Università degli Studi di Roma "La Sapienza"
Facoltà di Ingegneria
Corso di laurea in Ingegneria Informatica

Anno accademico 2006/2007
Tesi di laurea specialistica
indirizzo Architetture e Sistemi Distribuiti

Raccolta ed elaborazione di dati provenienti da reti di sensori distribuiti

Candidato: Rughetti Diego

Relatore: Prof. Ciciani Bruno
Correlatore: Prof. Romano Paolo

1. Introduzione

L'utilizzo di tag rfid in questi ultimi anni ha conosciuto una forte accelerazione. Diretti discendenti delle tecnologie radar sviluppate durante la seconda guerra mondiale e concepiti inizialmente per la realizzazione di sistemi antitaccheggio e di inventariato, i tag rfid si stanno diffondendo in numerosi altri campi. Il mondo aziendale e quello industriale hanno da subito intuito i vantaggi apportati da questa tecnologia, in particolar modo se relazionata con le tradizionali tecniche di catalogazione (ad es. codice a barre), attraverso la quale è possibile ad esempio automatizzare il sistema di gestione delle scorte di magazzino, da sempre uno dei punti chiave delle attività industriali e commerciali. Un aumento dell'efficienza nella gestione delle scorte permette infatti di minimizzare i costi ad esse relativi nonché di ridurre il rischio di ritardi nella fornitura del prodotto, con tutti i vantaggi che ciò comporta (riduzione dei costi di produzione, aumento della soddisfazione del cliente ecc....). Avvenimenti piuttosto recenti, come ad esempio epidemie tra animali da allevamento (encefalopatia spongiforme bovina o influenza aviaria), hanno fornito una forte spinta alla diffusione della tecnologia rfid. Attraverso quest'ultima è infatti possibile realizzare sistemi di tracciamento della “vita di un prodotto”, permettendo in questo modo di determinare con certezza non solo la sua provenienza ma anche eventuali eventi significativi che lo hanno interessato fino a quel momento. In generale possiamo vedere che i campi d'applicazione sono svariati, oltre a quelli industriali, commerciali e d'allevamento essa viene utilizzata durante manifestazioni sportive, nell'ambito sanitario, nel controllo degli accessi, nel settore dei trasporti ecc...
La tecnologia rfid è basata essenzialmente su tre categorie di componenti: trasponder, reader, software. Le prime due sono quelle che formano la componente prettamente hardware della tecnologia, consistono sostanzialmente in una serie di dispositivi, tra cui i tag rfid veri e propri, che realizzano la struttura fisica di raccolta dati. Il software è invece la porzione “intelligente” della tecnologia. I sui compiti principali sono: controllo dell'infrastruttura formata dai readers, manipolazione ed interpretazione dei dati da questa prodotti. Il nostro lavoro si è focalizzato su tale componente software, in particolare sui moduli che si occupano della raccolta, manipolazione ed interpretazione dei dati. Tale porzione di sistema viene chiamata con il nome di middleware. Attualmente esistono un certo numero di prodotti software commerciali realizzati per svolgere tali mansioni (Microsoft BizTalk RFID, IBM WebSphere 6.0, Oat Systems, Oracle, GlobeRanger, BEA). Le problematiche che devono essere affrontate nello sviluppo di un middleware per sistemi rfid sono molteplici, e un gran numero di queste sono strettamente legate all'ambiente operativo del sistema.
La nostra attenzione si è concentrata su quelle relative alla fault tolerance. Il lavoro svolto non si limita però ad affrontare tali problematiche solo nell'ambito dei sistemi rfid, ma più in generale

indirizza il problema della fault tolerance per l'acquisizione di dati da reti di sensori. La progettazione di sistemi affidabili di solito richiede l'utilizzo di tecniche basate su ridondanza delle componenti, in modo tale che, anche in caso di guasto, possano esserci sufficienti risorse per garantire il corretto svolgimento delle attività del sistema. Nei sistemi rfid e più in generale in quelli di acquisizione dati da reti di sensori, la ridondanza può interessare differenti porzioni del sistema. Partendo dai sensori stessi, quindi dai readers e dai trasponder nel caso rfid, vediamo che in tale porzione la replicazione è realizzabile con modalità abbastanza semplici: oltre a curarne l'affidabilità, è sufficiente ridondare le singole componenti hardware, in modo tale che in caso di guasto di una di esse sia possibile rimpiazzarla run-time con un'apposita replica. Passando invece alla porzione di elaborazione dei dati, vediamo che questa è formata sia da componenti hardware che software. La replicazione delle prime può essere affrontata utilizzando al solito componenti fisiche ridondate, mentre per quanto riguarda le seconde le cose sono un po' più complesse. E' necessario adottare opportune tecniche di replicazione software, che garantiscano oltre alla coerenza dello stato di ogni replica, se presente, anche una visione uniforme della realtà monitorata dal sistema.
La realizzazione di sistemi fault tolerant è di fondamentale importanza ogniqualvolta si vuole, o ancor più si deve, avere la massima disponibilità del servizio in esame, ma anche quando eventuali procedure di riparazione risultano essere impossibili, perché ad esempio il sistema o porzioni di esso sono situati in locazioni non raggiungibili fisicamente, o sono possibili ad intervalli di tempo relativamente lunghi (si pensi ad esempio a sistemi distribuiti su area geografica, in cui i vari nodi operativi sono sparsi su migliaia di km quadrati di superficie, per raggiungere uno specifico nodo può occorrere una quantità di tempo misurabile in giorni). La replicazione delle componenti, in particolar modo quelle relative al middleware, può essere sfruttata oltre che per la fault tolerance, anche per migliorare l'efficienza, la scalabilità e le prestazioni del sistema. Se ad esempio questo viene organizzato in modo che i flussi di dati provenienti dalla rete di sensori, che come nel caso di sistemi rfid possono avere dimensioni consistenti, vengano elaborati in parallelo da diverse entità, allora è possibile ridurre i tempi di risposta, riuscendo a fornire in maniera più tempestiva i risultati dell'elaborazione. Inoltre replicare il middleware dà la possibilità di gestire un numero maggiore di sensori/sorgenti dati, con conseguente aumento della capacità di migliorare la precisione di misurazione o estendere il limite massimo della dimensione spaziale del fenomeno monitorato. Infine vediamo che la replicazione può essere utilizzata anche per fornire risultati più attendibili, ad esempio nel caso in cui l'ambiente operativo sia particolarmente ostico, o per proprietà intrinseche del fenomeno monitorato o a causa di possibili interferenze ambientali che possono portare alla ricezione, presso il software di elaborazione, di dati non corrispondenti alla realtà (si pensi ad esempio ad uno scenario in cui si hanno sensori mobili collegati con il resto del sistema attraverso rete wireless che ad un certo punto si vengono a trovare in un'area interessata da campi elettromagnetici che possono interferire con le comunicazioni).
Gli obiettivi del nostro lavoro sono quindi stati la progettazione, lo

sviluppo e la valutazione di un sistema middleware fault tolerant multi tier per reti di sensori. La scelta di una realizzazione di tipo multi tier è stata fatta in seguito alla necessità di specializzare l'architettura per l'ambiente rfid. Quest'ultimo ha una caratteristica molto particolare: solitamente la frequenza di produzione dei dati dello strato formato dai readers è molto più elevata di quella del middleware. Il compito principale di quest'ultimo è infatti quello del filtraggio delle informazioni provenienti dai readers, e della propagazione verso l'utente o altre applicazioni client dei soli dati ritenuti significativi, che sono di solito quelli che indicano delle variazioni nel fenomeno monitorato. Più lente avvengono tali variazioni, più è alto il divario tra le due frequenze. Un altro importante aspetto che deve essere considerato è quello che, a causa di vincoli hardware, le sorgenti di dati potrebbero essere in grado di attivare comunicazioni solo su scala locale, come ad esempio nel caso dei nodi di una wireless sensor networks i quali comunicano utilizzando protocolli "wi-fi" tipo 802.11. Guidati da tali caratteristiche e dalla riflessione che la prossimità spaziale tra entità produttrici di dati ed entità filtranti permette di ottenere migliori prestazioni con costi più bassi (la vicinanza spaziale consente di connettere le varie entità attraverso un'economica LAN ad esempio di tipo ethernet), si è deciso di suddividere il middleware in due tier principali: il primo, che riferiremo con il nome di LOW LEVEL MIDDLEWARE, che effettua il filtraggio vero e proprio delle informazioni, localizzato in prossimità delle sorgenti dei dati, il secondo, riferito con il nome di HIGH LEVEL MIDDLEWARE, con il compito di fare da front end tra il sistema e gli utenti/applicazioni client, non necessariamente localizzato in prossimità del primo tier. Una tale struttura, oltre a permettere l'ottimizzazione del sistema in funzione della differenza tra le due frequenze di generazione di dati a cui abbiamo appena accennato, fornisce un numero più elevato di gradi di libertà in fase di progettazione, replicazione e realizzazione del sistema che risulterà quindi essere meglio modellabile ed adattabile alle esigenze operative.
Nel progettare e sviluppare il middleware ci siamo serviti di vari strumenti, primo tra tutti il framework Appia, descritto in dettaglio nel capitolo 4, il quale mette a disposizione una serie di implementazioni di astrazioni per la programmazione distribuita, come ad esempio atomic broadcast o astrazioni per la group membership. Il middleware realizzato risulta essere molto flessibile e largamente estendibile, è in grado infatti di operare con differenti tecniche di replicazione ed inoltre permette di essere esteso con nuove logiche di filtraggio e attraverso l'introduzione di semplici driver è in grado di supportare un elevato numero di periferiche per la produzione dei dati. L'introduzione di una nuova logica di filtraggio o di un driver per l'interazione con una nuova periferica richiede semplicemente l'aggiunta di poche classi java, sfruttando i meccanismi legati all'ereditarietà e all'uso delle intefacce per garantire la compatibilità con il middleware, e la manipolazione di opportuni file di configurazione, necessari al software per capire con quali filtri e con quali periferiche si trova ad operare. Una descrizione dettagliata del modello di programmazione verrà data nel capitolo 5. Durante il nostro lavoro sono state analizzate e valutate varie tecniche di replicazione, e ci siamo concentrati solo sugli aspetti riguardanti il LLM (Low level middleware). La replicazione del tier HLM (High level middleware) può essere spunto per

sviluppi futuri. E' stato realizzato un prototipo in grado di lavorare con differenti schemi, tra cui naturalmente troviamo quelli classici: Active Replication e Primary-Backup. Proprio per questa sua caratteristica tale prototipo costituisce una piattaforma/framework per lo sviluppo e valutazione di schemi di replicazione ad hoc, consentendo anche di misurare le differenze prestazionali tra le varie tecniche, e permettendo quindi di determinare in maniera oggettiva quella che meglio si adatta ad uno specifico ambiente operativo.
Non a caso il risultato principale del nostro lavoro è stato l'ideazione di un protocollo innovativo per la replicazione software, il quale sfrutta la silenziosità Finite State Machine (FSM) che modellano le regole di filtraggio e che sono implementate all'interno delle repliche che formano il LLM, e che si appoggia su un tier differente, l'HLM, per svolgere le operazioni di coordinazione necessarie per mantenere la consistenza dello stato del sistema. Riusciamo in questo modo ad evitare l'utilizzo di implementazioni di astrazioni di coordinazione classiche che solitamente hanno un costo temporale elevato. Come vedremo in dettaglio nel capitolo 6, l'aumento di prestazioni ottenibile con il nostro protocollo ha però degli effetti collaterali: ad una diminuzione del tempo di risposta corrisponde un aumento della probabilità di propagare all'utente/applicazione client dati che non corrispondono perfettamente alla situazione reale del fenomeno monitorato.
Le misure di performance effettuate hanno evidenziato i miglioramenti apportati dall'utilizzo del nuovo protocollo. Questo è stato confrontato con tecniche di replicazione classiche basate su atomic broadcast. I risultati dei test mostrano che, con l'aumentare del valor medio del rapporto δ tra il numero di messaggi in ingresso e il numero di eventi in uscita, il divario in termini di latenza di risposta e di throughput del sistema cresce notevolmente a favore della nostra soluzione: per valori di δ prossimi ad uno le tecniche basate su atomic broadcast permettono di ottenere prestazioni migliori, il nostro protocollo risente infatti del costo dei numerosi rollback effettuati dalle repliche di LLM in caso di carico elevato. Man mano che il valore di δ cresce, come ad esempio per $\delta = 10$, questo costo assume un peso via via sempre minore e i test hanno mostrato che la nostra soluzione garantisce prestazioni sensibilmente migliori rispetto a quelle basate su atomic broadcast, garantendo limiti di saturazione molto più elevati rispetto a queste ultime.
La tesi risulta essere così strutturata: nel capitolo 2 viene data una descrizione approfondita della tecnologia rfid e vengono illustrati alcuni casi di studio relativi al suo utilizzo; nel capitolo 3 vengono presentate una serie di astrazioni per sistemi distribuiti; nel capitolo 4 vengono descritti una serie di strumenti/framework utilizzate durante il processo di progettazione e sviluppo del middleware; il capitolo 5 fornisce una descrizione dettagliata dell'architettura di quest'ultimo e descrive il suo modello di programmazione; nel capitolo 6 vengono studiate le possibili tecniche di replicazione basate su active replication utilizzabili per rendere il middleware fault tolerant e viene illustrato approfonditamente un protocollo innovativo Post Synchronized Active Replication Protocol, per la sincronizzazione dello stato delle repliche; infine nel capitolo 7 forniamo una serie di conclusioni basate sui risultati sperimentali.

2. RFID: tecnologia ed applicazioni

2.1 Che cos'è l'RFID

Con l'acronimo RFID (Radio Frequency Identification), si intendono un insieme di tecnologie basate sulla identificazione tramite utilizzo di radio frequenza che consentono di realizzare applicazioni in aree molto diverse fra loro. L'RFID sostituisce il codice a barre introducendo importanti funzionalità e requisiti di sicurezza e performance. Il *tag* (componente elettronico) contenuto all'interno di quella che potremmo definire "etichetta intelligente", o "memoria mobile" è in grado di tenere traccia dell'intera storia del prodotto su cui è stato posto, inoltre non richiede alcun tipo di manutenzione o di fonte d'energia.

2.2 Funzionamento

L'elemento principale di un sistema RFID è un dispositivo chiamato *transponder* (o *tag*), che può essere collegato in qualsiasi modo ad un oggetto. Un lettore (*reader*) statico o portatile manda un segnale tramite un campo elettromagnetico generato attraverso un'antenna. Il segnale permette di caricare in brevissimo tempo (qualche millesimo di secondo) i componenti interni che costituiscono il circuito di alimentazione del transponder. Quest'ultimo, una volta riconosciuta la correttezza dell'operazione di interrogazione, manda verso il lettore un segnale che contiene il suo codice di identificazione nonché altri dati contenuti all'interno della sua memoria. I sistemi RFID fanno parte della cosiddetta tecnologia *Auto-ID*, o identificazione automatica, che fa riferimento a un sistema che consente:

- acquisizione di dati per l'identificazione

- introduzione di questi dati e di altri complementari all'interno di programmi presenti in un computer

I vantaggi dell'identificazione automatica sono principalmente:

- l'eliminazione di errori dovuti all'inserimento manuale dei dati
- tempi e quindi costi inerenti a tale operazione manuale.

2.3 RFID vs Codice a barre

L'RFID rappresenta una rivoluzione rispetto al codice a barre. La tecnologia che supporta i primi è completamente differente rispetto a quella dei secondi e offre dei vantaggi rispetto a questi ultimi che sono così riassumibili:

- identificazione univoca di ciascun oggetto
- nessun contatto fisico o visivo per la lettura/scrittura
- intervento umano limitato
- possibilità di utilizzo di differenti posizioni e distanze di lettura, da centimetri a diversi metri (in base alla frequenza e al tipo di etichetta)
- resistenza in ambienti difficili (alte temperature, umidità, sporco, ...)
- lettura simultanea di più tag
- lettura e scrittura reiterabili sui tag
- supporto volume di dati enormemente maggiore
- possibilità si implementazione ed utilizzo di metodi di crittografia e autenticazione
- possibilità di disattivazione (privacy)
- possibilità di realizzare tag di varie forme e dimensioni per adattarsi alle caratteristiche dell'oggetto su cui deve essere applicato e del contesto di utilizzo
- possibilità di recupero a processo esaurito
- possibilità di tacciare la “storia” del prodotto, permettendo di individuare i soggetti che lo hanno manipolato ed i luoghi in cui

ha stazionato.

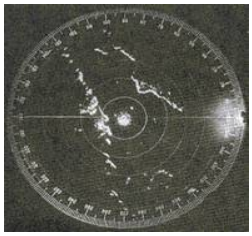
Di seguito è riportata una tabella che confronta contrappone alcune caratteristiche dei codici a barre con quelle degli RFID

Codici a barre	**RFID**
solo lettura	Lettura/scrittura
Contatto lettura “visivo”	Nessuna necessità visiva
Sensibili ad agenti meccanici o ambientali a meno di un'efficace protezione	Maggiore insensibilità agli agenti meccanici ed ambientali (totale ermeticità)
Lettura in movimento, apparati costosi	Nessun particolare apparato per lettura/scrittura in movimento
Estremamente sensibile ad alterazioni ottiche, abrasioni, macchie	Insensibile a sporco, macchie ecc.... Totalmente ermetico
Impossibilità di leggere contemporaneamente più codici	simultaneità di lettura di più codici
costo bassissimo	Costo maggiore di circa 40 volte
Tabella 2.1 – Confronto tra rfid e codice a barre	

2.4 Breve storia dell'RFID

I principi funzionali della tecnologia RFID (*Radio Frequency Identification*) derivano direttamente dallo sviluppo e dall'utilizzo dell'IFF (*Identification Friend or Foe* - Identificazione Amico o Nemico), del radar e del transponder. Questa tecnologia è nata durante la seconda guerra mondiale in concomitanza con i primi radar (radio detecting and ranging, rilevamento radio e misurazione di distanze). Questi non erano sofisticati e tecnologici come quelli moderni, anzi erano abbastanza artigianali. In pratica erano costituiti da:

- un'antenna di trasmissione fortemente direzionale (di forma paraboloide) che emetteva una serie di impulsi radio,
- un impianto di ricezione (che sfrutta la stessa antenna) montato su un piano rotante,
- un sistema di amplificazione,
- un primitivo schermo.



Il principio di funzionamento del radar consiste nell'inviare verso

l'oggetto cercato radioonde generalmente modulate a impulsi e nel ricevere le onde riflesse dall'oggetto medesimo (echi radar). Calcolando il tempo di eco, ossia il rimbalzo dell'impulso sulla carlinga dell'aereo, e conoscendo la posizione istantanea della rotazione dell'antenna ricevente, il sistema di amplificazione permetteva la visualizzazione di un punto sullo schermo, cioè dell'aereo. Il ministero della difesa britannico non ritenne completamente soddisfacenti i primi sistemi radar, in quanto non avrebbero dovuto solo avvistare gli aerei nemici, ma anche differenziali da quelli gli amici, così da ottenere la situazione in tempo reale delle battaglie aeree. La difesa britannica quindi ordinò la progettazione di un sistema IFF-Identification Friend or Foe (Identificazione amico o nemico). Gli ingegneri decisero allora di dotare gli aviogetti inglesi (o alleati in seguito) di una scatola contenente una ricetrasmittente, denominata successivamente "transponder", che all'atto dell'illuminazione radar (vale a dire, quando il fascio di radioonde colpiva l'aereo) rispondesse sulla stessa frequenza istantaneamente con un "bip" che amplificato permise al radar stesso l'identificazione degli aviogetti amici rispetto ai nemici. Successivamente, con l'evoluzione tecnologica, questi sistemi sono divenuti sempre più precisi. L'evoluzione successiva infatti fu non solo l'identificazione IFF ma l'identificazione univoca dell'aviogetto mediante un ID assegnato. Questo fu possibile modulando l'emissione del transponder (ecco i primi esperimenti di onde radio FM) a bordo dell'aereo, che non inviava più un semplice "bip", ma una serie opportunamente codificata: ciò permise di "numerare" gli aviogetti e conoscerne così la posizione univoca. Verso la fine degli anni '60 ha inizio la commercializzazione dei primi sistemi EAS (*Electronic Article Surveillance*). Questi primordiali sistemi RFID utilizzavano generalmente un transponder che gestiva un'informazione di 1 bit per poter permettere il rilevamento della presenza/assenza del transponder. L'EAS garantiva una elementare funzione antitaccheggio nei supermercati che si stavano affermando nei paesi sviluppati. L'EAS può quindi essere considerata il primo vero caso di effettivo utilizzo di massa della tecnologia RFID in attività non militari. Nei anni '70 si poté assistere all'investimento da parte di grandi industrie americane del settore militare in applicazioni RFID civili. General Electric, Westinghouse, Philips, Glenayre realizzarono la prima applicazione dell'RFID dedicata al controllo di oggetti e di mezzi in movimento. Gli anni '80 furono il periodo che vide l'affermarsi di quella degli RFID come una tecnologia completa, con la diffusione su scala mondiale. Negli Stati Uniti gli interessi puntarono sul controllo delle merci trasportate, sui mezzi di trasporto, l'accesso del personale e l'identificazione di animali. In Europa le materie più sviluppate, oltre a quest'ultima, furono le applicazioni per attività industriali e il controllo/accesso alle autostrade. Solo negli anni '90 si delinearono le condizioni per lo sviluppo dell'RFID moderno: maggiore miniaturizzazione dei circuiti, consentendo così un notevole risparmio d'energia, e sviluppo di standard internazionali condivisi. Avendo raggiunto tali requisiti, l'RFID può diventare a pieno titolo una tecnologia pervasiva.

2.5 Sicurezza

La comunicazione tra reader e transponder può essere garantita secondo modalità differenti in funzione del grado di sicurezza richiesto. Un primo criterio è la sicurezza nell'accesso della comunicazione che si può ottenere mediante una password che permette di identificare il reader e il transponder. La password di identificazione del reader può essere unica per tutti i transponder o specifica per transponder. Un sistema più sofisticato è la codifica delle trasmissioni attraverso password. Questa soluzione rende molto difficile la lettura del processo di comunicazione ai non esperti. Da ultimo si possono utilizzare veri e propri sistemi di crittografia che permettono di ottenere un'elevata sicurezza al prezzo di costi crescenti e prestazioni più modeste. Inoltre l'utilizzo della crittografia è limitato dalle leggi nazionali sulla sicurezza pubblica che impongono limitazioni agli algoritmi e la possibilità di essere comunque letti dalle forze di Polizia su richiesta delle autorità competenti. Alcune applicazioni RFID richiedono una elevata sicurezza del processo di comunicazione; per esempio le carte di prossimità per i pagamenti. In questo caso è stata prevista una procedura mutuale di identificazione tra reader e transponder che è stata normalizzata da ISO 9798-2.

2.6 Esposizione umana ai campi elettromagnetici

I sistemi RFID accoppiati magneticamente sono tra i più utilizzati nel mondo: questi transponder sono generalmente di tipo passivo ovvero sono alimentati dal reader stesso. La frequenza di 13,56 MHz rappresenta uno standard mondiale effettivamente disponibile. L'elemento critico di un sistema RFID è l'antenna, che deve andare in risonanza con la portante per ottimizzare il trasferimento d'energia e il trasferimento dei dati. Le dimensioni dell'antenna però non possono essere fisicamente paragonabili alla lunghezza d'onda: si utilizzano quindi antenne più piccole il cui raggio ottimale è 1,414 volte la distanza di lettura richiesta. Inoltre le intensità dei campi elettrici e magnetici sono limitati dalle norme vigenti che riducono i raggi d'antenna possibili. La normativa dà indicazioni precise che riguardano le possibili interferenze con altri dispositivi radio e gli assorbimenti (SAR) da parte del corpo umano. Il rispetto di queste norme permette l'omologazione degli apparati RFID. Una delle più importanti organizzazioni che si occupano dell'esposizione umana alle radiazioni non ionizzanti è la ICNIRP (*International Commission on Non-Ionizing Radiation Protection*).

2.7 Tecnologia

Un sistema RFID, *Radio Frequency Identification*, è composto da almeno tre elementi:

- TRANSPONDER
- READER
- SOFTWARE

Il *transponder*, come abbiamo già visto, è un ricetrasmettitore che invia un segnale radio in risposta a un comando ricevuto da una stazione remota. Il *reader*, letteralmente "lettore", ha la capacità d'interrogare individualmente i transponder, inviare e ricevere dati e interfacciarsi con i sistemi informativi esistenti. Il reader è composto da due parti:

- *l'unità di controllo,* che è un microcalcolatore con un sistema operativo in tempo reale
- *le antenne*, che sono interfacce fisiche tra l'unità di controllo e i transponder.

I transponder per essere attivati devono entrare nel campo magnetico generato da un'antenna che in questo modo ha la possibilità di alimentarli e di comunicare con loro. Il *software* è l'elemento indispensabile per la ricezione ed elaborazione in azienda delle grandi quantità di dati fornite dal sistema RFID.

2.7.1 Transponder

I transponder si differenziano secondo le seguenti caratteristiche:

- Chipless, chip
- Passivi, semipassivi, attivi
- Solo lettura, lettura/scrittura

Chipless/Chip: I transponder possono essere dotati o meno di chip, a seconda delle necessità di memorizzazione di informazioni complesse (il prezzo del tag evidentemente è diverso in funzione della complessità della funzione espletata). I transponder *chipless*, che non richiedono chip in silicio, sono indicati per applicazioni che hanno il solo scopo di creare un'etichetta per un'identificazione univoca del singolo prodotto fisico (segnale presenza/assenza). L'esempio migliore sono i prodotti di largo consumo.

Transponder passivi/semipassivi/attivi: Questa catalogazione

esprime le modalità di alimentazione e di trasmissione rispetto al reader.

- *transponder passivi*: sono alimentati dall'onda radio propagata dall'antenna del reader quando questo li interroga. Essi comunicano a distanze inferiori ai 10 metri.

- *transponder semipassivi*: hanno una fonte di alimentazione indipendente dal reader ma trasmettono solo se vengono interrogati dallo stesso. Essi comunicano a distanze di decine di metri. La fonte di alimentazione è utilizzata solo per le operazioni di trasmissione.

- *transponder attivi*: hanno una fonte di alimentazione indipendente dal reader e la capacità di trasmettere senza essere interrogati dal ques'ultimo. Essi comunicano a distanze dell'ordine di chilometri.

Sola lettura/lettura-scrittura: I transponder possono contenere componenti che hanno una memoria di sola lettura o una in cui si può scrivere una volta o più volte. Questo aspetto verrà approfondito nel prossimo paragrafo.

2.7.2 Le memorie dei transponder

Le memorie dei transponder sono: memoria di sola lettura ovvero **ROM** (*Read Only Memory*), memorie in cui si può scrivere e leggere senza limitazioni ovvero **RAM** ed **EEROM**, memoria di tipo intermedio ovvero **WORM** (*Write Once Read Memory*).

- **ROM** (Read Only Memory): Sono memorie di sola lettura poco costose con una vita utile molto lunga. Normalmente sono configurate dal produttore del componente con un numero limitato di informazioni, tra le quali non può mancare il codice univoco d'identificazione del tag.

- **RAM** (Random Access Memory): E' un tipo di memoria in cui si può scrivere e leggere senza limitazioni. Può raggiungere grandi densità di dati memorizzati a costi molto competitivi; l'unico limite è che necessita di una fonte di energia permanente per mantenere i dati in memoria, aspetto questo assolutamente non trascurabile nell'ambito RFID

- **EEROM** (Electrically Erasable Read Only Memory): Memoria in cui si può scrivere e leggere senza limitazioni, come la RAM, ma a differenza di questa ha il grande vantaggio di richiedere la presenza di una fonte di energia soltanto durante l'operazione di lettura e scrittura in memoria. Questo tipo di memoria può mantenere i dati senza alcuna alimentazione per almeno dieci

anni.

- **WORM** (Write Once Read Memory): E' un tipo di memoria intermedio che permette di scrivere una sola volta senza poterne successivamente cancellare il contenuto. Questa funzionalità è utile perché permette all'utente di personalizzare il tag senza dover richiederlo al produttore di chip. Una volta programmata la memoria WORM si comporta come una memoria ROM.

Le memorie EEROM sono le soluzioni più adeguate per transponder passivi, nonostante il maggiore costo e la minore densità di memorizzazione.

2.7.3 Reader

Il reader è il *lettore* dei dati inviati dal transponder. Esso interpreta individualmente i transponder, invia e riceve dati ed è in grado di interfacciarsi con i sistemi hardware/software esistenti. Per queste sue funzioni il reader è facilmente equiparabile a un canale di comunicazione fra i transponder ed il mondo esterno. E' composto per lo più da due elementi:

- unità di controllo
- antenne

L'*unità di controllo* è un microcalcolatore real-time che consente di gestire le interfacce con le antenne (fino a otto) e l'interrogazione dei transponder che entrano nel campo d'azione di un'antenna. L'unità di controllo, però, è anche in grado di gestire le collisioni fra i messaggi di risposta dei transponder e l'interfacciamento con i sistemi informativi aziendali. Le *antenne* sono le reali interfacce fisiche fra i transponder e l'unità di controllo.

2.7.4 Frequenze

Le applicazioni RFID utilizzano tre grandi bande di frequenza, che corrispondono a tecnologie di comunicazione molto differenti fra di loro:

- fino a 40,68 MHz (basse, medie ed alte frequenze)
- da 433 a 915 MHz (altissime frequenze)
- da 2,45 Ghz (micro-onde)

Le diverse bande di frequenza presentano caratteristiche diverse e sono quindi indicate per applicazioni differenti. In generale, al

crescere della frequenza crescono la distanza di lettura e la quantità di informazioni che si possono trasferire nell'unità di tempo, diminuiscono la capacità di resistenza a condizioni operative avverse e i costi. I tag a bassa frequenza utilizzano poca potenza, sono capaci di attraversare materiali non metallici e liquidi, ma il segnale per la lettura non supera i 30-40 centimetri. Le etichette ad alta frequenza lavorano meglio con oggetti metallici e arrivano a coprire una distanza di circa un metro. Le altissime frequenze offrono range di lettura più ampi e permettono di trasferire i dati velocemente, ma non attraversano facilmente i materiali.

Fino a 40,68 MHz: Nelle applicazioni RFID che utilizzano questa banda di frequenza i transponder sono accoppiati magneticamente con i reader. Le frequenze più utilizzate in questa banda sono:

- 13,56 MHz, è sicuramente la frequenza più utilizzata ad oggi per i progetti RFID, tanto da divenire uno standard. Adatta a transponder passivi a basso coso e a media/alta velocità, consente di trasmettere in media 106 Kbit/sec, inoltre è la sola frequenza ad essere standardizzata da tutti gli enti formatori mondiali. Sono state realizzate sia applicazioni su persona che su prodotti.

- 6,78 MHz, simile al 13,56 MHz, ma non utilizzabile in tutto il mondo

- 125-135 kHz, per lo più utilizzati per l'identificazione degli animali

Da 433 MHz a 915 MHz: In questo caso i transponder sono accoppiati elettricamente con i reader. Le frequenze più utilizzate sono:

- 433 MHz, per applicazioni su veicoli (ad esempio nel campo ferroviario).

- 915 MHz, per l'identificazione di oggetti in movimento (esempio il Telepass)

Da 2,45 GHz: Anche qui i transponder sono accoppiati elettricamente con i reader. La frequenza più utilizzata è proprio la 2,45 GHz, particolarmente adatta per l'identificazione di oggetti in movimento a velocità elevata, come ad esempio gli aerei.

I tag 125 kHz 13,56 MHz sono previsti dalle norme ISO come passivi (senza batterie) mentre per i tag RFID UHF e Ultrawide band esistono di tipo attivo, semi-attivo e passivo.

2.7.5 Software

Le etichette RFID sono in grado di fornire un livello di informazioni quantitativamente e qualitativamente impensabile ai sistemi gestionali come gli ERP o i WMS. Per collegare le etichette a questi sistemi si rende per lo più necessario un nuovo livello di applicazioni che va sotto il nome di *middleware software*.

Le principali funzioni svolte di questo strato di software sono così riassumibili:

- *Device and network management*: consente l'installazione e la configurazione degli apparati e la gestione di eventuali errori
- *Event management*: permette di gestire le regole di business associate alle attività di track and trace
- *Security*: per gestire e controllare gli accessi al sistema

2.8 Applicazioni sui prodotti

Lo scopo primario della supply chain è assicurare l'incontro tra la domanda del consumatore e offerta, mantenendo livelli di alta qualità e basso costo del prodotto e lead time minimi. Questo significa assicurare rapidità di risposta al mercato, conoscere e comprendere le necessità del cliente e controllare i processi produttivi coordinando gli obiettivi di tutti gli attori della filiera. In altre parole, il flusso di materiali da una parte deve necessariamente andare di pari passo con il flusso delle informazioni sul prodotto (dal consumatore finale fino a risalire all'inizio dell'intera supply chain), che devono essere quanto più di dettaglio, in tempo reale e accurate possibili. Per ottenere ciò è necessario un sistema che catturi automaticamente i dati e le transazioni di base, li coordini e traduca in informazioni utili all'azienda, rendendoli disponibili immediatamente. Indubbiamente la tecnologia RFID sembra essere la candidata ideale per raccogliere questa sfida. Di seguito verranno esaminati i vantaggi delle applicazioni della tecnologia RFID nell'ambito della supply chain. A grandi linee, i primi driver discriminanti che determinano la rilevanza di una specifica applicazione possono essere individuati nel binomio volumi - valore del prodotto. Ovvero, se da una parte le grandi movimentazioni rendono particolarmente interessanti soluzioni in grado, per esempio di automatizzare i processi di ricezione, picking e spedizione merce, dall'altra parte il valore del prodotto porta a focalizzare l'attenzione su applicazioni che ne assicurino la protezione dal furto così come l'autenticità, o che siano in grado di differenziare il prodotto per immagine e innovazione.

2.8.1 Applicazioni su prodotti di largo consumo

Nel mondo dei Beni di Largo Consumo, i cosiddetti *Fast Moving Consumer Goods* (FMCG), siamo in presenza della classica situazione di consistenti volumi prodotti e movimentati, con il conseguente focus di applicazioni che hanno un impatto sullo snellimento delle attività nell'ambito della logistica e del retail e sulla velocizzazione dei processi di repleinshment e reordering. I principali benefici individuabili nell'adozione della tecnologia RFID per i beni di largo consumo sono:

- disponibilità del prodotto a scaffale
- automazione della verifica dell'esito di spedizione e delle attività di controllo della merce in entrata e in uscita
- inventario accurato e in tempo reale
- riduzione delle scorte
- riduzione delle differenze inventariali lungo la catena
- blocco prodotto in caso di ritiro merce (conseguenza diretta della tracciabilità del prodotto)

2.8.2 Applicazioni su prodotti Moda

Per 'Moda' si intende qualunque tipo di prodotto - abbigliamento, pelletteria, orologi/gioiello, profumi e cosmetici, accessori - soggetto a diffusione e popolarità in un determinato contesto. I prodotti che ricadono in questo contesto sono caratterizzati da un continuo rinnovamento, sebbene esistano alcuni prodotti della moda divenuti continuativi perché percepiti come "classici" e quindi richiesti continuamente. Per semplicità, ci concentreremo sul settore del tessile/abbigliamento, le cui caratteristiche di filiera - con le opportune modifiche - possono essere generalizzabili a tutte le altre filiere del settore Moda. In linea del tutto generale si può notare che per quanto concerne l'attraversamento della filiera, i transponder, grazie alle loro caratteristiche tecniche assicurano la visibilità del prodotto e risultano fondamentali nell'accorciamento dei tempi necessari a diverse operazioni in ambito sillogistico della **gestione del ciclo di produzione** del capo, sia presso il confezionista, il centro distribuzione e il dettagliante.

I benefici apportati dall'RFID sono:

- semplificazione e riduzione dei tempi nel processo di picking
- gestione simultanea di più ordini

- accuratezza nell'evasione degli ordini
- agevolazione delle modalità di pre/post allocazione
- riduzione degli errori e velocizzazione del processo di spedizione
- controllo carico e sequenza di caricamento camion
- visibilità e tracciabilità del materiale in arrivo
- verifica immediata e automatica del materiale in entrata
- aggiornamento automatico e in tempo reale dell'inventario
- velocizzazione e semplificazione del processo di stoccaggio
- riduzione delle differenze inventariali

Per quanto riguarda invece la presenza di **etichette RFID sui capi presso i negozi**, i vantaggi sono individuabili in:

- conoscenza del momento di esposizione del prodotto nel negozio
- verifica della disponibilità del prodotto
- riordino immediato dei capi esauriti
- accoglienza cliente/proposte personalizzate e basate su merce disponibile
- totale automazione della cassa
- controllo dei capi in prova
- raccolta delle statistiche capi provati vs. acquistati vs. acquistati senza prova
- antitaccheggio

Inoltre la presenza di etichette RFID sui prodotti moda consente:

- controllo dell'autenticità dei capi e import parallelo
- verifica dei capi acquistati vs. capi esposti

2.8.3 Applicazioni su beni durevoli

Nei beni durevoli rientrano quei prodotti che non sono consumati velocemente, ma il cui utilizzo si protrae negli anni: elettrodomestici bianchi (frigoriferi, lavatrici, lavastoviglie, ecc...),

elettrodomestici bruni (televisori, lettori DVD, impianti stereo ecc..), automobili, mobili e arredi per la casa, piccoli elettrodomestici. A seconda delle categorie considerate, le applicazioni RFID sono diverse. In particolare, per i beni elettrici/elettronici, i campi di applicazione di maggior interesse sono:

- sistema logistico produttivo
- manutenzione e controllo post-vendita
- certificazione del collaudo
- gestione delle reverse logistics
- antifurto/autenticazione

2.8.4 Applicazioni su prodotti Freschi

Fra i beni di Largo Consumo, i freschi meritano un approfondimento per la peculiarità della filiera, caratterizzata fondamentalmente da una struttura estremamente semplificata al fine di contenere al massimo i tempi di attraversamento. Appartengono ai freschi tutti i prodotti caratterizzati da una vita molto breve dal punto di vista del consumatore, quindi latte, fiori recisi, vino, ma anche CD, DVD, film a noleggio e il circuito dei film nelle sale, trattandosi in questi ultimi casi di vita breve del prodotto sul mercato, più che di deperibilità. Le caratteristiche di questi prodotti pongono l'accento sulla rapidità e velocità di esecuzione sia in termini di tempi di produzione, sia di risposta del mercato, al fine di evitare stock out e deperimento, in senso allargato. La supply chain di questa tipologia di prodotto è quindi, come dicevamo, caratterizzata da una notevole semplicità, per cui il passaggio delle merci può avvenire direttamente del produttore al distributore, oppure attraverso un transit point (v. corrieri). Le applicazioni RFID più interessanti in questo caso sono quelle che puntano all'**ottimizzazione dei tempi di attraversamento della filiera** e alla gestione della shelf life (vita di scaffale del prodotto fresco) e della qualità del prodotto.

2.8.5 Applicazioni su animali

La radioidentificazione degli animali ha inizio negli anni '70 quando l'Università di Chicago definì un primo protocollo per l'identificazione dei ruminanti e il controllo remoto di caratteristiche fisiche come la temperatura. La società IDX sviluppò quindi un transponder ad alta frequenza rispondente a queste richieste, che può essere ancora oggi considerato attuale per molti aspetti. Dopo vent'anni di ricerca e di vari tentativi di applicazione disparati per l'identificazione degli animali tra cui pinguini, ostriche e salmoni, nel

campo dell'RFID sono rimasti unicamente i sistemi compatibili con le norme internazionali ISO 11784/11785 a bassa frequenza. Secondo tali norme il transponder, inserito nell'animale sotto cute o ingerito con il cibo devo rispondere a queste caratteristiche:

- essere ricoperto da materiale bio-compatibile resistente a impatti e urti
- avere un normale range di efficienza di un metro, anche quando l'animale si sposta a una velocità massima di 40 km/ora
- essere facile da impiantare; una volta inserito non deve migrare dalla sua posizione iniziale, e dopo la morte dell'animale deve essere facilmente rintracciabile per essere tolto
- essere di sola lettura in modo da non consentire la modifica dei dati presenti
- sia transponder che lettore, statico o portatile, devono essere compatibili con le norme ISO.

La Comunità Europea, mossa dalla pressante esigenza di identificare elettronicamente gli animali di allevamento e in particolare i ruminanti, si è espressa con la risoluzione CE n. 820/97, che richiede la completa tracciabilità dell'animale di allevamento dal momento della nascita fino al consumo finale delle carni e derivati. Questo piano ha altresì definito una serie di caratteristiche di cui devono essere dotati i sistemi di identificazione degli animali in materia di efficienza e di gestione in tempo reale dell'informazione.

2.8.6 Applicazioni RFID sulla persona

L'utilizzo della tecnologia RFID per identificare gli esseri umani pone problemi importanti ma offre opportunità maggiori rispetto alle applicazioni dei transponder sui prodotti. Innanzitutto si pone il problema del livello di intrusività accettabile per la privacy dell'individuo, che riguarda non tanto la registrazione dei dati personali, quanto i modi di utilizzo e la correlazione delle registrazioni. Poiché il transponder può permettere un'identificazione sicura e automatica del suo portatore, possiamo immaginare la possibilità di farlo interagire con il comportamento della persona. Un altro risvolto particolarmente interessante delle applicazioni RFID sulla persona, impossibile sui prodotti, è la possibilità di comunicare informazioni da e verso la persona stessa. Poiché tuttavia gli esseri umani non sono ancora neurologicamente connessi ai circuiti elettrici dei transponder, dobbiamo considerare qualche strumento elettronico d'interfaccia tra il transponder e gli umani stessi. Le soluzioni possibili sono differenti in funzione dell'ambito di applicazione. Per esempio, un operatore logistico di magazzino può già leggere e scrivere su un transponder

RFID di un pallet mediante un palmare PSION che fornisce tutti i servizi di comunicazione ed elaborazione necessari. Un'altra possibilità ampiamente utilizzata è quella di impiegare alcuni strumenti di comunicazione 'proprietari' che indichino le informazioni richieste, come nel caso di transponder utilizzati per la gestione degli accessi degli ospiti di una grande azienda, di un ospedale o di un aeroporto. In funzione del profilo dell'ospite e del suo piano di visita, è possibile programmare il transponder in modo coerente. Esso può attivare al passaggio del suo portatore indicazioni visive o messaggi acustici che indichino il percorso da seguire e l'abilitazione o l'interdizione ad entrare in alcune aree specifiche. Tutte queste soluzioni sono però specifiche del mondo professionale e non si adattano a realizzare l'interfaccia di massa tra il transponder e i comuni esseri umani. Un sistema del genere deve avere necessariamente caratteristiche diverse: un'interfaccia standard, facile da utilizzare e ampiamente diffondibile, costo addizionale minimo o nullo, pervasività. Lo strumento elettronico che ha tutte queste caratteristiche è uno solo: il *telefono cellulare*. A tale proposito sono già in corso alcuni progetti tra i principali operatori telefonici e i principali produttori di telefoni cellulari per sviluppare questa funzione d'interfaccia.

2.8.7 Altri esempi di applicazioni

Controllo integrità: Un container potrebbe essere sigillato con un Transponder che cessa di funzionare se il container viene aperto. In questo modo una gru che prelevi da un treno o da una nave un container, senza alcun bisogno di ispezione visiva umana, è in grado di determinare se il container è stato aperto e quindi, richiedendo una nuova ispezione doganale, deve essere depositato un un'area di ispezione specifica. Il numero di operazioni effettuate dalle gru, in questo modo, si dimezza.

Montaggio Meccanico: L'identificatore delle componenti da serrare con degli avvitatori controllati da computer in una linea di montaggio può essere effettuata direttamente dall'avvitatore e non da un lettore di codice a barre riducendo il tempo necessario agli operatori.

Controllo accessi: dall'auto all'ufficio, in modo trasparente. E' possibile stabilire quali varchi possono essere attraversati dal singolo individuo, quante volte, a distanza di quanto tempo, ecc... Una persona che entra in una stanza può essere riconosciuta e quando si avvicina ad un computer, questo, riconoscendolo, gli può presentare il suo ambiente di lavoro, senza dover richiedere un codice di identificazione.

Facility Management: Dalle ispezioni manutentive alle ronde di vigilanza. un metodo assai semplice per il controllo delle ronde di vigilanza è il notissimo bigliettino che queste lasciano presso gli

immobili che visitano. Questo meccanismo non consente di tracciare effettivamente il numero di visite compiute e i relativi orari, per poter assicurare il cliente che quanto stabilito nel contratto sia stato mantenuto. Se il luogo da visitare è attrezzato con un trasponder il personale di ronda può essere dotato di un lettore tascabile che registra l'ora di ogni effettiva visita, ottenendo quindi la certezza dell'attività svolta.

Sanità, controllo trasfusionale: Alcuni transponder possono rilevare anche pressione e temperatura. Una sacca di sangue per trasfusioni non può essere utilizzata se viene tenuta fuori da un idoneo frigorifero oltre una certa durata. In un transponder associato ad una sacca può essere scritto l'orario di estrazione da un frigorifero che può essere verificato in automatico prima della somministrazione al paziente.

Sanità, distribuzione di farmaci (“rischio clinico”): in ogni ospedale c'è un problema di identificazione dei pazienti e di somministrazione dei farmaci corretti, farmaci che vengono preparati prima del giro di corsia per ogni paziente presente. Attraverso un sistema di identificazione automatica è possibile gestire magazzino di approvvigionamento, identificazione del paziente, cartella clinica e prescrizione, e gestire così l'intero processo in maniera ciclica e controllata.

2.9 Casi di studio

2.9.1 Gli elettrodomestici intelligenti di Merloni

L'azienda Merloni è una delle maggiori imprese produttrici di elettrodomestici in Europa e nel mondo. La sua capacità di innovazione dei prodotti è riconosciuta da tutti gli operatori e analisti del mercato. Molti sono i casi di innovazione: da Margherita, la prima lavatrice intelligente, al sistema WRAP di connessione in rete degli elettrodomestici, al sistema Leonardo di interfaccia intelligente tra l'elettrodomestico e l'utente.

I progetti RFID di Merloni Elettrodomestici

Anche per la tecnologia RFID Merloni è stata la prima a muoversi verso una nuova generazione di prodotti che possono sfruttare questa tecnologia. Gli assi di sviluppo sono stati due:

- RFID per gestire il prodotto elettrodomestico
- elettrodomestici per gestire RFID sui prodotti

L'etichetta RFID su ogni elettrodomestico permette di gestire sia il

processo di produzione sia tutta l'attività post-vendita di assistenza e riparazione. D'altro canto gli elettrodomestici che gestiscono i prodotti RFID possono aggiungere nuove funzionalità: per esempio, la lavatrice Merloni con reader RFID può facilmente riconoscere i capi da lavare e ottimizzare il carico e la composizione, evitando incompatibilità di colore o di altre caratteristiche e utilizzando la minima quantità d'acqua, di energia e di detersivo. In modo analogo, il frigorifero Merloni con reader RFID può gestire automaticamente sia le date di validità dei prodotti sia i processi di riordino. Infone il forno a microonde ha un reader RFID che permette di rilevare le caratteristiche del cibo che si vuole riscaldare e quindi può suggerire un piano di cottura adeguato. Questo nuovi prodotti sono ovviamente dei concept sperimentali che hanno l'obiettivo di dimostrare la fattibilità dell'idea. In realtà il loro effettivo utilizzo dipende dal livello di diffusione dell'RFID nelle singole filiere.

