prima di proseguire nella trattazione, si precisa che le seguenti primitive devono poter essere eseguite dal server o da un superpeer su ogni

[14] In JADE sono disponibili i Threaded Behaviour che hanno la caratteristica di poter essere eseguiti in thread indipendenti da quello in cui viene eseguito l'agente (e tutti i behaviour non Threaded): i behaviour non Threaded devono quindi essere eseguiti uno alla volta mentre quelli Threaded vengono eseguiti concorrentemente. Nonostante ciò richieda una gestione della concorrenza per l'accesso alle risorse condivise con gli altri behaviour dell'agente, la versatilità offerta è molto elevata.

nodo del sistema, compreso quello locale. Inoltre un peer deve poter eseguire le primitive di tipo peer indirizzate ad altri peer della regione: ad esempio un peer può essere avviato (startP) o ibernato (freezeP) in locale senza che giunga un messaggio dal server; allo stesso modo un superpeer può essere disconnesso dal sistema (e con lui la regione di cui è a capo) tramite la primitiva removeR avviata sul superpeer stesso o su un altro superpeer. Questo vincolo non dovrebbe comportare grosse difficoltà dal punto di vista implementativo (si veda la *Tabella 3-1*, p. 66).

3.6.4 startP

La prima condizione necessaria affinchè sia possibile avviare un peer tramite la primitiva startP è che sia in esecuzione il thread MainBoot.

Questo thread, una volta avviato su un peer, si occupa di creare il container, di connetterlo al main container della regione e di avviare l'agente di supporto PMA. E' questo agente che riceve la primitiva startP indirizzata al peer e che la inoltra a tutti gli altri agenti del peer (su cui viene ricevuta ed elaborata da un behaviour "coordinatore" che si occupa di trasmetterla a tutti gli altri agenti).

Il comando startP può essere ricevuto localmente (ad esempio tramite un'interfaccia grafica offerta dall'agente PMA), da altri peer della regione o dall'esterno (per mezzo del server e del superpeer della regione: il server inoltra un messaggio al PMA del superpeer della regione in cui deve essere avviato il peer e il PMA del superpeer stabilisce quindi una connessione con il PMA ed il MainBoot del peer.

Il MainBoot crea agli agenti definiti per un peer, ed il PMA, a procedura ultimata, ne notificherà l'esito al superpeer della regione, facendo sì che il superpeer aggiorni nel suo database lo stato del peer appena avviato.

3.6.5 suspendP

La primitiva di sospensione prevede che il PMA del peer comunichi ad ogni agente del peer (tramite messaggio ACL) che deve sospendere la propria attività. Il messaggio viene letto (per ogni agente) da un behaviour "coordinatore"[15] appositamente aggiunto o da un behaviour già presente, a seconda della struttura dell'agente stesso: ogni agente svolge operazioni

[15] Cfr. Figura 3-8, p.81.

differenti e non tutti possono sospendersi nel momento esatto della ricezione del messaggio (vedi il DA o l'UA, per esempio); quindi la sospensione di un agente, come l'ibernazione o la terminazione, potrebbe richiedere del tempo e bisogna tenere in considerazione questo dato in fase di implementazione.

Inizialmente si era pensato di sfruttare il metodo `doSuspend()` della classe `jade.core.Agent`: una volta che tutti i behaviour dell'agente avessero confermato la propria disponibilità al passaggio di stato al behaviour "coordinatore", questo avrebbe effettivamente sospeso l'agente JADE (stato SUSPENDED, cfr. *Figura 3-1*, p.51). Tuttavia, riflettendo sul significato attribuito alla primitiva suspendP, si è ritenuto sufficiente fare in modo che ogni behaviour sospendesse le proprie attività (dandone comunque conferma al behaviour "coordinatore"), senza che l'agente passasse nello stato SUSPENDED: rimanendo nello stato ACTIVE l'agente non viene ignorato dallo scheduler di JADE e la ripresa delle attività in seguito al resumeP avviene più rapidamente.

Quindi, compatibilmente con la propria struttura, ogni behaviour esegue le ultime operazioni prima della sospensione, informa il behaviour "coordinatore" e rimane in attesa del resumeP controllando periodicamente[16] la coda dei messaggi o il cambiamento di una variabile di stato. Quando il behaviour coordinatore ha ricevuto conferma da tutti i behaviour dell'agente, ne dà notifica al PMA che si occuperà di informare il creatore della pimitiva sia in caso di successo che in caso di errore.

Si ricordi che il container del peer sospeso rimane in esecuzione così come tutti gli agenti che, pur continuando ad accodare i messaggi ACL nelle rispettive code, non eseguono alcuna operazione.

3.6.6 freezeP

Il freezing di un peer è un'attività che richiede particolare attenzione e che tiene presenti alcune considerazioni che verranno trattate nel 3.6.9 (in cui viene presentata un'ipotesi di soluzione per la primitiva stopP).

Una volta che il messaggio freezeP giunge al PMA del peer da ibernare, l'agente deve informare tutti gli altri agenti del peer, in modo simile a quanto descritto per suspendP. Quando ogni agente giunge in uno stato in

[16] Tramite il metodo `block(long millis)` offerto dalla classe denominata `jade.core.behaviours.Behaviour`.

cui ritiene che la propria ibernazione sia possibile, interrompe l'esecuzione di tutti i propri behaviour (rilasciando le risorse possedute) e lo notifica al PMA del peer. Quando tutti gli agenti del peer hanno confermato al PMA del peer il proprio stato, il PMA del peer ne dà conferma al PMA del superpeer il quale procede

1. chiedendo all'agente AMS della regione di salvare in modo persistente lo stato di tutti gli agenti del peer;
2. chiedendo al MainBoot del peer di interrompere[17] tutti gli agenti del peer (ad eccezione del PMA).

Si ricorda che il salvataggio dello stato di un agente richiede che tutte le variabili di stato utilizzate dall'agente e da tutti i suoi behaviour siano serializzabili. Nel caso sia necessario usarne di non serializzabili (ad esempio `java.net.Socket`), sarà necessario definirle con l'attributo `transient` in modo da escluderle dal salvataggio nel database.

E' importante notare che il metodo `setup()` di un agente non viene eseguito in seguito al risveglio dall'ibernazione; ciò implica che, alla ricezione della primitiva thawP, le variabili `transient` debbano essere re-inizializzate in modo da renderle nuovamente utilizzabili.

Inoltre si dovrà porre attenzione a come gestire la ripresa delle attività da parte dei behaviour ciclici: una possibile alternativa potrebbe consistere nel definire una macchina a stati finiti a supporto di ogni behaviour, in modo tale da poter salvare lo stato della macchina a stati finiti in cui si trovava il behaviour prima dell'ibernazione: in questo modo l'esecuzione riprenderebbe proprio da quello stato.

Dal momento che ibernare un peer significa salvare il suo stato in modo che, al suo successivo risveglio, sia in grado di proseguire la sua esecuzione da dove l'aveva interrotta, è necessario considerare alcuni aspetti.

Innanzitutto molte operazioni coinvolgono più agenti, quindi bisogna introdurre un ordinamento degli agenti secondo cui procedere alla loro ibernazione. In mancanza di questo ordinamento, l'ibernazione di un agente HA, ad esempio, prima che sia avvenuta l'ibernazione di un agente DA potrebbe comportare la starvation di quest'ultimo agente in quanto esegue molte operazioni per cui è richiesta l'interazione con l'HA.

[17] Stato DELETED, cfr. par. *3.3*.

In *Figura 3-9* è schematizzata la relazione di dipendenza fra gli agenti di un peer che deve essere osservata durante l'esecuzione delle primitive freezeP e stopP.

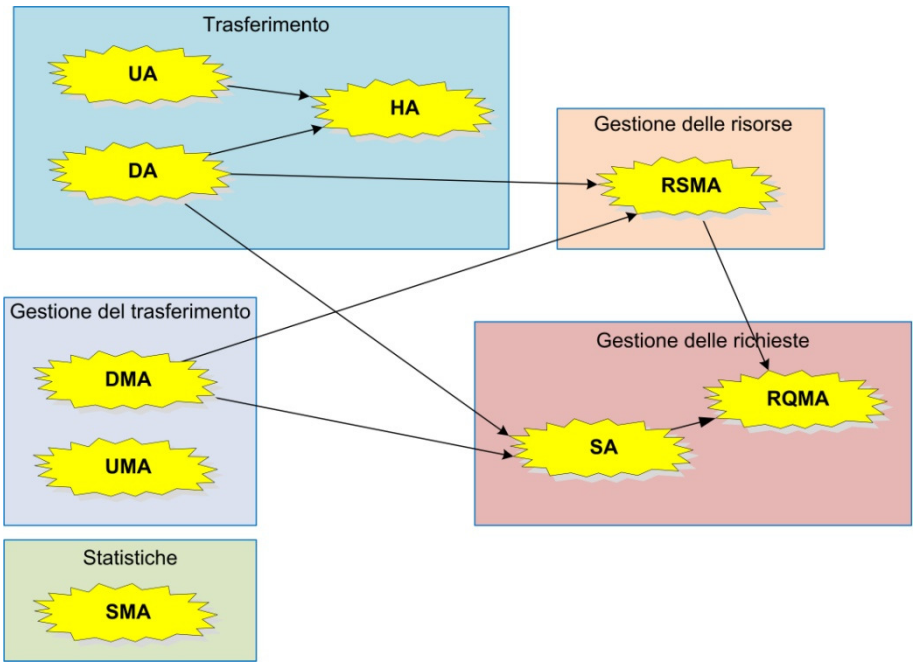


Figura 3-9 – freezeP: relazioni di dipendenza fra gli agenti.

In secondo luogo è necessario che alcuni agenti svolgano particolari operazioni prima di congelarsi: ad esempio l'agente RSMA deve modificare in MISSING lo stato di tutte le tranche che non sono PRESENT o NOT FOUND, in modo da evitare incongruenze con la successiva esecuzione dopo il thawP; analogamente l'agente RQMA deve rimuovere dal database tutte le request che non sono nello stato SATISFIED o ABORT.

Bisogna, poi, considerare la ricerca delle fonti: se un agente viene ibernato, non deve più essere considerato dal superpeer della sua regione una fonte valida per le tranche possedute; di conseguenza il PMA, quando riceve la notifica da tutti gli altri agenti del peer da ibernare, chiede al superpeer di rimuovere il nome del peer dalla lista dei tranche owner per tutte le tranche possedute dal peer.

Il "livello di ibernazione" che è possibile raggiungere dipende dalla possibilità di salvare in modo persistente per ogni agente:

- lo stato di ogni suo behaviour;
- il contenuto delle due code di messaggi ACL (in entrata e in uscita).

Infatti, nel caso in cui non si possa salvare il contenuto delle code dei messaggi, non sarà possibile far riprendere all'agente l'esecuzione delle proprie attività esattamente[18] dal punto in cui le aveva interrotte prima di ibernarsi (mantenendo valide le relazioni instaurate con gli altri agenti o con gli altri peer); le informazioni (relative ai contenuti) utilizzabili al termine dello "scongelamento" saranno invece limitate alle sole tranche già in possesso del peer.

3.6.7 resumeP

La ricezione da parte del PMA del peer della primitiva resumeP comporta la notifica della primitiva stessa da parte di questo agente al behaviour "coordinatore" di ogni altro agente JADE (rimasto nello stato ACTIVE) del peer. Tutti i behaviour di ogni agente controllano periodicamente l'arrivo della primitiva suspendP e, una volta giunta ed elaborata, confermano al behaviour "coordinatore" la riuscita (o meno) della primitiva.

Quando il PMA del peer ottiene la conferma da tutti gli agenti, lo stato del peer passa da pSUSPENDED a pACTIVE.

3.6.8 thawP

Questa primitiva esegue lo "scongelamento" di un peer e deve prevedere il ripristino degli stati dei vari agenti salvati all'interno del database.

Poiché in un peer ibernato sono comunque attivi il thread MainBoot e l'agente di supporto PMA, sarà compito di questo componenti supportare la procedura iniziata dal PMA del superpeer. Tale procedura prevede le seguenti operazioni.

1. Il PMA del superpeer inizia a rendere esecutiva la primitiva thawP inviando all'agente AMS tante richieste quanti sono gli agenti salvati all'interno del database[19].
2. L'agente AMS, tramite i servizi offerti da Persistence, si occupa di creare gli agenti e di caricare lo stato salvato. Per ogni agente che

[18] A patto che gli altri agenti non abbiano interrotto le interazioni iniziate in precedenza a causa di timeout.

[19] Le richieste al AMS devono seguire l'onlotogia definita in jade.domain.persistence.PersistenceOntology.

viene ripristinato, l'add-on Persistence esegue il metodo `afterLoad()` in cui si inizializzano tutte le variabili non serializzabili dell'agente e di tutti i suoi behaviour (come i Controller, i ServerSocket o le ThreadedBehaviourFactory necessarie alla creazione di Threaded Behaviour). A partire da questo momento, dopo un breve transitorio, si considerano attivi - stato ACTIVE del ciclo di vita di un agente JADE - tutti gli agenti che vengono a trovarsi, quindi, nello stato pSTOPPED.

3. Il PMA del superpeer inoltra la primitiva thawP al PMA del peer che la inoltra al behaviour "coordinatore" di ogni altro agente.
4. Ogni agente del peer (più precisamente ciascun behaviour di ogni agente) porta termine la primitiva: o ripristina il proprio normale funzionamento oppure conclude riscontrando un errore. Infine invia una conferma all'agente PMA del peer.
5. L'agente PMA del peer controlla di aver ricevuto tutte le conferme dagli agenti (potrebbe attendere un tempo massimo ad esempio avvalendosi dell'aiuto di un Treaded Behaviour Ticker) ed avvisa il PMA del superpeer che:
    a. aggiorna lo stato del peer;
    b. reinserisce all'interno del proprio database (o comunque le renderà nuovamente valide) le informazioni sulle tranche possedute dal peer, aggiornando alcuni record della tabella tranche_owner;
    c. avvisa il creatore della primitiva tramite messaggio ACL.

Si ricorda che, analogamente al caso della primitiva freezeP, è necessario rispettare una relazione di dipendenza fra gli agenti (*Figura 3-10*, p.86) del peer durante la fase di scongelamento: infatti gli stati degli agenti non vengono caricati tutti contemporaneamente e ciò potrebbe causare malfunzionamenti.

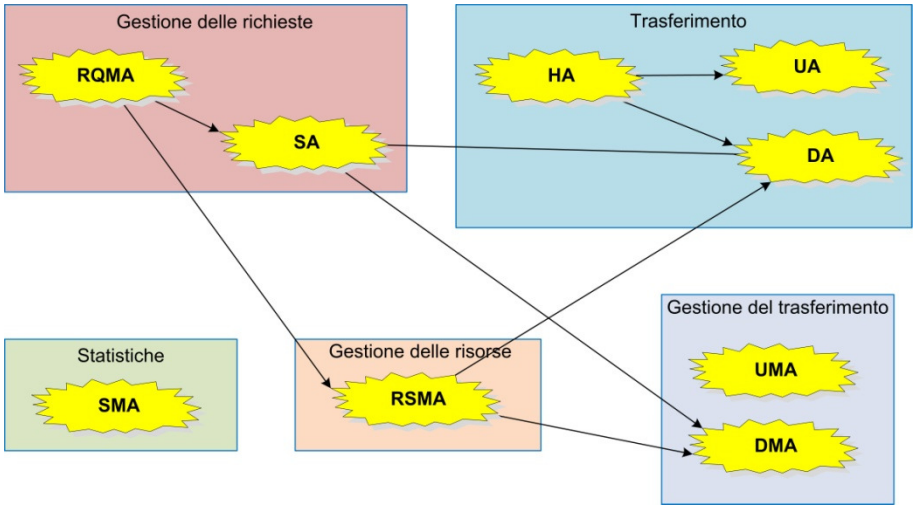


Figura 3-10 – thawP: relazioni di dipendenza fra gli agenti.

3.6.9 stopP

Per interrompere l'esecuzione di un peer è necessario che pervenga un messaggio di stopP al suo PMA.

Il PMA del peer inoltra il messaggio a tutti gli agenti del peer stesso i quali, in base allo stato in cui si trovano e al behaviour attualmente in esecuzione, provvederanno ad interrompere la loro esecuzione.

Una volta che tutti gli agenti hanno dato conferma, il PMA del peer lo notifica al PMA del superpeer il quale contatta il MainBoot del peer affinchè rimuova tutti gli agenti del peer (ad eccezione del PMA).

Di seguito si riporta un elenco di tutti gli agenti di un peer e si descrive la procedura da eseguire alla ricezione del messaggio di stopP a seconda dello stato in cui si trova l'agente. Ogni agente rimuove quindi tutti i suoi behaviour e, come ultima operazione, informa il PMA prima di terminare la propria esecuzione.

RSMA(ReSourceManagementAgent)

Appena l'agente, attraverso il behaviour "coordinatore", processa il messaggio di stopP, imposta a "missing" lo stato delle tranche che non sono presenti (stato = "present") perché associate a qualche processo di ricerca ancora in corso e interrompe le proprie attività a prescindere da quale fosse l'ultimo behaviour che ha completato l'esecuzione. Eventuali messaggi inviatigli dagli altri agenti, verranno semplicemente ignorati.

Si ricorda che ogni behaviour di un agente agisce autonomamente; quindi ciascun behaviour controlla indipendentemente dagli altri la presenza di una primitiva da eseguire e può richiedere tempi diversi dagli altri per portarla a termine.

Gli agenti RSMA, RQMA e SA gestiscono le attività di ricerca del peer, quindi l'interruzione immediata della loro esecuzione non comporta problemi.

RQMA(ReQuestManagementAgent)

Come per il RSMA, appena il messaggio di stopP viene processato, l'agente interrompe la propria esecuzione dopo aver impostato ad "abort" lo stato delle richieste che non sono soddisfatte (stato = "satisfied").

SA(SchedulerAgent)

Anche nel caso del SA l'agente interrompe la propria attività appena il messaggio di stopP viene processato. In questo modo non verranno più inoltrate nuove richieste al RRA del superpeer.

DMA (DownloaderManagementAgent)

Il DMA, processando le informazioni trasmesse dal RSMA, crea un agente MA per ogni fonte valida comunicatagli dal RRA: nel caso di fonte intra-regione il MA migra direttamente sul peer; nel caso di fonte extra-regione il MA migra verso il MainContainer del superpeer e da lì comunica con l'UMA del peer.

Data la struttura del behaviour *DMASelectBestSourceBhv*, la lettura del messaggio di stopP non può che essere effettuata nel primo stato (*START_SELECTION_SOURCE_AFTER_ANSWER_RECEIVED_FROM_RSMA*). Il DMA dovrà quindi inviare un messaggio di KILL a tutti gli MA creati.

MA(MigratiornAgent)

L'agente MA non è mai in esecuzione sul peer che l'ha creato, ma sempre sul peer fonte o, nel caso di richiesta extra-regione, sul superpeer (all'interno del mainContainer). Nel caso in cui debba essere interrotto il peer creatore del MA, all'MA viene inviato dal DMA del peer creatore un messaggio di KILL che viene processato dal suo behaviour

`MAKillAgentBhv`[20]; nel caso in cui debba invece essere interrotto il peer che ospita l'agente MA, allora sarà cura del UMA della fonte inviare il messaggio di kill al MA; infine, se il MA è in esecuzione sul superpeer, nel caso in cui si richieda di rimuovere la regione (e quindi di interrompere il superpeer), si dovrà decidere se il messaggio di KILL debba essere inviato dal UMA che il MA ha contattato o da un agente del superpeer (es. il PMA)

Si noti come l'invio del messaggio di KILL al MA sia necessario e non superfluo: teoricamente sarebbe possibile eliminare tutti gli agenti di un container invocando il metodo kill() sull'oggetto di tipo ContainerController; tuttavia in questo modo gli agenti DMA o UMA che erano in contatto con gli MA e devono continuare la loro esecuzione, potrebbero riscontrare alcuni problemi causati dall'improvvisa eliminazione degli MA.

UMA(UploaderManagementAgent)

I due behaviour significativi del UMA sono `UMAgetTicketBhv` e `UMAcallForTicketBhv`. Il primo rilascia nuovi ticket a nuovi MA e, quando il messaggio di stopP viene processato dal behaviour aggiunto appositamente all'agente, non deve più continuare la sua esecuzione. Il secondo, invece, se si trova negli stati 2 e 3 (rispettivamente *`WAITING ACTION`* e *`WAITING COMPLETED DOWNLOAD`*) deve continuare la propria esecuzione per consentire al MA appena chiamato o al UA impegnato nel upload di una tranche di terminare le operazioni previste; se si trova nello stato 1 (*`CALLING`*) allora dovrà chiamare tutti i ticket rilasciati fino a quel momento ed utilizzarlo non per informare gli MA che è giunto il loro turno, ma per inviar loro un messaggio di KILL (che sarà ricevuto dal behaviour `MAKillAgentBhv`).

Infine anche l'UMA potrà terminare la propria esecuzione, sicuro che tutti gli MA presenti sul peer termineranno la loro esecuzione.

DA (DownloaderAgent)

Questo agente potrà essere interrotto solo quando il behaviour `DAReceivedTrancheFromUABhv` si trova nello stato *`WAITING_FOR_MESSAGE`*

[20] Anche nel caso di migrazione extra-regione, come descritto nel CAPITOLO 2 (paragrafo 2.5.3), il messaggio dal DMA al MA può essere inviato senza difficoltà dal servizio di gestione dei messaggi ACL offerto da JADE (e quindi senza scomodare l'agente RPA della regione in cui si trova l'MA migrato): infatti l'agente MA si trova sul MainContainer della regione che dispone di un indirizzo MTP pubblico visibile da tutte le altre regioni. L'eventuale comunicazione inversa, dal MA al DMA, deve invece essere coordinata dal RPA della regione del DMA.

e quando tutti i threaded behaviour `DADownloaderTrancheFromUABhv` creati fino a questo momento hanno terminato la loro esecuzione: infatti che il download della tranche dal UA della fonte sia andato a buon fine o meno, è compito di questi behaviour sbloccare l'UMA associato. Si rivelerà fondamentale, quindi, tener traccia di tutti i threaded behaviour avviati.

Il behaviour `DAReceivedTrancheFromUABhv` processa i messaggi che iniziano una nuova fase di download quando si trova nello stato *`WAITING_FOR_MESSAGE`*. Per evitare il rischio che l'input di stopP venga considerato durante le fasi che precedono il download della tranche ed in particolare durante l'attesa, da parte del DA, della risposta del HA, è necessario far processare il messaggio di stopP da un behaviour aggiunto appositamente che verifichi lo stato di `DAReceivedTrancheFromUABhv` e che interrompa l'esecuzione dell'agente nel momento idoneo: una possibilità potrebbe consiste nell'impostare una variabile nel behaviour `DAReceivedTrancheFromUABhv` in modo che sia quest'ultimo ad interrompere la propria esecuzione nel momento appropriato. Il behaviour aggiuntivo dovrà anche informare il HA che non sarà più disponibile ad effettuare download.

UA(UploaderAgent)

Come nel caso del DA, anche l'UA può processare il messaggio di stopP solo quando il suo behavior principale *`UASendTrancheToDABhv`* si trova nello stato *`WAITING_FOR_MESSAGE`*. Il behaviour aggiunto appositamente per processare tale messaggio dovrà controllare questa condizione e comunicare al HA che non sarà più disponibile ad effettuare upload.

Nessun'altra operazione deve essere effettuata da questo agente, visto che lo sblocco dell'UMA è a carico del DA.

HA (HoleAgent)

L'agente si occupa di comunicare al di fuori della piattaforma Jade con l'altro agente HA su richiesta del UA (nel caso in cui l'agente sia sul peer fonte) o in risposta ad una richiesta effettuata dall'altro HA (nel caso l'agente sia sul peer richiedente) affinchè ciascuno dei due peer possa conoscere l'indirizzo e la porta pubblici dell'altro peer.

Sia l'UA che il DA non interrompono la loro esecuzione durante la fase di setup della connessione; inoltre non è detto che il messaggio di stopP venga processato dai due agenti nello stesso momento. Le operazioni eseguite dal HA sono fondamentali sia per l'UA che per il DA (entrambi gli

agenti, durante le normali operazioni di setup della connessione, necessitano del HA), quindi è necessario che il HA termini la propria esecuzione solo quando:

- processa il messaggio di stopP inviato dal PMA;
- il DA gli comunica che sta per terminare la sua esecuzione;
- anche l'UA gli comunica che sta per terminare la sua esecuzione.

NOTA IMPORTANTE: Il DA, l'UA e l'HA invocano durante la loro esecuzione il metodo `blockingReceive(template)` che interrompe l'esecuzione di tutti i behaviour dell'agente finchè non viene ricevuto un messaggio che soddisfi il template. Questo comportamento è accettabile se l'agente possiede un solo behaviour. Dal momento che per la lettura del messaggio di stopP è necessario un secondo behaviour, è indispensabile modificare il comportamento del behaviour esistente (eventualmente aggiungendo uno o più stati) in modo che venga usato `receive(template)` al posto di `blockingReceive(template)`.

SMA(StatisticManagementAgent)

Questo agente può terminare la propria esecuzione (notificando al PMA del peer) appena processa il messaggio di stopP, senza preoccuparsi di eseguire altre operazioni.

PMA (PeerManagementAgent)

Questo agente, quando eseguito su di un peer, dopo aver inviato i messaggi di stopP a tutti gli agenti del peer deve:

- ricevere le notifiche dagli agenti del peer;
- rilevare il verificarsi di errori durante l'esecuzione della primitiva da parte degli altri agenti (ad esempio tramite un timeout);
- notificare al PMA del superpeer l'esito della procedura di terminazione del peer, in modo che aggiorni la lista dei tranche owner.

3.6.10 downP

Questa primitiva necessaria per riportare ad uno stato consistente (pSTOPPED) un peer che si trova nello stato pFAILURE prevede l'arresto di tutti gli agenti del peer ancora in esecuzione.

Il PMA del superpeer della regione a cui appartiene il peer (verso il quale è indirizzata la primitiva), contatta il MainBoot del peer e chiede di eseguire il metodo `kill()` per tutti gli oggetti di tipo `AgentController` che identificano gli agenti del peer (ad escluzione del PMA).

3.6.11 addR

Le pre-condizioni per eseguire il comando di avvio su di una regione non ancora aggiunta al sistema sono le seguenti:

1. il thread MainBoot deve essere in esecuzione;
2. il main container della piattaforma deve essere in esecuzione e, con esso, l'agente RMA;
3. tutti i servizi necessari devono essere stati avviati;
4. il container del peer che funge da superpeer deve essere in esecuzione e, con esso, l'agente PMA;
5. l'agente PMA del superpeer deve aver contattato il server informandolo circa le informazioni di base che identificano la regione:
   a) nome;
   b) stato;
   c) GUID dell'agente AMS;
   d) indirizzo MTP pubblico del AMS;
   e) GUID dell'agente PMA;
   f) indirizzo MTP pubblico del PMA.

Il comando associato alla primitiva addR viene necessariamente inviato dal server che contatta l'agente PMA della regione da aggiungere. Il server comunica al PMA del superpeer la sua nuova posizione all'interno della overlay network (trasmette le informazioni relative ai futuri vicini); solo quando il superpeer dà conferma al server dell'avvio della regione, il server apporta le modifiche alla overlay network ed invia le informazioni sulla nuova regione ai superpeer delle regioni fra cui è stata inserita[21]. Questi due superpeer aggiornano i riferimenti relativi al nuovo vicino senza che ciò pregiudichi la attività di ricerca in corso: infatti i SearchAgent che dovessero trovarsi in corrispondenza delle modifiche appena apportate alla

[21] Questa condizione lascia la overlay network sempre connessa e consente alle attività di ricerca dei SerachAgent di non essere interrotte.

overlay network verrebbero comunque instradati correttamente verso il successivo superpeer da visitare.

Ad esempio si osservi la *Figura 3-11* (a).

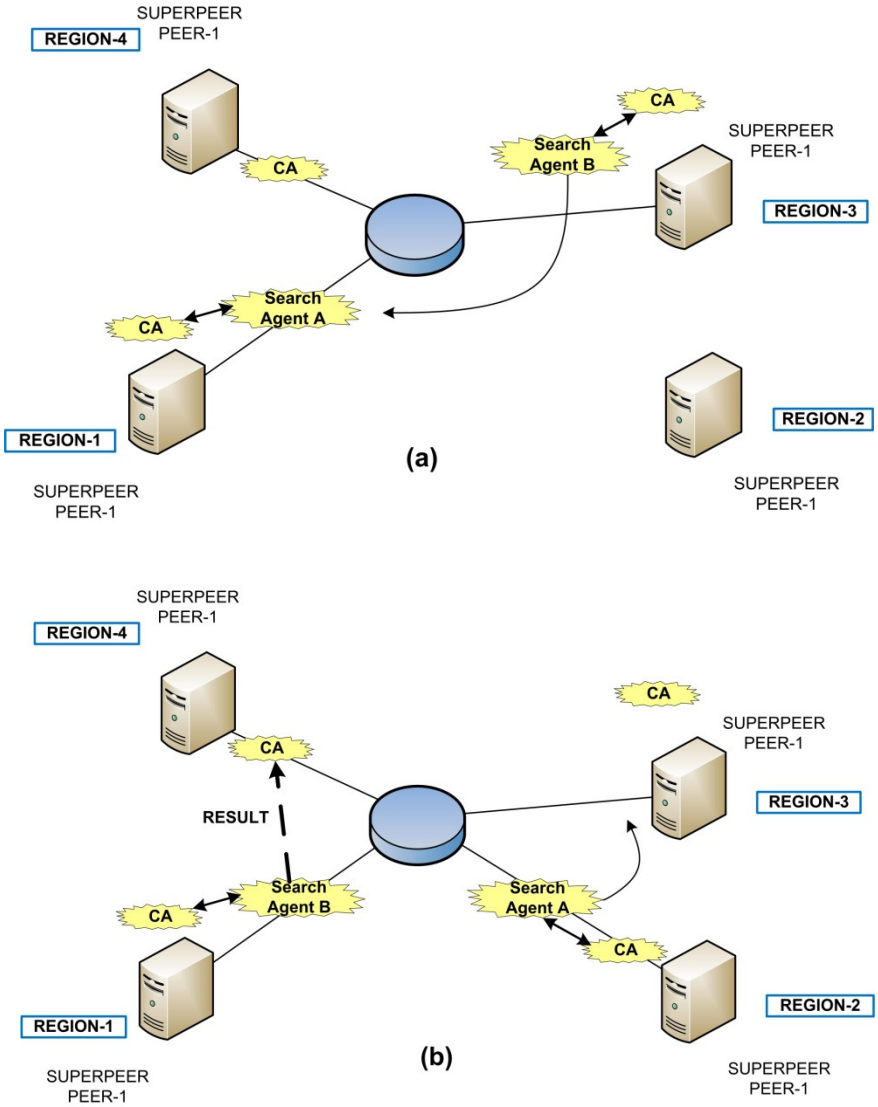


Figura 3-11 – Primitiva addR e ricerche extra-regione.

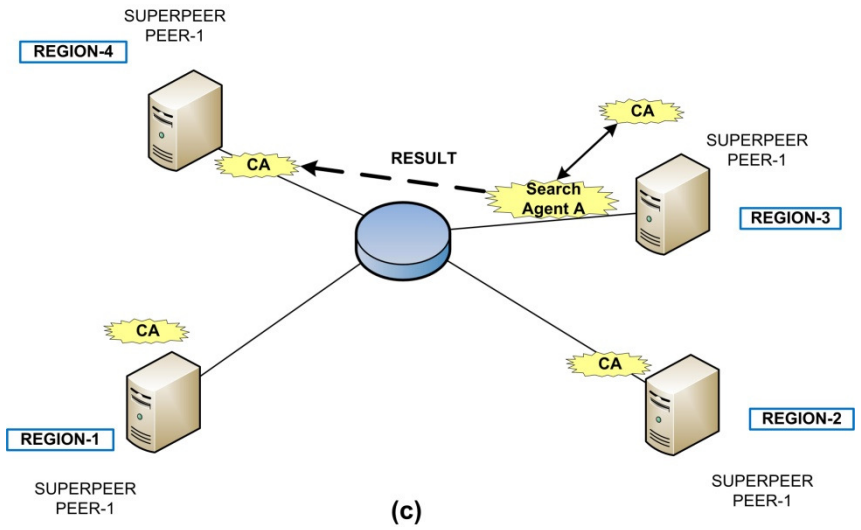
I due SearchAgent creati dall'agente CA della REGION4 si trovano sui due superpeer fra i quali si andrà ad inserire la REGION2: il SearchAgentA si trova nella REGION1 e sta ancora interagendo con l'agente CA mentre il SearchAgentB si trova nella REGION3 e, avendo finito di interagire con l'agente CA di tale regione è pronto a migrare verso la REGION1.

Nel momento in cui il PMA del superpeer della REGION2 conferma al server il completamento delle operazioni richieste dalla primitiva addR, il

server avvisa i superpeer delle regioni 1 e 3 dell'aggiunta della nuova regione (*Figura 3-11* (b)).

Mentre il SerachAgent B, raggiunta una regione (la REGION1) già visitata dal suo compagno di ricerca, invia i risultati al CA creatore nella REGION4, il SearchAgentA migra verso la nuova regione appena aggiunta, raccoglie le informazioni necessarie e migra verso la regione successiva (la REGION3).

A questo punto (*Figura 3-11* (c)) il CA della REGION 3 lo informa che il suo compagno di ricerca aveva già visitatato quella regione ed anche il SearchAgentA invia i risultati della propria ricerca al CA creatore nella REGION4.



Quando il PMA del superpeer riceve la primitiva addR, comunica al server l'indirizzo pubblico a cui può essere contattato il proprio MainBoot. Il PMA del server contatta il MainBoot della regione da aggiungere affinchè avvii tutti gli agenti tipici di un superpeer (oltre al RMA già in esecuzione, l'RPA e l'RRA) e tutti gli agenti tipici di un peer: un superpeer è essenzialmente un peer e, come tale, deve poter eseguire anche tutte le operazioni previste da questo tipo di nodo.

Successivamente il PMA del server chiede al PMA del superpeer di rendere operativa la regione: tutti gli agenti di tipo superpeer e di tipo peer del superpeer iniziano il normale funzionamento ed il peer (unico della regione) passa nello stato pACTIVE.

Infine il PMA del superpeer informa il creatore della primitiva.

3.6.12 removeR

Rimuovere una regione significa considerare lo stato della rete di overlay, gestire le ricerche già effettuate in passato che fanno riferimento alla regione da rimuovere e gestire l'interruzione di tutti i peer e del superpeer della regione.

Prima che il server comunichi ad un superpeer di rimuovere la regione di cui è a capo, è necessario che i vicini del superpeer ne siano informati in modo che le operazioni del proprio CA (in particolare quelle svolte da `CAGetNeighboursAIDBhv`, behaviour che si occupa di restituire ai SearchAgent l'AID dei superpeer vicini) facciano riferimento alla struttura aggiornata della overlay network.

Gli eventuali SearchAgent in transito sul superpeer della regione da rimuovere, verranno indirizzati come avveniva in precedenza, verso gli ormai "vecchi" vicini, evitando di inserire nella risposta eventuali fonti presenti in regione. Questo passo della pocedura è mostrato nella seguente *Figura 3-12*.

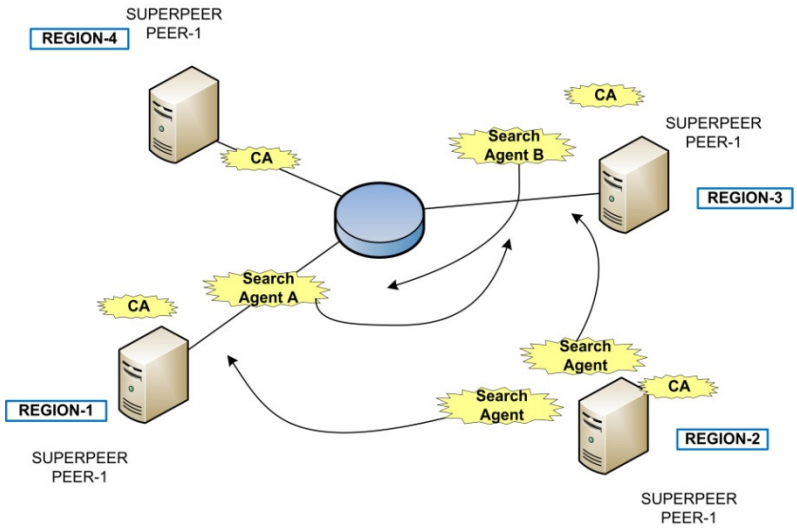


Figura 3-12 – Primitiva removeR e ricerche extra-regione [caso 1].

E' possibile, in alternativa, far in modo che il CA, prima di interrompersi, rifiuti (tramite messaggio ACL di tipo `TypeConversationId.REFUSE_FOREIGN_AGENT`) ogni SearchAgent attualmente presente sul superpeer ma non ancora "servito", facendo in modo che invii subito i propri risultati della ricerca al CA creatore senza proseguire la ricerca (*Figura 3-13*).

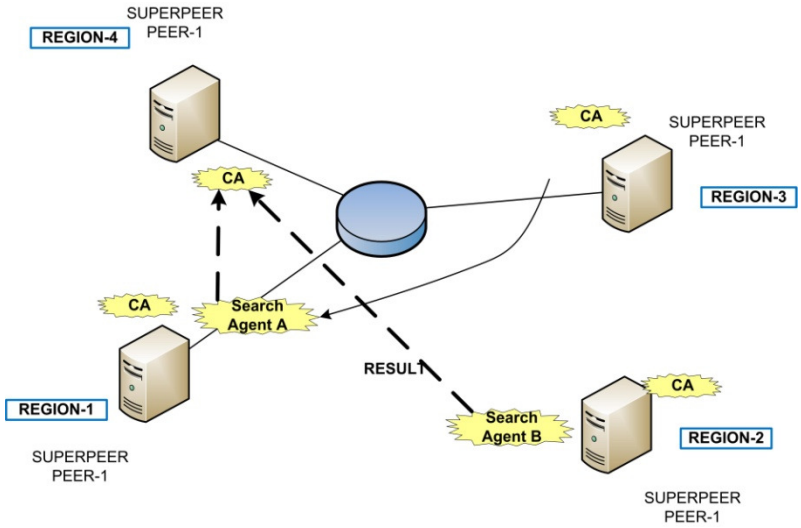


Figura 3-13 – Primitiva removeR e ricerche extra-regione [caso 2].

In questo modo la ricerca extra regione non viene pregiudicata, sia nel caso in cui l'eventuale SearchAgentA compagno di quello "rifiutato" abbia già superato la regione rimossa (vedi figura), sia nel caso in cui lo debba ancora fare. Nella *Figura 3-13*, ad esempio, si nota come il SearchAgentB (proveniente dalla REGION1) sia presente sul superpeer nel momento in cui la regione sta per essere rimossa. Questo agente viene rifiutato dal CA ed invia i risultati della ricerca al proprio creatore nella REGION4. Il SearchAgentA, invece, migra dalla REGION3 quando il superpeer ha già ottenuto dal server l'aggiornamento della overlay network, e quindi si sposta lungo il "bypass" verso la REGION1; poiché la REGION1 era già stata visitata dal suo compagno di ricerca, anche il SearchAgentA invia i risultati al CA creatore (che si trova nella REGION4).

Per quanto riguarda la gestione delle fonti presenti in regione, si potrebbe informare dell'imminiente arresto della regione tutte le regioni da cui sono state formulate le richieste attualmente pendenti. Tuttavia questa alternativa è difficilmente realizzabile con la struttura attuale del sistema anche perché alcuni MA potrebbero essere già stati creati e pronti per migrare. L'unica soluzione praticabile individuabile al momento consiste nell'impedire la migrazione in ingresso da parte di nuovi MA (impostando il parametro `Profile.ACCEPT_FOREIGN_AGENTS` della piattaforma a false) e fare in modo che quelli già migrati vengano interrotti dagli UMA che essi stessi contatteranno.

Di seguito si elencano tutti gli agenti riservati ad un superpeer più il PMA e si descrive la procedura da eseguire alla ricezione del messaggio di removeR a seconda dello stato in cui si trova l'agente.

Ogni agente rimuove quindi tutti i suoi behaviour e, come ultima operazione, informa il PMA prima di terminare la propria esecuzione.

PMA (PeerManagementAgent)

Il PMA del superpeer deve gestire l'interruzione dell'esecuzione degli agenti tipici di un peer e di quelli riservati ai superpeer. Inoltre deve occuparsi di interrompere tutti i peer della regione.

Alla ricezione del messaggio di removeR inviato dal server, il superpeer:

- invia i messaggi di pre-stopP all'agente RRA proprio del superpeer;
- invia i messaggi di stopP a tutti i peer;
- invia i messaggi di stopP a tutti gli agenti del peer locale;
- gestisce le risposte inviate dai peer;
- invia i messaggi di stopP a tutti gli agenti propri del superpeer(RRA, CA, RPA);
- gestisce le risposte inviate dagli agenti propri del superpeer (RRA, CA, RPA);
- notifica al PMA del server l'esito della procedura di rimozione della regione, in modo che aggiorni la struttura della rete di overlay se non è stato già fatto.

