ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – Scienza e Ingegneria DISI

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in
Calcolatori elettronici M

Soluzioni per conferire robustezza a un broker di informazioni semantiche per l'Internet of Things basato su OSGi

CANDIDATO
Cristiano Aguzzi

RELATORE:
Chiar.mo Prof. Tullio Salmon Cinotti

CORRELATORE
Prof. Alfredo D'Elia

Anno Accademico 20015/16

Sessione III

Sommario

Introduzione

Negli ultimi 50 anni Internet è passata da una piccola rete di ricerca, formata da pochi nodi, ad un'infrastruttura globale capace di connettere più di un milione di utenti. La progressiva miniaturizzazione e la riduzione di costi di produzione dei dispositivi elettronici, permette, tuttora, l'estensione della rete a una nuova dimensione: gli oggetti intelligenti (oggetti fisici aumentati da dispositivi elettronici integrabili). Essi coprono il divario tra il mondo fisico e dell'informazione, attingendo valore da questa simbiosi. Le problematiche riguardanti la cooperazione di questa tipologia di oggetti e la gestione dei dati da essi prodotti vengono affrontate da una recente disciplina informatica, l'internet of Things (IoT) [1].

In tale ambito, per gestire la complessità della interoperabilità, vengono spesso utilizzate tecnologie semantiche. Queste permettono un linguaggio comune tra un grande numero dispostivi eterogenei, grazie al quale è possibile condividere conoscenza ed agire coordinatamente verso un obbiettivo condiviso. Un insieme di oggetti intelligenti potrà quindi fornire un particolare servizio per un dato spazio, denominato Smart Space. In uno Smart Space perciò è cruciale fornire robustezza in caso di eventi avversi; primo fra tutti la perdita di comunicazione con uno dei dispositivi collegati. Poiché, come in qualsiasi applicazione distribuita, la perdita del collegamento con un oggetto intelligente consiste in una minaccia per la cooperazione degli agenti in gioco. Conseguentemente è necessario che le applicazioni sviluppate per questo ambito applicativo siano in grado sia di degradare in maniera dolce nei casi citati che di recuperare lo stato una volta che la comunicazione sia stata ripristinata.

Questa tesi magistrale indaga scenari di interazione fra molti agenti eterogenei, completando ed estendendo un studio precedente riguardo al confronto di due tecnologie per ambienti Smart: MQTT e SMART M3 [2]. In particolare i contributi maggiori apportati da questo lavoro di tesi sono:

- Introduzione del meccanismo di registrazione ultime volontà in una piattaforma semantica di gestione del contesto, per conferire robustezza ad applicazioni che la utilizzano.
- Estensione delle politiche di sottoscrizione al fine di garantire diversi livelli di robustezza
- Modifica dell'architettura interna del broker SMART M3 per aumentarne la modularità
- Introduzione di un nuovo paradigma di comunicazione intra moduli basato su eventi

Scenario di riferimento

Il lavoro nasce da un progetto sviluppato in *VTT Technical Research Centre of Finland* per un'applicazione d'agricoltura di precisione di nome Agri-Eagle. In particolare l'obbiettivo principale è lo sviluppo di un'architettura HW/SW che gestisca un insieme di droni come veicoli per la raccolta d'informazioni. Esse verranno elaborate con il fine di controllare attuatori installati su campi agricoli. Ognuno dei quali è coperto da un drone che rileva il livello di batteria di alcuni sensori e l'aridità del terreno. Raccolte queste informazioni, esse vengono inviate ad un sistema di gestione centrale che ha due mansioni principali: sostituzione di sensori non funzionanti o scarichi e ordinare ad erogatori idrici di irrigare determinate aree.

Nello scenario di riferimento possono verificarsi due eventi imprevisti che devono essere gestiti dall'applicativo software:

- Rottura di un sensore: i sensori possono rompersi e devono essere marcati come "da sostituire"
- Perdita segnale drone: anche il drone può rompersi o perdere il segnale con il nodo in cui è installato il sistema di gestione.

Il precedente studio ha mostrato come entrambe le tecnologie analizzate (SMART M3 e MQTT) siano in grado di soddisfare tutti i requisiti specificati ma, per quanto riguarda SMARTM3, ha evidenziato che gli eventi imprevisti richiedono uno sforzo implementativo maggiore. Ciò è dovuto alla mancanza di meccanismi di base lato server, quale la registrazione di ultime volontà (lastwill). Dal momento che SMART M3, tuttavia, garantisce a lungo termine un maggiore livello d'interoperabilità e una migliore espressività, è rilevante in questo lavoro di tesi fare il possibile per estenderne le funzionalità e gestire le casistiche descritte. Infatti sarà mostrato come le soluzioni proposte abbiano una maggiore precisione riguardo l'individuazione di rotture o perdite di connettività e impieghino meno sforzo nello sviluppo dell'applicativo in esame.