2.9.2 Il progetto RFID del MIT: EPC

Il Massachusetts Institute of Technology fondò nel 1999 l'*Auto-ID Center*, con l'ambiziosa missione di realizzare un network intelligente in grado di individuare e tracciare in modalità wireless (via RFID) ogni singolo oggetto del mondo. L'istituto è nato sotto la spinta dei grandi produttori di beni di largo consumo (prima fra tutti la Procter & Gamble), che hanno quindi orientato l'attività ai beni di largo consumo.

Il progetto

Se il sistema RFID deve poter essere applicato anche a prodotti estremamente economici, è fondamentale che il costo delle etichette sia estremamente basso. Da qui nasce la necessità di "alleggerire" il più possibile le informazioni contenute a livello di singolo chip, per poter ridurre al minimo la complessità necessaria per ogni singolo tag, il cui costo non deve superare il centesimo. Per poter identificare univocamente ogni singolo oggetto, è necessaria la creazione di un codice unico, la nuova generazione del codice a barre, che sia in grado di riconoscere non più solo la categoria merceologica e il prodotto, ma il singolo pezzo. L'*Electronic Product Code*, **EPC**, è la risposta dell'Auto-ID Center a questa necessità. Il codice, per poter essere effettivamente universale, oltre a essere sufficientemente grande da poter enumerare miliardi di oggetti, deve essere riconosciuto e accettato a livello mondiale, e quindi essere uno standard internazionale. L'EPC sarà l'unica informazione contenuta nel tag e servirà quindi da identificativo dell'oggetto etichettato.

2.9.3 La gestione dei vagoni delle ferrovie svizzere

Ferrovie Federali Svizzere (SBB) è un'azienda pubblica di trasporto passeggeri e merci. Essa ha iniziato nel 1994 l'implementazione del sistema AVI (*Automatic Vehicle Identification*), basato su tecnologia RFID.

Il progetto AVI

Il progetto *Automatic Vehicle Identification* è stato realizzato utilizzando la tecnologia della società americana Transcore, che è leader mondiale del settore con più di 4 milioni di tag installati e con circa 6000 reader. I sistemi AVI di Transcore permettono di gestire la totalità del sistema ferroviario USA e hanno significative implementazioni in decine di Paesi del mondo tra cui spicca il TGV in Francia. Obiettivi principali del progetto delle SBB sono stati l'inventario dei veicoli, la programmazione dei mezzi e la gestione integrata della manutenzione. Il primo passo è consistito nell'etichettatura di 1200 locomotive con tag di lettura/scrittura e l'installazione di 50 lettori lungo i binari, così da consentire la localizzazione automatica e in tempo reale dei treni. In un secondo momento sono stati installati lungo tutta la rete ferroviaria altri 230 lettori, e 4800 carrozze passeggeri sono state dotate di tag RFID, operazione eseguita durante i processi di manutenzione ordinaria delle carrozze per abbattere i costi dell'operazione stessa. I lettori sono stati installati in corrispondenza delle stazioni e degli incroci internazionali; il collegamento via modem con il sistema centrale SBB di Berna consente il caricamento dei dati in automatico presso la sede. Il sistema AVI è stato interfacciato con gli strumenti di manutenzione permettendo quindi il reporting in tempo reale di anomalie al sistema di gestione centrale. I vantaggi che sono stati realizzati vanno dall'ottimizzazione dell'utilizzo e distribuzione dei veicoli, alla riduzione dei costi e dei tempi di stazionamento dei treni per effettuare gli inventari; ma SBB punta soprattutto all'offerta di un miglior sevizio ai propri clienti. Per questa ragione SBB ha sviluppato un sito Internet (CIS, *Cargo Information System*) che permette ai propri clienti di trasporto merci di avere informazioni dettagliate e aggiornate on line su tutto il processo di spedizione, dalla prenotazione alla consegna, utilizzando le informazioni provenienti dal sistema AVI.

2.9.4 Benetton: la supply chain intelligente

La visione dell'RFID di Benetton è olistica di tutta la supply chain. Questo approccio nasce anche dal potere di indirizzo che la società esercita sugli attori della supply chain stessa, che va al di là della proprietà effettiva. Il potere nei confronti dei fornitori e dei propri franchiser (circa 5000 negozi nel mondo) permette a Benetton di

concepire, sperimentare e promuovere un sistema RFID che possa dare benefici a tutti gli attori.

La supply chain intelligente: dalla produzione, al negozio sino al consumatore

Sono stati sviluppati quindi alcuni moduli dell'architettura complessiva RFID per Benetton:

- sistema di gestione dei terzisti per rilevare automaticamente lo stato avanzamento dei programmi di produzione

- sistema di gestione della movimentazione e distribuzione, che prevede contenitori riutilizzabili intelligenti con reader a bordo, e gestione della movimentazione interna di Benetton Logistics, che utilizza il tag in tutte le sue parti

- negozio intelligente. Si basa su una struttura demotica che integra tutte le funzioni di gestione del negozio, sicurezza e monitoraggio da remoto. Il sistema è gestibile via Internet remotamente ed è scalabile per tutti i negozi

- integrazione dei sistemi con RFID garantita da SAP

Attualmente il progetto è in fase di test operativo che prevede lo sviluppo completo di un gruppo limitato di negozi pilota in Italia e all'estero e la sua alimentazione con prodotti Benetton con transponder RFID.

2.9.5 Maratona di Chicago

La maratona di Chicago è un evento sportivo che ha raccolto nel 2002 più di 37000 corridori. Data l'alta affluenza di atleti, molte attività risultano ovviamente complicate e imprecise, prima tra tutte, la rilevazione dei tempi di percorrenza. Questo perché, quando il numero di corridori è così elevato, alla partenza solo le prime file di sportivi iniziano la corsa in concomitanza con il segnale dello starter, e allo stesso modo sono solo coloro che arrivano per primi a ricevere intertempi e tempi finali accurati. In passato, all'arrivo i corridori, una volta passato il traguardo, venivano guidati dallo staff lungo corsie in pendenza, dove venivano registrati i numeri di pettorale e i tempi. Nel caso di arrivi di gruppi numerosi, risultava spesso difficile assegnare correttamente i tempi. La situazione è stata risolta grazie all'applicazione di un tag RFID, il *ChampionChip*, sulle scarpe di tutti i partecipanti alla gara. Posizionando in maniera oculata alcuni lettori lungo il percorso, non solo si è in grado di rilevare i tempi corretti, ma anche di dissuadere eventuali corridori dal prendere "scorciatoie". La possibilità di registrare i tempi durante la corsa in maniera automatica e in tempo reale, ha permesso anche di offrire servizi a corredo della maratona prima impossibili: il *Marathon Messenger*, ovvero un

servizio e-mail che offre ai propri iscritti informazioni su tempi intermedi e di fine gara degli atleti a parenti, amici e fan della gara. Questo servizio ha riscosso un certo successo; nel primo anno Marathon Messenger ha inviato ai propri iscritti, circa 15 mila, 60 mila messaggi di aggiornamento sulla gara.

2.10 Normativa

SC 17: Integrated Circuit(s) Cards		
ISO 7816	IC Cards with contact	
ISO 10536	Close coupling cards	WG 4
ISO 14443	Proximity Cards	WG 8/TF 1
ISO 15693	Vicinity Cards	WG 8/TF 2
ISO 10373	Test methods	WG 1/8
SC 17: Animal Identification		
ISO 117984/5	Code Structure and Technical Concept	TC 23/WG 19
ISO 14223	Advanced Transponders	TC 23/WG 19
SC 31: Item management		
ISO 10374	Freight containers	TC 104
ISO 15960	Application requirements transaction message profiles	WG 2/4
ISO 15961	For item management-data objects	WG 2/4
ISO 15962	RFID for item management, data notation	WG 2/4
ISO 15963	Unique Identification of RF Tag Registration Authority toManage the Uniqueness	WG 2/4
ISO 18000	Air interface standards	WG 4/SG 3
Legenda: SC = Subcommittees (sottocomitato) WG = Workgroups (gruppo di lavoro) TF = Taskforces (sottogruppo di lavoro)		
Tabella 2.2 – normativa ISO per rfid		

L'International Organization for Standardization (ISO), organo ufficiale mondiale di standardizzazione, ha un sottocomitato responsabile per l'identificazione automatica e la raccolta dati denominato **SC31**. Questo sottocomitato appartiene al compiato più ampio dell'area Information Technology, lo **JTC1** o *Joint Technical Committee.* La segreteria dello SC31 è formata da un gruppo di esperti che hanno il compito di gestire il processo di definizione dello standard. I principali standard attualmente in vigore o in fase di realizzazione sull'RFID sono riportati nella tabella 2.2.

2.10.1 Standard

Per l'Rfid esistono standard di architettura e protocollo di scambio dati e standard di conformità per le emissioni in radiofrequenza, che non devono sovrapporsi a bande di frequenza già allocate per altri impieghi (radiotelevisione analogica e digitale, telefonia cellulare, Wi-Fi...). Il problema degli standard è che non è stata ancora raggiunta la convergenza e l'unificazione a livello internazionale fra le due principali istituzioni che promuovono queste direttive: ISO ed EPCglobal, l'organismo formato per regolamentare l'Electronic Product Code nella produzione e distribuzione di beni di consumo e diventato uno standard industriale riconosciuto e adottato su scala mondiale. Gli standard Iso per i tag ad alta frequenza (ISO 18000-3) e ad altissima frequenza (ISO 18000-6) sono largamente utilizzati nelle applicazioni “tradizionali” del transponder per l’identificazione di persone, veicoli, il controllo fasi di lavorazione nei processi industriali. Ma esistono numerosi altri standard ISO per i tag RFID. Per quanto riguarda EPC sono stati sviluppati tag conformi allo standard di Classe 0 (UHF), Classe 1 (13,56 MHz e UHF) e Classe 2 (UHF). Nel dicembre del 2004 è stata rilasciato un nuovo standard chiamato Gen 2, che si pensa finirà per sostituire gli standard di Classe 1 e Classe 2.

Che cos'è l'EPC Gen 2?

Epc Gen2 significa EPC Generation 2. È il protocollo EPC di seconda generazione , progettato per operare a livello internazionale. L'EPC Gen è al centro dell'attenzione perché sembra probabile una convergenza fra gli standard UHF Gen 2 e una revisione dell'ISO 18000-6. Tutte le parti in causa (industria, ISO, EPC Global) hanno interesse a che ciò avvenga. Il processo di unificazione potrebbe contribuire a un’ulteriore accelerazione nell’adozione su scala globale dell'Rfid.

Le nazioni usano gli stessi standard per le frequenza Rfid?

La situazione per gli standard di frequenza Rfid è più complessa di quella degli standard di architettura e di protocollo in quanto le norme per le concessioni delle frequenze radio, in generale, variano nei vari paesi (Europa, Usa e Giappone). Risulta pertanto complicato riuscire stabilire una frequenza o una banda di frequenza da riservare all'Rfid su scala globale. L'unica che al momento si può considerare unificata sul pianeta è quella HF, fissata ovunque a 13,56 MHz. Alle basse frequenze la maggior parte delle nazioni ha assegnato la fascia 125 kHz o 134 kHz, ma la normativa non è unica, in quanto in Giappone e in Europa i livelli di potenza sono molto inferiori rispetto a quelli ammessi negli Usa. Il problema però in questo caso è solo apparente in quanto basse frequenze sono spesso utilizzate per applicazioni a livello locale. Più penalizzante la situazione per le UHF,

uno spettro di frequenza utilizzato da molti altri dispositivi elettronici. L'Europa utilizza la banda tra 869,40 e 869,65 MHz, mentre gli Stati Uniti utilizzano come frequenza 915 MHz. Il Giappone è orientato verso la banda dei 960 MHz. Alcune organizzazioni, come la Global Commerce Initiative, stanno facendo pressioni sui governi per incoraggiare l'adozione di bande condivise.

3. Replicazione

3.1 Modelli per sistemi distribuiti

3.1.1 Guasti dei processi

A meno che non fallisca, ovvero si guasti, si suppone che un processo esegua l'algoritmo assegnatogli, attraverso l'insieme dei componenti che, al suo interno, lo implementano. L'unità atomica di guasto è il processo. Quando questo si guasta, si suppone che tutti i suoi componenti falliscano allo stesso modo e nello stesso istante. Le astrazioni di processo differiscono in accordo con la natura dei guasti che vengono considerati. Di seguito ne discuteremo varie tipologie.

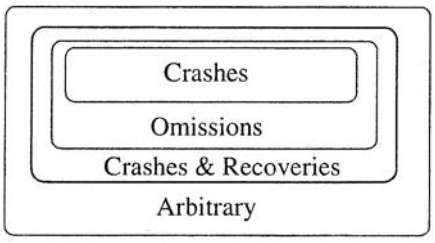


Figura 3.1 – Classificazione delle tipologie di guasto

Bugie ed Omissioni

Se un processo devia arbitrariamente dell'algoritmo che gli è stato assegnato allora esso è destinato a fallire in maniera arbitraria. Il guasto di tipo arbitrario è il più generale, non vengono fatte assunzioni circa il comportamento dei processi guasti, i quali sono liberi di produrre qualsiasi tipo di output e in particolare possono inviare qualsiasi tipo di messaggio. Questi tipi di guasto sono chiamati anche Bizantini o Maliziosi. Sono i più costosi da tollerare, ma questa risulta essere l'unica scelta plausibile quando è richiesto un grado di copertura estremante elevato o quando c'è il rischio che alcuni processi possano essere controllati da utenti maliziosi che deliberatamente provano a manomettere il corretto funzionamento del sistema. Un guasto arbitrario non necessariamente deve essere intenzionale e malizioso: esso può essere semplicemente causato da un bug nell'implementazione, nel linguaggio di programmazione o nel compilatore, che fa deviare il processo dall'algoritmo ad esso

assegnato. Un insieme più ristretto di tipi di guasto sono le Omissioni. Queste occorrono quando un processo non invia/riceve un messaggio che, in accordo con l'algoritmo, avrebbe dovuto inviare/ricevere. In generale, sono dovute a overflows dei buffer o a congestione della rete. Esse risultano in perdite di messaggi. Il verificarsi di un'omissione porta il processo a deviare dal suo algoritmo, omettendo alcuni messaggi che invece avrebbe dovuto scambiare con altri processi.

Crashes

Un caso particolare di omissione si ha quando un processo esegue il suo algoritmo correttamente, incluso lo scambio di messaggi con gli altri processi, fino ad un qualche istante t, dopo il quale non invia più nessun messaggio. Questo è ciò che accade se ad esempio esso si guasta all'istante t, dopo di che non farà più recovery. Se oltre a non spedire alcun messaggio, smette anche di eseguire qualsiasi computazione locale, allora possiamo parlare di guasto di tipo *Crash* e di astrazione di processo di tipo *Crash-Stop*. Con questa astrazione, un processo fallisce se esso si guasta. In tal caso viene definito *guasto all'istante t*. E' invece detto *corretto* se non si guasta mai ed esegue un numero infinito di passi. Di seguito discutiamo alcuni aspetti dell'astrazione appena presentata.

- E' abbastanza comune concepire algoritmi che implementano una data astrazione di programmazione supponendo che un numero minimo F di processi sia corretto, ad esempio almeno uno oppure una maggioranza. E' importante notare che tali assunzioni non significano che l'hardware sottostante deve essere in grado di operare correttamente per sempre. Piuttosto significano che per ogni esecuzione dell'algoritmo è molto improbabile che più di un certo numero N di processi si guasti, durante tutto il tempo di vita dell'esecuzione stessa. Un ingegnere che stia progettando un algoritmo per una data applicazione dovrebbe assicurarsi che gli elementi scelti per la realizzazione di ciò che è al di sotto dell'architettura software ed hardware rendano tale assunzione plausibile. In generale, è anche buona pratica, quando si concepiscono algoritmi per l'implementazione di un'astrazione distribuita, per la quale sono state fatte determinate assunzioni, verificare precisamente quali proprietà dell'astrazione sono preservate e quali possono essere violate quando uno specifico sottoinsieme delle assunzioni si trova a non essere più soddisfatto, ad esempio quando più di N processi si guastano.

- Considerare un'astrazione di tipo crash-stop significa assumere che un processo esegua il suo algoritmo correttamente, a meno che non si guasti, nel qual caso esso non fa recovery. Ovvero, una volta che si è guastato, il processo non esegue più alcuna computazione. Ovviamente, in pratica, processi che si guastano

possono essere fatti ripartire. E' importante notare che l'astrazione di processo crash-stop non preclude la possibilità di recovery, né implica che, per avere un comportamento corretto da parte di un determinato algoritmo, il recovery dovrebbe essere impedito. Significa semplicemente che, per portare avanti la propria esecuzione, gli algoritmi non possono fare affidamento sul fatto che alcuni dei processi possano o meno fare recovery. Questo potrebbe non verificarsi mai, oppure potrebbe verificarsi solo dopo un lungo periodo di tempo comprendente le rilevazione del guasto e il ritardo di rebooting. Intuitivamente, un algoritmo che non fa affidamento sul recupero di processi guasti dovrebbe tipicamente essere più veloce rispetto a uno che invece lo fa. Nulla impedisce ai processi che hanno fatto recovery di recuperare le informazioni circa lo stato della computazione e partecipare alle successive sottoistanze dell'algoritmo distribuito.

Recoveries

A volte, l'assunzione che alcuni processi non si guastino mai può essere una condizione troppo forte per alcuni ambienti distribuiti. Ad esempio, richiedere che una maggioranza di processi non si guasti, anche se solo per un periodo abbastanza lungo da permettere all'esecuzione di terminare, potrebbe essere una condizione non realistica o realizzabile. Se si presentano questi casi, un'interessante astrazione di processo alternativa da considerare è quella *crash-recovery,* alla quale è legato anche il concetto di *tipo di guasto crash-recovery*. In relazione a questa tipologia di astrazione un processo è definito guasto solo se fallisce e non recupera mai, o se eventi di crash e recovery si alternano infinitamente, altrimenti è detto corretto. In quest'ultimo caso, di base, alla fine il processo è sempre (durante il tempo di vita dell'esecuzione dell'algoritmo di interesse) attivo ed operativo. C'è da precisare che un processo che si guasta e fa recovery un numero finito di volte, in questo modello, è comunque considerato corretto. In accordo con l'astrazione crash-recovery, un processo può guastarsi e in questo caso smette di inviare messaggi, ma in seguito potrebbe tornare attivo. Tale scenario può essere visto come un guasto di omissione, però con un'eccezione: il processo potrebbe soffrire di amnesia, ovvero a seguito del guasto esso perde il proprio stato interno. Questa situazione complica significativamente il progetto dell'algoritmo perché, nel momento del recupero, il processo potrebbe inviare nuovi messaggi che contraddicono quelli che esso stesso potrebbe aver inviato prima di guastarsi. E' quindi necessario assumere che ogni processo abbia una memoria stabile (chiamata log), differente rispetto alla classica memoria volatile, che possa essere acceduta attraverso primitive di *store* e *retrieve*. Assumiamo inoltre che un processo è conscio del fatto che si è guastato e che ha fatto recovery. In particolare, si suppone che venga generato dall'ambiente run-time uno specifico evento <Recovery>, in maniera molto simile alla generazione dell'evento <init>, eseguito ogni volta

che un processo comincia ad eseguire un qualche algoritmo. La routine di processamento dell'evento <Recovery> potrebbe per esempio recuperare lo stato del processo dalla memoria stabile prima che la normale esecuzione venga ripristinata. I processi inoltre potrebbero aver perso tutti gli eventuali altri dati che erano in memoria volatile, i quali dovrebbero essere opportunamente reinizializzati. L'evento <Init> è considerato atomico rispetto al recovery. Più precisamente, se un processo si guasta nel mezzo della sua procedura di inizializzazione e fa recovery, senza aver terminato il processamento dell'evento <Init>, allora dovrebbe rifare nuovamente l'intera procedura di gestione di <Init> prima di procedere con l'esecuzione di quella che gestisce l'evento <Recovery>. Il tipo di guasto crash-recovery può essere ridotto ad un guasto di omissione se supponiamo che ogni processo memorizza tutti gli aggiornamenti e ognuna delle sue variabili in memoria stabile. Ciò non è molto pratico perché l'accesso al log è di solito costoso in termini di tempo. Non a caso una questione cruciale nella realizzazione di algoritmi per l'astrazione di crash-recovery è minimizzare gli accessi alla memoria stabile. Discutiamo di seguito alcuni aspetti strettamente legati all'astrazione crash-recovery.

- Un modo per alleviare il bisogno di accedere al log è assumere che alcuni dei processi non si guastino mai (durante il tempo di vita dell'esecuzione dell'algoritmo). A prima vista ciò potrebbe sembrare contraddittorio con la reale motivazione dell'introduzione dell'astrazione crash-recovery. In realtà non ci sono contraddizioni, vediamo perché: con i guasti di tipo crash-stop, alcune astrazioni di programmazione distribuite possono essere implementate solo a condizione che un certo numero di processi non si guasti mai, diciamo una maggioranza di processi che partecipano alla computazione, ad esempio 4 su 7. Questa assunzione potrebbe essere considerata poco realistica in alcuni ambienti. Potrebbe essere più ragionevole assumere che almeno due processi non si guastino durante l'esecuzione dell'algoritmo (Il resto dei processi possono invece guastarsi e recuperare). Tali condizioni rendono qualche volta possibile realizzare algoritmi assumendo l'astrazione di processo crash-recovery senza alcuna necessità di accedere alla memoria stabile. Infatti, i processi che non si gustano possono essere utilizzati per implementare un'astrazione di memoria stabile virtuale, e ciò è possibile senza conoscere in anticipo quali dei processi non si guasterà durante una data esecuzione dell'algoritmo.

- A prima vista, si potrebbe pensare che l'astrazione di tipo crash-stop possa anche catturare situazioni dove i processi si guastano e recuperano, semplicemente permettendo loro di modificare il proprio identificativo in seguito ad un recovery. Ovvero un processo che recupera dopo un guasto potrebbe comportarsi, rispetto agli altri, come se esso fosse un'entità

differente che fino a quel momento semplicemente non aveva fatto ancora alcuna operazione. Ciò potrebbe essere facilmente fatto attraverso una procedura di reinizializzazione che, oltre ad inizializzare lo stato del processo come se avesse appena iniziato la propria esecuzione, gli permetta anche di cambiare la propria identità. Inoltre il processo che ha fatto recovery dovrebbe essere aggiornato dagli altri con ogni informazione persa, come se non le avesse mai ricevute. Sfortunatamente tale visione è fuorviante, come spieghiamo di seguito. Consideriamo ancora un algoritmo realizzato utilizzando l'astrazione crash-stop, e assumendo che una maggioranza di processi non si guastino mai, diciamo 4 al fronte di un totale di 7. Consideriamo ulteriormente uno scenario in cui 4 di essi invece si guastano ed uno solo fa recovery. Pretendere che quest'ultimo sia un processo differente (un nuovo processo) porterebbe ad avere un sistema composto in realtà da 8 processi, 5 dei quali non dovrebbero guastarsi, e lo stesso ragionamento potrebbe essere fatto per un qualsiasi numero più elevato. Questo perché una assunzione fondamentale che abbiamo costruito in precedenza è che l'insieme dei processi coinvolti in una data computazione è statico ed essi si conoscono tutti a vicenda.

- Un aspetto delicato dell'astrazione di processo di tipo crash-recovery è l'interfaccia tra i moduli software. Assumiamo che un modulo, chiamato per comodità A, coinvolto nell'implementazione di una qualche specifica astrazione distribuita, consegni messaggi o decisioni allo strato superiore (diciamo l'applicazione) e successivamente il processo che li ospita si guasti. Al recovery A non può determinare se lo strato sovrastante ha processato o meno il messaggio/decisione prima del crash. Ci sono almeno due modalità per affrontare questo problema:

  1. Un primo modo è cambiare l'interfaccia tra i moduli. Invece di consegnare un messaggio (o una decisione) allo strato superiore, A può memorizzare il messaggio (decisione) in memoria stabile la quale è accessibile anche allo strato superiore. E' allora compito di quest'ultimo accedere alla memoria stabile e reperire le informazioni consegnate.

  2. Il secondo approccio consiste nel realizzare A in modo che questo consegni periodicamente il messaggio/decisione allo strato applicativo, fino a che quest'ultimo esplicitamente chiede di interrompere la consegna. In questo caso la responsabilità dell'accertamento dell'utilizzo dell'informazione consegnata da parte dell'applicazione ricade sull'astrazione di programmazione distribuita implementata da A.

3.1.2 Astrazioni delle comunicazioni

L'astrazione di link è utilizzata per rappresentare i componenti di rete di un sistema distribuito. Assumiamo che ogni coppia di processi sia connessa da un link bidirezionale, una topologia questa che fornisce connettività totale tra i processi. Nella realtà, per implementare questa astrazione, possono essere utilizzate differenti topologie, basate ad esempio su componenti che utilizzano algoritmi di routing per l'instradamento dei messaggi. Esempi concreti, come quelli illustrati nella figura 3.2, includono l'utilizzo di broadcast medium (come ad esempio Ethernet), di un anello, o di una maglia di links interconnessi da bridges e routers (come Internet). Molte implementazioni raffinano e specializzano la visione astratta della rete per fare uso di proprietà messe a disposizione della topologia sottostante.

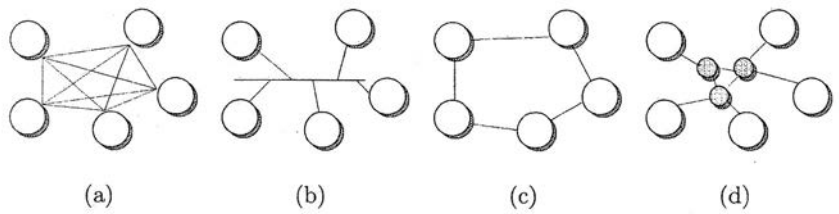


Figura 3.2 – Esempi di topologie di rete

Assumiamo che i messaggi scambiati tra i processi siano univocamente identificati e ogni messaggio trasporta abbastanza informazione da permettere al ricevente di identificare univocamente il mittente. In più supponiamo che, quando comunicano in modalità request-reply, i processi siano in grado di determinare le corrispondenze richiesta-risposta tra i messaggi ricevuti/inviati. Questo di solito viene realizzato utilizzando timestamps generati dai processi e basati su numeri di sequenza o su clock locali. Questa assunzione ci permette di evitare di introdurre esplicitamente tali timestamps e la relativa gestione all'interno degli algoritmi.

Guasti dei Links

In un sistema distribuito, non è difficile che tra migliaia di messaggi propagati attraverso la rete, alcuni possano andare persi. Inoltre, è ragionevole assumere che la probabilità per un messaggio di raggiungere la sua destinazione non sia zero. Così un modo naturale di superare l'inaffidabilità inerente della rete è utilizzare meccanismi di ritrasmissione dei messaggi fino a che questi non raggiungono le

loro destinazioni. Nel seguito descriveremo differenti tipi di astrazioni di link: alcune sono più forti di altre, rispetto alle garanzie di affidabilità offerte. Tutte e tre sono astrazioni di tipo *punto-punto,* ovvero supportano la comunicazione tra coppie di processi. Descriveremo prima l'astrazione di link *fair-loss*, che cattura l'idea base che i messaggi possono essere persi, ma che la probabilità che ciò non accada non è zero. Poi descriveremo astrazioni di più alto livello che potrebbero essere implementate al di sopra dei links fair-loss utilizzando meccanismi di ritrasmissione per nascondere al programmatore parte dell'inaffidabilità della rete. Più precisamente considereremo le astrazioni di link di tipo *stubborn* e *perfect.* Se non specificato diversamente, assumeremo di operare con astrazione di processo di tipo crash-stop. Definiremo la proprietà di ognuna delle astrazioni di link utilizzando due tipi di primitive: *send* e *deliver.* Il termine deliver è preferito rispetto al più generale termine *receive* per rendere chiaro che stiamo parlando di una specifica astrazione di link che deve essere implementata al di sopra di una rete: un messaggio può tipicamente essere ricevuto (receive) presso un dato porto e memorizzato all'interno di un qualche buffer, poi un qualche algoritmo sarà eseguito per assicurare che le proprietà dell'astrazione di link richiesta siano soddisfatte, prima che il messaggio sia realmente consegnato (deliver). Per richiedere la spedizione di un messaggio utilizzando una determinata astrazione di link un processo deve semplicemente invocare la relativa primitiva di send. Quando quest'ultima viene invocata, diciamo che si sta inviando un messaggio. E' ora compito dell'astrazione di link di eseguire delle operazioni per trasmetterlo al processo destinatario, in accordo con la reale specifica. La primitiva di deliver è invocata dall'algoritmo che implementa l'astrazione presso il destinatario. Quando questa è invocata presso un processo p per un messaggio m, diciamo che p consegna m.

Fair-loss Links

L'interfaccia dell'astrazione di *link fair-loss* è descritta dal modulo 3.1. Questa consiste di due eventi: un evento di richiesta, utilizzato per la spedizione dei messaggi, e un evento di notifica, utilizzato per consegna. I links di tipo fair-loss sono caratterizzati dalle proprietà **FLL1-FLL3**. Di base, la proprietà di *fair-loss* garantisce che un link non perde sistematicamente ogni messaggio. Inoltre, se né il sender né il receiver si guastano, e se un messaggio viene ritrasmesso in continuazione, il messaggio alla fine (prima o poi) verrà consegnato. La proprietà di *finite duplication* in maniera intuitiva assicura che la rete non effettua più ritrasmissioni di quelle realmente effettuate dai processi stessi. Infine la proprietà di *no creation* assicura che nessun messaggio è creato o corrotto dalla rete.

Module:

Name: FairLossPointToPointLinks (flp2p).

Events:

Request: ⟨ *flp2pSend* | dest, m ⟩: Used to request the transmission of message m to process *dest*.

Indication: ⟨ *flp2pDeliver* | src, m ⟩: Used to deliver message m sent by process *src*.

Properties:

FLL1: *Fair loss:* If a message m is sent infinitely often by process p_i to process p_j, and neither p_i nor p_j crash, then m is delivered an infinite number of times by p_j.

FLL2: *Finite duplication:* If a message m is sent a finite number of times by process p_i to process p_j, then m cannot be delivered an infinite number of times by p_j.

FLL3: *No creation:* If a message m is delivered by some process p_j, then m has been previously sent to p_j by some process p_i.

Modulo 3.1 – Specifica di Fair-Loss Link

Stubborn Links

L'astrazione di canale *stubborn* è definita nel modulo 3.2. Questa astrazione nasconde il meccanismo di ritrasmissione degli strati sottostanti utilizzato dal processo sender, quando questo effettivamente utilizza dei canali di tipo fair-loss, per essere sicuro che i messaggi alla fine saranno consegnati dal processo destinatario. L'algoritmo 3.1 realizza un'implementazione molto semplice di link stubborn al di sopra di uno di tipo fair-loss. Nel seguito discutiamo la correttezza dell'algoritmo e faremo anche alcune considerazioni sulle performance.

Correttezza: La proprietà di fair-loss del link sottostante garantisce che, se il processo destinatario è corretto, esso consegnerà, infinite volte, ogni messaggio inviato da ogni altro processo che non si guasta dopo l'invio. Questo perché l'algoritmo assicura che il mittente chiamerà infinite volte la primitiva sp2pSending per ogni messaggio da spedire, a meno che il mittente stesso non si guasti. La proprietà di no creation è semplicemente garantita dal link sottostante.

Performance: L'algoritmo è chiaramente non performante, ma è utile per capire il funzionamento dei link stubborn. E' chiaro che, all'interno di applicazioni pratiche, non ha molto senso per un processo, ad ogni passo, inviare infinite volte un messaggio. Ci sono almeno tre modi complementari per evitare ciò e rendere così l'algoritmo più pratico. Primo, il processo mittente potrebbe introdurre un ritardo tra due invii successivi. Secondo, è molto importante ricordare che le nozioni di infinito e infinitamente sono sempre dipendenti dal contesto: esse di base dipendono dall'algoritmo che sta utilizzando i links stubborn. Dopo che l'algoritmo ha terminato la propria esecuzione non c'è bisogno di perseverare con l'invio dei messaggi. Terzo, Può essere

aggiunto un meccanismo di acknowledgement, possibilmente utilizzato per gruppi di messaggi, per indicare ad un sender che non c'è bisogno di continuare a inviare un determinato insieme di messaggi.

Module:

Name: StubbornPointToPointLink (sp2p).

Events:

Request: ⟨ *sp2pSend* | *dest*, *m* ⟩: Used to request the transmission of message *m* to process *dest*.

Indication: ⟨ *sp2pDeliver* | *src*, *m* ⟩: Used to deliver message *m* sent by process *src*.

Properties:

SL1: *Stubborn delivery:* Let p_i be any process that sends a message m to a correct process p_j. If p_i does not crash, then p_j delivers m an infinite number of times.

SL2: *No creation:* If a message m is delivered by some process p_j, then m was previously sent to p_j by some process p_i.

Modulo 3.2 – Specifica di Stubborn Link

L'ack può essere inviato ogniqualvolta un processo destinatario ha propriamente consumato dei messaggi, o ne ha consegnati alcuni la cui consegna include semanticamente quella di altri spediti in precedenza. Un tale meccanismo dovrebbe però essere visto come un algoritmo esterno, e non può essere integrato in quello implementante lo stubborn link, altrimenti l'algoritmo potrebbe non realizzare più l'astrazione originaria.

Implements:
StubbornPointToPointLink (sp2p).

Uses:
FairLossPointToPointLinks (flp2p).

upon event ⟨ *Init* ⟩ **do**
sent := ∅;
startTimer (TimeDelay);

upon event ⟨ *Timeout* ⟩ **do**
forall (*dest*, *m*) ∈ sent **do**
trigger ⟨ *flp2pSend* | *dest*, *m* ⟩;
startTimer (TimeDelay);

upon event ⟨ *sp2pSend* | dest, m ⟩ **do**
trigger ⟨ *flp2pSend* | dest, m ⟩;
sent := sent ∪ {(dest,m)};

upon event ⟨ *flp2pDeliver* | src, m ⟩ **do**
trigger ⟨ *sp2pDeliver* | src, m ⟩;

Algoritmo 3.1 – Stubborn Link realizzato su Fair-Loss link

Perfect Links

Con la precedente astrazione, il compito di verifica dell'effettiva consegna di un messaggio ricade sul processo destinatario. Meccanismi per l'individuazione dei messaggi duplicati, insieme a quelli di ritrasmissione, aiutano a costruire un'astrazione ancora più alta: quella *perfect-link*, qualche volta chiamata anche *reliable-channel*. Questa specifica è catturata dal modulo *Perfect Point to Point Link*, ovvero il modulo 3.3. La sua interfaccia consta di due eventi: uno di richiesta (per spedire un messaggio) e uno di notifica (usato per effettuare una consegna). I perfect links sono caratterizzati dalle proprietà **PL1-PL3**. L'algoritmo 2.2 descrive un'implementazione molto semplice di perfect-links al di sopra degli stubborn links.

Correttezza: Consideriamo la proprietà di *reliable delivery* dei perfect links. Sia m un qualsiasi messaggio *pp2pSent* da un qualche processo p ad un qualche altro processo q e assumiamo che nessuno di questi due si guasti. Per l'algoritmo, p sp2pSends m a q utilizzando il sottostante link stubborn. Per la proprietà stubborn delivery dei links sottostanti, q alla fine sp2pDelivers m almeno una volta, e da qui q pp2pDelivers m. La proprietà di *no duplication* segue dal test effettuato dall'algoritmo prima di effettuare la consegna allo strato applicativo, ovvero ogniqualvolta un messaggio è sp2pDelivered e prima che esso sia pp2pDelivered. La proprietà di *no creation* semplicemente segue dall'omonima proprietà dei sottostanti stubborn links.

Module:

Name: PerfectPointToPointLink (pp2p).

Events:

Request: ⟨ *pp2pSend* | *dest*, *m* ⟩: Used to request the transmission of message *m* to process *dest*.

Indication: ⟨ *pp2pDeliver* | *src*, *m* ⟩: Used to deliver message *m* sent by process *src*.

Properties:

PL1: *Reliable delivery:* Let p_i be any process that sends a message m to a process p_j. If neither p_i nor p_j crashes, then p_j eventually delivers m.

PL2: *No duplication:* No message is delivered by a process more than once.

PL3: *No creation:* If a message m is delivered by some process p_j, then m was previously sent to p_j by some process p_i.

Modulo 3.3 – Specifica di Perfect Link

Performance: Insieme alle considerazioni di performance discusse per l'implementazione di stubborn links e che chiaramente si applicano anche all'implementazione del perfect link attraverso l'algoritmo 3.2, c'è un aspetto aggiuntivo relativo al mantenimento, presso ogni processo, dell'insieme delivered utilizzato per memorizzare i

messaggi ricevuti, il quale cresce in maniera illimitata con il passare del tempo scontrandosi con la realtà di una memoria fisica di dimensioni finite.

```
Implements:
    PerfectPointToPointLinks (pp2p).

Uses:
    StubbornPointToPointLinks (sp2p).

upon event ⟨ Init ⟩ do
    delivered := ∅;

upon event ⟨ pp2pSend | dest, m ⟩ do
    trigger ⟨ sp2pSend | dest, m ⟩;

upon event ⟨ sp2pDeliver | src, m ⟩ do
    if (m ∉ delivered) then
        delivered := delivered ∪ { m };
        trigger ⟨ pp2pDeliver | src, m ⟩;
```

Algoritmo 3.2 – Perfect Link realizzato al di sopra di uno Stubborn Link

A prima vista, potrebbe sembrare semplice realizzare uno schema per la risoluzione di tale problema basato sull'invio periodico di ack da parte dei destinatari per i messaggi ricevuti e consegnati. Con tali ack i mittenti verrebbero a conoscenza dei messaggi sicuramente ricevuti, sospendendo quindi il loro invio continuo. Non ci sono però garanzie che tali messaggi non siano ancora in transito nella rete e che in un qualche futuro, più o meno remoto, essi raggiungano il processo destinatario, che di conseguenza non può rimuovere da delivered i messaggi per i quali ha già inviato un ack. Potrebbero essere utilizzati allora dei meccanismi addizionali, basati ad esempio su timestaps, utili per riconoscere e scartare i messaggi obsoleti.

FIFO Perfect Links

Quest'astrazione deriva direttamente dalla precedente, dalla quale eredita direttamente le tre proprietà: *reliable delivery*, *no duplication*, *no creation*. In più la estende aggiungendo un'ulteriore proprietà: *FIFO delivery*, la quale impone dei vincoli sull'ordine di consegna di messaggi da parte dei destinatari. Più precisamente:

- *FIFO delivery*: data una qualsiasi coppia di messaggi m_1 e m_2 , se il processo mittente p_i spedisce prima m_1 e poi m_2 , allora il processo destinatario p_j , se li consegna entrambi, consegnerà prima m_1 e poi m_2 .

Una possibile implementazione di un algoritmo che realizza un'astrazione di FIFO perfect links può essere facilmente realizzata utilizzando dei timestamps sequenziali, utilizzati dai processi mittenti per etichettare i messaggi inviati nel seguente modo: indichiamo con

$t(m_i)$ il timestamp utilizzato per etichettare il messaggio m_i , se p_i invia m_1 e successivamente m_2 allora $t(m_1)<t(m_2)$ dove la relazione < è definita in funzione della relazione d'ordine totale valida per i timestamps sequenziali. Dal lato del destinatario poi le consegne verranno effettuate rispettando l'ordine stabilito dai timestamps dei messaggi.

3.1.3 Assunzioni temporali

Un importante aspetto nella caratterizzazione di un sistema distribuito è quello relativo al comportamento dei suoi processi e links in funzione del trascorrere del tempo. In breve, quando si definisce un sistema distribuito è di primaria importanza la possibilità di determinare se è possibile fare delle assunzioni sull'esistenza di limiti di tempo nei ritardi di comunicazione e velocità (relative) dei processi. Di seguito arriveremo ad alcune conclusioni relative al tempo e poi illustreremo l'astrazione di *failure detector* come metodo principale per astrarre alcune utili assunzioni temporali.

Sistemi asincroni

Assumere che un sistema distribuito sia asincrono significa non fare alcuna ipotesi temporale riguardo processi e canali. Questo è ciò che è stato fatto in precedenza quando ad esempio abbiamo definito le astrazioni di processo e link. Infatti non abbiamo fatto assunzioni circa la possibilità da parte dei processi di accedere ad una qualche sorta di clock fisico, né abbiamo assunto alcun limite sui ritardi di processamento e di comunicazione. Anche senza avere accesso a clocks fisici, è possibile misurare il passaggio del tempo in funzione degli eventi di trasmissione e consegna dei messaggi, ovvero esso può essere definito rispetto alla comunicazione. Il tempo misurato in questo modo è chiamato tempo logico. Di seguito riportiamo delle regole che possono essere utilizzate per misurare il passaggio del tempo in un sistema distribuito asincrono:

- ogni processo p mantiene un intero chiamato clock logico l_p , inizializzato a 0
- ogni volta che si verifica un evento presso il processo p, il clock logico l_p è incrementato di una unità e tale valore aggiornato viene associato all'evento
- quando un processo spedisce un messaggio, associa a quest'ultimo il valore del clock logico calcolato al momento della spedizione ed etichetta il messaggio stesso con tale valore (la generazione di un messaggio naturalmente è un evento e quindi porta all'incremento del clock logico). Denotiamo il timestamp dell'evento *e* con $t(e)$.

- quando un processo p riceve un messaggio m con timestamp l_m , p incrementa il suo timestamp in modo seguente: $l_p = max(l_p, l_m) + 1$

Un aspetto interessante dei clock logici è il fatto che essi catturano la relazione *causa-effetto* in sistemi in cui i processi possono interagire soltanto attraverso scambio di messaggi. Diciamo che un evento e_1 potrebbe aver causato un altro evento e_2 , denotato con $e_1 \rightarrow e_2$ se è possibile applicare la seguente relazione, chiamata ***relazione happened-before***:

- e_1 e e_2 hanno luogo nello stesso processo p ed e_1 si verifica prima di e_2 (figura 3-4 (a))
- e_1 corrisponde alla trasmissione di un messaggio m presso un processo p ed e_2 corrisponde alla ricezione dello stesso messaggio presso un qualche altro processo q (figura 3.4 (b))
- $\exists e_a$ t.c. $e_1 \rightarrow e_a \wedge e_a \rightarrow e_2$ (figura 3.4 (c))

Può essere dimostrato che se gli eventi sono etichettati con i clock logici, allora $e_1 \rightarrow e_2 \Rightarrow t(e_1) < t(e_2)$. Notare che *l'implicazione opposta non è valida*. Anche in assenza di una qualsiasi assunzione sul tempo fisico, ed utilizzando solo la nozione di tempo logico, possiamo implementare alcune utili astrazioni di programmazione distribuita. Molte astrazioni però, per essere realizzate, necessitano di assunzioni temporali. Ad esempio anche una forma di accordo veramente semplice, chiamata *consenso*, è impossibile da risolvere in un sistema asincrono anche se soltanto uno dei processi si guasta. In questo problema i processi partono, ognuno con un valore individuale, e devono accordarsi su un unico valore comune finale, scelto tra tutti quelli iniziali. La conseguenza di questo risultato è immediata: senza le opportune assunzioni è impossibile derivare algoritmi per molte astrazioni di accordo, inclusa la group membership o le comunicazioni di gruppo totalmente ordinate.

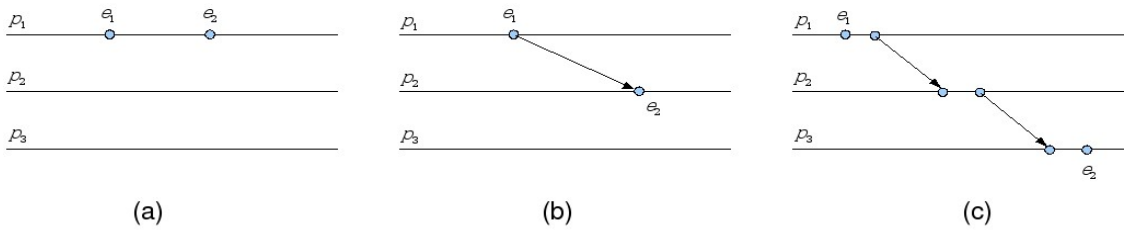


Figura 3.3 – Regole della relazione Happened-Before

Sistemi sincroni

Mentre assumere che un sistema sia asincrono consiste nel non fare alcuna ipotesi temporale sui processi e sui links, assumere che un sistema sia sincrono comporta ipotizzare la validità delle tre seguenti proprietà:

1. *computazione sincrona*: esiste un upper bound sui ritardi di processamento. Ovvero, il tempo speso da ogni processo per eseguire un passo è sempre minore di una determinata soglia di tempo. Ricordiamo che un passo comprende la consegna di un messaggio (anche nullo) spedito da un qualche altro processo, una computazione locale (che possibilmente può coinvolgere interazioni tra numerosi layer dello stesso processo), e la spedizione di un messaggio a qualche altro processo.

2. *comunicazione sincrona*: esiste un upper bound conosciuto per il ritardo di trasmissione dei messaggi. Ovvero, il periodo di tempo tra l'istante in cui il messaggio è spedito e l'istante in cui esso è consegnato dal processo destinatario è inferiore a tale bound.

3. *clock fisici sincroni*: ogni processo è equipaggiato con clock fisico locale. C'è un limite superiore alla frequenza a cui il clock locale devia dal clock reale globale (ricordiamo che facciamo qui l'assunzione che tale clock globale esista nel nostro universo ma non sia accessibile dai processi)

In un sistema distribuito sincrono, possono essere forniti numerosi servizi utili, come ad esempio, tra gli altri:

- rilevazioni dei guasti basata sul tempo: ogni guasto di un processo può essere individuato in tempo finito. Ogniqualvolta un processo p si guasta, tutti quelli corretti, individuano il guasto di p in un determinato intervallo di tempo finito. Tale servizio può essere realizzato utilizzando ad esempio un meccanismo di heartbeat, dove i processi periodicamente si scambiano messaggi e determinano, all'interno di un intervallo finito di tempo, quali sono quelli che si sono guastati.