RRA(ResourceRegionAgent)

L'RRA riceve il messaggio di pre-stopP dal PMA appena quest'ultimo lo riceve dal server, in modo che il RRA possa subito mettere in atto alcuni accorgimenti, senza aspettare la terminazioni degli altri peer o degli altri agenti del superpeer. Il vero e proprio messaggio di stopP, invece, viene ricevuto solo quando tutti i peer hanno confermato al PMA del superpeer la loro terminazione.

- messaggio di pre-stopP

I behaviour del RRA non possono essere tutti rimossi subito: `RRARequestAnswerBhv` (che si occupa di rispondere a richieste intra-regione) e `RRAInsertNewSourceForTrancheBhv` (che inserisce un peer come nuova fonte per una tranche appena ottenuta) potrebbero essere ancora attivati da messaggi presenti in coda e inviati dai peer

della regione. Il behaviour `RRAExtraRegionSearchBhv` risponde alle richieste extra-regione formulate dai SearchAgent in transito sul superpeer e il behaviour `RRAWaitForExtraRegionResultBhv` attende la risposta dal CA che riferisca l'esito delle varie ricerche extra-regione: alla ricezione del messaggio di pre-switch off `RRAExtraRegionSearchBhv` risponderà a tutte le eventuali[22] richieste extra-regione formulate dai SearchAgent in transito evitando di comunicare i nomi dei peer della regione come possibili fonti per una tranche (nel caso lo fossero); `RRAWaitForExtraRegionResultBhv`, invece, continuerà la sua esecuzione come di consueto.

- messaggio di stopP

Alla ricezione del messaggio di stopP inviato dal PMA, tutti i peer hanno terminato la loro esecuzione, quindi tutti i behaviour citati possono essere rimossi senza problemi.
Il RRA può quindi terminare la propria esecuzione notificandolo al PMA del superpeer.

CA (CreatorAgent)

L'agente gestisce i SearchAgent necessari ad effettuare una ricerca extra-regione, comunica i risultati al RRA che ha fatto richiesta e collabora con i SearchAgent inviati dagli altri superpeer.

Quando il CA processa il messaggio di stopP, tutti i peer della regione hanno interrotto la loro esecuzione ed eventuali SearchAgent di passaggio sul superpeer della regione da interrompere non sono più presenti perché la struttura della overlay network è già stata aggiornata dal server e comunicata ai superpeer vicini che dirottano i SearchAgent sul superpeer corretto[23]. I SearchAgent creati dall'agente CA potrebbero essere ancora attivi su altri peer: consultando la tabella Searches del database, il CA (e precisamente il behaviour `CAWaitForSearchResultBhv`) può sapere quali ricerche extra-regione siano ancora in corso. Tuttavia, visto che i due SearchAgent relativi alla stessa ricerca quando si raggiungono lungo la rete di overlay inviano semplicemente un messaggio al CA creatore, non costituisce un problema l'interrompere il CA prima che tutti i SearchAgent

[22] La struttura della overlay network è già aggiornata dal server e i vicini del superpeer della regione da rimuovere inoltrano già i SearchAgent al superpeer corretto, evitando il superpeer che verrà interrotto. Eventuali SearchAgent che hanno iniziato la migrazione durante l'invio del messaggio di switch off da parte del server al superpeer della regione da interrompere, tuttavia, vedono tale superpeer come prossimo hop della rete di overlay

[23] Nel caso ne fosse rimasto qualcuno, questo verrebbe rifiutato con un messaggio di tipo `TypeConversationId.REFUSE_FOREIGN_AGENT` come schematizzato in Figura 3-13.

abbiano comunicato l'esito della ricerca. Il behaviour `CACreateSearchAgentBhv` non viene più attivato perché i peer sono tutti terminati; i behaviour `CAGetNeighboursAIDBhv` e `CARegisterForeignSerchAgentBhv` non vengono attivati perché i SearchAgent di passaggio sono sono più presenti sul superpeer.

Il CA può quindi terminare la propria esecuzione notificandolo al PMA.

SearchAgent(SearchAgent)

I SearchAgent creati dal CA del superpeer della regione da rimuovere sono in esecuzione su altri superpeer e termineranno la loro esecuzione in modo autonomo quando visiteranno lo stesso superpeer. Quei SearchAgent creati su altri superpeer e in visita su quello da terminare verranno fatti migrare sui "vecchi" vicini[24]. I "vecchi" vicini ottengono subito l'aggiornamento della struttura della rete di overlay (prima che la regione da rimuovere venga rimossa) e non inviano più i SearchAgent sul superpeer da rimuovere.

RPA(RendezvousProxyAgent)

L'agente RPA alla ricezione del messaggio di stopP termina semplicemente la sua esecuzione notificandolo al PMA: non si verifica nessuna complicazione poiché tutti i peer della regione per i quali sono necessari i servizi offerti dal RPA hanno terminato la loro esecuzione.

3.6.13 downR

La primitiva downR ricalca la primitiva removeR ma è concettualmente molto simile a downP: l'unica differenza sta nel fatto che non è solo un peer nello stato pFAILURE ad essere portato nello stato pSTOPPED, ma sono tutti i peer della regione che non si trovano nello stato pSTOPPED ad essere portati in questo stato; in questo modo anche il superpeer può portare la regione nello stato rREMOVED e renderla nuovamente avviabile tramite addR.

[24] I superpeer adiacenti sulla rete di overlay a quello della regione da rimuovere quando ne faceva ancora parte.

3.6.14 shutdownP/shutdownR

E' possibile definire anche la primitiva shutdown che può essere eseguita in locale su di un peer (provoca l'arresto del container e di tutti gli agenti) o su di un superpeer (provoca l'arresto anche del main container e di tutti gli agenti).

3.6.15 resetP/resetR

Potrebbe essere utile definire delle primitive di reset per peer e regioni che si occupino di eseguire lo shutdown del nodo ed un nuovo avvio del container (e/o main container) appena interrotti.

3.6.16 Primitive di supporto

Per poter eseguire le primitive di stato appena presentate e per offrire la possibilità di ottenere informazioni di stato sul sistema nel suo complesso o sui singoli nodi, è utile progettare alcune primitive di supporto.

La primitiva *getMainBootAddress* serve per ottenere da un nodo l'endpoint locale del proprio MainBoot affinchè possa essere utilizzato per completare quelle primitive di stato che lo richiedono (come ad esempio, startP, stopP o freezeP).

Le primitive *getPeerState* e *getAllPeerState* restituiscono lo stato di uno o più peer della regione coordinata dal superpeer che riceve tali primitive.

Con le primitive *getRegionState* e *getNodeState* un superpeer è in grado di restituire lo stato della propria regione o tutte le informazioni dell'*overlayNode* locale (nomi identificativi e indirizzi locali, stato della regione locale e informazioni di contatto sui vicini all'interno della overlay network). Il server, invece, quando riceve la primitiva getNodeState è in grado di restituire tutte le informazioni dell'overlayNode associato ad una qualsiasi regione.

La primitiva *overlayNetworkUpdate* viene usata dal server affinchè i superpeer che la ricevano possano aggiornare le informazioni relative ai propri vicini all'interno della overlay network.

CAPITOLO 4
Gestione dei guasti

La tolleranza ai guasti è un aspetto molto importante, soprattutto per i sistemi distribuiti. Si tratta di un argomento molto complesso ed articolato, dal momento che si devono tenere in considerazione molti fattori (soprattutto) esterni al sistema e poiché coinvolge pressochè tutti i suoi componenti: per poter essere affrontato in modo esaustivo, richiederebbe un'analisi ben più approfondita di quando s'intende fare all'interno di questo capitolo.

I prototipi realizzati finora ipotizzano l'assenza quasi totale di condizioni di fallimento; perciò si ritiene opportuno fornire alcune nozioni sulla classificazione dei guasti, procedere ad una prima analisi delle criticità del vecchio e del nuovo sistema, descrivere alcune ipotesi di soluzione per il rilevamento e la gestione di alcune tipologie di guasto, presentare alcune proposte per affrontare altri tipi di fallimento e introdurre il concetto di recovery.

4.1 Introduzione

La possibilità di nascondere il verificarsi di un guasto relativo ad un componente di un sistema, presuppone che si possa prima rilevarne la presenza, poi effettuarne la gestione e, se necessario, procedere al rispristino del componente che ha subito il fallimento.

In un sistema distribuito si può parlare di *guasto parziale* quando si guasta un suo componente e tale guasto, che può influenzare il corretto comportamento di alcuni componenti, tuttavia non viene percepito da altri. In sistemi di questo tipo sarebbe auspicabile un certo grado di tolleranza a

questa categoria di guasti, in modo da consentire il corretto funzionamento del sistema nel suo complesso.

Il concetto di tolleranza ai guasti, osservato da una prospettiva più ampia, è strettamente connesso alla definizione di *dependability* (Tanenbaum p. 316-317) (caratteristica attribuita ai cosiddetti "sistemi affidabili") che è associata ai concetti di[1]:

- disponibilità;
- affidabilità;
- sicurezza;
- manutenibilità.

Nei sistemi distribuiti basati su agenti, come il sistema che si sta progettando, si deve tenere in considerazione l'"autonomia" che caratterizza gli agenti stessi, concetto che viene trattato nel paragrafo *4.2* ed in cui risiede la sostanziale differenza fra i MultiAgentSystems (MAS) e i sistemi distribuiti che non sono basati sugli agenti.

Per quanto riguarda il problema della gestione dei guasti per il sistema da sviluppare, sarà necessario svolgere una fase di analisi preventiva atta a delineare i punti critici del sistema ed a porre le basi per le ipotesi di soluzione che - a seconda dei casi - verranno effettivamente implementate (totalmente o in parte).

Prima di tale attività, tuttavia, si reputa utile riportare in modo conciso (nel paragrafo seguente) alcune considerazioni su alcuni schemi di classificazione dei guasti.

[1] Un sistema è:

- *disponibile*: quando è possibile utilizzarlo immediatamente ed in qualsiasi momento;
- *affidabile*: quando può funzionare in maniera continuativa senza guasti;
- *sicuro*: quando, nel caso smetta di funzionare per un certo periodo di tempo, breve o protratto che sia, non accade niente di irrimediabile;
- *manutenibile*: quando può essere riparato con facilità dopo il verificarsi di un guasto.

4.2 Classificazione dei guasti

Sono stati proposti diversi schemi di classificazione con lo scopo di definire le categorie di guasto e di favorire la comprensione della loro reale gravità. In questo paragrafo se ne presenta un primo abbastanza sintetico e dopo se ne riporta uno un po' più dettagliato. Successivamente si discute del concetto di "autonomia" degli agenti e, sulla base delle considerazioni riportate in (Potiron, et al. p. 5, par.3.1), si presenta brevemente una nuova sotto-categoria di guasti: i "behavioural faults".

4.2.1 Sistemi distribuiti "standard"

Una semplice categorizzazione dei guasti è quella descritta in (Tanenbaum p. 318-320) che viene riassunta nella seguente tabella:

Tipo di guasto	Descrizione	
Fail-stop	Il processo si arresta ma funziona correttamente fino a quel momento; gli altri processi sono in grado di rilevare questo stato.	
Crash	Il processo si arresta ma funziona correttamente fino a quel momento; gli altri processi potrebbero non essere in grado di rilevare questo stato.	
Omissione	in ricezione:	Il processo non riceve i messaggi in ingresso.
	in invio:	Il processo non riesce ad inviare messaggi.
Errore nella temporizzazione	Il tempo di risposta del processo ad una richiesta è fuori dall'intervallo di tempo specificato (possibile ritardo nella comunicazione).	
Fallimento nella risposta	errore nel valore:	Il valore della risposta non è corretto.
	errore nella transizione di stato:	Il processo devia dal flusso di controllo corretto.
Guasto arbitrario (Byzantine)	Il processo può produrre risposte arbitrarie in momenti arbitrari.	

Tabella 4-1 – Schema di classificazione dei guasti.

Una declinazione molto più dettagliata del concetto di "failure" è fornita in (Potiron, et al. p. 5, par.3.1)[2] e di seguito se ne riporta uno schema riassuntivo:

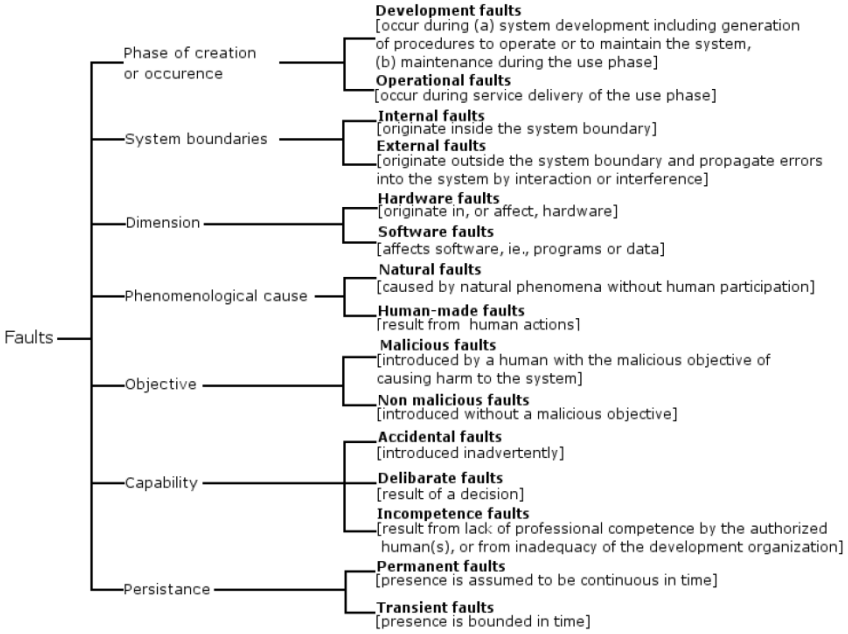


Figura 4-1 –Classificazione dei guasti (attributi) [schema dettagliato].

4.2.2 MultiAgentSystems (MAS)

Un MultiAgentSystem, in quanto sistema distribuito, è costituito da più componenti che cooperano per raggiungere un obiettivo comune. Data la sua natura, quindi, può essere soggetto a guasti appartenenti alle 3 macro-categorie seguenti:

1. development fault;
2. physical fault;
3. interaction fault.

Tuttavia ciò che differenzia un MAS da un sistema distribuito "standard" è la presenza degli agenti: in particolare ogni agente gode di un certo grado di autonomia e non è sempre in grado di prevedere il

[2] L'argomento viene affrontato in maniera approfondita e la trattazione porta alle considerazioni riportate nel paragrafo *4.2.2*.

comportamento degli altri agenti. Questa peculiarità può comportare dei malfunzionamenti che non possono essere catalogati esclusivamente come (cfr. *Figura 4-1*, p.104) operational fault (poiché non sono dovuti solo al servizio che ne permette l'esecuzione) o come development fault (poiché il "comportamento" di un agente varia a seconda della sua esecuzione).

Per questo motivo in (Potiron, et al. p. 5, par.3.1) si introduce una nuova macro-categoria che comprende tutti quei guasti associati al "comportamento autonomo"[3] degli agenti:

4. behavioural fault.

Inoltre si propone un nuovo valore per l'attributo "*Phase of creation or occurrence*" della *Figura 4-1* (p.104): "*Autonomy faults*".

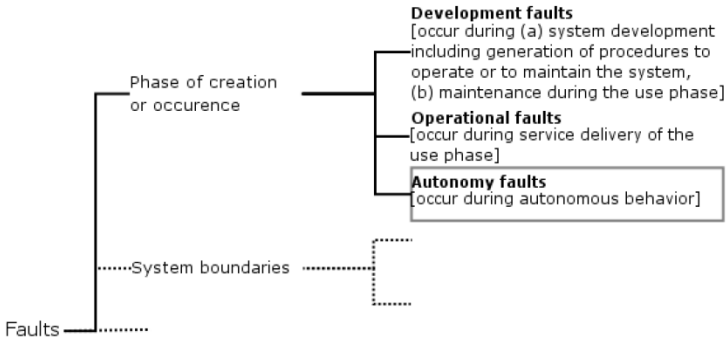


Figura 4-2 – "Autonomy faults" (cfr. Figura 4-1)

Un'interessante considerazione riguarda la possibilità di prevenire questi "*Automomy faults*" (a patto che siano "*non-malicious*") tramite dei messaggi come, ad esempio, una richiesta di delay, nel caso un agente non sia in grado di fornire nei tempi previsti una risposta ad una precedente richiesta, oppure una richiesta di tipo CANCEL per le nuove richieste, nel caso l'agente non sia in grado di elaborare tutte quelle già pervenute.

[3] Per comportamento autonomo di un agente s'indende, ad esempio, la sua capacità di decidere, in base alla situazione esistente, di:

- non rispondere ad una richiesta;
- provocare un errore che rende un altro agente incapace di svolgere i suoi compiti;
- non eseguire le operazioni previste.

4.3 Analisi delle criticità del sistema

Progettare una struttura coerente e pervasiva per la rilevazione e la gestione dei guasti per l'intero sistema si rivela un'attività che non è possibile inserire *as is* all'interno del progetto attuale.

Si ritiene opportuno effettuare una scelta su quali tipi di guasti considerare per i seguenti motivi:

a) ci si trova a dover realizzare nuove funzionalità per un sistema esistente e ciò significa dover tenere in considerazione non solo i componenti che si stanno progettando, ma anche tutti i componenti già sviluppati;
b) le tipologie di guasti che si possono verificare sono molto variegate e si rivela quasi impossibile tentare di rilevarli e gestirli tutti.

Di seguito si elencano alcuni degli ambiti per cui potrebbe rivelarsi sensata una prima attività di studio volta alla rilevazione ed alla gestione dei guasti:

- interazione (a livello di container JADE) fra i container dei peer ed i main-container dei superpeer;
- comunicazioni fra gli agenti di peer situati all'interno della stessa regione (che appartengono, nello specifico, allo stesso dominio di collisione);
- comunicazioni fra agenti di peer situati in regioni distinte connesse attraverso Internet;
- interazioni connesse alle funzioni di persistenza offerte dall'add-on Persistence;
- crash dei nodi.

Visto che il nucleo di questo progetto consiste nel dotare il sistema della possibilità di funzionare attraverso Internet, si decide di concentrare l'attenzione sui "*Guasti di omissione*" e sugli "*Errori di temporizzazione*" - descritti nella *Tabella 4-1* (p.103) - con particolare riferimento (terzo punto dell'elenco precedente) alle comunicazioni inter-regione descritte[4] nel

[4] Si citano, ad esempio, le comunicazioni necessarie per la gestione dei trasferimenti, per la gestione delle primitive o per i trasferimenti veri e propri delle tranche.

CAPITOLO 2. Queste due tipologie di guasti sono (sotto certi aspetti) abbastanza assimilabili, e permettono di essere rilevati con strategie simili[5].

4.3.1.1 Aspetti critici

Si elencano di seguito gli aspetti specifici su cui si decide di concentrare l'attenzione per la rilevazione e la gestione dei guasti:

1. raggiungibilità dei peer presenti in regione e dei superpeer che compongono l'overlay network (soprattutto per quanto riguarda la gestione delle primitive di amministrazione);
2. gestione ed esecuzione delle primitive di amministrazione e di stato (cambiamento di stato di peer e regioni):
   a) a livello dei singoli behaviour (nel caso in cui non tutti i behaviour di un agente confermano la propria disponibilità al cambiamento di stato dell'agente);
   b) a livello dei peer (nel caso in cui non tutti gli agenti di un peer confermano la propria disponibilità al cambiamento di stato del peer);
   c) a livello delle regioni (nel caso in cui non tutti i peer di una regione confermano la propria disponibilità al cambiamento di stato della regione);
3. gestione dei trasferimenti extra-regione:
   a) interazione UMA-UA-DMA (selezione delle fonti);
   b) interazione DMA-MA (selezione delle fonti - propose_ticket);
   c) interazione DA-UA (per il setup della connessione);
   d) interazione HA-RPA (per la fase di rendezvous della procedura di tcpHolePunching);
   e) interazione HoleClientBhv-HoleServerBhv (per l'instaurazione della vera e propia connessione TCP ottenuta col TCP Hole Punching);
4. effettivo trasferimento extra-regione di una tranche:
   a) interazione DA-UA (per il trasferimento vero e proprio);

[5] Un meccanismo rudimentale ma relativamente efficace consiste nell'utilizzo dei *timeout*, mentre una strategia più elaborata può prevedere il rilevamento dei guasti tramite *gossiping* (cfr. paragrafo 4.4).

b) interazione DA-UMA (per lo "sblocco" dell'agente UMA della fonte in caso di errore durante il trasferimento).

4.3.1.2 Punti di forza del nuovo sistema

Nel *CAPITOLO 3* sono stati introdotti stati e primitive proprio per facilitare la gestione dei guasti.

Per peer e regioni sono stati creati gli stati pFAILURE e rFAILURE e le primitive downP e downR; si è parlato anche delle primitive resetP e resetR (che andrebbero ad operare direttamente sui container dei peer e sui main-container del superpeer)

Inoltre è stato presentato l'add-on Persistence, che offre le primitive SaveAgent e LoadAgent (per lo stato dei singoli agenti) e che permette la realizzazione delle primitive freezeP e thawP (a livello di peer).

Grazie a questi strumenti ed alla struttura prevista per i singoli behaviour potrebbero essere teoricamente realizzate, magari in un secondo momento, anche strategie come il backward recovery (cfr. 4.5.2).

4.4 Rilevamento e gestione dei guasti

Se non è possibile prevenire il vericarsi di un guasto, prima di poterlo gestire è indispensabile poterlo rilevare. Di seguito si indicano tre strategie che possono essere utilizzate per il rilevamento dei guasti e, successivamente, si presentano delle ipotesi di soluzione da implementare in relazione ai vincoli del sistema (cfr. *CAPITOLO 5*).

4.4.1 Strategie per il rilevamento dei guasti

4.4.1.1 Timeout

Un meccanismo molto semplice ma che mostra sicuramente efficacia è quello dei *timeout*; tuttavia utilizzare esclusivamente uno strumento del genere può rivelarsi controproducente dal momento che è riduttivo ipotizzare il verificarsi di un guasto solo in occasione di una mancata risposta ad un messaggio.

4.4.1.2 Gossiping

Un'alternativa è rappresentata dal *gossiping* che consiste in un approccio che coinvolge più processi nella verifica delle funzionalità di uno di essi. In particolare il *gossiping* prevede che i processi si inviino reciprocamente e periodicamente dei messaggi in cui elencano i servizi che sono in grado di offrire.

In questo modo un processo, analizzando le informazioni sparse attraverso la rete tramite gossiping, è in grado di ipotizzare se un altro processo ha subito un fallimento: più nello specifico, soprattutto nel caso in cui si tratti di peer o nodi generici di un sistema distribuito, per escludere errori legati all'infrastruttura di rete, il nodo che si è accorto del possibile guasto può chiedere conferma ai vicini del nodo che si ipotizza essere in una condizione di guasto, affinchè controllino se anche loro riscontrano la stessa condizione di errore.

4.4.1.3 Un esempio di fault-tolerant MAS

In (Chen, et al., 2008) viene presentato un metodo per la rilevazione e la gestione dei guasti ideato proprio per i Multi Agent Systems. Questa strategia prevede, in particolare, che un certo task da eseguire T sia suddiviso in stage S_i e che un agente mobile abbia il compito di portare a termine le operazioni di ogni stage su un host diverso.

La soluzione proposta da Chen prevede essenzialmente due tipologie di agenti:

1. un "actual agent": che svolge le operazioni previste per il singolo stage;
2. un "witness agent": che svolge un ruolo di supervisore (testimone) per le attività di migrazione dell'actual agent e per lo svolgimento delle operazioni previste.

Inoltre il sistema deve mettere a disposizione, per ogni stage *i*, una o più strutture dati per memorizzare in modo persistente

- il *log* delle operazioni eseguite (Log List - LL);
- i *checkpoint* (salvataggi intermedi dello stato dell'eleborazione) (Checkpoint Box - CB);
- i *messaggi* scambiati fra gli agenti (Mail Box - MB).

Macroscopicamente accade che l'actual agent si sposta da un host all'altro eseguendo le operazioni richieste, memorizzando l'esito intermedio dell'elaborazione tramite checkpoint (dati e stato dell'actual agent) e tenendo informati i witness agent dei progressi ottenuti che vengono memorizzati anche localmente nella Log List (sono comprese le informazioni sulla migrazione da/verso gli host).

Dal momento che l'actual agent memorizza tramite dei checkpoint lo stato dell'elaborazione eseguita in ogni stage, è possibile gestire il verificarsi di un guasto in uno stage ripristinando, nel caso migliore, l'elaborazione effettuata in quello immediatamente precedente, perdendo i progressi di un solo stage.

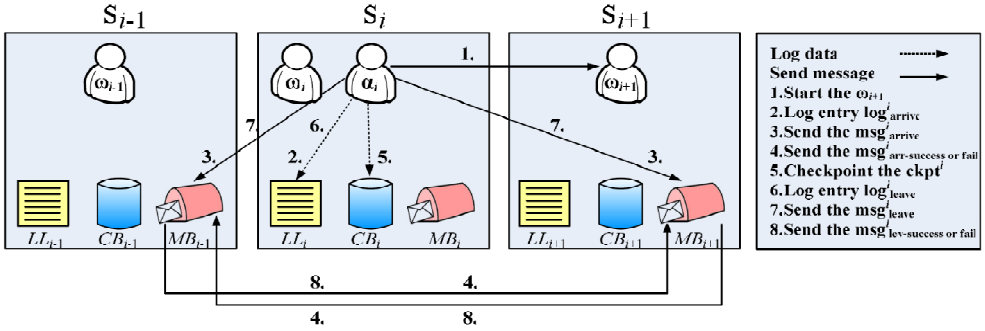


Figura 4-3 – Esempio di fault-tolerant MAS

In *Figura 4-3* è rappresentato lo schema di funzionamento di questo sistema (caso generico). Per lo stage *i* si utilizza la seguente convenzione:

- α_i (actual agent)
- ω_i (witness agent)
- LL_i (log list)
- CB_i (checkpoint box)
- MB_i (mail box)

CASO BASE: S_1

Step1: L'actual agent α_1 inizia la propria esecuzione nel primo stage S_1 solo quando è stato avviato il witness agent ω_1; viene avviato ω_2 nello stage S_2 (che fa partire un timer);

Step2: α_1 salva in LL_1 il log^1_{arrive} (inizio dell'esecuzione in S_1);

Step3: α_1 invia alla MB_2 il messaggio msg^1_{arrive} (inizio elaborazione in S_1) e ω_2, dopo aver reagito all'arrivo del messaggio, fa partire un altro timer;

Step4: se ω_2, entro il timeout avviato nello step1, riceve da α_1 il messaggio msg^1_{arrive} , invia un $msg^1_{arr\text{-}success}$ a MB_1 (inizio elaborazione nello stage S_1 confermato); altrimenti invia un $msg^1_{arr\text{-}fail}$ (errore durante l'inizio dell'elaborazione);

Step5: Quando α_1 termina il task nello stage S_1, memorizza un ckeckpoint in CB_1;

Step6: α_1 salva in LL_1 il log^1_{leave} (fine dell'esecuzione in S_1);

Step7: α_1 invia alla MB_2 il messaggio msg^1_{leave} (fine elaborazione in S_1) e ω_2 reagisce all'arrivo del messaggio;

Step8: se ω_2, entro il timeout avviato nello step3, riceve da α_1 il messaggio msg^1_{leave} , invia un $msg^1_{lev\text{-}success}$ a MB_1 (fine elaborazione in S_1 confermata); altrimenti invia un $msg^1_{lev\text{-}fail}$ (errore durante la terminazione dell'elaborazione nello stage S_1).

CASO GENERICO: S_i (*Figura 4-3*)

Dopo che l'elaborazione nello stage S_{i-1} è terminata, i witness agent ω_{i-1} e ω_i sono già in esecuzione.

Step1: Prima che l'elaborazione cominci effettivamente nello stage S_i, viene avviato il witness agent ω_{i+1} che fa partire un timer;

Step2: α_i salva in LL_i il log^i_{arrive} (inizio dell'esecuzione in S_i);

Step3: α_i invia a MB_{i-1} e a MB_{i+1} il messaggio msg^i_{arrive} (inizio elaborazione in S_i); ω_{i-1} e ω_{i+1}, dopo aver reagito all'arrivo del messaggio, fanno partire un altro timer;

Step4: se ω_{i-1} o ω_{i+1} ricevono da α_1 entro il timeout previsto il messaggio msg^i_{arrive} , inviano un $msg^i_{arr\text{-}success}$ rispettivamente a MB_{i+1} e MB_{i-1} (inizio elaborazione nello stage S_i confermato); altrimenti inviano un $msg^i_{arr\text{-}fail}$ (errore durante l'inizio dell'elaborazione);

Step5: Quando α_i termina il task nello stage S_i, memorizza un ckeckpoint in CB_i;

Step6: α_i salva in LL_i il log^i_{leave} (fine dell'esecuzione in S_i);

Step7: α_i invia a MB_{i-1} e MB_{i+1} il messaggio msg^i_{leave} (fine elaborazione in S_i); ω_{i-1} e ω_{i+1} reagiscono all'arrivo del messaggio;

Step8: se ω_{i-1} o ω_{i+1} ricevono da α_i entro il timeout avviato nello step3 il messaggio msg^i_{leave} , inviano un $msg^i_{lev\text{-}success}$ rispettivamente a MB_{i+1} e MB_{i-1} (fine elaborazione in S_i confermata); altrimenti invia un $msg^i_{lev\text{-}fail}$ (errore durante la terminazione dell'elaborazione nello stage S_i).

Quando uno dei witness agent si accorge del mancato arrivo di un messaggio entro il tempo predefinito, prima ci si accerta della presenza di un guasto controllando l'opinione dell'altro witness agent e poi si provvede a riprisitnare (compatibilmente con la tipologia di operazioni richieste) l'elaborazione sull'ultimo stage per il quale è stato memorizzato correttamente il checkpoint.

4.4.2 Ipotesi di soluzione

Per il rilevamento dei guasti non direttamente associabili al lancio di eccezioni, si decide di avvalersi del primo metodo presentato, quindi dell'uso dei timeout. Nonostante non rappresenti la soluzione più efficace, la si è reputata sufficientemente funzionale ai fini degli obiettivi individuati fin dalle prime fasi del progetto e si ritiene che la relativa semplicità di implementazione di questa strategia costituisca un valore aggiunto. Nulla vieta, nelle eventuali implementazioni future del sistema, di elaborare una soluzione che si basi su metodi differenti.

Per quanto riguarda l'approccio che si intende adottare per la gestione dei guasti elencati nel paragrafo *4.3.1.1*, si tenga presente che lo scopo principale è quello di individuare dei protocolli di comunicazione adeguati: più precisamente si mira a progettare[6] dei protocolli di interazione che considerino l'eventualità del verificarsi di errori di comunicazione tra le entità coinvolte.

Di seguito si effettua un'analisi abbastanza approfondita che, per esigenze di chiarezza, fa riferimento ai capitoli 2, 3 (progettazione) e 5 (implementazione).

[6] Nel caso sia fattibile, ci si limiterà a migliorare quelli esistenti.

4.4.2.1 Raggiungibilità

Affinchè i peer di una regione o i superpeer che compongono l'overlay network siano raggiunti dalle primitive di amministrazione, è necessario che siano correttamente connessi al sistema.

Ciò si traduce nella necessità, da parte di un peer, di conoscere a priori gli estremi[7] del superpeer della regione in cui è inserito e, da parte di un superpeer, quelli del sever[8].

Affinchè il superpeer di una regione sia consapevole della presenza di un nuovo peer verso cui è possibile inoltrare delle primitive di amminstrazione (ad esempio startP), è necessario che tale peer (il suo PMA) informi il PMA del superpeer appena il container del peer viene connesso al main container del superpeer, comunicando l'endpoint locale a cui è reperibile il proprio MainBoot. Allo stesso modo, quando viene avviato il MainBoot di un superpeer, il suo agente PMA deve contattare il PMA del server ed informarlo circa l'endpoint pubblico a cui è reperibile il proprio MainBoot.

La raggiungibilità di un nodo (peer o superpeer) target di una primitiva avviene:

- a cura del superpeer, prima di inviarla ad un peer;
- a cura del server, prima di inviarla al superpeer di una regione.

Il superpeer o il server inviano una primitiva getMainBootAddress con cui chiedono all'agente PMA del target l'endpoint del thread MainBoot e controllano che la risposta arrivi entro un periodo di tempo predeterminato.

Il controllo della raggiungibilità dei peer e dei superpeer potrebbe essere effettuato in maniera periodica (e a prescindere dall'esecuzione delle primitive di amministrazione) tramite l'invio di messaggi di tipo "alive". Con l'aumentare delle dimensioni del sistema potrebbe essere sensato, per i peer, introdurre una gerarchia o, comunque, una rete logica che permetta di rilevare la non raggiungibilità di un nodo riducendo il numero di messaggi di "alive" inviati dal nodo stesso. Evetualmente si potrebbe ricorrere ad altre strategie di rilevamento del guasto, ad esempio al gossiping (cfr. *4.4.1.2*)

[7] Identificativo locale, indirizzo IP locale e pubblico.

[8] Identificativo globale, indirizzo IP pubblico.

4.4.2.2 Gestione delle primitive di amministrazione

Durante l'elaborazione di una primitiva di amministrazione vengono coinvolti diversi componenti e quindi, per rilevare il verificarsi di un errore di omissione/temporizzazione, è necessario un controllo su più livelli: behaviour/agente, peer e superpeer.

Il modello ideato fa riferimento al *Two phase commit* (2PC)[9] che viene descritto brevemente di seguito. L'esecuzione della transazione prevede la presenza di un processo coordinatore che invia un messaggio di tipo VOTE_REQUEST ai processi che rispondono con un VOTE_ABORT o con un VOTE_COMMIT, esprimendo, rispettivamente, in modo negativo o posivito la loro disponibilità al commit della transazione. Il coordinatore, se riceve le conferme da tutti gli altri processi, si porta nello stato COMMIT ed invia un messaggio GLOBAL_COMMIT altrimenti, se riceve anche un solo VOTE_ABORT, si porta nello stato ABORT ed invia un messaggio GLOBAL_ABORT.

La procedura che si intende utilizzare prende spunto dal *2PC* e prevede, come detto in precedenza, l'utilizzo di timeout.

Il coordinatore (a livello di behaviour/agenti, peer o superpeer) quando riceve la primitiva da inoltrare, la inoltra ai processi sottoposti e avvia un timer. Se tutti i processi confermano l'esecuzione delle operazioni richieste dalla primitiva entro la scadenza del timeout, il coordinatore notifica l'esito positivo dell'elaborazione a quello di livello superiore[10]. In caso contrario:

- se riceve un messaggio di failure da uno dei processi ma il timer non è ancora scaduto, attende comunque la scadenza del timeout;
- se scade il timeout, invia la notifica dell'errore al coordinatore di livello superiore.

Questo accorgimento è stato adottato per dare la possibilità ad altri processi (in grado di farlo) di portare a termine le operazioni richieste dalla primitiva senza essere condizionati dall'errore riscontrato da uno di essi; inoltre in questo modo il coordinatore è in grado di restituire una lista

[9] Cfr. (Tanenbaum p. 349-354). A differenza del 2PC, il *Three Phase Commit* (3PC) è caratterizzato da un maggiore tolleranza ai guasti (soprattutto al crash del coordinatore che compromette il 2PC) in quanto sfrutta la presenza di un ulteriore stato detto PRECOMMIT.

[10] Ad esempio il PMA del peer informa quello del superpeer, o il PMA del superpeer informa quello del server.

completa di tutti i processi che hanno riscontrato problemi durante l'elaborazione.

Per i seguenti tre punti si faccia riferimento ai paragrafi *5.2.3.2*, *5.2.3.3*, *5.2.3.4*, alla *Figura 5-10* (p.143), alla *Figura 5-11* (p.145) ed alla *Tabella 5-5* (p.146).

A livello dei behaviour

Il behaviour "coordinatore"[11] di ogni agente controlla, con l'ausilio di un behaviour Ticker avviato su un thread indipendente (Threaded Behaviour), che tutti i behaviour dell'agente confermino la propria disponibilità al cambiamento di stato dell'agente: tale conferma consiste in un messaggio di tipo BEHAVIOUR_STATE_UPDATE e giunge quando il behaviour ha effettuato tutte le operazioni che era previsto eseguisse prima del cambiamento di stato.

Se non tutti i behaviour dell'agente hanno dato conferma al behaviour "coordinatore" prima che il behaviour Ticker gli abbia inviato un messaggio BEHAVIOUR_STATE_UPDATE_TIMEOUT, allora il behaviour "coordinatore" informerà il PMA dell'agente del fallimento dell'operazione richiesta; altrimenti, notificherà che l'elaborazione della primitiva ha avuto esito positivo (questa comunicazione avviene con messaggi di tipo AGENT_STATE_UPDATE).

A livello dei peer

La procedura è identica a quella descritta nel caso precedente, ma cambiano gli attori: il coordinatore è l'agente PMA del peer e i processi da controllare sono gli altri agenti (i vari behaviour di ogni agente fanno capo al behaviour "coordinatore" dell'agente).

I messaggi inviati dagli agenti al PMA sono di tipo AGENT_STATE_UPDATE, quello relativo alla scadenza del timeout è di tipo AGENT_STATE_UPDATE_TIMEOUT e quello inviato dall'agente PMA del peer all'agente PMA del superpeer è di tipo PEER_STATE_UPDATE.

[11] Cfr. paragrafi *3.6.3* e *5.2.1.7* – xxxxReceiveAdminPrimitiveBhv.

A livello delle regioni

Anche in quest'ultimo caso la procedura rispetta il modello presentato. Il coordinatore è il PMA del superper e i processi da controllare sono i peer (che fanno capo all'agente PMA locale). I messaggi inviati dagli agenti al PMA sono di tipo PEER_STATE_UPDATE, quello relativo alla scadenza del timeout è di tipo PEER_STATE_UPDATE_TIMEOUT e quello inviato dall'agente PMA del peer all'agente PMA del superpeer è di tipo REGION_STATE_UPDATE.

4.4.2.3 Gestione dei trasferimenti extra-regione

L'attività di gestione dei trasferimenti extra regione consiste nella selezione delle fonti, nella configurazione del trasferimento e, all'occorrenza, nella connessione diretta (procedura di TCP Hole Punching suddivisa nelle fasi di rendezvous e di instaurazione della connessione TCP).

DMA-MA-UMA

I tre agenti svolgono la fase di selezione delle fonti (come descritto in *2.2.3*) ed è già previsto un meccanismo che considera il caso in cui il DMA non risponda alla proposta inviatagli dall'agente MA su richiesta dell'UMA.

DMA-MA

Il DMA attende che uno fra gli MA gli notifichi (messaggio PROPOSE_TICKET) che la fonte contattata sia disponibile per l'upload della tranche. Nel caso non arrivi nessuna risposta degli MA, è già previsto un meccanismo di timeout[12] che rileva questa condizione, rende URGENT la richiesta per quella tranche e dà inizio alla procedura prevista.

DA-UA

L'interazione fra i due agenti UA e DA per quanto riguarda il setup della connessione per il trasferimento extra-regione di una tranche è necessaria per stabilire se sia o meno possibile sfruttare una connessione TCP utilizzata in precedenza.

[12] Specificatamente realizzato da un behaviour DMACheckUrgency che viene eseguito dal DMA su un thread indipendente (come Threaded Behaviour).

La necessità del riutilizzo della connessione è stata espressa in precedenza nel *2.3.5* e si rimanda al punto successivo "HA-RPA" per ulteriori delucidazioni.

Nel caso sia presente una tale connessione, l'UA effettuerà una richiesta al DA per conoscere la sua disponibilità a riutilizzarla; nel caso il DA neghi il consenso, i due agenti procederanno ad instaurarne una nuova utilizzando la tecnica del TCP Hole Punching: delegano la fase di rendezvous all'agente HA (cfr. punto "HA-RPA") e successivamente si occupano della vera e propria instaurazione della connessione (cfr. punto "HoleServerBhv-HoleClientBhv" di questo paragrafo); nel caso il DA non risponda entro un tempo prestabilito, l'agente UA sbloccherà l'UMA (messaggio di tipo COMPLETE_DOWNLOAD) scongiurando l'eventualità che quest'ultimo agente rimanga bloccato in attesa di un messaggio dal DA.

Per maggiori dettagli si veda il paragrafo *5.4.1*, la *Figura 5-35* (p.206) e la *Figura 5-36* (p.208).

HA-RPA

Le comunicazioni tra questi due agenti avvengono durante la fase di rendezvous della procedura di TCP Hole Punching (cfr. 2.5 per la definizione del problema e *5.4.3* per l'implementazione della procedura).

L'agente HA della fonte (risp. del richiedente) deve interagire con l'agente RPA della regione del richiedente (risp. della fonte) attraverso una connessione TCP, affinchè l'RPA sia in grado di rilevare l'endpoint pubblico dell'HA. L'agente HA deve associare tale connessione con l'RPA allo stesso endpoint locale che verrà utilizzato dall'agente UA (risp. dal DA) per instaurare la connessione attraverso la quale avverrà l'effettivo trasferimento della tranche.