La tesi in breve

Il presente elaborato è strutturato come segue: nel capitolo *Tecnologie* si descriveranno brevemente le strumentazioni utilizzate; nel capitolo *Robustezza* verrà data la definizione di riferimento di robustezza ed esposto lo studio teorico svolto su tale tematica; il *Progetto* descrive brevemente i moduli software e le modifiche architetturali apportate; in fine nel *Impatto sviluppo applicativi* saranno esposti i benefici delle funzionalità introdotte riguardo alla facilità di sviluppo applicazioni

software. In *Appendice: Studio robustezza sottoscrizioni con singola socket* viene riportato uno studio parallelo inerente al livello di robustezza di sottoscrizioni con singola connessione.

Tecnologie

In questa sezione descriveremo brevemente i concetti fondamentali forniti dagli strumenti utilizzati. Ci limiteremo a darne un'introduzione finalizzata alla trattazione del problema in esame, in modo da avere un insieme di nozioni condivise con il lettore e di usarle per descrivere le soluzioni create. Dato lo scenario di rifermento abbiamo utilizzato il framework SMART M3 in particolare è stata studiata ed estesa una versione del broker semantico, basata su tecnologia OSGi. Per informazioni più approfondite riguardo a queste due tecnologie rimandiamo il lettore alla letteratura [3] [4].

Smart M3

La piattaforma Smart-M3 è stata progettata all'interno del programma Europeo SOFIA (Smart Object For Intelligent Application), un progetto Artemis della durata di tre anni comprendente diciannove partner di quattro nazioni europee diverse. Obiettivo di SOFIA è stato realizzare una piattaforma di interoperabilità semantica sfruttando il formalismo emergente del Web 3.0.

Smart M3 (Smart Spaces Multi vendor, Multi device, Multi domain) è la concretizzazione di questa speranza. Il suo scopo è fornire una piattaforma di scambio delle informazioni tramite la quale entità eterogenee possano operare. L'interoperabilità è raggiunta dalla condivisione di informazioni in un modello di dati semantico. Esso utilizza il formalismo ontologico ampiamente impiegato nel Web semantico [5]. Infatti secondo il World Wide Web Consortium (W3C):

"The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries" [6]

Cioè il web semantico fornisce una base comune che permette ai dati di essere riutilizzati e condivisi tra le varie applicazioni, aziende e comunità. Tale modello si basa sul concetto di relazione semantica descritta da Resource Description Framework (RDF) nel quale l'unità minima di informazione è rappresentata da una tripla: ***<soggetto> <predicato> <oggetto>***. Riutilizzando soggetti come argomenti di tipo oggetto è possibile creare un grafo nel quale le informazioni sono presentate in maniera semantica. Oltre a questo SMART M3 definisce anche strumenti per la comunicazione e il dialogo tra parti. Stabilisce, appunto, che il modello comunicativo sia quello di sottoscrizione e pubblicazione basata sul contenuto semantico. Questo significa che le entità che utilizzino questo middleware possono dialogare in maniera disaccoppiata specificando solo l'interesse per tipologie semantiche di dato e pubblicando tale informazione in un intermediario.

L'architettura proposta quindi è quella mostrata in Figura 1 dove sono mostrate le due parti costituenti di un'applicazione sviluppata sul framework in esame.