- misura dei ritardi di transito: è possibile determinare con buona approssimazione i ritardi spesi dai messaggi sui link di comunicazione e, da qui, dedurre quali sono i nodi più distanti o connessi da link più lenti o sovraccarichi.

- coordinazione basata sul tempo: si potrebbe implementare una astrazione di lease che garantisce la possibilità di eseguire alcune azioni per un intervallo fisso di tempo, come ad esempio per la manipolazione di uno specifico file.
- performance nel caso peggiore: assumendo un limite al numero

di guasti e al carico del sistema, è possibile determinare il tempo di risposta nel caso peggiore per un dato algoritmo. Questo permette ad un processo di determinare quando un messaggio che ha spedito sarà ricevuto dal processo destinatario (presupposto che quest'ultimo sia corretto). Se limitiamo anche il numero di fallimenti di omissione che possono interessare i singoli processi allora è possibile determinare l'istante di ricezione di un messaggio anche nel caso in cui un processo sia soggetto ad una o più omissioni senza però guastarsi.

- clocks sincronizzati: è possibile sincronizzare clocks di differenti processi in modo tale che essi non differiscono mai per più di una determinata costante δ, conosciuta anche come *precisione di sincronizzazione* del clock. La sincronizzazione dei clocks permette ai processi di coordinare le loro azioni ed infine di eseguire passi globali sincronizzati. E' inoltre possibile associare dei timestamps agli eventi utilizzando il valore del clock locale all'istante in cui essi occorrono. Questi timestamps possono essere utilizzati ad esempio per ordinare gli eventi. Se ci fosse un sistema in cui tutti i ritardi sono costanti, sarebbe possibile raggiungere clock perfettamente sincronizzati (ovvero con δ pari a 0). Sfortunatamente, un tale sistema non può essere costruito. In pratica δ è sempre più grande di 0 e gli eventi che cadono all'interno di un intervallo pari a δ non possono essere ordinati.

La maggiore limitazione nell'assunzione di sistema sincrono sta nella sua copertura, ovvero nella difficoltà di costruire un sistema reale in cui le assunzioni di tempo valgano con alta probabilità. Ciò richiede tipicamente un'analisi accurata della rete e del carico di processamento e l'uso di appropriati algoritmi di scheduling per il processore e per la rete. Potrebbe essere fattibile per alcune reti in area locale, ma potrebbe non essere possibile, anche se desiderabile, in sistemi su larga scala come Internet. In quest'ultimo caso ci sono periodi in cui i messaggi possono impiegare molto tempo per arrivare alla loro destinazione. Per catturare i bound di processamento e di comunicazione si potrebbero allora considerare valori molto larghi. Questo però significa considerare i valori nel caso peggiore che sono tipicamente molto più alti di quelli medi, sono di solito così elevati che ogni applicazione basata su di essi risulterebbe essere veramente molto inefficiente.

Sincronia Parziale

Generalmente i sistemi distribuiti sono completamente sincroni per la maggior parte del tempo. Più precisamente, per molti sistemi che conosciamo, è relativamente facile definire limiti di tempo fisico che vengono rispettati per la maggior parte del tempo. Ci sono però periodi durante i quali le assunzioni temporali non sono valide, ovvero

periodi in cui il sistema è asincrono. Questi sono intervalli di tempo in cui la rete è ad esempio sovraccarica, o alcuni processi sono soggetti a riduzioni di memoria che li portano a rallentare. Ad esempio il buffer che un processo sta utilizzando per memorizzare i messaggi in ingresso o in uscita potrebbe andare in overflow e alcuni messaggi potrebbero così essere persi, violando i limiti di tempo nella consegna. La ritrasmissione può aiutare ad assicurare l'affidabilità dei canali ma introduce ritardi impredicibili. E' in tal senso che i sistemi reali sono parzialmente sincroni. Un modo per catturare l'osservazione di sincronia parziale è ipotizzare che le assunzioni di tempo valgano solo "*alla fine*" (prima o poi, senza determinare quando esattamente). Questo porta ad assumere che c'è un tempo dopo il quale queste assunzioni valgono per sempre, ma questo non è conosciuto a priori. In questo modo, invece di assumere un sistema sincrono si assume un sistema che è prima o poi sincrono (*eventually synchronous*). E' importante notare che fare tale assunzione non significa in pratica che:

1. c'è un tempo dopo il quale il sistema sottostante (includendo componenti hardware, di rete ed applicativi) è sincrono per sempre

2. il sistema ha bisogno di essere inizialmente asincrono e poi solo dopo qualche periodo (più o meno lungo) diventa sincrono

Essa semplicemente cattura il fatto reale che il sistema potrebbe essere non sempre sincrono, e che a priori non esiste un limite per i periodi di asincronia. Però ci aspettiamo che ci siano periodi durante i quali il sistema è sincrono, e che alcuni di questi intervalli siano abbastanza lunghi da permettere la terminazione dell'esecuzione dell'algoritmo.

3.1.4 Astrazioni di tempo

Failure Detection

Fino ad ora, abbiamo contrapposto la semplicità con le inerenti limitazioni dell'assunzione di sistema asincrono, come anche il potere con la copertura limitata dell'assunzione sincrona, e abbiamo discusso l'assunzione intermedia di sistema parzialmente sincrono. Ognuna di queste acquisisce un significato per ambienti specifici, e vengono considerate assunzioni plausibili quando si ragiona circa implementazioni general purpose di astrazioni di programmazione distribuita di alto livello. Fino a che si considera il modello di sistema asincrono, non ci sono da fare ipotesi sul tempo e le nostre astrazioni di link e di processo catturano direttamente questo caso. Queste non sono tuttavia sufficienti in caso di sistema sincrono o parzialmente sincrono. Però, invece di integrarle con capacità temporali, considereremo un tipo separato di astrazione, chiamata *failure*

detector, che le incapsula. Come discuteremo nella prossima sezione, il failure detector fornisce informazioni (non necessariamente perfettamente accurate) circa quali processi sono guasti. Introdurremo in particolare un failure detector che incapsula assunzioni di tempo per i sistemi sincroni, ed anche uno che incapsula le assunzioni di quelli parzialmente sincroni. Come vedremo le informazioni circa i guasti dei processi fornite dal primo tipo saranno molto più accurate di quelle fornite dal secondo. Più in generale, più forti saranno le assunzioni temporali fatte su un sistema più accurate potranno essere le informazioni. Ci sono almeno due vantaggi nell'utilizzare l'astrazione di failure detector, rispetto ad un approccio dove si fanno direttamente delle assunzioni temporali sui processi e link:

- l'astrazione di failure detector alleggerisce il bisogno di estendere le astrazioni di processo e link introdotte precedentemente con assunzioni temporali: la loro semplicità è preservata.

- come vedremo nel seguito, possiamo ragionare circa le proprietà dei failure detector utilizzando proprietà assiomatiche senza espliciti riferimenti al tempo fisico. Questi ultimi sono di solito molto soggetti ad errori.

In pratica, ad eccezione di specifiche applicazioni come il controllo di processo, le assunzioni temporali sono utilizzate principalmente per individuare guasti dei processi, ovvero per implementare failure detectors: questo è esattamente ciò che noi faremo.

Perfect failure detection

In sistemi sincroni, ed assumendo un'astrazione di processo di tipo crash-stop, i guasti possono essere accuratamente individuati utilizzando dei timeouts. Per esempio, assumiamo che un processo spedisca un messaggio ad un altro processo e poi attenda una risposta. Se il ricevente non si guasta, allora la risposta sicuramente arriverà in un intervallo di tempo minore o al più uguale al caso peggiore del ritardo di processamento più due volte il caso peggiore nel ritardo di trasmissione(ignorando i clock drifts). Utilizzando il proprio clock, il mittente può misurare il ritardo nel caso peggiore richiesto per ottenere una risposta e determinare un guasto in assenza di quest'ultima all'interno dell'intervallo di timeout: l'individuazione di un guasto di solito avvia una procedura correttiva. Incapsuliamo tale modo di individuare i guasti in un sistema sincrono attraverso l'utilizzo di un'astrazione chiamata *perfect failure detector*.

Specifica: Il *perfect failure detector* fornisce in output, ad ogni processo, l'insieme dei processi che ha individuato essere guasti. Un perfect failure detector può essere descritto attraverso le proprietà di *accuracy* e di *completeness* dichiarate nel modulo 3.4. L'atto di

individuare un guasto coincide con la generazione di un evento di crash (modulo 3.4): una volta che il guasto di un processo p è individuato da un qualche processo q, *l'individuazione è permanente*, ovvero esso non cambierà più idea.

Module:

Name: PerfectFailureDetector ($\mathcal{P}$).

Events:

Indication: $\langle$ *crash* | p_i $\rangle$: Used to notify that process p_i has crashed.

Properties:

PFD1: *Strong completeness:* Eventually every process that crashes is permanently detected by every correct process.

PFD2: *Strong accuracy:* If a process p is detected by any process, then p has crashed.

Modulo 3.4 – Specifica di Perfect Failure Detector

Algoritmo: l'algoritmo 3.3 implementa un perfect failure detector facendo l'assunzione di sistema sincrono. I link di comunicazione non perdono messaggi spediti da un processo corretto ad un altro processo corretto (pp2pLink) ed il periodo di trasmissione di ogni messaggio è limitato da un upper bound conosciuto, in confronto al quale il tempo di processamento locale di un processo e i clock drifts sono trascurabili. L'algoritmo fa uso di uno specifico meccanismo di timeout inizializzato con un intervallo scelto per essere abbastanza esteso da permettere ad ogni processo di avere tempo a sufficienza per inviare messaggi a tutti, e da permettere ad ognuno di questi messaggi di arrivare ed essere consegnati a destinazione. Ogniqualvolta il periodo di timeout scade, viene generato uno specifico evento TimeDelay. C'è da notare che l'algoritmo genera un numero infinito di eventi di failure detection per ogni processo guasto. Si può facilmente modificare l'algoritmo in modo tale da evitare di generare più eventi di crash per lo stesso processo p_i , inserendo un riferimento a quest'ultimo all'interno di una variabile locale suspected e controllandone il contenuto prima della generazione di ogni evento di failure detection.

Correttezza: Consideriamo la proprietà di *strong completeness* di un perfect failure detector. Se un processo p si guasta, esso smette di inviare messaggi di heartbeat e nessuno consegnerà i suoi messaggi: ricorda che i perfect links assicurano che nessun messaggio è consegnato senza che esso sia stato precedentemente inviato. Ogni processo corretto individuerà così il guasto di p. Consideriamo ora la proprietà di *strong accuracy* di un perfect failure detector. Il guasto di p è individuato da un qualche altro processo q, se e solo se q non consegna un messaggio da p durante l'intervallo di timeout. Questo può solo accadere se p si è effettivamente guastato perché l'algoritmo assicura che altrimenti p avrebbe spedito il messaggio e le assunzioni di sincronia implicano che questo sarebbe stato

consegnato prima della scadenza del timeout.

```
Implements:
    PerfectFailureDetector (P).

Uses:
    PerfectPointToPointLinks (pp2p).

upon event ⟨ Init ⟩ do
    alive := Π;
    detected := ∅;
    startTimer (TimeDelay);

upon event ⟨ Timeout ⟩ do
    forall p_i ∈ Π do
        if (p_i ∉ alive) and (p_i ∉ detected) then
            detected := detected ∪ { p_i };
            trigger ⟨ crash | p_i ⟩;
        trigger ⟨ pp2pSend | p_i, [HEARTBEAT] ⟩;
    alive := ∅;
    startTimer (TimeDelay);

upon event ⟨ pp2pDeliver | src, [HEARTBEAT] ⟩ do
    alive := alive ∪ { src };
```

Algoritmo 3.3 – Implementazione PFD basata su timeout

Eventually perfect failure detection

Proprio come è possibile incapsulare le assunzioni temporali di un sistema sincrono in un'astrazione di perfect failure detector, è possibile incapsulare quelle di un sistema parzialmente sincrono all'interno di un'astrazione di *eventually perfect failure detector*.

Specifica: Di base, l'astrazione di eventually perfect failure detector garantisce che c'è un tempo t dopo il quale i guasti possono essere accuratamente individuati. Viene così catturata l'intuizione che gli intervalli di timeout possono essere regolati in modo tale da individuare accuratamente i guasti. Ci sono però periodi in cui l'asincronia del sottostante sistema impedisce alla failure detection di essere accurata portando alla generazione di *falsi sospetti*. In questo caso, parliamo di *failure suspicion* invece di failure detection. Più precisamente, per implementare un'astrazione di eventually perfect failure detector, l'idea è di utilizzare i timeouts e di sospettare i processi che non inviano messaggi di heartbeat all'interno dell'intervallo di timeout. Un processo p potrebbe però sospettare un processo q, anche se q non si è guastato, semplicemente perché l'intervallo scelto da p per sospettare q era troppo corto. In questo caso il sospetto di p riguardo a q è quindi falso. Quando p riceve un messaggio da q, e se sia p che q sono corretti questo sicuramente accadrà, p cambierà il suo giudizio e smetterà di sospettare q. A seguito di un falso sospetto il processo p incrementa anche il suo intervallo di timeout: questo perché p non conosce a priori l'upper bound effettivo per il ritardo di comunicazione; sa però che ne esiste

uno. Chiaramente, se q ora si guasta, p alla fine sospetterà q e non cambierà mai più il suo giudizio. Se q non si guasta, allora c'è un istante t dopo il quale p smetterà di sospettare erroneamente q, in quanto prima o poi il ritardo di timeout utilizzato da p per sospettare q sarà sufficientemente ampio da evitare falsi sospetti, dato che p lo incrementa ogni volta che ne rileva uno. Tutto ciò è possibile in quanto si è fatta l'assunzione che esiste un istante t dopo il quale il sistema è sincrono. Un eventually perfect failure detector può essere descritto dalle proprietà di *accuracy* e *completeness* (**EPFD1-2**) del Modulo 3.5. Si dice che un processo q sospetta un processo p ogniqualvolta q genera l'evento *suspect(p)* e non genera successivamente l'evento *restore(p)*.

Module:

Name: EventuallyPerfectFailureDetector ($\Diamond\mathcal{P}$).

Events:

Indication: $\langle$ *suspect* | p_i $\rangle$: Used to notify that process p_i is suspected to have crashed.

Indication: $\langle$ *restore* | p_i $\rangle$: Used to notify that process p_i is not suspected anymore.

Properties:

EPFD1: *Strong completeness:* Eventually, every process that crashes is permanently suspected by every correct process.

EPFD2: *Eventual strong accuracy:* Eventually, no correct process is suspected by any correct process.

Modulo 3.5 – Specifica di Eventually Perfect Failure Detector

Algoritmo: l'algoritmo 3.4 implementa un eventually perfect failure detector assumendo un sistema parzialmente sincrono. Come per l'algoritmo 3.3, fa uso di uno specifico meccanismo di timeout inizializzato con un intervallo predefinito. La differenza principale è che qui l'intervallo di timeout incrementa ogniqualvolta un processo si rende conto di aver falsamente sospettato un processo che in realtà è corretto.

Correttezza: La proprietà di *eventual strong completeness* è soddisfatta come per l'algoritmo 3.3. Se un processo si guasta allora smette di inviare messaggi, di conseguenza sarà sospettato da ogni processo e nessuno di questi cambierà più il suo giudizio. Consideriamo ora la proprietà di *eventual strong accuracy*. Considerare anche l'istante t dopo il quale il sistema diventa sincrono e l'intervallo di timeout diventa più esteso del ritardo di trasmissione dei messaggi (più i clock drifts e i periodi di processamento locali). Dopo t, qualsiasi messaggio spedito da un processo corretto ad un altro processo corretto è consegnato entro l'intervallo di timeout. Così alla fine ogni processo corretto sospetterà per sempre i processi guasti e reintegrerà tutti quelli che erroneamente sono stati sospettati.

```
Implements:
    EventuallyPerfectFailureDetector (◇P).

Uses:
    PerfectPointToPointLinks (pp2p).

upon event ⟨ Init ⟩ do
    alive := Π;
    suspected := ∅;
    period := TimeDelay;
    startTimer (period);

upon event ⟨ Timeout ⟩ do
    if (alive ∩ suspected) ≠ ∅ then
        period := period + Δ;
    forall p_i ∈ Π do
        if (p_i ∉ alive) ∧ (p_i ∉ suspected) then
            suspected := suspected ∪ {p_i};
            trigger ⟨ suspect | p_i ⟩;
        else if (p_i ∈ alive) ∧ (p_i ∈ suspected) then
            suspected := suspected \ {p_i};
            trigger ⟨ restore | p_i ⟩;
        trigger ⟨ pp2pSend | p_i, [HEARTBEAT] ⟩;
    alive := ∅;
    startTimer (period);

upon event ⟨ pp2pDeliver | src, [HEARTBEAT] ⟩ do
    alive := alive ∪ {src};
```

Algoritmo 3.4 – Implementazione EPFD basata su intervalli di timeout incrementali

Eventual leader election

Spesso, si potrebbe aver bisogno non di individuare quali siano i processi guasti, ma piuttosto di raggiungere un accordo circa la correttezza di un processo il quale può agire come coordinatore in alcuni passi di un algoritmo distribuito. Questo processo è in un certo senso “confidato” dagli altri processi e eletto come loro leader. L'astrazione di *Leader Election* che presentiamo qui di seguito fornisce proprio tale supporto.

Specifica: l'astrazione di *eventual leader election*, con le proprietà (**CD1-2**) mostrate nel modulo 3.6, e denotata con Ω, incapsula un algoritmo di leader election che assicura che prima o poi i processi corretti eleggeranno lo stesso processo corretto come loro leader. Niente preclude la possibilità per i leaders di cambiare in maniera e per un periodo di tempo arbitrari. Una volta eletto un unico leader, e questo non viene successivamente cambiato, diciamo che il leader si è stabilizzato. Tale stabilizzazione è garantita dalla specifica del Modulo 3.6.

Algoritmo: con un'astrazione di processo di tipo crash-stop Ω può essere ottenuto direttamente da ◇P. Infatti, è sufficiente eleggere il processo con il più basso identificatore tra tutti i processi che non sono sospettati da ◇P. Alla fine esattamente un processo corretto sarà eletto da tutti i processi corretti, supponendo che ne esista almeno

uno. Ω può anche essere implementato con l'astrazione di processo crash-recovery, utilizzando i timeouts e assumendo che il sistema sia parzialmente sincrono. L'algoritmo 3.5 descrive tale implementazione assumendo che almeno un processo sia corretto. Ricordiamo che ciò implica, con l'astrazione di processo di tipo crash-recovery, che almeno un processo non si guasta mai, o che alla fine fa recovery e non si guasta più (in ogni esecuzione dell'algoritmo).

Name: EventualLeaderDetector (Ω).

Events:

Indication: $\langle$ *trust* | p_i $\rangle$: Used to notify that process p_i is trusted to be leader.

Properties:

CD1: *Eventual accuracy:* There is a time after which every correct process trusts some correct process.

CD2: *Eventual agreement:* There is a time after which no two correct processes trust different correct processes.

Modulo 3.6 – Specifica di Eventual Leader Detector

Indichiamo l'insieme di tutti i processi che compongono il sistema con π. Nell'algoritmo 3.5, ogni processo p_i tiene traccia di quante volte esso si è guastato e ha fatto recovery, attraverso una variabile intera chiamata *epoch*. Questa variabile, rappresentante l'*epoch number* di p_i , è recuperata, incrementata e poi memorizzata in memoria stabile ogniqualvolta p_i fa recovery a seguito di un guasto. Ogni processo p_i periodicamente invia a tutti gli altri un messaggio di heartbeat contenente il suo epoch number corrente. Ogni processo p_i mantiene una lista dei possibili processi leader, all'interno della variabile *possible*. Inizialmente, per ogni p_i , *possible* è uguale a π. Poi ogni processo che non comunica periodicamente con p_i viene eliminato da *possible*. L'eliminazione non è però definitiva: dati p_j e p_i , se p_i elimina p_j da possible perché p_j è lento nella comunicazione o perché ha subito un crash con successivo recovery, e successivamente a p_i perviene un messaggio di p_j allora quest'ultimo è semplicemente reinserito in possible, ovvero viene nuovamente considerato come potenziale leader da p_i . Inizialmente, il leader per tutti i processi è lo stesso ed è il processo p_1 . Dopo ogni scadenza di timeout, p_i verifica se p_1 può ancora essere leader. Questo test è effettuato attraverso una funzione select che ritorna uno tra un insieme di processi, o nulla se l'insieme è vuoto. La funzione è la stessa per tutti i processi e ritorna sempre lo stesso processo (identificatore) per lo stesso dato insieme(alive), in maniera deterministica, seguendo le seguenti regole: *seleziona tra tutti i processi con il più basso epoch number, il processo con il più basso indice (Identificatore).* Questo garantisce che, se un processo p_j è eletto leader, e p_j comincia a fare crash e recover per

sempre, p_j alla fine sarà sostituito da un qualche p_k corretto. Per definizione, l'epoch number di un processo corretto alla fine smetterà di essere incrementato. Il suo periodo di timeout viene incrementato ogni volta che c'è un cambiamento di leader. Questo garantisce che, alla fine, se i leaders vengono cambiati perché l'intervallo di partenza era troppo corto rispetto ai ritardi di comunicazione, il periodo sarà incrementato e diverrà sufficientemente largo da permettere la stabilizzazione del leader quando il sistema diventa sincrono.

```
Implements:
    EventualLeaderDetector (Ω).

Uses:
    FairLossPointToPointLinks (flp2p)

upon event ⟨ Init ⟩ do
    leader := p1; trigger ⟨ trust | leader ⟩;
    epoch := 0; period := TimeDelay;
    forall pi ∈ Π do
        trigger ⟨ sp2pSend | pi, [HEARTBEAT, epoch] ⟩;
    candidateset := ∅; startTimer (period);

upon event ⟨ Recovery ⟩ do
    leader := p1; trigger ⟨ trust | leader ⟩;
    period := TimeDelay; retrieve(epoch); epoch := epoch + 1; store(epoch);
    forall pi ∈ Π do
        trigger ⟨ sp2pSend | pi, [HEARTBEAT, epoch] ⟩;
    candidateset := ∅; startTimer (period);

upon event ⟨ Timeout ⟩ do
    newleader = select(candidateset);
    if (leader ≠ newleader) then
        period := period + Δ;
        leader := newleader;
        trigger ⟨ trust | leader ⟩;
    forall pi ∈ Π do
        trigger ⟨ sp2pSend | pi, [HEARTBEAT, epoch] ⟩;
    candidateset := ∅; startTimer (period);

upon event ⟨ flp2pDeliver | src, [HEARTBEAT, epc] ⟩ do
    if exists (s, e) ∈ candidateset such that (s=src) ∧ (e<epc) then
        candidateset := candidateset \ {(s, e)};
    candidateset := candidateset ∪ {(src, epc)};
```

Algoritmo 3.5 – Implementazione di Eventual Leader Detector basata su epoch number

<u>*Correttezza:*</u> consideriamo la proprietà di *eventual accuracy* e assumiamo per contraddizione che c'è un tempo dopo il quale un processo corretto p_i permanentemente confida un processo guasto, diciamo p_j . Ci sono due casi da considerare (ricordiamo che stiamo considerando l'astrazione crash-recovery):

1) il processo p_j alla fine si guasta e non fa più recovery
2) p_j è soggetto ad una sequenza infinita di crash e recover.

Nel caso 1) p_j invierà a p_i un numero finito di messaggi di heartbeat. Per la proprietà di no creation e finite duplication dei links

(fair loss) sottostanti, c'è un tempo dopo il quale p_i smette di consegnare messaggi da p_j . Alla fine quest'ultimo sarà escluso dall'insieme (*possible*) dei potenziali leader per p_i che eleggerà un nuovo leader. Consideriamo ora il caso 2) dato che p_j comincia a guastarsi e a fare recovery per sempre il suo epoch number crescerà in maniera indefinita. Se p_k è un processo corretto, allora c'è un tempo dopo il quale il suo epoch number diventerà più piccolo di quello di p_j . Dopo questo tempo, o p_i smetterà di consegnare messaggi provenienti da p_j , e ciò può accadere se p_j si guasta e fa recovery così velocemente che non ha abbastanza tempo per spedire sufficienti messaggi a p_i (ricordare che con i link di tipo fair loss, un messaggio è garantito essere consegnato dal suo destinatario solo se è spedito infinite volte), o p_i consegna messaggi da p_j ma con un più alti epoch numbers rispetto a quello di p_k . In entrambi i casi p_i smette di confidare in p_j . Ogni processo p_i quindi alla fine confiderà solo processi corretti. Consideriamo ora la proprietà di *eventual agreement*. E' necessario spiegare perché c'è un tempo t dopo il quale non ci sono due processi differenti che sono confidati da due distinti processi corretti. Consideriamo un sottoinsieme S dei processi corretti in una data esecuzione. Consideriamo ulteriormente il tempo dopo il quale:

a) il sistema diventa sincrono,

b) i processi in S non si guastano più,

c) i loro epoch numbers smettono di incrementarsi per ogni processo,

d) per ogni processo corretto p_i ed ogni processo guasto p_j , p_i smette di consegnare messaggi da p_j o l'epoch number di p_j presso p_i diventa strettamente maggiore del più grande epoch number appartenente ai processi in S presso p_i .

Per l'assunzione di sistema parzialmente sincrono, le proprietà del sottostante canale fair-loss e l'algoritmo, questo istante prima o poi sarà raggiunto. Dopo che ciò accade, ogni processo che è confidato da un processo corretto sarà uno di quelli presenti in S. Per la funzione select (deterministica) tutti i processi corretti confideranno nello stesso processo all'interno dell'insieme.

3.1.5 Broadcast

Un'astrazione di broadcast permette ad un processo di inviare un messaggio, attraverso un'unica operazione, a tutti i processi nel sistema, incluso se stesso. Diamo di seguito la specifica e un

algoritmo per realizzare una forma debole di broadcast chiamata *Best Effort Broadcast*.

Best Effort Broadcast

Name: BestEffortBroadcast (beb).

Events:

Request: ⟨ *bebBroadcast* | m ⟩: Used to broadcast message m to all processes.

Indication: ⟨ *bebDeliver* | src, m ⟩: Used to deliver message m broadcast by process *src*.

Properties:

BEB1: *Best-effort validity:* For any two processes p_i and p_j. If p_i and p_j are correct, then every message broadcast by p_i is eventually delivered by p_j.

BEB2: *No duplication:* No message is delivered more than once.

BEB3: *No creation:* If a message m is delivered by some process p_j, then m was previously broadcast by some process p_i.

Modulo 3.7 – Specifica di Best Effort Broadcast

Specifica: Con il best effort broadcast la responsabilità di assicurare l'affidabilità delle consegne dei messaggi ricade soltanto sul mittente. I restanti processi non vengono coinvolti dagli aspetti di affidabilità nella consegna dei messaggi ricevuti. Non sono offerte garanzie di consegna nel caso in cui il mittente si guasti. Più precisamente, il best effort broadcast è caratterizzato dalle proprietà **BEB1-3** presenti nel modulo 3.7. La BEB1 è una proprietà di liveness mentre le BEB2-3 sono proprietà di safety. Va sottolineato che i messaggi spediti in broadcast sono implicitamente inviati a tutti i processi e che ogni messaggio è univocamente identificato.

Implements:
BestEffortBroadcast (beb).

Uses:
PerfectPointToPointLinks (pp2p).

upon event ⟨ *bebBroadcast* | m ⟩ **do**
forall $p_i \in \Pi$ **do**
trigger ⟨ *pp2pSend* | p_i, m ⟩;

upon event ⟨ *pp2pDeliver* | p_i, m ⟩ **do**
trigger ⟨ *bebDeliver* | p_i, m ⟩;

Algoritmo 3.6 – Implementazone di Best Effort Broadcast al di sopra di Perfect links

Algoritmo: Forniamo un algoritmo che implementa il best effort broadcast utilizzando l'astrazione di perfect link. Questo algoritmo non fa alcuna assunzione sulla failure detection: è un algoritmo fail-silent. Fornire l'astrazione di best effort broadcast al di sopra di quella di perfect link è molto semplice. E' sufficiente spedire una copia del

messaggio a ogni processo nel sistema, come mostrato nell'algoritmo 3.6 e in figura 3.4. Fino a quando il mittente del messaggio non si guasta, le proprietà dell'astrazione di perfect link assicurano che tutti i processi corretti consegneranno il messaggio.

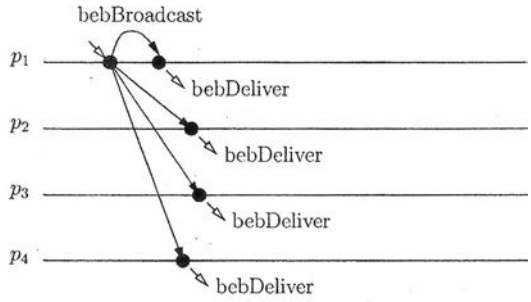


Figura 3.4 – Esempio di invio di un messaggio in broadcast

Correttezza: le proprietà sono derivate in maniera triviale da quelle dei links perfect point-to-point. La *no duplication* e la *no creation* sono proprietà di safety che derivano dalle proprietà PL2 e PL3 (vedi modulo 3.3). La *validity* è una proprietà di liveness che è derivata dalla proprietà PL1 (vedi modulo 3.3) e dal fatto che il mittente spedisce il messaggio a ogni altro processo nel sistema.

Performance: L'algoritmo richiede un singolo passo di comunicazione e scambia N messaggi, dove N è il totale dei processi presenti nel sistema

3.1.6 Modelli di sistemi distribuiti

Una combinazione di 1) astrazione di processo, 2) astrazione di link e 3) (possibilmente) un'astrazione di failure detector, definisce un modello di sistema distribuito. Nel seguito presenteremo alcuni modelli, più precisamente, tra tutti i possibili, ne descriveremo cinque che sono particolarmente interessanti. Descriveremo anche alcune importanti proprietà di specifiche di astrazioni e di algoritmi.

Combinare le astrazioni

Chiaramente non considereremo tutte le possibili combinazioni delle astrazioni di base. D'altro canto, è interessante esaminare più di una possibile combinazione per avere un'idea di come certe assunzioni influenzino il progetto degli algoritmi.

Abbiamo selezionato cinque combinazioni specifiche per definire altrettanti differenti modelli:

Fail-Stop: consideriamo l'astrazione di processo di tipo *crash-stop*, dove i processi eseguono l'algoritmo deterministico ad essi

assegnato, a meno che non si guastino, nel qual caso non fanno recovery. I links sono di tipo *perfect*. Infine assumiamo l'esistenza di un *perfect failure detector*. In generale, tali assunzioni permettono di progettare algoritmi distribuiti più semplici rispetto a quelli progettati per i modelli che seguono.

Fail-Silent: consideriamo anche qui l'astrazione di processo *crash-stop* insieme all'astrazione di canale di tipo *perfect-link*. Differentemente da prima però qui non assumiamo alcuna astrazione di failure detection: i processi non hanno quindi modo di avere informazioni circa lo stato di salute degli altri.

Fail noisy: questo modello è in qualche modo una via di mezzo tra i due precedenti. Consideriamo anche qui astrazioni di processo di tipo *crash-stop* insieme con astrazioni di tipo *perfect-link*. In più, assumiamo qui l'esistenza di un *eventually perfect failure detector* del tipo descritto dal modulo 3.4 e/o di un *eventual leader detector* (modulo 3.6).

Fail recovery: consideriamo qui astrazioni di processo di tipo *crash-recovery,* in accordo con il fatto che i processi possono guastarsi, recuperare e partecipare ancora alla computazione. Algoritmi sviluppati con queste astrazioni richiedono l'utilizzo di tecniche per la gestione di memoria stabile, che comportano la necessità di dover affrontare difficoltà legate alle amnesie, ovvero al fatto che un processo potrebbe dimenticare ciò che ha fatto prima di guastarsi. I links si assumono essere di tipo *stubborn* e noi potremmo fare affidamento sull'*eventual leader detector* del modulo 3.6

Randomized: considereremo qui una caratteristica particolare nell'astrazione di processo: gli algoritmi possono non essere deterministici. I processi potrebbero utilizzare un oracolo randomico per scegliere tra numerosi passi da eseguire. Tipicamente, il corrispondente algoritmo implementa una data astrazione con una qualche probabilità.

E' importante puntualizzare che non tutte le astrazioni sono implementabili in tutti i modelli. Per esempio, l'astrazione di coordinazione non ha una soluzione nel modello fail-silent e non è neanche chiaro come poter realizzare soluzioni randomizzate significative. Per altre invece le soluzione potrebbe esistere ma la sua realizzazione è ancora un'area attiva di ricerca. Questo è per esempio il caso di soluzioni randomizzate dell'astrazione di memoria condivisa.

3.1.7 Misura delle performance

Nel presentare un algoritmo che implementa una data astrazione, è importante analizzare i suoi costi. Per fare ciò possiamo utilizzare due indici:

1) il numero di messaggi richiesti per terminare un'operazione dell'astrazione

2) il numero di passi di comunicazione richiesti per terminare l'operazione.

Quando si valutano le performance degli algoritmi distribuiti nel modello crash-recovery, insieme al numero di passi di comunicazione e al numero dei messaggi, va considerato anche:

3) il numero di accessi alla memoria stabile

In generale vengono contati i messaggi, i passi di comunicazione, il numero di accessi al disco durante specifiche esecuzioni dell'algoritmo, specialmente quelle durante le quali non occorrono guasti. Queste ultime nella realtà si manifestano con probabilità più elevata e sono quelle per le quali l'algoritmo viene ottimizzato. Ha senso pianificare per il caso peggiore, fornendo all'algoritmo gli strumenti necessari a tollerare i guasti, e sperare per il meglio, ottimizzando per i casi in cui non si verificano guasti. Quegli algoritmi le cui performance degradano proporzionalmente con l'incremento del numero dei guasti sono qualche volta chiamati algoritmi *gracefully degrading*. Precisi studi delle performance aiutano a selezionare il migliore algoritmo per una data astrazione all'interno di uno specifico ambiente e richiedono analisi real-time.

3.1.8 Consenso e risultati di impossibilità

Nei paragrafi precedenti si è accennato al problema del consenso e ai risultati di impossibilità ad esso associati. Vediamo ora tali aspetti in maniera un po' più dettagliata.

Problema del consenso

Questo problema è definito nel seguente modo: i processi facenti parte di un gruppo devono accordarsi su un valore, ad esempio commit/abort di una transazione. E' l'astrazione di una classe di problemi in cui i processi partono con una loro “proposta” (ognuna delle quali può essere differente da quella degli altri) e devono accordarsi su un'unica opinione comune. Si tratta di un problema fondamentale in quanto qualsiasi soluzione per la mutua esclusione, leader election, comunicazione totalmente ordinata risolve anche il consenso. In sostanza quest'ultimo può essere ridotto a gran parte dei problemi che si incontrano durante la progettazione di algoritmi per sistemi distribuiti. Di seguito vedremo algoritmi e modelli minimali per risolvere questo problema in presenza di guasti.

Risultato di impossibilità

Parliamo ora del risultato di impossibilità legato al consenso. Partiamo con un esempio: un castello è assediato da quattro eserciti alleati, ognuno dei quali è guidato da un generale. Per riuscire a conquistare il castello devono attaccare tutti insieme. Per poter combattere, un esercito ha bisogno di un generale, se quindi questo viene a mancare, dato che può essere ucciso, l'esercito non può più scendere in battaglia. Supponiamo che le comunicazioni tra i generali siano affidabili, ma il tempo impiegato dal messaggio per giungere da un generale all'altro è impredicibile. Ora i generali devono accordarsi, quindi devono raggiungere un consenso, per decidere tra attacco o ritirata. Se ci troviamo nella situazione in cui un generale non fa pervenire la propria risposta, allora i restanti non possono determinare se tale mancanza sia dovuta al fatto che le comunicazioni sono lente o al fatto che il generale in questione è stato ucciso. Alla fine quindi non riusciranno a prendere una decisione. Enunciamo ora il risultato di impossibilità:

E' impossibile risolvere il problema del consenso in modo deterministico in sistemi asincroni in cui anche un solo processo può guastarsi per crash (FLP result).

Tale risultato è stato dimostrato da Fisher, Lynch e Patterson nel 1985 [1]. La tabella successiva mostra la relazione tra modelli di sistema ed il precedente risultato.

Sistema Asincrono	**Sistema Sincrono**
E' un modello attraente in quanto non vengono fatte assunzioni di sincronia	E' un modello attraente per la tolleranza ai guasti
E' debole sotto l'aspetto della tolleranza ai guasti: Il consenso non può essere risolto in questo modello, anche supponendo che esiste un solo processo che possa guastarsi e che i canali siano affidabili	E' debole per l'assunzione di sincronia: è difficile che un sistema reale riesca a rispettare sempre le assunzioni che caratterizzano questo modello di sistema.
Tabella 3.1 – Relazione tra modelli di sistema e risultato di impossibilità	

E' possibile aggirare l'FLP result ricorrendo al modello parzialmente sincrono: vengono fatte alcune assunzioni di sincronia necessarie per risolvere il consenso, ad esempio:

- processi con velocità limitata e clock sincronizzati, oppure
- ritardo dei messaggi limitato ma sconosciuto

Si sfrutta la sincronia parziale esibita dalla maggioranza dei sistemi reali, ovvero ritardo dei messaggi e/o tempo di transizione di stato dei processi limitato nella vasta maggioranza dei casi. In un modello di sistema parzialmente sincrono il consenso e i problemi ad esso correlati vengono risolti usando timeouts per rilevare i processi guasti.

3.1.9 Consenso, specifiche ed algoritmi

Esaminiamo ora delle specifiche formali del problema del consenso. In particolare tratteremo lo *Uniform Consensus* e il *Regular Consensus,* chiamato anche *Non Uniform Consensus*.

Regular Consensus

E' specificato in termini di due primitive: *propose* e *decide*. Ogni processo possiede un valore iniziale il quale viene proposto a tutti gli altri attraverso la primitiva propose. Tale valore è “privato” e l'atto di proposta è locale. Quest'ultimo tipicamente genera eventi di broadcast attraverso i quali i processi si scambiano i valori proposti con lo scopo di raggiungere alla fine un accordo. Tutti quelli corretti devono decidere su un unico valore, attraverso la primitiva decide. Questo valore deciso deve essere uno di quelli proposti. Il consenso deve soddisfare le proprietà C1-4 elencate nel modulo 3.8.

Module:

Name: (regular) Consensus (c).

Events:

Request: ⟨ *cPropose* | v ⟩: Used to propose a value for consensus.

Indication: ⟨ *cDecide* | v ⟩: Used to indicate the decided value for consensus.

Properties:

C1: *Termination:* Every correct process eventually decides some value.

C2: *Validity:* If a process decides v, then v was proposed by some process.

C3: *Integrity:* No process decides twice.

C4: *Agreement:* No two correct processes decide differently.

Modulo 3.8 – Regular Consensus

Algoritmo fail-stop: consenso con diffusione

L'algoritmo 3.6 utilizza, accanto al perfect failure detector, un'astrazione di comunicazione di tipo *best-effort broadcast*. L'idea di base dell'algoritmo è la seguente. I processi seguono rounds sequenziali. Ogni processo mantiene un insieme di valori proposti, chiamato *proposal*, che esso ha visto, il quale di solito è incrementato ogni volta che si passa da un round al seguente (se si viene a conoscenza di nuovi valori proposti). In ogni round, ogni processo propaga il proprio insieme a tutti gli altri, utilizzando l'astrazione di best-effort broadcast, ovvero il sistema viene inondato da tutte le proposte che i processi hanno visto nei round precedenti. Quando un insieme di proposte è ricevuto, esso viene fuso con proposal. Ovvero, in ogni round, viene calcolata l'unione di tutti gli insiemi di valori proposti ricevuti fino a quel momento. Un processo decide uno specifico valore tra quelli contenuti nel suo insieme proposal quando è

sicuro di aver ricevuto tutte le possibili proposte visibili da ogni processo corretto. Di seguito illustriamo alcuni passi fondamentali dell'algoritmo.

1) *Terminazione di un round e passaggio al round successivo*: ogni messaggio è etichettato con il numero di round durante il quale è stato inviato in broadcast. Un round termina, presso p_i quando riceve un messaggio da ogni processo che non è stato sospettato durante il round corrente. Ovvero, un round non viene lasciato a meno che non vengano ricevuti messaggi, etichettati con quel numero di round, da tutti i processi che non sono stati sospettati durante il round corrente.

2) *Determinazione del momento giusto per prendere la decisione:* la decisione di consenso deve essere presa quando un processo sa di avere lo stesso insieme di valori proposti di tutti i processi corretti. In un round dove viene individuato un nuovo guasto, un processo p_i non può essere sicuro di avere esattamente lo stesso insieme di valori di tutti gli altri. Questo potrebbe accadere perché il processo che si è guastato potrebbe aver propagato alcuni valori i quali sono pervenuti a tutti tranne che a p_i . Per poter determinare il momento in cui è sicuro prendere una decisione, ognuno tiene traccia dei processi che non ha sospettato nel precedente round, e del numero di quelli dai quali ha ricevuto una proposta nel round corrente. Se un round termina con lo stesso numero di processi non sospettati del round precedente, allora è possibile prendere una decisione.

3) *Selezione della proposta:* per prendere una decisione, un processo può applicare una qualche funzione deterministica all'insieme dei valori accumulati, naturalmente tale funzione dovrà essere la stessa per tutti. Nel nostro caso, il processo decide per il valore *minimo*: assumiamo implicitamente qui che l'insieme di tutte le possibili proposte sia totalmente ordinato e che la relazione d'ordine sia conosciuta da tutti. Un processo che decide, dissemina la decisione a tutti gli altri utilizzando l'astrazione di best-effort broadcast.

Un'esecuzione dell'algoritmo è illustrata in figura 3.5. Il processo p_1 si guasta durante il primo round dopo aver inviato la propria proposta. Nessun altro successivamente si guasta. p_2 vede la proposta di tutti i processi e può quindi decidere. Questo perché l'insieme da cui ha ricevuto la proposta nel primo round è uguale all'insieme iniziale dei processi con cui è iniziato l'algoritmo. p_2 prende il minimo dei valori

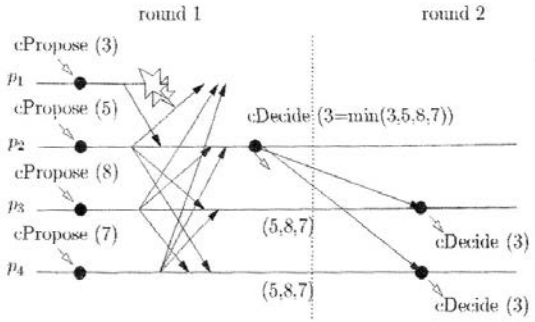


Figura 3.5 – Esempio di esecuzione con guasto

proposti e decide quindi per il valore 3. p_3 e p_4 individuano il guasto di p_1 e non avendo ricevuto la sua proposta, non possono decidere.

Implements:
Consensus (c);

Uses:
BestEffortBroadcast (beb);
PerfectFailureDetector ($\mathcal{P}$);

upon event ⟨ *Init* ⟩ **do**
correct := correct-this-round[0] := Π;
decided := $\perp$; round := 1;
for $i = 1$ **to** N **do**
correct-this-round[i] := proposal-set[i] := $\emptyset$;

upon event ⟨ *crash* | p_i ⟩ **do**
correct := correct \{p_i};

upon event ⟨ *cPropose* | v ⟩ **do**
proposal-set[1] := proposal-set[1] ∪ {v};
trigger ⟨ *bebBroadcast* | [MySet, 1, proposal-set] ⟩;

upon event ⟨ *bebDeliver* | p_i, [MySet, r, set] ⟩ **do**
correct-this-round[r] := correct-this-round[r] ∪ {p_i};
proposal-set[r] := proposal-set[r] ∪ set;

upon correct ⊂ correct-this-round[round] ∧ (decided = $\perp$) **do**
if (correct-this-round[round] = correct-this-round[round-1]) **then**
decided := *min* (proposal-set[round]);
trigger ⟨ *cDecide* | decided ⟩;
trigger ⟨ *bebBroadcast* | [Decided, decided] ⟩;
else
round := round +1;
trigger ⟨ *bebBroadcast* | [MySet, round, proposal-set[round-1]] ⟩;

upon event ⟨ *rbDeliver* | p_i, [Decided, v] ⟩ ∧ p_i ∈ correct ∧ (decided = $\perp$) **do**
decided := v;
trigger ⟨ *cDecide* | v ⟩;
trigger ⟨ *bebBroadcast* | [Decided, decided] ⟩;

Algoritmo 3.7 – Flooding Regular Consensus algorithm

Così avanzano al round successivo, il round 2. Notare che se uno di

questi due processi decide per il valore minimo tra le proposte in loro possesso dopo il round 1, il valore deciso sarebbe differente rispetto a quello di p_2 , ovvero 5. Dato che p_2 ha deciso, egli dissemina la sua decisione attraverso un best-effort broadcast. Quando la decisione è consegnata , anche i processi p_3 e p_4 decidono per il valore 3.

Correttezza: le proprietà di *Validity* ed *Integrity* seguono dall'algoritmo e dalle proprietà delle astrazioni di comunicazione. La *Termination* segue dal fatto che alla fine al round N, dove N è il numero totale di processi presenti nel sistema, tutti prendono una decisione. L'*Agreement* è assicurato perché la funzione minimo è deterministica ed è applicata da tutti i processi corretti sullo stesso insieme.

Performance: se non ci sono guasti, l'algoritmo richiede un singolo passo di comunicazione: tutti i processi decidono alla fine del round 1. Ogni guasto potrebbe causare al più un passo di comunicazione addizionale. Quindi nel caso peggiore l'algoritmo richiede N passi, se N-1 processi si guastano in sequenza. Nel caso migliore l'algoritmo scambia $2\mathrm{N}^2$ messaggi, ai quali vanno aggiunti N^2 messaggi scambiati per ogni round aggiuntivo dovuto al guasto di un processo.

Uniform Consensus

E' la variante uniforme del consenso. La sua specifica è presentata nel modulo 3.9: i processi corretti decidono un valore che deve essere consistente con i valori decisi da eventuali processi che potrebbero aver preso una decisione prima di guastarsi. In breve, lo *uniform consensus* assicura che non esistono due processi, corretti o no, che decidono per valori differenti.

Module:

Name: UniformConsensus (uc).

Events:

⟨ *ucPropose* | *v* ⟩, ⟨ *ucDecide* | *v* ⟩: with the same meaning and interface as the consensus interface.

Properties:

C1-C3: from consensus.

C4': *Uniform Agreement:* no two processes decide differently.