E' possibile che si verifichino le situazioni seguenti:

a) l'HA rileva un errore di tipo "Address already in use";
b) l'HA non riesce ad associare - `bind()` - la connessione alla porta locale;
c) l'HA non riesce a comunicare con l'RPA nel tempo previsto;
d) l'HA del peer fonte non riceve in tempo (tramite il proprio RPA) una risposta dall'HA del peer richiedente;
e) l'HA viene informato che l'HA dell'altro peer ha riscontrato un errore.

Gli errori appena elencati possono essere rilevati tramite timer che verifichino la scadenza di timeout o tramite il sollevamento di eccezioni specifiche.

L'HA può prevenire alcuni di essi lasciando attiva (parametro KEEP_ALIVE) la connessione TCP con l'RPA (in modo simile a quanto fanno gli agenti UA e DA). Così facendo:

1. si evita che per ogni trasferimento extra-regione l'agente HA debba instaurare una nuova connessione verso l'RPA dell'altra regione;
2. si fa in modo che, se un peer deve ricevere delle tranche da più fonti situate all'interno della stessa regione, sia necessaria una sola connessione TCP tra l'HA del peer e l'agente RPA della una regione[13];
3. si fa in modo che il NAT mantenga l'associazione tra l'enpoint privato usato dall'HA e l'endpoint pubblico associato dal NAT stesso;
4. si escludono eventuali problemi connessi alle tempistiche di rilascio dei socket da parte del sistema operativo

L'occupazione di risorse è maggiore rispetto al caso in cui la connessione TCP viene chiusa al termine della procedura di rendezvous (o, nel caso degli agenti UA e DA, al termine del trasferimento della tranche), tuttavia si ritiene che sia più conveniente optare per una soluzione del genere anche perché si riducono gli inconvenienti che possono derivare dal riutilizzo di uno stesso socket per due connessioni successive[14] e perché si riducono i tempi richiesti per le successive procedure di rendezvous verso lo stesso RPA.

Nel caso si verifichi uno degli errori elencati in precedenza, l'HA può:

a) provare ripetutamente (fino ad un numero massimo di tentativi) la riconnessione all'RPA;
b) può eliminare (interrompendo la connessione associata ad esso) l'eventuale socket memorizzato in precedenza con cui comunicava con l'RPA ed informare l'agente UA (se si tratta

[13] Le connessioni TCP fra i peer, invece, sono vengono stabilite direttamente fra gli stessi peer.

[14] Può capitare, tra l'altro, che il sistema operativo stenti a rilasciare subito realmente il socket appena chiuso.

dell'agente HA del peer fonte) o l'agente DA (se si tratta dell'agente HA del peer richiedente) affinchè informino l'agente HA dell'altro peer;

c) può cambiare la porta locale a cui associare le successive connessioni (dovrà informare di ciò gli agenti UA e DA che si occuperanno di stabilire la connessione diretta per il trasferimento della tranche).

I problemi derivanti dall'occupazione di risorse (legata al mantenere attive le connessioni tra RPA e HA e tra DA e UA) si possono risolvere predisponendo un processo che controlli le connessioni attive e chiuda quelle che non vengono utilizzate da maggior tempo.

HoleClientBhv-HoleServerBhv

Come accennato nel *CAPITOLO 2*, questi due componenti si occupano di instaurare la vera e propria connessione TCP diretta tra i due peer che devono effettuare il trasferimento di una tranche.

Il behaviour lato-server si mette in attesa controllando periodicamente se il suo compagno lato-client (sullo stesso peer) ha già completato la connessione verso il behaviour lato-server dell'altro peer; il behaviour lato-client tenta periodicamente di connettersi al server dell'altro peer gestendo le eccezioni che derivano dal drop dei pacchetti SYN effettuato dal NAT della regione dell'altro peer.

Entrambi i behaviour svolgono le loro attività per un tempo massimo predeterminato e ne comunicano l'esito all'agente che li ha creati (UA se sono sul peer fonte, DA se sono sul peer richiedente).

Sarà compito degli agenti UA e DA di ciascun peer stabilire attraverso il controllo dei due behaviour se sia stata o meno creata la connessione diretta. In caso negativo, ciascun peer è in grado di rilevare la condizione di errore e di proseguire con le sue attività: il DA del richiedente sblocca l'UMA della fonte e torna nel proprio stato di default; l'UA torna nel proprio stato di default.

Per ulteriori dettagli (anche dal punto di vista implementativo) si veda il paragrafo *5.4.4*.

SearchAgent

La ricerca extra-regione deve essere garantita nel caso in cui la struttura dell'overlay network venga modificata dalle primitive addR e removeR. A tal proposito si rimanda ai paragrafi *3.6.12* e *5.3.2.2*.

4.4.2.4 Trasferimento extra-regione

DA-UA

L'interazione fra gli agenti UA e DA avviene anche durante l'effettivo trasferimento di una tranche. Il DA si occupa di questa operazione attraverso il suo behaviour DADownloaderTrancheBhv che viene avviato su un thread indipendente (come Threaded Behaviour) e lascia l'agente libero di eseguire il download da altre fonti.

E' compito proprio di questo behaviour rilevare l'errore di comunicazione che avviene durante il trasferimento della tranche e gestirlo sbloccando l'UMA della fonte.

DA-UMA

In condizioni standard è l'agente DA che, al termine di un download, sblocca l'UMA della fonte. Nel caso il DA sia in grado di rilevare un errore nel trasferimento, sbloccherà l'UMA come di consueto; nel caso, invece, in cui il DA non riesca a rilevarlo (ad esempio perché si è verificato un gasto che non gli permette di eseguire le operazioni necessarie) l'UMA rimarrebbe bloccato e non sarebbe in grado di servire le altre richieste di upload. Per questo si può dotare l'UMA di un timer che permetta all'UMA di sbloccarsi autonomamente se non arriva alcune notifica (COMPLETED_DOWNLOAD) dall'agente DA.

4.5 Altre proposte: considerazioni

4.5.1 Fault tolerance per guasti di tipo crash

Per la rilevazione e la gestione del crash di peer o superpeer si devono considerare in modo accurato le numerose interazioni che avvengono tra i singoli agenti, anche quando appartengono allo stesso peer. Infatti è necessario che lo stato complessivo del sistema venga preservato e che il crash di un nodo non si ripercuota sugli altri.

Per la rilevazione di questo tipo di guasto potrebbe essere utilizzata la tecnica del gossiping in quanto più scalabile e più efficiente all'aumentare delle dimensioni del sistema, mentre per la gestione si dovrebbe valutare caso per caso; per quanto riguarda il ripristino si potrebbe ricorrere, in funzione della causa che ha provocato il crash, alle primitive resetP e resetR che operano a livello di container e main container e che riuscirebbero, quindi, a ripristinare anche la piattaforma JADE e le connessioni da essa utilizzate[15]. Tuttavia bisognerebbe porre la massima attenzione nell'analisi dei rapporti che intercorrono fra il nodo che ha subito il crash e gli altri nodi.

Ad esempio, nel caso si tratti di un superpeer, bisognerà innanzitutto considerare come debba avvenire il rilevamento del crash e, in secondo luogo, come gestire la struttura della overlay network. Il rilevamento potrebbe avvenire a cura dei due superpeer delle regioni adiacenti e la ristrutturazione della overlay network a cura del server.

Caso particolare è rappresentato dal crash del server che lascerebbe il sistema attuale senza un coordinatore. Nello specifico, sempre a titolo esemplificativo, si potrebbe decidere di implementare una gestione distribuita della struttura della overlay network in modo che il crash del server comporti, eventualmente, solo l'impossibilità di aggiungere nuove regioni alla overlay network[16] senza pregiudicare la gestione del guasto di una o più regioni già connesse al sistema.

Un'altra considerazione riguarda il caso che vede una semplice disconnessione temporanea di una regione (del suo superpeer) dalla overlay network. In tal caso si potrebbe prevedere un "bypass" temporaneo della regione che, mentre proseguono le attività locali alla regione stessa, dovrebbe essere aggiunta nuovamente al sistema appena possibile.

Se, poi, si volesse prevedere la possibilità di nominare un nuovo superpeer per la regione, si dovrebbero affrontare diversi aspetti.

Innanzitutto sarebbe utile aver precedentemente provveduto a replicare il main container ed il database del superpeer su almeno un altro peer della regione; altrimenti sarebbe un po' più complicato per i peer raggiungere un

[15] Tutto questo a patto che il nodo attualmente non connesso al sistema sia ancora operativo e raggiungibile.

[16] Infatti si presume che un superpeer debba conoscere l'indirizzo ip pubblico di almeno uno dei superpeer che compongono l'overlay network e, nel caso specifico, si suppone che tale superpeer sia il server.

accordo. La replicazione del main container è una funzionalità che viene offerta da JADE tramite il servizio `jade.core.replication.MainReplicationService` - cfr. (Bellifemine, et al., 2004 p. 173-179) - che provvede a mantere costantemente aggiornata ciascuna copia di backup del main container (master). In questo modo, usando la replicazione del main container, la topologia di una regione verrebbe modificata come mostrato nella *Figura 4-4*.

Poi si dovrebbero considerare alcuni aspetti, fra cui:

- elaborare ed implementare, se necessario, un algoritmo di elezione per la scelta del nuovo superpeer;
- strutturare tutti i peer affinchè siano in grado di aggiornare dinamicamente l'identità del superpeer della propria regione;
- risolvere i problemi associati alla presenza del NAT[17].

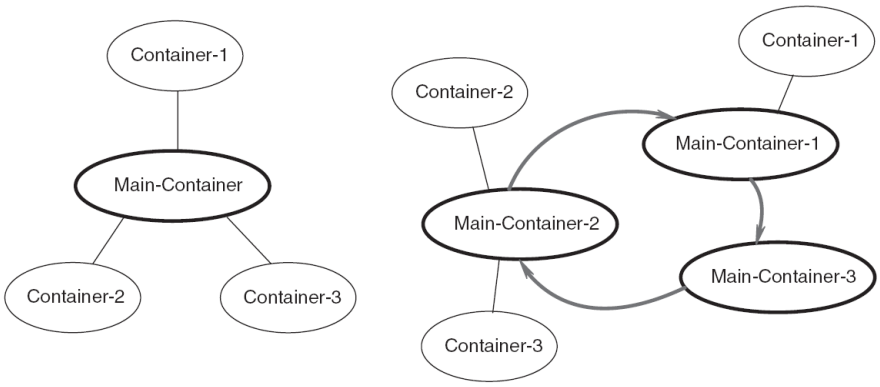


Figura 4-4 – MainReplicationService: topologia di una piattaforma JADE senza (sinistra) e con (destra) la replicazione del main container.

4.5.2 Backward recovery

Il ripristino dello stato di un sistema rappresenta un argomento davvero molto complesso che non può certo essere affrontato se non con una trattazione molto accurata. Pertanto in questo breve paragrafo si

[17] Il NAT, infatti, considera costante nel tempo l'identità IP locale del superpeer della regione ed è proprio verso tale IP che inoltra (port forwarding) le comunicazioni in ingresso dirette a questo particolare peer. Nel caso in cui ne venga eletto un altro, sarebbe necessario che il port forwarding venga ridefinito o che sia stata prevista la presenza di un sottoinsieme di peer della regione che possono candidarsi a superpeer.

riportano solo alcune idee che potranno o meno convergere in una futura realizzazione.

Affinchè un sistema sia tollerante ai guasti (e ciò non significa che sia possibile garantire una completa trasparenza ai guasti) è necessario che i componenti che verificano una situazione di fallimento possano tornare in un precedente stato stabile (backward recovery) o portarsi in un nuovo stato da cui riprendere l'esecuzione (forward recovery)[18].

Nel caso del backward recovery è necessario, quindi, memorizzare lo stato dei componenti tramite checkpoint e log in modo da poterne tentare il ripristino[19] (non è sempre possibile ripristinare uno stato precedente).

Dal momento che di solito i componenti di un sistema sono molto interconnessi, è necessario individuare e memorizzare uno stato consistente a livello globale che tenga conto delle comunicazioni interocorse fra i componenti.

In (Araragi, 2005), ad esempio, si fa riferimento all'algoritmo di Chandy e Lamport (Chandy, February 1985) e si propone un approccio leggermente diverso asserendo che, nonostante questo algoritmo non interferisca col funzionamento del sistema durante il salvataggio dello stato globale, tuttavia non sia applicabile a sistemi dinamici (in cui gli agenti vengono creati e terminati dinamicamente) né a sistemi troppo estesi (dal momento che il salvataggio coinvolge tutti gli agenti del sistema).

Il metodo proposto suggerisce la definizione di "*partial snapshot*" (salvataggio dello stato di un sottoinsieme degli agenti del sistema) e di "*comunication dependecy set*" (cDS) di un agente (l'insieme degli agenti con cui ha comunicato dopo l'ultimo salvataggio dello stato). Quindi si spiega che, dal momento in cui viene avviato il processo che si occupa periodicamente di salvare lo stato del sistema, vengono salvati dei *partial snapshot.* Nel momento in cui un agente subisce un guasto, vengono

[18] Un esempio, cfr. (Tanenbaum p. 356-357), di backward recovery è la richiesta di ritrasmettere un pacchetto perso durante una comunicazione; un esempio di forward recovery è l'utilizzo di un metodo (erasure correction) con cui si effettua la ricostruzione di un pacchetto perso a partire da un insieme di pacchetti ricevuti correttamente (a patto di utilizzare una codifica opportuna come, per esempio, quella di Reed-Solomon).

[19] Si ricorda che nel CAPITOLO 3 sono stati introdotti degli elementi utili ad un miglioramento del sistema in questa direzione: ad esempio gli stati pFAILURE e rFAILURE, le primitive downP e downR (a livello di peer e regioni), le primitive resetP e resetR (a livello di container e main container) e l'add-on Persistence (primitive SaveAgent e LoadAgent a livello dei singoli agenti).

ripristinati individualmente solo gli stati di quegli agenti che sono direttamente o indirettamente correlati (a livello di comunicazioni) all'agente fallito (*Figura 4-6*).

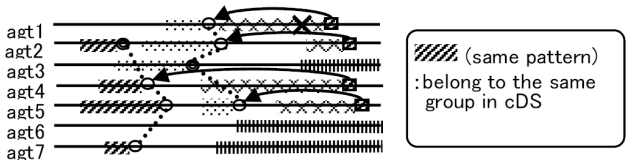


Figura 4-5 – Backward recovery per tutti gli agenti.

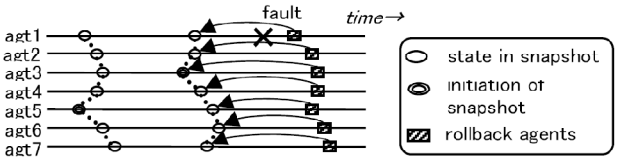


Figura 4-6 – Backward recovery di un cDS.

Per quanto riguarda il sistema che si sta progettando, si noti comunque che, a prescindere da quale strategia di recovery si scelga, non rappresenta certo una situazione poco problematica voler introdurre allo stato attuale di realizzazione un meccanismo del genere: infatti la situazione ideale avrebbe voluto che tale meccanismo venisse previsto fin dalle fasi primordiali del progetto.

CAPITOLO 5

Scelte implementative

In questo capitolo viene presentato il nuovo sistema prima da una prospettiva abbastanza ampia e poi molto più dettagliatamente: a partire dai componenti che sono stati realizzati, fino alle interazioni che avvengono fra di essi; dalle soluzioni adottate per l'implementazione delle primitive di amministrazione, alle strutture che ne gestiscono l'esecuzione; dai protocolli di comunicazione per il trasferimento extra-regione delle tranche, ad alcune considerazioni sul collaudo e sugli sviluppi futuri del nuovo sistema

5.1 Panoramica sul sistema

La trattazione prosegue sulla scia delle considerazioni riportate nei capitoli precedenti, fornendo una prima panoramica ad alto livello sul sistema implementato, sul deployment dei componenti e sulle linee guida seguite durante l'implementazione.

La struttura di base del sistema e le funzioni svolte dagli agenti sono rimaste pressochè invariate rispetto a quelle già esistenti (descritte nel *CAPITOLO 1*): le uniche sostanziali differenze consistono nelle modifiche apportate[1]:

[1] Si citano, a titolo esemplificativo, l'aggiornamento dell'entità Request per supportare le comunicazioni attraverso l'agente RPA e la modifica del comportamento degli agenti DA e UA per supportare la creazione di una connessione TCP diretta attraverso la tecnica del TCP Hole Punching.

- alla struttura di ogni behaviour affinchè fosse in grado di rilevare ed eseguire le primitive di amministrazione;
- alle strutture dati ed alle procedure di elaborazione delle stesse, nell'ottica di una interazione inter-regione attraverso Internet.

Sulla base delle considerazioni riportate all'interno del *CAPITOLO 2* e del *CAPITOLO* 3, sono stati introdotti ed integrati col sistema esistente i seguenti componenti:

- il thread MainBoot;
- l'agente PMA;
- l'agente RPA;
- l'agente HA;
- i behaviour "coordinatori"
- una Graphic User Interface;
- l'add-on Persistence.

In *Figura 5-1* (p.127) si riporta uno schema che rappresenta il deployment dei nuovi componenti (in arancione) sulle varie tipologie di nodi del sistema.

Il thread MainBoot, i servizi offerti dall'add-on Persistence, l'interfaccia grafica e gli agenti PMA e HA vengono avviati su ogni nodo e svolgono funzioni diverse a seconda di dove vengono eseguiti (peer, superpeer o server). L'agente RPA, invece, viene avviato soltanto sul superpeer.

Ad ogni agente esistente, ad eccezione di RMA, MA e SearchAgent, è stato aggiunto un behaviour "coordinatore" (presentato nel paragrafo *3.6.3* e descritto accuratamente nel *5.2.1.7*) in grado di comunicare con l'agente PMA; la struttura di ogni behaviour è stata modificata in modo da poter interagire col behaviour "coordinatore".

Su ogni peer del sistema, affinchè possa essere controllato attraverso le primitive di amministrazione, è in esecuzione il container locale che è connesso al main container della regione presente sul superpeer.

Ogni regione, per poter essere aggiunta all'overlayNetwork, contatta preventivamente il server (overlayNetworkUpdate) in modo da comunicargli la propria presenza rendendo noto l'endpoint del proprio MainBoot.

Il database utilizzato dall'add-on Persistence viene creato sul main container; tramite un servizio offerto dall'add-on stesso, qualsiasi peer è in

grado, all'occorrenza, di connettersi a tale database per eseguire delle query.

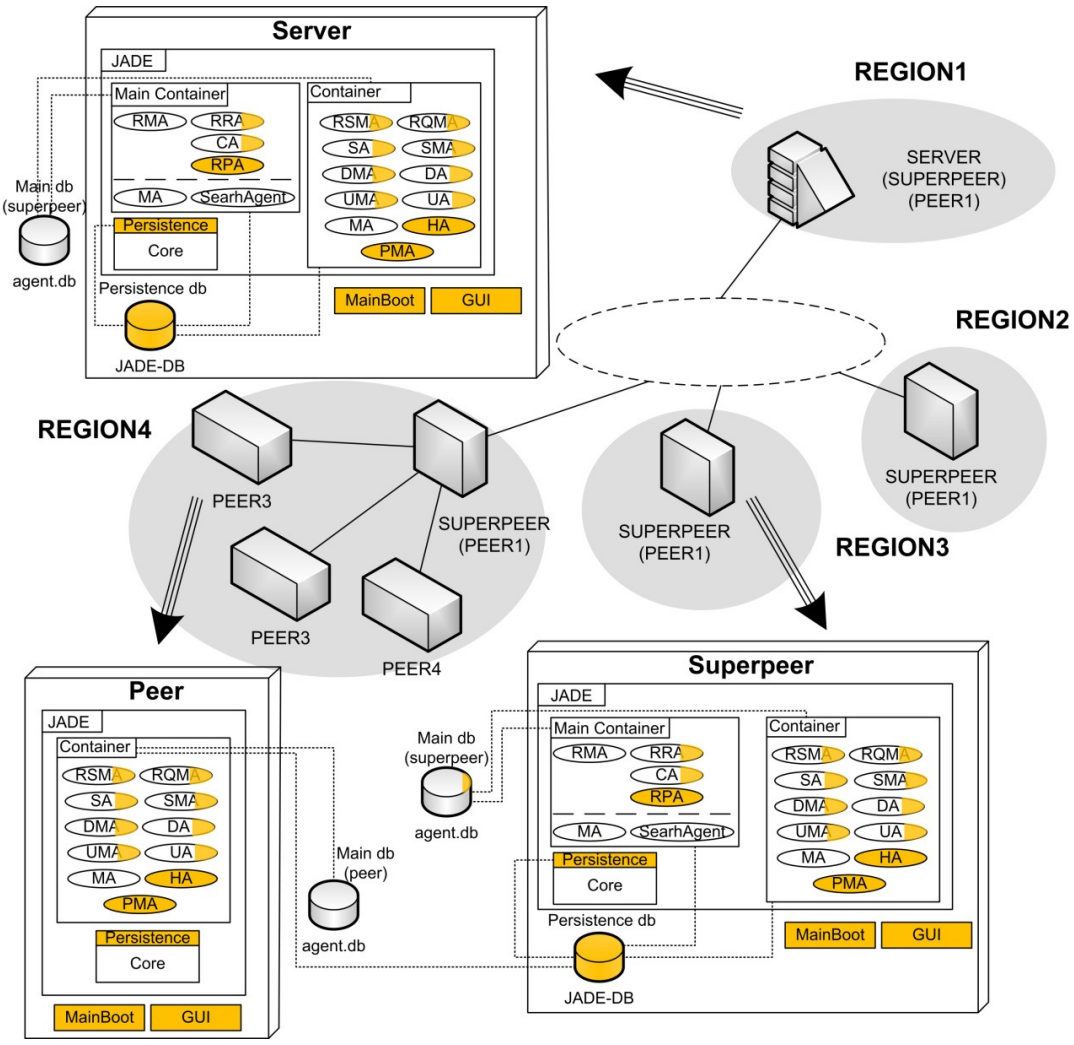


Figura 5-1 – Diagramma di Deployment.

5.2 Componenti

In questo paragrafo, dopo alcune considerazioni preliminari, viene descritto il funzionamento dei nuovi componenti del sistema e vengono documentate le principali modifiche apportate agli agenti, ai protocolli di comunicazione ed alle strutture dati esistenti.

Verranno approfonditi il comportamento dei vari moduli e l'interazione fra di essi. L'analisi del codice verrà riportata solo in caso di necessità.

5.2.1 Considerazioni preliminari

Prima di procedere all'analisi dei componenti si ritiene opportuno presentare le nuove strutture dati e riassumere quelle modifiche di base che hanno permesso la realizzazione del nuovo prototipo.

5.2.1.1 Tabella tranche-owner

La tabella `tranche_owner`, presente solo nel database principale del superpeer, mantiene l'associazione fra ogni peer della regione e le tranche da esso possedute. E' stato aggiunto un campo che indica la validità di ogni entry: nel caso di ibernazione di un peer, le informazioni sulle tranche da esso possedute non vengono rimosse; tali informazioni vengono considerate "non valide" ai fini delle ricerche (intra-regione o extra-regione) per poter essere ripristinate in seguito allo scongelamento del peer.

Alla modifica della tabella sono conseguite, ovviamente, modifiche all'entità Java *TrancheOwner* e a tutti i metodi che operano su tale entità.

5.2.1.2 StateVector

Struttura dati utilizzata dagli agenti per mantenere informazioni sullo stato dei behaviour locali, lo *StateVector* viene usato anche dal PMA per gestire lo stato degli agenti di un peer e dei peer di una regione.

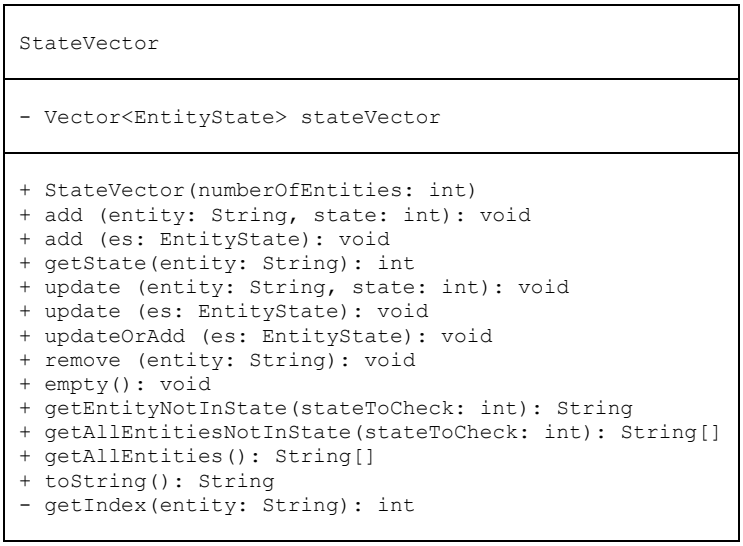


Figura 5-2 – Oggetto StateVector.

Uno *StateVector* incorpora un oggetto `Vector<EntityState>` (classe `java.util.Vector`) e offre metodi per la gestione degli oggetti *EntityState* in esso presenti (inserimento, ricerca, ricerca condizionata sullo stato, etc..)

5.2.1.3 EntityState

Entità costituita semplicemente da un campo stringa e da un intero. Viene utilizzata:

- da *StateVector* per manterere le informazioni relative, ad esempio, allo stato di un behaviour o di un agente;
- all'interno del campo answer di oggetti *Primitve* per comunicare aggiornamenti di stato di behaviour, agenti o peer.

5.2.1.4 Entità OverlayNode

Un *OverlayNode* rappresenta una regione del sistema e racchiude tutte le informazioni necessarie per la sua corretta gestione. Questa entità (*Figura 5-3*, p.130) viene utilizzata dall'agente PMA dei superpeer e del server e costituisce la codifica del campo answer dell'oggetto *Primitive* quando si tratta di una prinitiva di tipo regione (addR, removeR, overlayNetworkUpdate, etc..).

5.2.1.5 Entità Request

L'entità *Request* viene utilizzata per identificare una richiesta formulata da un peer e per comunicare alla fonte informazioni sul richiedente. Questa entità è stata modificata aggiungendo gli attributi privati:

- `String idRPAAgentToContact;`
- `String httpMtpRPAAgentToContact;`
- `String rpaIP;`
- `int rpaPort.`

I primi due permettono quelle comunicazioni (con messaggi ACL) che vedono i due agenti intelocutori posti su peer di regioni distinte, che non sono quindi in grado di inviarsi direttamente messaggi ACL, ma che devono richiedere l'intervento dell'agente RPA.

Gli altri due servono durante la procedura di TCP Hole Punching per le comunicazioni fra l'agente HA e il servizio di rendezvous offerto dall'agente RPA.

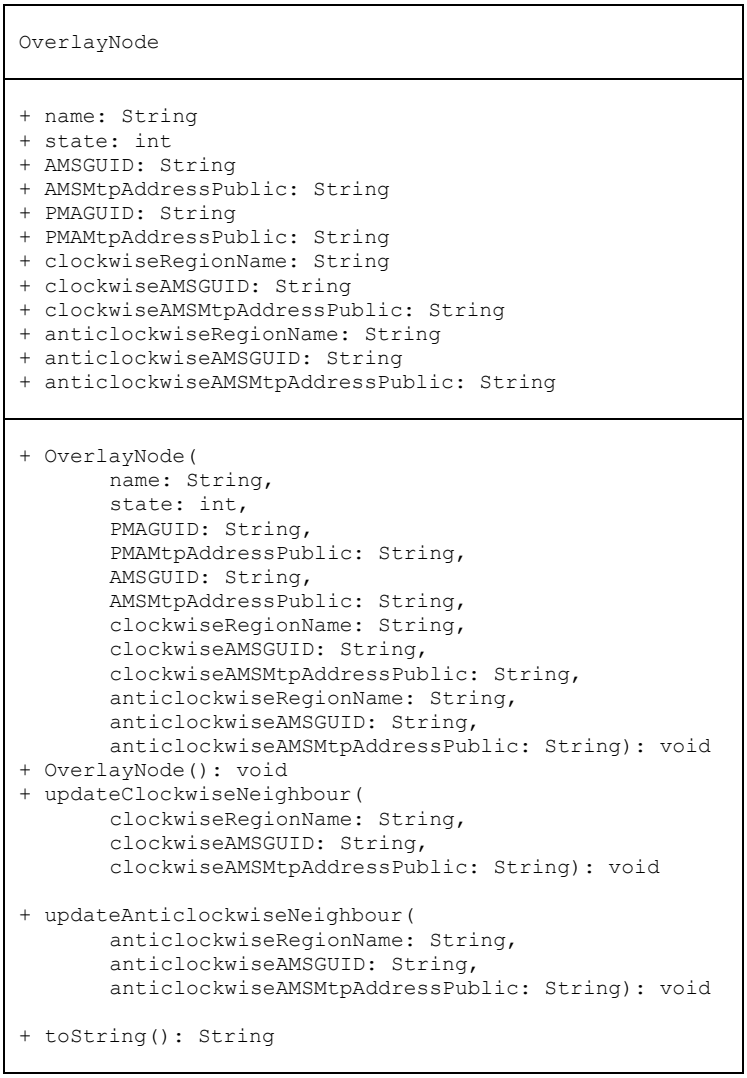


Figura 5-3 – Oggetto OverlayNode.

5.2.1.6 Entità Primitive

L'oggetto *Primitive* rappresenta una primitiva di amministrazione e viene utilizzato durante l'interazione fra gli agenti PMA dei vari nodi.

Contiene diversi attributi per mezzo dei quali vengono identificati tutti i dettagli della primitiva, del creatore e del target. Il campo answer, invece, contiene l'esito dell'esecuzione della primitiva che può consistere in:

- un oggetto *EntityState*;
- un oggetto *OverlayNode*;
- una stringa human-readable che identifica un errore.

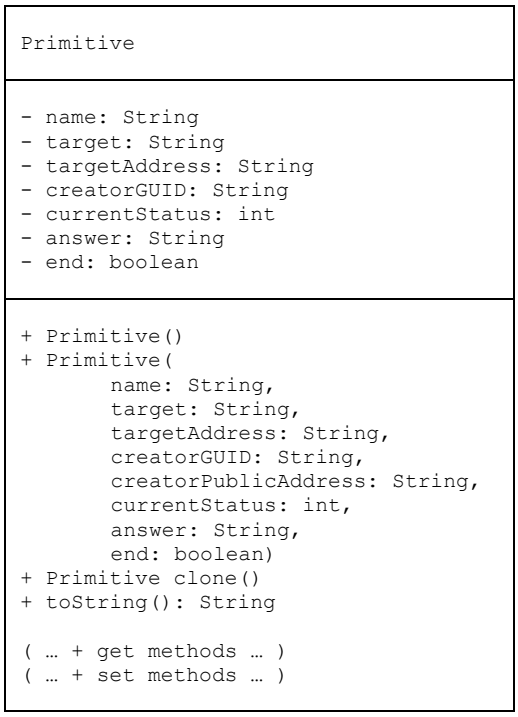


Figura 5-4 – Oggetto Primitive.

Il campo target può contenere, ad esempio, REGION3 se la primitiva è indirizzata ad una regione o PEER2@REGION3 se è indirizzata ad un peer; l'attributo targetAddress, invece, può contenere il `CONTAINER_MTP_ADDRESS` locale[2] (es. http://192.168.0.4:7778/acc) nel caso il target sia un peer, oppure può contenere il `CONTAINER_MTP_ADDRESS_PUBLIC` (es. http://151.94.30.30:6668/acc) nel caso il target sia il superpeer di una regione.

[2] L'informazione è indispensabile soprattutto per le attività previste da thawP che richiedono l'indirizzo MTP preciso su cui dovrà avvenire il caricamento degli stati salvati nel database.

Il campo creatorGUID contiene l'identificatore globale del nodo che ha creato lanciato la primitiva, mentre il campo end indica se il contenuto dell'oggetto Primitive si riferisce all'esito finale della primitiva specificata dal campo name (end = true) oppure se si riferisce ad un aggiornamento (ad esempio relativo allo stato di un peer durante la primitiva removeR).

5.2.1.7 Behaviour ####ReceiveAdminPrimitiveBhv

Questo behaviour rappresenta l'implementazione del behaviour "coordinatore" descritto nel paragrafo *3.6.3*.

Dal momento che l'analisi delle funzionalità è stata già effettuata nel suddetto paragrafo, si reputa opportuno riportare direttamente un esempio concreto che è possibile declinare, con le opportune considerazioni, per tutti gli agenti che ne fanno uso: il behaviour dell'agente RSMA che è denominato RSMAReceiveAdminPrimitiveBhv.

La struttura del behaviour è costituita dalla macchina a stati finiti rappresentata in *Figura 5-5*.

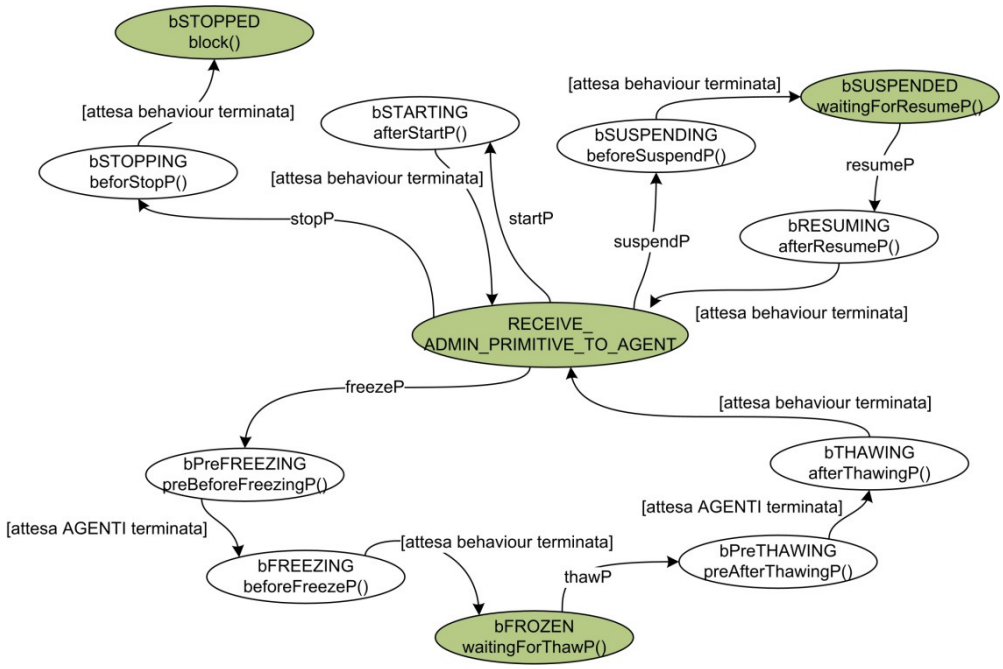


Figura 5-5 – Behaviour "coordinatore": FSM [esempio].

Questo behaviour esegue le seguenti operazioni:

1. riceve i messaggi ACL relativi all'esecuzione di una primitiva inviati dall'agente PMA all'agente RSMA (stati in verde);

2. imposta la variabile targetPeerState dell'agente RSMA (classe RSMA.java) in base allo stato obiettivo richiesto (tutti gli altri behaviour dell'agente RSMA controllano[3] periodicamente questa variabile e agiscono di conseguenza);
3. avvia il behaviour RSMACheckStateTimeoutBhv (paragrafo *5.2.1.8*) che controlla la scadenza del timeout previsto;
4. attende la conferma da parte di tutti i behaviour dell'agente o il messaggio che indica la scadenza del timeout (stati di transizione con desinenza in -ING).
   Il behaviour si comporta in modo differente a seconda del tipo di messaggio ricevuto:
   a) BEHAVIOUR_STATE_UPDATE [ACLMessage.INFORM]:
      informa il PMAReceiveAdminPrimitiveAsPeerBhv dell'aggiornamento di stato di un behaviour;
   b) BEHAVIOUR_STATE_UPDATE [ACLMessage.FAILURE]:
      inserisce il nome del behaviour che ha comunicato l'errore nella lista che verrà restituita al PMA del peer;
   c) BEHAVIOUR_STATE_UPDATE [ACLMessage.CONFIRM]:
      memorizza la conferma di avvenuta esecuzione della primitiva da parte del behaviour mittente del messaggio; poi effettua un controllo: se tutti i behaviour hanno già inviato un CONFIRM, notifica l'esito positivo della primitiva al PMA del peer, altrimenti ritorna in attesa;
   d) BEHAVIOUR_STATE_UPDATE_TIMEOUT:
      allo scadere del timeout invia in ogni caso una notifica (AGENT_STATE_UPDATE) al PMA del peer: se tutti i behavior hanno inviato un CONFIRM, l'esito della primitiva eseguita dall'agente sarà positivo; in caso contrario, negativo.
5. notifica all'agente PMA del peer l'esito della primitiva ricevuta dall'agente RSMA quando tutti i behaviour hanno risposto;

[3] Si veda il paragrafo *5.2.1.9*.

6. fa uso degli stati bPreFREEZING e bPreTHAWING per rispettare le dipendenze dagli altri agenti riassunte rispettivamente dalla *Figura 3-9* (p.83) e dalla *Figura 3-10* (p.86).

In ogni stato di transizione (-ING) il behaviour si mette in attesa dei messaggi di notifica da parte di tutti gli altri behaviour dell'agente o del messaggio che indica la scadenza del timeout[4]. In ogni stato per così dire "stabile" (es. bSTOPPED, bFROZEN, bSUSPENDED, etc..) il behaviour rimane in attesa del messaggio corrispondente alla successiva primitiva da eseguire (es. in bFROZEN attende thawP, in bSUSPENDED attende resumeP, etc..).

Nel caso scada il timeout `Parameters.MAXIMUM_BEHAVIOUR_STATE_CHECK_PERIOD` controllato da un ####CheckStateTimeoutBhv, il behaviour informa il PMAReceiveAdminPrimitivaAsPeerBhv comunicando quali behaviour non hanno dato conferma prima dello scadere del timeout.

La presenza degli stati bPreFREEZING e bPreTHAWING si è rivelata necessaria dal momento che, durante le operazioni di congelamento e scongelamento dell'agente, l'agente RSMA deve mettersi in attesa di messaggi ACL prima di poter proseguire: nel primo caso deve attendere che altri agenti abbiano già proceduto all'ibernazione e nel secondo caso che altri agenti abbiano confermato la ripresa delle attività dopo lo scongelamento (thawP). Infatti l'agente RSMA deve rispettare i seguenti vincoli[5]:

Primitiva ricevuta	Stato attuale	targetPeerState	Agenti da attendere	Agenti da sbloccare
freezeP	pACTIVE	pFROZEN	DA, DMA	RQMA
thawP	pFROZEN	pACTIVE	RQMA	DA, DMA

Tabella 5-1 – RSMA: vincoli esecuzione di freezeP e thawP [esempio].

[4] Si ricorda che è necessario che l'attesa avvenga in uno stato creato appositamente in quanto altrimenti, quando giunge un nuovo messaggio (dopo l'invocazione del metodo `block()` e la nuova schedulazione del behaviour), il behaviour riprenderebbe ad essere eseguito dall'inizio.

[5] Per gli altri agenti si applica un ragionamento analogo, alla luce delle considerazioni riassunte dalla *Figura 3-9* e dalla *Figura 3-10* (p.83 e p.86).

5.2.1.8 Behaviour ####CheckStateTimeoutBhv

Questo semplice behaviour oneShot dal comportamento ciclico può essere aggiunto da un qualsiasi agente o behaviour che voglia avvalersi di un processo di timeout.

Ad esempio viene usato dal PMAReceiveAdminPrimitiveAsPeerBhv per controllare se tutti gli agenti di un peer confermano entro un certo limite di tempo la propria disponiblità al cambiamento di stato richiesto da una primitiva[6].

Il behaviour deve essere eseguito su un thread indipendente, in modo da poter controllare la scadenza del periodo di tempo previsto senza sottostare allo scheduler di JADE; alla scadenza di tale periodo, il behaviour inserisce nella coda dei messaggi dell'agente da cui è stato creato un messaggio definito in fase di inizializzazione, che notifica la scadenza del timeout.

Durante la creazione del behaviour si fa ricorso ad un oggetto della classe `jade.core.beheaviours.ThreadedBehaviourFactory` che è in grado proprio di gestire i threaded behaviour.

5.2.1.9 Nuova architettura dei behaviour

Per poter dialogare con il behaviour coordinatore ####ReceiveAdminPrimitiveBhv di ogni agente e per poter eseguire le primitive richieste, tutti i behaviour dell'agente controllano il contenuto della variabile `targetPeerState` dell'agente (memorizzata nella classe che estende la classe `jade.core.Agent`).

Il controllo periodico di questa variabile è regolamentato in funzione delle operazioni eseguite da ciascun behaviour: ogni behaviour svolge compiti diversi che richiedono tempi di esecuzione differenti. La nuova struttura in cui viene inserito ogni behaviour assicura che sia possibile controllare precisamente quando permettere al behaviour di controllare la variabile `targetPeerState`, evitando così l'interruzione delle operazioni.