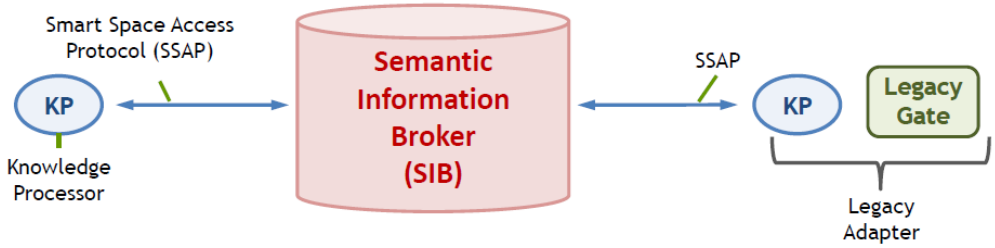

Figura 1 Architettura di un'applicazione SMART M3

Il paradigma è quello di un'applicazione distribuita con comunicazione asimmetrica cliente servitore. I clienti vengono chiamati Knowledge Processor (Manipolatori di conoscenza) e comunicano con il server attraverso il protocollo applicativo Smart Space Access Protocol (SSAP). Il server invece prende il nome di Semantic Information Broker (SIB), il quale è responsabile per la condivisione, gestione e memorizzazione dell'informazione. È provvisto di un motore per riconoscere modifiche di sotto grafi della base di conoscenza e fornire notifiche incrementali ai vari sottoscrittori. Questo permette di inviare solamente le aggiunte e le rimozioni al grafo avendo uso efficiente delle risorse di rete. Le operazioni principali offerte da questo componente software sono quelle di aggiornamento dati e la sottoscrizione. L'aggiornamento è una semplice push di un'informazione dal cliente al servitore, mentre la sottoscrizione è una query semantica persistente e remota [7]. Con tale funzionalità infatti i clienti specificano a quale parte dell'informazione contenuta nel broker sono interessati a ricevere notifiche. Ogni interazione tra KP passa perciò attraverso il broker e non sono ammesse, nel modello formale, dialoghi che lo scavalchino.

Una soluzione software sviluppata in questo ambiente sarà quindi formata da diversi tipi di KP che condividono un modello dei dati comune. I manipolatori di conoscenza possono essere divisi in tre tipi a seconda delle loro mansioni:

- ***Produttori***: Producono informazioni il loro accesso alla base di conoscenza è in scrittura, sostanzialmente si limitano a pubblicare dati.
- ***Attuatori***: Sono interessati solamente alla modifica della base di conoscenza, il loro accesso è in lettura. Il nome deriva dal campo

applicativo IoT nel quale solitamente questo di Knowledge Processor sono collegati ad un sistema fisico (attuano azioni nel mondo reale: bracci robotici, motori …)

- ***Aggregatori***: Trasformano informazione. Accedono al grafo sia in lettura che in scrittura. Tipicamente aggregano più informazioni semplice per fornire un dato d'insieme ad alto contenuto semantico.

Lo sviluppo di questi clienti può essere portato avanti grazie a delle API chiamata Knowledge Processor Interfaces disponibili per la maggior parte dei linguaggi industrialmente in uso (C,C#,Java, Javascript, Python e PHP).

L'SSAP implementatovi è basato su TCP con formato dei dati in XML. In particolare è formato dalle seguenti primitive:

- JOIN: Ogni KP per operare con il broker deve presentarsi attraverso questo messaggio. In questo sono contenute le informazioni identificative del cliente come l'id e il nome dello spazio intelligente a cui fa riferimento.
- LEAVE: È il messaggio duale con il quale il KP esprime la volontà di uscire dalla comunicazione con il broker.
- SUBSCRIBE: Il cliente richiede il servizio di sottoscrizione per una parte dell'informazione semantica contenuta nella SIB. In questo caso deve presentare un query con cui specifica a quali dati è interessato. Come risposta riceverà lo stato iniziale del grafo indicato da essa.
- UPDATE: Il cliente aggiorna il grafo. È composto da due sotto comandi:
    - INSERT: operazione di inserimento dati

    - DELETE: operazione di eliminazione dati
- UNSUBSCRIBE: il cliente comunica di non essere più interessato ad una certa informazione.
- QUERY: Il KP richiede una lettura dei dati contenuti nel broker.

In principio il protocollo prevedeva l'uso di triple RDF (estese con il concetto di wildcard) come argomenti dei comandi di QUERY, UPDATE, INSERT, DELETE e SUBSCRIBE. Recentemente tale protocollo è stato esteso con l'utilizzo di SPARQL, un nuovo standard per le interrogazioni di basi di conoscenza tipo RDF.

SPARQL è stato proposto nel 2008 dal W3C come linguaggio inspirato a SQL per la manipolazione di dati semantici [8] nel quale sono presenti due primitive query e update. La loro sintassi può essere consultata in [8] dove vengono presentate anche le possibilità espressive di questo linguaggio. Quindi con l'SSAP esteso è possibile sottoscriversi a dati e interrogarli tramite una query SPARQL ed aggiornare il grafo utilizzando la primitiva update.