Modulo 3.9 – Uniform Consensus

L'algoritmo 3.7 non assicura uniform agreement, sostanzialmente perché alcuni decidono troppo presto: senza essere sicuri che la loro decisione sia stata vista da un numero sufficiente di processi. Consideriamo ad esempio lo scenario in cui p_1 , alla fine del round 1, riceve messaggi da tutti. Assumiamo ulteriormente che p_1 decida il proprio valore e questo sia quello più basso. Assumiamo inoltre che p_1 si guasti dopo aver deciso e i suoi messaggi di decisione non

raggiungano nessun altro processo. Il resto dei processi passa al round 2 senza aver ricevuto i messaggi di p_1 . Di nuovo, i processi passano a decidere un qualche altro valore che nel nostro scenario risulta essere differente da quello che aveva deciso p_1 . L'algoritmo seguente può essere visto come una variante del 3.7, e lo chiameremo algoritmo di *flooding uniform consensus*. L'algoritmo 3.8 implementa un consenso uniforme. Ogni processo segue round sequenziali. Come nell'algoritmo di flooding regular consensus, ognuno mantiene un insieme di proposte ricevute e tale insieme viene disseminato verso gli altri processi, utilizzando una primitiva di best-effort broadcast. Un'importante differenza con l'algoritmo 3.7 è che tutti attendono fino al round N prima di prendere una decisione.

```
Implements:
    UniformConsensus (c);

Uses:
    BestEffortBroadcast (beb).
    PerfectFailureDetector (P);

upon event ⟨ Init ⟩ do
    correct := Π; round := 1; decided := ⊥; proposal-set := ∅;
    for i = 1 to N do delivered[i] := ∅;

upon event ⟨ crash | p_i ⟩ do
    correct := correct \ {p_i};

upon event ⟨ ucPropose | v ⟩ do
    proposal-set := proposal-set ∪ {v};
    trigger ⟨ bebBroadcast | [MySet, 1, proposal-set] ⟩;

upon event ⟨ bebDeliver | p_i, [MySet, r, newSet] ⟩ do
    proposal-set := proposal-set ∪ newSet;
    delivered[r] := delivered[r] ∪ {p_i};

upon (correct ⊆ delivered[round]) ∧ (decided = ⊥) do
    if round = N then
        decided := min (proposal-set);
        trigger ⟨ ucDecide | decided) ⟩;
    else
        round := round + 1;
        trigger ⟨ bebBroadcast | [MySet, round, proposal-set] ⟩;
```

Algoritmo 3.8 – Flooding Uniform Consensus algorithm

Correttezza: la V*alidity* e l'*Integrity* seguono dall'algoritmo e dalle proprietà del best-effort broadcast. La *Termination* è assicurata perché tutti i processi corretti raggiungono il round N e decidono in tale round. La *Uniform Agreement* è assicurata perché tutti i processi che raggiungono il round N hanno lo stesso insieme di valori.

Performance: L'algoritmo, per prendere una decisione, richiede N passi di comunicazione e $N*(N-1)^2$ messaggi per ogni processo corretto.

Rotating coordinator consensus

Fino ad ora abbiamo affrontato il problema del consenso ipotizzando che il modello di sistema fosse sincrono. In questo paragrafo proponiamo una soluzione adatta ad essere utilizzata in caso di modello di sistema parzialmente sincrono. Il consenso in questo caso può essere realizzato attraverso l'algoritmo *rotating coordinator consesus* (algoritmo 3.9.a/3.9.b). Ogni processo ha accesso ad un modulo D_i di failure detection di tipo ◇P. A differenza degli algoritmi precedenti però questo tollera un numero di processi guasti minore o uguale a t, dove $2*t<N$, con N pari al numero totale di processi presenti nel sistema. In sostanza l'algoritmo richiede l'assunzione di maggioranza corretta. Utilizza il paradigma di *rotating coordinator*: ogni processo è a conoscenza del fatto che durante il round r il ruolo di coordinatore è rivestito dal processo c-esimo, dove $c=(r \bmod n)+1$.

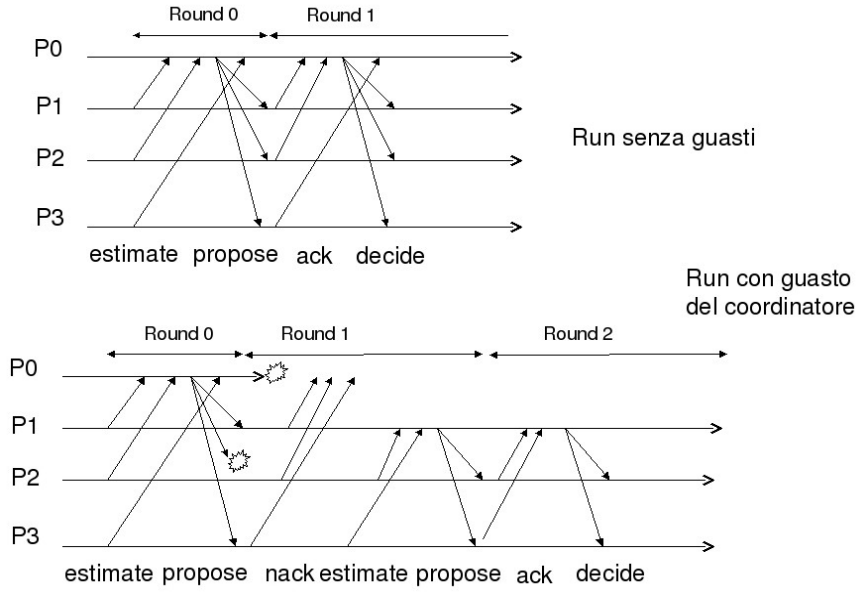


Figura 3.6 – Scenari di funzionamento

Il coordinatore p_c prova a decidere il valore. Se è corretto e non è sospettato l'operazione di decisione avrà successo e p_c invierà in reliable-broadcast il valore deciso. Ogni round dell'algoritmo è suddiviso in 4 step:

1. ogni processo invia la stima corrente del valore deciso (etichettato con un timestamp pari al valore dell'ultimo round in cui il valore è stato aggiornato) al coordinatore c

2. il coordinatore c raccoglie una maggioranza di tali valori stimati, seleziona quello con timestamp più alto e lo invia a tutti i processi come nuova stima

3. per ogni processo p corretto ho due possibilità:

a) p riceve il valore stimato da c e gli invia un ack per indicare che ha adottato questo valore come nuova stima

b) p sospetta c, quindi invia un nack a c

4. c raccoglie sia ack che nack. Se riceve una maggioranza di ack il valore della nuova stima è locked. Il coordinatore lo invia in reliable-broadcast e la decisione è presa. Se invece c riceve almeno un nack allora si passa al round successivo.

Nella figura 3.6 vengono mostrati due scenari, uno con guasto del coordinatore, l'altro senza guasti.

```
Uses:
    PerfectPointToPointLinks (pp2p);
    ReliableBroadcast (rb);
    BestEffortBroadcast (beb);
    EventuallyPerfectFailureDetector (◇P);

function leader (r) returns processid is
    return p_i: (rank(p_i) = (r mod N + 1));

upon event ⟨ Init ⟩ do
    round := 1; proposal := decided := ⊥;
    suspected:= estimate-set[] := ack-set[] := nack-set[] := ∅;
    froall r do estimate[r] := ack[r] := false; proposed[r] := ⊥

upon event ⟨ ucPropose | v ⟩ ∧ proposal ≠ ⊥ do
    proposal := (v,0);

upon event ⟨ pp2pDeliver | p_i, [ESTIMATE, r, e] ⟩ do
    estimate-set[r] := estimate-set[r] ∪ {e};

upon (leader(round)=self) ∧ (|estimate-set[round]| > N/2)) do
    proposal := highest(estimate-set[round]);
    trigger ⟨ bebBroadcast | [PROPOSE, round, proposal] ⟩;

upon event ⟨ pp2pDeliver | p_i, [ACK, r] ⟩ do
    ack-set[r] := ack-set[r] ∪ {p_i};

upon event ⟨ pp2pDeliver | p_i, [NACK, r] ⟩ do
    nack-set[r] := nack-set[r] ∪ {p_i};

upon (leader(round)=self) ∧ nack-set[round] ≠ ∅ do
    round := round + 1;

upon (leader(round)=self) ∧ (|ack-set[round]| > N/2) do
    trigger ⟨ rbBroadcast | [DECIDE, proposal] ⟩;
```

Algoritmo 3.9.a – Rotating Coordinator Consensus algorithm - coordinatore

Correttezza: Le proprietà di *Validity* ed *Integrity* seguono direttamente dalle proprietà dei sottostanti canali di comunicazione. Per quanto riguarda la *Termination*, per l'ipotesi di sistema parzialmente sincrono, ci sarà un tempo t dopo il quale un processo corretto non sarà mai sospettato dagli altri e uno guasto invece verrà sospettato definitivamente. Se l'algoritmo non è già terminato prima di t, allora prima o poi prenderà una decisione e terminerà inviandola in reliable

broadcast. Infine vediamo la proprietà di *Agreement*. Consideriamo due round successivi dove i coordinatori hanno proposto rispettivamente due valori differenti v e v'. Per decidere c'è bisogno di una maggioranza di voti e per definizione l'intersezione di due maggioranze non è mai vuota. Ciò porta ad una contraddizione, in quanto i due valori v e v' non possono essere differenti.

Performance: se nessun processo si guasta o è sospettato per arrivare ad una decisione sono necessari $4*(N-1)$ messaggi per ogni processo corretto.

```
upon event (proposal ≠ ⊥) ∧ (estimate[round] = false) do
    estimate[round] := true;
    trigger ⟨ pp2pSend | leader(round), [ESTIMATE, round, proposal] ⟩;

upon event ⟨ bebDeliver | p_i, [PROPOSE, r, v] ⟩ do
    proposed[r] := v;

upon event (proposed[round]≠ ⊥) ∧ (ack[round] = false) do
    proposal := (proposed[round],round);
    ack[round] := true;
    trigger ⟨ pp2pSend | leader(round), [ACK, round] ⟩;
    round := round + 1;

upon event (leader(round) ∈ suspected) ∧ (ack[round] = false) do
    ack[round] := true;
    trigger ⟨ pp2pSend | leader(round), [NACK, round] ⟩;
    round := round + 1;

upon event ⟨ rbDeliver | p_i, [DECIDED, v] ⟩ ∧ (decided = ⊥) do
    decided := v;
    trigger ⟨ ucDecided | v ⟩;

upon event ⟨ suspect | p_i ⟩ do
    suspected := suspected ∪ {p_i};

upon event ⟨ restore | p_i ⟩ do
    suspected := suspected \ {p_i};
```

Algoritmo 3.9.b – Rotating Coordinator Consensus algorithm - non coordinatore

3.1.10 Finite State Machine (FSM) o automi a stati finiti

Un automa a stati finiti o Finite State Machine è un sistema dinamico, invariante, discreto nell'avanzamento e nelle interazioni nel quale gli insiemi dei possibili valori di ingresso, uscita e stato sono insiemi finiti:

- *dinamico*: evolve nel tempo passando da uno stato all'altro in funzione dei segnali d'ingresso e dello stato precedente;
- *invariante*: a parità di condizioni iniziali il comportamento del sistema è sempre lo stesso;
- *discreto*: le variabili d'ingresso, di stato, d'uscita, possono

assumere solo valori discreti.

L' automa a stati finiti è un modello di calcolo semplice rappresentabile come un piccolo dispositivo, che mediante una testina legge una stringa di input su un nastro e la elabora, facendo uso di un meccanismo molto semplice di calcolo e di una memoria limitata. L'esame della stringa avviene un carattere alla volta attraverso precisi passi computazionali che comportano l'avanzamento della testina. In sostanza un ASF è un caso particolare di macchina di Turing, utilizzato per l'elaborazione di quei linguaggi che nelle Grammatiche di Chomsky sono definiti di Tipo 3 o Regolari. Distinguiamo due tipi di automi a stati finiti: gli automi a stati finiti deterministici (ASFD) e gli automi a stati finiti non deterministici ASFND. La definizione formale di automa segue uno schema che solitamente viene utilizzato anche per definire modelli più generali.

Def. Automa a stati finiti (ASF):

- A = < $\Sigma, K, \delta, q_0, F$ >
- Σ = { $\sigma_{1,\ldots}, \sigma_n$ }: alfabeto di input
- K = { $q_{0,\ldots}, q_m$ } : insieme finito non vuoto di stati
- $F \subseteq K$: insieme di stati finali
- $q_0 \in K$: stato iniziale
- $\delta : K x \Sigma \rightarrow K$: funzione di transizione, funzione totale che determina lo stato successivo

La funzione di transizione di un ASF puo' essere rappresentata mediante matrice (tabella) di transizione o attraverso un diagramma degli stati.

Def. Configurazione di un ASF: < q, x> con q_K stato interno e $x \in \Sigma^*$ sequenza di input. La configurazione di un ASF: < q, x> e':

- iniziale se q = q_0
- finale se x = ε
- accettante se x = ε e $q \in F$

Def. Uno step o transizione di un ASF è un'applicazione della funzione di transizione ad una configurazione (relazione binaria sulle configurazioni): |–

$$<q,x> \vdash <q',x'> \Leftrightarrow x = ax' \wedge \delta(q,a) = q'$$

Def. Step ripetuto: |–*

Gli automi consentono di rappresentare in generale sistemi di transizione in un insieme finito di stati, ad esempio:

- un ascensore
- un distributore di biglietti dell'autobus
- il sistema elettrico di un automobile
- ecc.

Esempio. Automa che riconosce il linguaggio delle parole che contengono un numero pari di a oppure un numero pari di b (ad esempio abbaabb)

$$\Sigma = a, b$$
$$K = q_{0,} q_{1,} q_{2,} q_3$$
$$F = q_{0,} q_{1,} q_2$$

Rappresentazione della funzione di transizione:

Diagramma degli stati:

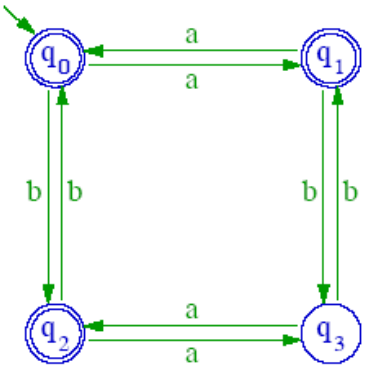


Figura 3.7 – Diagramma degli stati dell'automa che riconosce stringhe che contengono un numero pari di a o un numero pari di b

Matrice di transizione:

	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_3	q_0
q_3	q_2	q_1

Def. Automa a stati finiti non deterministico:

- $A_N = <\Sigma, K, \delta_N, q_0, F>$
- $\Sigma = \{ \sigma_{1,\dots}, \sigma_n \}$: alfabeto di input
- $K = \{ q_{0,\dots}, q_m \}$: insieme finito non vuoto di stati
- $F \subseteq K$: insieme di stati finali
- $q_0 \in K$: stato iniziale
- $\delta_N : K \times \Sigma \rightarrow P(K)$: funzione di transizione, funzione totale che determina l'insieme (eventualmente vuoto) degli stati successivi.

Attenzione a non confondere il non determinismo degli automi con altri concetti affini: il non determinismo dei fenomeni naturali oppure gli aspetti probabilistici che si presentano in sistemi stocastici o ancora il parallelismo di più processi di calcolo simultanei. Il non determinismo ci consente di rappresentare un calcolo anziché come una traiettoria come un albero.

Esempio. ASFND che riconosce le stringhe che terminano con bb o ba o baa.

$$<\{a, b\}, \{q_0, q_1, q_2, q_3\}, \delta, q_0, \{q_3\}>$$

Matrice di transizione:

	a	b
q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2, q_3\}$	$\{q_3\}$
q_2	$\{q_3\}$	$\emptyset$
q_3	$\emptyset$	$\emptyset$

Def. Configurazione di un ASFND: < q, x> con $q \in K$ stato interno e $x \in \Sigma^*$ sequenza di input. La configurazione di un ASFND: < q, x> è:

- iniziale se q = q_0
- finale se x = ε
- accettante se x = ε e $q \in F$

Def. Uno step o transizione di un ASF è un'applicazione della funzione di transizione ad una configurazione (relazione binaria sulle configurazioni): |–

$$<q, x> \vdash - <q', x'> \Leftrightarrow x = ax' \wedge q' \in \delta_N(q, a)$$

Def. Step ripetuto: |–*

Def. Una computazione e' una sequenza di configurazioni $c_0, c_1, \ldots, c_n$ (n ≥ 0) tali che, se n > 0, per ogni i = 0, ..., n-1 si ha che c_i |– $c_{(i+1)}$. (Definizione valida sia per ASF che per ASFND)

NOTA BENE. La chiusura riflessiva e transitiva |-* della relazione binaria |– tra configurazioni indica che esiste una computazione che porta da una configurazione ad un'altra.

Se chiamiamo 'stato non deterministico' un insieme di stati deterministici, il non determinismo degli automi ci consente di rappresentare una computazione in modo equivalente come:

1. una traiettoria nello spazio degli stati non deterministici,
2. un albero nello spazio degli stati deterministici.

Mentre un automa deterministico associa ad una stringa di input una computazione avente struttura lineare, un automa non deterministico associa sia una struttura lineare che una più complessa, detta albero delle computazioni.

4. Tools

4.1 Il simulatore RIFIDI

Rifidi è un IDE open source per applicazioni RFID scritto interamente in Java. In sostanza si tratta di un emulatore software di reader RFID fisici. Permette di realizzare un sistema RFID interamente con componenti software, eliminando di conseguenza le dipendenze dall'hardware e delle infrastrutture che gli ambienti RFID tipicamente richiedono. Attraverso Rifidi è quindi possibile virtualizzare un'intera infrastruttura, definendo via software sia i readers, sia i tags rfid, sia eventi rfid, i quali si comportano tutti esattamente come le controparti reali. Tra le caratteristiche principali dell'emulatore troviamo:

- Supporto per LLRP Virtual Reader:
  - permette la creazione di LLRP Reader
  - supporta i moduli LLRP: ROSpecs, AISpecs e ROReports
  - genera Tag Events con configurabilità limitata
- Il motore di emulazione è basato su Web Services
  - XML-RPC reader engine
  - amministrazione ed eventi basati su XML-RPC
  - capacità si esecuzione locale o remota
- è compatibile con una serie di Edge Server:
  - BEA Edge Server
  - University of Arkansas – TagCentric
  - LogicAlloy ALE Server
- emulazione di readers commerciali:
  - Alien ALR 9800 Gen 2
  - AWID MPR 2010 Gen 2

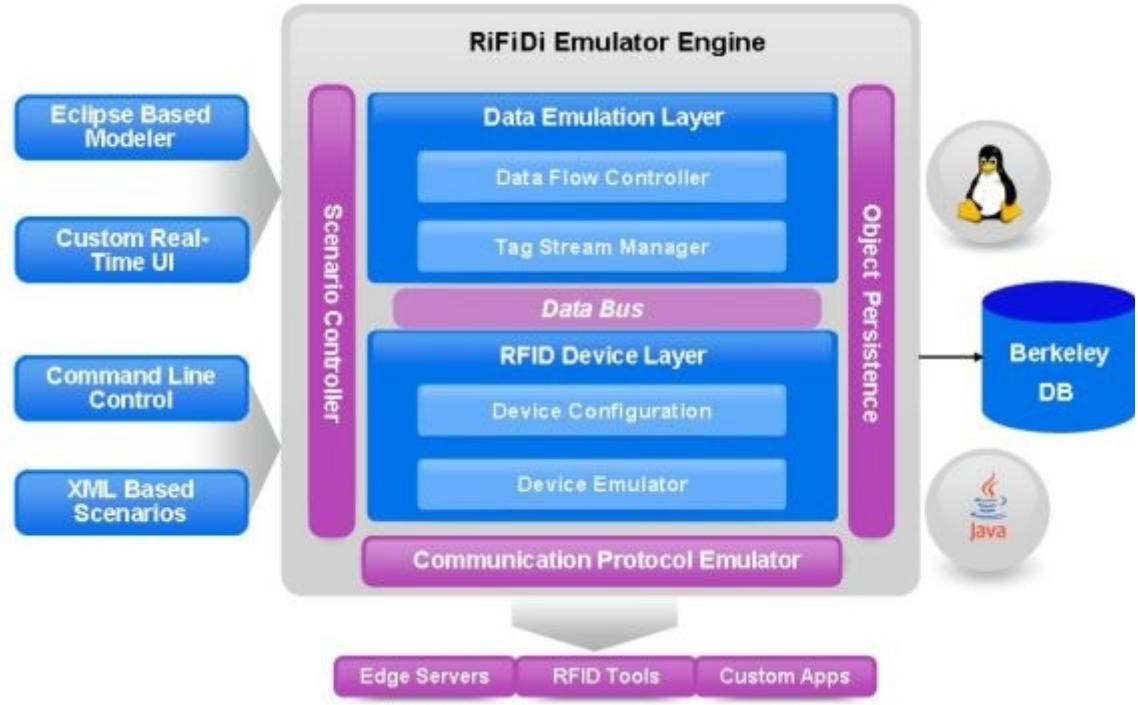


Figura 4.1.1 – Architettura dell'emulatore

4.1.1 Interfaccia grafica ed utilizzo

L'utilizzo dell'emulatore risulta essere abbastanza semplice ed intuitivo grazie all'interfaccia grafica che permette di accedere a tutte le sue funzionalità. E' possibile creare un certo numero di reader virtuali, definendo la tipologia di ognuno di essi. A seconda del tipo di reader creato, è necessario specificare alcuni parametri come ad esempio l'indirizzo ip ed il numero di porto o la porta seriale da assegnare al reader virtuale e il numero di antenne presenti. Indipendentemente dal numero di reader creati è possibile generare un numero arbitrario di tag RFID. L'emulatore consente di creare un tag alla volta, permettendo all'utente di specificare la tipologia di tag e l'id univoco che lo identifica oppure permette di generare automaticamente un numero specificato di tag. Nel caso si utilizzi la seconda modalità di creazione l'utente è tenuto ad indicare il numero di tag da creare, mentre il software in automatico genera ed assegna ai tag i relativi id. L'emulatore permette di utilizzare 4 differenti tipologie di tag:

- SGTIN-96
- GID-96
- SSCC-96
- DoD-96

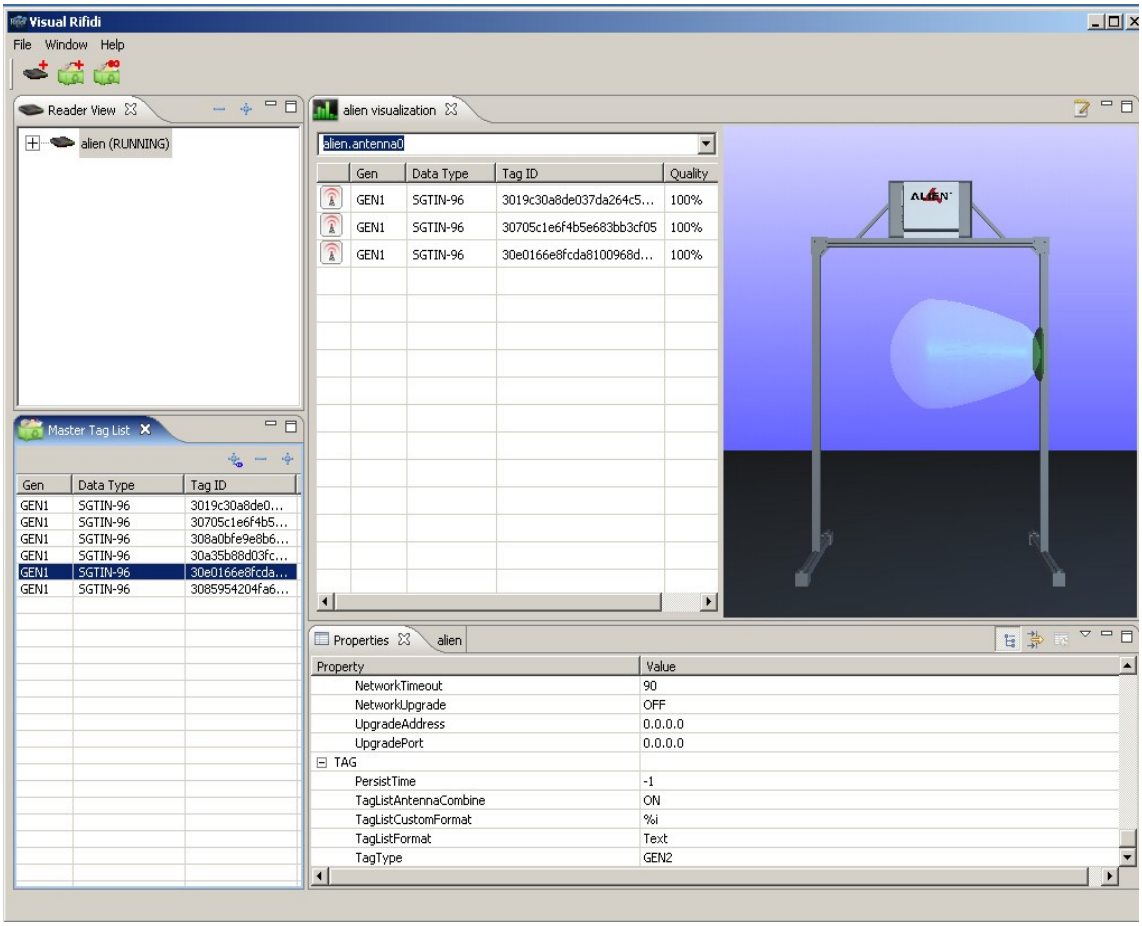


Figura 4.1.1 – GUI dell'emulatore

E' inoltre possibile, per ogni tipologia, specificare la generazione del tag, scegliendo tra le due attualmente esistenti:

- Gen 1
- Gen 2

Una volta creati readers e tags è possibile avviare le simulazioni. Ogni reader può essere controllato indipendentemente dagli altri, è infatti possibile avviare, fermare ed eliminare ogni singolo reader direttamente dall'interfaccia grafica. Una volta avviato è possibile colloquiare con un reader ed inviare comandi semplicemente stabilendo una connessione tcp con l'emulatore, o attraverso un'applicazione ad-hoc o utilizzando una classica sessione telnet, attraverso la quale si inviano comandi secondo il protocollo specifico del reader che si sta emulando. Purtroppo non è presente una funzionalità per simulare il movimento dei tag. L'ingresso o l'uscita di un reader deve essere simulata manualmente, aggiungendo o togliendo un tag da una delle finestre che rappresentano l'area di illuminazione di una delle antenne del reader. Lo sviluppo del nostro prototipo di middleware è stato portato avanti utilizzando l'emulazione di un reader commerciale, quindi realmente esistente, l'Alien ALR 9800. Dato che RIFIDI emula fedelmente tale reader nel

prossimo paragrafo ne verrà fornita una breve descrizione, limitandoci ad indicare le caratteristiche principali.

4.1.2 Il reader Alien ALR 9800

l'Alien ALR 9800 è un reader per tag RFID prodotto dalla Alien Technology. Questo reader è supportato da un vasto numero di piattaforme software RFID tra cui Microsoft BizTalk RFID, IBM WebSphere 6.0, Oat Systems, Oracle, GlobeRanger, BEA. Sono inoltre disponibili librerie Java e .NET per la realizzazione di interfacce di controllo personalizzate per il reader. Supporta la gestione del firmware ed il monitoraggio remoti attraverso il protocollo SNMP (Simple Network Management Protocol). La caratteristica più interessante di questo reader è che fornisce una modalità di funzionamento chiamata “Autonomus mode” la quale implementa una macchina a stati programmabile che abilita il reader a operare in maniera autonoma. Gli eventi che possono generare passaggi di stato sono raggruppati in tre categorie:

- eventi legati all'arrivo di segnali dall'esterno: al reader, attraverso un'interfaccia di tipo GPIO, possono essere collegate una serie di periferiche che attraverso opportuni sensori riescono a misurare variazioni nei parametri ambientali monitorati. Tali periferiche in funzione delle misurazioni effettuate possono quindi inviare uno o più segnali al reader che può, a seconda della conformazione della macchina a stati implementata, variare o meno il proprio stato ed effettuare o meno delle operazioni. Ad esempio se al reader viene collegata una fotocellula che rileva il passaggio di oggetti, esso può essere programmato in modo tale da effettuare una lettura ogni volta che la fotocellula gli invia un segnale di passaggio.

- eventi periodici: le variazioni di stato vengono scatenate da eventi che arrivano periodicamente al reader, come ad esempio un clock. Il classico esempio è quello della lettura dei tag ad intervalli regolari di tempo.

- eventi generati da software: questa categoria può comprendere un vastissimo numero di eventi. In generale ad essa appartengono tutti quegli eventi che possono essere generati dal software (ad esempio il middleware) con il quale il reader interagisce.

Utilizzando la modalità Autonomus Mode, i readers sono attivati solo quando necessario, riducendo la probabilità che più reader operino in simultanea sullo stesso ambiente e abbassando di conseguenza il risultante livello di radiazioni elettromagnetiche.

Figura 4.1.3 – Alien ALR-9800 con antenne

Ciò porta ad una diminuzione significativa del rischio di interferenza tra readers. Inoltre, quando opera in Autonomus Mode, nel caso in cui dovesse venire a mancare la connessione tra il reader e l'infrastruttura a cui vengono inviati i dati, l'ALR-9800 è in grado di collezionare fino a 2500 records relativi a tag letti. Poi, al momento del recovery della connessione, il middleware potrà scaricare i dati accumulati dal reader. Anche in caso di interruzioni di energia, dati critici relativi ai tag letti dal reader prima dell'interruzione non vengono persi. L'ALR-9800 infatti memorizza la lista dei tag in memoria non volatile, preservando i dati anche in caso di mancanza di alimentazione. L'ALR-9800 è compatibile con la specifica EPC Gen 2 Dense Interrogation, la quale riduce l'impatto delle interferenze tra reader vicini. La modalità riduce significativamente il rumore fuori canale introdotto da ogni reader, consentendo in questo modo ad un gran numero di questi di coesistere senza ridurre le frequenze di lettura. Inoltre l'ALR-9800 è dotato di un'architettura per il filtraggio del segnale molto potente ed autoadattativa, che permette di ripulire efficacemente il segnale anche in presenza di numerosi altri readers. Altre caratteristiche:

- Ottimizzato per elevate frequenze di lettura con 2 o 4 antenne
- interoperabilità EPC Gen 2
- Gestibile ed aggiornabile
- modalità di notifica e opzioni di routing configurabili
- formato dei dati configurabile (messaggi semplici o in formato XML)

- Un'API flessibile con largo supporto software
- Protezione dei dati

Di seguito viene riportata una tabella riassuntiva delle caratteristiche del reader.

Architecture	Intel XScale processor, Linux, 64 MBytes RAM, 32 MBytes Flash
Supported RFID Tag Protocols	EPC Gen 2; ISO 18000-6c
Reader Protocols	Alien Reader Protocol, SNMP, firmware upgradeable
LAN Protocols	DHCP, TCP/IP, NTP
Dense reader management	Dense Reader Mode, Event triggering
Frequency	902.75 MHz – 927.25 MHz
Channels	50
Channel Spacing	500 KHz
Communications	RS-232 (DB-9 F), LAN TCPI/IP (RJ-45)
Antennas	4 ports; multistatic topology; circular or linear polarization, reverse polarity TNC; requires minimum of 2 antennas or external circulator
General Purpose Inputs/ Outputs	4 inputs, 8 outputs, optically isolated
Software SDK	Java and .NET APIs
Tabella 4.1.1 – Caratteristiche dell'Alien ALR-9800	

4.2 Il framework APPIA

4.2.1 Introduzione - Che cos'è Appia?

Appia è un framework di comunicazione implementato in Java che fornisce possibilità di programmazione e configurazione molto estese. Contiene numerosi protocolli come ad esempio alcuni per le comunicazioni di gruppo e alcuni per l'ordinamento (dei messaggi). Per poter offrire ai propri utilizzatori servizi sempre più ricchi e potenti, le applicazioni distribuite stanno diventando sempre più complesse. Tale aumento di complessità e di funzionalità però va a discapito delle prestazioni. Per poter quindi garantire performance soddisfacenti c'è bisogno di un valido supporto alle comunicazioni. E' facile trovarsi in situazioni che richiedono l'uso simultaneo di numerosi canali, come ad esempio ambienti virtuali, simulazioni distribuite e lavoro collaborativo supportato da computer (CSCW). Una particolarità di queste applicazioni è il bisogno di scambiare e disseminare molti tipi di dati, ognuno con differenti requisiti di qualità del servizio. Ad esempio per i messaggi di testo o i trasferimenti di file ci si aspetta che le informazioni siano consegnate in maniera affidabile rispettando anche l'ordine con cui queste sono state inviate (ordine FIFO), mentre in altri scenari come lo streaming video alcuni pacchetti possono essere persi, in quanto la continuità nella riproduzione ha un'importanza maggiore rispetto alla correttezza puntuale dei dati. Come risultato queste applicazioni si affidano volentieri a canali di comunicazione multipli per lo scambio informazioni e dati. Nonostante sia facile trovare substrati di comunicazione che supportino l'uso di canali indipendenti, nelle applicazioni multicanale sussistono spesso vincoli inter-canale che devono essere preservati per semplificare la logica dell'applicazione. Ad esempio si potrebbe aver bisogno di forzare vincoli di FIFO, causal o total order, frammentare dati in differenti canali utilizzando la stessa chiave di sessione, o assicurare che a tutti i canali venga fornito un servizio di failure detection consistente. Se l'insieme dei protocolli implementati non fornisce alcun supporto per la coordinazione tra i canali, questo compito è lasciato al progettista dell'applicazione. Un approccio promettente per affrontare la complessità di questi sistemi è fare affidamento su architetture di comunicazione configurabili che siano in grado di supportare la composizione ed il riutilizzo di componenti. Framework recenti, come Ensemble e Coyote offrono un ambiente dove è possibile combinare micro protocolli “off-the-shelf” per ottenere differenti QoSs. Pochi sistemi però forniscono supporto per la coordinazione di canali multipli. Maestro e CCTL supportano la coordinazione solo per un limitato insieme di proprietà come la membership e l'ordinamento. Vincoli inter-canale che non erano stati previsti devono ancora essere implementati a livello applicazione. Appia è un nucleo di protocolli che

offre alle applicazioni una modalità pulita ed elegante per esprimere vincoli inter-canale. Ciò è ottenuto attraverso l'estensione della funzionalità già offerte dai sistemi correnti. Ovvero Appia mantiene un design flessibile e modulare, che in passato si è dimostrato essere molto vantaggioso, permettendo di comporre e riconfigurare gli stack di comunicazione run-time.

Perché il nome "Appia"?

La prima delle grandi strade romane, la via Appia, iniziata dal censore Appius Claudius Caecus nel 312 AC, originariamente correva per 216 km dal sudest di Roma fino a Tarentum (la moderna Taranto) e fu successivamente estesa alla costa Adriatica fino a Brundisium (la moderna Brindisi). Le strade furono fondamentali per lo sviluppo dell'impero Romano che governò gran parte dell'Europa per numerosi secoli. Fu costruita, per quel tempo, una enorme rete stradale che collegava tutto l'impero romano, dando vita al proverbio “Tutte le strade portano a Roma”. All'epoca le strade erano l'unico canale di comunicazione disponibile. Esse erano utilizzate per trasportare beni e notizie e furono fondamentali per l'espansione dell'impero. Le strade romane erano famose per la loro linearità, per le loro fondamenta solide, per la superficie curva che ne facilitava il drenaggio, e per l'uso di calcestruzzo ricavato dalla pozzolana e calce. All'estero, pur adattando le loro tecniche di costruzione ai materiali disponibili localmente, gli ingegneri romani seguirono gli stessi principi di base già adottati per la realizzazione della rete in Italia. Il sistema di strade Romano rese possibile la conquista e l'amministrazione dei territori e successivamente fornì le rotte principali per le grandi migrazioni interne all'impero. Fu anche un mezzo fondamentale per la diffusione della cristianità.

4.2.2 Appia, modello e funzionamento

Le comunicazioni tra processi sulle reti, al fine di fornire servizi sempre più complessi, richiedono che numerose proprietà ben distinte siano combinate in vario modo. Alcuni standard di rete sono stati sviluppati seguendo tale principio e sono ora largamente utilizzati. Questo è il caso di protocolli internet come IP, TCP, UDP [27,28,29] e il modello OSI [29]. Molti di loro assumono una struttura stratificata, ovvero i protocolli vengono disposti uno sopra l'altro a formare una pila. Ogni strato, per fornire il proprio servizio agli strati sovrastanti, fa affidamento sulle proprietà documentate dei protocolli sottostanti. Il Transmission Control Protocol (TCP), per esempio, fa affidamento sulle capacità di routing di IP per assicurare che i pacchetti spediti vengano consegnati alla destinazione corretta. Dato che IP non assicura l'ordine FIFO, è TCP che si fa carico di fornire tale proprietà. Ogni combinazione di layer (protocolli) nello stack fornisce un differente insieme di proprietà (un ordinamento differente dei protocolli può anche fornire differenti insiemi di proprietà) che può

essere considerato come un unico servizio fornito dallo stack stesso. Di seguito le proprietà risultanti da ogni combinazione e i protocolli presenti nel relativo stack verranno utilizzati in maniera intercambiabile e riferiti come Quality of Service (QoS). APPIA è un framework per il supporto alla comunicazione stratificata. Il suo scopo è definire un'interfaccia standard che deve essere rispettata da tutti i layer e facilitare la comunicazione tra di essi. Appia è indipendente dal protocollo. Ovvero, il framework accetta ogni protocollo fino a che esso rispetta l'interfaccia predefinita, non facendo assunzioni o operazioni finalizzate a validare il risultato finale della composizione (Appia di fatto offre una limitata forma di validazione dello stack).

4.2.2.1 Concetti APPIA

Questo paragrafo descrive brevemente i concetti e la terminologia utilizzati in Appia. Concetti statici e dinamici: Appia presenta una chiara distinzione tra la dichiarazione di qualcosa (sia un protocollo o uno stack) e la sua implementazione. Un Layer è definito attraverso un insieme di proprietà richieste ed un insieme di proprietà fornite. Una Sessione è un'istanza in esecuzione di un protocollo. Le sessioni sono sempre create in funzione di un layer e il loro stato è indipendente da quello di altre istanze. Un QoS è una descrizione statica di un insieme ordinato di protocolli. Un Channel è un'istanziazione dinamica di un QoS. Le istanze di protocollo (chiamate sessioni) comunicano utilizzando un'infrastruttura Channel. Tutti questi concetti sono illustrati nella figura 4.2.1 e nella tabella 4.2.1

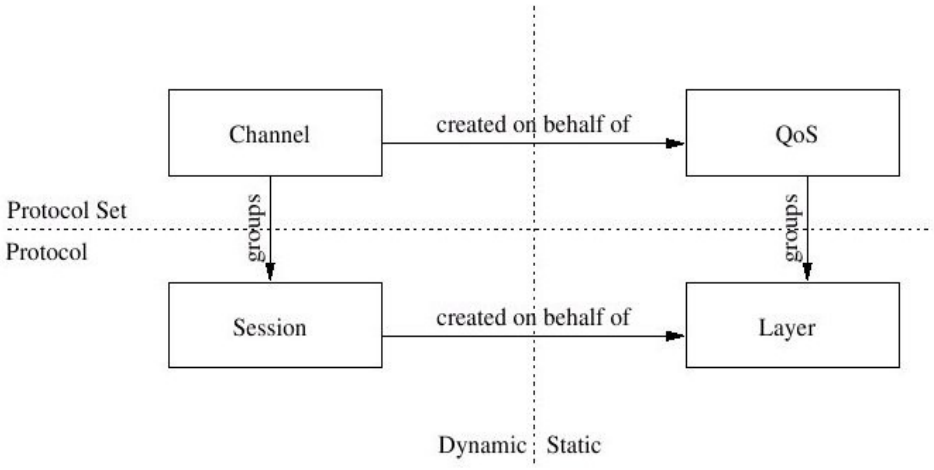


Figura 4.2.1 – Relazione tra concetti Appia

Essendo entità statiche, i layers non scambiano informazioni tra di loro. Essi dichiarano l'interfaccia di comunicazione delle loro istanziazioni dinamiche: le sessioni. Le comunicazioni tra le sessioni e

tra sessioni e channel sono realizzate utilizzando oggetti chiamati Events o Eventi. Appia ne fornisce un insieme predefinito, ed ogni elemento di quest'ultimo assume un differente significato. I programmatori sono incoraggiati a estendere questo insieme, in modo da dettagliare eventi più specifici da utilizzare con i propri protocolli. A partire dalla sessione che lo ha generato, l'evento fluisce lungo lo stack verso una direzione predefinita. Le informazioni contenute in ogni evento specifico estendono l'insieme base dei campi che tutti gli eventi devono contenere.

Concetto	Tipo	#	Descrizione
Layer	Static	1	Descrizione statica di un protocollo. Dichiara le proprietà che esso richiede e fornisce
Session	Dynamic	n	Istanza di esecuzione di un protocollo. Keeps lo stato di un protocollo ed implementa le proprietà descritte nel corrispondente layer
QoS	Static	1	Insieme ordinato di layer. Descrive le proprietà che un'istanza in esecuzione di tale combinazione di protocolli dovrebbe avere.
Channel	Dynamic	n	Insieme ordinato di sessioni, modellate da un QoS. E' in sostanza l'entità che fornisce l'insieme di proprietà specificate nel QoS.

Tabella 4.2.1 – Descrizione concetti Appia

Riusabilità: La riusabilità in appia è basata sull'ereditarietà. Dato che molti dei protocolli dipendono (al più lascamente) dai servizi forniti dagli altri, aggiornarne alcuni potrebbe produrre incompatibilità. Appia utilizza l'ereditarietà per rendere tali aggiornamenti trasparenti. Quando una nuova versione del protocollo è rilasciata, ci si aspetta che gli eventi generati contengano un insieme di informazioni più ricco o al più uguale rispetto alla versione precedente. Assumendo che il formato di nessuna delle informazioni fornite con la vecchia versione sia cambiato, i protocolli possono semplicemente creare nuovi eventi estendendo i precedenti. In questo modo, la retro-compatibilità è assicurata.

Ottimizzazione: L'ereditarietà è anche utilizzata per incrementare le performance. Gli eventi Timer, per esempio, sono generati dai protocolli (come richieste) e gestiti dal Channel. Ogni sessione è libera di estendere gli eventi timer standard, aggiungendo informazioni che altrimenti dovrebbero essere conservate nello stato della sessione. Un layer di consegna affidabile ad esempio può includere il messaggio che deve essere ritrasmesso nell'evento di richiesta del timeout. Quando scade il timeout, la sessione semplicemente estrae il messaggio dell'evento e lo rispedisce. Il tempo di processamento degli eventi inoltre è ridotto evitando che le istanze di protocollo gestiscano eventi non voluti. Ogni protocollo manifesta esplicitamente il proprio interesse nel ricevere un determinato insieme di eventi indicandoli all'interno del layer, con granularità al singolo evento.

Istanze di classi di eventi non dichiarate dal layer non sono consegnate alla corrispondente sessione.

4.2.2.2 Definizione di un protocollo

Ogni protocollo è definito da due classi differenti: una che estende la classe di base Layer, e l'altra che estende la classe Session. Per convenzione, la prima è di solito chiamata ProtocolLayer mentre la seconda ProtocolSession indicando con Protocol il nome del protocollo. La classe ProtocolLayer è quella che partecipa alla definizione del QoS. Il suo scopo è esportare l'insieme di eventi e creare istanze della classe ProtocolSession. La classe ProtocolSession è invece quella che partecipa ai Channels e che una volta istanziata coincide con l'istanza di protocollo. Essa ha due scopi fondamentali: cooperare nella definizione del Channel e gestire/generare eventi, garantendo le proprietà attese del protocollo.

4.2.2.3 Relazione tra sessioni e canali

In Appia una sessione (ovvero un'istanza in esecuzione di un protocollo) può partecipare a numerosi Channel contemporaneamente anche se essi hanno differenti QoS's. Questo significa che una singola istanza può partecipare a combinazioni protocollari multiple. Questo è uno degli aspetti innovativi di Appia e offre nuove prospettive riguardo le modalità con cui è possibile relazionare tipologie di dati differenti. Per esempio, avendo solo una singola sessione FIFO su due canali, uno con un appropriato QoS per le trasmissioni video e l'altro per l'audio, il ricevente è in grado di imporre l'ordine di spedizione dei messaggi tra i due differenti media senza alcuno sforzo di programmazione addizionale. La partecipazione o meno di una sessione a canali multipli dipende dall'implementazione e dal protocollo. Alla ricezione di un evento, le sessioni sono libere dal vincolo di interrogare il canale. Gli eventi possono essere propagati senza che la sessione conosca il canale che si sta utilizzando.

4.2.2.3 Classi di implementazione

Ci sono undici classi rilevanti per l'implementazione di protocolli in appia: QoS, Channel, ChannelCursor, Layer, Session, Event, Message, MsgWalk, MsgBuffer, Direction e Appia. Sono presenti altre classi ma queste non forniscono caratteristiche rilevanti per lo sviluppo dei protocolli.

4.2.2.4 Modello

Il modello di Appia evidenzia la distinzione tra le proprietà dello stack, catturate dal concetto di QoS, e ogni specifica istanza di un dato Qos, individuata dal concetto di canale. I dati fluiscono attraverso canali specifici. In molti sistemi, il processamento di messaggi richiede che ogni layer effettui operazioni di demultiplexing locale per determinare l'appropriato contesto di sessione. In Appia, come in Ensemble, il demultiplexing è effettuato una sola volta, quando i messaggi entrano nel sistema ed il canale di destinazione viene individuato. Mentre però Ensemble utilizza funzioni di kernel per recuperare i messaggi dalla rete e trovare il canale appropriato, Appia rende il modello più flessibile delegando ai protocolli entrambe queste operazioni. Alla creazione, un canale è un array di slot vuoti ad ognuno dei quali viene associato un tipo di layer. Ognuno di questi slot deve essere poi riempito con una sessione del layer specificato nel QoS per la posizione occupata dallo slot stesso. Le sessioni possono essere associate agli slots esplicitamente o implicitamente da altre sessioni (binding automatico). Di default, agli slot non soggetti a binding esplicito o automatico, verranno associate sessioni nuove, create appositamente. Utilizzando il binding esplicito è possibile associare specifiche istanze di Layer a specifici canali. Queste sessioni possono essere sia già in uso o possono essere create intenzionalmente per il nuovo canale. Questo tipo di binding permette all'utente di avere un controllo fine sulla configurazione. Utilizzando quello automatico è possibile delegare a sessioni già istanziate e linkate al relativo slot il compito di specificare ed instanziare quelle rimanenti. Tipicamente viene utilizzato un mix di bindings espliciti ed impliciti. In Appia, la coordinazione inter-canale può essere realizzata portando canali differenti a condividere una o più sessioni.