[6] Questo tipo di behaviour viene utilizzato anche dal ####ReceiveAdminPrimitiveBhv di ogni agente per verificare la scadenza periodo pervisto per il cambiamento di stato dei behaviour dell'agente stesso; viene usato anche dal PMAReceiveAdminPrimitiveAsSuperpeer per eseguire l'analogo controllo sui peer di una regione.

La soluzione adottata mira ad essere utilizzabile con i behaviour esistenti, sia nel caso in cui siano già organizzati in base ad una macchina a stati finiti, sia nel caso in cui siano semplici behaviour ciclici o oneShot; la soluzione implementata, quindi, prevede la presenza di una semplice macchina a stati finiti:

- nello stato di default viene controllata le variabile `targetPeerState` e, nel caso ne venga rilevata la modifica (rispetto allo stato attuale del behaviour) vengono eseguite le operazioni richieste; se non viene rilevata alcuna modifica, si procede come di consueto con le operazioni tipiche del behaviour;
- negli altri stati (zero o più) il behaviour continua a svolgere le sue attività secondo le specifiche necessità.

Per descrivere dettagliatamente questa nuova struttura si prenda come riferimento l'agente DA.

Questo agente ha un behaviour ciclico che si occupa di configurare la ricezione delle tranche dagli agenti UA delle fonti (DAReceivedTrancheFromUABhv - *Figura 5-6*). Per ottenere questo scopo instaura comunicazioni con altri agenti ed è fondamentale che l'esecuzione di una primitiva, ad esempio, di interruzione (come stopP) non pregiudichi tali comunicazioni.

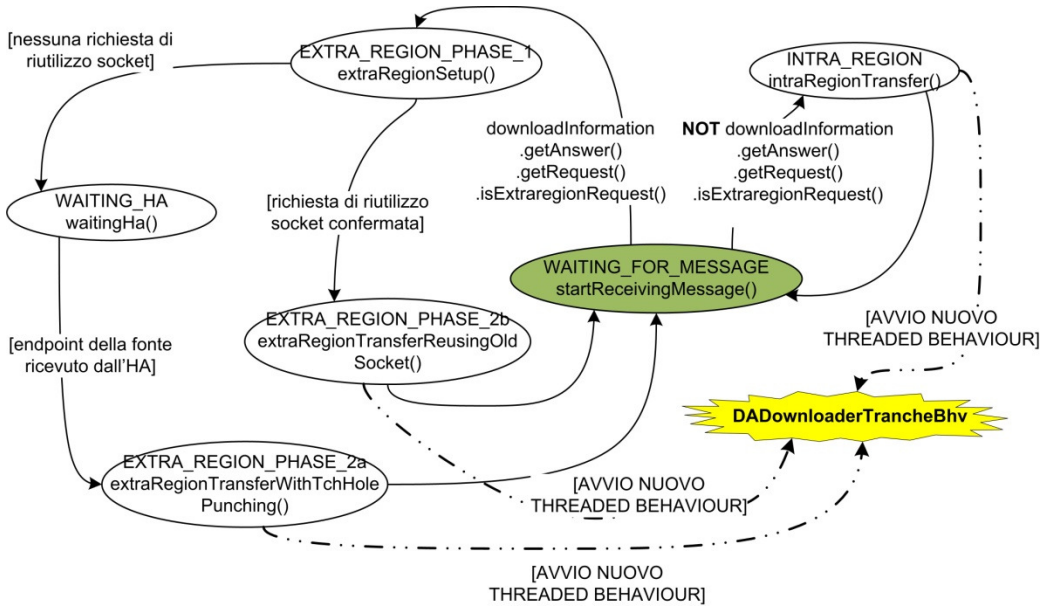


Figura 5-6 – Behaviour DAReceivedTrancheFromUABhv: FSM [esempio].

Il metodo `action()` si comporta come una macchina a stati finiti:

```
...
public void action()
{
        switch (step)
        {
            case WAITING_FOR_MESSAGE:
            {
                startReceivingMessage();
                break;
            }

            case INTRA_REGION:
            {
                intraRegionTransfer();
                break;
            }

            case EXTRA_REGION_PHASE_1:
            {
                extraRegionSetup();
                break;
            }

            case WAITING_HA:
            {
                waitingHa();
                break;
            }

            case EXTRA_REGION_PHASE_2a:
            {
                extraRegionTransferWithTcpHolePunching();
                break;
            }

            case EXTRA_REGION_PHASE_2b:
            {
                extraRegionTransferReusingOldSocket();
                break;
            }


         }

}
...
```

Figura 5-7 - action() del DAReceivedTrancheFromUABhv [esempio].

Lo stato di default (`WAITING_FOR_MESSAGE`) consiste nel metodo `startReceivingMessage()` ed è quello che rileva i cambiamenti di stato della variabile `targetPeerState` (tramite il metodo `targetPeerState()`):

```
private void startReceivingMessage()
{

        int targetPeerState = targetPeerState();
```

```
        if (myState != targetPeerState)
        {
            if(targetPeerState == TypePeerStatus.pFROZEN)
            {
                //OPERAZIONI SPECIFICHE RELATIVE ALLA PRIMITIVA
                // freezeP
                ...
                ... myState = targetPeerState;
            }
            else if(targetPeerState == TypePeerStatus.pSTOPPED
                    || targetPeerState == TypePeerStatus.pDOWNING)
            {
                //OPERAZIONI SPECIFICHE RELATIVE ALLE PRIMITIVE
                // stopP o downP
                ...
                ... myState = targetPeerState;
            }
            else
            {
                //ricezione primitiva che non richiede operazioni
                // specifiche
                myState = targetPeerState;

                //notifica al behaviour "coordinatore"
                // DAReceiveAdminPrimitive
                updateMyState(targetPeerState);
            }
        }


        if ( targetPeerState == TypePeerStatus.pACTIVE)
        {
           /*
              SVOLGIMENTO DELLE NORMALI OPERAZIONI che possono o
              meno richiedere la presenza di altri stati oltre
              a quello di default
           */
           ...
        }
        else
        {
            block(Parameters.MAXIMUM_BLOCKING_PERIOD);
        }
}
```

Figura 5-8 - FSM del DAReceivedTrancheFromUABhv: stato di default.

Si precisa che la nuova struttura dei behaviour prevede anche la presenza di un metodo `afterThawP()` che deve essere invocato dalla classe principale dell'agente prima di portare a termine la primitiva thawP: solo in questo modo è possibile re-inizializzare le variabili non serializzabili (`transient`)[7] di ogni singolo behaviour che non sono state salvate nel database.

[7] Ad esempio gli oggetti ServerSocket oppure le ThreadedBehaviourFactory necessarie all'avvio di ThreadedBehaviour come i ####CheckStateTimeout.

5.2.1.10 Mapping classes

Le classi seguenti vengono usate all'interno del sistema per mappare stringhe su costanti. Alcune di esse sono state create appositamente, altre sono state modificate per consentire l'utilizzo di nuovi protocolli.

Ad esempio la classe `TypeConversationId` gestisce i ConversationId utilizzati dagli agenti duante lo scambio di messaggi ACL per differenziare più conversazioni che avvengono contemporaneamente; sono stati aggiunti molti valori per identificare le conversazioni fra i nuovi agenti del sistema.

La classe `TypeRegionStatus`, invece, associa lo stato di una regione ad un intero, in modo che sia possibile, ad esempio, utilizzare un costrutto switch. In modo analogo opera la classe `TypePeerStatus`.

TypePeerStatus	
Stato	Valore
pACTIVE	0
pSUSPENDING	1
pSUSPENDED	2
pRESUMING	3
pFREEZING	4
pFROZEN	5
pTHAWING	6
pSTOPPING	7
pSTOPPED	8
pSTARTING	9
pFAILURE	10
pDOWNING	11

Tabella 5-2 – TypePeerStatus

TypeRegionStatus	
Stato	Valore
rADDED	0
rREMOVING	1
rREMOVED	2
rADDING	3
rFAILURE	4
rDOWNING	5

Tabella 5-3 - TypeRegionStatus

La classe `TypeAdminPrimitiveError` mantiene l'associazione tra codici di errore e stringhe di testo che descrivono l'errore stesso.

5.2.1.11 Interfaccia grafica (javax.swing.JFrame)

Il sistema è stato dotato di una semplice interfaccia grafica, creata estendendo la classe `javax.swing.JFrame`; tale componente permette l'interazione fra l'utente ed il sistema e verrà descritto più accuratamente nel paragrafo *5.2.7*.

5.2.2 Thread Mainboot [new]

Questo thread viene eseguito al di fuori dell'ambiente di esecuzione offerto da JADE e costituisce un'estensione del componente analogo (Main.java) utilizzato dal precedente prototipo.

Per maggiore chiarezza si osservi la seguente figura in cui viene rappresentato lo schema di funzionamento del componente.

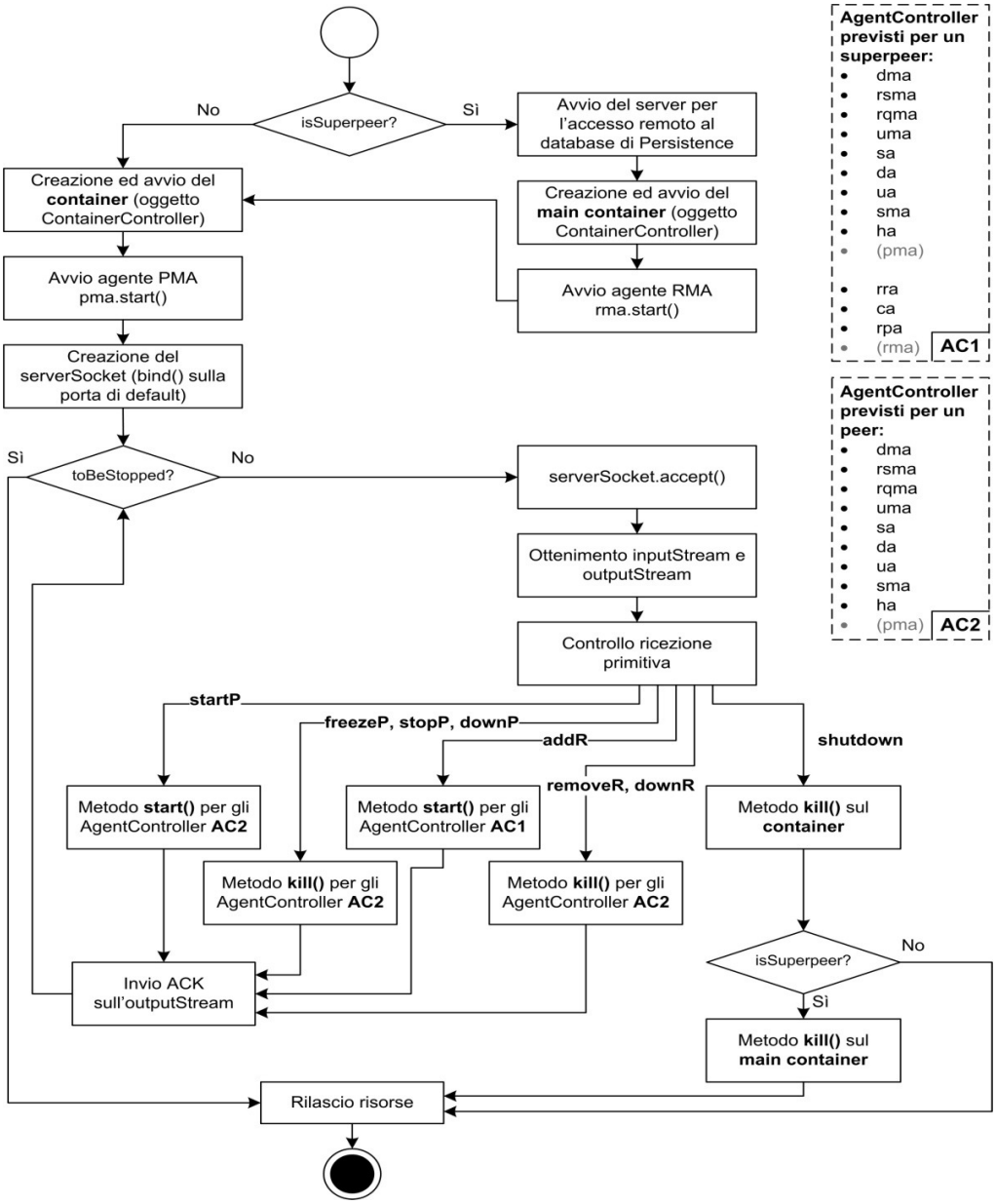


Figura 5-9 – Thread MainBoot.

Nelle prime fasi svolge le stesse operazioni previste per il vecchio componente; in più configura i servizi dell'add-on Persistence ed è in grado di portare a termine le ultime operazioni richieste dall'agente PMA per il completamento delle primitive di stato per peer e superpeer.

5.2.3 Agente PMA (PeerManagementAgent) [new]

L'insieme degli PMA distribuiti all'interno dell'intero sistema costituiscono il punto nodale di tutta l'attività di gestione delle primitive di amministrazione, compresa l'interazione con l'interfaccia grafica.

5.2.3.1 Classificazione dei behaviour

Come accennato nel *CAPITOLO 3*, le funzioni di un agente PMA variano a seconda del nodo su cui è in esecuzione (peer, superpeer o server); ogni behaviour è responsabile di una certa categoria di funzioni e l'aggiunta dei behaviour da parte dell'agente - precisamente all'interno del metodo `setup()` - avviene secondo le seguenti regole:

Tipologia nodo	Behaviour aggiunti
Peer	PMAReceiveAdminPrimitiveAsPeerBhv
	PMASimpleUserInterfaceBhv
Superpeer	PMAReceiveAdminPrimitiveAsSuperpeerBhv
	PMAForwardAdminPrimitiveToPeerBhv
	PMAReceiveAdminPrimitiveAsPeerBhv
	PMASimpleUserInterfaceBhv
Server	PMAReceiveAdminPrimitiveAsServerBhv
	PMAForwardAdminPrimitiveToSuperpeerBhv
	PMAReceiveAdminPrimitiveAsSuperpeerBhv
	PMAForwardAdminPrimitiveToPeerBhv
	PMAReceiveAdminPrimitiveAsPeerBhv
	PMASimpleUserInterfaceBhv

Tabella 5-4 – Classificazione dei behaviour del PMA.

Si precisa che, al fine di garantire la possibilità di eseguire più primitive contemporaneamente, tutti i behaviour dell'agente PMA vengono eseguiti in modo ciclico[8] ad eccezione di:

- PMAForwardAdminPrimitiveToSuperpeerBhv;
- PMAForwardAdminPrimitiveToPeerBhv.

Gli altri behaviour per così dire "stabili" possono all'occorrenza aggiungere una o più volte questi due behaviour[9], affinchè eseguano una specifica operazione su un thread indipendente e lascino il behaviour principale libero di eseguirne altre. Quando terminano la propria esecuzione, i behaviour "temporanei" non vengono più schedulati dallo scheduler di JADE finchè non vengono aggiunti nuovamente.

5.2.3.2 Interazione fra i behaviour

Nella *Figura 5-10* si fornisce un esempio della dislocazione dei singoli behaviour del PMA e delle interazioni che avvengono durante l'esecuzione della primitiva "removeR REGION2", dove REGION2 identifica la regione target che contiene due peer oltre al superpeer.

La primitiva viene creata attraverso l'interfaccia grafica [1] dal superpeer della REGION3 e, tramite il PMASimpleUserInterfaceBhv, raggiunge [2] il PMAReceiveAdminPrimitiveAsServerBhv del server (PEER1@REGION1).

Il server crea un PMAForwardAdminPrimitiveToSuperpeerBhv [3] che inoltra [4] la primitiva al PMAReceiveAdminPrimitiveAsSuperpeerBhv della regione target.

A questo punto vengono creati [5] i tre behaviour PMAForwardAdminPrimitiveToPeerBhv: uno per il peer che fa da superpeer (PEER1@REGION2) e due per i restanti peer della regione (PEER2@REGION2 e PEER3@REGION2).

I tre PMAForwardAdminPrimitiveToPeerBhv comunicano [6] con i rispettivi PMAReceiveAdminPrimitiveAsPeerBhv i quali, dopo aver interagito con i behaviour "coordinatori" (####ReceiveAdminPrimitiveBhv) dei singoli agenti, informano [7] i propri PMAForwardAdminPrimitiveToPeerBhv

[8] Estendono `jade.core.behaviours.CyclicBheaviour`.

[9] Estendono `jade.core.behaviours.OneShotBehaviour` e vengono eseguiti in un thread indipendente (ThreadedBehaviour).

presenti nel superpeer della regione che chiederanno al MainBoot dei peer di rimuovere tutti gli agenti dal container (eccetto il PMA).

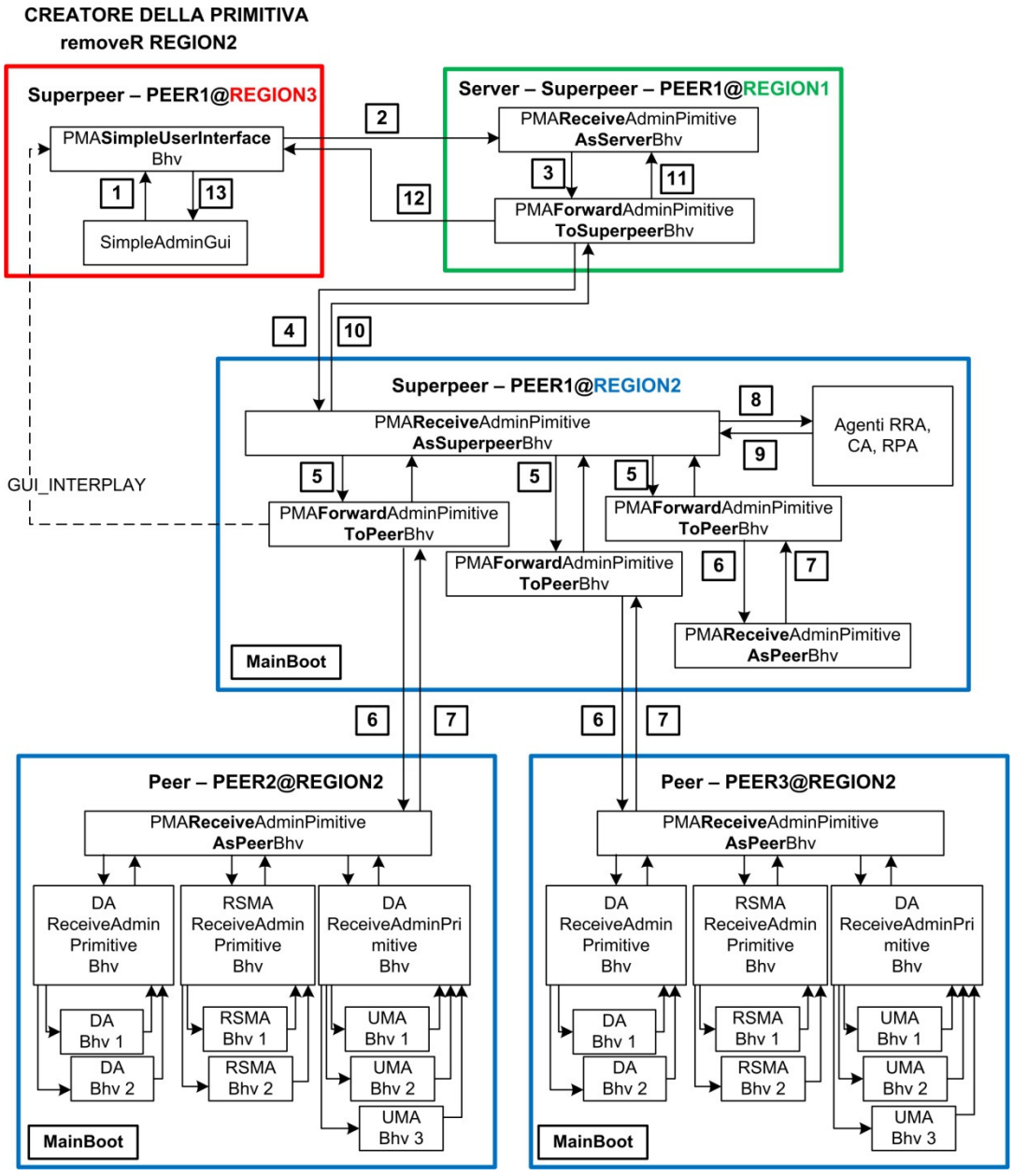


Figura 5-10 –Interazioni dei behaviour del PMA durante removeR.

A questo punto il PMAReceiveAdminPrimitiveAsSuperpeerBhv, se non si sono verificati errori, può interrompere direttamente gli agenti RRA, RPA e CA propri del superpeer [8] e [9] ed avvisare [10] il PMAForwardAdminPrimitiveToSuperpeerBhv presente sul server.

Quest'ultimo behaviour chiederà al MainBoot del superpeer di rimuovere dal container e dal main container tutti gli agenti (eccetto il PMA e l'RMA); inoltre chiederà [11] al PMAReceiveAdminPrimitiveAsServerBhv di aggiornare lo stato della overlayNetwork ed invierà un messaggio di tipo `TypeConversationId.GUI_INTERPLAY` al PMASimpleUserInterfaceBhv del creatore il quale mostrerà [12] sull'interfaccia grafica l'esito della primitiva.

N O T E I M P O R T A N T I: sull'interfaccia grafica del creatore non viene mostrato solo l'esito finale dell'esecuzione della primitiva, ma anche i messaggi di aggiornamento di stato dei peer della regione da rimuovere e gli eventuali messaggi di errore generati ai vari livelli. Nel caso della rimozione di un peer è possibile far visualizzare sulla GUI del creatore anche i messaggi di aggiornamenti di stato dei singoli behaviour del peer.

5.2.3.3 Gestione delle "conversazioni"

Dal momento che i vari behaviour del PMA vengono eseguiti all'interno dello stesso agente e condividono, quindi, la stessa coda dei messaggi, è stato necessario utilizzare un metodo per differenziare le numerose conversazioni. Questo accorgimento si basa sull'utilizzo di ConversationId differenti a seconda dei behaviour coinvolti.

Ad esempio (*Figura 5-11*, p. 145) il PMAForwardAdminPrimitive ToSuperpeerBhv del server invia messaggi di tipo `ADMIN_PRIMITIVE_TO_SUPERPEER` ai PMAReceiveAdminPrimitive AsSuperpeerBhv dei superper, i quali rispondono con messaggi di tipo `REGION_STATE_UPDATE+"#"+REGION_NAME` in modo da differenziare le comunicazioni in entrata verso il server originate da regioni (`REGION_NAME`) diverse.

Allo stesso modo il PMAForwardAdminPrimitiveToPeerBhv del superpeer invia messaggi di tipo `ADMIN_PRIMITIVE_TO_PEER` ai PMAReceiveAdminPrimitiveAsPeerBhv dei peer della regione, i quali rispondono con messaggi di tipo `PEER_STATE_UPDATE+"#"+PEER_NAME` in modo da differenziare le comunicazioni in entrata verso il superpeer originate da peer (`PEER_NAME`) diversi.

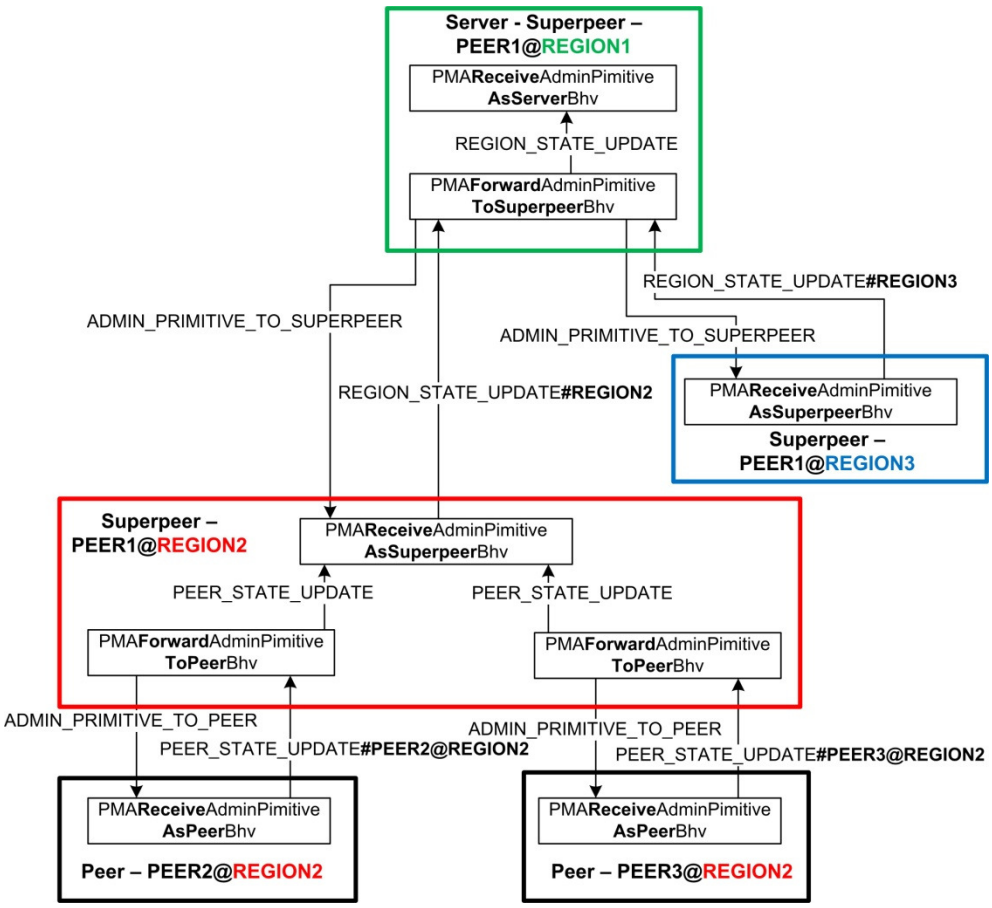


Figura 5-11 - ConversationId usati dai behaviour degli agenti PMA.

5.2.3.4 Gestione della temporizzazione

Tutti i behaviour, compresi quelli dell'agente PMA, possono avvalersi di un ulteriore behaviour chiamato ####CheckStateTimeoutBhv che è strutturato secondo l'analisi svolta nel paragrafo *5.2.1.8* e che si occupa, quindi, di verificare la scadenza del timeout configurato in fase di inizializzazione del behaviour.

Nella *Tabella 5-5* (p.146) si riassumono i timeout utilizzati per la gestione delle primitive di amministrazione. Si consideri il behaviour coordinatore ####ReceiveAdminPrimitiveBhv dell'agente RSMA (rappresentato in *Figura 5-5*, p.132).

Behaviour	Obiettivo da controllare	Timeout
RSMAReceiveAdmin Primitive Bhv	Tutti i behaviour dell'agente RSMA	Parameters.MAXIMUM_**BEHAVIOUR**_STATE_CHECK_PERIOD
PMAReceive Admin Primitive AsPeer Bhv	Tutti gli agenti del peer	Parameters.MAXIMUM_**AGENT**_STATE_CHECK_PERIOD
PMAReceive Admin Primitive AsSuperpeer Bhv	Tutti i peer della regione	Parameters.MAXIMUM_**PEER**_STATE_CHECK_PERIOD
	Gli agenti di tipo superpeer (RRA, RPA, CA)	Parameters.MAXIMUM_**BEHAVIOUR**_STATE_CHECK_PERIOD
PMAForward Admin Primitive ToSuperpeer Bhv	Regione nel suo complesso	Parameters.MAXIMUM_**PEER**_STATE_CHECK_PERIOD

Tabella 5-5 – Timeout utilizzati dei behaviour del PMA.

5.2.3.5 Analisi dei behaviour

Prima di proseguire con la descrizione dei behaviour dell'agente PMA si precisa che la classe principale dell'agente PMA (quella che estende la classe `jade.core.Agent`) ha un riferimento diretto all'interfaccia grafica[10] e mantiene l'elenco di tutti gli agenti di tipo peer (riquadro AC2 della *Figura 5-9*, p.140) e di tipo superpeer (riquadro AC1 della stessa figura) che devono essere controllati in seguito alla ricezione di una primitiva: tutti i behaviour dell'agente PMA sono in grado di accedere alle due variabili di tipo `StateVector` (`peerAgentStateVector` e `superpeerAgentStateVector`).

[10] Oggetto della classe SimpleAdminGUI, paragrafo *5.2.7*

Inoltre si tenga presente che ogni behaviour dell'agente PMA dispone di alcune variabili locali che costituiscono le strutture dati necessarie al corretto svolgimento delle sue attività.

PMAReceiveAdminPrimitiveAsServerBhv

Le principali funzioni svolte da questo behaviour sono le seguenti:

1. gestione della struttura della overlay network (riceve messaggi REGION_STATE_UPDATE);
2. inoltro delle primitive (di tipo peer o di tipo regione) inviate da una regione ad un'altra;
3. esecuzione delle primitive di supporto descritte in *5.3.4.3*.

La funzione 1 si basa sulla presenza di una variabile di tipo `HashMap<String, OverlayNode>`, chiamata `overlayNetwork`, che rappresenta appunto la struttura della overlay network. Il behaviour aggiorna tale struttura dati in seguito alla ricezione di messaggi ACL di tipo `TypeConversationId.REGION_STATE_UPDATE`.

Per svolgere le funzioni 1 e 2 si avvale dell'aiuto del behaviour PMAForwardAdminPrimitiveToSuperpeerBhv che viene aggiunto (tramite una `jade.core.behaviour.ThreadedBehaviourFactory`) come ThreadedBehaviour ed eseguito su un thread indipendente.

PMAForwardAdminPrimitiveToSuperpeerBhv

Questo behaviour di tipo OneShot può essere aggiunto da un PMAReceiveAdminPrimitiveAsServerBhv e si occupa di eseguire sulle regioni del sistema le primitive addR, removeR e downR (si veda anche il paragrafo *5.3.2*) informando dell'esito il creatore della primitiva e il behaviour padre PMAReceiveAdminPrimitiveAsServerBhv che gestisce l'overlayNetwork.

Durante l'esecuzione delle primitive suddette è compito di questo behaviour informare il creatore anche sugli aggiornamenti di stato dei peer della regione target. In caso di errore, verrà inoltrata al creatore non solo la lista dei peer che non hanno completato correttamente l'esecuzione della primitiva, ma anche la lista dei singoli agenti che hanno riscontrato degli errori.

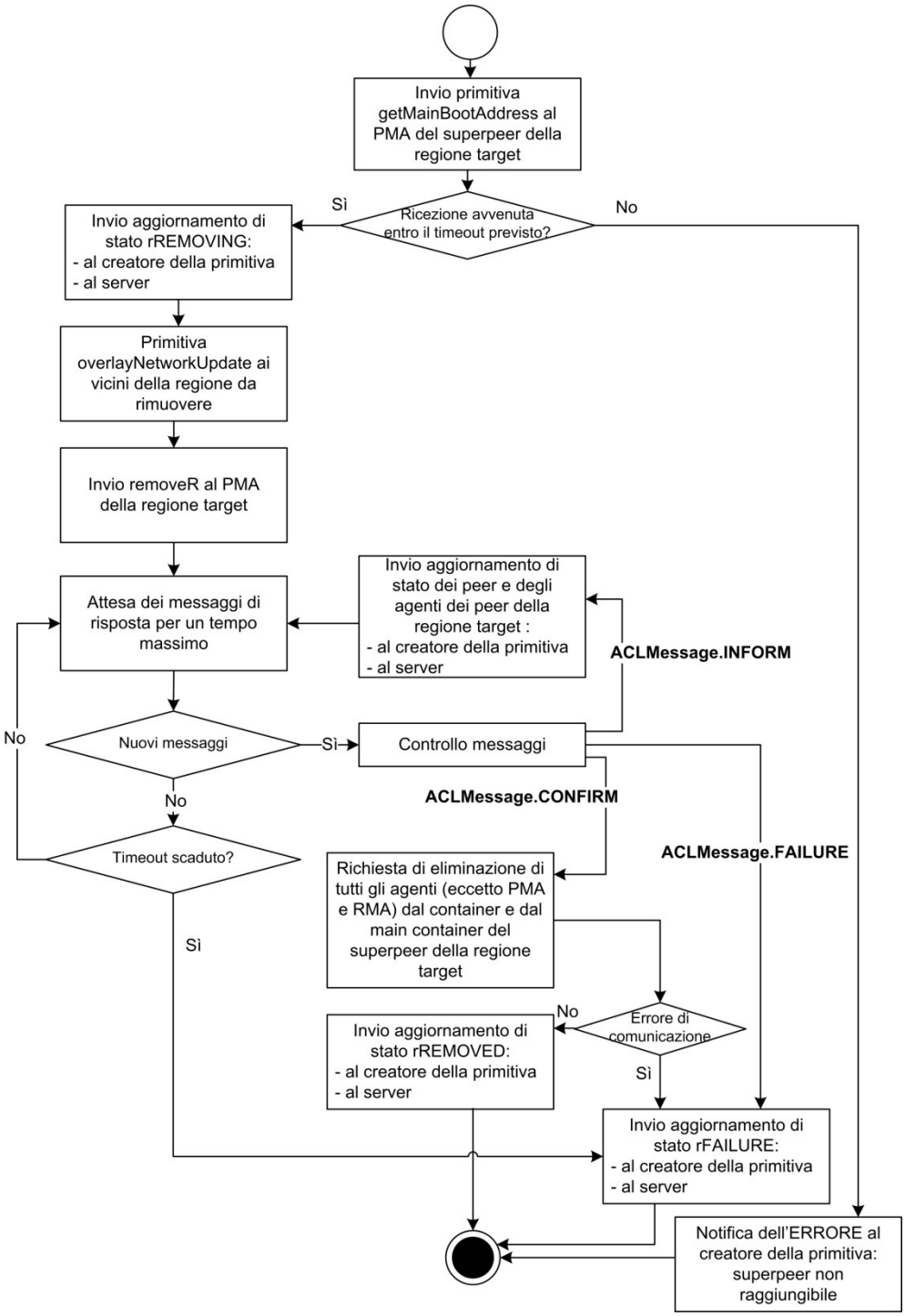


Figura 5-12 – PMAForwardAdminPrimitiveToSuperpeer: removeR.

A titolo esemplificativo si riporta la *Figura 5-12* (p. 148) in cui viene schematizzato il comportamento di questo behaviour quando deve inoltrare la primitiva removeR. Nel caso della primitiva addR la comunicazione con i superpeer vicini della regione da aggiungere avviene DOPO l'aggiunta della regione stessa, quando essa è già operativa: in questo modo, come spiegato in *3.6.11*, viene preservata la struttura delle overlay network e le ricerche extra-regione non vengono interrotte.

Tra i vari accorgimenti adottati in sede di implementazione, si cita la presenza di timeout relativi:

- all'esecuzione della primitiva getMainBootAddress, con cui il behaviour chiede al PMAReceiveAdminPrimitiveAsSuperpeerBhv l'endpoint pubblico del proprio MainBoot;
- al controllo dell'avvenuta confema da parte di tutti i peer della regione (`Parameters.MAXIMUM_PEER_STATE_CHECK_PERIOD`).

NOTA: La richiesta per l'eliminazione di tutti gli agenti (eccetto PMA e RMA) dal container e dal main container del superpeer viene inviata al MainBoot.

Le comunicazioni instaurate con gli altri behaviour e gli altri agenti durante l'esecuzione di addR e removeR su una regione sono rappresentate rispettivamente dai diagrammi delle comunicazioni in *Figura 5-28* (p.184) ed in *Figura 5-31* (p.189).

PMAReceiveAdminPrimitiveAsSuperpeerBhv

Questo behaviour si occupa di gestire:

1. lo stato dei peer della regione tramite una variabile `StateVector` che contiene oggetti `EntityState` associati ai peer (riceve messaggi `PEER_STATE_UPDATE`)
2. l'inoltro delle primitive (di tipo peer) che hanno come target i peer della regione e che sono inviate sia dall'esterno che dall'interno della regione stessa;
3. l'esecuzione delle primitive di supporto descritte in *5.3.4.2*.

La struttura del behaviour ciclico è quella di una macchina a stati finiti costituita dai seguenti stati (cfr. *Figura 5-13*, p.150):

- RECEIVE_ADMIN_PRIMITIVE_TO_SUPERPEER

  In questo stato il behaviour rimane in attesa delle primitive di supporto,delle primitive di stato per l'intera regione e delle

primitive di stato rivolte ai peer della regione. Per inoltrare le primitive ai peer della regione si avvale dei behaviour PMAForwardAdminPrimitiveToPeerBhv.

- rPreREMOVING1

  In questo stato, dopo aver ricevuto la primitiva removeR ed aver inviato un messaggio di pre-stop all'agente RRA[11], il behaviour attende le conferme da parte dei peer della regione che vengono inviate dal PMAReceiveAdminPrimitiveAsPeerBhv di ciascun peer al rispettivo PMAForwardAdminPrimitiveToPeerBhv presente sul superpeer e da quest'ultimo behaviour al PMAReceiveAdminPrimitiveAsSuperpeerBhv.

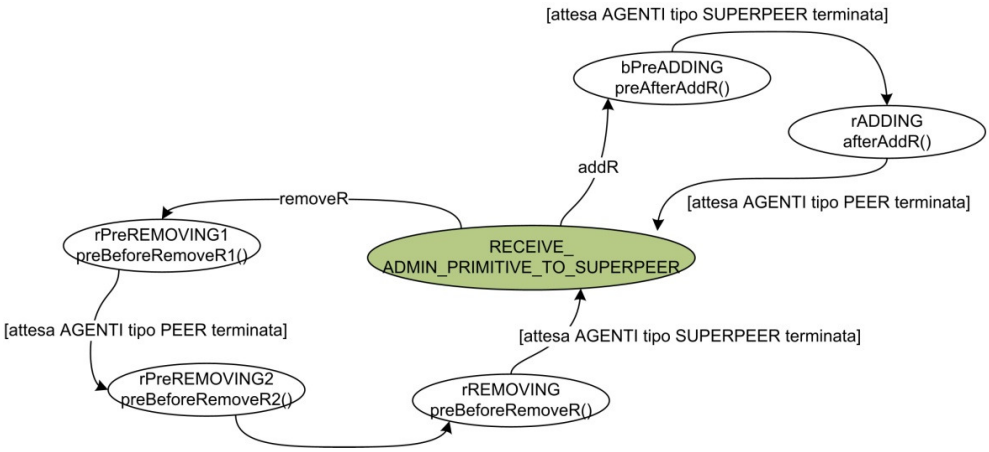


Figura 5-13 – PMAReceiveAdminPrimitiveAsSuperpeer: FSM

- rPreREMOVING2

  In questo stato il behaviour comunica la primitiva agli agenti di tipo superpeer da interrompere (RPA, CA e RRA).

- rREMOVING

  Nello stato rREMOVING il behaviour attende risposta direttamente dagli agenti CA, RPA e RRA (messaggi di tipo `SUPERPEER_AGENT_STATE_UPDATE`).

[11] Si veda il paragrafo 3.6.12.

- rPreADDING

  In questo stato il behaviour attende l'avvio degli agenti di tipo superpeer CA, RPA e RRA (messaggi di tipo `SUPERPEER_AGENT_STATE_UPDATE`) affinchè siano in esecuzione quando verranno avviati quelli di tipo peer.

- rADDING

  In questo stato il behaviour attende le conferme da parte degli agenti di tipo peer.

Nel caso si verifichi un errore durante l'esecuzione di una primitiva di tipo regione o nel caso scada il timeout `Parameters.MAXIMUM_PEER_STATE_CHECK_PERIOD` controllato da un PMACheckStateTimeoutBhv, il behaviour informa il PMAForwardAdminPrimitiveToSuperpeerBhv presente sul server tramite messaggi di tipo `TypeConversationId.REGION_STATE_UPDATE+"#"+Parameters.REGION_NAME`[12].

PMAForwardAdminPrimitiveToPeerBhv

Questo behaviour di tipo OneShot può essere aggiunto da un PMAReceiveAdminPrimitiveAsSuperpeerBhv e si occupa di eseguire sui peer del sistema le primitive startP, stopP, suspendP, resumeP, freezeP, thawP e downP (si veda anche il paragrafo *5.3.3*) informando dell'esito il creatore della primitiva e il behaviour padre PMAReceiveAdminPrimitiveAsSuperpeerBhv che lo ha generato.

Durante l'esecuzione delle primitive (ad eccezione dello startP e dello stopP associati alle primitive addR e removeR a livello di regione) è compito di questo behaviour informare il creatore anche sugli aggiornamenti di stato degli agenti del peer. In caso di errore, verrà inoltrata al creatore non solo la lista degli agenti che non hanno completato correttamente l'esecuzione della primitiva, ma anche anche la lista dei singoli behaviour che hanno riscontrato degli errori.

A titolo esemplificativo si riporta la *Figura 5-14* (p. 152) in cui viene schematizzato il comportamento di questo behaviour quando deve inoltrare la primitiva startP (si noti l'analogia con la *Figura 5-10* sull'esecuzione della primitiva removeR da parte del suerpeer).