Ricapitoliamo dicendo che in una applicazione SMART M3 i dati sono descritti da un'ontologia che è condivisa tra le varie entità che la compongono, i Knowledge Processor. Questa ontologia è fisicamente gestita da un middleware chiamato Semantic Information Broker che riceve richieste dai vari clienti tramite un protocollo specifico chiamato SSAP.

OSGi

L'Open Service Gateway initiative Alliance, è un'organizzazione fondata nel 1999 da Ericsson, IBM, Oracle e altri. Da questa cooperazione di aziende partono una serie di raccomandazioni che confluiscono in framework chiamato, appunto, OSGi. Tale direttive sono estensioni e

buone pratiche da utilizzare per lo sviluppo di applicazioni industriali basate su tecnologia Java. Quando si tratta di affrontare la complessità di problemi del mondo reale i quali richiedono applicativi software in grado di fornire vari servizi quale la scalabilità e affabilità, l'approccio da utilizzare è quello della decomposizione del problema. Prendendo inspirazione da altri ambiti ingegneristici, questo è possibile attraverso l'uso di moduli. Come la progettazione di un The Boeing 747-400 ha richiesto l'utilizzo di 75 disegni ingegneristici per descriverne le varie parti [4], allora anche lo sviluppo di un'applicazione complessa dovrà essere diviso in varie sotto parti ognuna della quale interagente con le altre.

L'utilizzo di moduli infatti permette di [9]:

- Dividere il lavoro: è possibile assegnare diversi individui o gruppi di lavoro a progettare diversi moduli. Le persone assegnate ad un modulo avranno una comprensione completa di esso ma una visione parziale degli altri.
- Astrarre: è pensabile immaginare il progetto come composizione astratta di parti, senza specificarne ogni dettaglio.
- Riutilizzare: Data la quantità di lavoro necessaria a costruire anche una minima parte del piano di lavoro, sarebbe opportuno riutilizzare queste risorse in progetti che abbiano simili specifiche. Questo diventa semplice se è stata utilizzata una divisione delle responsabilità nei vari componenti.
- Facilità di mantenimento e riparazione: Un design modulare permette di contenere gli sforzi di riparazione e di mantenimento poiché si possono concentrare solo su singole parti invece che in tutto il sistema.

D'altra parte Java non supporta la modularità in maniera utile. Infatti il meccanismo fornito è quello dei Jar ma è noto per creare problemi. In letterature è appunto definito come "Jar Hell" [4] [10] per la difficoltà di gestione delle dipendenze e versioni, per i problemi a runtime riguardo al caricamento lineare di tutte le classi in un unico class path ed infine per l'impossibilità di nascondere le informazioni in maniera chiara. Queste difficoltà saranno superate almeno in parte dalla nuova versione di Java la numero 9 ma il suo rilascio è previsto per i primi mesi del 2018 e si stanno già investigando la sua compatibilità con OSGi [11] [12] .

Con il fine di superare tali problematiche nasce il framework OSGi, unico obbiettivo del quale è quello di proporsi come un semplice sistema a moduli per la macchina virtuale Java. Esso definisce come questi vengano costruiti e come essi interagiscano tra loro a runtime. L'idea è quella di eliminare tutti i problemi del classpath globale e piatto di Java imponendo che ogni modulo ne ha abbia uno privato. Inoltre impone delle regole a come vengono condivise classi tra di essi, attraverso il meccanismo degli export e import espliciti.

Un modulo OSGi, chiamato in gergo *bundle*, è quindi un semplice JAR decorato da informazioni aggiuntive quali:

- Un nome univoco, utilizzando per l'identificazione del modulo
- Una versione
- Una lista di import e export. Gli import identificano la lista di package da cui il *bundle* dipende, mentre gli export definiscono quali vengano esposti al pubblico. Questa tecnica permette efficacemente di nascondere l'implementazione all'interno del modulo e esporre solo le API necessarie per richiedere i suoi servigi.

- Opzionalmente la versione minima di Java supportata dal *bundle*
- Informazioni miscellanee *human-readable* come: copyright, contatti, distributore etc.

Questi dati vengono descritti nel file MANIFEST.MF all'interno dell'archivio contente il codice del modulo. Utilizzando tale espediente si ottiene la compatibilità con applicazioni Java semplici poiché i campi aggiuntivi vengono normalmente ignorati dalla JVM.