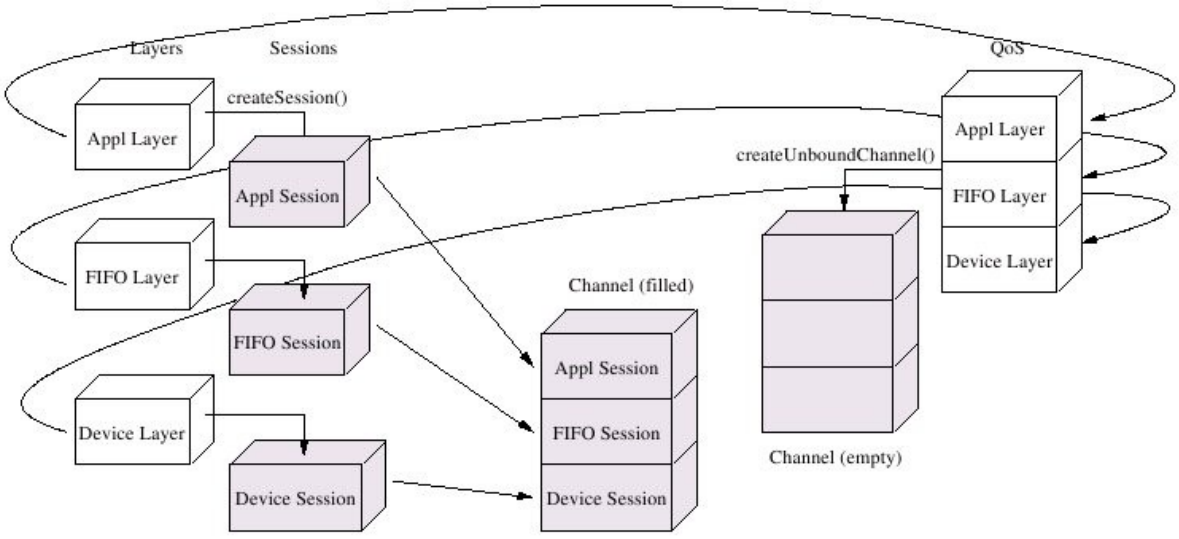


Figura 4.2.2 – Istanziazione di un Appia Channel

4.2.2.5 Capacità di configurazione

Un protocollo è definito come channel-aware se il suo algoritmo è in grado di distinguere eventi che fluiscono da canali diversi ed è in grado di agire differentemente in funzione del canale di provenienza. Ed esempio, un protocollo di FAILURE DETECTION dovrebbe essere del tipo channel-unaware, mentre un protocollo di tipo INTERMEDIA SYNC è, per definizione, channel-aware (esso cioè seleziona il QoS più opportuno per la spedizione del messaggio). Se la channel-awareness è obbligatoria allora la riusabilità del protocollo in Appia dovrebbe essere limitata. Inoltre, è possibile creare protocolli che sono indifferenti al numero di canali che attraversano la loro sessione. Un identificatore di canale CID viene presentato alla sessione ad ogni consegna di evento. Questo valore può essere considerato opaco dalla sessione. Se un nuovo evento è generato in risposta a uno appena arrivato, la sessione dovrebbe propagare il valore del CID associato all'evento originale. Molte delle sessioni che possono essere trovate negli stack già definiti in Appia sono channel-unaware. La Channel-awareness consente grande flessibilità di configurazione. Le sessioni possono utilizzare le informazioni di canale per capire quali sono i QoSs disponibili a poi scegliere quello più adatto a trasportare i propri eventi.

4.2.2.6 Eventi

Alcuni framework supportano solo un insieme predefinito di eventi. La conoscenza della semantica di ogni evento è così utilizzata per implementare ottimizzazioni event-specific. Questi frameworks supportano quindi un modello chiuso di eventi. I modelli chiusi sono molto difficili da applicare in differenti contesti dato che essi supportano solo un insieme precostituito e non mutabile di interazioni: l'insieme predefinito di eventi potrebbe non essere sufficiente ad esprimere, in maniera efficiente, le interazioni richieste in altri protocolli. Un framework che utilizza un modello aperto, permette invece la definizione di nuovi eventi. Naturalmente, è difficile implementare ottimizzazioni nel routing quando l'insieme non è noto a priori. Il modello qui presentato prova a fondere i vantaggi di un modello chiuso con la flessibilità di uno aperto. Gli eventi in Appia sono strutture dati object-oriented. Se ne possono creare di nuovi derivando da una classe evento definita in precedenza (in particolare direttamente dalla classe evento principale). Per permettere future ridefinizioni, i test sui tipi di evento sono sempre effettuati sulla classe che soddisfa più largamente i requisiti necessari. Lo scopo è quello di forzare la specializzazione degli eventi utilizzando l'ereditarietà. In questo modo, vecchi protocolli, non a conoscenza degli attributi di nuovi eventi, continueranno ad essere eseguiti correttamente. Come in Ensemble [30], le sessioni interagiscono con l'ambiente solo attraverso eventi. Concettualmente, il canale è posizionato al di sotto della sessione più bassa dello stack e al di sopra della più alta, ovvero

è come se lo stack fosse delimitato superiormente ed inferiormente dal canale.

4.2.2.7 Controllo di compatibilità run-time

A tempo di definizione del QoS, ai layers è richiesto di dichiarare tre insiemi di eventi: Ψ_l contenente gli eventi richiesti dal layer l per fornire una corretta esecuzione; Φ_l contenente gli eventi che l si appresterà a ricevere e Γ_l contenente gli eventi che l genererà. Uno stack è definito corretto se ogni elemento presente in Ψ è contenuto anche in Γ (Ψ e Γ sono rispettivamente l'unione di tutti gli insiemi Ψ_l e Γ_l nello stack). Le definizioni di QoS che non rispettano i vincoli precedenti genereranno un'eccezione e non saranno create. Naturalmente, ci si aspetta che per ogni insieme Φ_l valga $\Psi_l \subseteq \Phi_l$. Si capisce facilmente come questo non sia uno strumento completo per la validazione dello stack. Utilizzando le caratteristiche funzionali del linguaggio ML per validare la corretta esecuzione dello stack, Ensemble prende un approccio più forte, più robusto al problema. Appia semplicemente effettua un controllo run-time della composizione, assumendo: 1) che i protocolli siano stati codificati correttamente e 2) la semantica degli eventi sia ben conosciuta.

4.2.2.8 Routing efficiente degli eventi

Durante una normale esecuzione, soltanto pochi protocolli aggiungono informazioni significative ai messaggi [31, 32, 30]. Per esempio, in uno stack per comunicazioni di gruppo non tutti i protocolli sono interessati a ricevere informazioni sul cambio delle view. Di conseguenza un determinato evento potrebbe non essere di interesse per uno o più layer all'interno di uno stack. Per evitare di eseguire lavoro inutile e ridurre di conseguenza le prestazioni, sarebbe preferibile quindi che ogni layer ricevesse solo gli eventi di interesse. L'approccio utilizzato in Appia permette ai layer di dichiarare esplicitamente gli eventi ai quali i loro protocolli sono interessati. Gli insiemi di eventi, specificati a tempo di definizione del QoS, sono utilizzati per ottimizzare l'esecuzione run-time nel seguente modo: per ogni evento $e \in \Gamma$, sarà costruita una lista, contenente tutti i layer che hanno indicato *e* nel loro insieme Φ . Alla creazione del canale, ad ogni tipologia di evento *e* verrà associata la relativa tabella di routing. Per ogni canale il routing è statico. All'arrivo di un evento, ne viene determinata la tipologia. Con quest'ultima si va a reperire la relativa tabella di routing definita presso il canale destinatario la quale viene infine associata all'evento stesso. Quest'ultimo è quindi in grado di trovare la prossima sessione che deve essere visitata semplicemente mantenendo un puntatore ad un array. In Appia, gran parte dell'overhead per il routing viene eseguito

durante la fase di definizione del QoS. Quest'ultima però interessa solo il lasso di tempo relativo allo start-up dell'applicazione. Utilizzando questo approccio quindi la flessibilità introdotta non compromette le performance run-time.

4.3 PostgreSQL

4.3.1 Cos'è PostgreSQL

PostgreSQL è un Database Management Systems relazionale orientato agli oggetti (ORDBMS) basato su POSTGRES, versione 4.2, sviluppato all'Università della California presso il Berkeley Computer Science Department. POSTGRES ha anticipato molti concetti che sono divenuti disponibili sui DBMS commerciali solo molto tempo dopo. PostgreSQL è un discendente open source del codice originale sviluppato al Berkeley. Esso supporta SQL92 e SQL99 ed offre molte caratteristiche moderne:

- query complesse
- foreign keys
- triggers
- views
- integrità transazionale
- controllo della concorrenza multiversione (MVCC)

In più, PostgreSQL può essere esteso dall'utente in molti modi, per esempio aggiungendo nuovi:

- tipi di dato
- funzioni
- operatori
- funzioni aggregate
- metodi di indicizzazione
- linguaggi procedurali

La licenza con la quale è distribuito postgreSQl è di tipo libero, in stile BSD.

4.3.2 Un po' di storia

Inizialmente il dbms era chiamato Ingres e come abbiamo già detto era un progetto del Berkeley. Nel 1982 il capo progetto, Michael

Stonebraker, ha lasciato il Berkeley per commercializzare il prodotto, ma in seguito è tornato all'accademia. Nel 1985 l'ha lasciata nuovamente per dare vita a un progetto post-Ingres (**Postgres**) che superasse gli evidenti limiti dei prodotti concorrenti dell'epoca. Le basi dei sorgenti di Ingres e di Postgres erano, e sono rimasti nel tempo, ben distinti. Il nuovo progetto puntava a fornire un supporto completo ai tipi di dati, in particolare la possibilità di definire nuovi tipi di dati (UDF, User Defined Types). Vi era anche la possibilità di descrivere la relazione tra le entità (tabelle), che fino ad allora veniva lasciata completamente all'utente. Perciò non solo Postgres preservava l'integrità dei dati, ma era in grado di leggere informazioni da tabelle relazionate in modo naturale, seguendo le regole definite dall'utente. Dal 1986 gli sviluppatori diffusero un gran numero di articoli che descrivevano il nuovo sistema e nel 1988 venne rilasciato un primo prototipo funzionante. La versione 1 venne rilasciata nel giugno del 1989 per un numero di utenti contenuto. Seguì una versione 2 nel giugno del 1990, in cui il sistema delle regole venne completamente riscritto. Nella versione 3, del 1991, questo sistema venne nuovamente riscritto, ma venne aggiunto anche il supporto a gestori multipli di immagazzinamento dei dati e un motore di query migliorato. Nel 1992 vi era già un numero di utenti notevole che inondava il team di sviluppo con richieste di supporto e di nuove features. Dopo aver rilasciato la versione 4, che fu principalmente una pulizia del codice, il progetto terminò. Sebbene il progetto Postgres fosse ufficialmente abbandonato, la licenza BSD dava modo agli sviluppatori OpenSource di ottenere una copia del software per poi migliorarlo a loro discrezione. Nel 1994 due studenti del Berkeley, Andrew Yu e Jolly Chen aggiunsero a Postgres un interprete SQL per rimpiazzare il vecchio QUEL che risaliva ai tempi di Ingres. Il nuovo software venne quindi rilasciato sul web col nome di **Postgres95**. Nel 1996 cambiò nome di nuovo: per evidenziare il supporto al linguaggio SQL, venne chiamato **PostgreSQL**. Il primo rilascio di PostgreSQL fu la versione 6. Da allora ad occuparsi del progetto è una comunità di sviluppatori volontari provenienti da tutto il mondo che si coordina attraverso Internet. Alla versione 6 ne sono seguite altre, ognuna delle quali ha portato nuovi miglioramenti. Nel gennaio 2005 è stata rilasciata la versione 8.

4.3.3 Una breve descrizione

Un rapido esame di PostgreSQL potrebbe suggerire che sia simile agli altri database. PostgreSQL usa il linguaggio SQL per eseguire delle query sui dati. Questi sono conservati come una serie di tabelle con foreign keys (n.d.t. chiavi esterne) che servono a collegare i dati correlati. La programmabilità di PostgreSQL è il suo principale punto di forza ed il principale vantaggio verso i suoi concorrenti: PostgreSQL rende più semplice costruire applicazioni per il mondo reale, utilizzando i dati prelevati dal database. I database SQL conservano dati semplici in “Flat tables” (tabelle piatte n.d.t.),

richiedendo che sia l'utente a prelevare e raggruppare le informazioni correlate utilizzando le query. Questo contrasta con il modo in cui sia le applicazione che gli utenti utilizzano i dati: come ad esempio in un linguaggio di alto livello con tipi di dato complessi dove tutti i dati correlati operano come elementi completi, normalmente definiti oggetti o record (in base al linguaggio). Convertire le informazioni dal mondo SQL a quello della programmazione orientata agli oggetti presenta difficoltà dovute principalmente al fatto che i due mondi utilizzano modelli di organizzazione dei dati molto differenti. L'industria chiama questo problema *impedance mismatch* (discrepanza di impedenza): mappare i dati da un modello all'altro può assorbire fino al 40% del tempo di sviluppo di un progetto. Un certo numero di soluzioni di mappatura, normalmente dette “object-relational mapping”, possono risolvere il problema, ma tendono ad essere costose e ad avere i loro problemi, causando scarse prestazioni o forzando tutti gli accessi ai dati ad aver luogo attraverso il solo linguaggio che supporta la mappatura stessa. PostgreSQL può risolvere molti di questi problemi direttamente nel database. PostgreSQL permette agli utenti di definire nuovi tipi basati sui normali tipi di dato SQL, permettendo al database stesso di comprendere dati complessi. Per esempio, si può definire un indirizzo come un insieme di diverse stringhe di testo per rappresentare il numero civico, la città, ecc. Da qui in poi si possono creare facilmente tabelle che contengono tutti i campi necessari a memorizzare un indirizzo con una sola linea di codice. PostgreSQL, inoltre, permette l'ereditarietà dei tipi, uno dei principali concetti della programmazione orientata agli oggetti. Ad esempio, si può definire un tipo codice_postale, quindi creare un tipo C.A.P. (codice di avviamento postale) o un tipo us_zip_code basato su di esso. Gli indirizzi nel database potrebbero quindi accettare entrambi i tipi, e regole specifiche potrebbero validare i dati in entrambi i casi. Nelle prime versioni di PostgreSQL, implementare nuovi tipi richiedeva scrivere estensioni in C e compilarle nel server di database. Dalla versione 7.4 è diventato molto più semplice creare ed usare tipi personalizzati attraverso il comando "CREATE DOMAIN". La programmazione del database stesso può ottenere grandi vantaggi dall'uso delle funzioni. La maggior parte dei sistemi SQL permette agli utenti di scrivere una procedura, un blocco di codice SQL che le altre istruzioni SQL possono richiamare. Comunque SQL stesso rimane inadatto come linguaggio di programmazione, pertanto gli utenti possono sperimentare grandi difficoltà nel costruire logiche complesse. Ancora peggio, SQL non supporta molti dei principali operatori di base dei linguaggi di programmazione, come le strutture di controllo di ciclo e condizionale. Pertanto ogni venditore ha scritto le sue estensioni al linguaggio SQL per aggiungere queste caratteristiche, e pertanto queste estensioni non per forza operano su diversi DBMS.

In PostgresSQL i programmatori possono implementare la logica in

uno dei molti linguaggi supportati:

- Un linguaggio nativo chiamato PL/pgSQL simile al linguaggio procedurale di Oracle PL/SQL, che offre particolari vantaggi nelle procedure che fanno un intensivo uso di query.

- Wrappers per i più diffusi linguaggi di scripting come Perl, Python, Tcl e Ruby che permettono di utilizzare la loro potenza nella manipolazione delle stringhe e nel link ad estese librerie di funzioni esterne.

- Le procedure che richiedono prestazioni maggiori e logiche di programmazione complesse possono utilizzare il C ed il C++.

- Inoltre è disponibile anche un interfacciamento all'esoterico linguaggio R, ricco di funzioni statistiche e per il calcolo matriciale

Il programmatore può inserire il codice sul server come *funzioni*, che lo rendono riutilizzabile come *stored procedure*, in modo che il codice SQL possa richiamare funzioni scritte in altri linguaggi (come il C o il Perl). Punti di forza della programmabilità di PostgreSQL:

- Incremento delle prestazioni, in quanto la logica viene applicata direttamente dal server, riducendo il passaggio di informazioni tra il client ed il server.

- Incremento dell'affidabilità, dovuto alla centralizzazione del codice di controllo sul server, non dovendo gestire la sincronizzazione della logica tra molteplici client e i dati memorizzati sul server.

- Inserendo livelli di astrazione dei dati direttamente sul server, il codice del client può essere più snello e semplice.

Questi vantaggi fanno di PostgreSQL, probabilmente, il più avanzato sistema database dal punto di vista della programmabilità. Utilizzare PostgreSQL può ridurre fortemente il tempo totale di programmazione di molti progetti, con i vantaggi suddetti che crescono con la complessità del progetto stesso.

4.3.4 Altre caratteristiche

Oltre a quelle già citate PostgreSQL presenta numerose altre caratteristiche interessanti, di seguito ne riportiamo un breve elenco:

- Compatibilità con i maggiori sistemi operativi: Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64), e Windows.
- supporto alla Replicazione: permette la duplicazione di database

master presso macchine slave multiple.

- è totalmente ACID compliant
- è ANSI SQL compliant.
- offre supporto SSL
- ha un supporto completo per foreign keys, joins, views, triggers e stored procedures
- supporta la memorizzazione di grandi oggetti binari, incluse immagini, file audio e video
- fornisce funzionalità di: point in time recovery, tablespaces, replicazione asincrona, transazioni annidate (savepoints), backups on-line, ottimizzazione di query sofisticate e write ahead logging per la tolleranza ai guasti.
- supporta insiemi di caratteri internazionali, la codifica di caratteri multibyte, Unicode.
- grazie al progetto PostGIS, PostgreSQL può essere arricchito con un supporto per oggetti geografici, permettendo il suo utilizzo all'interno di sistemi GIS (Geografic Information Systems).

5. Il middleware e la sua architettura

5.1 Introduzione

La finalità del nostro lavoro è stata quella dello sviluppo di un middleware per architetture rfid. Il suo compito è quello di interfacciare la rete di rfid data sources con un back-end e con eventuali applicazioni lato utente che utilizzano dati relativi a tag rfid, come ad esempio un client per monitoraggio real-time di oggetti all'interno di un ambiente, più o meno esteso. L'interfacciamento consiste sostanzialmente nel filtraggio dei dati provenienti dagli rfid data sources, con propagazione al back end solo di quelli significativi. Naturalmente la logica di filtraggio e di conseguenza le sue regole dipendono esclusivamente dal tipo di ambiente operativo in cui viene utilizzato il middleware. Di conseguenza quest'ultimo è stato realizzato in modo tale da poter operare con differenti logiche di filtraggio, semplicemente attraverso l'aggiunta di plug-in contenenti le relative regole da adottare. Sul mercato esistono già prodotti commerciali con caratteristiche molto simili a quelle appena descritte, quindi il nostro lavoro si è concentrato sulla realizzazione di un middleware di tipo *fault tolerant*, il quale attraverso opportune tecniche di replicazione è in grado di resistere a guasti, ovvero di garantire il servizio anche in presenza di malfunzionamenti di alcune sue parti. E' un aspetto questo che ancora non è stato curato, se non in maniera marginale (è ancora agli inizi), dai prodotti commerciali. In questo capitolo descriveremo dettagliatamente l'architettura del middleware, del quale è stato realizzato un prototipo funzionante, illustrando e motivando le scelte fatte in fase di progettazione. Parleremo anche del modello di programmazione e delle tecniche di replicazione supportate dall'architettura.

5.2 Descrizione generale del middleware

Il middleware è di tipo multi-tier. Più precisamente è strutturato su due tier ai quali si aggiungono tre ulteriori tier: un data source tier, un tier di back end ed un tier di presentation logic, non inclusi direttamente nell'architettura ma necessari per la realizzazione di un completo sistema operante.

I tier che compongono il middleware sono:

- Low Level Middleware tier (LLM)
- High Level Middleware tier (HLM)

Il data source tier è formato dall'insieme di tutti gli rfid data sources. Il suo compito è quello di raccogliere le informazioni contenute nei tag-rfid presenti nelle aree di illuminazione delle antenne dei data sources e convogliarli verso il Low Level Middleware tier. Quest'ultimo invece ha il compito di filtrare attraverso la logica che implementa le regole di filtraggio. In base a tali regole vengono selezionati un'insieme di dati che verranno poi inviati all'High Level Middleware tier. Quest'ultimo si occupa dell'interfacciamento con il back-end tier e con le user application, attraverso un presentation logic tier. Sostanzialmente, propaga le informazioni provenienti dall'LLM tier ad una serie di destinatari, secondo opportune regole di smistamento.

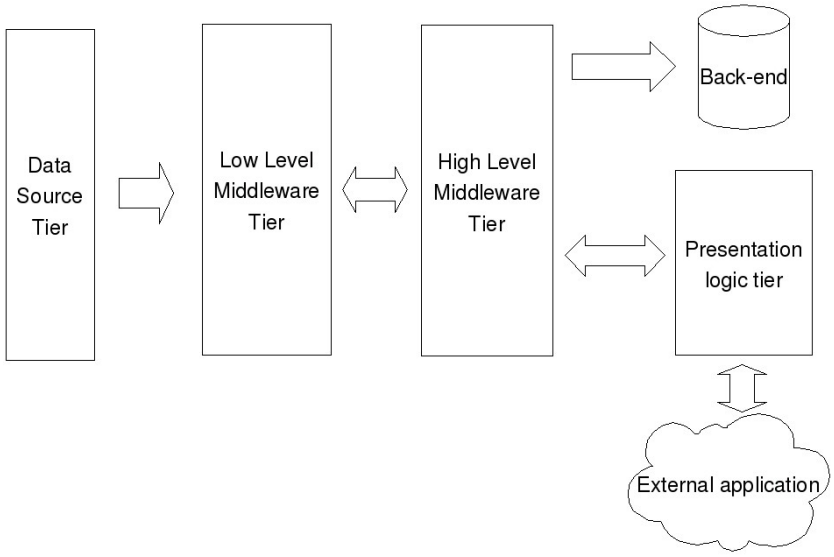


Figura 5.1 – Architetture multitier del middleware

Naturalmente un middleware per rfid può essere realizzato su un unico tier, soluzione adottata inoltre da numerosi prodotto commerciali, ma noi abbiamo adottato una strategia differente spinti da motivazioni legate alle caratteristiche dell'ambiente operativo rfid, di seguito riportiamo le principali:

- generazione di ingenti quantitativi di dati ridondanti
- varianza della distribuzione spaziale della rete di rfid data sources: da reti distribuite su area locale a reti su scala geografica. L'estensione della rete è funzione del fenomeno monitorato(es. monitoraggio libri biblioteca: rete su area locale; monitoraggio fauna selvatica: rete su area geografica)

- Sensibile disparità di frequenza nella produzione di dati tra il tier dei data source e il tier del middleware

Vediamo ora come tali caratteristiche hanno influenzato le nostre scelte. C'è da premettere innanzitutto che la suddivisione in due tier ha rispetto alla soluzione con unico tier un vantaggio fondamentale, permette di sfruttare i vantaggi legati alla prossimità spaziale. Partendo dall'inizio vediamo che per il data source tier possono valere le seguenti assunzioni:

- i data source sono distribuiti su un territorio molto vasto, magari in configurazione ad isole, ovvero organizzati in gruppi distanziati tra di loro ed ogni gruppo ha il compito di monitorare una data porzione di ambiente (un esempio tipico di tale scenario è quello del monitoraggio dei magazzini di una multinazionale, i quali singolarmente hanno un'estensione ridotta, ma nel complesso possono essere distribuiti su scala geografica)

- La frequenza di scansione delle aree da parte dei data sources è sufficientemente elevata da permettere una corretta rilevazione dello stato dell'ambiente, producendo però di conseguenza una elevata quantità di dati.

In una tale situazione con un middleware centralizzato i dati dovrebbero attraversare grosse porzioni di reti geografiche. Di conseguenza è probabile che i dati sperimentino grandi latenze di rete e che in caso di congestione la probabilità di perdita risulti elevata. Inoltre data l'ingente quantità di dati trasmessi da ogni gruppo di data sources, la quantità di banda necessaria per la loro trasmissione potrebbe essere molto elevata. Questa potrebbe non essere disponibile o nel caso in cui lo fosse potrebbe avere costi elevati. Suddividendo il middleware in due tier, e distribuendo il LLM su più istanze, è possibile posizionare un'istanza di LLM nell'immediata prossimità di un gruppo di data sources. Ciò potrebbe permettere di utilizzare una rete locale dedicata a basso costo per interconnettere il gruppo di data sources all'istanza locale di LLM, con tutti i vantaggi derivanti dal caso. A questo punto, dato che il LLM effettua filtraggio, si può sfruttare il vantaggio di una frequenza di generazione di dati più bassa da parte dell'LLM tier: su rete geografica faccio viaggiare i dati in uscita dalle istanze di LLM. La quantità di dati è nettamente inferiore, è richiesta meno banda, aumenta la probabilità che questa sia disponibile, diminuiscono i costi (monetari), diminuisce la probabilità di congestione e quindi di perdita di dati. Un altro vantaggio di un'architettura basata su due tier è quello relativo alla minore richiesta di risorse di calcolo per la dislocazione di ogni singolo componente. Ciò permette anche di coprire un maggior numero di scenari operativi, si pensi ad esempio a quelli in cui le istanze di LLM sono mobili e “seguono” lo spostamento del gruppo di

tag rfid monitorati. In tal caso le risorse di calcolo sono del tipo a basso consumo energetico e quindi con capacità limitate. Infine vediamo che un'architettura basata su due tier fornisce un numero maggiore di gradi di libertà in fase progettuale e realizzativa, risulta quindi esser meglio modellabile ed adattabile alle esigenze operative. Ad esempio è possibile utilizzare differenti modalità di replicazione per i due tier e ciò può consentire di ottenere un miglior trade-off tra affidabilità e prestazioni dell'intero sistema. Dal canto suo un middleware centralizzato risulta di più facile progettazione ed implementazione rispetto ad uno multi-tier e in alcuni casi può dimostrarsi una soluzione migliore rispetto a quest'ultimo. Si pensi ad esempio ad uno scenario in cui il monitoraggio e l'utilizzo delle relative informazioni si svolgono per intero in un ambiente ristretto, come ad esempio un singolo edificio. In una tale situazione, dato che le ipotesi che spingono alla progettazione multi-tier(es. necessità di attraversare porzioni di rete geografica) vengono a mancare, la soluzione centralizzata può essere la scelta migliore in quanto si elimina tutto l'hoverhead di comunicazione/sincronizzazione tra i vari tier e possono essere fatte specifiche ottimizzazioni. In tali casi quindi un middleware centralizzato permette di ottenere prestazioni migliori riducendo brobabilmente anche i costi di realizzazione e manutenzione. Nonostante la nostra analisi si concentri nell'ambito delle applicazioni rfid, il middleware è stato progettato in modo da essere general purpose. Può essere utilizzato infatti per generiche sensor networks. Data la sua struttura modulare risulta inoltre essere altamente estendibile, fornisce infatti la possibilità di inserire logiche di filtraggio personalizzate, supporta differenti modalità di replicazione, attraverso l'aggiunta di un semplice driver specifico è inoltre in grado di interfacciarsi con qualsiasi tipo di data source. Prima di passare ad esaminare in maniera approfondita l'architettura del middleware, dobbiamo spendere due parole riguardo l'aspetto della replicazione: per il momento si è scelto di progettare un'architettura in cui solo il tier LLM è ridondato, la replicazione dell'HLM può essere spunto per sviluppi futuri.

5.3 Architettura del Sistema

La figura 5.2 mostra una visione d'insieme del sistema, indicando oltre al middleware anche i tier aggiuntivi e le altre entità che interagiscono con essi. Diamo un rapido sguardo d'insieme ai singoli componenti:

- **Data source**: è il tier relativo alle sorgenti dei dati, nel nostro scenario specifico è formato dai readers rfid

- **LLMs**: è il Low level middleware tier, il quale può essere replicato o meno. Verrà esaminato dettagliatamente di qui a poco

- **Configuration & administration Monitor**: è un modulo logicamente separato da entrambi i tier del LLM che ha il compito di monitorare il sistema, più esattamente l'insieme di processi che compongono quest'ultimo. In caso di guasto o sovraccarico di alcune istanze può avviare procedure correttive finalizzate alla redistribuzione del carico in modo tale da evitare che alcune porzioni del sistema rimangano isolate o comunque non possano essere in grado di svolgere il proprio compito. Il monitoraggio svolto da tale modulo è basato sulla raccolta di dati di carico e correttezza relativi alle singole istanze di HLM e LLM. Tale modulo inoltre gestisce l'eventuale riconfigurazione run-time dei data source.

- **HLMs**: è l'High level middleware tier, il quale al pari del LLM può essere o meno replicato.

- **DBMS**: si tratta di un database management system che ricopre il ruolo di back-end per l'attività svolta dal middleware e che gestisce anche la persistenza dei dati di configurazione utilizzati dal configuration & administration monitor.

- **External application notificator**: si tratta in sostanza di un presentation tier per l'interfacciamento dell'HLM con le applicazioni esterne.

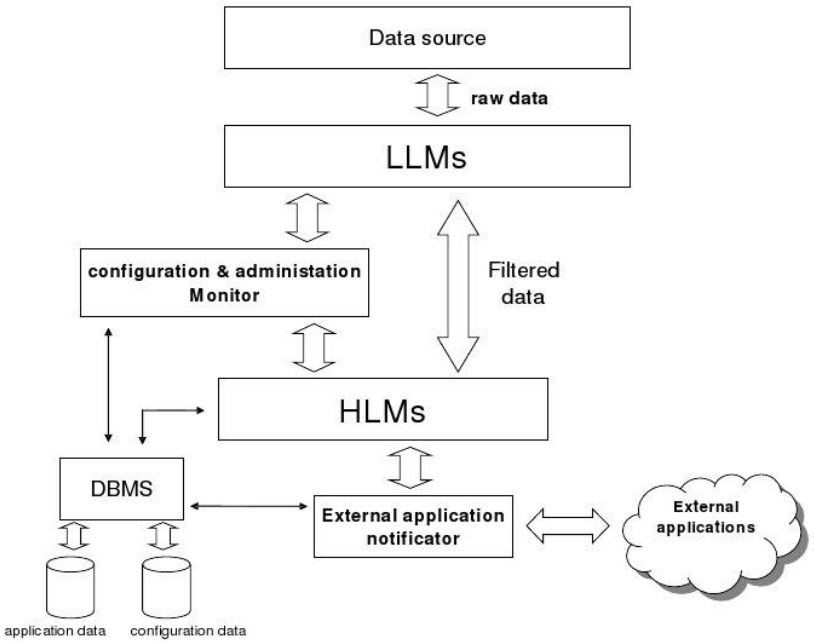


Figura 5.2 – Visione d'insieme del sistema

Nella figura sono indicati anche i flussi di dati scambiati tra i vari dati, analizziamone il contenuto:

- tra il tier data source e il LLM il flusso è composto dai messaggi contenenti le informazioni sui tag rfid presenti nell'area di illuminazione dei readers

- tra il LLM e l'HLM come abbiamo già visto in precedenza il flusso è formato dai dati filtrati

- tra LLM e monitor e tra HLM e monitor il flusso è formato da messaggi contenenti informazioni di carico e di correttezza delle istanze

- tra monitor e dbms il flusso è formato dalle informazioni di stato e di configurazione necessarie per lo svolgimento del compito del monitor

- tra HLM e External Application notificator il flusso di dati è formato dalle informazioni smistate dall'HLM e dirette alle applicazioni

- infine tra HLM e dbms il flusso è formato da quelle che sono le informazioni destinate al back-end

Passiamo ora ad esaminare l'architettura interna del middleware, ovvero dei tier LLM e HLM

5.3.1 Il Low Level Middleware

La struttura interna del LLM è di tipo modulare. Ad ogni modulo componente è stata assegnata una specifica funzione:

1. **LLM interface**: questo modulo/interfaccia si occupa dell'interazione tra LLM e Data source. Il driver specifico per quest'ultima può essere visto o meno come parte del modulo. Se viene visto come una porzione esterna allora si va a collocare tra l'interfaccia LLM e il data source.

2. **HLM comunication module**: ha il compito di gestire la comunicazione con l'HLM.

3. **Replication Module**: implementa tutte le funzionalità necessarie alla replicazione del LLM. Tale modulo è configurabile, ovvero è possibile scegliere il modello di replicazione desiderato, active replication vs. primary-backup, e lo schema (topologia) più adatto all'ambiente operativo. Dato che ogni istanza di LLM conserva delle informazioni di stato, tale modulo si occupa di assicurare la consistenza e la correttezza di queste informazioni tra tutte le repliche di LLM.
4. **Filtering**: si occupa del filtraggio dei dati. E' in grado di accedere alle regole di filtraggio, interpretarle ed applicarle ai

dati ricevuti dai data source

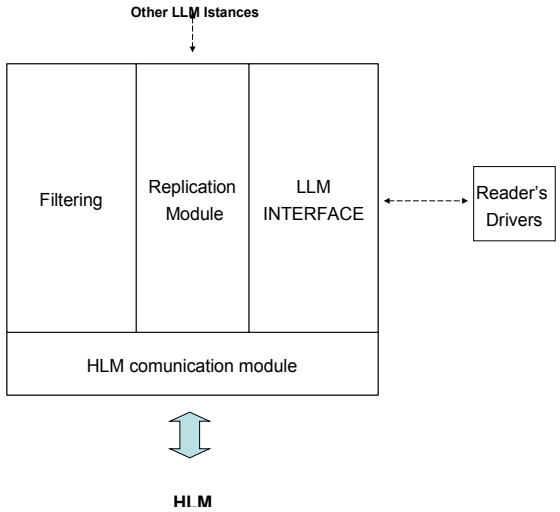


Figura 5.3 – Struttura interna del LLM

La figura seguente mostra il mappaggio di questi moduli sui componenti reali che realizzano il LLM.

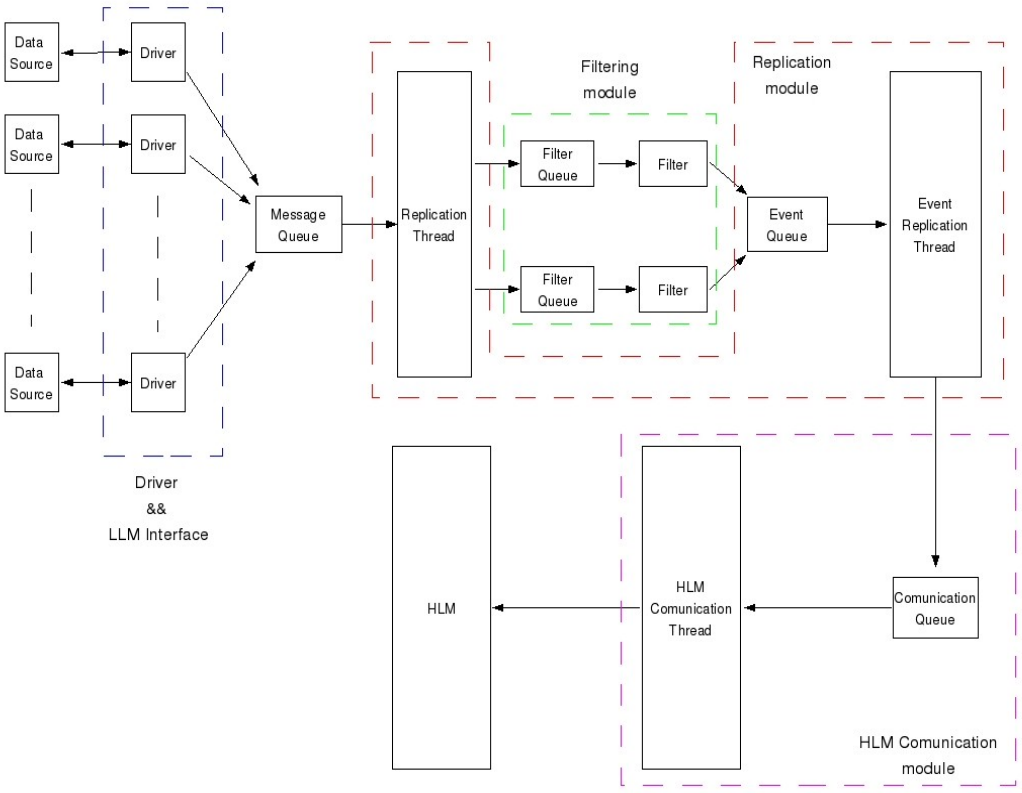


Figura 5.4 – Mappaggio dei moduli logici sui componenti reali

Vediamo ora in dettaglio i singoli componenti.

5.3.1.1 Drivers

Questi componenti si occupano dell'interfacciamento del LLM con le periferiche data source. Concettualmente il loro compito è praticamente identico a quello di un driver di periferica per sistemi operativi. Mappa i comandi proprietari del data source su quelli standard riconosciuti dal LLM ovvero in sostanza effettua un mappaggio tra l'interfaccia del data source e la LLM interface. Inoltre incapsula le informazioni provenienti dal data source in oggetti gestibili dal LLM. L'utilizzo di tali driver permette al middleware di gestire una vasta gamma di data source. Grazie all'utilizzo di un apposito file di configurazione il middleware è in grado di caricare dinamicamente le istanze di driver e in linea di principio è possibile gestire un numero infinito di tipologie di driver. Il numero massimo di istanze contemporaneamente gestibili è invece funzione della quantità di thread istanziabili sulla macchina su cui il LLM è in esecuzione. E' stato implementato un driver per il modello di rfid data sources Alien 9800. Il modello di programmazione per i driver verrà descritto nel sottoparagrafo 5.4.

5.3.1.2 Replication Module

Il replication module è mappato su due componenti il replication thread e l'event replication thread. Questi due componenti implementano le tecniche di replicazione necessarie per garantire l'affidabilità in caso di guasti del LLM. Vediamoli nel dettaglio. Il replication thread realizza quelle che nel paragrafo 6.2 verranno chiamate tecniche di sincronizzazione a priori. Per ora ci limitiamo a dire che questo componente, grazie ad un apposito file di configurazione è in grado di stabilire se l'istanza di LLM di cui fa parte deve o meno agire come un componente replicato, ovvero la replicazione può essere attivata o disattivata attraverso l'apposito file di configurazione. Se agisce da componente replicato è anche possibile scegliere tra varie tecniche di replicazione, ad esempio tra quelle basate su atomic broadcast oppure su optimistic atomic broadcast, semplicemente modificando il relativo file di configurazione. Il compito principale del replication thread, oltre a gestire eventualmente la replicazione, è quello di smistare opportunamente i dati provenienti dai driver dei data source verso la logica di filtraggio. Vediamo come avviene tale operazione. Le varie istanze della logica di filtraggio, che da ora in poi indicheremo con il nome di filtri, quando vengono istanziate indicano al replication thread quali sono i data source ai quali sono interessate. Il replication thread riceve i dati dai driver locali dei data source e grazie alle sottoscrizioni raccolte dai filtri è in grado di smistare opportunamente i dati. Naturalmente se ci sono due filtri differenti interessati ai dati provenienti dalla stessa sorgente, questi vengono prima duplicati e poi smistati ai destinatari. Se nel replication thread è attiva la

modalità di replicazione allora i dati in ingresso non vengono immediatamente smistati ma il thread esegue prima le operazioni necessarie per assicurare le proprietà del modello di replicazione utilizzato. Solo al termine di tali operazioni i dati vengono smistati ai filtri. La comunicazione tra drivers e replication thread è di tipo asincrono, viene infatti utilizzata una coda di messaggi. Questa è unica, quindi tutte le istanze di filtro vanno a depositare i messaggi ricevuti sulla stessa coda. La gestione FIFO della coda fa in modo che anche a livello di replication thread valga la proprietà fifo dei singoli canali che collegano l'LLM con i data source. La comunicazione tra replication thread e filtri è sempre di tipo asincrono ma avviene attraverso più code. Più precisamente è presente una coda per ogni istanza di filtro nel sistema. Tale coda è creata insieme all'istanza di filtro e la sua presenza è resa nota al replication thread attraverso la sottoscrizione dell'interesse del filtro per i data sources. Ancora la politica di gestione delle code garantisce che l'ordine FIFO dei messaggi sul singolo canale data source – driver venga propagato fino ai singoli filtri. Il replication thread ed il sistema di sottoscrizione sono realizzati in modo tale che i filtri possano essere istanziati anche run-time, creazione dinamica di istanze di filtro, ovvero il sistema può partire con un determinato numero di istanze filtranti, ma tale numero può cambiare durante la vita del sistema. Passiamo ora ad esaminare il componente event replication thread. Questo realizza una delle tecniche di replicazione tra quelle che nel paragrafo 6.3 vengono chiamate tecniche di sincronizzazione a posteriori, più precisamente realizza la tecnica che abbiamo indicato con il nome di sincronizzazione a posteriori a livello di LLM. Anche in questo caso, come per il replication thread le modalità di replicazione possono essere attivate o disattivate semplicemente modificando opportunamente un apposito file di configurazione. La sincronizzazione in questo caso riguarda naturalmente i dati in uscita dai filtri. Benché il middleware materialmente lo permetta, non sarebbe logico utilizzare contemporaneamente le tecniche di replicazione a priori e a posteriori. La correttezza del funzionamento è comunque garantita, ma ci sarebbe un sostanziale spreco di risorse dato che si eseguono due procedure di sincronizzazione, che pur se diverse, portano entrambe allo stesso risultato. A differenza del replication thread, l'event replication thread ha un solo compito principale, quello della replicazione. Ma data la posizione in cui si trova, è costretto ad eseguire anche un'altra operazione fondamentale, interfaccia i filtri con il modulo di comunicazione con l'HLM. Se non fossero state implementate tecniche di replicazione a posteriori tale modulo non sarebbe stato necessario in quanto i filtri avrebbero potuto interfacciarsi direttamente con l'HLM comunication thread. Il passaggio di dati dai filtri all'event replication thread avviene in modalità asincrona, attraverso un'opportuna coda, la stessa tecnica, utilizzando una coda differente, viene utilizzata anche per la comunicazione tra event replication thread e HLM comunication thread. Il funzionamento generale dell'event replication thread può essere riassunto in pochi passi: reperisce le informazioni filtrate dalla

coda di comunicazione con i filtri, esegue, se opportunamente configurato, le operazioni di sincronizzazione con le altre repliche di LLM, al termine di queste ultime immette i dati nella coda di comunicazione con l'HLM comunication Thread. Infine c'è da segnalare il fatto che le tecniche di replicazione sono state realizzate utilizzando alcuni dei servizi di comunicazione, in particolare servizi che implementano primitive di broadcast, messi a disposizione del framework Appia descritto al paragrafo 4.2.

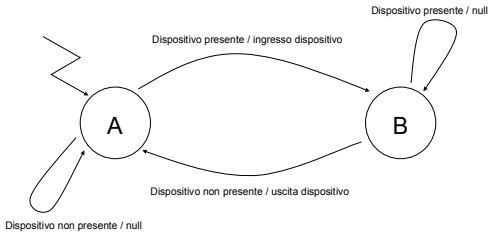
5.3.1.3 Modulo di filtering

Il modulo di filtering viene mappato su un insieme variabile di componenti, già introdotti nel sottoparagrafo precedente: i filtri. Questi sono componenti che implementano la logica di filtraggio dei dati. Attraverso tale logica i dati vengono vagliati e solo alcuni tra tutti quelli pervenuti vengono selezionati per la propagazione all'HLM. I filtri si differenziano tra di loro proprio in funzione della logica implementata, ovvero a logiche differenti corrispondono tipologie di filtro differenti. Sono state implementate solo poche tipologie di filtro, ma il middleware è stato sviluppato in modo tale da poter essere facilmente integrato con nuove tipologie. Non presentiamo qui il modello di programmazione per i filtri il quale verrà approfonditamente descritto nel sottoparagrafo 5.4. Un aspetto molto importante è la modellazione della logica di filtraggio. Per fare ciò si è scelto di usare il modello di macchina a stati finiti (FSM). Come è possibile vedere dagli esempi nei box 5.1, 5.2 e 5.3 il numero degli stati varia a seconda della logica di filtraggio e come ci si poteva facilmente aspettare a non tutti i passaggi di stato corrisponde una produzione di output da inviare all'HLM. E' qui che dobbiamo sottolineare un aspetto di fondamentale importanza che riguarda la replicazione del LLM: lo stato di una replica all'istante t è dato dall'insieme degli stati in cui le FSM che modellano ogni filtro si trovano proprio all'istante t, uno stato per ogni FSM. Semplifichiamo un po' il discorso ipotizzando la presenza di una sola istanza di filtro presso l'LLM. Lo stato di una replica di LLM all'istante t è dato dallo stato in cui si trova all'istante t la FSM che modella il filtro istanziato nella replica. Quando nel seguito della tesi ci troveremo a parlare di riallineamento dello stato allora si farà riferimento all'insieme di operazioni atte a fare in modo che le FSM replicate presenti in ogni replica di LLM e facenti capo ad un unico flusso di filtraggio siano forzate a posizionarsi tutte sullo stesso stato. Infine osserviamo che, come abbiamo già visto nei due sottoparagrafi precedenti, la comunicazione tra i filtri e i componenti ad essi adiacenti avviene in modalità asincrona attraverso l'utilizzo di code.

Filtro di tipo ENTRATO/USCITO

Tale tipo di filtro viene utilizzato quando gli eventi di cui si vuole tenere traccia sono solo quelli relativi

all'entrata e all'uscita di un determinato dispositivo dal campo di illuminazione di un data source. Ovvero quando si usa un filtro di questo tipo il LLM invia una messaggio all'HLM appena il data source rileva la presenza di un nuovo dispositivo nel suo campo di illuminazione. Se il dispositivo continua a stazionare in tale campo il LLM, anche se il data source continua a segnalarne la presenza, non invia più messaggi riguardanti tale dispositivo all'HLM. Quando invece il dispositivo esce dal campo di illuminazione il data source, non rilevandone più la presenza, non comunica più al LLM la presenza del dispositivo. Il LLM a questo punto capisce che il dispositivo non è più nel campo di illuminazione del data source e invia all'HLM un messaggio che indica l'uscita del dispositivo.