[12] Il ConversationId deve essere strutturato in questo modo per le considerazioni riportate precedentemente.

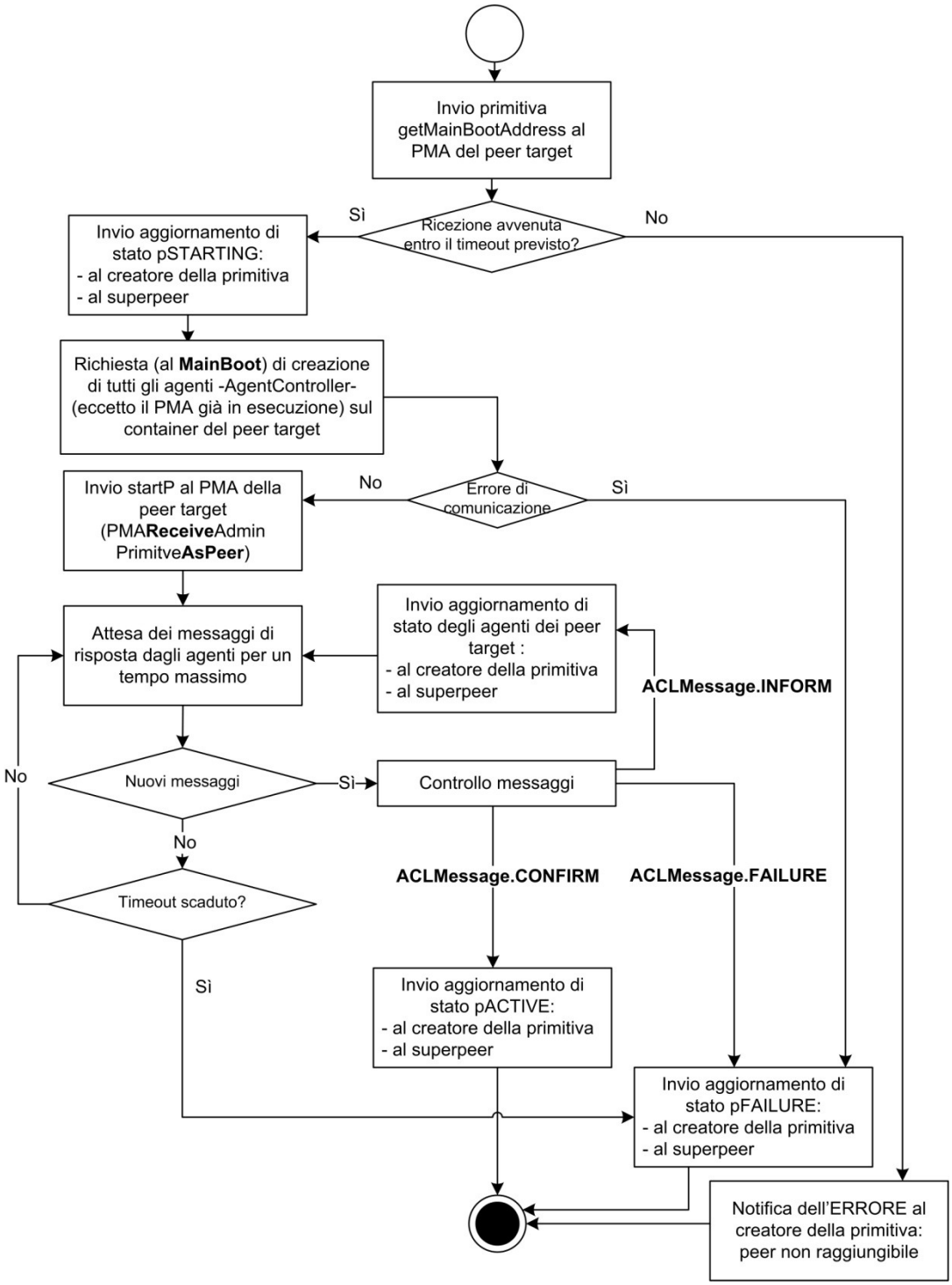


Figura 5-14 – PMAForwardAdminPrimitiveToPeerBhv: startP.

Tra i vari accorgimenti adottati in sede di implementazione, si cita la presenza di timeout relativi:

- all'esecuzione della primitiva getMainBootAddress, con cui il behaviour chiede al PMAReceiveAdminPrimitiveAsPeerBhv l'endpoint pubblico del suo MainBoot;
- al controllo dell'avvenuta confema da parte di tutti gli agenti del peer (Parameters.MAXIMUM_AGENT_STATE_CHECK_PERIOD).

Come accennato nel paragrafo *3.6* che descrive le ipotesi di realizzazione per le primitive di amministrazione, si ricorda che per quanto riguarda le primitive stopP, freezeP e downP è necessario che l'agente PMA del superpeer (tramite il suo behaviour PMAForwardAdminPrimitiveToPeerBhv) rimuova (nel caso di stopP e downP) o renda temporaneamente non valide (nel caso di freezeP) le informazioni relative alle tranche possedute dal peer che sta per essere arrestato o ibernato.

Queste operazioni vengono richieste dal suddetto behaviour all'agente RRA tramite, rispettivamente, un messaggio di tipo `UPDATE_DB_SUPERPEER_`*`REMOVE`*`_SOURCE_FOR_TRANCHE` ed un messaggio di tipo `UPDATE_DB_SUPERPEER_`*`HIDE`*`_SOURCE_FOR_TRANCHE` che viene elaborato dal behaviouur del UMA denominato RRAUpdateSourceForTrancheBhv.

Per la primitiva thawP, invece, il behaviour richiede all'agente RRA di rendere nuovamente attive le informazioni precedente nascoste da freezeP attraverso un messaggio di tipo `UPDATE_DB_SUPERPEER_`*`SHOW`*`_SOURCE_FOR_TRANCHE`.

PMAReceiveAdminPrimitiveAsPeerBhv

Questo behaviour si occupa di gestire:

1. lo stato degli agenti del peer tramite una variabile StateVector che contiene oggetti EntityState associati agli agenti (il behaviour riceve messaggi `AGENT_STATE_UPDATE`). Il behaviour si comporta in modo differente a seconda del tipo di messaggio ricevuto:
   a) `AGENT_STATE_UPDATE [ACLMessage.INFORM]`:

informa il PMAReceiveAdminPrimitiveAsPeerBhv dell'aggiornamento di stato di un behaviour;

b) `AGENT _STATE_UPDATE [ACLMessage.FAILURE]`:
inserisce il nome del behaviour che ha comunicato l'errore nella lista che verrà restituita al PMA del peer;

c) `AGENT _STATE_UPDATE [ACLMessage.CONFIRM]`:
memorizza la conferma di avvenuta esecuzione della primitiva da parte del behaviour mittente del messaggio; poi effettua un controllo: se tutti i behaviour hanno già inviato un `CONFIRM`, notifica l'esito positivo della primitiva al PMA del peer, altrimenti ritorna in attesa;

d) `AGENT_STATE_UPDATE_TIMEOUT`:
allo scadere del timeout invia in ogni caso una notifica (`AGENT_STATE_UPDATE`) al PMA del peer: se tutti i behaviour hanno inviato un `CONFIRM`, l'esito della primitiva eseguita dall'agente sarà positivo; in caso contrario, negativo.

2. l'inoltro delle primitive (di tipo peer) ai behaviour coordinatori di tutti gli agenti del peer;
3. l'esecuzione delle primitive di supporto descritte in *5.3.4*.

La struttura del behaviour ciclico è quella di una macchina a stati finiti costituita dai seguenti stati (cfr. *Figura 5-15*, p.156):

- RECEIVE_ADMIN_PRIMITIVE_TO_PEER

  In questo stato (di default) il behaviour rimane in attesa delle primitive di supporto e delle primitive di stato per il peer. Generalmente inoltra subito le primitive agli agenti del peer e precisamente ai loro behaviour coordinatori (####RECEIVEADMINPRIMITIVE) per poi cambiar stato e mettersi così in attesa delle risposte.

  Nel caso della primitiva thawP, prima di inoltrarla agli agenti del peer, attende che Persistence abbia caricato tutti i behaviour e che (attraverso il metodo `afterLoad()` dell'agente) sia stato eseguito per ognuno di essi il proprio metodo `afterThawP()`; nel caso della primitiva startP, si noti come il behaviour la riceva sia quando si trova in questo stato che quando si trova

nello stato bSTOPPED: la riceve in questo stato quando si tratta del primo avvio, mentre la riceve nello stato bSTOPPED quando si tratta di un nuovo avvio in seguito ad uno stopP.

- bSTARTING

  In questo stato il behaviour attende le risposte degli agenti relative alla primitiva startP (inoltrata loro quando si trovava nello stato di default o nello stato bSTOPPED) analogamente a quanto fa PMAReceiveAdminPrimitiveAsSuperpeerBhv, che attende risposte dal PMA dei peer; il behaviour assume un comportamento diverso a seconda del tipo di messaggi ricevuti dagli agenti, come specficato al punto numero 1 dell'elenco precedente.

- bSTOPPING

  In questo stato il behaviour attende le risposte degli agenti relative alla primitiva stopP (inoltrata loro quando si trovava nello stato di default), analogamente a quanto accade nello stato bSTARTING.

- bSTOPPED

  Quando si trova in questo stato il behaviour attende l'arrivo delle primitive getMainBootAddress e startP. La primitiva startP viene ricevuta anche nello stato di default RECEIVE_ADMIN_PRIMITIVE_TO_SUPERPEER quando il peer deve essere avviato per la prima volta. Quando, invece, si tratta di un avvio avvenuto in seguito ad un'arresto ottenuto con stopP, invece, il behaviour riceve la rimitiva startP proprio in questo stato pSTOPPED.

  Quanto perviene la primiitva startP, la inoltra a tutti gli agenti del peer.

- bSUSPENDING

  In questo stato il behaviour attende le risposte degli agenti relative alla primitiva suspendP (inoltrata loro quando si trovava nello stato di default), analogamente a quanto accade nello stato bSTARTING.

- bSUSPENDED

  Quando si trova in questo stato il behaviour attende l'arrivo delle primitive getMainBootAddress e resumeP; quando perviene quest'ultima primitiva, provvede ad inoltrarla a tutti gli agenti del peer.

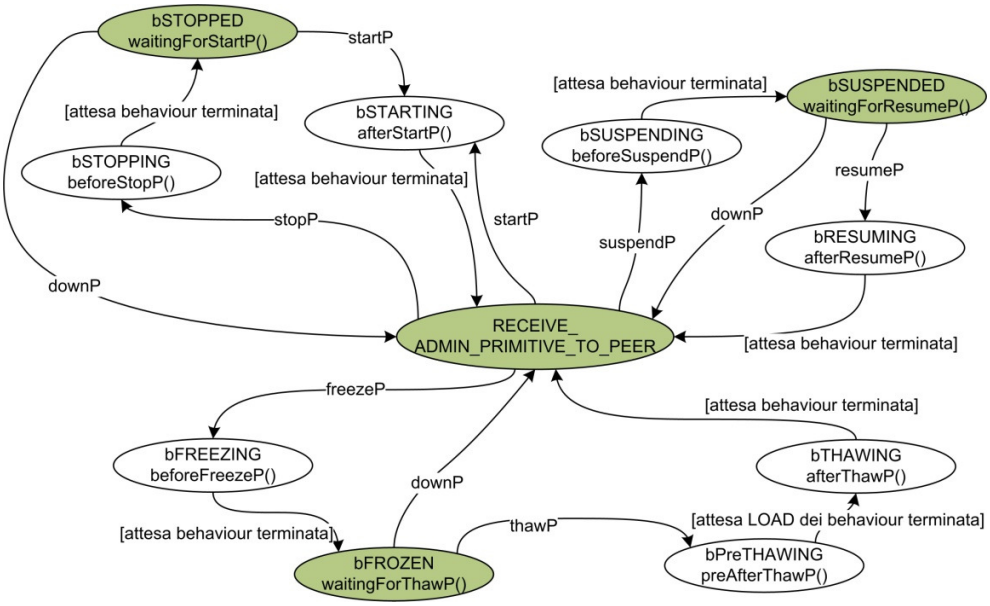


Figura 5-15 – PMAReceiveAdminPrimitiveAsPeerBhv: FSM.

- bRESUMING

  In questo stato il behaviour attende le risposte degli agenti relative alla primitiva suspendP (inoltrata loro quando si trovava nello stato di default), analogamente a quanto accade nello stato bSTARTING.

- bFREEZING

  In questo stato il behaviour attende le risposte degli agenti relative alla primitiva freezeP (inoltrata loro quando si trovava nello stato di default), analogamente a quanto accade nello stato bSTARTING.

- bFROZEN

  Quando si trova in questo stato il behaviour attende l'arrivo delle primitive getMainBootAddress e thawP; quando perviene quest'ultima primitiva, NON provvede ad inoltrarla subito a tutti gli agenti, ma passa nello stato successivo bPreTHAWING.

- bPreTHAWING

  In questo stato il behaviour attende che gli agenti del peer vengano creati dall'add-on Persistence e che confermino la propria corretta esecuzione tramite l'invio di un messaggio `AGENT_STATE_UPDATE` che comunichi lo stato pTHAWING. Una volta che tutti gli agenti hanno inviato il messaggio di conferma, il behaviour inoltra la primitiva thawP agli agenti e passa nello stato bTHAWING.

- bTHAWING

  In questo stato il behaviour attende le risposte degli agenti relative alla primitiva thawP (inoltrata loro quando si trovava nello stato bPreTHAWING), analogamente a quanto accade nello stato bSTARTING.

N O T E I M P O R T A N T I:

In ogni stato (stato di default, bSTOPPED, bFROZEN, bSUSPENDED) in cui il behaviour attende le primitive di stato (startP, stopP, freezeP, thawP) dal PMAForwardAdminPrimitiveToPeerBhv, il behaviour rimane sempre in attesa anche della primitiva downP: in questo modo, nonostante eventuali problemi verificatisi a livello di regione o relativamente alla scadenza di un timeout, il behaviour è in grado di riportarsi nello stato di default.

Nel caso si verifichi un errore durante l'esecuzione di una primitiva o nel caso scada il timeout `Parameters.MAXIMUM_AGENT_STATE_CHECK_PERIOD` controllato da un PMACheckStateTimeoutBhv, il behaviour informa il PMAForwardAdminPrimitiveToPeerBhv presente sul superpeer tramite messaggi di tipo `TypeConversationId.PEER_STATE_UPDATE+"#"+Parameters.PEER_NAME`[13].

PMASimpleUserInterfaceBhv

Questo behaviour dell'agente PMA si occupa di fare da *traits d'union* tra l'interfaccia grafica realizzata dalla classe SimpleAdminGUI.java (che estende `javax.swing.JFrame`) e tutti gli agenti, i peer o, più in generale, i

[13] Il ConversationId deve essere strutturato in questo modo per le considerazioni riportate precedentemente.

nodi che siano intenzionati a pubblicare un'informazione nell'area di testo della GUI del peer locale[14].

Inoltre si occupa di validare e tradurre le primitive tipo "linea di comando" inserite dall'operatore attraverso la GUI in oggetti di tipo Primitive per poi inoltrarli tramite messaggi ACL al corretto destinatario[15].

L'informazione che è possibile visualizzare nell'area di testo dell'interfaccia grafica viene comunicata attraverso messaggi di tipo `TypeConversationId.GUI_INTERPLAY`. Vengono interpretati in modo diverso (cambia in modo sostanziale il contenuto del campo "answer" dell'oggetto Primitive) a seconda di come sono contrassegnati:

- `ACLMessage.INFORM`
  il messaggioACL contiene semplicemente la stringa di testo da visualizzare;
- `ACLMessageCONFIRM`
  il messaggio ACL contiene un oggetto Primitive che porta informazioni circa l'esito comunque positivo dell'esecuzione della primitiva: tale esito può essere quello definitivo (che informa ad esempio del completamento della rimozione di una regione) o intermedio (che informa, ad esempio dell'aggiornamento di stato degli agenti di un peer durante la primitiva freezeP);
- `ACLMessage.FAILURE`
  il messaggio ACL contiene un oggetto Primitive associato ad una condizione di errore.

L'informazione da visualizzare, quindi, può essere una semplice stringa di testo, oppure l'esito (finale o intermedio) di una primitiva: in quest'ultimo caso il behaviour effettua una "traduzione" dell'oggetto Primtive in una sequenza di frasi più facilmente comprensibili.

[14] Tale intenzione viene palesata attraverso l'invio di messaggi di tipo `TypeConversationId.GUI_INTERPLAY` direttamente all'agente PMA del nodo su cui si vuole visualizzare l'informazione.

[15] Verranno inoltrati direttamente al peer target nel caso esso sia all'interno della regione oppure al server se la primitiva riguarda altre regioni o peer di altre regioni.

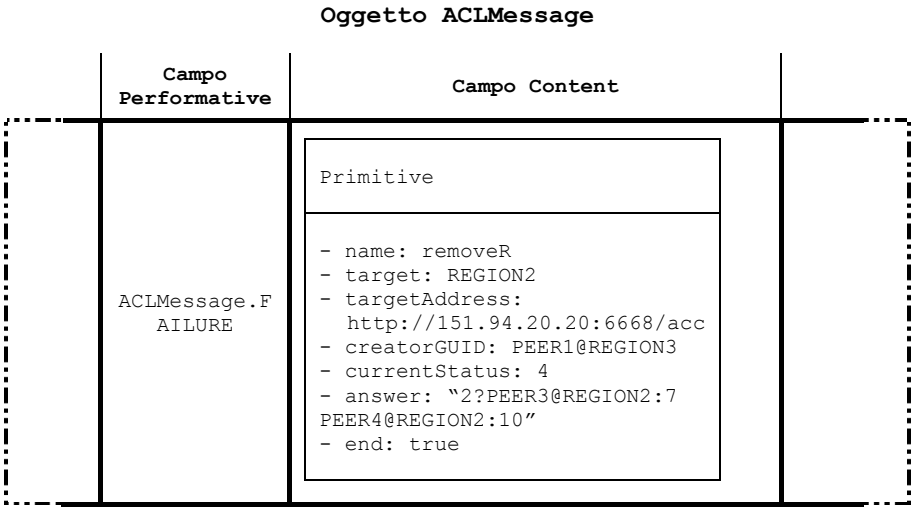


Figura 5-16 – Oggetto Primitive in un messaggio ACL [esempio].

Ad esempio la primitiva nella *Figura 5-16* creata dal superpeer della REGION3 (PEER1@REGION3) viene tradotta[16] nel messaggio:

"

[ERRORE] durante removeR REGION2.

I seguenti agenti/peer non hanno confermato lo stato pSTOPPED: PEER3@REGION2: pSTOPPING, PEER4@REGION2: pFAILURE

"

Nell'oggetto Primitive presente nel campo content dell'ACLMessage vengono riportate la codifiche numeriche degli stati definiti dalle classi `TypeRegionStatus` e `TypePeerStatus`:

- "currentStatus: 4":
  la regione è nello stato 4→rFAILURE
- "answer : 2?PEER3@REGION2:7 PEER4@REGION2:10":
  i peer PEER3@REGION2 e PEER4@REGION2 non sono riusciti a raggiungere lo stato obiettivo (2 → pSTOPPED) per i peer della regione e si trovano rispettivamente negli stati 7→pSTOPPING e 10→pFAILURE.

[16] Attraverso l'invocazione del metodo: `private String getTextFromPrimitive(Primitive primitive, ACLMessage msg)`

Il messaggio viene stampato a video tramite il metodo `public synchronized void printTextByAgent(final String text)` dell'oggetto SimpleAdminGUI[17].

Come verrà descritto accuratamente nel paragrafo *5.2.7*, in cui si scrive della classe SimpleAdminGUI.java, il metodo `printTextByAgent(..)` "incapsula" la chiamata al metodo locale `printText((String text))` di SimpleAdminGUI all'interno del metodo `run()` di un oggetto "`action`" di tipo interfaccia `java.lang.Runnable`, che viene poi schedulato attraverso il comando `SwingUtilities.invokeLater(action)`.

Questo espediente è necessario per garantire la corretta interazione fra il thread in cui vive l'agente PMA e quello in cui viene eseguita l'interfaccia grafica (SimpleAdminGUI).

L'operazione inversa[18], che dal comando inserito (stringa di testo) produce l'oggetto `Primitive` e restituisce il messaggio ACL pronto per essere inviato, prevede anche una prima fase di validazione dell'input che verifica la sintassi del comando. La seconda fase di validazione viene effettuata direttamente dal behaviour PMAReceiveAdminPrimitive AsServerBhv o dal PMAReceiveAdminPrimitiveAsSupereerBhv del nodo destinatario, in modo da evitare di eseguire una determinata primitiva su un nodo che non è in grado di eseguirla (ad esempio non viene permesso di eseguire la primitiva suspendP su un peer che si trova nello stato pSTOPPED o pFROZEN, oppure non si accetta di inoltrare una primitiva addR su una regione già aggiunta al sistema).

Il comando inserito attraverso la SimpleAdminGUI raggiunge il behaviour PMASimpleUserInterfaceBhv come messaggio ACL di tipo `TypeConversationId_GUI_INTERPLAY`, contrassegnato come `ACLMessage.REQUEST`.

[17] Il behaviour di cui si sta eseguendo l'analisi dispone di un riferimento diretto alla GUI locale - oggetto SimpleAdminGUI – in quanto tale riferimento è reso disponibile dalla classe principale dell'agente PMA, quella che estende `jade.core.agent`.

[18] Eseguita attraverso l'invocazione del metodo:

`private ACLMessage validateInputAndMakeMessage(String input)`

5.2.4 Agente RPA (RendezvousProxyAgent) [new]

L'agente RPA (RendezvousProxyAgent) è situato nel container del superpeer e offre principalmente due servizi: quello di rendezvous (per la procedura di TCP Hole Punching) e quello di proxy (per l'inoltro dei messaggi provenienti dall'esterno della regione verso peer interni alla regione che non dispongono di un endpoint pubblico direttamente associato ad un loro endpoint privato). Il primo si basa su comunicazioni TCP e gestisce oggetti `ServerSocket` e `Socket`, mentre il secondo si appoggia al sistema del messaggi ACL messo a disposizione da JADE.

L'agente è costituito dai seguenti behaviour:

1. **RPARendezvousBhv**

   Questo behaviour ciclico, aggiunto direttamente dalla classe principale dell'agente, si mette in ascolto sull'oggetto `ServerSocket` associato alla porta di default per il servizio di rendezvous e, quando rileva una connessione in ingresso, avvia il behaviour OneShot RPARendezvousServerBhv in un thread indipendente (viene avviato come ThreadedBehaviour).

   RPARendezvousBhv mantiene un riferimento a tutti i behaviour RPARendezvousServer creati e a tutti i socket utilizzati, in modo da poterli interrompere in qualsiasi momento, consentendo il rilascio delle risorse (`Socket`) da essi utilizzate.

   La struttura dati adibita allo scopo è un `Vector<RendezvousServer>`, dove RendezvousServer è un oggetto composto da un attributo di tipo RpaRendezvousServerBhv e da un attributo di tipo `Socket`.

2. **RPARendezvousServerBhv**

   Il behaviour gestisce la connessione in ingresso effettuata dall'agente HA dell'altro peer durante la procedura di Hole Punching: preleva l'endpoint pubblico (ip:porta) dell'interlocutore e lo comunica all'agente HA locale.

   Continua a rimanere in esecuzione mantenendo aperta la connessione finchè l'agente HA non la chiude o finchè non viene rimossa la regione in cui l'RPA è situato (cfr. nota 20, p. 164).

3. **RPAProxyBhv**

   Questo behaviour rimane in ascolto di messaggi provenienti dall'esterno da inoltrare a specifici agenti dei peer interni alla regione: inoltra i vari messaggi prelevando l'AID dell'agente

destinatario attraverso i metodi `getIdAgentToContact()` e `getHttpMtpAgentToContact()` dell'oggetto `Request` contenuto nell'oggetto `Answer` presente nel campo `Content` del messaggio ACL ricevuto

Il behaviour è sensibile ai seguenti `ConversationId`:

ConversationID	Agente
PROPOSE_TICKET	DMA
IP_DA_UPLOAD_TRANCHE	DA
RESPONSE_TO_REUSE_OLD_SOCKET	UA
RENDEZVOUS_TO_FONTE _OTHER_PEER_PUBLIC_ADDRESS	HA
RENDEZVOUS_TO_RICHIEDENTE _OTHER_PEER_PUBLIC_ADDRESS	HA
COMPLETED_DOWNLOAD	UMA

Tabella 5-6 – ConversationID gestiti dal behaviour RPAProxyBhv.

4. **RPACheckLogFileBhv**
   Questo behaviour (identico a quello previsto per gli altri agenti) si occupa di gestire i file di log.
5. **RPAReceiveAdminPrimitiveBhv**
   Questo è il behaviour "coordinatore" dell'agente RPA e si comporta come descritto al paragrafo *5.2.1.7*
6. **RPACheckStateTimeoutBhv**
   cfr. par. *5.2.1.8*.

Quando l'agente RPA riceve le primitive di stato stopP e downP relative all'esecuzione delle primitive removeR e downR, il behaviour RPARendezvousBhv esegue le seguenti operazioni:

- Interrompe l'esecuzione dei tutti i behaviour RPARendezvousServerBhv: tramite l'invocazione di
  ```
  tbf.getThread(rendezvousServerVector.get(i).bhv)
  .interrupt();
  ```
- esegue il metodo `close()` su tutti i `Socket` utilizzati dai behaviour appena interrotti.

5.2.5 Agente HA (HoleAgent) [new]

L'agente HA è costituito, oltre che dai behaviour "comuni" a tutti gli altri behaviour (HACheckLogFileBhv, HAChechStateTimeoutBhv, HAReceiveAdminPrimitiveBhv), anche dal behaviour HARendezvousAndTcpHolePunchingBhv e dai due behaviour HAHoleClientBhv e HAHoleServerBhv.

HARendezvousAndTcpHolePunchingBhv

Questo behaviour interagisce con l'agente RPA al fine di portare a termine la fase di rendezvous[19] della procedura di TCP Hole Punching.

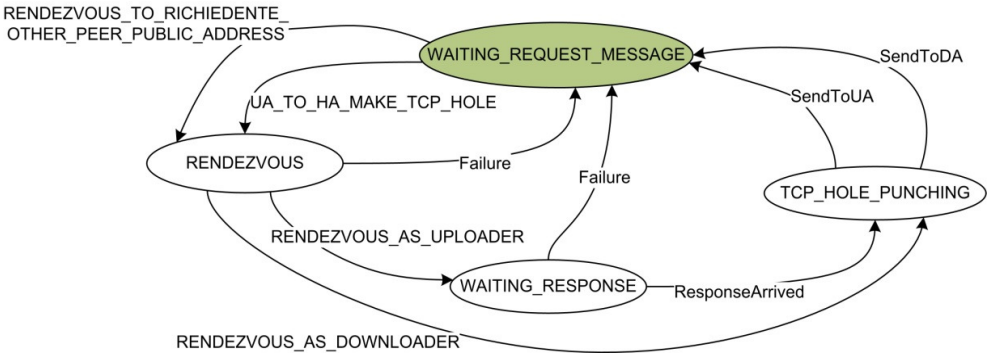


Figura 5-17 – Behaviour HARendezvousAndTcpHolePunchingBhv: FSM.

- Nello stato di default WAITING_REQUEST_MESSAGE attende un messaggio di richiesta dall'agente UA (che gli chiede di eseguire la procedura di Hole Punching come UPLOADER) o dall'agente RPA (che gli fornisce direttamente l'endpoint del peer uploader e che gli chiede, quindi, di eseguire la procedura come DOWNLOADER).
- Nello stato RENDEZVOUS l'HA tenta di instaurare una comunicazione TCP con l'agente RPA dell'altro peer: attraverso tale comunicazione, l'RPA è in grado di rilevarne l'endpoint pubblico.
- Nello stato WAITING_RESPONSE si può trovare solo l'HA del peer UPLOADER, in quando è l'unico a dover ancora attendere dal RPA l'endpoint pubblico dell'altro HA (quello del DOWNLOADER).

[19] Si ricorda che la fase di rendezvous consiste nella comunicazione, attraverso l'RPA, dell'endpoint pubblico del HA di un peer verso l'HA dell'altro peer.

- Lo stato TCP_HOLE_PUNCHING, inizialmente progettato per instaurare la vera e propria connessione TCP, vede la semplice comunicazione da parte del HA dell'endpoint del DOWNLOADER al UA e dell'endpoint dell'UPLOADER al DA: saranno questi due agenti ad instaurare la connessione TCP.

Si noti che:

- l'agente HA tenta di contattare l'agente RPA dell'altra regione attraverso socket precedentemente utilizzati (le connessioni con l'RPA non vengono chiuse dopo la fase di rendezvous; ovviamente anche l'agente RPA è strutturato in modo tale da supportare questa funzionalità)[20];
- nel caso in cui si verifichi un errore di comunicazione con l'RPA, l'HA ne dà tempestiva notifica al UA (o al DA a seconda del ruolo del peer in cui si trova l'agente HA), il quale provvederà a notificare la rilevazione dell'errore all'HA dell'altro peer. Le comunicazioni successive verso l'RPA saranno instaurate da una porta locale diversa (che corrisponderà ad un endpoint pubblico diverso).

HAHoleClientBhv e HAHoleServerBhv

Questi due behaviour si occupano della effettiva instaurazione della connessione TCP prevista dalla procedura di TCP Hole Punching. Inizialmente si era progettato che venissero aggiunti dall'agente HA, tuttavia, dal momento che la comunicazione di un oggetto Socket dall'agente HA all'agente UA o DA si rivelava meno lineare del previsto, si è deciso di far eseguire tali behaviour direttamente dagli agenti UA e DA: in questo modo tali agenti diventano gli effettivi responsabili della creazione della connessione TCP.

[20] E' stato necessario adottare questo accorgimento per tre motivi:

a. evitare di dover instaurare una connessione con l'RPA ogni volta che si deve trasferire un file da/verso peer nella stessa regione;
b. escludere eventuali problematiche relative alla mappatura da parte del NAT tra endpoint pubblico ed endpoint privato delle comunicazioni instaurate dall'HA
c. escludere eventuali problemi connessi alle tempistiche di rilascio dei socket da parte del sistema operativo.

Il comportamento dei due behaviour viene descritto approfonditamente nel paragrafo *5.4.4*.

5.2.6 Add-on Persistence [new]

L'integrazione di questo componente aggiuntivo ideato dagli stessi progettisti di JADE si è rivelata abbastanza laboriosa.

Infatti è stato necessario documentarsi adeguatamente sulla libreria Hibernate utilizzata da Persistence ed è stato necessario utilizzare un altro DBMS (HyperSQL), dal momento che quello usato dal prototipo esistente (SQLite) non era supportato da Hibernate.

L'integrazione dell'add-on comincia dalla configurazione dei file:

- di Hibernate (hibernate.properties);
- di HyperSQL (server.properties);
- di Persistence (main.properties e container.properties).

Tali file di configurazione definiscono i parametri utilizzati dai tre componenti: in particolare quelli di HyperSQL servono a delineare le funzioni offerte dal servizio server del DBMS, con il quale si rende possibile la connessione da remoto (dagli altri peer) al database localizzato sul superpeer[21].

Persistence richiede che Hibernate (il vero motore dell'add-on) e HyperSQL (che garantisce la persistenza) funzionino a dovere su ogni nodo del sistema: solo in questo modo riesce a garantire le funzioni di memorizzazione persistente degli agenti che sono richieste da questo progetto.

C'è da dire che l'add-on dispone di primitive che permettono il salvataggio in maniera persistente (e quindi su database) di interi container o che consentono una forma di ibernazione degli agenti che in realtà li mantiene in una condizione di "letargo". Tuttavia si è ritenuto che tali funzioni rivelassero una sostanziale rigidità e una bassa duttilità per gli scopi previsti dal sistema. Quindi si è preferito concentrarsi su un approccio più diretto come descritto nei paragrafi seguenti.

[21] L'avvio del server HyperSQL avviene ad opera del MainBoot del superpeer.

5.2.6.1 Esecuzione delle primitive

Le modalità con sui possono essere eseguite le primitive offerte da Persistence sono state elencate nel paragrafo 3.5.3. L'implementazione attuale del sistema prevede che le operazioni di salvataggio (su database) e di caricamento (da database) di tutti gli agenti del peer da ibernare o scongelare vengano richieste dall'agente PMA del superpeer all'agente AMS della regione.

Questa scelta è stata adottata in modo da rendere le procedure di freezeP e thawP il più versatili possibile: infatti in questo modo è possibile far eseguire l'operazione di salvataggio dello stato di un agente qualsiasi da un nodo qualunque del sistema, dal momento che in realtà l'unica operazione richiesta è l'invio di un messaggio ACL all'agente AMS della regione.

Per quanto riguarda questo progetto, la procedura prevede che sia l'agente PMA del superpeer ad inviare un messaggio ACL all'agente AMS: con tale messaggio chiede all'AMS di eseguire una Action (`jade.content.onto.basic.Action`) secondo l'ontologia specificata per Persistence (definita nel package `jade.domain.persistence`).

5.2.6.2 Procedura per l'esecuzione di freezeP

L'implementazione della primitiva freezeP viene descritta nel paragrafo *5.3.3.4* (p.195).

Al fine di chiarire meglio l'interazione fra gli agenti PMA e AMS per l'utilizzo delle funzionalità offerte dall'add-on Persistence, di seguito si riportano:

- la porzione di codice che esegue le operazioni richieste (nel caso di freezeP) per tutti gli agenti del peer da congelare;
- il metodo `requestAMSToSave(..)` che esegue la richiesta all'AMS.

```
...

//elenco agenti da salvare
String[] agentsToSaveGUID =
                ((AgentLayer.PMA)(myAgent)).getPeerAgentStateVector().
                getAllEntities();
agentsToSaveGUID = fixAgentsToSaveGUID(agentsToSaveGUID);

boolean errorePersistence = false;
```

```
String dettagliErrorePersistence = "";
PersistenceHelper helper = null;

//ottenimento PersistenceHelper
try
{
   helper = (PersistenceHelper)
                myAgent.getHelper(PersistenceHelper.NAME);
}
catch (Exception e)
{
   e.printStackTrace();
   errorePersistence = true;
   dettagliErrorePersistence += ": errore durante l'esecuzione di
                getHelper()";
}

System.out.println("Default Repository:" +
                PersistenceManager.DEFAULT_REPOSITORY);

//Configurazione ContentManager (Ontologia, codec e language)
Ontology ontology = PersistenceOntology.getInstance();
ContentManager manager  = (ContentManager)
                myAgent.getContentManager();
Codec codec = new SLCodec();

manager.registerLanguage(codec, FIPANames.ContentLanguage.FIPA_SL0);
manager.registerOntology(ontology);

//salvataggio di tutti gli agenti
for(int i = 0; i<=agentsToSaveGUID.length-1; i++)
{
   try
   {
      requestAMSToSave(agentsToSaveGUID[i], ontology, manager);
   }
   catch (Exception e)
   {
      e.printStackTrace();
      errorePersistence = true;
      dettagliErrorePersistence += ": errore durante il salvataggio
                dell'agente "+agentsToSaveGUID[i];
      break;
    }
}

//verifica del salvataggio
if(!errorePersistence)
{
   try
   {
      Thread.sleep(7000);
   }
   catch (InterruptedException e)
   {
       e.printStackTrace();
   }

   System.out.println("PMAForwardAdminPrimitiveToPeer: Elenco agenti
                salvati..");

try
{

   int peerNodeIndex = getNodeNameIndex(helper.getNodes(),
                primitiveReceived.getTarget()); // PEER2 REGION1
```

```
                invece di PEER2@REGION1

   System.out.println("PMAForwardAdminPrimitiveToPeer:
                NODO["+peerNodeIndex+"] =
                "+helper.getNodes()[peerNodeIndex]);

   String[] savedAgents =
                helper.getSavedAgents(helper.getNodes()[peerNodeIndex]
                , PersistenceManager.DEFAULT_REPOSITORY);

   System.out.println("Agenti Salvati con Persistence:");
   for (int i = 0; i<=savedAgents.length-1; i++)
   {
       System.out.println("["+i+"] "+ savedAgents[i]);
   }

   System.out.println("TOTALE Agenti Salvati con Persistence:
                "+savedAgents.length);

   if(!allAgentsSaved(savedAgents))
   {
      errorePersistence = true;
      dettagliErrorePersistence += ": non tutti gli agenti del peer
                sono stati salvati correttamente.";
   }


   } catch (ServiceException e1) {
     // TODO Auto-generated catch block
     e1.printStackTrace();
     errorePersistence = true;
   } catch (IMTPException e1) {
     // TODO Auto-generated catch block
     e1.printStackTrace();
     errorePersistence = true;
   } catch (NotFoundException e1) {
   // TODO Auto-generated catch block
   e1.printStackTrace();
   errorePersistence = true;
   }

}
if(errorePersistence)
{
                    ...
}
else
{
   ...
                }

...
```

Figura 5-18 – Persistence: salvataggio degli agenti di un peer [codice].

```
private void requestAMSToSave(String agentToSaveGUID, Ontology
                ontology, ContentManager manager) throws
                OntologyException, CodecException
{
   //creazione messaggio ACL
   ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);

   //creazione AID dell'agente
   AID agentToSaveAID = new AID(agentToSaveGUID, AID.ISGUID);
  agentToSaveAID.addAddresses(primitiveReceived.getTargetAddress());

   //AID dell'agente AMS
   AID amsAID = myAgent.getAMS();

   //composizione messaggio ACL
   msg.setSender(myAgent.getAID());
   msg.addReceiver(amsAID);
   msg.setLanguage(FIPANames.ContentLanguage.FIPA_SL0);
   msg.setOntology(ontology.getName());

   //creazione SaveAgent
   SaveAgent saveAgent = new SaveAgent();
   saveAgent.setAgent(agentToSaveAID);
   saveAgent.setRepository(PersistenceManager.DEFAULT_REPOSITORY);

   //creazione Action
   Action action = new Action();
   action.setActor(amsAID);
   action.setAction(saveAgent);

   //riempimento campo content del messaggio ACL
   manager.fillContent(msg, action);

   //invio del messaggio
   System.out.println( "[" + myAgent.getLocalName() + "] Sending the
                message..." + msg.toString());

   myAgent.send(msg);
}
```

Figura 5-19 – Persistence e AMS: salvataggio di un agente [codice].

5.2.6.3 Procedura per l'esecuzione di thawP

L'implementazione della primitiva thawP viene descritta nel paragrafo *5.3.3.5* (p.197). Per completezza si riportano di seguito le porzioni di codice previste dalla primitiva thawP che eseguono le operazioni inverse rispetto a quelle utilizzate per freezeP (cfr. paragrafo precedente).

```
...

//elenco agenti da caricare
String[] agentsToLoadGUID =
                ((AgentLayer.PMA)(myAgent)).getPeerAgentStateVector().
                getAllEntities();

agentsToLoadGUID = fixAgentsToLoadGUID(agentsToLoadGUID);
boolean errorePersistence = false;
```

```
String dettagliErrorePersistence = "";

//ottenimento PersistenceHelper
PersistenceHelper helper = null;
try
{
    helper = (PersistenceHelper)
                myAgent.getHelper(PersistenceHelper.NAME);
    System.out.println("Nodo 0:" +helper.getNodes()[0]);
    System.out.println("Default Repository:" +
                PersistenceManager.DEFAULT_REPOSITORY);
    System.out.println("Repository 0:" +
                (helper.getRepositories(helper.getNodes()[0]))[0]);
}
catch (Exception e)
{
   e.printStackTrace();
   errorePersistence = true;
   dettagliErrorePersistence += ": errore durante l'esecuzione di
                getHelper()";
}

//Configurazione ContentManager (Ontologia, codec e language)
Ontology ontology = PersistenceOntology.getInstance();
ContentManager manager  = (ContentManager) myAgent.getContentManager();
Codec codec = new SLCodec();

manager.registerLanguage(codec, FIPANames.ContentLanguage.FIPA_SL0);
manager.registerOntology(ontology);

StringTokenizer tk = new StringTokenizer(primitiveReceived.getTarget(),
                "@"); //PEER2_REGION1 e non PEER2@REGION1
String destinationContainerIDName = tk.nextToken()+"_"+tk.nextToken();

//Caricamento di tutti gli agenti
for(int i = 0; i<=agentsToLoadGUID.length-1; i++)
{
   try
   {
      requestAMSToLoad(agentsToLoadGUID[i], destinationContainerIDName,
                ontology, manager);
   }
   catch (Exception e)
   {
      e.printStackTrace();
      errorePersistence = true;
      dettagliErrorePersistence += ": errore durante l'invio del
                messaggio a "+agentsToLoadGUID[i];
      break;
    }
}


if(errorePersistence)
{
   ...
}
else
{
   ...
}

...
```

Figura 5-20 – Persistence: caricamento degli agenti di un peer [codice].

```
private void requestAMSToLoad(String agentToLoadGUID, String
                destinationContainerIDName, Ontology ontology,
                ContentManager manager) throws OntologyException,
                CodecException
{
   //creazione messaggio ACL
   ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);

   //creazione AID dell'agente
   AID agentToLoadAID = new AID(agentToLoadGUID, AID.ISGUID);
                agentToLoadAID.addAddresses(primitiveReceived.getTa
                rgetAddress());

   //AID dell'agente AMS
   AID amsAID = myAgent.getAMS();

   //composizione messaggio ACL
   msg.setSender(myAgent.getAID());
   msg.addReceiver(amsAID);
   msg.setLanguage(FIPANames.ContentLanguage.FIPA_SL0);
   msg.setOntology(ontology.getName());

   //creazione ContainerID
   ContainerID containerID = new ContainerID();
   containerID.setName(destinationContainerIDName);
   containerID.setAddress(primitiveReceived.getTargetAddress());

   //creazione LoadAgent
   LoadAgent loadAgent = new LoadAgent();
   loadAgent.setAgent(agentToLoadAID);
   loadAgent.setRepository(PersistenceManager.DEFAULT_REPOSITORY);
   loadAgent.setWhere(containerID);

   //creazione Action
   Action action = new Action();
   action.setActor(amsAID);
   action.setAction(loadAgent);

   //riempimento campo content del messaggio ACL
   manager.fillContent(msg, action);

   //invio del messaggio
   System.out.println( "[" + myAgent.getLocalName() + "] Sending the
                message..." + msg.toString());

   myAgent.send(msg);
}
```

Figura 5-21 – Persistence e AMS: caricamento di un agente [codice]..