Altre tecnologie offrono questo tipo di servizio come ad esempio Maven [13], Ivy [14] e Gradle [15]. Tali strumenti però consentono l'utilizzo del formalismo a moduli solo durante il compile time, essendo appunto, utensili software impiegati nella compilazione di applicazioni Java. OSGi al contrario è soprattutto un sistema a moduli dinamico il che significa che il modulo è un'entità manipolabile e sfruttabile runtime. Il modello di interazione proposto è quello in Figura 2. Gli utilizzatori dei servigi di un modulo, chiamati clienti, ne devono richiedere l'accesso ad un'entità terza definita Service Broker. Esso contiene le istanze dei moduli, i quali si possono registrare con la primitiva *register* esponendo un contratto di servizio (interfaccia). A regime quindi si instaura un collegamento dinamico tra il richiedente del servizio e il modulo richiesto, relazione simile al modello cliente e servitore.

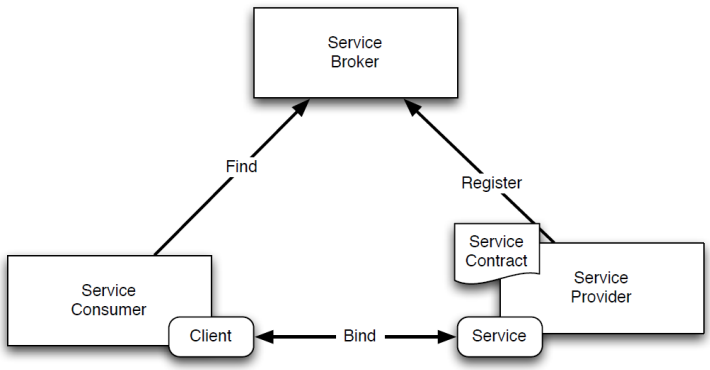

Figura 2 Modello di interazione tra moduli

Analogamente allo scenario distribuito il collegamento tra cliente e servitore è volatile e può rompersi qualora il modulo che fornisce il servizio venga aggiornato o eliminato [16]. In tal caso tramite un meccanismo di notifiche il consumatore può:

- sospendere parte della sua funzionalità che ne richiedeva l'uso
- richiedere a sua volta l'eliminazione dall'elenco dei moduli attivi
- Tentare di richiedere di nuovo il servizio, magari perché è stato aggiornato con uno più recente

Infatti il numero dei bundle durante l'esecuzione di un'applicazione non è statico (come negli strumenti di build) ma può variare nei casi in cui si voglia aggiornare incrementalmente il software o aggiungere nuove funzionalità a runtime. Per questo motivo i componenti di un applicativo OSGi seguono un preciso ciclo di vita mostrato in Figura 3. Si può notare due categoria di azioni: esplicite e implicite. Quelle implicite sono attuate dal framework in maniera automatica mentre le prime vengono impartite dall'utente ed è possibile sia eseguire per via programmatica che tramite l'utilizzo di una console preposta. L'automa a stati mostra come ogni modulo all'inizio si trovi in uno stato indefinito fin quando tramite l'azione esplicita di *install* viene eseguita. A questo punto si transita nello stato INSTALLED dove inizia il processo di risoluzione di dipendenze. La transizione al prossimo stato di RESOLVED avviene infatti dopo che il framework ha verificato il soddisfacimento delle seguenti condizioni:

- L'ambiente di esecuzione Java è uguale o maggiore a quello specificato nel MANIFEST.MF

- Le dipendenze di import sono fornite con la versione compresa nel range richiesto da moduli in stato RESOLVED o che possono passare in RESOLVED assieme al bundle in esame

A questo punto OSGi utilizzando la primitiva *resolve* porta l'automa allo stato considerato. In RESOLVED il modulo è pronto per essere eseguito e questa attivazione avviene con la primitiva esplicita *start*.