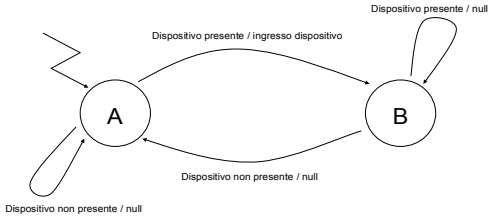


A = Dispositivo non presente
B = Dispositivo presente

Box 5.1

Filtro di tipo PASSED

Tale filtro viene utilizzato quando gli eventi di cui si vuole tenere traccia sono quelli relativi al passaggio di un dispositivo (TAG-ID) all'interno del campo di illuminazione di un data source.
Utilizzando questo tipo di filtro l'istanza di LLM invia un messaggio all'HLM non appena rileva la presenza di un nuovo dispositivo. L'istanza di LLM filtra tutti i successivi messaggi inviati dal data source che ne segnalano la presenza. Tale filtraggio persiste fino a quando il data source non invia un messaggio che non contiene più il riferimento al dispositivo in esame. Alla ricezione di tale messaggio l'istanza di LLM capisce che esso ha lasciato l'area illuminata. A differenza del filtro precedente però questa volta tale evento non viene comunicato all'HLM. Se il dispositivo, dopo essere uscito, rientra nuovamente nel campo illuminato, allora l'istanza di LLM comunica il nuovo ingresso all'HLM. In sostanza l'istanza di LLM invia un messaggio all'HLM alla prima rilevazione di un dispositivo, solo se questo non è mai stato presente nel campo di illuminazione o a seguito di una precedente uscita del dispositivo dal suddetto campo. Di seguito riportiamo la macchina a stati che modella il comportamento di questo tipo di filtro.



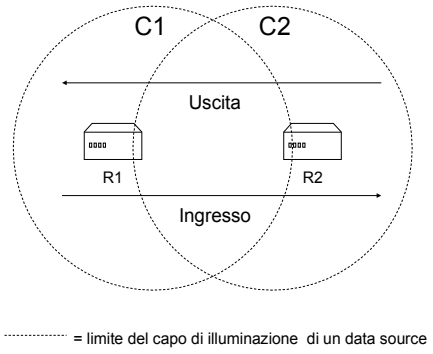
A = Dispositivo non presente
B = Dispositivo presente

Notare come le due macchine a stati differiscono solo nel passaggio dallo stato A allo stato B. La condizione di passaggio è la stessa, ma non viene generato nessun messaggio si segnalazione dell'uscita del dispositivo dal campo.

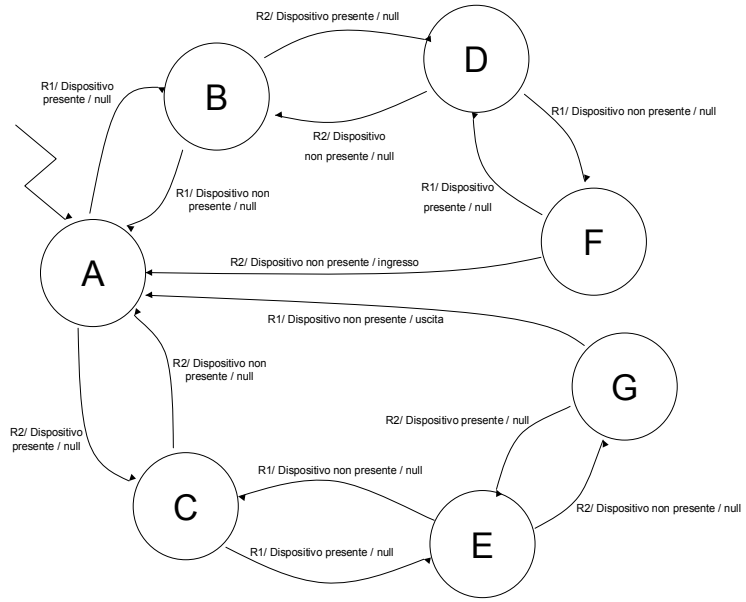
Box 5.2

Filtro di tipo PASS-THRU

Questo tipo di filtro può essere realizzato utilizzando il filtro di tipo PASSED appena visto.
Se però si vuole gestire la DIREZIONE di passaggio del dispositivo allora è necessario utilizzare un gruppo (almeno due) di data source i cui campi di illuminazione non coincidano totalmente.
La figura seguente mostra un esempio di dislocazione dei data source con indicazione del campo di illuminazione di ognuno di essi. Le frecce indicano l'interpretazione, ingresso o uscita, che viene data al moto del dispositivo.



Dato questo scenario, la macchina a stati che modella il comportamento del filtro è la seguente.



Notare come la macchina segnali l'uscita o l'ingresso del dispositivo solo se questo effettivamente attraversa entrambe le aree di illuminazione dei due data source.
Se un dispositivo attraversa una sola delle aree oppure se l'ingresso e l'uscita (del dispositivo dai campi di illuminazione) interessano un solo campo di illuminazione (anche se tra tale ingresso e uscita il dispositivo si è trovato a passare sull'altro campo illuminato) allora il filtro non invia nessuna segnalazione.
Quest'ultima viene in sostanza inviata solo ed esclusivamente se il dispositivo entra in un campo di illuminazione, senza uscire passa nell'altro campo di illuminazione ed infine esce dai confini di quest'ultimo.

Box 5.3

5.3.1.4 HLM comunication module

Questo modulo è mappato sul componente HLM comunication thread. Il compito principale di quest'ultimo è quello di trasmettere i dati filtrati all'HLM. La comunicazione tra HLM comunication module e HLM è basata su un'implementazione di astrazione di canale affidabile messa a disposizione dal framework Appia (vedi paragrafo 4.2). Se però si configura il middleware in modo da lavorare con la tecnica di sincronizzazione decritta al sottoparagrafo 6.1.3.2 allora l'HLM comunication module ha anche un altro compito fondamentale, attendere i dati di sincronizzazione dall'HLM e comunicarli al replication thread che provvederà ad ordinare ai filtri di riallineare il proprio stato.

5.3.2 L'High Level Middleware

Come indicato nel sottoparagrafo 5.2 l'architettura prevede la presenza di un tier High Level Middleware non replicato. Tale condizione ha permesso di progettare un tier con una struttura molto più semplice di quella del LLM.

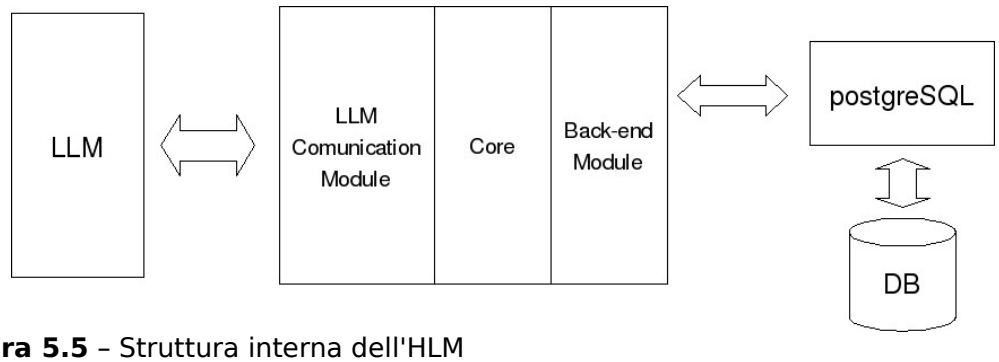


Figura 5.5 – Struttura interna dell'HLM

L'HLM può essere suddiviso in tre moduli principali:

- **LLM Comunication Module**: si occupa dell'interazione con l'LLM tier
- **Back-end Module**: si occupa dell'interazione con il back-end
- **Core**: rappresenta il nucleo dell'HLM. E' configurabile, ovvero in funzione della tecniche di sincronizzazione e invocazione utilizzate (capitolo 6), può o meno intraprende azioni atte a garantire il corretto funzionamento del sistema e la consistenza dei dati.

Descriviamo ora tali moduli separatamente.

5.3.2.1 LLM Comunication Module

Questo modulo è basato sul framework Appia. Il suo compito fondamentale è quello di realizzare la comunicazione tra il tier LLM e quello HLM di cui fa parte. Deve svolgere quindi tutte quelle operazioni atte a garantire che le proprietà di comunicazione necessarie per la correttezza dell'esecuzione delle operazioni siano rispettate. Il suo comportamento cambia a seconda della tecnica di sincronizzazione utilizzata, in funzione di quest'ultima infatti la comunicazione può essere unidirezionale, da LLM a HLM, o bidirezionale

5.3.2.2 Back-end Module

Questo è il modulo che si occupa dell'interazione tra HLM e back-end. Esso sfrutta il driver Java JDBC per interagire con il dbms, nel nostro caso PostgreSQL, che gestisce il DB di back-end sul quale vengono memorizzati tutti i dati filtrati provenienti dal LLM.

5.3.2.3 Core

Questo è il modulo centrale dell'HLM. Il suo compito è quello di sincronizzare le operazioni degli altri due moduli, back-end module e LLM comunication module, nonché di realizzare, vista la presenza di un tier LLM replicato, di una serie di operazioni finalizzate a garantire la consistenza dello stato del sistema. In sostanza il modulo Core si occupa di:

- smistare opportunamente i dati reperiti dall'LLM comunication module verso il back-end module
- Eseguire ulteriori operazioni di controllo e filtraggio su tali dati
- partecipare ad eventuali protocolli finalizzati al mantenimento di uno stato consistente per il sistema.

Le operazioni agli ultimi due punti verranno discusse con maggior dettaglio nel prossimo capitolo.

5.4 Modello di programmazione

Passiamo ora ad esaminare il modello di programmazione utilizzato nel middleware. In particolare ci concentreremo sul modello di programmazione che deve essere utilizzato per introdurre nuovi drivers e filtri all'interno del middleware.

5.4.1 Modello di programmazione per i drivers

Scrivere un driver per arricchire le capacità di interazione del middleware è un'operazione abbastanza semplice. Un driver è composto essenzialmente da tre classi:

- Una classe *DataSource*, la quale modella localmente il data source per il quale si sta scrivendo il driver

- Una classe *DriverThread*, la quale gestisce l'interazione tra LLM e il data source vero e proprio, attraverso un oggetto istanza di DataSource.

- Una classe *DataSourceObject*, la quale incapsula i dati inviati dalla specifica tipologia di data source gestita dal driver.

La prima classe dovrà implementare un'opposita interfaccia, chiamata *DataSourceInterface*. Questa contiene tre metodi che verranno utilizzati dal DriverThread, attraverso la loro invocazione su un'istanza di DataSource, per interagire con il data source vero e proprio:

- *startInteraction()*: utilizzato nelle comunicazioni con modalità di tipo push per avviare la produzione di dati da parte del data source. Tale metodo verrà invocato una sola volta.

- *waitMessage()*: utilizzato sempre nelle comunicazioni con modalità di tipo push. Permette al driver thread di mettersi in attesa dei dati prodotti dal data source. E' quindi di tipo bloccante. Questo metodo viene invocato più volte dal driver thread, ma la sua prima invocazione deve essere fatta obbligatoriamente dopo l'invocazione di startInteraction(). Restituisce i dati prodotti dal data source appena questi arrivano al driver incapsulati all'interno di oggetti istanze di DataSourceObject

- *requestData()*: utilizzato nelle comunicazioni con modalità di tipo pull. Permette al driver thread di inviare una richiesta di produzione di dati al data source vero e proprio. Anche questo metodo è ti tipo bloccante. Il thread quindi rimane in attesa fino a quando esso non ritorna. Ciò accade quando al driver arriva un messaggio proveniente dal data source. Naturalmente restituisce i dati prodotti incapsulati all'interno di oggetti istanze di DataSourceObject.

La seconda classe invece dovrà semplicemente estendere la classe Thread, predefinita in JAVA. L'istanza di DriverThread recupera i dati provenienti dal data source ed incapsulati in oggetti istanze di DataSourceObject, attraverso i metodi messi a disposizione dalla classe DataSource. Tali oggetti verranno poi immessi all'interno di una coda di tipo MsgReplicationQueue utilizzata per la comunicazione con il ReplicationThread. La terza classe come abbiamo già fatto notare in precedenza, incapsula i dati specifici della tipologia di data

source gestita dal driver. Tale classe estende la classe DataObject che a sua volta estende la classe astratta QueueObject che all'interno del nostro middleware contraddistingue gli oggetti che possono essere accodati nelle code di comunicazione. Discuteremo meglio in seguito la gerarchia e le caratteristiche di tali tipologie di oggetti.

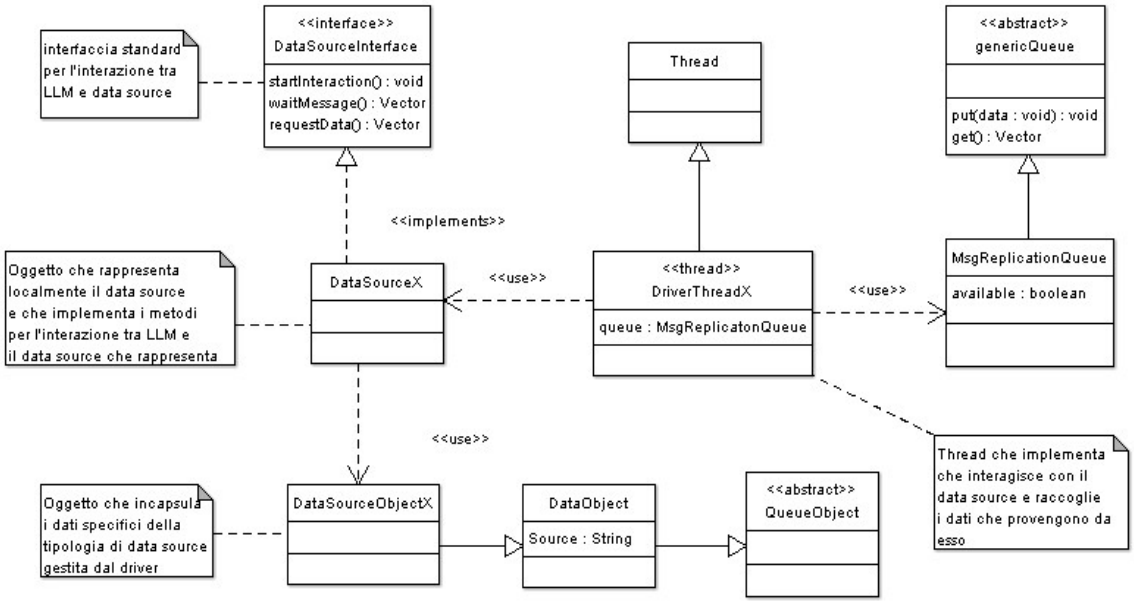


Figura 5.6 – Gerarchia di classi per la realizzazione di un driver

5.4.2 Modello di programmazione per i filtri

Anche questo modello di programmazione risulta essere abbastanza semplice. Per realizzare una nuova tipologia di filtro X è sufficiente implementare le seguenti classi:

- La classe *FiltroX*: implementa la logica di filtraggio vera e propria modellata con una FSM
- La classe *StatoX*: le istanze di questa classe modellano gli stati della FSM implementata dal filtro
- La classe *DataEventX*: le istanze di questa classe incapsulano i dati che devono essere propagati all'HLM

La prima classe dovrà estendere un'apposita classe Filter la quale implementa l'interfaccia FilterInterface. Quest'ultima prevede due metodi:

- *getStato()*: restituisce un oggetto di classe MachineState che modella lo stato in cui si trova il filtro al momento dell'invocazione
- *setStato()*: prende in ingresso un oggetto della classe

MachineState e forza il filtro a posizionarsi su tale stato

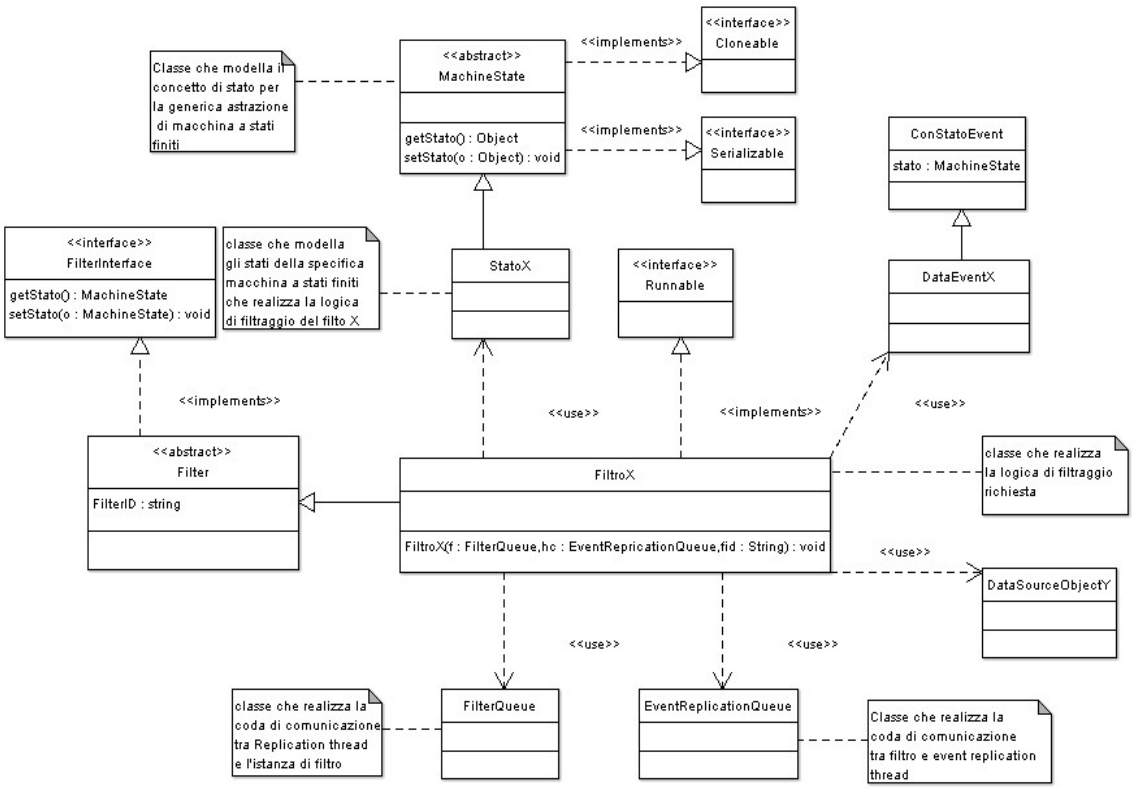


Figura 5.7 – Gerarchia di classi per la realizzazione di un filtro

Questi due metodi vengono realmente implementati all'interno della classe FiltroX. Il comportamento di quest'ultima può essere riassunto dei seguenti passi:

- reperisce i dati provenienti dai data source dalla propria istanza di FilterQueue
- applica la logica di filtraggio a tali dati
- ogni volta che vengono individuati dei dati da inviare all'HLM viene generato un oggetto DataEventX il quale incapsula:
  - I dati da propagare all'HLM
  - un oggetto della classe statoX che modella lo stato che ha generato i dati del punto precedente
- L'oggetto istanza di DataEventX viene immesso nella coda istanza di EventReplicationQueue utilizzata per la comunicazione tra filtri ed EventReplicationThread

Passiamo ora alla classe StatoX. Le istanze di questa classe incapsulano gli stati della FSM istanziata attraverso l'oggetto di tipo FiltroX. E' necessario definire una classe StatoX per ogni tipologia di

filtro in quanto ognuna di queste ha una FSM i cui stati possono essere caratterizzati da proprietà differenti. A questo punto dobbiamo evidenziare che gli oggetti di tipo StatoX incapsulati a loro volta all'interno di oggetti dataEventX creati dalle istanze di FiltroX dovranno giungere fino all'HLM. Tale operazione richiede una trasmissione dell'oggetto attraverso la rete, che a sua volta richiede la necessità di *serializzare* l'oggetto, è proprio per questo motivo che la classe MachineState, superclasse di StatoX, implementa l'interfaccia Serializable. La classe dataEventX incapsula i dati prodotti dal filtraggio da inviare all'HLM. Questa classe estende la classe ConStatoEvent, la quale identifica eventi che devono essere propagati ad altri tier e contenenti anche informazioni di stato. Come abbiamo visto poco fa lo stato incapsulato da dataEventX è quello relativo alla FSM istanziata dall'oggetto di tipo FilterX che ha prodotto i dati da propagare e incapsulato all'interno di un'istanza della classe StatoX. Come per lo stato, dato che ogni tipo di filtro può produrre un differente insieme di dati, differenti sia nella quantità che nella tipologia, c'è bisogno di una classe specifica che incapsuli le differenze in modo da renderle trasparenti al middleware. Infine ricordiamo che per poter ricevere i dati raccolti dai drivers ogni istanza della nuova tipologia di filtro dovrà sottoscrivere run-time presso il replication thread l'insieme di data source di interesse. Tale sottoscrizione viene effettuata attraverso l'inserimento di un'istanza della classe FDSAccoppiamento, contenente un riferimento ad un oggetto filtro, un riferimento all'insieme di data source sottoscritti e un riferimento alla coda del filtro, in una coda istanza della classe FilterListSharedQueue, controllata dal Replication thread.

5.4.3 Gli oggetti scambiati tra i moduli del LLM e tra LLM e HLM

In questo sottoparagrafo descriviamo in dettaglio le classi che incapsulano i dati scambiati tra i vari moduli che formano il nostro middleware, e delle quali abbiamo avuto modo di parlare nei due sottoparagrafi 5.4.1 e 5.4.2. La tassonomia di tali classi è mostrata nella figura 5.8. Cominciamo con l'analisi della gerarchia di sinistra. Partendo dalla superclasse QueueObject diciamo che questa modella tutti i possibili dati che i moduli si scambiano durante le fasi di comunicazione asincrona, basata sull'utilizzo di code. In queste ultime quindi vengono accodati tutti e soli gli oggetti riferibili da questa classe. QueueObject è una classe astratta, attraverso di essa si vuole semplicemente catturare la proprietà relativa alla possibilità di trasferimento attraverso inserimento in code che caratterizza tutti i dati scambiati tra i moduli del middleware. Per poter istanziare oggetti accessibili con riferimenti di questa classe è quindi necessario estenderla con classi di specializzazione. Ciò permette di far coesistere all'interno del sistema oggetti con caratteristiche differenti tutti trasferibili a mezzo delle stesse code, in sostanza ciò permette a dati differenti incapsulati da oggetti profondamente diversi l'uno

dall'altro di fluire in maniera trasparente all'interno del middleware.

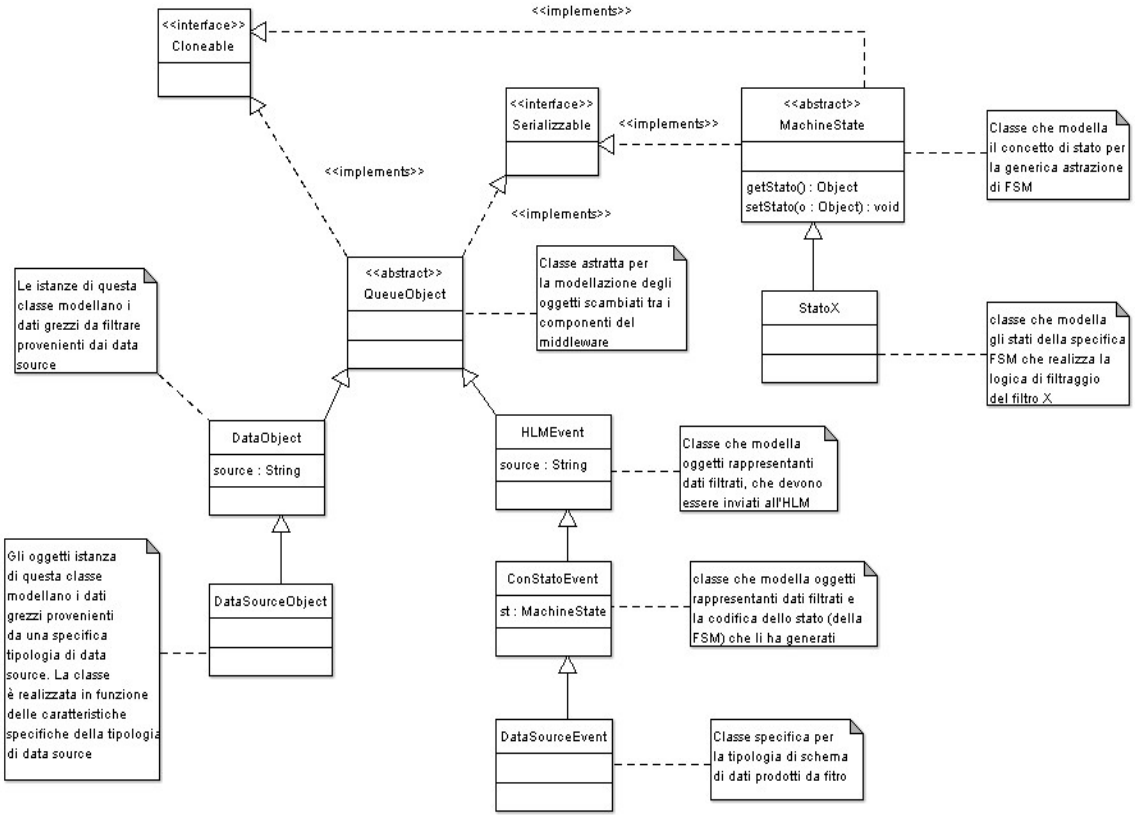


Figura 5.8 – Gerarchia di classi per l'incapsulamento dei dati e degli stati

Inoltre ciò consente allo sviluppatore di definire oggetti specifici per incapsulare dati provenienti da data source differenti e dati in uscita da filtri differenti senza generare problemi di compatibilità con il middleware. Notiamo infine che La classe QueueObject implementa l'interfaccia Serializable. Ciò è necessario in quanto gli oggetti che la estendono non solo vengono trasferiti da un modulo all'altro attraverso code, ma anche da un tier all'altro o da un'istanza di LLM all'altra attraverso la rete e quindi devono essere serializzati per poter essere trasmessi. Proseguendo verso il basso nella gerarchia vediamo che troviamo due classi che estendono QueueObject: DataObject e HLMEvent. La prima viene utilizzata per l'incapsulamento dei dati provenienti dal data source e diretti verso i filtri, la seconda viene utilizzata per incapsulare i dati generati dai filtri durante la procedura di filtraggio e diretti verso l'HLM. DataObject è poi esteso da DataSourceObject. Tale classe è stata immessa nello schema come esempio generico, ovvero, come abbiamo visto nel sottoparagrafo 5.4.1, la classe DataObject deve essere estesa da un'apposita classe progettata per incapsulare i dati di una specifica tipologia di data source. Di conseguenza per ognuna di queste dovrò avere un'opportuna classe che estende DataObject le cui istanze incapsulano i dati specifici della tipologia di data source. Passiamo ora all'altro ramo. Per la classe HLMEvent il discorso è leggermente differente rispetto alla classe DataObject. Nell'incapsulare i dati in

uscita dai filtri e diretti verso l'HLM infatti ci troviamo di fronte ad una situazione particolare. Potrebbe essere o meno necessario incapsulare insieme ai dati anche lo stato in cui transita il filtro a seguito della generazione dei dati stessi. Se non si manifesta tale necessità allora è sufficiente estendere HLMEvent attraverso una un'apposita classe le cui istanze incapsulano direttamente i dati prodotti dallo specifico filtro. La classe ConStatoEvent invece modella l'altra eventualità, quella in cui è necessario incapsulare anche lo stato. La classe ConStatoEvent non può però essere istanziata direttamente. Essa infatti è in grado di modellare solo la necessità di incapsulare anche lo stato, ma per poter accogliere dati specifici deve essere estesa da un'apposita classe, la cui realizzazione è guidata dalla tipologia e dalla quantità dei dati stessi. Nel diagramma diamo un esempio estendendo ConStatoEvent attraverso la classe DataSourceEvent. Naturalmente per ogni schema di dati specifici da incapsulare si ha bisogno di progettare una classe che estenda ConStatoEvent o direttamente HLMEvent. Passiamo ora alla gerarchia di destra. Questa risulta essere strettamente legata alla classe ConStatoEvent. La classe MachineState modella il concetto generico di stato di una macchina FSM. Ogni filtro presente nel middleware implementa una FSM che modella la logica di filtraggio. Le istanze di tale classe verranno quindi utilizzate per modellare gli stati delle FSM implementate dei filtri. Attenzione però, MachineState è una classe astratta, quindi non può essere istanziata direttamente. Ciò è coerente con il fatto che essa modella il concetto generico di stato. Lo sviluppatore, nel progettare e realizzare un filtro da introdurre all'interno del middleware dovrà realizzare una classe che estende MachineState e che realizza uno specifico modello di codifica ed incapsulamento degli stati della FSM implementata dal filtro. Nella figura la classe StatoX rappresenta un esempio di tale classe non astratta. Come abbiamo già osservato, dato che un'istanza di una classe che estende MachineState può essere riferita da un oggetto istanza di una classe che estende ConStatoEvent e questo può essere trasferito tra due tier differenti attraverso la rete, MachineState deve implementare l'interfaccia Serializzable, dato che un oggetto istanza della classe che estende MachineState potrebbe essere trasferito via rete.

5.5 Supporto agli schemi classici di replicazione

Il middleware, almeno per quel che riguarda il Low Level Middleware tier è predisposto per supportare i classici schemi di replicazione basati ad esempio sulle tecniche Primary - Backup e Active Replication. Alcuni di questi schemi di replicazione sono stati effettivamente implementati nel nostro prototipo, come ad esempio due schemi basati su atomic broadcast, più precisamente uno basato su atomic Broadcast e uno basato su optimistic Atomic broadcast. E'

stata inoltre progettata e realizzata all'interno del prototipo una tecnica innovativa per la replicazione che sfrutta una collaborazione tra tier per garantire la coerenza tra gli stati delle repliche. E' proprio l'ideazione di tale tecnica e la progettazione e formalizzazione di un protocollo per la sua realizzazione uno dei principali contributi di questa tesi. Le possibili tecniche di replicazione adottabili verranno illustrate in maniera dettagliata nel prossimo capitolo. Infine c'è da notare che il middleware fornisce una buona piattaforma/framework per lo sviluppo e la valutazione di schemi di replicazioni ad hoc, siano essi classici o innovativi

5.6 Integrazione con il framework di comunicazione

In questo sottoparagrafo diamo uno sguardo alla tecnica utilizzata per integrare il framework per la programmazione distribuita Appia all'interno dell'architettura del nostro middleware. La figura 5.9 mostra uno schema riassuntivo del modello di programmazione utilizzato per l'integrazione. Gli oggetti Channel del framework Appia vengono interfacciati con il LLM attraverso degli oggetti wrapper. I moduli del LLM e i componenti del framework Appia accedono ai servizi dei Channel invocando sugli oggetti wrapper i metodi che questi mettono a disposizione. Per descrivere in maniera più approfondita tale meccanismo, e dato che il framework è stato utilizzato esclusivamente per istanziare primitive di comunicazione di gruppo, utilizziamo l'esempio di un round di comunicazione in broadcast tra componenti. Se uno di questi vuole inviare delle informazioni agli altri allora invoca un metodo di send messo a disposizione del wrapper, passando come argomento un riferimento alle informazioni da inviare. Il wrapper accede alle informazioni e genera quello che nel framework Appia viene definito come evento. Quest'ultimo viene poi propagato sul relativo canale. Al momento della ricezione di informazioni inviate in broadcast un evento contenente che le contiene risale lungo l'Appia Channel fino all'Application layer. Arrivato a quest'ultimo l'apposito handler di gestione dell'evento recupera i dati ed invoca il metodo di consegna sull'apposito oggetto wrapper. Tale metodo non fa altro che andare a memorizzare delle informazioni all'interno di un'apposita coda alla quale accede anche il modulo destinatario delle informazioni. Perché si è scelto di utilizzare tale schema basato su oggetti wrapper? Esso garantisce un'elevata flessibilità permettendoci di operare sia in modalità sincrona, sia in modalità asincrona, potendo realizzare attraverso di esso operazioni di send e receive sia bloccanti che non bloccanti. Infine c'è da dire che per ogni QoS Appia, ovvero per ogni specifico protocollo di comunicazione distribuita, le cui istanze sono appunto i Channel Appia (vedi paragrafo 4.2), è necessario definire un wrapper dedicato.

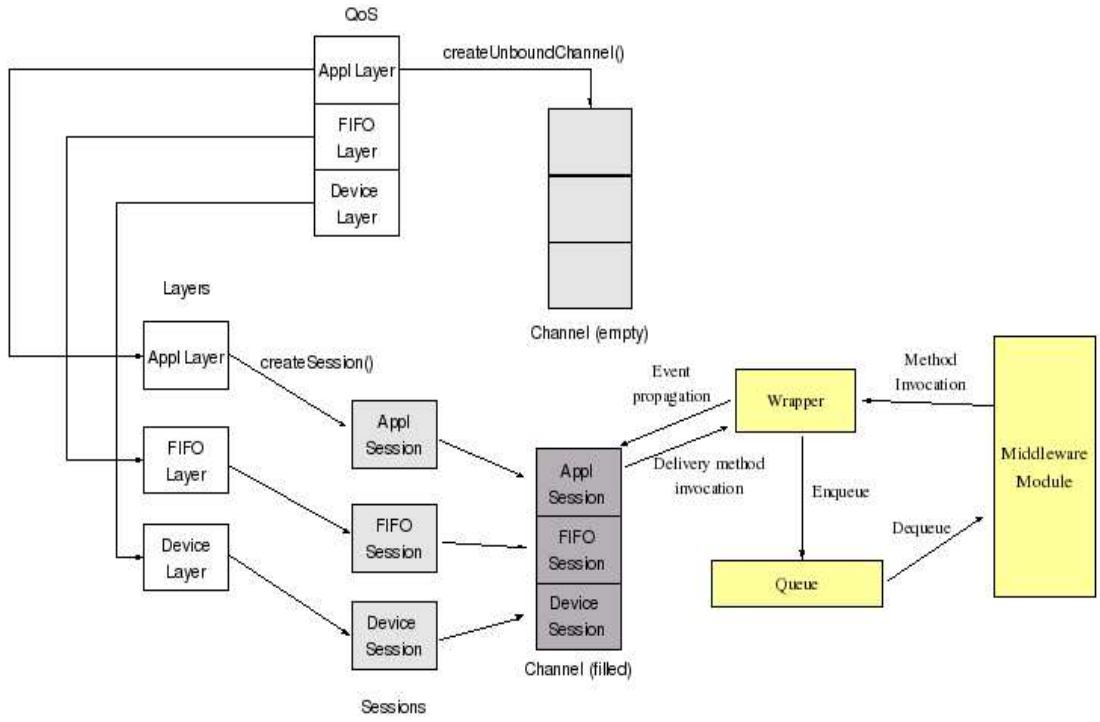


Figura 5.9 – Modello di integrazione del framework Appia nel middleware

Come si definisce e realizza un wrapper? E' sufficiente implementare un classe che metta a disposizione dei metodi mappabili sulle primitive previste dall'astrazione che si vuole utilizzare e che mantenga un riferimento per l'accesso al Channel istanza del QoS che modella l'astrazione. E poi compito del modulo che vuole utilizzarlo istanziare un oggetto wrapper ed invocare su di esso i metodi per l'utilizzo dell'astrazione implementata. L'inizializzazione dell'oggetto wrapper attraverso il metodo init(), che deve essere messo a disposizione dalla classe del wrapper, genera l'istanziazione dell'opportuno Appia Channel. Ciò permette quindi di isolare il framework dal middleware, con il vantaggio che eventuali modifiche sull'uno non implicano la necessità di riadattare anche l'altro, almeno fino a quando le interfacce dei servizi offerti dal framework rimangono inalterate.

6. LLM: Replicazione

6.1 Strategie di replicazione per il LLM

Andiamo ora ad esaminare le possibili modalità di replicazione del LLM inserendo nell'analisi anche questioni riguardanti le modalità di interazione tra le istanze di LLM con i data source e le garanzie offerte dagli strati di comunicazione che realizzano il broadcast dei dati provenienti dai sensori. Come mostrato dallo schema sottostante è possibile suddividere le architetture in due grandi rami, individuando in tal modo due famiglie di strategie: una basata sulla sincronizzazione dello stato delle repliche di LLM a priori, ovvero prima che queste processino i dati provenienti dai data source, l'altra basata sulla sincronizzazione a posteriori, ovvero dopo l'elaborazione dei dati, al momento della generazione di un evento da inviare all'HLM.

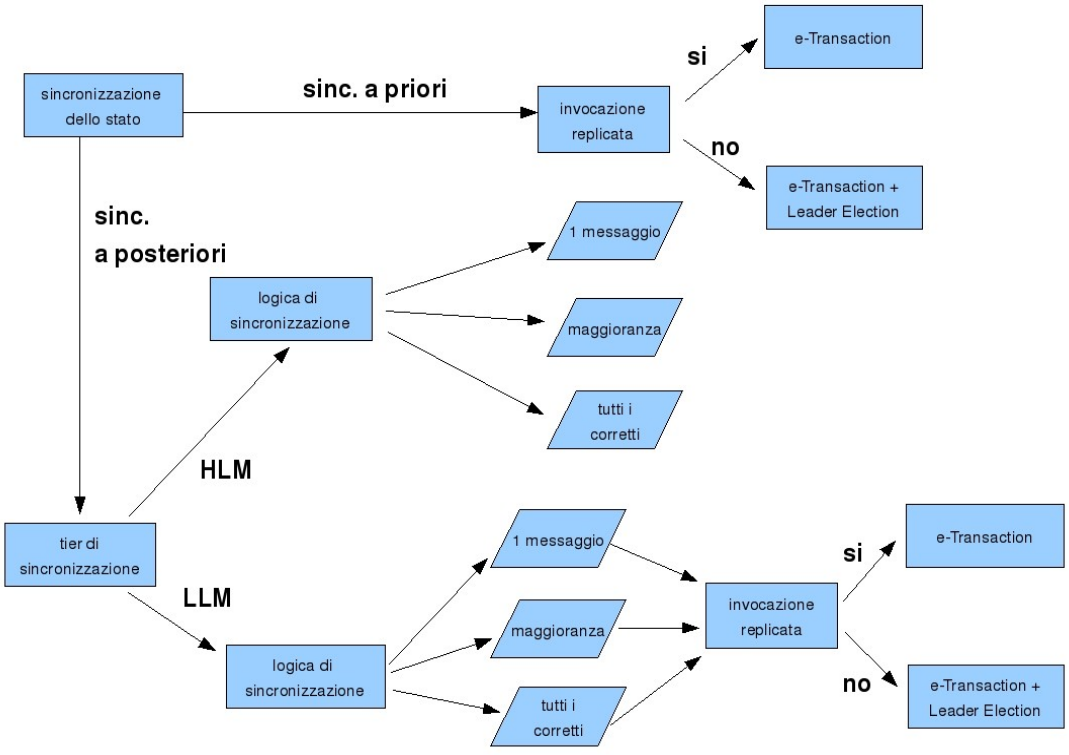


Figura 6.1 – Soluzioni di replicazione basate su Active Replication

Come è possibile intuire dalla figura, concentriamo la nostra attenzione solo su soluzioni di replicazione riconducibili alla tecnica classica Active Replication. Perché suddividere l'analisi in categorie

differenziate in funzione della “posizione” in cui viene eseguita la procedura di sincronizzazione? I motivi sono legati alle caratteristiche dello scenario applicativo di filtraggio di dati provenienti da tag rfid da noi indirizzato, le quali determinano la condizione che la posizione di esecuzione della procedura di sincronizzazione influisca in maniera sostanziale sulle prestazioni del sistema, vediamo in che modo. Consideriamo la frequenza di generazione dei messaggi da parte dei data source indicata con $F_{ws \rightarrow LLM}$ e quella di generazione degli eventi da parte del LLM indicata con $F_{wLLM \rightarrow HLM}$. Dato che la funzione principale del LLM è quella di filtrare i dati provenienti dai data source e propagare all'HLM solo eventi significativi determinati in base alle regole di filtraggio, in uno scenario tipico si avrà che $F_{ws \rightarrow LLM} \gg F_{wLLM \rightarrow HLM}$. Considerando ora la procedura di coordinazione, dato che questa viene eseguita sui messaggi/eventi, in linea di principio la dimensione e la quantità di messaggi ed il numero di round di comunicazione che la interessano saranno tutti direttamente proporzionali alle frequenze appena definite, con l'indice di proporzione che sarà funzione della specifica implementazione della procedura di coordinazione. Si arriva facilmente quindi alla conclusione che, in linea generale, una coordinazione a priori ha un costo in termini di tempo e di risorse di rete maggiore rispetto alla coordinazione a posteriori. Tale differenza si attenua man mano che si riduce il divario tra $F_{ws \rightarrow LLM}$ e $F_{wLLM \rightarrow HLM}$. Precisiamo che queste ultime considerazioni sono valide nel nostro specifico ambiente applicativo, in cui la relazione tra le due frequenze è determinata dalla particolare logica applicativa del LLM, in ambienti diversi, in cui la relazione che si determina tra le frequenze è differente, le considerazioni non risultano più valide.

6.1.1 Capacità dei data source e tecniche di replicazione

Prima di proseguire con l'analisi delle possibili soluzioni presentate nello schema di figura 6.1 è doveroso spendere qualche parola riguardo il legame esistente tra le capacità dei data source e le tecniche di replicazione adottabili. Per capacità di un data source intendiamo la possibilità di tale entità di poter comunicare secondo differenti modalità: pull vs. push, unicast vs. broadcast. Restringendoci al campo dei data source per rfid, chiamati comunemente rfid reader, ad esempio, vediamo che sul mercato è disponibile una grande quantità di modelli, le cui capacità possono differire di molto da modello a modello. Se disponiamo di un data source dalle prestazioni medio alte allora questo sicuramente sarà in grado di utilizzare differenti modalità di comunicazione, le quali possono essere anche riconfigurate run-time. In tal caso è possibile scegliere la tecnica di replicazione che si adatta meglio al nostro ambiente operativo. Se invece i data source a nostra disposizione hanno capacità limitate, allora la scelta della tecnica di replicazione

può risultare forzata. Ad esempio se il data source ammette solo la modalità di comunicazione di tipo pull unicast allora, a meno di non introdurre artificiosi meccanismi di comunicazione e sincronizzazione, è possibile utilizzare solo la tecnica primary-backup, la quale si adatta perfettamente alle caratteristiche del data source. Se quest'ultimo invece può comunicare in modalità push e ha la possibilità di propagare i dati in broadcast allora potrebbero essere usate con facilità tecniche di replicazione di tipo active replication. Se poi il data source è particolarmente evoluto allora potrebbe anche mettere a disposizione la possibilità di selezionare le modalità di interazione e di propagazione dei messaggi, non imponendo quindi vincoli sulle tecniche di replicazione utilizzabili. Nella nostra trattazione supporremmo di avere a disposizione data source, nello specifico rfid reader, che siano almeno in grado di implementare meccanismi, anche semplici, di broadcast in modalità push.

6.2 Sincronizzazione dello stato a priori

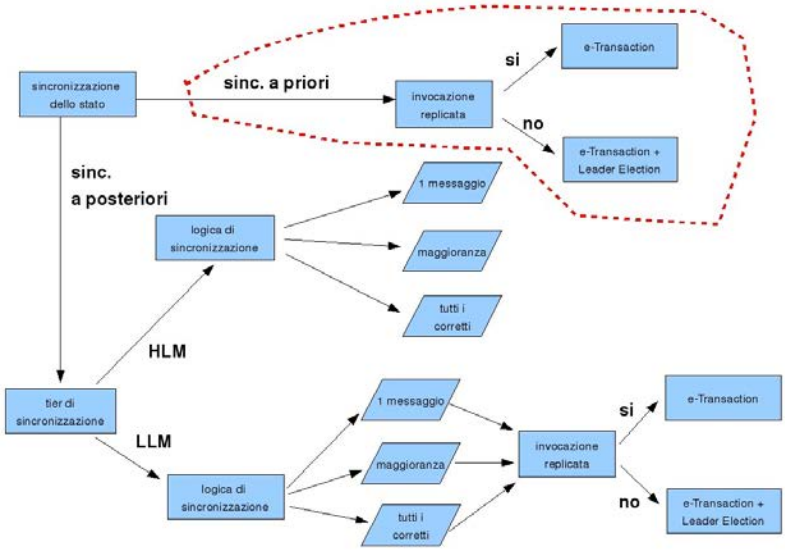


Figura 6.2 – Soluzioni con sincronizzazione a priori

Le tecniche di replicazione che ricadono in questo gruppo sono quelle che eseguono una qualche forma di sincronizzazione per l'allineamento dello stato prima del processamento dei dati provenienti dai data source. Riportandoci al nostro ambito operativo vediamo che a tale famiglia appartengono tutte quelle tecniche che prevedono la sincronizzazione dello stato delle repliche di LLM prima del processamento dei dati riguardanti i tag provenienti dai data source. La sincronizzazione dello stato consiste sostanzialmente nel fare in modo che tutte le repliche di LLM ricevano lo stesso insieme di messaggi dai data source, nello stesso ordine. Dato che ogni replica istanzia una macchina a stati finiti deterministica, ciò è sufficiente a

garantire che tutte le istanze di LLM seguano la stessa traiettoria di stato. Soluzioni per realizzare tale modalità di sincronizzazione possono essere basate ad esempio su primitive di comunicazione di tipo Atomic Broadcast. L'utilizzo di tale primitiva permette alle istanze di LLM di scambiarsi i messaggi ricevuti, in modo tale da determinare un insieme univoco di dati da processare, e di stabilire inoltre un unico ordinamento per i messaggi stessi. Queste due proprietà, unicità dell'insieme dei messaggi e univocità dell'ordinamento, sono sufficienti a garantire, insieme al determinismo delle repliche, che queste producano sempre output identici l'una all'altra. In pratica, la sincronizzazione a priori ed il determinismo delle istanze di LLM sono sufficienti a garantire anche la sincronizzazione a posteriori. Tale soluzione può però essere sottoposta ad una critica abbastanza pesante: la primitiva di Atomic Broadcast richiede un consistente overhead di comunicazione. Ovvero la quantità di messaggi scambiati per garantire l'unicità dell'insieme e il suo ordinamento è molto elevata, come può esserlo anche il numero di round di comunicazione necessari. La primitiva richiede cioè una sostanziale quantità di tempo e di risorse di rete, con conseguente ritardo della produzione dell'output. Una soluzione, anche se non definitiva, a tale problema può essere data dall'utilizzo della primitiva di Optimistic Atomic Broadcast. L'overhead di comunicazione è lo stesso, ma sfruttando la possibilità di consegna anticipata (ottimistica), nel caso in cui la frequenza di rollback e riprocessamenti, dovuta a differenze nell'ordine delle consegne ottimistiche e autoritative, si mantenga sufficientemente bassa, allora l'output può essere prodotto in un periodo di tempo sensibilmente inferiore rispetto a soluzioni basate su Atomic Broadcast. Qualsiasi sia la modalità di sincronizzazione selezionata, alla fine mi trovo ad avere una serie di repliche le quali producono più o meno all'unisono gli stessi output. A questo punto, nella nostra architettura, tali output prodotti devono essere inviati all'HLM. Per comodità d'ora in poi riferiremo un output prodotto da una istanza di LLM anche con il nome di '*evento*'. Le modalità di invocazione dell'HLM da parte del LLM sono oggetto di studio del prossimo sottoparagrafo.