Per tutti i behaviour di ogni agente che viene risvegliato da una primitiva thawP deve essere invocato il metodo pubblico `afterThawP()` definito localmente per ogni behaviour.

Tale operazione, come accennato in precedenza, deve essere eseguita dall'interno del metodo `afterLoad()` della classe principale dell'agente (quella che estende `jade.core.Agent`) e serve per re-inizializzare quelle variabili che ogni behaviour definisce localmente ma che non possono

essere salvate in modo persistente nel database (vengono appositamente marcate con la keyword **transient**).

5.2.7 SimpleAdminGUI [new]

La classe SimpleAdminGUI realizza una semplice interfaccia grafica con cui l'utente è in grado di interagire con l'agente PMA. L'obiettivo con cui è stata sviluppata questa interfaccia è principalmente quello di poter inserire i comandi associati alle primitive di amministrazione e di riceverne, quindi, l'esito.

I componenti principali dell'interfaccia, che estende javax.swing.JFrame, sono i seguenti:

1. **javax.swing.JTextArea**
   Questo componente si occupa di mostrare all'utilizzatore le stringhe di testo inviate tramite il JCheckBox o giunte dall'agente PMA (PMASimpleUserInterfaceBhv);
2. **javax.swing.JScrollPane**
   Questo componente consente di scorrere la JTextArea per leggere testo al di fuori del campo visivo;

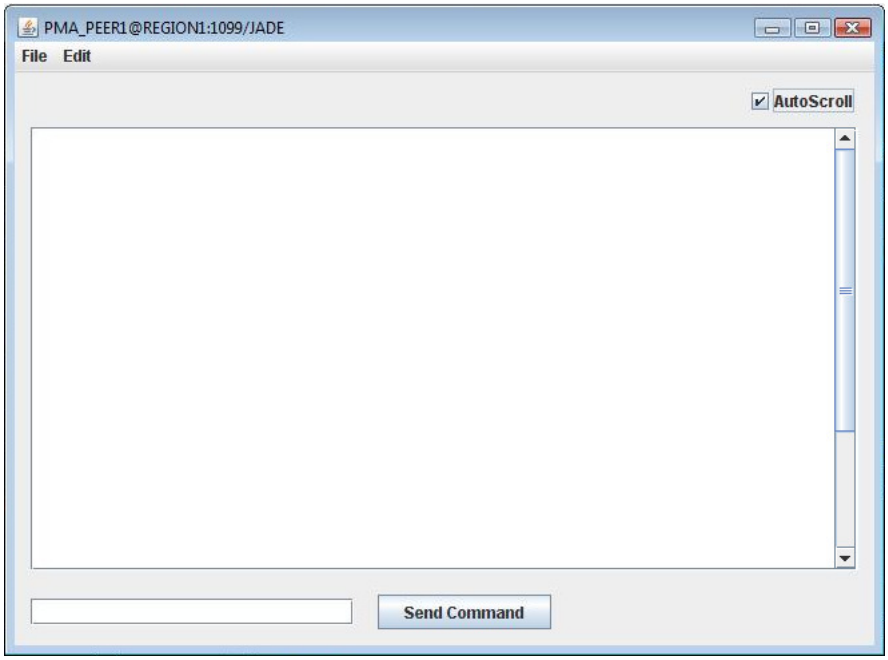


Figura 5-22 – SimpleAdminGUI: interfaccia grafica.

3. **javax.swing.JCheckBox**
   Consente di attivare/disattivare lo scrolling automatico della JTextArea;
4. **javax.swing.JTextField**
   Consente all'utilizzatore di inserire un comando per eseguire una primitiva;
5. **javax.swing.JButton**
   Invia (SendCommand) il messaggio all'agente PMA che si occuperà di validare l'input e costruire la primitiva;
6. **javax.swing.JMenu**
   Offre la possibilità di chiudere la GUI, di interrompere la JavaVirtualMachine (`System.exit(0)`) e di inserire alcuni comandi.

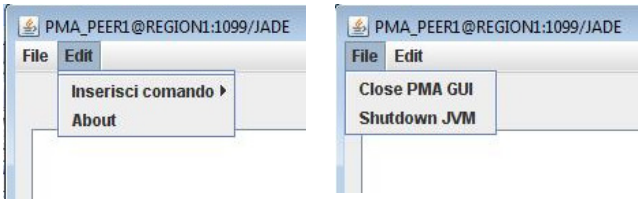


Figura 5-23 – SimpleAdminGUI: Menu.

Per pubblicare del testo direttamente nella `JTextArea` è possibile utilizzare il metodo pubblico `printText()` ma, affinchè lo possa fare un agente JADE, è necessario adottare un accorgimento: gli agenti devono interagire con questo componente SimpleAdminGUI tramite il metodo pubblico `synchronized printTextByAgent(final string text)` offerto dal componente stesso.

Questo metodo assicura la corretta esecuzione del metodo `printText()` creando un oggetto `Runnable` che, all'interno del proprio metodo `run()`, esegua `printText()`.

In questo modo, tramite `invokeLater(Runnable action)` della classe `SwingUtilities`, è possibile far schedulare in modo corretto l'operazione richiesta.

```
public synchronized void printTextByAgent(final String text)
{
    Runnable action = new Runnable()
            {
                public void run()
                {
                    printText(text);
                }
            };

    SwingUtilities.invokeLater(action);

}
```

Figura 5-24 – SimpleAdminGUI.java: metodo printTextByAgent(..).

5.2.8 Agente DA (DownloaderAgent) [modified]

L'agente DA si occupa di ricevere le tranche inviate dall'agente UA del peer fonte.

Dopo che gli agenti DMA, MA e UMA hanno portato a termine la procedura di selezione delle fonti, l'UA tenta di mettersi in contatto col DA per iniziare ad inviargli la tranche.

Nelle comunicazioni extra-regione, a differenza di quanto accade nel caso intra-regione, gli agenti UA e DA delle due regioni non hanno un endpoint pubblico visibile dall'esterno della propria regione; quindi devono appoggiarsi ai rispettivi agenti HA come descritto nel *CAPITOLO 3* ed analizzato più approfonditamente nel paragrafo *5.4*.

Il behaviour più importante dell'agente DA, oltre agli ormai consueti behaviour di supporto (elencati, ad esempio, per l'agente HA) è il DAReceivedTrancheFromUaBhv. Questo behaviour, per effettuare il vero e proprio download della tranche dal peer fonte, si avvale di altri behaviour chiamati DADownloaderTrancheBhv: ne viene aggiunto uno (su un thread indipendente) per ogni tranche da trasferire. Siccome, ai fini della gestione delle primitive di amministrazione, è necessario mantenere il controllo di tutti i behaviour aggiunti, il DAReceivedTrancheFromUaBhv mantiene in memoria, per ognuno di essi, il riferimento diretto al behaviour ed il riferimento all'oggetto Socket utilizzato per il trasferimento[22].

[22] Viene utilizzata una struttura dati simile a quella usata dal behaviour RPARendezvousBhv per controllare i vari RPARendezvousServerBhv creati.

Il behaviour DAReceivedTrancheFromUaBhv è organizzato come una macchina a stati finiti (Figura 5-25, p.175) e gli stati possibili sono i seguenti:

- WAITING_FOR_MESSAGE

  Il behaviour attende il messaggio `IP_DA_UPLOAD_TRANCHE` dall'agente della fonte (se è in un'altra regione tale messaggio giunge via RPA) con il quale vengono comunicate le DownloadInformation che identificano la Tranche da ricevere;

- INTRA_REGION

  In questo stato il behaviour svolge tutte le attività previste dal precedente prototipo quando la fonte si trovava nella stessa regione del peer downloader (richiesta diretta di ip e porta locali per l'invio della tranche, etc..);

- EXTRA_REGION_PHASE_1

  Il behaviour procede alla configurazione del trasferimento. Le attività si differenziano in virtù del fatto che UA richieda o meno il riutilizzo di un socket utilizzato in precedenza: se viene richiesto il riutilizzo e il DA dispone di un socket valido, si passerà nello stato EXTRA_REGION_PHASE_2b; altrimenti nello stato WAITING_HA;

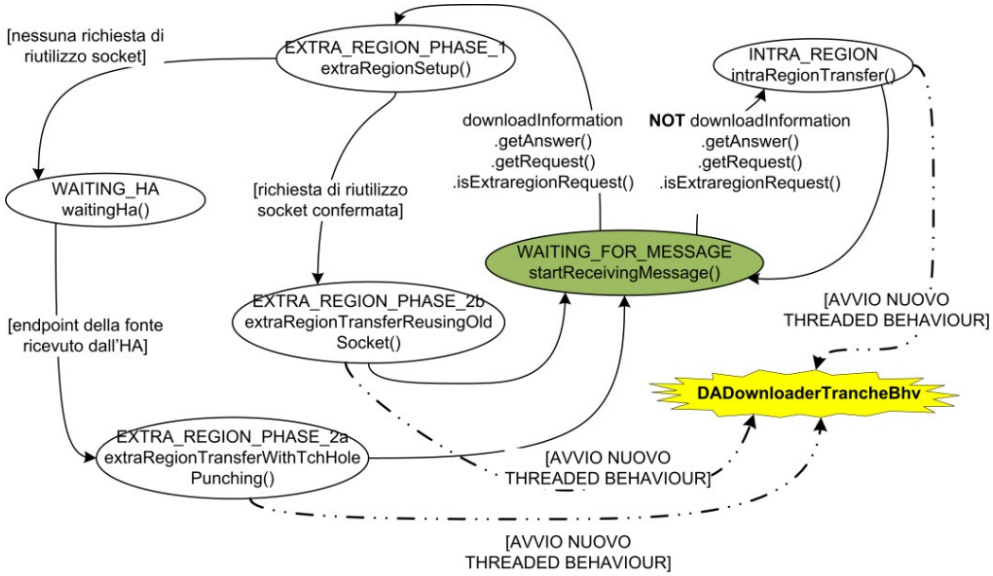


Figura 5-25 – Behaviour DAReceiveDTrancheFromUABhv: FSM.

- WAITING_HA

  In questo stato il behaviour attende un messaggio dal HA che lo informi:

  a) del verificarsi di un errore durante la fase di rendezvous della procedura di TCP Hole Punching;
  b) dell'endpoint pubblico del peer fonte (gestito dal UA).

  Se viene comunicato l'endpoint dell'altro peer, si passa nello stato EXTRA_REGION_PHASE_2a, altrimenti si torna nello stato di default;

- EXTRA_REGION_PHASE_2a

  In questo stato avviene la vera e propria instaurazione della connessione TCP diretta fra i due peer prevista dalla procedura di TCP Hole Punching descritta nel *CAPITOLO 2* e analizzata in parte nel paragrafo 5.4;

- EXTRA_REGION_PHASE_2b

  In questo stato avviene il download della tranche dal peer fonte attraverso una connessione instaurata in precedenza e lasciata attiva proprio per questo scopo.

Il behaviour, quando perviene la primitiva freezeP, stopP, o downP si comporta in modo simile a quanto fa il behaviour RPARendezvousBhv nei confronti degli RPARendezvousServerBhv: interrompe i behaviour e invoca il metodo `close()` su sutti gli oggetti `Socket` da essi utilizzati, chiudendo le connessioni verso gli agenti UA dei peer fonte contattati in precedenza.

Per ulteriori dettagli sull'interazione con gli agenti UA e HA si veda il paragrafo *5.4*.

5.2.9 Agente UA (UploaderAgent) [modified]

La struttura dell'agente UA è praticamente identica a quella dell'agente DA. Ovviamente cambiano le operazioni che vengono compiute all'interno dei vari stati e cambia il ruolo svolto dal behaviour durante l'interazione con il DA.

La macchina a stati finiti che costituisce la struttura del behaviour più importante dell'agente UA (l'UASendThancheToDaBhv) è rappresentata nella *Figura 5-26* (p.178).

- WAITING_FOR_MESSAGE

  Il behaviour attende dall'agente UMA il messaggio `READY_FOR_UPLOAD_TRANCHE`, con cui vengono comunicate le DownloadInformation che identificano la Tranche da ricevere;

- INTRA_REGION

  In questo stato il behaviour svolge tutte le attiità previste del precedente prototipo quando il peer downloader si trovava nella stessa degione del peer uploader (richiesta diretta di ip e porta locali per l'invio della tranche, etc..);

- EXTRA_REGION_PHASE_1

  Il behaviour procede alla configurazione del trasferimento. Le attività si differenziano in virtù del fatto che sia presente o meno un socket utilizzato in precedenza verso lo stesso DA: se è presente, viene richiesto il riutilizzo e, se tale riutilizzo viene confermato, si passerà nello stato EXTRA_REGION_PHASE_2b; altrimenti nello stato WAITING_HA;

- WAITING_HA

  In questo stato il behaviour attende un messaggio dal HA che lo informi:

  a) del verificarsi di un errore durante la fase di rendezvous della procedura di TCP Hole Punching
  b) dell'endpoint pubblico del peer richiedente (gestito dal DA)

  Se viene comunicato l'endpoint dell'altro peer, si passa nello stato EXTRA_REGION_PHASE_2a, altrimenti si torna nello stato di default;

- EXTRA_REGION_PHASE_2a

  In questo stato avviene la vera e propria instaurazione della connessione TCP diretta fra i due peer prevista dalla procedura di TCP Hole Punching descritta nel *CAPITOLO 2* e analizzata in parte nel paragrafo *5.4*;

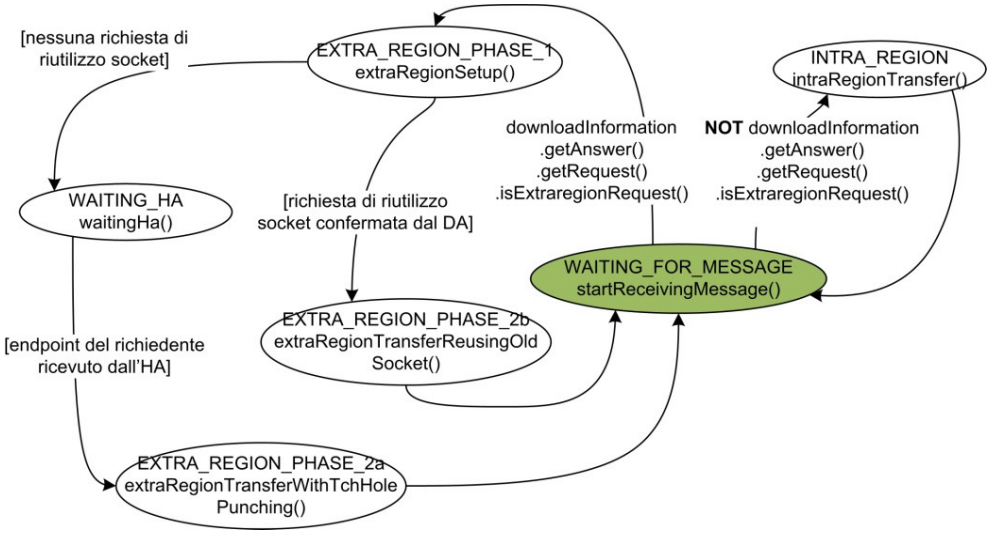


Figura 5-26 – Behaviour UASendTrancheToDABhv: FSM.

- EXTRA_REGION_PHASE_2b

  In questo stato avviene l'upload della tranche dal peer fonte attraverso una connessione instaurata in precedenza e lasciata attiva proprio per questo scopo.

Il behaviour, quando perviene la primitiva freezeP, stopP, o downP si comporta in modo simile a quanto fa il behaviour dell'agente DA nei confronti dei DADownloaderTrancheBhv: invoca il metodo `close()` su tutti gli oggetti `Socket` utilizzati per i trasferimenti precedenti e memorizzati nell'apposita struttura dati, chiudendo così le connessioni verso gli agenti DA dei peer downloader contattati in precedenza; ovviamente non interrompe alcun behaviour poiché l'upload avviene direttamente ad opera del UASendTrancheToDABhv.

Per ulteriori dettagli sull'interazione con gli agenti UA e HA si veda il paragrafo *5.4*.

5.2.10 Altri componenti

I prossimi componenti che vengono presentati sono stati modificati in minima parte e quindi si ritiene sufficiente riportare soltanto alcune informazioni rilevanti, come le nuove funzioni svolte da ciascuno di essi o le operazioni eseguite in concomitanza dell'arrivo di una primitiva di amministrazione.

Agente RRA (ResourceRegionAgent) [modified]

L'agente RRA del superpeer, che è responsabile delle ricerche intra-regione ed extra-regione, nel nuovo sistema è in grado di nascondere-rimuovere-ripristinare dalla tabella `tranche_owner` le informazioni relative alle tranche possedute dai peer della regione quando vengono eseguite le primitive freezeP , removeP, thawP. Tali richieste gli vengono inviate dall'agente PMA del superpeer (precisamente dal behaviour PMAForwardAdminPrimitiveToPeerBhv) durante l'inoltro delle primtive al peer della regione.

In particolare il suo behaviour RPAUpdateSourceForTranche ora è diventato sensibile a messaggi dei seguenti tipi:

ConversationID	**Azione**
UPDATE_DB_SUPERPEER _NEW_SOURCE _FOR_TRANCHE	Il peer ha appena completato il download di una tranche e può essere considerato una fonte per essa.
UPDATE_DB_SUPERPEER _REMOVE_SOURCE _FOR_TRANCHE	Il peer è stato rimosso dalla regione. Le informazioni sulle tranche da esso possedute vengono rimosse.
UPDATE_DB_SUPERPEER _HIDE_SOURCE _FOR_TRANCHE	Il peer è nello stato pFROZEN. Le informazioni vengono nascoste ma non rimosse.
UPDATE_DB_SUPERPEER _SHOW_SOURCE _FOR_TRANCHE	Il peer è tornato nello stato pACTIVE dopo essere stato in pFROZEN. Le informazioni vengono rese nuovamente disponibili.
UPDATE_DB_SUPERPEER _HIDE_ALL_REGION_SOURCE _FOR_TRANCHE	Vengono nascoste le informazioni sulle tranche possedute da tutti i peer della regione.

Figura 5-27 – RRAUpdateSourceForTrancheBhv: ConversationId.

L'ultimo tipo di azione viene utilizzata insieme alla penultima in modo da garantire che, nel momento in cui una regione viene aggiunta al sistema dopo essere stata precedentemente rimossa, le tranche ricevute dal

superpeer possano essere rese disponibili per tutti i peer della regione: in questo modo le nuove richieste dei peer della regione dopo il ri-avvio rimarranno all'interno della regione stessa e non genereranno ricerche extra-regione.

Agente CA (CreatorAgent) [modified]

L'agente CA del superpeer gestisce le rcerche extra-regione e, quindi, i rapporti con i SearchAgent creati da altri superpeer.

Viene raggiunto dalla primitiva overlayNetworkUpdate che viene inviata dal server (precisamente dal PMAForwardAdminPrimitive ToSuperpeerBhv) al superpeer (PMAReceiveAdminPrimitive AsSuperpeerBhv) e dal superpeer all'agente CA.

In questo modo il CA è in grado di aggiornare i riferimenti ai superpeer vicini (clockwise a anticlockwise) quando avvengono delle modifiche alla regione stessa o ai superpeer delle regioni adiacenti.

Agente UMA (UploaderManagementAgent) [modified]

L'agente UMA ha subito due aggiornamenti di lieve entità relativamente al suo behaviour UMACallForTicketBhv.

Innanzitutto è stato aggiunto timeout per scongiurare l'eventuale ipotesi di un'attesa indefinita del messaggio di sblocco `COMPLETED_DOWNLOAD` inviato dall'agente DA al termine di un download. Questa possibilità è legata al fatto che si possono verificare errori di comunicazione oppure dei problemi relativamente all'invio o al recapito dei messaggi ACL.

In secondo luogo si è stabilito che il behaviour, nel momento in cui giunge la primitiva freezeP o la primitiva stopP, esegua la chiamata di tutti i ticket[23] rilasciati agli agenti MA che lo hanno contattato fino a quel momento.

[23] I ticket che vengono chiamati appartengono ad entrambe le code: quella gestita da un `highPriorityTicketManager` ad alta priorità per le richieste urgenti e quella gestita e da un `normalPriorityTicketManager` per le richieste a priorità normale.

Agenti RSMA, RQMA, SA, DMA, MA, SMA e SearchAgent

Per tutti i rimanenti agenti del sistema, ad eccezione del SearchAgent e del MA, si sono seguite le direttive generiche trattate nel paragrafo *5.2.1* ed in particolare nei *5.2.1.7* e *5.2.1.9*.

Quindi la struttura esistente di ogni behaviour è stata inglobata all'interno di quella generica richiesta per supportare le primitive di amministrazione; è stato aggiunto, ad ogni behaviour, un metodo `afterThawP()` che esegue operazioni specifiche per il behaviour che viene raggiunto da una primitiva thawP; per ogni agente è stato aggiunto un behaviour "coordinatore" che verifica l'arrivo di nuove primitive (tramite il controllo della variabile `targetPeerState` della classe principale dell'agente), che verifica l'esecuzione delle primitive da parte di tutti gli altri behaviour dell'agente e che interagisce con l'agente PMA del peer.

Gli agenti MA e SearchAgent migrano da una piattaforma all'altra e la loro esecuzione viene gestita da altri agenti statici presenti nella piattaforma in cui gli MA si vengono a trovare: sono questi agenti ad essere direttamente coinvolti nel processo di gestione delle primitive di amministrazione; gli agenti MA vengono controllati dall'agente UMA, mentre i SearchAgent dall'agente CA.

5.3 Primitive

La descrizione del sistema effettuata fino a questo momento ha privilegiato un approccio basato sulla descrizione dei singoli componenti e, per quanto riguarda gli agenti JADE, anche dei numerosi behaviour.

Le interazioni fra gli agenti e, nello specifico, fra i loro behaviour rappresentano un elemento sostanziale del nuovo sistema; alcune di esse sono state descritte in modo accurato mentre altre, richiedendo un'analisi più attenta, sono state presentate più sommariamente.

Lo scopo di questo paragrafo e di quello successivo è quello di scendere più in profondità e di analizzare in modo trasversale (rispetto ai vari componenti):

- le primitive di amministrazione;
- i protocolli per il trasferimento inter-regione delle tranche (paragrafo *5.4*).

Ci si avvarrà di alcuni diagrammi che sono stati appositamente prodotti con lo scopo di rendere più agevole la comprensione; per non appesantire troppo la trattazione, inoltre, molti concetti verranno presentati in riferimento ad alcuni esempi concreti, in modo tale da dare un'idea del comportamento del nuovo sistema.

5.3.1 Avvio del sistema

La procedura di avvio del sistema ha inizio dal superpeer a cui viene assegnato il ruolo di server.

Su tale server viene avviato il MainBoot (che inzializza l'agente PMA, il container ed il main container JADE) e viene creata la struttura dati con cui viene gestita l'overlay network: in queste fasi iniziali l'overlay newtork è composta unicamente dal server.

Tutte le altre regioni devono conoscere a priori l'indirizzo MTP pubblico del container localizzato sul server; prima di poter essere aggiunte al sistema devono informare della propria presenza l'agente PMA del server, comunicandogli l'indirizzo pubblico dei propri agenti AMS (indirizzo MTP pubblico del main container) e PMA (indirizzo MTP pubblico del container).

Il server aggiungerà le nuove regioni una alla volta, per evitare di incorrere in problemi di inconsistenza[24].

5.3.2 Primitive di stato per le regioni

In questo paragrafo, con l'aiuto di qualche esempio e di alcune immagini, sono descritte le principali interazioni che avvengono fra gli agenti del sistema durante l'esecuzione delle primitive:

- addR;
- removeR;
- downR.

[24] In questo modo, peraltro, si evita di dover effettuare la gestione della concorrenza per l'accesso in scrittura, da parte dei behaviour dell'agente PMA del server, alla struttura dati che gestisce l'overlay network.

5.3.2.1 addR

Per la descrizione della primitiva addR si fa riferimento alla *Figura 5-28* (p.183) che rappresenta il seguente scenario: il sistema è composto da quattro regioni e la REGION5 sta per essere aggiunta ad opera del superpeer della REGION2. Gli agenti rappresentati sono:

1. il PMA del server;
2. il PMA del superpeer creatore della primitiva;
3. il PMA del futuro vicino "sinistro" della nuova regione;
4. il PMA del futuro vicino "destro" della nuova regione;
5. il thread MainBoot del superpeer della REGION5 che verrà aggiunta al sistema;
6. il PMA del superpeer della REGION5;
7. gli agenti "tipo superpeer" (CA, RPA, RRA) ;
8. gli agenti "tipo peer" (RSMA, DMA, UA, SA, etc..).

La descrizione delle comunicazioni che intercorrono fra i vari componenti viene suddivisa in quattro fasi.

FASE 1. La prima operazione [1] necessaria per l'esecuzione della primitiva è che il MainBoot della REGION5 venga avviato: viene inizializzato JADE (container e main container) e viene avviato l'agente PMA che invia [2] una primitiva overlayNetworkUpdate al server con cui lo informa della presenza.

FASE 2. Attraverso la GUI del superpeer della REGION2 viene creata la primitiva addR per la REGION5 ed il PMA la inoltra [3] al server.

Il server richiede [4] al PMA del superpeer della REGION5 l'indirizzo pubblico del suo MainBoot ed il PMA risponde [5] comunicando la coppia (ip:porta).(nel caso il PMA non dovesse rispondere entro un tempo limite, il server informerà dell'errore il creatore della primitiva).

Successivamente il server richiede [6] al MainBoot del superpeer della REGION5 di avviare gli agenti di tipo superpeer [7] e di tipo peer [8]; dopo la conferma del MainBoot [9], il server invia la primitiva addR al superpeer della REGION5.

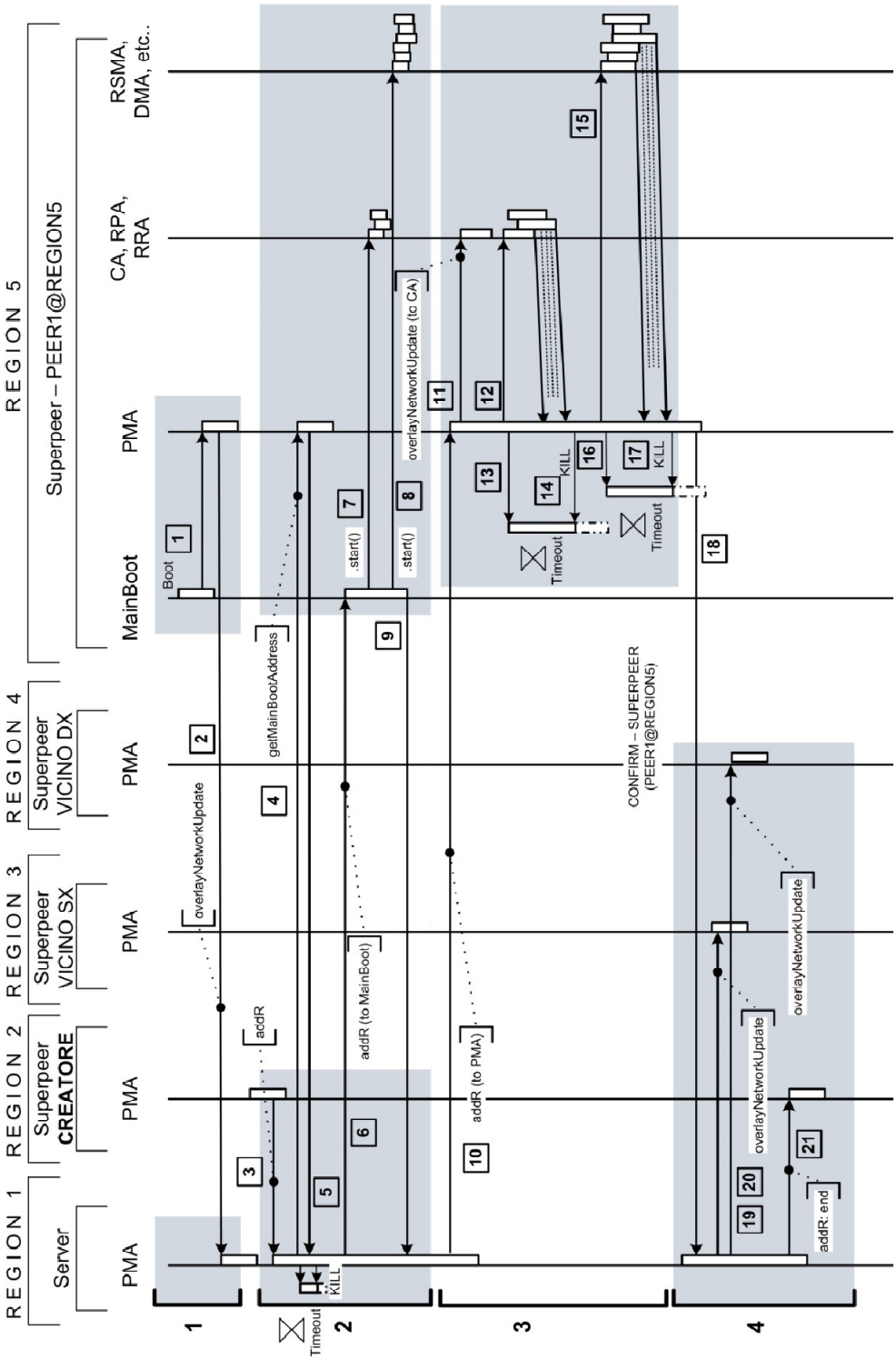


Figura 5-28 – Primitiva addR.

FASE 3. Quando il PMA del superpeer della REGION5 riceve la primitiva (attraverso il PMAReceive...AsSuperpeerBhv), comunica [11] i nuovi riferimenti ai futuri vicini all'agente CA e procede ad inoltrare [12] la primitiva addR a tutti gli agenti "tipo superpeer" presenti sul superpeer stesso.
Se l'ultimo dei suddetti agenti conferma l'esecuzione della primitiva addR entro il timeout avviato in precedenza [13], il PMA del superpeer interrompe il timeout [14] e prosegue; altrimenti informa il server che provvederà ad avvisare il creatore della condizione di FAILURE.
Successivamente il PMA del superpeer (attraverso il suo PMAForward..ToPeerBhv) lancia [15] la primitiva startP verso tutti gli agenti "tipo peer", avvalendosi sempre di un timeout [16].
Quando tutti gli agenti hanno confermato, interrompe il timeout [17] ed informa il server.

FASE 4. Quando gli giunge la conferma del PMA del superpeer [18], il server invia ai due superpeer delle regioni fra cui è stato inserita la REGION5 due primitive overlayNetworkUpdate [19] e [20].
Infine il server informa il creatore della primitiva (il superpeer della REGION2) dell'esito della primitiva stessa [21].

Il creatore della primitiva, in realtà, viene tenuto frequentemente aggiornato dal server durante lo svolgersi delle varie fasi: ad esempio, come è possibile vedere nella *Figura 5-29* (p.186), appena riceve la primitiva dal creatore, il server gli risponde subito [3 bis] confermando la ricezione della primitiva stessa o informandolo del verificarsi di un errore; tale errore potrebbe essere legato al fatto che non è pervenuta ancora nessuna informazione sulla regione che si sta tentando di aggiungere oppure potrebbe riguardare l'impossibilità di aggiungere la regione richiesta poiché è già in corso l'aggiunta/rimozione di un'altra regione.

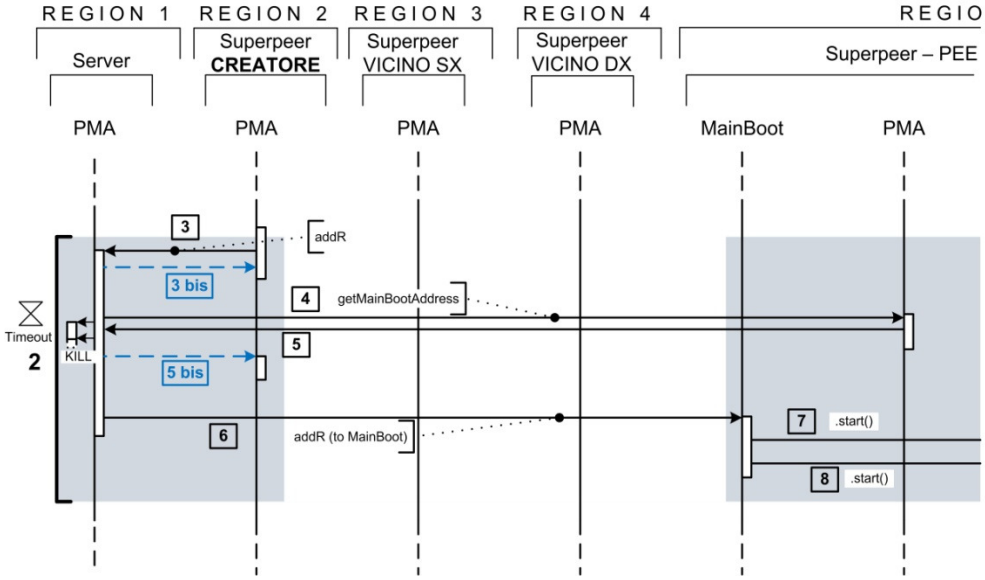


Figura 5-29 – Primitiva addR: dettaglio.

Sempre nella *Figura 5-29* si può vedere come il server confermi al creatore [5 bis] la possibilità di aggiungere la REGION5: nel caso in cui il PMA non risponda entro il tempo previsto, il creatore verrà informato della non raggiungibilità del superpeer della regione che si sta tentando di aggiungere.

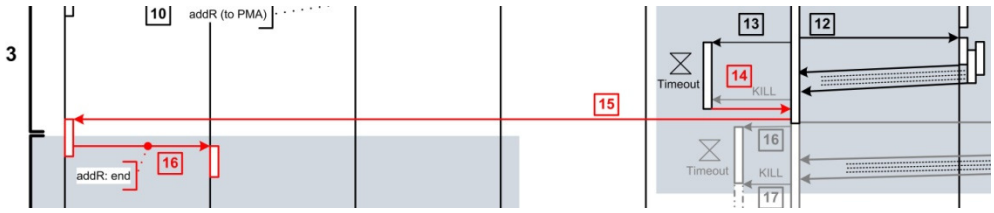


Figura 5-30 – Primitiva addR: gestione di un errore (esempio).

Nella *Figura 5-30* (p.186), invece, si mostra la reazione dei componenti al verificarsi di un errore durante l'ultima parte della FASE 3 della *Figura 5-28* (p.183). Nello specifico il PMA del superpeer si accorge [14] che non tutti gli agenti di "tipo superpeer" hanno confermato l'esecuzione della primitiva entro il timeout previsto[25]; quindi informa [15] il server che tralascia le modifiche apportate alla overlay network, rilascia la struttura

[25] Più in dettaglio accade che al PMAReceive...AsSuperpeerBhv perviene un messaggio di tipo SUPERPEER_AGENT_STATE _UPDATE_TIMEOUT dal behaviour PMACheckStateTimeoutBhv, che lo informa dello scadere del tempo prestabilito.

dati che ne gestisce la composizione (permettendo, così, l'aggiunta/rimozione di altre regioni) e notifica [16] al creatore che si è verificato un errore.

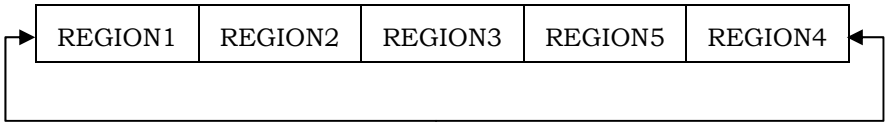
5.3.2.2 removeR

Le interazioni previste dalla primitiva removeR sono descritte nella *Figura 5-31* (p.189). I componenti rappresentati sono:

1. il PMA del server;
2. il PMA del superpeer creatore della primitiva;
3. il PMA del futuro vicino "sinistro" della nuova regione;
4. il PMA del futuro vicino "destro" della nuova regione;
5. il thread MainBoot del superpeer della REGION5 che verrà aggiunta al sistema;
6. il PMA del superpeer della REGION5;
7. il thread MainBoot del PEER2@REGION5 (nella regione da rimuovere);
8. l'agente PMA del PEER.

Lo scenario è molto simile a quello presentato per la primitiva addR; l'unica differenza, oltre alla natura della primitiva da eseguire, consiste nella presenza di un PEER (il PEER2@REGION5) oltre al superpeer (PEER1@REGION5) nella regione da rimuovere.

La struttura dell'anello composto dai superpeer è la seguente:



La descrizione delle comunicazioni che intercorrono fra i vari componenti viene suddivisa in cinque fasi.

FASE 1. Nel momento in cui la primitiva "removeR REGION5" viene creata attraverso la GUI del superpeer della REGION2 e inviata [1] dal suo PMA al server, la REGION5 da rimuovere è connessa al sistema ed il suo unico peer oltre al supereer è nel pieno delle sue attività.

Analogamente a quanto detto per la primitiva addR del peragrafo precedente, il server richiede [2] al PMA del superpeer della REGION5 l'endpoint pubblico del suo thread MainBoot inviandogli la primitiva getMainBootAddress. Se il

PMA risponde alla richiesta entro il tempo prestabilito [3], il server può proseguire; altrimenti informa dell'errore il superpeer creatore.

FASE 2. Il server informa subito [4] e [5] i PMA dei superpeer delle attuali regioni adiacenti a quella da rimuovere attraverso delle primitive overlayNetworkUpdate: in questo modo viene creato immediatamente un "bypass" che estromette la REGION5 dai percorsi dei SearchAgent delle nuove ricerche extra-regione lungo l'anello dei superpeer[26].

FASE 3. Il PMA del server inoltra [6] la primitiva removeR al PMA del superpeer della REGION5 che prima informa il proprio agente RRA[27] e che poi, attraverso il proprio behaviour PMAForward..ToPeerBhv, invia [7] la primitiva stopP all'unico peer della regione (oltre al superpeer)[28]. Dopo aver inizializzato [8] il timeout (basato sul parametro `Parameters.MAXIMUM_PEER_STATE_CHECK_PERIOD`) attende che il PMA di ogni peer della regione (in questo caso uno solo) esegua la primitiva stopP (attende al massimo un periodo di durata pari a `Parameters.MAXIMUM_AGENT_STATE_CHECK_PERIOD`). Quando tutti gli agenti del peer hanno confermato [9] la propria disponibilità al cambiamento di stato in pSTOPPED, il PMA del peer informa [10] il PMA del superpeer il quale contatta il MainBoot del peer e gli chiede [11] di eseguire il metodo kill() su tutti gli oggetti AgentController del peer stesso (ad esclusione del PMA).
Quando il MainBoot del peer ha confermato [12], il PMA del superpeer informa il server [13] che invia una notifica al creatore [14].

[26] Cfr. paragrafo 3.6.12.

[27] Il messaggio di pre-stop informa l'agente RRA dell'imminente rimozione della regione; in questo modo l'RRA non inserirà più alcuna fonte locale nella lista delle fonti extra-regione richiesta dai SearchAgent in transito.

[28] Nel caso siano presenti più peer, viene inviata la primitiva stopP ad ognuno di essi.