6.2.1 Sincronizzazione dello stato a priori, invocazione replicata dell'HLM

Una volta prodotto l'output, se questo non è nullo, ogni replica può potenzialmente invocare l'HLM con lo scopo di comunicare i risultati del messaggio. Ci troviamo qui difronte ad una scelta. Quante e quali repliche si fanno carico di contattare l'HLM? Le possibili risposte a questa domanda sono tre:

- una sola replica
- tutte le repliche

- un sottoinsieme delle repliche

Diciamo subito che sotto l'aspetto della progettazione e dei meccanismi necessari per garantire la coerenza dei dati presso L'HLM le ultime due possibilità sono equivalenti, ovvero richiedono le stesse soluzioni, quindi possiamo far collassare la terza possibilità nella seconda. Alla fine quindi ci rimangono da esaminare due possibili scenari: invocazione replicata dell'HLM, invocazione non replicata dell'HLM. Per quanto riguarda il primo scenario, vediamo che sono state già studiate tecniche per la soluzione dei problemi di consistenza dei dati che possono presentarsi. Scendendo un po' nel dettaglio e calandoci nel nostra architettura di riferimento, l'invocazione replicata dell'HLM richiede la necessità da parte di quest'ultimo di individuare e filtrare eventi duplicati. Infatti, alla generazione di un evento, ogni replica invia un messaggio con lo stesso payload all'HLM.

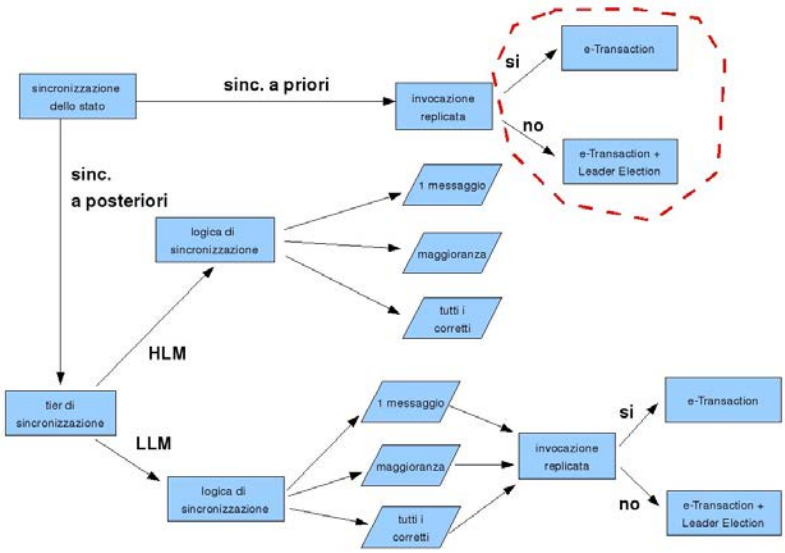


Figura 6.3 – Sincronizzazione a priori, invocazione dell'HLM

Quest'ultimo deve quindi evitare l'insorgenza di transazioni duplicate. Tale problema può essere risolto utilizzando tecniche basate su E-Transaction, presentate nel paragrafo 3.3.4. Per fare ciò è però necessario assegnare un ID univoco per ogni evento. Il momento ideale per il calcolo di tali ID potrebbe essere durante la fase di sincronizzazione a priori, le repliche quindi potrebbero scambiarsi oltre alle informazioni relative ai messaggi e al loro ordine anche quelle necessarie alla determinazione dell'ID. Questo potrebbe essere generato ad esempio a partire da informazioni contenute nei messaggi inviati dai data source, sempre che esse risultino adatte allo scopo. Passando invece al secondo scenario, vediamo che anche in questo caso esistono delle tecniche affermate per garantire la consistenza dei dati presso l'HLM. Scendiamo un po' più nel dettaglio. L'invocazione non replicata richiede la necessità di selezionare, tra tutte le repliche quella che dovrà contattare l'HLM per comunicargli

l'evento generato dalle operazioni di filtraggio. Tale problema viene risolto utilizzando primitive di leader election ed è quindi riducibile a quello del consenso. Ciò comporta la necessità di fare delle ipotesi di sincronia sul sistema, il quale deve essere almeno parzialmente sincrono, affinché sia ammessa l'esistenza di una soluzione. Supponiamo quindi di operare proprio con quest'ultima tipologia di sistema e di avere a disposizione anche un *eventually perfect failure detector.* Attraverso la primitiva di leader election quindi viene selezionata la replica che contatterà l'HLM per comunicargli il risultato del filtraggio. Se questa non si guasta allora l'interazione andrà a buon fine e non si avranno transazioni duplicate. Si consideri ora il caso in cui, dopo la leader election e prima della generazione di un nuovo ulteriore evento, l'eventual perfect failure detector segnali il guasto della replica selezionata. Le istanze di LLM restanti a questo punto non sanno se l'interazione tra replica sospettata e HLM è andata o meno a buon fine. Di conseguenza avviano nuovamente la procedura di leader election per selezionare una nuova istanza è tentare nuovamente di comunicare l'evento. A questo punto, dal lato HLM, si hanno due possibili situazioni: a) l'interazione precedente è andata a buon fine e quindi la seconda segnalazione genera un'ulteriore transazione per lo stesso evento, b) l'interazione precedente non è stata portata a termine e quindi la seconda segnalazione porta ad una nuova transazione. In quest'ultimo caso non si hanno problemi, in quanto dal punto di vista dell'HLM, è come se l'evento fosse stato ricevuto per la prima volta. Nel primo caso invece si presenta il cosiddetto problema delle transazioni duplicate, ovvero si genera una nuova transazione per un evento per il quale ne è già stata conclusa una. Di conseguenza anche in questo caso, come in quello delle invocazioni duplicate, risulta necessario adottare delle tecniche che garantiscano la consistenza dei dati presso l'HLM. Di nuovo possiamo utilizzare soluzioni basate su E-Transaction. Alla luce di quest'ultima conclusione, dato che in entrambi gli scenari è necessario fare affidamento su tali tecniche, sembrerebbe migliore la soluzione che prevede l'invocazione replicata dell'HLM, la quale non necessita di primitive di leader election e quindi di consenso. In generale è proprio così, fa eccezione un unico caso, quando il costo degli accessi in memoria secondaria (database) richiesti dalle soluzioni basate su E-Transaction per il filtraggio dei duplicati supera il costo introdotto dalle primitive di leader election dello scenario di invio non replicato.

6.3 Sincronizzazione dello stato a posteriori

In questa categoria ricadono le tecniche che eseguono la sincronizzazione solo dopo aver elaborato i dati provenienti dai data source, in pratica la sincronizzazione riguarda i risultati dell'elaborazione. Mappando tali concetti sul nostro scenario applicativo vediamo che la sincronizzazione riguarda lo stato delle repliche del LLM ed essa ha luogo solo a seguito della generazione di un evento da parte di una o più repliche. Più nel dettaglio, ogni replica processa i dati provenienti dai data source nell'ordine spontaneo di ricezione e senza preoccuparsi di ciò che è stato “visto” dalle altre. In sostanza le primitive di broadcast utilizzate per la comunicazione tra data source e LLM non forniscono le stesse garanzie del caso con sincronizzazione a priori. Supponiamo infatti di utilizzare qui primitive di broadcast semplici (es. BEB). Tale assunzione può portare ogni istanza di LLM a seguire una traiettoria di stato differente rispetto a quella delle altre. Appena viene generato un evento, da parte di una o più repliche, in funzione della tecnica utilizzata, si avvia una procedura di sincronizzazione che ha come risultato il riallineamento dello stato delle repliche, ovvero le loro traiettorie di stato vengono fatte convergere in un unico punto.

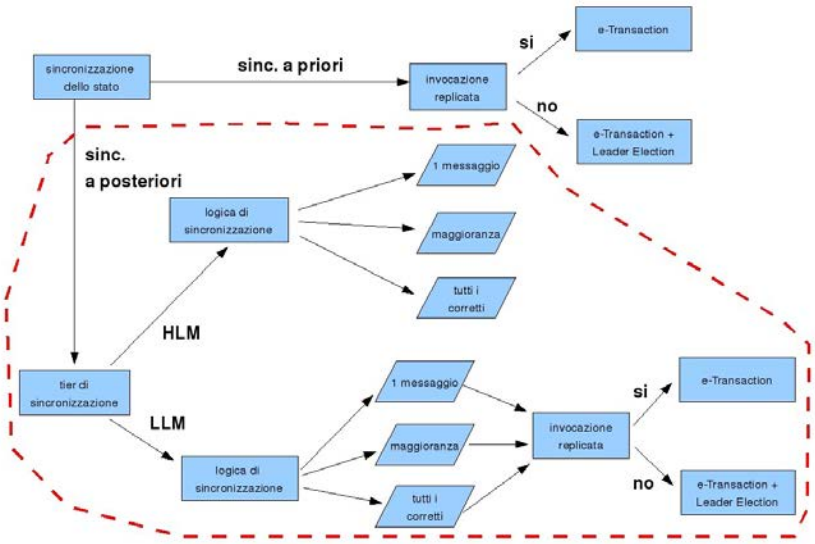


Figura 6.4 – Soluzione con sincronizzazione a posteriori

Le tecniche utilizzabili sono diverse e possono essere raggruppate in due categorie, a seconda del tier in cui viene eseguita la procedura di riallineamento:

- sincronizzazione a livello di LLM
- sincronizzazione a livello di HLM

Nel primo caso sono le repliche stesse che, prima di contattare l'HLM avviano una procedura attraverso la quale viene selezionato l'output da inviare all'HLM e da utilizzare per riallineare gli stati. Nel secondo caso invece è l'HLM che, dopo aver ricevuto un certo numero di eventi, seleziona quello che ritiene “corretto” secondo modalità che vedremo in seguito, e poi procede a comunicare tale selezione alle istanze di LLM che la utilizzeranno per riallineare gli stati. Esaminiamo ora separatamente e con maggior dettaglio le due categorie.

6.3.1 Sincronizzazione a posteriori a livello di LLM

In questa categoria ricadono tutte quelle tecniche che eseguono la procedura di sincronizzazione all'interno del tier LLM dopo la produzione dell'evento di output da parte di una o più repliche e prima di contattare l'HLM. Anche queste tecniche sono basate sostanzialmente su procedure distribuite di consenso. A seconda della logica di sincronizzazione utilizzata, della quale parleremo di qui a breve, a seguito della generazione di uno o più eventi, viene avviato un round di consenso. Le repliche sono chiamate a selezionare uno tra i gli eventi proposti. Al termine del consenso tutte le repliche saranno a conoscenza dell'evento selezionato. Procederanno quindi a riallineare il proprio stato interno, portandosi su quello associato alla selezione. Perché riallineare lo stato alla generazione di ogni evento?

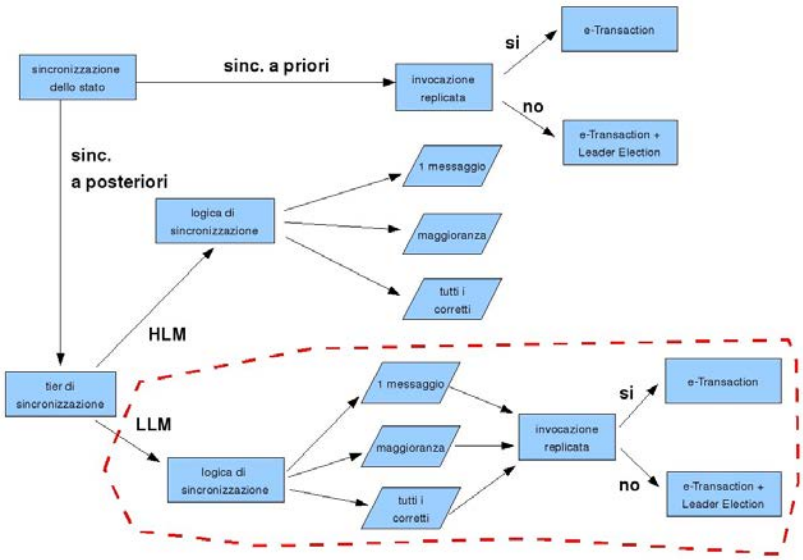


Figura 6.5 – Sincronizzazione a livello di LLM

Se si volesse rispondere brevemente a tale domanda sarebbe sufficiente dire che il riallineamento viene fatto per evitare che le traiettorie delle varie repliche divergano troppo con il passare del tempo. Scendendo un po' più nel dettaglio vediamo che utilizzando una specifica di astrazione di comunicazione utilizzata per il trasferimento di dati tra data source e LLM debole (es. BEB), esiste la probabilità che i messaggi vengano persi o consegnati in ordine

differente, probabilità che aumenta con l'aumentare della congestione della rete. La consegna di insiemi di messaggi differenti, eventualmente anche in ordine discorde, può portare le macchine deterministiche a divergere le une dalle altre. Tale rischio aumenta ancora di più quando l'ordine di consegna riveste un ruolo fondamentale nella semantica di generazione degli eventi e se questi vengono generati in funzione di un'interpretazione congiunta di dati provenienti da data source differenti. Se arriviamo ad una situazione tale per cui le traiettorie di stato divergono al punto tale da portare ogni replica a generare una serie di eventi totalmente differenti gli uni dagli altri, con la possibilità che questi siano anche discordi, allora, anche se si sono implementate politiche di sincronizzazione altamente efficienti per l'ambiente operativo, rimane molto alta la probabilità che vengano selezionati eventi non corrispondenti alla realtà monitorata. Per mantenere tale probabilità bassa procediamo al riallineamento degli stati sfruttando la procedura di sincronizzazione utilizzata per la scelta dell'evento da inviare all'HLM. Dopo questa digressione, torniamo a parlare delle caratteristiche della sincronizzazione a posteriori, riprendendo il discorso da un suo aspetto fondamentale, ovvero dalla logica di sincronizzazione utilizzata per la scelta dell'evento da comunicare all'HLM. Anche in questo caso abbiamo la possibilità di scegliere tra differenti soluzioni. Queste si differenziano in funzione di due aspetti fondamentali:

- la quantità minima di repliche che propongono un evento per iniziare il processo di sincronizzazione

- la tipologia di funzione di selezione utilizzata dalla procedura di consenso per la selezione della proposta che diventerà poi decisione

Per quanto riguarda il primo aspetto, anche qui abbiamo un numero piuttosto elevato di possibilità di scelta, le quali però sono riconducibili a tre classi generiche: procedura iniziata da n repliche, dove n è un numero prestabilito compreso tra 1 ed il numero totale di repliche di LLM; procedura iniziata dalla maggioranza delle repliche; procedura iniziata da tutte le repliche ritenute corrette. La prima classe verrà approfondita nel seguito. Per quanto riguarda le altre due possiamo dire brevemente che la seconda si sposa bene con quei modelli di sistema in cui si suppone di avere una maggioranza di processi corretti mentre la terza può essere utilizzata quando è richiesta un'elevata accuratezza nella generazione degli eventi e si hanno a disposizione dei meccanismi di failure detection per l'individuazione dei processi guasti. Esaminiamo ora l'altro aspetto fondamentale: la funzione di selezione della decisione tra tutte le proposte. Tale funzione, di tipo deterministico, è strettamente legata all'ambiente operativo, ovvero la funzione dovrebbe essere studiata in modo tale da assicurare la massima accuratezza possibile, ovvero dovrebbe cercare di selezionare l'evento che rispetta maggiormente i criteri di correttezza che caratterizzano l'ambito operativo. Alcuni

criteri utilizzabili sono: maggioranza, evento più recente, priorità ad eventi provenienti da istanze ritenute più affidabili. Anche questi criteri verranno esaminati in dettaglio nel seguito di questo capitolo. Passiamo ora ad esaminare la fase di invocazione dell'HLM, la quale è immediatamente successiva a quella di selezione dell'evento. Valgono in linea di massima le stesse considerazioni fatte per l'omonima fase del caso di sincronizzazione a priori (sottoparagrafo 6.2.1), con un'unica differenza: nel caso di invocazione non replicata non è necessario eseguire una procedura di leader election dedicata, per selezionare la replica che dovrà contattare l'HLM può infatti essere utilizzata la stessa procedura di consenso con la quale si seleziona una delle proposte.

6.3.2 Sincronizzazione a posteriori a livello di HLM

Tale modalità sfrutta un altro tier, l'HLM, per la selezione dell'evento “corretto” e per coordinare il riallineamento dello stato delle repliche del LLM. Essa è basata su un meccanismo innovativo, non riconducibile ad alcuna delle tecniche di replicazione già note. Diamo di seguito una breve descrizione del funzionamento di tale meccanismo che verrà descritto nel dettaglio per tutto il resto di questo capitolo, specializzandolo naturalmente per il nostro ambiente operativo: middleware per sistemi rfid. Scendendo un po' più nel dettaglio vediamo che nel tier relativo al LLM non vengono eseguite procedure di sincronizzazione, né prima né dopo il processamento dei messaggi provenienti dai data source. I messaggi inviati in broadcast da questi ultimi possono andare persi lungo la rete e possono essere consegnati dai processi in ordine arbitrario. Supponiamo però che ogni singola astrazione di canale tra un data source e un'istanza di LLM sia di tipo FIFO. Di conseguenza i messaggi spediti da uno specifico data source vengono consegnati dal ogni istanza di LLM nello stesso ordine, corrispondente a quello di invio, mentre messaggi provenienti da data source differenti possono essere consegnati in ordine arbitrario. Da ciò segue che anche in questo caso le macchine a stati deterministiche implementate dalle repliche possono seguire traiettorie di stato differenti e produrre quindi eventi anche discordi. Ogniqualvolta viene prodotto un evento questo viene inviato all'HLM. Una replica, in seguito ad un invio, non prosegue con la computazione dei messaggi provenienti dai data source, ma rimane bloccata in attesa di una risposta da parte dell'HLM. Quest'ultimo attende un certo numero di eventi, dopo di che tra tutti quelli pervenuti ne seleziona uno, quello che ritiene essere più “corretto”. Dopo aver selezionato l'evento invia in broadcast a tutte le repliche di LLM un messaggio di imposizione, che in generale dovrà contenere informazioni riguardanti l'evento selezionato e lo stato che lo ha generato. Le repliche, nel momento in cui ricevono tale messaggio, sia se sono in attesa a seguito di un invio di un evento, sia se invece si trovano in piena fase di filtraggio, allineano il proprio stato in

funzione dei dati trasportati dal messaggio dell'HLM. Si riesce in tal modo a realizzare una sincronizzazione tra le repliche senza eseguire una costosa ed esplicita procedura di consenso. Le interazioni tra LLM e HLM sono scandite da round, identificati con ID univoci, portati dai messaggi scambiati tra i due layer. Ogni round quindi termina con il riallineamento delle repliche di LLM.

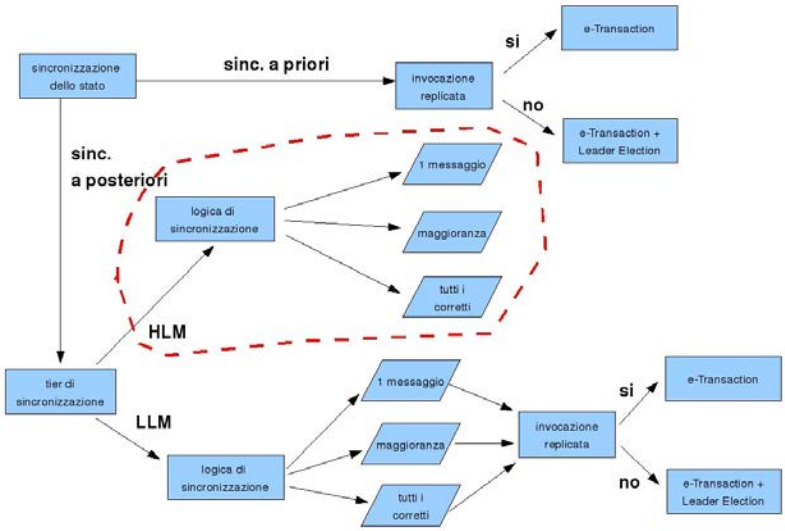


Figura 6.6 – Sincronizzazione a livello di HLM

Per quanto riguarda la logica di sincronizzazione, possono essere fatte considerazioni molto simili a quelle già viste per la sincronizzazione a posteriori a livello di LLM, traslandole dal punto di vista del tier HLM. Queste verranno comunque discusse in maniera dettagliata nei prossimi paragrafi, insieme con le politiche di selezione dell'evento corretto. Possiamo infine notare che la differenza più importante tra la sincronizzazione a livello di LLM e quella a livello di HLM sta nel fatto che la seconda non richiede l'esecuzione di procedure distribuite di consenso. Dato che di default quest'ultima tipologia prevede che più repliche di LLM contattino l'HLM non si pone il problema di scegliere tra invocazione replicata o invocazione non replicata dell'HLM.

6.4 Post Synchronized Active Replication Protocol

In questo paragrafo illustreremo un protocollo innovativo, chiamato Post Synchronized Active Replication Protocol (PSARP), che implementa una schema di sincronizzazione a posteriori a livello di HLM tier. Prima di procedere con la sua descrizione dettagliata è necessario definire il modello di sistema su cui esso è stato costruito. Tale modello è riportato nel box 6.1.

Processi

Sistema di tipo three tier:

- Data Source Tier
- LLM Tier
- HLM Tier

Data Source Tier
Questo tier è formato da una serie di entità non replicate, diciamo n. L'astrazione di processo per tali entità è di tipo Crash – Stop. Può quindi essere modellato nel seguente modo:

$$DST=\{D_{1,}....,D_n\} \quad \text{dove} \quad \forall D_i \in DST \quad \text{allora} \quad D_i \quad \text{è di tipo Crash-Stop}$$

Di conseguenza un'entità data source una volta che si guasta smette di inviare dati.

LLM Tier
Questo tier è formato da una serie di repliche, diciamo m, ognuna delle quali implementa lo stesso algoritmo. L'astrazione di processo per tali entità è di tipo Crash – Stop. Viene modellato nel seguente modo:

$$LLMT=\{LLM_{1,....},LLM_m\} \quad \text{dove} \quad \forall LLM_i \in LLMT \quad \text{allora} \quad LLM_i \quad \text{è di tipo Crash-Stop}$$

Di conseguenza una replica LLM una volta gusta smette di eseguire qualsiasi operazione.

HLM Tier
Questo tier è formato da una sola entità non replicata. L'astrazione di processo per tale entità è di tipo Crash-Recovery. Viene modellato nel seguente modo:

$$HLM \quad \text{di tipo Crash-Recovery}$$

Nel nostro caso l'astrazione di processo Crash – Recovery implica la possibilità per il processo HLM di guastarsi e fare recovery un numero arbitrario di volte. Ci sarà però un tempo t dopo il quale il processo farà recovery e non si guasterà più.

Canali di comunicazione

Utilizziamo esclusivamente l'astrazione di canale first-in-first-out point-to-point perfect link (FifoPp2p). Si suppone l'esistenza di un canale bidirezionale tra ogni coppia di entità presenti nel sistema. Più precisamente, indicando con (X,Y) il canale di comunicazione che collega l'entità X con l'entità Y, abbiamo:

$$\forall D_i \in DST \wedge \forall LLM_j \in LLMT \Rightarrow \exists (D_i, LLM_j) \quad \text{canale FifoPp2p}$$
$$\forall LLM_i \in LLMT \wedge LLM_j \in LLMT \Rightarrow \exists (LLM_i, LLM_j) \quad \text{canale FifoPp2p}$$
$$\forall LLM_i \in LLMT \Rightarrow \exists (LLM_i, HLM) \quad \text{canale FifoPp2p}$$

Sincronia

Non facciamo assunzioni riguardo la sincronia del sistema

Box 6.1

6.4.1 PSARP: Modello del Low Level Middleware

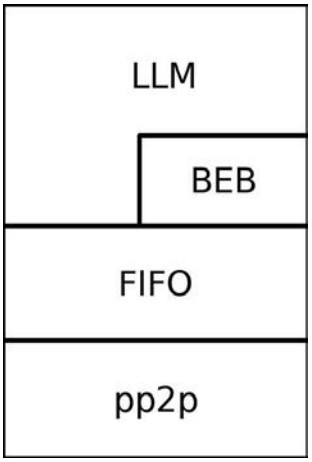


Figura 6.7 – Stack delle astrazioni su cui è basato il LLM

Ricordiamo che il compito fondamentale di ogni istanza di LLM come abbiamo visto è quello di filtrare gli eventi che arrivano dai data source che essa gestisce. Di seguito daremo una descrizione generale del comportamento di queste repliche e della loro modalità di interazione con i data source e con l'HLM. Il modello di funzionamento è illustrato attraverso un algoritmo scritto in pseudocodice e da una sua descrizione testuale.

Algoritmo 1 – LLM:

```
int localEventId := 0;
VectorClock localVc[|DST|];
Set DataSources := DST;
MessageNumber lastFromHLM := 0;
MsgList BufferedMsgs[DST];
Message lastMessageSend;

Upon event  <generateEvent | [output, statoFiltro]> do
        lastMessageSend = ([output, statoFiltro, localVc], HLM,
                                                        localEventId);
        pp2pFifoSend([output, statoFiltro,localVc], HLM, localEventId);
        localEventId++;
        startRetransmissionTimer();

Upon event <pp2pFifoDeliver | msg,  D_i  > do
        if(  msg.id>localVC[D_i]  ){
                bufferedMsgs[  D_i  ].add(msg);
                localVC[  D_i  ] = m.id;
                process(m);
        }
```

```
Upon event <BEBDeliver | [statoFiltro, vectorClk, eventId], HLM > do
        if (  eventId>lastFromHLM  ){
                resetRetransmissionTimer();
                lastFromHLM = eventId;
                clearBuffer(bufferedMsgs, vectorClk);
                SetState(statoFiltro);
                  ∀m∈BufferedMsgs  {
                        process(m);
                }
                localVC = max(localVC, vectorClk);
        }

Upon event <RetransmissionTimerExpired> do
        pp2pFifoSend(lastMessageSend);
        startRetransmissionTimer();

function clearBuffer(MsgList bufferedMsgs, VectorClock vectorClk){
                        ∀D_i∈DataSources
                  ∀msg∈bufferMsg[D_i] t.c. msg.id≤vectorClk[D_i]
                        bufferMsg[  D_i  ].remove(msg);
}

function VectorClock max(VectorClock localVC,VectorClock vectClk){
        VectorClock  temp[|DataSources|]
          ∀D_i∈DataSources
                temp[  D_i  ] = max(localVC[  D_i  ], vectClk[  D_i  ]);
        return temp;
}

function void SetState(Stato statoFiltro){
        funzione  che  impone  uno  stato  alla  macchina  a  stati
        implementata dall'istanza di LLM
}
```

NOTA : Mentre vengono eseguite le operazioni di gestione dell'evento <BEBDeliver | nack, statoFiltro, vectorClk, eventId, HLM> il thread di ricezione dei nuovi messaggi dai data source viene bloccato

Commento all'algoritmo:

int localEventId := 0;
VectorClock localVc[|DST|];
Set DataSources := DST;
Message lastFromHLM := 0;
MsgList BufferedMsgs[DST];

Ogni istanza di LLM deve mantenere traccia delle seguenti informazioni:

- un identificativo univoco degli eventi da essa generati (*int localEventId*);
- un vector clock la cui cardinalità è pari alla quantità di data source gestiti dalla replica e che all'istante t, nella entry i-esima, contiene l'identificativo dell'ultimo messaggio inviato dall'i-esimo data source (*VectorClock localVC*)
- un insieme che determina i data source da cui la replica riceve messaggi(*Set DataSources*);
- l'ultimo messaggio inviato dall'HLM e ricevuto dalla replica (*Message lastFromHLM*);
- un buffer strutturato come un array di liste in cui la lista i-esima contiene i messaggi inviati dall'i-esimo data source (*MsgList BufferedMsgs[DST]*)

Upon event <generateEvent |[output, statoFiltro]> do
pp2pFifoSend([output, statoFiltro, localVc], HLM, localEventId);
localEventId++;

Quando i messaggi processati portano ad uno stato che prevede la generazione di un output quest'ultimo viene esternato attraverso la creazione di un evento, il quale deve essere inviato all'HLM. Per fare ciò si sfrutta la primitiva di send dello strato di comunicazione pp2p (passando attraverso lo strato fifo). Vengono inviate le seguenti informazioni:

1. l'output generato (*output*)
2. l'identificativo dello stato che ha generato l'output (*statoFiltro*)
3. il vector clock locale che descrive la history dei messaggi processati (*localVC*)
4. un identificativo univoco per l'evento (*localEventId*)

Dopo l'invio viene generato un nuovo identificativo per l'evento

successivo.

```
Upon event <BEBDeliver | statoFiltro, vectorClk, eventId, HLM> do
    if (  eventId>lastFromHLM  ){
        lastFromHLM = eventId;
        clearBuffer(bufferedMsgs);
        SetState([statoFiltro]);
          ∀m∈OrderedArrived   {
              process(m);
        }
        localVC = max(localVC, vectorClk);
    }
```

Per il momento diciamo che l'HLM una volta esaminati un certo numero di eventi provenienti dalle repliche di LLM ne seleziona uno. Tale scelta viene poi comunicata al LLM inviandola a tutte le repliche. Per fare ciò l'HLM utilizza una primitiva di broadcast di tipo Best Effort (BEB), la quale garantisce che se il processo mittente è corretto allora tutti i processi destinatari corretti riceveranno il messaggio inviato. In caso di guasto del mittente però il BEB non garantisce la consegna presso i destinatari, quindi posso trovarmi davanti a tre possibili situazioni: a) nessuna replica di LLM ha ricevuto il messaggio, b) tutte le repliche lo hanno ricevuto, c) alcune lo hanno ricevuto altre no. Dato che il modello di processo per l'HLM è di tipo crash-recovery e che l'HLM prima di iniziare un broadcast salva su log il messaggio da inviare, al momento del recovery, non potendo determinare quali repliche hanno ricevuto il messaggio e quali no, l'HLM recupererà sempre dal log l'ultimo messaggio salvato e lo invierà in broadcast. Le repliche da parte loro sono in grado di riconoscere messaggi duplicati, dato che questi sono sempre etichettati con l'ID del round e quindi se lo hanno già ricevuto lo scartano, altrimenti dato che è nuovo, passano ad allineare il proprio stato in funzione del messaggio. Nel caso in cui invece l'HLM si guasti in fase di ricezione delle proposte, ovvero prima di effettuare la selezione e salvarla nel log, supponiamo durante il round n, si potrebbe avere una situazione in cui tutte le repliche di LLM hanno inviato la proposta, l'HLM si guasta, e queste rimangono bloccate in attesa della sua risposta. Quando l'HLM fa recovery, dato che non è riuscito a salvare nel log la decisione relativa al round n, non può inviarla alle repliche, piuttosto reperisce dal log la risposta relativa al round n-1, l'ultimo che si è concluso con successo, e la invia. Questa è scartata dalle repliche dato che esse sono in attesa di quella relativa al round n e quindi queste continuano a rimanere bloccate. Ci si trova così in una fase di stallo. Questa situazione viene risolta con un semplice meccanismo di ritrasmissione della proposta da parte delle repliche di LLM, ad esempio ad intervalli regolari. In questo modo è certo che prima o poi, quando l'HLM rimane attivo per un periodo sufficientemente lungo, il round sarà completato correttamente, garantendo l'evoluzione del protocollo. Naturalmente il meccanismo di ritrasmissione richiede all'HLM di scartare i duplicati, ciò viene fatto sfruttando l'ID del round e

l'identificativo del mittente. Il messaggio contenente l'evento selezionato, che da qui in poi verrà identificato anche con il termine “risposta”, porta le seguenti informazioni:

- l'id univoco dell'evento (*eventId*)
- la codifica dello stato che lo ha generato (*statoFiltro*)
- il vector clock ad esso associato (*vectorClk*)

La replica, appena consegna il messaggio, verifica innanzitutto che non si tratti di una vecchia risposta o anche di un duplicato. Se il controllo va a buon fine allora deve passare a riallineare il proprio stato con quello selezionato dall'HLM. Riassumendo: l'HLM seleziona una proposta tra tutte quelle che gli sono pervenute, dopo di che impone al LLM di riallinearsi ad essa. L'imposizione viene fatta a tutte le repliche, comprese quelle che sono già allineate, ne segue quindi che l'operazione di riallineamento deve essere idempotente, ovvero, sia x uno stato, se la replica cerca di riallinearsi ad x trovandosi già in x, allora vi rimane. Il riallineamento è preceduto da una serie di operazioni di aggiornamento delle variabili locali della replica:

- l'id della risposta *eventID* viene salvato in *lastFromHLM* in modo tale da poter tenere traccia dell'ultimo messaggio pervenuto dall'HLM
- il buffer dei messaggi ricevuti *bufferedMsgs* viene ripulito, eliminando i messaggi considerati obsoleti, scartabili.

Successivamente viene riallineato lo stato e vengono riprocessati tutti i messaggi rimasti in *bufferedMsgs*. Infine viene aggiornato il vector clock locale *localVC* nel seguente modo:

Ad ogni entry i in *localVC* viene assegnato il massimo tra il valore della i-esima entry in *localVC* e il valore della i-esima entry del vector clock portato dal messaggio.

Maggiori dettagli su tale procedura vengono dati in seguito, insieme alla descrizione della funzione che effettua l'aggiornamento. A questo punto è necessario aprire una digressione riguardo la manipolazione del buffer *bufferedMsgs* e il riprocessamento dei messaggi in esso contenuti. Di seguito è riportata la funzione che esegue la pulizia del buffer:

function clearBuffer(MsgList bufferedMsgs, VectorClock vectorClk){
    $\forall D_i \in DataSources$
        $\forall msg \in bufferMsg[D_i]$ t.c. $msg.id \leq vectorClk[D_i]$
            bufferMsg[D_i *].remove(msg);*
}

Prima di analizzare la funzione è necessario dare la definizione di due

concetti che verranno utilizzati nell'analisi:

DEF. *History(E_n,LLM_i)* : sequenza di messaggi processati dalla replica LLM_i che ha generato l'evento E_n a partire dall'istante successivo alla generazione dell'evento corretto $E_{(n-1)}$.

DEF. *Mc(E_n)* : messaggi appartenenti a History(E_n,LLM_i) t.c. LLM_i è stato scelto da HLM

Dato un evento E_n selezionato dall'HLM, una replica che riceve il messaggio di riallineamento contenente E_n può trovarsi in una delle seguenti situazioni indipendentemente dal fatto che il suo stato sia o meno già allineato con quello ricevuto:

- il suo vector clock locale coincide con quello portato dal messaggio

- il suo vector clock locale non coincide con quello portato dal messaggio.

Nel primo caso la replica ha ricevuto gli stessi messaggi della/e repliche che hanno generato E_n . In una tale situazione la funzione *clearBuffer(bufferedMsgs)* svuota totalmente *bufferedMsgs*, e non viene riprocessato alcun messaggio. Quando poi viene eseguita l'operazione *setState(statoFiltro)*, se la replica si trova in uno stato differente da *statoFiltro* viene riportata in quest'ultimo stato, se invece si trova già in *statoFiltro* allora vi permane. Nel secondo caso invece la replica ha ricevuto un insieme di messaggi differenti rispetto a quelli presenti in Mc(E_n). In questa situazione la funzione *clearBuffer(bufferedMsgs)* agisce su *bufferedMsgs* nel seguente modo:

- i messaggi appartenenti a $bufferedMsgs \cap Mc(E_n)$ vengono eliminati: tutti i messaggi presenti in *bufferedMsgs* con id minore o uguale a quello presente nel vector clock portato dalla risposta dell'HLM vengono eliminati

- i messaggi appartenenti a $BufferedMsgs \setminus Mc(E_n)$ permangono nel buffer e vengono immediatamente riprocessati: tutti i messaggi presenti in *bufferedMsgs* con id maggiore di quello presente nel vector clock portato dalla risposta dell'HLM permangono in *bufferedMsg* e vengono subito riprocessati. Se così non fosse, dato che sono stati già consegnati e non sono stati considerati nella generazione dell'evento prescelto, verrebbero persi e di conseguenza si avrebbe alta probabilità di un futuro disallineamento (in quanto le repliche che non li hanno ancora consegnati, lo faranno in seguito).
- i messaggi appartenenti a $Mc(E_n) \setminus BufferedMsgs$ arriveranno

alla replica corrente di LLM in futuro, ma dovranno essere scartati perché già considerati nella generazione di E_n : come si può avere una differenza di consegne in eccesso, si può avere anche una situazione di differenza di consegne in difetto, nel senso che il vector clock portato dalla risposta può segnalare il fatto che E_n è stato generato in seguito alla consegna e processamento di messaggi che nella replica locale (per la quale *localVC* è differente dal vector clock associato a E_n) ancora non sono stati consegnati. Nel momento in cui la replica si va a riallineare con le informazioni che gli sono comunicate dall'HLM, l'operazione di settaggio dello stato portato dalla risposta comprende il processamento dei messaggi non ancora ricevuti ma che hanno generato E_n . Ovvero l'imposizione dello stato associato a E_n include intrinsecamente la consegna ed il processamento di tutti i messaggi che hanno portato alla sua generazione. Le repliche che non li hanno ancora consegnati tutti e che si riallineano, prima o poi li consegneranno. Per evitare di avere situazioni di doppio processamento, dovranno scartare tali messaggi, ovvero dovranno scartare tutti quelli che loro ancora non hanno ricevuto ma che sono presenti in Mc(E_n). Il discarding viene fatto aggiornando opportunamente *localVC* con il vector clock portato dalla risposta e non processando quei messaggi che abbiano un id minore o uguale a quello contenuto in *localVC*. La procedura di aggiornamento del vector clock locale è realizzata attraverso la seguente funzione:

```
function VectorClock max(VectorClock localVC, VectorClock vectClk){
        VectorClock temp[|DataSources|]
          ∀D_i∈DataSources
                temp[ D_i ] = max(localVC[ D_i ], vectClk[ D_i ]);
        return temp;
}
```

Per completare l'analisi degli aspetti relativi al riallineamento dello stato vediamo ora la funzione con la quale effettivamente esso viene riallineato:

```
function void SetState(Stato statoFiltro){
    funzione che impone uno stato alla macchina a stati
    implementata dall'istanza di LLM
}
```

Questa funzione non fa altro che portare la macchina a stati che realizza il filtro nello stato passato come argomento alla funzione (*statoFiltro*). Come già detto essa è idempotente, ovvero se la macchina si trova già in *statoFiltro* allora essa vi rimane. Infine vediamo come viene gestita la ricezione dei messaggi provenienti dai data source. La porzione di algoritmo che si occupa di questo

compito è la seguente:

Upon event <pp2pFifoDeliver | msg, D_i > do
if(msg.id>localVC[D_i]){
bufferedMsgs[D_i].add(msg);
localVC[D_i] = m.id;
process(m);
}

Il messaggio viene consegnato dal canale pp2p. L'algoritmo verifica che l'id del messaggio sia superiore a quello contenuto nella entry di *localVC* (vector clock locale) relativa al data source mittente (devono essere scartati tutti i duplicati e tutti quei messaggi che pur se ancora mai ricevuti sono da considerare processati a seguito di una imposizione di stato da parte dell'HLM). Se tale verifica va a buon fine allora il messaggio viene bufferizzato, viene aggiornata la relativa entry di *localVC* ed infine il messaggio viene processato.

6.4.2 PSARP: Modello dell'High Level Middleware

Dato che stiamo ipotizzando di utilizzare un HLM centralizzato, è chiaro che tutte le repliche di LLM inviano gli eventi in output ad un unico destinatario. Analizziamo cosa può succedere al lato HLM durante un round di ricezione di eventi. L'HLM può ricevere tutti o solo un sottoinsieme degli eventi:

- riceve tutti gli eventi quando il round di comunicazione procede senza problemi

- se invece uno o più repliche di LLM si guastano, il messaggio spedito da ogni replica guasta potrebbe non giungere all'HLM, nonostante il canale sia di tipo perfect.

Nel primo caso non sorgono problemi, l'HLM può passare a scegliere quello che ritiene l'evento corretto tra tutti quelli ricevuti. Precisiamo che si suppone che le varie repliche di LLM evolvono in maniera indipendente l'una dall'altra e che se gli insiemi e/o l'ordine dei messaggi che le repliche ricevono in ingresso è differente allora le traiettorie di stato delle varie repliche possono divergere. Ciò comporta la possibilità di ricevere lato HLM eventi con lo stesso identificativo ma con contenuto differente e di conseguenza la necessità di selezionare uno solo tra gli eventi pervenuti, secondo un qualche criterio o funzione deterministica. Quindi se l'HLM riceve gli eventi da tutte le repliche, passa a scegliere quello corretto e poi invia una notifica di riallineamento alle repliche di LLM. Maggiori problemi invece si hanno quando la cardinalità dell'insieme degli eventi ricevuti non corrisponde alla cardinalità dell'insieme delle repliche di LLM. Ciò si può ad esempio verificare come già detto in

precedenza a seguito di guasti. Per garantire il corretto funzionamento del sistema potrebbe essere in questo caso necessario fare delle ipotesi di correttezza sull'insieme dei processi (come ad esempio supporre che la maggioranza dei processi sia corretta e quindi possa verificarsi il mancato ricevimento di al più (n/2)-1 messaggi). Continuando con l'analisi vediamo che, indipendentemente dal numero di eventi ricevuti, possono verificarsi, come già accennato in precedenza, due situazioni:

- tutti gli eventi ricevuti sono concordi
- gli eventi ricevuti non sono tutti concordi

Nel primo caso la situazione è semplice, l'HLM in sostanza non deve fare alcuna operazione particolare, si limita infatti a propagare l'evento ricevuto verso il back-end. Il fatto che tutti gli eventi siano concordi segnala che tutte le istanze di LLM di cui è pervenuto l'evento stanno evolvendo tutte sulla stessa traiettoria di stato. Nel secondo caso invece la situazione è più complessa. Avendo ricevuto eventi discordi l'HLM è costretto ad effettuare una scelta. Secondo un qualche criterio (ad esempio criterio di maggioranza) o comunque una qualche funzione deterministica deve selezionare l'evento da ritenere “corretto”. Sia se l'evento pervenuto è unico (primo caso), sia se l'HLM è costretto a stimare quello corretto (secondo caso), l'evento viene comunicato al LLM, forzando in questo modo le eventuali repliche divergenti a riallinearsi. Le possibili situazioni alla fine risultano quindi essere 4:

- Tutti gli eventi ricevuti, tutti identici
- Tutti gli eventi ricevuti, presenza di discordanze
- Ricevuti eventi da un sottoinsieme di repliche, tutti identici
- Ricevuti eventi da un sottoinsieme di repliche, presenza di discordanze

In caso di presenza di discordanze, qualsiasi sia la scelta fatta dall'HLM, il sistema deve garantire:

1) I messaggi già processati dai LLMs il cui output è stato scelto dall'HLM devono essere scartati da tutte le repliche di LLM. Per fare ciò è necessario aggiungere all'evento comunicato all'HLM un vector clock che riporta per ogni data source D_i l'identificativo (messageID) dell'ultimo messaggio processato proveniente da D_i . Ovvero, all'evento aggiungo un un messageID per ogni data source con cui la replica interagisce, ognuno di questi messageID è quello dell'ultimo messaggio processato proveniente dal data source a cui il messageID è riconducibile, in sostanza un vector clock, con tante entry

quanti sono i data source, ognuna delle quali contiene il messageID relativo all'ultimo messaggio processato proveniente dal data source al quale la entry si riferisce.

2) Riprendendo due definizioni date prima:

   def. $History(E_n, LLM_i)$ **:** sequenza di messaggi processati dalla replica LLM_i che ha generato l'evento E_n a partire dall'istante successivo alla generazione dell'evento corretto $E_{(n-1)}$.

   def. Mc: messaggi appartenenti a $History(LLM_i)$ t.c. LLM_i è stato scelto da HLM

   e aggiungendone una terza:

   def. Mnc: messaggi appartenenti a $History(LLM_i)$ t.c. LLM_i non è stato scelto da HLM

   allora i messaggi t.c. Mnc > Mc devono essere *riprocessati*, essendo la relazione d'ordine > definita in funzione del valore degli identificatori dei messaggi.

Il punto 2) può essere implementato attraverso la bufferizzazione dei messaggi ricevuti dai data source fino alla generazione di un evento. Questo tipo di implementazione pero può generare il problema di crescita unbounded del buffer, che può però essere risolto attraverso un meccanismo di sincronizzazione dei buffer delle repliche di LLM avviato da timeout periodici o da un evento generato nel momento in cui la dimensione del buffer supera una soglia predefinita.

Il Modello dell'High Level Middleware

Figura 6.8 – Stack delle astrazioni su cui è basato l'HLM

Vediamo ora il modello dell'High Level Middleware. Ricordiamo

che il compito principale di questo tier è quello di ricevere gli eventi dal LLM e di propagarli al back-end e a varie tipologie di applicazioni. Segue l'algoritmo che ne modella il comportamento.

Algoritmo 1 - HLM:

```
Set processes := LLMT;
MsgList receivedEvents := {}
int currentID = 0;
boolean firstEvent = true;

Upon event <pp2pFifoDeliver|[output,statoFiltro,vectorClk],eventId, LLM_i > do
        if (([output,statoFiltro, vectorClk], eventId, LLM_i ) ∉ receivedEvents
                                                        &&eventId == currentID ){
                receivedEvents.add([output,statoFiltro,vectorClk,eventId, LLM_i ]);
                if(firstEvent){
                        firstEvent = false;
                        startTimer();
                }
        }

Upon event <TimerExpired || receivedEvents≥β >* do
    resetTimer();
    currentID ++;
    decision = select(receivedEvent);
    log.store(decision, currentID);
    BEBSend(decision.statoFiltro, decision.vectorClock, currentID, processes);
    toBackEnd(decision);
    receivedEvents.clear();

Upon event <recoverFromCrash> do
        decision = log.getLastDecision();
        currentID = log.getLastID();
        BEBSend(decision, currentID, processes);
        toBackEnd(decision);

function [event, vector, process] select(Vector a)
        return ( [SF_i, VC_i] | [SF_i, VC_i] appare il maggior numero di volte in a);
                // SF = stato filtro        VC = vector clock

primitive:

void store(Decision d, EventID currentId)
        registra la decisione in memoria stabile, ad esempio su un file di LOG

decision getLastDecision()
        restituisce l'ultima decisione registrata nella memoria stabile
EventId getLastId()
```

restituisce l'identificativo univoco dell'ultima decisione registrata in memoria stabile

void toBackEnd([$Output_i, Stato_i, VectorClock_k$])
comunica al back-end la decisione presa, propagazione dell'evento al back-end

void startTimer();
inizializza e fa partire un timer, alla scadenza del conteggio del timer verrà generato un interrupt

void resetTimer();
resetta e disattiva un eventuale timer avviato in precedenza

Commento all'algoritmo:

Cominciamo il commento all'algoritmo supponendo che l'HLM, per svolgere i compiti ad esso assegnati, possa accedere ad una serie di primitive che gli vengono messe a disposizione dai componenti software di cui l'HLM stesso è composto. Tali primitive sono:

void store(Decision d, , EventID currentId)
registra la decisione in memoria stabile, ad esempio su un file di LOG

decision getLastDecision()
restituisce l'ultima decisione registrata nella memoria stabile

EventId getLastId()
restituisce l'identificativo univoco dell'ultima decisione registrata in memoria stabile

Dato che l'HLM segue il modello crash-recovery per assicurare che ogni round di interazione LLM-HLM venga portato a termine correttamente è necessario, prima di comunicarla alle repliche di HLM, che la decisione venga salvata in memoria stabile, insieme al suo ID univoco. In tal modo se l'HLM si guasta durante la procedura di imposizione della decisione, al successivo recovery esso è in grado di reperire la decisione presa prima del guasto, insieme al suo id, potendo in questo modo anche reinizializzare correttamente il valore di currendID con quello letto dalla memoria stabile.

void toBackEnd([$Output_i, Stato_i, VectorClock_k$])
comunica al back-end la decisione presa, propagazione dell'evento al back-end

Permette di inviare la decisione presa al back-end, per essa prevediamo una primitiva in quanto per il momento non entriamo nei dettagli di tale operazione.
void startTimer();

inizializza e fa partire un timer, alla scadenza del conteggio del timer verrà generato un interrupt

void resetTimer();
resetta e disattiva un eventuale timer avviato in precedenza

Queste due primitive mi permettono di utilizzare dei timer. startTimer() inizializza un timer e ne avvia il conteggio. resetTimer() blocca il conteggio disattivando sostanzialmente il timer. Se questo arriva a fine conteggio, non è quindi bloccato con resetTimer(), allora invia un interrupt all'HLM che lo gestisce avviando un'opportuno handler che vedremo di seguito.

```
Set processes := LLMT;
MsgList receivedEvents := {}
int currentID = 0;
boolean firstEvent = true;
```

Ogni istanza di HLM deve mantenere le seguenti informazioni:

- un riferimento all'insieme delle istanze di LLM da cui può ricevere eventi (Set processes). Come indicato nel modello del sistema con *LLMT* indichiamo l'insieme iniziale di tutte le repliche di LLM presenti nel sistema

- una lista che tiene traccia degli eventi pervenuti dalle istanze di LLM durante un round di interazione, che viene svuotata all'inizio di ogni nuovo round (MsgList receivedEvents)

- l'identificativo del round corrente (currentID)

- un booleano che mi indica se l'evento ricevuto da una replica di LLM è il primo del nuovo round di interazione

```
Upon event <pp2pFifoDeliver|[output,statoFiltro,vectorClk],eventId, LLM_i >do
    if (eventId == currentID){
        receivedEvents.add([output, statoFiltro, vectorClk, LLM_i ]);
        if(firstEvent){
            firstEvent = false;
            startTimer();
        }
    }
```

All'arrivo di un evento da una replica di LLM, l'HLM controlla innanzitutto se questo fa parte del round di interazione corrente, se si allora lo salva nella lista receivedEvents. Dopo di che verifica se si tratta del primo evento ricevuto per il round corrente. Se si allora avvia un timer utilizzando l'opportuna primitiva.

Upon event <TimerExpired || receivedEvents $\geq \beta$ *>* **(*)** *do*

```
resetTimer();
currentID ++;
        // commento: decision = [Output, StatoFiltro, VectorClock]
decision = select(receivedEvent);
log.store(decision, currentID);
BEBSend(decision.statoFiltro, decision.vectorClock, currentID, processes);
toBackEnd(decision);
arrived = {};
```

All'arrivo di un interrupt da timer (TimerExpired) o nel momento in cui all'HLM sono pervenute un numero di risposte $\geq \beta$ (*receivedEvents*$\geq \beta$) allora viene attivato questo handler. Si tratta sostanzialmente della procedura di decisione vera e propria. L'handler resetta il timer in modo da bloccare eventuali interrupt incoerenti (il reset è necessario nel caso in cui l'handler sia partito a seguito della condizione *receivedEvents*$\geq \beta$), dopo di che incrementa il currentID di uno in quanto si appresta a generare una nuova decisione. Dopo l'incremento seleziona tra tutti gli eventi ricevuti quello che ritiene più corretto. Le modalità con cui viene effettuata tale selezione verranno illustrate tra poco quando commenteremo l'opportuna funzione. La scelta dell'evento genera sostanzialmente la decisione. Quest'ultima e l'id corrente vengono successivamente salvati nella memoria stabile. La decisione presa viene inviata in broadcast alle repliche di LLM. Tale operazione è necessaria per poter effettuare il riallineamento di eventuali repliche disallineate. Dopo il broadcast la decisione viene inviata al back-end e la lista degli eventi pervenuti viene svuotata per permettere la corretta esecuzione del round successivo. Se l'HLM si guasta dopo la propagazione della decisione del round n al Back-end ma prima di aver effettuato l'operazione di decisione del round n+1, allora nel momento in cui esso fa recovery la decisione del round n verrà nuovamente propagata al Back-end. Di conseguenza è necessario che quest'ultimo sia in grado di rilevare eventuali duplicati e scartarli opportunamente.

perché introdurre il parametro β? (*)

La condizione che scatena l'evento dipende molto dal contesto operativo, dalle richieste di prestazioni e dalle garanzie che si vogliono fornire. Attendere che *correct*$\subseteq$*arrived* significa attendere che l'HLM riceva una proposta da tutte le repliche di LLM ritenute corrette. Dato che le repliche possono divergere si potrebbe avere una situazione in cui almeno una replica ritenuta corretta tarda ad inviare la proposta in quanto si trova su una traiettoria che porta ad una generazione tardiva dell'evento. Questa situazione prolunga in maniera consistente i tempi di decisione dell'HLM, riducendo le prestazioni del sistema. In merito c'è da fare la seguente considerazione: se rinuncio ad attendere tale proposta e prendo ugualmente una decisione con quelle pervenute? Se queste ultime sono in numero consistente rispetto al totale delle repliche molto probabilmente prendere una decisione su un sottoinsieme di proposte

(e non su tutto l'insieme delle proposte delle repliche ritenute corrette) non genera problemi. Volendo incrementare le prestazioni si potrebbe attendere un numero di proposte via via minore, fino ad arrivare all'estremizzazione di attendere una sola proposta ed eleggerla subito a decisione, imponendola immediatamente a tutte le repliche e propagandola al back-end. In questo modo si ha un sensibile aumento delle prestazioni, in quanto se le traiettorie delle repliche di LLM divergono il periodo tra l'arrivo della prima e l'ultima proposta può subire sensibili variazioni con l'evolversi del tempo. Il prezzo da pagare per l'aumento di prestazioni è però quello di un aumento della probabilità di eleggere a decisione un evento non corretto (correttezza intesa in termini dell'ambiente applicativo, nel nostro caso dell'ambiente monitorato dalla rete di sensori). Riassumendo:

- per $\beta = 1$ ho il massimo in termini di prestazioni ma la più alta probabilità di eleggere a decisione un evento “non corretto”

- per $\beta = |correct|$ ho la massima variabilità in termini di prestazioni, quindi il massimo rischio che non siano buone, ma ho la probabilità più bassa ottenibile di eleggere un evento “non corretto”.

- per $1 < \beta < |correct|$ ho tutta la serie di situazioni intermedie tra le due precedenti, con variabilità di prestazioni direttamente proporzionale a β e probabilità di “decisione sbagliata” inversamente proporzionale a β.

```
Upon event <recoverFromCrash> do
    decision = log.getLastDecision();
    currentID = log.getLastID();
    Bsend(decision, currentID, processes);
    toBackEnd(decision);
```

Dato il modello definito per l'HLM nel box 6.1, anche se se esso si guasta un numero arbitrario di volte, prima o poi tornerà on-line, e vi rimarrà per sempre. Quando ciò accade riuscirà sicuramente ad eseguire una serie di macroistruzioni finalizzate a concludere eventuali operazioni lasciate in sospeso a seguito del guasto e a reinizializzare opportunamente il proprio stato, ovvero:

- l'ultima decisione presa viene recuperata dalla memoria stabile e propagata in broadcast alle repliche di LLM (n.b. se queste hanno già ricevuto questa decisione prima del guasto dell'HLM allora si limiteranno a scartare la ritrasmissione della decisione)

- l'id corrente viene reinizializzato con quello letto dalla memoria stabile

- la decisione viene ripropagata al back end

function [event, vector, process] select(Vector a)
return ([SF_i, VC_i] | [SF_i, VC_i] appare il maggior numero di volte in a);
// SF = stato filtro VC = vector clock

Questa funzione è quella che effettua la scelta dell'evento “corretto” tra tutti quelli pervenuti all'HLM. essa semplicemente seleziona l'evento che occorre più volte nel vettore che gli viene passato come argomento. Nel caso in cui non sia presente un evento con tali caratteristiche può essere valutata una seconda discriminante per la sua selezione, ad esempio dare la precedenza a quelli provenienti da repliche di LLM con identificativo più basso (N.B. è sempre possibile stabilire una relazione d'ordine totale tra gli identificativi delle repliche di LLM). Ovvero la scelta potrebbe essere fatta nel modo seguente: prima si cerca di determinare l'evento che compare più volte nel vettore, se poi dovessero esserci degli ex-equo in vetta allora si passa a valutare gli identificativi delle repliche e tra tutti gli eventi a parimerito si sceglie quello inviato dalla replica con identificativo più basso. (Si potrebbe anche introdurre un'ulteriore discriminante basata sull'ordine parziale dei vector clock degli eventi da valutare prima di quella basata sugli id delle repliche). Indipendentemente dalla nostra implementazione, volendo analizzare le caratteristiche della funzione di selezione in maniera più generica possiamo vedere che essa è strettamente legata all'ambiente operativo, ovvero la funzione dovrebbe essere studiata in modo tale da assicurare la massima accuratezza possibile, dovrebbe cioè cercare di selezionare l'evento che rispetta maggiormente i criteri di correttezza che caratterizzano l'ambiente operativo. Alcuni criteri utilizzabili per realizzare una funzione di selezione deterministica sono:

- maggioranza
- evento più recente
- priorità a istanze ritenute più affidabili

Nel primo caso la funzione di selezione è basata sulla selezione dell'evento che, tra tutti quelli proposti ricorre un numero maggiore di volte, criterio questo basato sul concetto che “la maggioranza di solito è nel giusto”, usato ad esempio nel nostro protocollo. Nel secondo caso, applicabile quando è possibile stabilire una relazione d'ordine tra gli eventi, viene selezionato quello generato per ultimo, secondo il criterio che l'evento più recente con alta probabilità è quello che descrive meglio lo stato attuale del fenomeno monitorato. Infine il terzo caso è applicabile quando è possibile definire una funzione per il calcolo dell'affidabilità di una replica in esecuzione, in tal caso viene semplicemente scelto l'evento generato dall'istanza con maggiore indice di affidabilità. In tutti i casi potrebbero verificarsi situazioni in cui due eventi si ritrovano a parimerito in testa ad

un'ipotetica graduatoria stilata attraverso i criteri di selezione. E' necessario quindi introdurre una discriminante deterministica per affrontare tali situazioni. Quest'ultima, come già detto in precedenza, potrebbe essere ad esempio la relazione d'ordine totale tra le coppie (indirizzo IP, numero di porto) associate ad ogni replica del LLM. Facciamo notare inoltre che dato il modello di sistema per il tier HLM, di tipo crash-recovery, con evento selezionato salvato su log prima di propagarlo a back end e repliche di LLM, e dato che abbiamo fatto l'assunzione che non sia replicato, è possibile utilizzare anche una funzione di selezione che non sia deterministica. In caso di guasto e successivo recovery, un'eventuale imposizione lasciata in sospeso può essere portata a termine correttamente andando a recuperare l'evento selezionato direttamente dal log, che altrimenti non potrebbe essere ricalcolato.

7. Risultati Sperimentali

Dopo aver realizzato il nostro prototipo, questo è stato utilizzato per testare il protocollo PSARP e confrontarlo con le soluzioni basate su atomic broadcast. In questo capitolo descriveremo gli scenari di testing e i risultati ottenuti.

7.1 L'ambiente operativo

I test sono stati svolti tutti su una local area network (LAN), di conseguenza è stato necessario simulare artificialmente il ritardo subito dai messaggi in caso di deployment su reti di tipo wide area network (WAN). Ciò è stato fatto semplicemente ritardando la consegna di ogni messaggio di una quantità di tempo generata aleatoriamente e compresa tra due bound, uno superiore e uno inferiore, caratteristici per il tipo di rete. In questo modo è stato possibile simulare differenti scenari operativi, i quali corrispondono a diversi ambienti di utilizzo del middleware. I test sono stati effettuati distribuendo i vari processi su due macchine, le quali presentavano le seguenti caratteristiche: 1) Processore AMD Athlon 64 X2 4200+ Dual-Core, 2GB SDRAM DDR2, HD Serial ATA, sistema operativo Microsoft Windows XP SP2; 2) Processore AMD Sempron 2400+, 1GB SDRAM DDR, HD UltraATA 133, Sistema operativo Linux Ubuntu 5.10 con kernel 2.6.12. Il deploy dei processi è stato fatto secondo il seguente schema:

- sulla macchina 1) sono stati dislocati gli emulatori per i data sources, l'HLM tier, e il back-end.

- sulla macchina 2) è stato dislocato l'intero LLM tier

Le scelte di deployment sono state fatte in funzione delle seguenti considerazioni:

- la porzione di middleware che richiede maggiori risorse di calcolo è quella relativa al LLM (comunicazione, filtraggio, pulizia dei buffer, imposizione dello stato). Il tier HLM è piuttosto semplice, di conseguenza può condividere le risorse con altri processi senza risentire troppo della condivisione stessa;

- le minori risorse di calcolo della macchina 1) consentono di far arrivare facilmente a saturazione i protocolli testati (Il collo di

bottiglia non è la rete, bensì l'hardware del tier LLM), permettendoci quindi di determinare i limiti di ogni soluzione;

- eseguire gli emulatori di data sorces e l'HLM sulla stessa macchina permette di calcolare in maniera facile e precisa il tempo di risposta del sistema in funzione dei timestamp dei messaggi inviati dai data sources e dei timestamp delle transazioni che li riguardano, dato che tutti quanti questi timestamp vengono generati sfruttando lo stesso clock fisico.

Andando a dettagliare il deployment dei singoli processi istanziati vediamo che il carico della macchina 1) è costituito da sei repliche di LLM, mentre quello della macchina 2) da dodici istanze di emulatori di data source, una istanza di HLM e una istanza di back-end (server postgreSQL).

7.2 Parametri e modalità di esecuzione dei Test

Per effettuare i test è stato sviluppato un apposito filtro sintetico per il LLM. Il funzionamento di tale filtro è il seguente: viene generato un evento da comunicare all'HLM ogni x messaggi pervenuti al LLM, dove x è un numero casuale generato secondo una distribuzione di probabilità esponenziale. Il filtro di conseguenza utilizza un generatore di sequenze di numeri casuali, basato su distribuzione di probabilità esponenziale. Il funzionamento di tale generatore è ovviamente deterministico, ovvero se inizializzato con lo stesso seme e con lo stesso valor medio genererà sempre la stessa sequenza di numeri casuali. Tale caratteristica di determinismo è fondamentale per ottenere dati che permettano di effettuare un confronto significativo tra i diversi protocolli implementati. I test sono stati eseguiti facendo variare non solo la configurazione delle reti di interconnessione delle varie istanze, ma anche il carico e il valor medio δ della distribuzione esponenziale utilizzata dal filtro sintetico. Per quanto riguarda la variazione del carico questa può essere realizzata in tre modi differenti: facendo variare il numero di data sources con i quali il LLM tier deve interagire, facendo variare la frequenza di invio dei messaggi da parte dei data sources, o combinando le due modalità precedenti. I nostri test sono stati svolti facendo variare la frequenza di generazione dei messaggi, tenendo quindi fisso il numero di data sources. Per quando riguarda il valor medio δ questo è stato fatto variare di volta in volta di un ordine di grandezza o quasi (1, 10, 50). Le singole run hanno avuto una durata compresa tra i 10 ed i 30 minuti, a seconda della combinazione frequenza di generazione–valor medio utilizzata, in modo tale da raccogliere un numero sufficiente di dati, dopo il transitorio iniziale, per estrapolare indici coerenti.

7.3 Risultati

In questo paragrafo discuteremo nel dettaglio i risultati dei test effettuati. Attraverso l'utilizzo di alcuni grafici, i quali riportano dati di throughput e di latenza di risposta del sistema, verranno confrontate le prestazioni, nei vari scenari, di due protocolli di replicazione: PSARP, ovvero la nostra soluzione, e Atomic Broadcast, nella sua versione più performante, Sequencer Atomic Broadcast. Naturalmente facciamo sempre riferimento al nostro modello di sistema, il quale prevede che i processi non siano mai soggetti a guasti. Un tale modello risulta particolarmente vantaggioso per il protocollo Sequencer Atomic Broadcast, il quale subisce un sensibile degrado prestazionale in caso di guasto del processo sequencer, eventualità questa che non può verificarsi date le assunzioni sul modello di sistema da noi considerate. Analizziamo separatamente i risultati ottenuti nei due scenari di testing indicati nella tabella 7.1.

7.3.1 Scenario 1: componenti distribuiti su scala geografica

Il grafico 7.1 mostra il limite di saturazione delle due soluzioni al variare del valor medio δ. Tale limite rappresenta il valore del throughput del sistema in corrispondenza del suo punto di saturazione, esso naturalmente è espresso in termini di messaggi al secondo. Per punto di saturazione intendiamo il livello di carico che porta il sistema ad avere una latenza di risposta doppia rispetto a quella presentata nel caso in cui sia scarico.

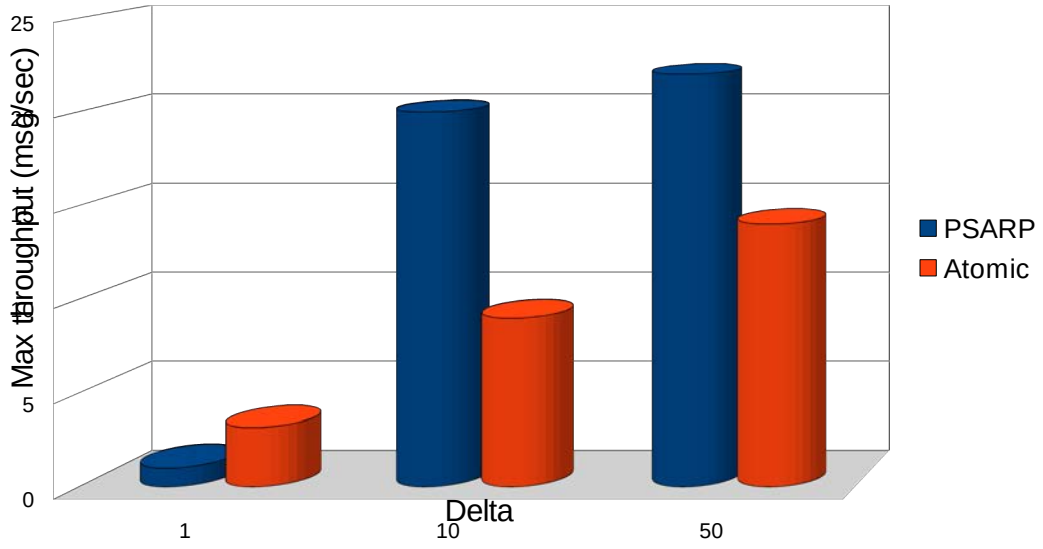


Grafico 7.1 – Throughput massimo sostenibile al variare di δ

Partendo da sinistra e procedendo verso destra possiamo vedere che per δ = 1 la soluzione basata su atomic broadcast offre prestazione

migliori rispetto a quella basata su PSARP. La prima infatti presenta un limite di saturazione più elevato rispetto alla seconda, ciò gli permette di supportare un carico più elevato in termini di messaggi in input. Le basse prestazioni di PSARP sono dovute essenzialmente all'elevato numero di rollbacks che si verificano ad alto carico. All'aumentare di δ però vediamo che si verifica un'inversione di tendenza. Nel grafico, per δ = 10, la nostra soluzione supera sensibilmente, di oltre il 50%, quella basata su atomic broadcast. Per valori di δ che via via si allontanano da 1, il peso del round aggiuntivo di comunicazione influisce sempre di meno sulle prestazioni, la quantità assoluta di rollbacks diminuisce, e quindi il punto di saturazione tende a spostarsi verso l'alto. C'è da sottolineare però il fatto che ad un aumento di δ corrisponde anche un aumento del costo computazionale per l'imposizione dello stato presso ogni replica di LLM. E' quindi necessario curare l'implementazione di tale procedura al fine di minimizzare il tempo necessario per il suo completamento. Se la differenza tra la riduzione di carico dovuta al minor numero di rollbacks e l'aumento del costo di imposizione dello stato è positiva allora il punto di saturazione di PSARP tende a spostarsi verso l'alto, e di conseguenza le prestazioni migliorano. Passiamo ora ad esaminare i tempi medi di risposta del sistema, al variare di δ e del carico in ingresso. Questi in generale aumentano con l'aumentare del numero di messaggi in ingresso al sistema per unità di tempo. I relativi grafici sono il 7.2, il 7.3 e il 7.4 che rispettivamente indicano la latenza di risposta del sistema per δ = 50, δ = 10, δ = 1. Partendo dal primo grafico vediamo come in media PSARP garantisce tempi di risposta molto più bassi rispetto alla soluzione basata su Atomic Broadcast. La latenza della prima soluzione, quando il sistema non è in saturazione, è influenzata dai ritardi di rete, comunicazione data source – LLM e del round trip tra LLM e HLM per l'invio della proposta e l'imposizione dello stato, e dal tempo di attraversamento dei tier LLM e HLM.

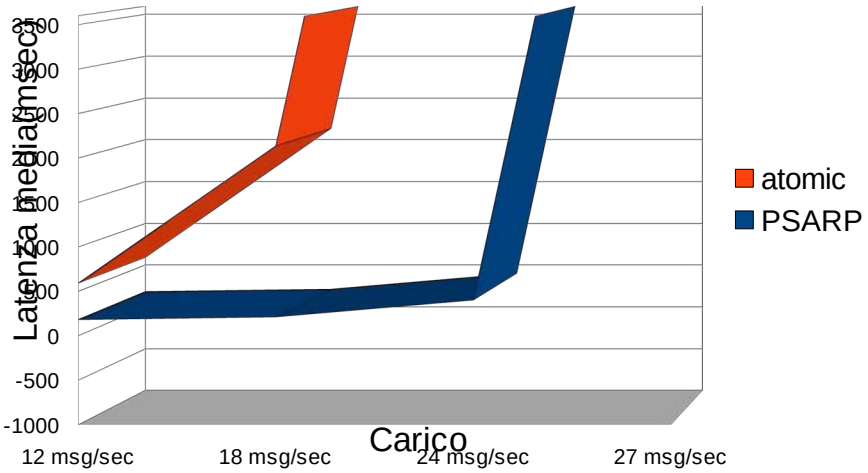


Grafico 7.2 – Latenza media per delta = 50

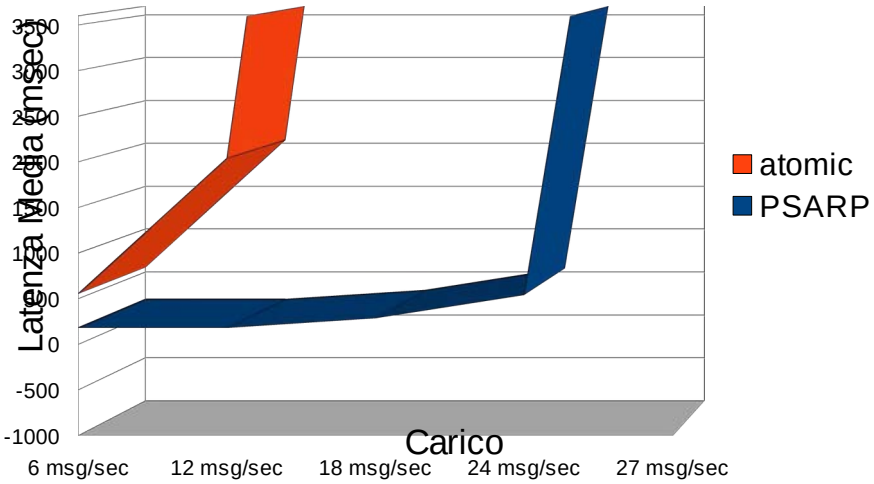


Grafico 7.3 – Latenza media per delta = 10

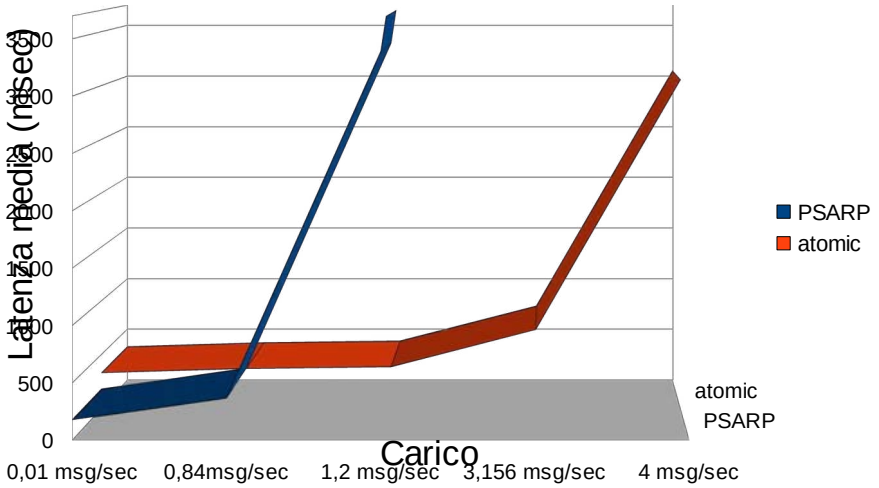


Grafico 7.4 – Latenza media per delta = 1

L'insieme di tali costi risulta essere di molto inferiore rispetto al totale di quelli sostenuti da atomic broadcast per garantire la consegna ordinata a priori dei messaggi provenienti dai data source. Le stesse considerazioni possono essere considerate valide anche per δ = 10. In questo caso naturalmente, come già visto in precedenza, il limite di saturazione è più basso, e quindi la crescita esponenziale dei tempi di risposta inizia a valori di carico di ingresso più bassi: circa 22 messaggi al secondo per δ = 50, circa 20 messaggi al secondo per δ = 10. Passando invece al grafico 7.4 vediamo che in questo caso è la soluzione basata su atomic broadcast a fornire una latenza media più bassa rispetto a PSARP. Il degrado di prestazioni di quest'ultimo è dovuto all'aumento della frequenza del round trip di comunicazione tra HLM e LLM, che ora in media viene eseguito una volta per ogni messaggio in ingresso al LLM, ma soprattutto, dato che aumenta il carico, all'incremento del numero di rollback, determinato dall'aumento della probabilità di consegna dei messaggi in ordine

differente presso ogni replica.

7.3.2 Scenario 2: componenti distribuiti su scala locale

Passiamo ora ad esaminare le prestazioni dei due protocolli nel secondo scenario, quello in cui tutte le istanze dei vari tier sono interconnesse attraverso local area network. Come ci si poteva aspettare e come confermeranno i dati discussi nel resto del paragrafo vedremo che, rispetto allo scenario 1, abbiamo una riduzione generale dei tempi medi di latenza di risposta del sistema ed uno spostamento verso l'alto delle soglie di saturazione dei protocolli.

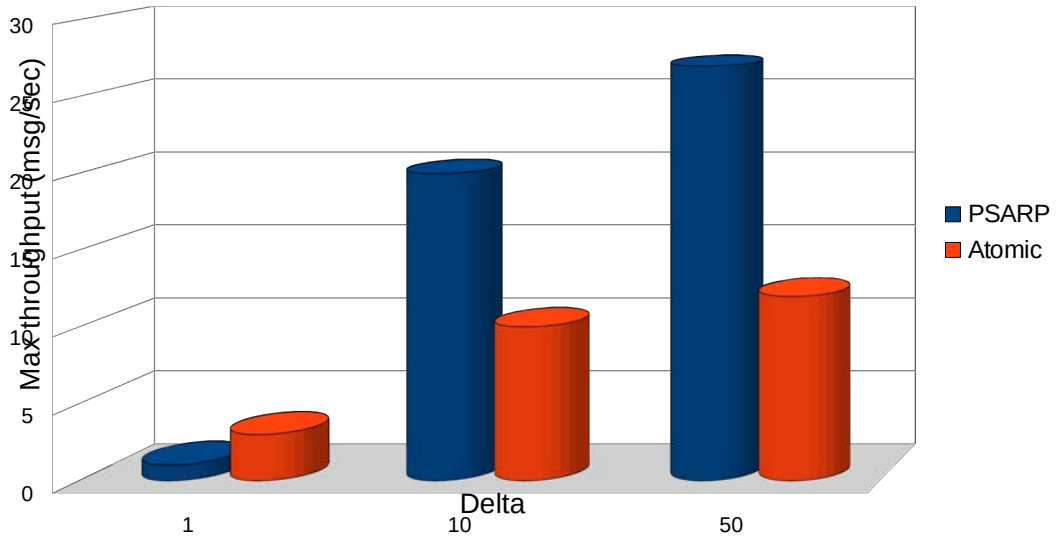


Grafico 7.5 – Limite di saturazione al variare di δ

La relazione tra i limiti di saturazione delle due soluzioni è simile a quella dello scenario precedente: per valori di δ prossimi ad 1 prevale il protocollo atomic broadcast, il quale presenta un limite più elevato rispetto a PSARP; con l'aumentare del valor medio δ il divario tra i due diminuisce e da un certo punto in poi è la nostra soluzione a prevalere. Per δ = 10 il limite di saturazione di PSARP è doppio rispetto ad atomic broadcast e per δ = 50 la differenza si porta quasi al 56%. Passiamo ora ad esaminare la latenza media di risposta del sistema. Dai grafici 7.6, 7.7 e 7.8 si evince chiaramente che anche per quanto riguarda le latenze medie di risposta del sistema l'andamento ricalca quello dello scenario 1: atomic broadcast ha latenze medie più basse per valori di δ prossimi ad 1, mentre al crescere del valor medio δ la situazione si inverte ed è PSARP a presentare tempi di risposta più bassi.

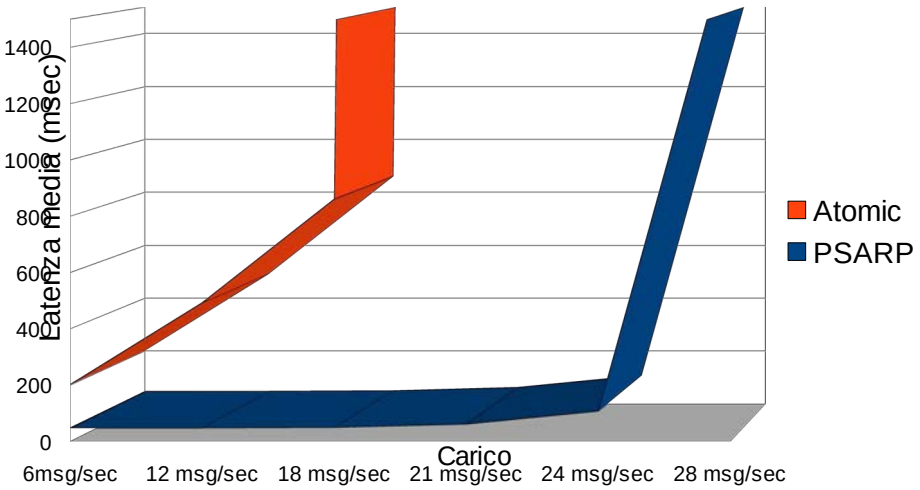


Grafico 7.6 – Latenza media per delta = 50

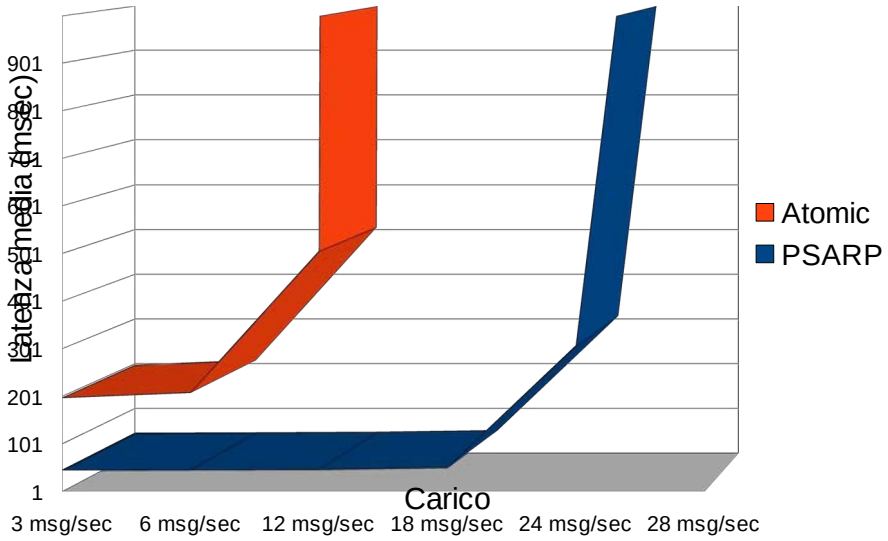


Grafico 7.7 – Latenza media per delta = 10

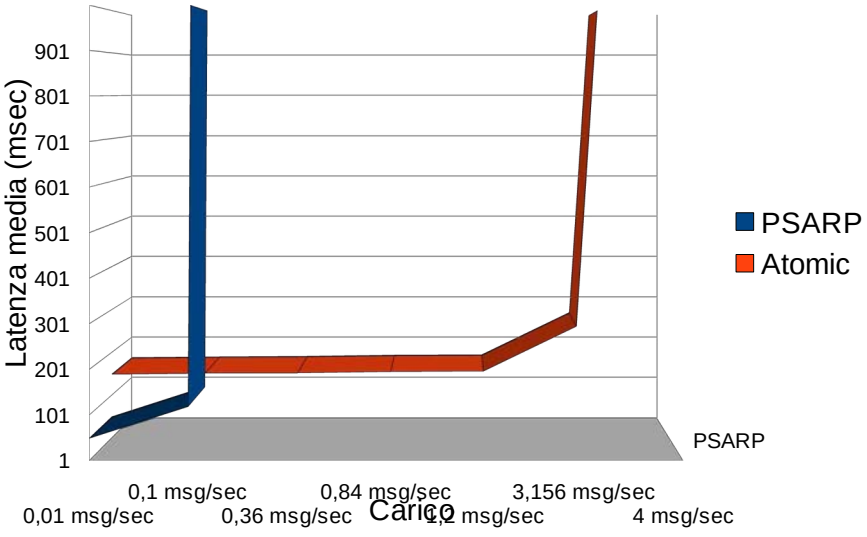


Grafico 7.8 – Latenza media per delta = 1

7.3.4 Considerazioni generali

Facciamo ora qualche considerazione generale riguardo le prestazioni dei due protocolli. Data la configurazione dell'hardware utilizzato per i test, il collo di bottiglia durante le run è rappresentato dalla macchina su cui si è allocato il LLM tier. Supponiamo di avere a disposizione un hardware che ci fornisca risorse di calcolo infinite, nel senso che qualsiasi sia la richiesta di risorse di da parte del LLM sia in grado di soddisfarla, in questo caso il collo di bottiglia si sposta sulla comunicazione tra LLM e HLM. O meglio il bottleneck del protocollo è rappresentato dalla latenza di comunicazione tra LLM e HLM, sperimentata durante i round per l'imposizione dello stato. Per quanto riguarda invece la soluzione basata su sequencer atomic broadcast il collo di bottiglia si sposta sulla rete, ovvero il bottleneck è rappresentato dalla banda di comunicazione disponibile.
Supponendo poi di cambiare il modello di sistema, assumendo che le repliche di LLM possano guastarsi e che solamente l'HLM sia sempre corretto, il protocollo PSARP presenta un vantaggio fondamentale rispetto a Sequencer Atomic Broadcast: quest'ultimo, in caso di guasto del sequencer, è soggetto a un forte degrado delle prestazioni, a causa ad esempio della procedura di failover necessaria per l'elezione di un nuovo sequencer, mentre PSARP non risente minimamente del guasto di una o più repliche, almeno dal punto di vista prestazionale.
Infine, considerando che il PSARP è stato ideato per operare in sistemi con valori di δ elevati, risulta essere una soluzione di gran lunga preferibile rispetto ai protocolli classici come Atomic Broadcast.

8. Conclusioni

La diffusione delle reti di sensori e delle tecnologie rfid anche al di fuori degli ambienti industriali e commerciali è ormai un dato di fatto. Queste cominciano ad essere adottate con successo anche in ambiti molto distanti da quelli per i quali originariamente erano state ideate, si pensi ad esempio alle recenti applicazioni nei settori sportivo e medico.
Un ruolo fondamentale nei sistemi rfid e nelle reti di sensori in generale è rivestito dal software che si occupa dell'elaborazione dei dati da queste prodotti. L'utilizzo di tali tecnologie per la gestione di processi critici introduce la necessità di progettare e realizzare sistemi che soddisfino stringenti requisiti di disponibilità del servizio. Di conseguenza, in tali ambiti, i problemi relativi alla fault tolerance assumono un'importanza primaria.
Il nostro lavoro si è concentrato proprio sullo studio di tali problematiche, nello specifico di quelle che riguardano la porzione software dei sistemi. Si sono studiate nel dettaglio soluzioni per applicazioni basate su tecnologia rfid, ma i risultati ottenuti sono del tutto generali e quindi valevoli per qualsiasi tipologia di sistema sensor based.
Nell'affrontare le problematiche relative alla tolleranza ai guasti sono state esaminate e valutate soluzioni basate su tecniche di replicazione software classiche. Successivamente, attraverso l'analisi e l'astrazione delle caratteristiche comuni a tutti i sistemi sensor based, sono state vagliate una serie di soluzioni alternative ed innovative. Questo lavoro ci ha portato all'ideazione di un nuovo protocollo di replicazione attiva, utilizzabile all'interno di sistemi di tipo multitier e particolarmente efficace in applicazioni in cui, come per quelle che interagiscono con reti di sensori, la quantità di eventi/ messaggi significativi generati in output è sensibilmente inferiore alla quantità di informazioni ricevute in input.
Il nuovo protocollo è stato chiamato Post Synchronized Active Replication Protocol (PSARP), ed è sostanzialmente basato sull'idea di sfruttare la silenziosità delle repliche software, modellate attraverso l'astrazione di Finite State Machine, per ritardare il più possibile la fase di sincronizzazione la quale non è realizzata attraverso tecniche di consenso distribuito ma sfrutta un secondo tier per la selezione dello stato adottato da tutte le repliche (replicazione a posteriori). Con il termine replica silenziosa, o più in generale con Silent Finite State Machine (SFSM), indichiamo un'entità che produce in output un

messaggio/evento solo dopo aver ricevuto un numero impredicibile, ma in media strettamente maggiore di uno, di messaggi in input.
I test realizzati sul nuovo protocollo hanno evidenziato un comportamento migliore di PSARP rispetto a tecniche classiche basate su Atomic Broadcast in sistemi in cui il rapporto δ tra il numero di messaggi in input ed il numero di eventi in output è maggiore di uno. Entrando un po' più nel dettaglio vediamo che se tale rapporto è prossimo ad uno allora soluzioni basate su atomic broadcast forniscono prestazioni, in termini di throughput massimo e latenza di risposta, migliori rispetto a PSARP, il quale risulta, per alti carichi, penalizzato dalle numerose operazioni di rollback alle quali è soggetto. Via via che il rapporto di cui sopra aumenta, le prestazioni del nuovo protocollo migliorano, per $\delta = 10$ ad esempio il throughput massimo di PSARP risulta essere doppio rispetto a quello fornito da soluzioni basate su atomic broadcast ed anche la latenza di risposta del primo risulta sensibilmente inferiore rispetto a quella delle seconde. In conclusione, in scenari in cui il rapporto δ non è prossimo ad uno il protocollo da noi ideato risulta preferibile alle soluzioni classiche.
Oltre a PSARP il nostro lavoro ci ha portato alla realizzazione di una piattaforma middleware fault tolerant per reti di sensori. Quest'ultima è stata progettata in modo tale da essere altamente estensibile. Mette infatti a disposizione del programmatore di applicazioni sensor based una serie di interfacce standard per:

- realizzare driver di interfacciamento tra il middleware e specifiche sorgenti hardware;
- modellare i dati provenienti da tali sorgenti e lo stato delle applicazioni;
- osservare e manipolare tale stato;
- regolare l'avanzamento delle FSM che modellano il comportamento delle applicazioni;
- introdurre schemi di replicazione ad-hoc.

Definisce inoltre uno schema generale per lo sviluppo di applicazioni e rappresenta un'ottima piattaforma per la valutazione di protocolli di replicazione innovativi. Non a caso tale middleware è stato utilizzato per la valutazione del protocollo PSARP.

Bibliografia e riferimenti

Testi e materiale didattico

[a] Rachid Guerraoui, Luìs Rodrigues, “Introduction to distribuited Algorithms” Springer-Verlag

[b] Rachid Guerraoui, Andrè Schiper, “Fault-Tolerance by Replication in Distribuited Systems”

[c] Roberto Baldoni, dipartimento di informatica e sitemistica “Antonio Ruberti”, Università degli studi di Roma “La Sapienza”- Materiale didatico del corso di “Sistemi distribuiti” anno accademico 2006-2007

[d] Paolo Romano, “Protocols for End-to-End Reliability in Multi-Tier Systems”, Tesi di dottorato, XIX Ciclo 2006, dipartimento di informatica e sitemistica “Antonio Ruberti”, Università degli studi di Roma “La Sapienza”

Papers

[1] Michael J. Fischer, Nancy A. Lynch and Michael S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," Journal of the ACM, April 1985, 32(2):374-382.

[2] K. Birman, A. Shiper, P. Stephenson. Lightweight Causal and Atomic Group Multicast. ACM Trans. on Computer Systems, 9(3):272-314, August 1991

[3] A. Shiper and A. Sandoz. Uniform Reliable Multicast in Virtually Synchronous Environment. In IEEE 13th Intl.Conf. Distribuited Computing Systems, pages 561-568, May 1993

[4] F. Pedone – A. Shiper. Optimistic Atomic Broadcast

[5] G. Gama, K. Nagaraja, R. Bianchini, R. Martin, W. Meira Jr., and T. Nguyen. State maintenance and its impact on the performability of multi-tiered internet services. In Proc. of the 23rd Symposium on Reliable Distributed Systems (SRDS), pages 146–158, 2004.

[6] M. Andreolini, M. Colajanni, and R. Morselli. Performance study of dispatching algorithms in multi-tier web architectures. SIGMETRICS Performance Evaluation Review, 30(2):10–20, 2002.

[7] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The state of the art in locally distributed Web-server systems. ACM Computer Surveys, 34(2): 263–311, 2002.

[8] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. M. Dias. Design and performance of a web server accelerator. In Proc. of the 18th Conference on Computer Communications (INFOCOM), pages 135–143, 1999.

[9] B. A. Shirazi, K. M. Kavi, and A. R. Hurson. Scheduling and Load Balancing in Parallel and Distributed Systems. IEEE Computer Society Press, 1995.

[10] S. Frølund and R. Guerraoui. Implementing e-Transactions with asynchronous replication. IEEE Transactions on Parallel and Distributed Systems, 12(2):133–146, feb 2001.

[11] S. Frølund and R. Guerraoui. e-Transactions: End-to-end reliability for three-tier architectures. IEEE Transaction on Software Engineering, 28(4):378–395, 2002.

[12] N. Budhirja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. In Mullender [20], pages 199–216.

[13] X. D´ fago, A. Schiper, and N. Sergent. Semi-passive replication. In Proc. of the 17th Symposium on Reliable Distributed Systems (SRDS'98), pages 43–50, West Lafayette, Indiana, Oct. 1998.

[14] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. IEEE Computer, 30(4):68–74, April 1997.

[15] D. Powell. Delta4: A generic architecture for dependable distributed computing. volume 1 of ESPRIT Research Reports. Springer Verlag, 1991.

[16] F. Schneider. Replication management using the state-machine approach. In Mullender [20], pages 169–198.

[17] K. Mazouni, B. Garbinato, and R. Guerraoui. Filtering duplicated invocations using symmetric proxies. In Proc. of 4th Int. Workshop on Object Orientation in Operating Systems (IWOOOS'95), Lund, Sweden, Aug. 1995.

[19] S. Poledna. Replica determinism in distributed real-time systems: A brief survey. Real-Time Systems, 6(3):289–316, May 1994.

[20] R. Jim´ nez-Peris, M. Pati˜ o-Mart´nez, and S. Arevalo. Deterministic Scheduling for Transactional Multithreaded Replicas. In Proc. of 19th IEEE Symposium on Reliable

Distributed Systems (SRDS'00), pages 164–173, Nuremberg, Germany, Oct. 2000.

[21] P. Narasimhan. Transparent Fault Tolerance for CORBA. PhD thesis, University of California, Santa Barbara, USA, September 1999.

[22] S.Pleisch, A Kupsys, A.Shiper. Preventing Orphan Request in the Context of replicated invocation

[23] M. Y. Luo and C. S. Yang. Constructing zero-loss web services. In Proc. of the 20th Conference on Computer Communications (INFO-COM), pages 1781–1790, 2001.

[24] D. K. Pradhan. Fault-Tolerant Computer System Design. Prentice Hall PTR, 2003.

[25] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Communications of the ACM, 43(2):225–267, 1996.

[26] S. Frølund and R. Guerraoui. A pragmatic implementation of e-Transactions. In Proc. of the 19th Symposium on Reliable Distributed Systems (SRDS), pages 186–195. IEEE Computer Society Press, 2000.

[27] J. Postel. Internet Protocol. request for Comments 791, Usc Inf. S. Inst., September 1981.

[28] J. Postel. Transmission Control Protocol. Request for Comments 793, Usc Inf. S. Inst., September 1981.

[29] H. Zimmermann. OSI Reference model – Thre ISO Model of Architectur for Open Systems Interconnection. *IEEE Transactions on Communications*, COM-28(4):425-432, April 1980

[30] M. Hayden. The Ensemble System. PhD thesis, Cornell University, Computer Science Department, 1998

[31] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu. Coyote: A system for constructiong e-grain configurable communication services. ACM Trans on Computer Systems, 16(4):321-366, November 1998.

[32] R. van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communications system. *Communications of the ACM*, 39(4):76-83, April 1996

Siti Web

[I] http://www.rfid-lab.it

[II] http://it.wikipedia.org

[III] http://www.rifidi.org

[IV] http://www.alientechnology.com

[V] http://appia.di.fc.ul.pt

[VI] http://ww.postgresql.org