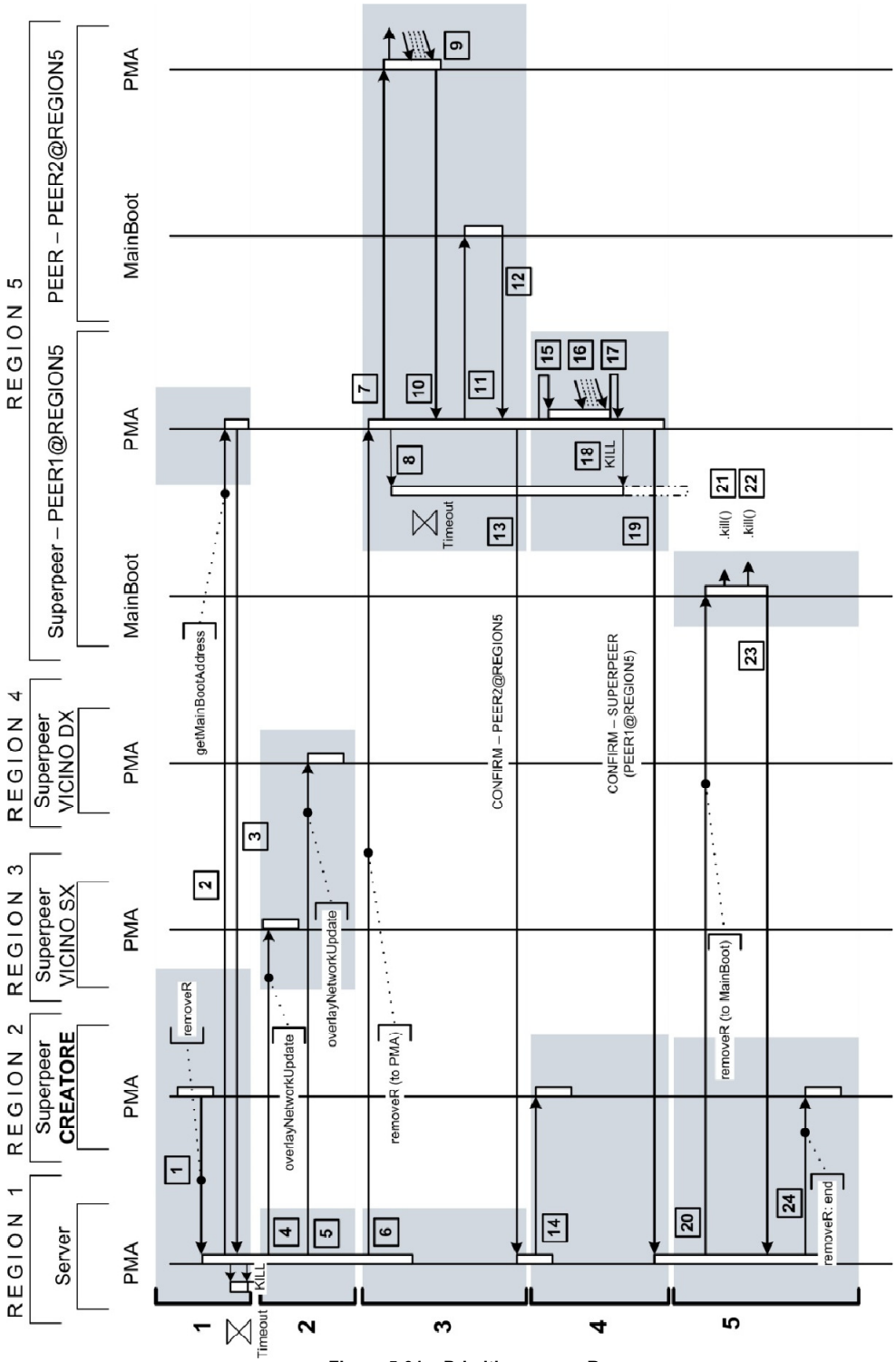


Figura 5-31 – Primitive removeR.

FASE 4. Successivamente il PMA del superpeer si occupa di interrompere ([15] e [16]) prima il peer locale (agenti di "tipo peer" presenti sul superpeer) e poi tutti gli agenti di "tipo superpeer" (RRA, RPA e CA), preoccupandosi di rimuovere dal database tutte le informazioni sulle tranche possedute dai peer della regione[29]. Quando tutti gli agenti hanno dato conferma [17], interrope il timeout [18] ed informa il PMA del server [19].

FASE 5. Il PMA del server, a questo punto, chiede [20] al MainBoot del superpeer della REGION5 di rimuovere gli agenti "tipo peer" [21] e "tipo superpeer" [22] del superpeer stesso tramite il metodo kill() invocato sugli oggetti AgentController. Quando Il MainBoot dà conferma [23], il PMA del server invia [24] l'esito finale al creatore.

5.3.2.3 downR

La primitiva downR, concettualmente simile a removeR, viene invocata su una regione nello stato rFAILURE, in cui uno o più nodi sono nello stato pFAILURE in seguito ad un errore verificatosi durante l'esecuzione di una primitiva addR o removeR.

Questa primitiva richiede principalmente:

- che vengano avvisati i superpeer delle regioni adiacenti a quella da rimuovere affinchè estromettano dall'anello dei superpeer la regione che si trova nello stato di errore;
- che gli agenti PMA del superpeer e di tutti i peer della regione tornino in uno stato "stabile" per poter ricevere ed eseguire altre primitive: questo stato è pSTOPPED;
- che gli agenti di "tipo superpeer" presenti sul superpeer della regione in rFAILURE rilascino le risorse possdute (in particolare l'agente RPA che mantiene attive le connessioni necessarie alla procedura di TCP Hole Punching);
- che gli agenti di ogni peer rilascino le risorse possedute (in particolare gli agenti UA e DA che mantengono attive le

[29] Tale operazione viene eseguita dall'agente RRA.

connessioni verso altri peer utilizzate per il trasferimento extra-regione delle tranche);

- che tutte le informazioni sulle tranche possedute dai peer della regione vengano rimosse dal database, ad esclusione di quelle associate alle tranche possedute dal superpeer.

Per questi motivi la primitiva downR raggiunge gli agenti PMA del superpeer e di tutti i peer della regione nello stato rFAILURE e viene inoltrata a quegli agenti che devono rilasciare delle risorse prima di essere rimossi dal MainBoot

Infine il MainBoot del superpeer e quello di ogni peer rimuove tutti gli agenti del nodo eccetto il PMA, riportando ogni nodo nella stessa situazione in cui si trova quando viene avviato per la prima volta.

5.3.3 Primitive di stato per i peer

Tutta la nuova architettura utilizzata per la gestione delle primitive di stato dei peer (e delle regioni) è stata descritta molto accuratamente nel corso del paragrafo *5.2.3*, in cui sono stati analizzati nel dettaglio tutti i behaviour dell'agente PMA.

Inoltre, sempre nel paragrafo *5.2*, si è cercato di descrivere le operazioni più rilevanti che compiono i singoli agenti durante il loro normale funzionamento e alla ricezione di una primitiva.

Lo scopo di questo paragrafo è quello di fornire un quadro complessivo di ogni primitiva, illustrandone il funzionamento a livello degli agenti coinvolti.

Per quanto riguarda le primitive freezeP e thawP verrà fatto un discorso più approfondito, in quanto coinvolgono l'agente AMS e si appoggiano ai servizi messi a disposizione dall'add-on Persistence.

5.3.3.1 startP

La primitiva rende operativi tutti gli agenti di un peer e può essere originata:

- dal superpeer della regione in cui è presente il peer da avviare;
- da qualsiasi altro superpeer del sistema (via server);
- dal server;

- da qualsiasi peer della regione in cui si trova il peer target;
- dallo stesso peer target[30].

In ogni caso l'esecutore principale di tale primitiva deve essere il superpeer della regione in cui si trova il peer target, dal momento che è necessaria l'interazione diretta (via TCP) col MainBoot del peer.

Per l'aggiunta di un peer ad una regione valgono considerazioni simili a quelle proposte per la primitiva addR: per poter essere avviato un peer deve aver prima dato notizia di sé (tramite un messaggio `PEER_STATE_UPDATE`) al superpeer della regione all'interno della quale sarà aggiunto.

Il superpeer[31] esegue le seguenti operazioni nell'ordine indicato:

1. controlla che il peer abbia dato comunicazione della propria presenza con un `PEER_STATE_UPDATE`;
2. invia al peer (al suo PMAReceive..AsPeerBhv) una primitiva getMainBootAddress per ottenere l'endpoint del suo MainBoot e per verificarne la raggiungibilità;
3. chiede al MainBoot di avviare gli agenti del peer;
4. invia al PMA del peer la primitiva startP affinchè la inoltri verso tutti gli altri agenti;
5. attende dal peer messaggi di tipo:
   a. `INFORM`: notifica del cambiamento di stato di un agente del peer);
   b. `CONFIRM`: notifica del cambiamento di stato del peer - di tutti gli agenti -;
   c. `FAILURE`: notifica del verificarsi di un errore durante le fasi di avvio di uno o più agenti).
6. aggiorna la struttura dati[32] che mantiene le informazioni sullo stato di tutti i peer della regione;

[30] L'esecuzione di startP da un peer di altre regioni differenti da quella in cui è situato il peer target non è stata prevista a causa del ruolo ricoperto dai singoli peer all'interno del sistema; tuttavia tale funzionalità non nasconde particolari problematiche nel caso la si voglia implementare in un secondo momento.

[31] Attraverso il suo behaviour PMAForward..ToPeerBhv.

[32] Costituita da un oggetto StateVector (paragrafo 5.2.1.2) composto da EntityState che individuano lo stato di ogni peer.

7. informa il creatore tramite:
    a. messaggi di aggiornamento[33], per ogni agente del peer che conferma l'esecuzione della primitiva richiesta;
    b. messaggi di stato[34], che comunicano l'esito (positivo o negativo) della primitiva nel suo complesso.

5.3.3.2 suspendP

La primitiva può essere inizializzata dalla stessa tipologia di nodi elencati per la primitiva startP.

La natura della primitiva non prevede che venga interrotta l'esecuzione di tutti gli agenti del peer, ma che ne vengano soltanto sospese le attività; di conseguenza non è necessaria l'interazione diretta (via TCP) col MainBoot del peer.

Ogni agente rimane in esecuzione e viene raggiunto dai messaggi ACL degli altri agenti o degli altri peer; tali messaggi vengono archiviati nelle code appositamente predisposte e verranno processati quando l'agente tornerà nel pieno delle sue attività con la primitiva resumeP.

Gli agenti UA e DA, che possiedono oggetti Socket associati a connessioni attive[35], non chiudono le connessioni instaurate con gli altri peer: le comunicazioni attraverso tali Socket vengono contrattate preventivamente tramite messaggi ACL e, dal momento che i messaggi non vengono processati nello stato pSUSPENDED, si evita che altri peer rimangano bloccati in attesa di queste comunicazioni.

Le operazioni eseguite dal PMA del superpeer della regione a cui appartiene il peer target sono le seguenti:

1. controllo della presenza del peer target nella regione (agente PMA in esecuzione);

[33] Primitive con l'attributo end settato a "false" e con il campo answer che contiene un oggetto EntityState associato ad un agente del peer target che sta eseguendo la primitiva (es. RSMA_PEER2@REGION5$$0 per informare che che l'agente RSMA del PEER1 della REGION5 ha confermato la propria disponibilità a passare nello stato pACTIVE = 0).

[34] Oggetti Primitive con l'attributo end = true.

[35] Parametro KEEP_ALIVE settato a true.

2. invio al peer (al suo PMAReceive..AsPeerBhv) di una primitiva getMainBootAddress per ottenere l'endpoint del suo MainBoot ma, soprattutto per controllarne la raggiungibilità;
3. inoltro al PMA del peer della primitiva suspendP affinchè la inoltri verso tutti gli altri agenti;
4. attesa dal peer di messaggi di tipo:
   a. `INFORM`: notifica del cambiamento di stato di un agente del peer;
   b. `CONFIRM`: notifica del cambiamento di stato del peer – di tutti gli agenti -;
   c. `FAILURE`: notifica del verificarsi di un errore durante le fasi di avvio di uno o più agenti.
5. aggiornamento della struttura dati[36] che mantiene le informazioni sullo stato di tutti i peer della regione;
6. notifica al creatore tramite:
   a. messaggi di aggiornamento[37], per ogni agente del peer che conferma l'esecuzione della primitiva richiesta;
   b. messaggi di stato[38], che comunicano l'esito (positivo o negativo) della primitiva nel suo complesso

5.3.3.3 resumeP

La primitiva resumeP riporta un peer che si trova nello stato pSUSPENDED nello stato pACTIVE ed esegue le stesse operazioni previste per la primitiva suspendP ad eccezione del passo numero 3 che ovviamente prevede, in questo caso, l'invio della primitiva resumeP.

Ogni agente controlla periodicamente l'arrivo di questa primitiva, pronto a riprendere le sue attività processando i messaggi ACL rimasti in coda.

---

[36] Costituita da un oggetto StateVector (paragrafo 5.2.1.2) composto da EntityState che individuano lo stato di ogni peer.

[37] Primitive con l'attributo end settato a "false" e con il campo answer che contiene un oggetto EntityState associato ad un agente del peer target che sta eseguendo la primitiva (es. RSMA_PEER2@REGION5$$0 per informare che l'agente RSMA del PEER1 della REGION5 ha confermato la propria disponibilità a passare nello stato pACTIVE = 0).

[38] Oggetti Primitive con l'attributo end = true.

5.3.3.4 freezeP

La primitiva interrompe l'esecuzione di tutti gli agenti del peer (eccetto il PMA) dopo averne salvato gli stati all'interno del database (utilizzato dall'add-on Persistence).

Lo stato di un agente è costituito:

- dal valore di tutte le variabili oggetto usate da ogni suo behaviour;
- dal contenuto delle code dei messaggi in ingresso e in uscita.

Dal momento che non tutte le variabili oggetto utilizzate da vari behaviour sono associate a classi serializzabili, alcune di esse sono state contrassegnate con la keyword "`transient`", ad indicare che non devono essere salvate; il valore di tali variabili, quindi, quando verrà ricevuta la primitiva thawP dovrà essere configurato a dovere[39].

Le interazioni previste dalla primitiva freezeP sono descritte nella *Figura 5-32* (p. 196). I componenti rappresentati sono:

1. l'add-on Persistence;
2. l'agente AMS del superpeer;
3. l'agente PMA del superpeer;
4. il thread MainBoot del peer da ibernare;
5. il PMA del peer;
6. gli agenti "tipo peer" (RSMA, DMA, UA, SA, etc..).

Lo scenario vede la creazione delle primitiva freezeP ad opera del peer target stesso (PEER2@REGION5); l'esecuzione della primitiva non è interrotta da alcun errore.

La descrizione delle comunicazioni che intercorrono fra i vari componenti viene suddivisa in quattro fasi:

FASE 1. La primitiva viene creata dall'interfaccia grafica dello stesso peer target (PEER2@REGION5); il relativo PMA la inoltra [1] al PMA del superpeer.
Il PMA del superpeer richiede [2] al PMA del peer target l'endpoint del MainBoot ed il PMA dal peer glielo comunica [3] confermando la propria raggiungibilità.

[39] Il compito appena descritto è responsabilità del metodo `afterThawP()` di ogni behaviour.

Successivamente il PMA del superpeer nasconde [4], all'interno del database - tabella tranche_owner -, le informazioni sulle tranche posseduta dal peer target.

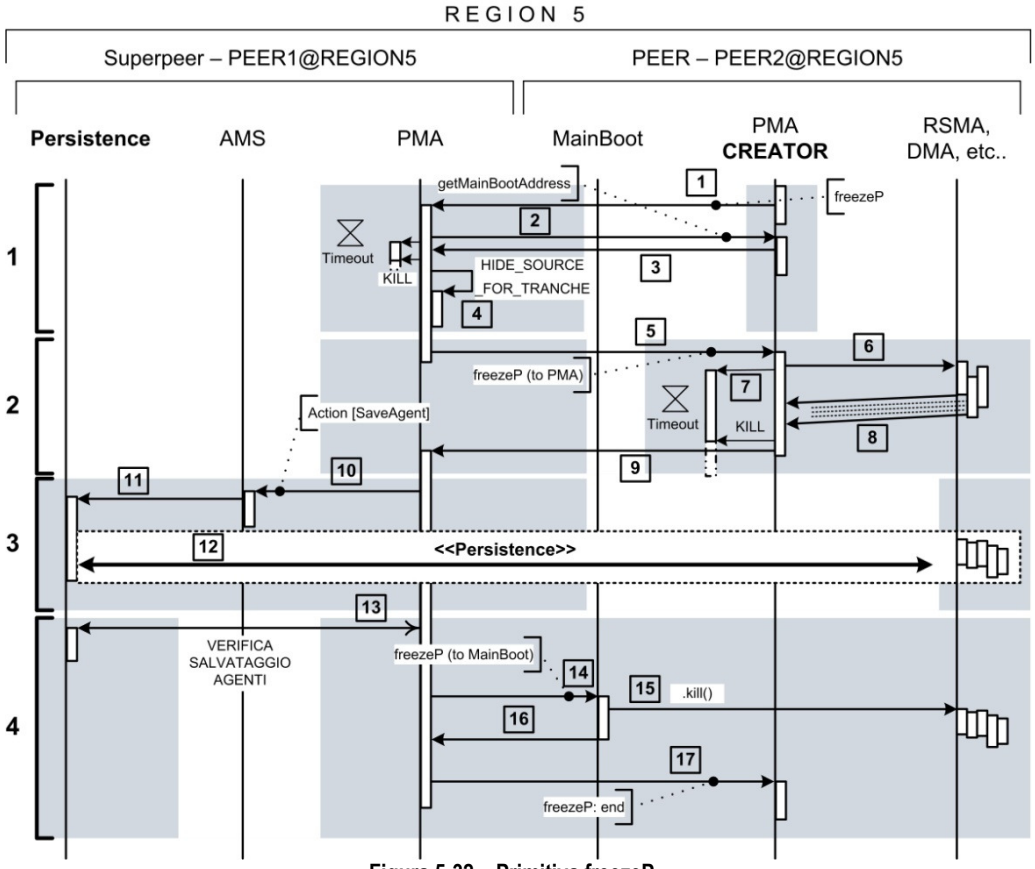


Figura 5-32 – Primitiva freezeP.

FASE 2. Il PMA del superpeer inoltra [5] la primitiva frese al PMA del peer, il quale la inoltra a sua volta [6] a tutti gli altri agenti, si mette in attesa delle risposte e controlla il processo con un timeout [7] di di durata pari a (`Parameters.MAXIMUM_AGENT_STATE_CHECK_PERIOD`); quanto tutti gli agenti hanno confermato[40] [8] informa il PMA del superpeer [9].

[40] Il PMA del peer inoltra al PMA del superpeer i seguenti tipi di messaggi PEER_STATE_UPDATE#PEER2@REGION5):

a. INFORM: cambiamento di stato di un agente del peer;
b. CONFIRM: cambiamento di stato del peer (di tutti gli agenti);
c. FAILURE: errore durante le fasi di avvio di uno o più agenti.

Il superpeer li inoltra tempestivamente al creatore della primitiva aggiornando adeguatamente le strutture dati locali.

FASE 3. Successivamente il PMA del superpeer procede al salvataggio individuale di ogni singolo agente del peer: per ognuno di essi invia [10] all'agente AMS un messaggio ACL contenente un Action che descrive l'azione che l'AMS deve compiere per salvare lo stato dell'agente[41]. L'agente AMS interagisce [11] con i servizi offerti dall'add-on Persistence il quale procede [12] al salvataggio degli agenti del peer target.

FASE 4. Dopo aver verificato [13] il corretto salvataggio di tutti gli agenti del peer, il PMA del superpeer chiede ([14],[15]e[16]) al MainBoot, secondo le consuete modalità, di interrompere l'esecuzione degli agenti del peer e, successivamente, ne dà conferma [17] al creatore.

Si sottolinea che la primitiva è stata progettata in modo che il salvataggio degli agenti avvenisse ad opera dell'agente AMS; questa scelta ha reso il sistema molto versatile in quanto, nel caso lo si ritenesse opportuno, sarebbe possibile far congelare e scongelare un qualsiasi agente da un punto qualunque del sistema.

Per i frammenti di codice che definiscono l'interazione con l'AMS si veda la *Figura 5-19*(p.169).

5.3.3.5 thawP

La primitiva thawP ripristina lo stato di tutti gli agenti di un peer che si trova nello stato pFROZEN.

Le interazioni previste dalla primitiva thawP sono descritte nella *Figura 5-33* (p. 199). I componenti rappresentati sono:

1. l'add-on Persistence;
2. l'agente AMS del superpeer;
3. l'agente PMA del superpeer;
4. il thread MainBoot del peer da ibernare;
5. il PMA del peer;
6. gli agenti "tipo peer" (RSMA, DMA, UA, SA, etc..).

[41] Per i frammenti di codice che definiscono l'interazione con l'AMS si veda la *Figura 5-19* (p.158).

Lo scenario, in questo caso, vede la creazione della primitva thawP da parte del PMA del superpeer.

La descrizione delle comunicazioni che intercorrono fra i vari componenti viene suddivisa in quattro fasi:

FASE 1. La primitiva viene inserita[42] attraverso la GUI del PMA del superpeer [1]. Successivamente il PMA del superpeer chiede [2] al PMA del peer l'indirizzo [3] del suo MainBoot con la primitiva getMainBootAddress.

FASE 2. Il PMA del superpeer, analogamente a quando fatto per la primitiva freezeP, invia [4] all'AMS una richiesta che lo invita ad interagire [5] con i servizi di Persistence al fine di [6]:

a. creare gli agenti del peer target all'interno del container specifico del peer target;
b. caricare uno alla volta lo stato di tutti gli agenti del peer avviandone l'esecuzione;
c. eseguire il metodo afterLoad()[43] di ogni agente in sostituzione del metodo setup() eseguito in condizioni di avvio classico;
d. portare nello stato (JADE) ACTIVE ogni agente caricato. (N.B. non si tratta dello stato pACTIVE del peer)

N O T A I M P O R T A N T E : a partire da questo momento tutti gli agenti del peer sono in esecuzione ma non stanno svolgendo le normali attività: appena dopo l'avvio ad opera di Persistence i behaviour di supporto ####ReceiveAdminPrimitiveBhv[44] di tutti gli agenti sono nel loro stato bFROZEN in attesa della primitiva thawP e tutti gli altri behaviour controllano periodicamente la variabile

[42] Nel dettaglio dalla GUI (componente SimpleAdminGUI) raggiunge il PMASimpleUserInterfaceBhv che la inoltra al PMAReceive..AsSuperpeerBhv dello stesso agente. Tale behaviour aggiunge un PMAForward..ToPeerBhv come threadedBehaviour il quale inoltra [1] la primitiva al PMA del peer. Per maggiori dettagli sul funzionamento degli agenti PMA si faccia riferimento al paragrafo 5.2.3.

[43] Si ricorda che il metodo afterLoad() invocato da Persistence per ogni agente si occupa di inizializzare le variabili transient della classe principale dell'agente stesso e di eseguire il metodo `afterThawP()` di OGNI behaviour dell'agente, in modo che vengano inizializzate anche le variabili transient esistenti localmente per ognuno di essi.

[44] cfr. paragrafo *5.2.1.7*.

targetPeerState in attesa di vederle assumere il valore pACTIVE.

FASE 3. A questo punto tutti gli agenti del peer sono in esecuzione e l'agente PMA del superpeer inoltra [7] la primitiva thawP al PMA del peer target che, come accade per le altre primitive e come è stato descritto –ad esempio- per freezeP, procede nelle comunicazioni col PMA del peer [8], [9] e [10] informando il fine il PMA creatore (in questo caso coincide col PMA del superpeer, quindi i messaggi `GUI_INTERPLAY` raggiungono proprio la GUI di questo nodo).

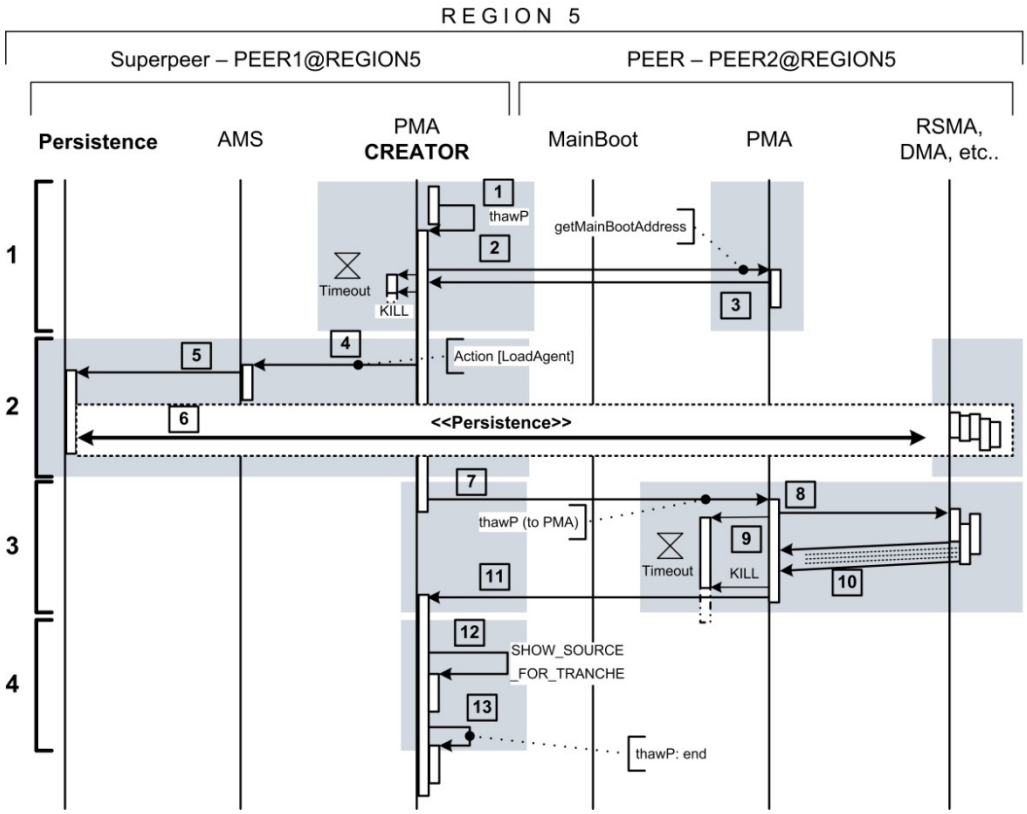


Figura 5-33 – Primitiva thawP.

FASE 4. Nell'ultima fase il PMA del superpeer ripristina [12] la validità delle informazioni relative alle tranche possedute dal peer appena "scongelato" ed informa [13] il creatore (come descritto per la primitiva startP, par. *5.3.3.1*, punto 5).

Per i frammenti di codice che definiscono l'interazione con l'AMS si veda la *Figura 5-21* (p.171).

5.3.3.6 stopP

La primitiva stopP esegue operazioni riconducibili alla primitiva startP. La differenza sostanziale risiede nel fatto che prima viene inoltrata la primitiva al PMA del peer target e che solo successivamente viene richiesto al MainBoot di interrompere l'esecuzione di tutti gli agenti (sempre ad eccezione del PMA).

Per quanto riguarda le interazioni con il creatore ed il peer target, si faccia riferimento alle considerazioni riportate nei paragrafi precedenti per le altre primitive di stato per i peer.

5.3.3.7 downP

La primitiva downP viene utilizzata per riportare nello stato pSTOPPED un peer che si trova nello stato pFAILURE e nel loro stato di default tutti i behaviour dell'agente PMA.

Dal momento che lo stato pFAILURE indica che si è verificato un errore durante l'esecuzione di una primitiva di stato eseguita sul peer, si è scelto di risolvere la situazione facendo in modo che l'esecuzione di tutti gli agenti venisse interrotta e che le eventuali risorse occupate venissero rilasciate.

La struttura di questa primitiva ricalca quella della primitiva stopP; tuttavia, dal momento che si tratta di gestire una condizione di errore, non si dà per certa l'operatività di tutti gli agenti: quindi può capitare che l'esecuzione di tali agenti venga interrotta in modo brusco, senza attendere le rispettive conferme.

Gli agenti UA e DA, in particolare, gestiscono direttamente le connessioni da/verso altri peer per il trasferimento delle tranche. Quelle che riguardano i trasferimenti extra-regione vengono lasciate attive e, in caso di FAILURE, potrebbero provocare problemi per le successive comunicazioni (instaurate dopo il riavvio del peer) attraverso le stesse porte locali. Per questo motivo la primitiva downP viene inoltrata specificatamente verso questi due agenti e, nel caso non potesse essere eseguita, al successivo riavvio sarà compito dell'agente HA determinare un'altra porta locale per il trasferimento extra-regione[45].

[45] Maggiori dettagli verranno forniti nel paragrafo *5.4*.

5.3.4 Primitive di supporto

In questa breve sezione si descrivono sommariamente le primitive di supporto previste per il nuovo sistema. Alcune possono essere lanciate direttamente tramite l'interfaccia grafica dell'agente PMA, mentre altre hanno un'utilità prettamente connessa all'esecuzione di altre primitive.

Quelle che vengono riportate subito qui di seguito prima del paragrafo *5.3.4.1*, invece, sono state implementate solo in parte e non possono, quindi, essere utilizzate. Tuttavia si ritiene opportuno delinearne sinteticamente le caratteristiche principali.

shutdownP/shutdownR

La primitiva shutdownP (lanciata tramite GUI) che ha come target un peer:

1. raggiunge il superpeer della regione in cui è presente il peer target;
2. viene inoltrata dal superpeer della regione direttamente al MainBoot del peer target;
3. interrompe immediatamente senza ulteriori passaggi l'esecuzione di TUTTI gli agenti del peer, compreso il PMA;
4. disconnette il container dal main container della regione;
5. arresta definitivamente il container;
6. lascia in esecuzione il MainBoot.

La primitiva shutdownR che ha, invece, come target una regione (superpeer):

1. raggiunge il server;
2. viene inoltrata al PMA del superpeer target affinchè esegua una primitiva shutdownP su tutti i peer della regione;
3. viene inoltrata dal server direttamente al MainBoot del superpeer target affinchè interrompa immediatamente senza ulteriori passaggi l'esecuzione di TUTTI gli agenti di "tipo peer", compreso il PMA, e di "tipo superpeer";
4. arresta definitivamente il container ed il main container presenti sul superpeer;
5. lascia in esecuzione il MainBoot.

resetP/resetR

Le primitive resetP e resetR eseguono le stesse operazioni previste da downP e downR ma, in aggiunta, procedono ad un nuovo avvio del container (per i peer) e di container e main container (per i superpeer).

5.3.4.1 Peer

getMainBootAddress

La primitiva non può essere lanciata tramite la GUI e viene utilizzata tra gli agenti PMA con due scopi:

1. ottenere l'endpoint del MainBoot del nodo target (peer o superpeer);
2. verificare la raggiungibilità dell'agente PMA.

5.3.4.2 Superpeer

Per un superpeer sono previste le seguenti primitive di supporto in aggiunta a quelle previste per un peer (paragrafo *5.3.4.1*).

getRegionState

La primitiva raggiunge il PMA del superpeer e richiede lo stato (`TypeRegionStatus`) del superpeer stesso che viene comunicato attraverso un oggetto `EntityState` memorizzato come stringa di testo nel campo `answer` dell'oggetto `Primitive` trasportato (campo `content`) dal messaggio ACL verso il creatore della primitiva.

getPeerState

La primitiva richiede lo stato di un particolare peer della regione coordinata dal superpeer che riceve il messaggio. La risposta alla richiesta viene restituita con un oggetto `EntiyState` sottoforma di stringa, analogamente a quanto accade per la primitiva getRegionState.

getAllPeerState

Viene richiesto lo stato di tutti i peer della regione. Tali informazioni vengono attualmente comunicate inserendo nel campo answer dell'oggetto `Primitive` (trasferito tramite messaggio ACL) la stringa restituita dal

(nuovo) metodo `toString()` dell'oggetto `StateVector` che mantiene le informazioni sullo stato di tutti i peer.

overlayNetworkUpdate

La primtiva che raggiunge un superpeer contiene nel campo answer l'oggetto `OverlayNode` aggiornato.

5.3.4.3 Server

Per il server sono previste le seguenti primitive di supporto in aggiunta a quelle previste per un superpeer (paragrafo *5.3.4.2*).

printOverlayNetwork

La primitiva restituisce la composizione dell'anello dei superpeer attualmente connessi al sistema. Tale informazione viene comunicata tramite una semplice stringa di testo che mostra il contenuto di tutti i campi degli oggetti `OverlayNode`.

getNodeState

La primitiva che raggiunge il server (che appartiene, ad esempio alla REGION1) nella forma "getNodeState REGION2" restituisce le informazioni contenute nell'oggetto di tipo OverlayNode associato alla REGION2.

L'oggetto `OverlayNode` viene trasmesso come stringa di testo nel campo `answer` dell'oggetto `Primitive` trasportato (campo `content`) dal messaggio ACL verso il creatore della primitiva.

5.4 Protocolli di comunicazione per il trasferimento extra-regione

Data la criticità delle fasi che riguradano i trasferimenti della tranche tra regioni diverse attraverso Internet, si ritiene necessario descrivere più accuratamente i protocolli di comunicazione fra gli agenti maggiormente coinvolti:

1. UA – DA;
2. UA – HA e DA – HA;
3. HA – RPA;
4. HAHoleClientBhv – HaHoleServerBhv.

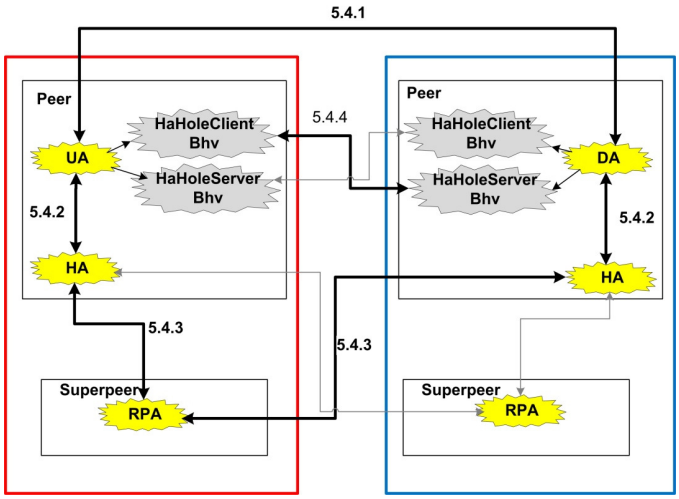


Figura 5-34 – Agenti coinvolti nel trasferimento extra-regione.

5.4.1 Protocollo UA - DA

Gli agenti UA e DA si occupano in modo diretto del trasferimento delle tranche. Il caso che vede i due peer (fonte e richiedente) appartenenti alla stessa regione viene trattato esattamente allo stesso modo di come veniva gestito dal prototipo precedente: l'agente UA del peer fonte effettua una richiesta al DA del peer richiedente al fine di conoscere il suo endpoint (ip:porta); successivamente l'UA instaura una connessione TCP verso il DA e comincia a trasferire la tranche.

Il caso del trasferimento extra-regione, invece, richiede maggiore attenzione. Innanzitutto si ricorda che, generalmente, i due agenti (localizzati su peer di regioni distinte) non hanno modo di essere raggiunti direttamente dall'esterno della propria regione tramite connessioni TCP senza una preventiva fase di negoziazione (svolta dai rispettivi agenti HA). In secondo luogo si precisa che è utile non chiudere la connessione TCP appena utilizzata per il vero e proprio trasferimento di una tranche: in questo modo, infatti, si evita che si verifichino errori di tipo "Address already in use" per le successive comunicazioni attraverso la stessa porta locale (per più dettagliate considerazioni sul ruolo del NAT e sulla necessità del riutilizzo della stessa porta, si rimanda al *CAPITOLO 2*).

Al fine di illustrare al meglio le caratteristiche dell'interazione fra gli agenti UA e DA, si consideri la *Figura 5-35* (p.206). In questa immagine si distinguono i due agenti (appartenenti a peer di regioni differenti) e vengono rappresentate le attività svolte da ognuno di essi a seconda della situazione in cui si trovano:

- in VERDE e in BLU sono evidenziate le operazioni eseguite in condizioni standard, quando i due agenti dei due peer non hanno mai instaurato alcun tipo di comunicazione;
- in ROSSO, invece, risaltano le attività svolte nel caso in cui l'agente UA del peer uploader si accorge della presenza di una connessione TCP già instaurata e funzionante verso l'agente DA del peer donwloader (connessione evidentemente utilizzata in precedenza e lasciata attiva - KEEP_ALIVE = true -): in questo caso il flusso ROSSO sostituisce quello VERDE.

FASE1a (VERDE)

La procedura comincia quando l'UA riceve dall'agente UMA il messaggio `READY_FOR_UPLOAD_TRANCHE` che lo informa della necessità di effettuare l'upload di una tranche (tutte le informazioni necessarie sono contenute all'interno di un oggetto di tipo `DownalodInformation`).

Se l'UA (peer uploader) NON rileva la presenza di un `Socket` valido verso il DA dell'altro peer:

1. prima invia le `DownloadInformation` al DA (che così si mette in attesa di ricevere informazioni dal proprio agente HA);
2. poi richiede all'agente HA (peer uploader) di iniziare la fase di rendezvous della procedura di TCP Hole Punching[46];
3. quindi si mette in attesa di una risposta dall'HA.

L'agente HA può:

a) rilevare un errore ed informare l'UA affinchè lo notifichi all'agente HA dell'altro peer, il quale, poi, sbloccherà il DA in attesa;
b) notificare all'UA che l'HA del peer dowloader ha rlevato un errore;
c) restituire all'UA l'endpoint pubblico del peer downloader verso cui poter tentare di instaurare la connessione TCP.

Nei casi a. e b. l'UA tornerà nel suo stato di default WAITING_FOR_MESSAGE, mentre nel caso c. proseguirà con le attività.

Se l'HA ha comunicato l'endpoint dell'altro peer, l'UA avvierà i due behaviour HAHoleClientBhv e HAHoleServerBhv (che proseguiranno la procedura di TCP Hole Punching) e attenderà che gli venga restituito un oggetto Socket verso l'altro peer.

[46] cfr. *2.5.3* e *Figura 2-16* (p.47).

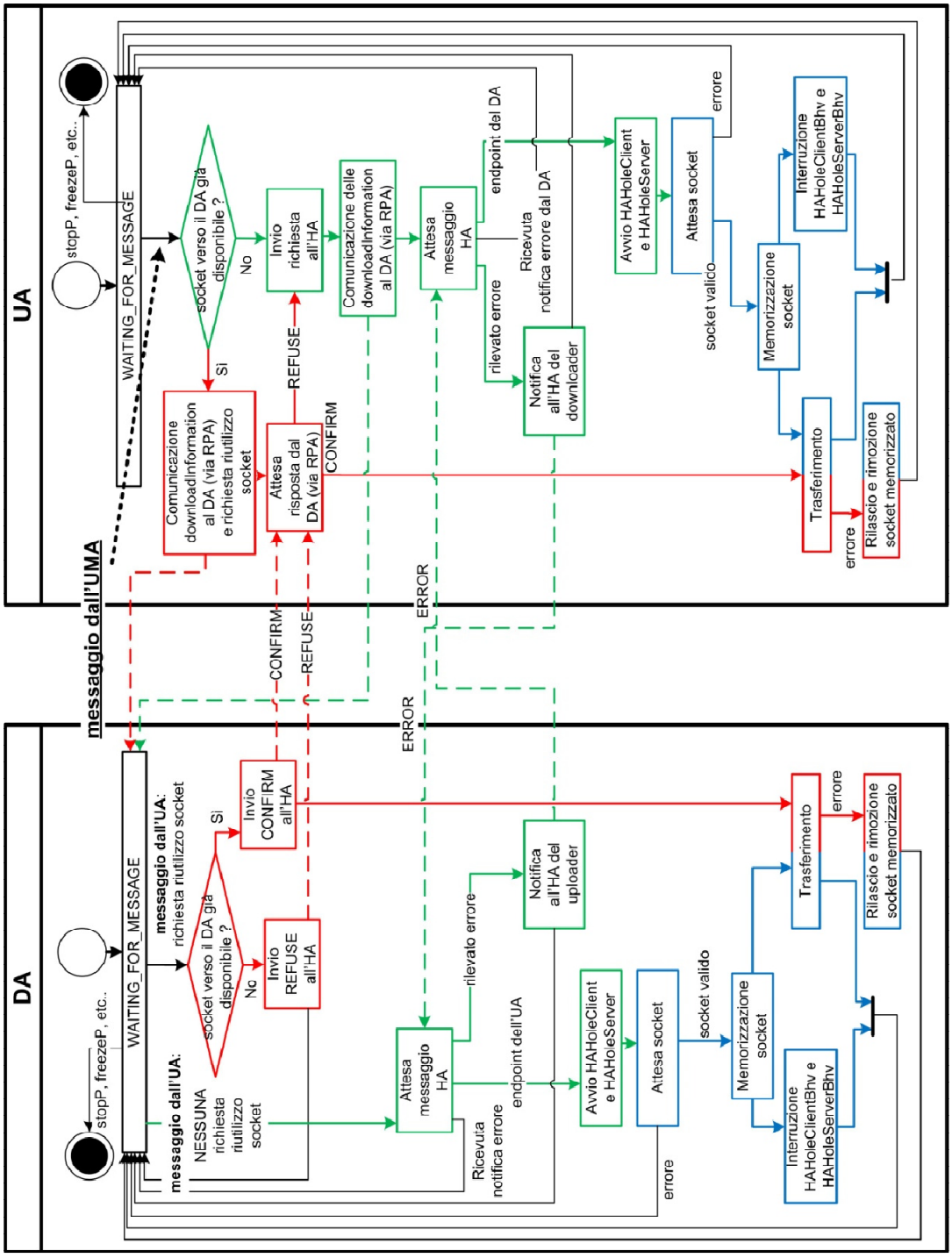


Figura 5-35 – Protocollo UA – DA

Nel frattempo anche l'agente HA del peer downloader ha proceduto allo stesso modo dell'agente HA del peer uploader:

a) ha rilevato un errore ed ha informato il DA affinchè lo notificasse all'agente HA dell'altro peer;
b) oppure ha notificato al DA che l'HA del peer uploader ha rilevato un errore;
c) oppure ha restituito al DA l'endpoint pubblico del peer uploader verso cui poter tentare di instaurare la connessione TCP.

Quindi anche l'agente DA del peer donwloader ha avviato i due behaviour HAHoleClientBhv e HAHoleServerBhv ed è in attesa di un Socket.

FASE2 (BLU)

Se i due ThreadedBehaviour HAHoleClientBhv e HAHoleServerBhv di ciascun peer sono riusciti ad instaurare una connessione TCP attraverso Internet: ne daranno notifica ai rispettivi agenti che li hanno creati (UA nel caso del peer uploader; DA nel caso del downloader), i quali memorizzeranno l'associazione fra il Socket restituito e l'agente (DA o UA) dell'altro peer.

Successivamente il DA e l'UA si assicureranno della corretta terminazione dei due ThreadedBehaviour e cominceranno il trasferimento della tranche richiesta; se il trasferimento andrà a buon fine, torneranno direttamente nel loro stato di default `WAITING_FOR_MESSAGE`, altrimenti prima rimuoveranno l'associazione tra il `Socket` e l'altro agente e poi torneranno nel loro stato di default.

Nel caso in cui, invece, i due ThreadedBehaviour di ciascun peer non siano stati in grado di instaurare una connessione, gli agenti UA e DA torneranno semplicemente nello stato di default.

FASE1b (ROSSO)

La FASE1b è da intendersi in sostituzione della FASE 1a quando l'UA rileva la presenza di un `Socket` valido verso il DA dell'altro peer.

In questa situazione l'UA richiede al DA se anche lui ha a disposizione un `Socket` valido (gli invia anche le `DownaloadInformation`, inoltrandole prima all'agente RPA della regione del DA affinchè le inoltri al DA[47]).

[47] cfr. 5.2.4 (Agente RPA).

- Se il DA conferma la propria disponibilità al riutilizzo del Socket esistente (`CONFIRM`), entrambi gli agenti (UA e DA) eseguiranno il trasferimento attraverso tale `Socket`;
- in caso contrario:
  a) il DA torna nel suo stato di default in attesa del primo messaggio dall'agente UA;
  b) l'UA prosegue inviando al proprio agente HA la stessa richiesta che gli avrebbe inviato se avesse intrapreso il flusso di attività VERDE fin dall'inizio.

NOTA

In *Figura 5-36* è rappresentato un diagramma di comunicazioni di esempio: nella fase 1 (comunicazioni [1] e [2]) si distiguono le comunicazioni relative al flusso di attività ROSSO della *Figura 5-35* (p.206); nella fase 2 (comunicazioni [3] – [14]) il flusso VERDE; nella fase 3 il flusso BLU.

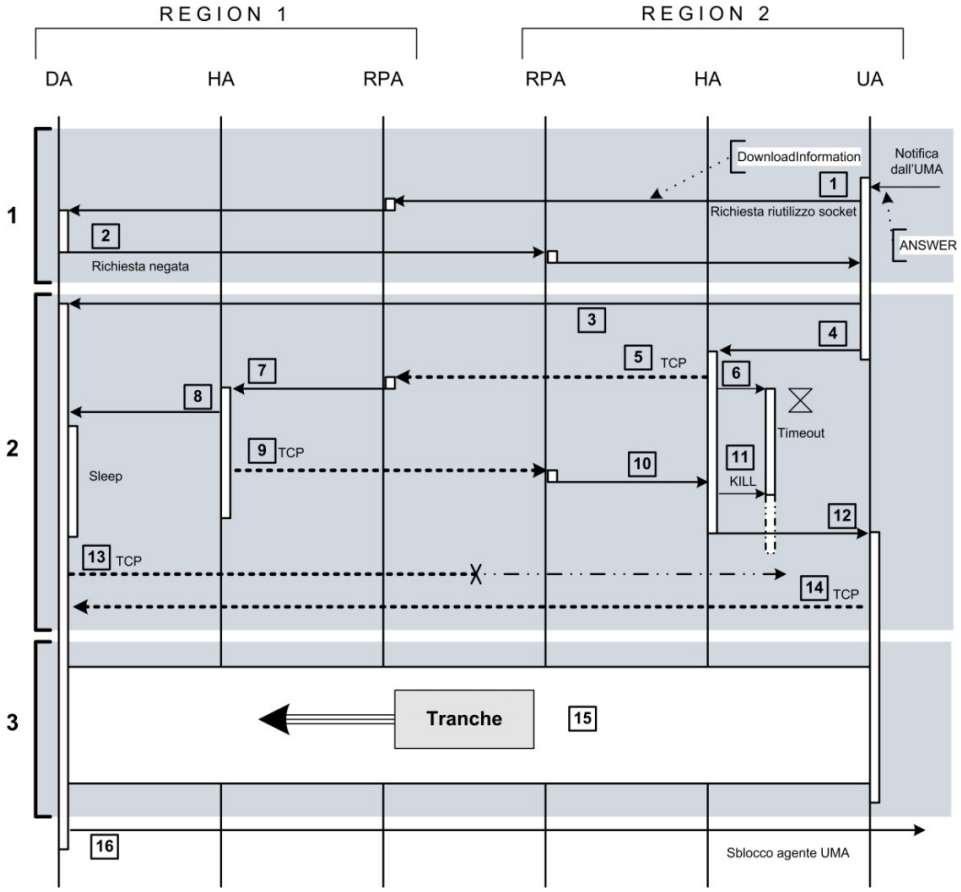


Figura 5-36 – Trasferimento extra-regione.

5.4.2 Protocollo UA - HA e DA - HA

L'interazione tra gli agenti DA e HA e fra gli agenti UA e HA è stata in parte analizzata nel paragrafo precedente. Per dare un'idea più dettagliata delle comunicazioni che avvengono fra questi agenti, si osservi la *Figura 5-36* (fase 2, comunicazioni [3] – [14]).

Dopo che l'UA ha avvisato [3] il DA (tramite l'agente RPA dela regione del DA) dalla possibilità di eseguire il downlaod di una tranche, richiede [4] all'agente HA di ottenere l'endpoint pubblico dell'altro peer e si mette in attesa di una risposta. Anche Il DA, ricevuto il messaggio dal'UA, è in attesa di una risposta dal proprio HA.

In condizioni ideali ogni HA riesce ad instaurare ([5] e [9]) una comunicazione via TCP verso l'agente RPA dell'altra regione, portando a termine la fase di rendezvous della procedura di TCP Hole Punching[48] e notificando ([8] e [12]) l'endpoint pubblico dell'altro peer al rispettivo agente (l'HA del downloader lo comunica [8] al DA, mentre l'HA dell'uploader [12] all'UA).

Se, però, l'HA della fonte (destra) si accorge (tramite un timeout avviato in [6] o in seguito ad errori di comunicazione) che è impossibile raggiungere l'RPA del peer richiedente (la comunicazione [4] non va a buon fine), allora informa l'UA; l'HA del richiedente non è stato informato dal proprio RPA (attraverso quella che sarebbe dovuta essere la comunicazione [5]) circa l'endpoint pubblico della fonte e ciò crea un problema dal momento che il DA è bloccato in attesa della comunicazione [6]. Per risolvere la situazione l'UA invia al HA del richiedente (sinistra) tramite l'RPA della REGION1 un messaggio ACL che lo avvisi della condizione di failure e che gli permetta di sbloccare il DA che potrà tornare, quindi, nello stato di default.

Un discorso analogo si può fare nel caso sia l'HA del richiedente (sinistra) ad accorgersi di un errore durante il tentativo di comunicazione con l'RPA della regione della fonte.

5.4.3 Protocollo HA - RPA

Un esempio "ad alto livello" dell'interazione fra l'agente HA e l'agente RPA è riportato in *Figura 5-36* (comunicazioni [5], [7], [9] e [10]).

[48] cfr. 2.5.3 e Figura 2-16 (p.49).

Il protocollo di interazione fra questi due agenti rappresenta l'implementazione della fase di rendezvous progettata nel paragrafo *2.5.3* e schematizzata in *Figura 2-16* (p.46); inoltre si fa riferimento alle considerazioni sulla gestione dei guasti presentate in 4.4.2.3 (p.117).

L'agente HA tenta di instaurare una comunicazione TCP con l'agente RPA dell'altro peer: infatti è attraverso tale comunicazione che l'RPA è in grado di rilevare l'endpoint pubblico associato dal NAT della regione del HA alla connessione in uscita verso la regione del RPA[49].

L'agente HA, quando riceve una richiesta dal UA prima di tutto controlla se ha già a disposizione un Socket valido verso l'RPA della regione del DA:

- se esiste una connessione TCP utilizzabile (le connessioni con l'RPA non vengono chiuse dopo la fase di rendezvous; all'altro capo della connessione è presente un behaviour RPARendezvousServerBhv) contatta l'RPA chiedendogli di rilevare il proprio endpoint pubblico e di inviarlo all'agente HA del peer downloader[50];
- altrimenti tenta di instaurare una nuova connessione TCP verso l'agente RPA dell'altra regione:
  - se la connessione va a buon fine, si mette in attesa di ricevere dall'agente RPA l'enpointpubblico dell'altro peer;
  - altrimenti, se l'HA riscontra qualche errore nella comunicazione con l'RPA, ne dà tempestiva notifica al UA e la prossima volta che l'HA si troverà a dover instaurare una connessione TCP verso un RPA, lo farà attraverso una nuova porta locale[51], in modo da escludere problemi relativi all'associazione effettuata dal NAT tra l'endpoint privato usato dall'agente HA e l'enpoint pubblico.

[49] E' questo l'endpoint (ip:porta) che l'RPA comunica all'agente HA del peer che si trova nella sua stessa regione e che verrà poi inoltrato dall'HA all'agente DA, nel caso l'HA sia nel peer downloader, o all'agente UA nel caso dell'uploader.

[50] Questa richiesta avviene tramite il semplice invio di una stringa di testo attraverso la connessione TCP.

[51] Si tratta della stessa porta attraverso cui avverrà il trasferimento della tranche.

La scelta di mantenere attive le connessioni verso gli agenti RPA e, quindi, di memorizzare l'associazione tra l'agente RPA ed il relativo `Socket` utilizzato è stata guidata dalla necessità di:

1. evitare di dover instaurare una connessione con l'RPA ogni volta che si deve trasferire un file da/verso peer nella stessa regione;
2. escludere eventuali problematiche relative alla mappatura da parte del NAT tra endpoint pubblico ed endpoint privato delle comunicazioni instaurate dall'HA;
3. escludere eventuali problemi connessi alle tempistiche di rilascio dei socket da parte del sistema operativo.

5.4.4 Protocollo HAHoleClientBhv - HAHoleServerBhv

Le interazioni fra questi due behaviour quando sono localizzati su peer diversi sono descritte nella *Figura 5-37*, che analizza nel dettaglio le operazioni [13] e [14] della *Figura 5-36*.

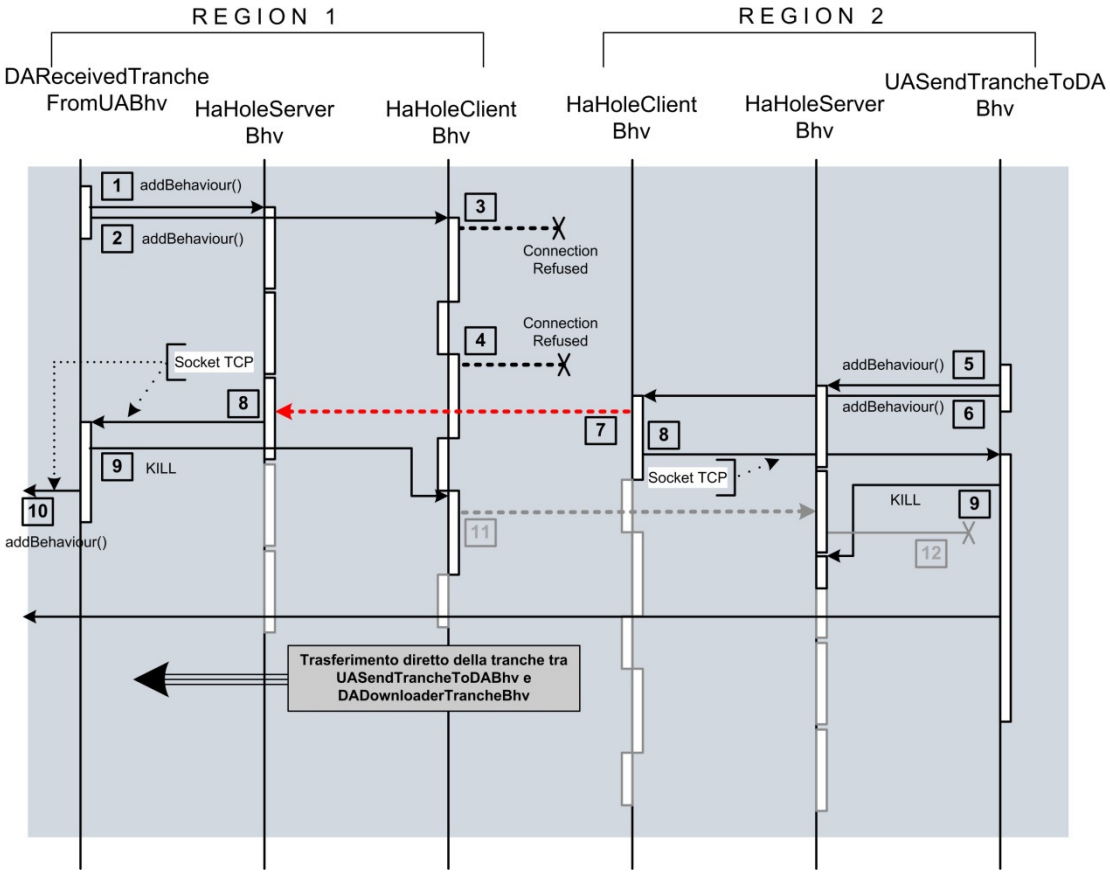


Figura 5-37 – Creazione del Hole TCP.

Quando il DA riceve dall'HA l'endpoint pubblico dell'UA dell'altro peer (comunicazione [8], Figura *5-36*), dopo una breve attesa avvia ([1] e [2]) i due behaviour su thread indipendenti (come ThreadedBehaviour):

- il behaviour HAHoleServerBhv si mette in attesa di ricevere una connessione in ingresso proveniente dal behaviour HAHoleClientBhv dall'altro peer;
- il behaviour HAHoleClientBhv, invece, tenta ripetutamente, attraverso la STESSA porta locale, di iniziare una connessione verso il behaviour HAHoleServerBhv dell'altro peer.

Si precisa che ciascuno dei due behaviour rimane, comunque, sensibile ad eventuali messaggi di interruzione provenienti dall'agente (UA o DA) che li ha creati e relativi all'esecuzione di primtive di amministrazione come stopP, downP o freezeP.

Nella figura si vede che l'HAHoleClientBhv avviato dal DA tenta invano (comunicazioni [3] e [4]) di raggiungere tramite connessione TCP l' HAHoleServerBhv avviato dall'UA: infatti non solo l'HAHoleServerBhv non è stato ancora avviato dal UA, ma la connessione viene bloccata dal NAT della regione dell'UA, in quanto nessuna comunicazione uscente verso il peer downloader è stata già iniziata dall'agente UA.

Quando, dopo essere stato avviato [6] dall'UA, l'HAHoleClientBhv tenta di instaurare [7] una connessione verso l'HAHoleServerBhv del downloader, connessione viene finalmente instaurata ed il socket viene comunicato [8] dall'HAHoleClientBhv all'UA e dall'HAHoleServerBhv al DA.

Infine UA e DA si preoccupano di terminare l'esecuzione del secondo ThreadedBehaviour, ignorando l'eventuale ulteriore connessione che potrebbe essere instaurata (comunicazioni [11] e [12]) eseguendo il trasferimento della tranche che può quindi avvenire dierttamente attraverso una connessione TCP fra i due peer delle due regioni.

5.5 Collaudo

Parallelamente all'attività di implementazione è stata svolta quella di test e collaudo, con lo scopo di verificare la fattibilità di alcune scelte realizzative, le funzionalità dei singoli componenti e le interazioni fra essi ed il sistema nel suo complesso. Il vantaggio principale dell'aver eseguito queste operazioni di controllo durante la realizzazione è stato di aver potuto

apportare, ove necessario, delle modifiche al progetto, riuscendo a salvaguardare le specifiche di modello definite in principio.

5.5.1 Ambiente operativo

La struttura dell'ambiente operativo in cui è stato svolto il collaudo non è rimasta invariata dall'inizio alla fine di questa attività e ha seguito l'evolversi del sistema: la prima struttura ha visto la presenza di due regioni localizzate in due luoghi distinti connessi tramite WAN, mentre la seconda è stata caratterizzata dall'aggiunta di una terza regione.

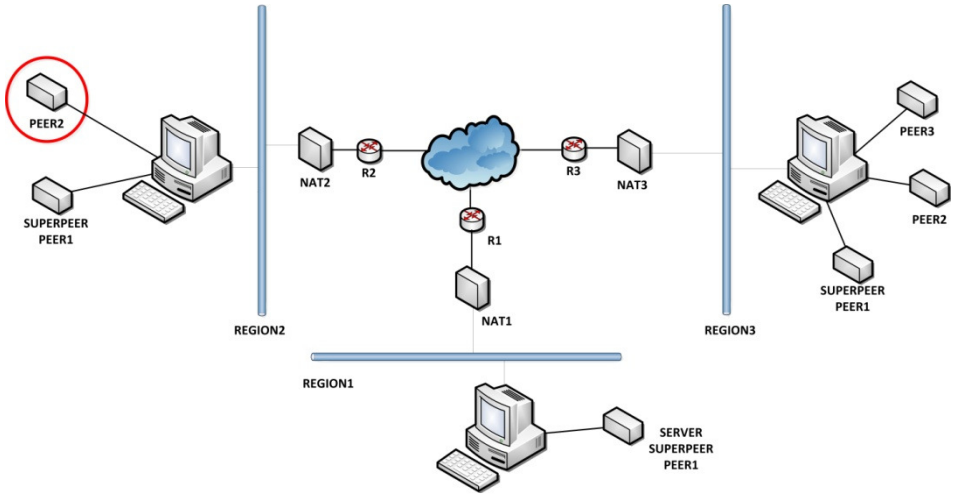


Figura 5-38 – Ambiente operativo usato per il collaudo.

Ciascuna regione, in base alle esigenze specifiche di ogni test, poteva essere composta solo da un superpeer oppure da un superpeer e da uno o più peer (eventualmente nessuno).

E' stato necessario garantire l'accesso simultaneo a tutti i peer di tutte le tre regioni da un unico centro di controllo, affinchè fosse possibile modificare all'occorrenza le varie configurazioni, apportare le necessarie modifiche e controllare l'esito dei test[52].

Per garantire la presenza ed il controllo di più peer nella stessa regione si è ricorso a tecniche di virtualizzazione di sistemi operativi e, nello

[52] Oltre alla configurazione tipica del sistema durante la normale attività (più regioni connesse tramite WAN), è stato verificato il funzionamento che vede due regioni all'interno della stessa LAN. Questo tipo di scenario richiede necessariamente alcune modifiche ai file di configurazione e la corretta impostazione delle regole di port forwarding sul NAT che coordina le comunicazioni da/verso le regioni che coesistono nella stessa LAN.

specifico, all'applicativo Microsoft Virtual PC: in ognuna delle sedi (due prima, tre in seguito) era in esecuzione un'unica macchina che fungeva da superpeer e su cui venivano virtualizzati fino a quattro sistemi ospite (che fungevano da peer). L'accesso alla macchina host era garantito dalla combinazione degli applicativi UltraVNC[53] e SSH (con cui si instaurava un tunnel SSH, attraverso il quale operava VNC).

Di seguito si riporta, a titolo esemplificativo, il contenuto dei campi più significativi dei file di configurazione di un superpeer e di un peer appartenenti alla stessa regione REGION2; il server è nella regione REGION1 mentre la terza regione è denominata REGION3.

Descrizione	Nome	Superpeer	Peer
Nome della regione	REGION_NAME	REGION2	REGION2
Nome del peer all'interno della regione	PEER_NAME	PEER1	PEER2
IP locale del peer	LOCAL_IP	Profile .getDefault NetworkName();	Profile .getDefault NetworkName();
il peer è il server (anche superpeer)	IS_SERVER	false	false
il peer è un superpeer	IS_SUPERPEER	true	false
Nome del superpeer	SUPERPEER _NAME	PEER1	PEER1
IP pubblico del superpeer	SUPERPEER _PUBLIC_IP	getIpByName("** **.dyndns.org");	getIpByName("* ***.dyndns.org");
Indirizzo IP locale (LAN) del superpeer	SUPERPEER _LOCAL_IP	LOCAL_IP	192.168.0.4
Porta utilizzata dal superpeer per il rendezvous	SUPERPEER _RENDEZVOUS_PORT	5560	5560
Nome del server	SERVER_NAME	PEER1	-
Regione del = "REGION1";	SERVER_REGION	REGION1	-
IP pubblico del server (visibile da tutti i superpeer)	SERVER_PUBLIC_IP	getIpByName("** **.dyndns.org");	-

Figura 5-39 – Parameters.java [esempio]

[53] UltraVNC è stato utilizzato anche in associazione al plugin MSRC4Plugin che consente una cifratura a chiave simmetrica RC4 a 128bit.

Nelle prossime due figure si evidenzia il parametro che consente a ciascun peer di connettersi al database utilizzato da Persistence che viene creato sul superpeer con IP locale 192.168.0.4 e a cui viene permessa la connessione da un servizio server in ascolto sulla porta 2000 (cfr server.properties).

```
...
hibernate.dialect
org.hibernate.dialect.HSQLDialect

hibernate.connection.driver_class
org.hsqldb.jdbcDriver

hibernate.connection.username sa

hibernate.connection.password


hibernate.connection.url
jdbc:hsqldb:file:/Workspace/MultiRegione/JADE-DB

...
```

Figura 5-40 – hibernate.properties (Persistence): superpeer [esempio]

```
...

hibernate.dialect
org.hibernate.dialect.HSQLDialect

hibernate.connection.driver_class
org.hsqldb.jdbcDriver

hibernate.connection.username sa

hibernate.connection.password


#hibernate.connection.url
jdbc:hsqldb:hsql://192.168.0.4:2000/JADE-DB

...
```

Figura 5-41 – hibernate.properties (Hibernate): peer [esempio]

Nella figura seguente si riporta un esempio della configurazione del file main.properties di Persistence.

```
...
main=true
gui=true
port=2000
services=jade.core.persistence.PersistenceService;
      jade.core.event.NotificationService;jade.cor
      e.mobility.AgentMobilityService;org.hsqldb.S
      erver;org.hsqldb.util.DatabaseManager
meta-db=hibernate.properties
...
```

Figura 5-42 – main.properties (Persistence) [esempio]

Di seguito è indicata la configurazione del file server.properties utilizzato da HyperSQL per offrire agli altri peer la possibilità di connettersi da remoto al database localizzato in `Workspace/MultiRegione/JADE-DB`.

```
...
server.database.0=file:/Workspace/MultiRegione/JADE-
      DB;ifexists=false
server.dbname.0=JADE-DB
server.port=2000
server.trace=false
...
```

Figura 5-43 – server.properties (HyperSQL): superpeer [esempio]

Di seguito vengono elencate le porte per cui devono essere configurate le regole per il port forwarding (tali porte devono avere visibilità pubblica).

Descrizione	Visibilità	TCP
JADE Main port	locale	1099
Container Local Port	locale	7778
MainBoot Port	locale	4460
Tranche Transfer Local Port	locale	9977, 9978, 9979
Extra-region Tranche Transfer Local Port	locale	8888, 8889, 8890

Tabella 5-7 – Peer: requisiti di comunicazione e port forwarding.

Descrizione	Visibilità	TCP
Server SSH	pubblica	2222
UltraVNC Server	pubblica	5900, 5800
JADE Main port	locale	1099
Container Local Port	locale	7778
Container Public Port	pubblica	6668
Main Container Local Port	locale	12000
Main Container Public Port	pubblica	13000
Rendezvous Port	pubblica	5560
MainBoot Port	pubblica	4460
Tranche Transfer Local Port	locale	9977, 9978, 9979
Extra-region Tranche Transfer Local Port	locale	8888, 8889, 8890

Tabella 5-8 – Superpeer: requisiti di comunicazione e port forwarding.

5.5.2 Descrizione ed esiti dell'attività

L'attività di collaudo è stata svolta in due fasi: inizialmente si è proceduto ad un collaudo di unità, successivamente si è passati ad un collaudo di integrazione.

Durante la prima fase è stata verificata la presenza di errori introdotti durante la scrittura del codice e si è concentrata l'attenzione sui singoli componenti e sulle relazioni dirette di ogni componente con quelli ad esso correlati.

Durante la seconda fase è stata focalizzata di volta in volta l'attenzione su gruppi di componenti e sulle interazioni fra di essi, fino ad arrivare alla verifica di quelle funzionalità che coinvolgono trasversalmente più aree del sistema.

I primi componenti collaudati sono stati quelli relativi alla gestione dei trasferimenti extra-regione: dopo aver sviluppato un primo semplice prototipo basato su thread per verificare l'efficacia della tecnica del TCP Hole Punching, si è proceduto alla realizzazione ed al collaudo degli agenti interessati da queste operazioni. E' stata controllata la capacità di questi componenti di reagire alle condizioni di guasto previste e di lasciare il sistema in uno stato consistente.

Successivamente si è passati alla realizzazione ed al collaudo dei componenti che elaborano e gestiscono le primitive di amministrazione, con particolare attenzione al rilevamento ed alla gestione dei guasti a tutti i livelli (behaviour, agenti, peer/superpeer), così come espresso dalle considerazioni riportate nel *CAPITOLO 4* e nel *CAPITOLO 5*.

Quindi è stata verificata la corretta esecuzione delle primitve di amministrazione per i peer da parte di tutti gli agenti coinvolti dall'eleborazione delle stesse: le primitve freezeP e thawP, che sfruttano l'add-on Persistence, hanno richiesto maggiore impegno e si sono verificate le più ostiche insieme a stopP.

Successivamente sono state verificate le operazioni connesse alle primitive di amministrazione per le regioni, compreso l'avvio del sistema.

Infine si è proceduto a collaudare il funzionamento del sistema nel suo complesso (al fine di controllarne la stabilità) alternando aggiunta/rimozione di regioni, avvio/interruzione/ibernazione /sospensione di peer e ripristino di peer e regioni in seguito al verificarsi di situazioni di guasto.

L'esito delle attività di collaudo si è rivelato complessivamente positivo. L'esecuzione delle primitive stopP e freezeP lascia il sistema in uno stato consistente ed il trasferimento delle tranche da parte dei peer viene garantito anche dopo l'esecuzione delle primitive thawP e resumeP. Sono stati riscontrati dei problemi nel trasferimento delle tranche solo da/verso la regione situata sulla macchina host messa a diposizione dall'azienda: si ipotizza che la causa di tali errori sia legata alla configurazione di alcuni componenti che ne consentono l'accesso ad Internet. A parte questo, la gestione delle primitive per peer e regioni avviene correttamente e le attività di rilevamento e gestione dei guasti, così come sono state definite, non presentano problemi.

In definitiva la fase di test e collaudo ha accompagnato lo sviluppo del sistema fin dall'inizio e ha rappresentato un valido strumento di verifica "in itinere" di cui si è fatto largo uso per correggere anomalie e migliorare il prodotto finale.

5.6 Conclusioni e sviluppi futuri

L'attività di progettazione e realizzazione documentata in questo lavoro di tesi ha portato alla definizione di un sistema dotato di nuove funzionalità ed alla conseguente realizzazione di un prototipo che, grazie ad un'attenta fase analitica e progettuale, è andato ad integrarsi con quello precedente, facendo leva sulla presenza di un nucleo di base piuttosto che considerare i vincoli esistenti come un ostacolo limitante.

Nonostante l'attività progettuale si sia rivelata abbastanza proficua, molti aspetti sono stati trattati solo marginalmente sia per esigenze legate alle tempistiche imposte, sia per questioni connesse ad alcune scelte implementative.

Di seguito si passano in rassegna i punti che si ritiene debbano essere oggetto di attenzione durante una eventuale nuova rielaborazione del sistema; inoltre si riportano alcune considerazioni che potrebbero rivelarsi utili per migliorare alcune caratteristiche del sistema ottenuto e per introdurre nuove funzionalità.

Innanzitutto è necessaria qualche precisazione sulle fasi di avvio del sistema. Dal momento che questo argomento non è stato mai affrontato in modo sufficientemente concreto, si è preferito spostare l'attenzione (altrimenti rivolta ad esso) verso altri aspetti; tuttavia riveste una certa

importanza, dal momento che è proprio in questa prima fase che dovrebbe avvenire, da parte del server, l'invio ragionato[54] delle tranche dei vari contenuti verso le regioni connesse al sistema: in questo modo si avrebbe una rapida diffusione delle tranche che verrebbero scambiate fra i peer interessando sempre meno frequentemente il server.

Un'altra considerazione riguarda la gestione dei trasferimenti extra-regione. Come per molti altri aspetti, l'attività svolta ha avuto come obiettivo primario quello di "tradurre" il funzionamento del prototipo esistente in un ambiente più esteso che coinvolgesse la rete Internet; per questo motivo si è deciso di lasciare invariati i protocolli utilizzati per la ricerca extra-regione e per la configurazione del trasferimento. Potrebbe essere utile ideare dei protocolli meno soggetti ad errori nel caso si verifichino guasti, o più efficienti di quelli attualmente utilizzati.

Per quanto riguarda la gestione dei guasti, si può dire che è stato raggiunto un primo importante obiettivo: quello di considerare la possibilità che si possano verificare.

Infatti finora si era supposto che il sistema operasse in un ambiente quasi totalmente scevro di errori; adesso alcuni di essi vengono rilevati in specifiche condizioni (come gli errori di omissione o di temporizzazione che si verificano durante il cambiamento di stato o i trasferimenti delle tranche) al contrario di altri (come gli errori di tipo crash) che invece non vengono ancora gestiti. Nonostante nel *CAPITOLO 4* siano state riportate alcune considerazioni sul rilevamento del crash di un peer (ad esempio - gossiping - con la collaborazione degli altri peer della regione), l'argomento si rivela essere molto vasto e solo un'attività di analisi rivolta proprio in questa direzione potrebbe incrementare realmente il livello di tolleranza ai guasti del sistema[55].

Per ciò che concerne la gestione e l'elaborazione delle primitive di amministrazione, si può affermare che è stato ottenuto un risultato notevole: sono stati introdotti con successo i componenti necessari alla

[54] In funzione del contenuto dei rispettivi palinsesti.

[55] Ad esempio si potrebbe implementare una gestione distribuita della struttura della overlay network o una conoscenza distribuita della dislocazione dei contenuti; si potrebbe sfruttare il meccanismo di replicazione del main container offerto da JADE per ridurre l'entità delle conseguenze derivanti dal crash di un superpeer o elaborare una nuova strategia che preveda meccanismi più complessi (si noti la necessità di tenere sempre in considerazione la presenza dei NAT che possono complicare notevolmente la situazione).

corretta esecuzione di queste primitive e, per di più, è stata posta particolare attenzione affinchè potessero soddisfare un certo grado di modularità.

Infatti, dal momento che la gestione delle primitive pervade tutto il sistema e deve tenere in cosiderazione tutti i behaviour di tutti gli agenti, aver scelto di implementare una struttura modulare permette successive modifiche in modo molto intuitivo: l'aggiunta di nuove primitive da far eseguire ai vari agenti, l'aggiunta di nuovi agenti o l'aggiunta di nuovi behaviour per gli agenti esistenti può avvenire con il minimo sforzo, andando ad aggiornare alcune sezioni di codice presenti nei componenti principali e rispettando la nuova struttura di base che è stata definita[56].

Ad esempio eventuali aggiornamenti come l'aggiunta di nuovi agenti potranno essere effettuati seguendo la struttura definita per gli agenti esistenti e agendo sul PMA (e, specificatamente, sulla lista degli agenti interessati dalle primitive di amministrazione).

Si ritiene che possa essere utile implementare quelle primitive che sono state solo ideate durante questo progetto e che riguardano le operazioni di "shutdown" e "reset" di peer e regioni. Tali primitive (shutdownP, shutdownR, resetP, resetR) operano a livello di container e main container e potrebbero assumere una sostanziale importanza a livello di gestione dei guasti; infatti, poiché interessano la struttura di base della piattaforma JADE, sarebbe utili durante le fasi di ripristino[57].

Inoltre potrebbe rivelarsi sensato elaborare delle nuove primitive che operino a livello di sistema e che, ad esempio, ne permettano l'arresto completo (coordinando quello delle varie regioni e dei peer) o che lo inizializzino facendo riferimento ad una configurazione standard (sempre a livello di regioni e peer).

Da un punto di vista più implementativo si potrebbe migliorare la tecnica utilizzata per la connessione diretta fra peer di regioni diverse, magari alla luce di nuove sperimentazioni; oppure ci si potrebbe limitare a migliorare la gestione delle connessioni extra-regione dirette fra peer. Al

[56] L'aggiunta di agenti da inserire nella struttura di gestione delle primitive è facilmente ottenibile agendo solo sulle strutture dati del PMA e seguendo la struttura prevista per agenti e behaviour (macchina a stati finiti per la ricezione delle primitive per ogni behaviour, metodo afterLoad() per le operazioni richieste ad ogni behaviour dalla primitiva thawP, behaviour coordinatore, etc..)

[57] Cfr. CAPITOLO 4.

momento, infatti, non vengono interrotte dopo ogni trasferimento, ma vengono lasciate attive in attesa di un nuovo utilizzo, evitando così i problemi descritti nel *CAPITOLO 4* e nel *CAPITOLO 5*. Si potrebbe pensare di introdurre un meccanismo che, per ogni peer, analizzi il pool di connessioni attive e, ad esempio, interrompa quelle più datate o quelle che non corripondono ai peer che sono stati contattati più assiduamente.

Un'altra miglioria potrebbe consistere nell'aggiornare l'elaborazione delle richieste urgenti, considerando la presenza del server o ipotizzando che un peer possa chiedere al superpeer della propria regione (il superpeer è sempre raggiungibile direttamente dall'esterno) di ottenere per lui la tranche mancante (ciò ha senso qualora il peer si accorga di non poter raggiungere direttamente altri peer in altre regioni).

Un ulteriore miglioramento potrebbe essere quello di prevedere l'upload simultaneo verso più fonti, almeno da parte del server che si suppone possa disporre di una largezza di banda maggiore.

Osservando il sistema da una prospettiva più ampia si potrebbe pensare di aumentare il livello di configurabilità dei singoli peer e delle regioni durante il loro normale funzionamento, in modo che si possa aprire la strada, nel caso lo si ritenga opportuno, verso nuove funzionalità: ad esempio si potrebbe voler modificare l'indirizzo pubblico del server o del superpeer, oppure si potrebbe voler cambiare il palinsesto dei contenuti da visualizzare su ogni peer.

Si potrebbe concentrare l'insieme dei parametri di configurazione in un file di testo, in modo da facilitare l'installazione del sistema e, a livello di interazione con l'utente, si potrebbe aumentare il grado di usabilità modificando la semplice interfaccia grafica ideata per interagire con l'agente PMA in modo da renderla più *user-friendly*: ad esempio introducendo degli elementi grafici che rappresentino visivamente la struttura della overlay network e che diano informazioni sullo stato dei vari nodi, oppure organizzando nuove funzioni di amministrazione in menu che rendano più rapida l'interazione col sistema.

Ultima, ma non per questo meno importante, qualche considerazione sulla sicurezza, che è stato un aspetto finora trascurato durante tutte le tre fasi di sviluppo.

Così come in precedenza, infatti, si è dato per scontato che le comunicazioni avvengano in un ambiente sicuro o che, comunque, non ci siano problemi di questo tipo. Tuttavia bisognerebbe iniziare ad introdurre

qualche strategia per garantire un certo livello di sicurezza sia per quanto può riguardare l'autenticazione della figura dell'amministratore, sia relativamente alla migrazione degli agenti da una piattaforma ad un'altra, sia, eventualmente, per quanto attiene agli stessi trasferimenti delle tranche.

Indice delle figure

Indice delle tabelle

Bibliografia

Araragi, T. 2005. *Fault Tolerance for Internet Agent Systems in cases of stop failure and Byzantine failure.* Nippon telegraph and telephone Corporation : s.n., 2005. AAMAS'05.

Bellifemine, F., Caire, G. e Greenwood, D. 2004. *Developing multi-agent systems with Jade.* s.l. : Wiley, 2004.

Bellifemine, F., et al. 2007. *Jade administrator's guide.* 2007.

—. 2007. *Jade programmer's guide.* 2007.

Biggadike, A., Ferullo, D. e Geoffrey, W. 2005. *Natblaster: Establishing tcp connections between hosts behind nats.* s.l. : SIGCOMM Asia Workshop, 2005.

Biscuola, A. 2009. *Un sistema ad agenti mobili per la condivisione selettiva di contenuti multimediali su sistemi P2P.* s.l. : Master's thesis, University of Padova, 2009.

Bobice, A. 2009. *Sistema peer-to-peer per la distribuzione di contenuti multimediali.* s.l. : Master's thesis, University of Padova, 2009.

Caire, G. 2002. *Jade Tutorial, Application-defined content languages and ontologies.* s.l. : TILab S.p.A., 2002.

Chandy, K.M. February 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems.* February 1985, Vol. 3, No. 1, Pages 63-75.

Chen, J., et al. 2008. *An Efficient Forward and Backward Fault-Tolerant Mobile Agent System.* Institute of Networking and Communication Engineering : ChaoYang University of Tecnology, 2008. ISDA2008.

Eppinger, Jeffrey L. 2005. *Tcp connections for p2p apps: A software approach to.* CarniegeMellon University : Institute for Software Research, International School of Computer Science, 2005.

Ford, B., Srisuresh, P. e Kegel, D. 2005. *Peer-to-peer communication across network address translators.* Berkeley, CA, USA : USENIX Association, 2005. ATEC '05 - Proceedings of the annual conference on USENIX Annual Technical Conference - pag 13.

Francis, P. e Guha, S. Simple traversal of UDP through NATs and TCP too (STUNT). [Online] http://nutss.gforge.cis.cornell.edu/stunt.php.

Francis, P. 2003. Is the internet going nutss? *IEEE Internet Computing.* 2003, Vol. VII, 6.

Hibernate Community Documentation. [Online] http://docs.jboss.org/hibernate/stable/core/reference/en/html/.

Holdrege, M. e Srisuresh, P. January 2001. *Protocol complications with the IP network address translator.* s.l. : RFC 3027, January 2001.

JADE (Java Agent DEvelopment Framework). [Online] http://jade.tilab.com/.

JADE source code. [Online] http://jade.tilab.com/doc/api/index.html.

Koo, R. 2001. *Checkpointing and Rollback-Recovery for Distributed Systems.* Department of Computer Science : Cornell University, 2001.

Lloret, J., et al. *A Fault-Tolerant P2P-based Protocol for Logical Networks Interconnection.* Department of Communications : Polytechnic University of Valencia.

Magillo, P. 2009. Programmazione II (SMID: Statistica Matematica e trattamento Informatico dei Dati). *Elaborazione GraphicalUserInterface.* [Online] 2009. http://www.disi.unige.it/person/MagilloP/P2_SMID09/p2_smid09.html.

Persistence ontology source code. [Online] http://svn.assembla.com/svn/AIADA/JADE3_6/Src/src/jade/domain/persistence/.

Potiron, K., Tailliber, P. e Seghrouchni, A.E.F. *A Step towards Fault Tolerance for Multi-Agent.*

Pressmann, R.S. 2004. *Principi di Ingegneria del software.* s.l. : McGraw-Hill, 2004.

Rosenberg, J., et al. March 2003. *STUN - simple traversal of user datagram protocol (UDP) through network address translators (NATs).* s.l. : RFC 3489, March 2003.

Russell, J. W. Java authoring info. [Online] http://home.cogeco.ca/~ve3ll/jaintro.htm.

Shriushek, P. e Holdrege, M. August 1999. *Ip network address translator(traditional nat) terminology and considerations.* s.l. : RFC 2663, Internet Engineering Task, August 1999.

Tanenbaum, A. S. *Sistemi distribuiti.* s.l. : PEARSON, Prentice Hall.

The HSQL Development Group, Blaine Simpson, and Fred Toussi. 2009. HyperSQL User Guide. *HyperSQL Database Engine, aka HSQLDB.* 2009.

The Java tutorials. *Creating a GUI with JFC/Swing.* [Online] http://java.sun.com/docs/books/tutorial/uiswing/.