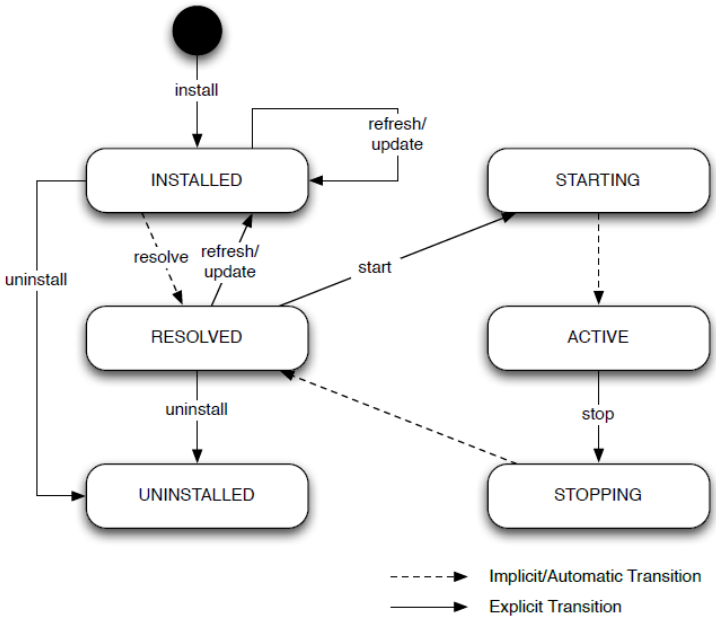

Figura 3 Ciclo di vita di un modulo OSGi

La macchina a stati passerà, a fronte di questa azione, a STRATING dove verrà data l'opportunità al bundle di inizializzare lo stato e metterlo al corrente dell'imminente attivazione. L'attivazione è automatica e si passa allo stato ACTIVE. In questa situazione il modulo è pronto per essere utilizzato da altri bundle e può fornire i suoi servizi. A fronte dell'azione di *stop* invece si passa allo stato STOPPING nel quale si avrà l'opportunità di liberare risorse e pulire lo stato inizializzato in STRATING, passando quindi nuovamente a RESOLVED. Come si può notare in questo stato è anche possibile agire sulla manutenzione del modulo come la disinstallazione e l'aggiornamento. Tali operazioni sono

però vietate durante la fase attiva e quindi saranno disponibili solo previa chiamata alla primitiva *stop*.

Descriviamo infine come utilizzando queste particolarità è stato possibile progettare un broker semantico per la piattaforma SMART M3.

Architettura OSGi per una SIB

Come introdotto il broker semantico è un'applicazione server che ha il compito di processare messaggi formattati secondo il protocollo SSAP. In questa implementazione è stata sfruttata la modularità e l'adattività offerta da OSGi per permettere una migliore estendibilità del software è sfruttare la possibilità di manutenzione moduli. Nonostante sia passato sotto diverse rivisitazioni ha buone performance ed è ampiamente utilizzato nello sviluppo di applicazioni di ricerca. In Figura 4 sono mostrati i moduli implementati e le relazioni che li legano.

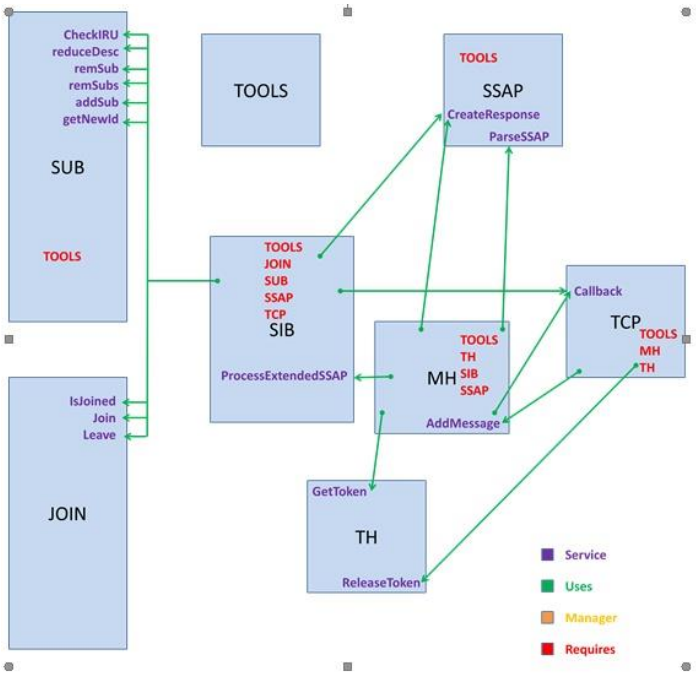

Figura 4 Architettura OSGi del broker semantico

Cerchiamo di descrivere il loro comportamento seguendo il percorso che fa un ipotetico messaggio di xml in SSAP. Per prima cosa viene creata una connessione tra cliente e servitore dal bundle

sofia.sib.gates.tcp.TcpIpService(in figura TCP) che, per indentificarla in seguito, richiede un token a *sofia.sib.tokenHandler.tokenHandlerService* (TH). A questo punto il modulo TCP legge completamente il messaggio testuale e lo consegna a *sofia.sib.messageHandler.MessageHandlerService* (MH) tramite il metodo *AddMessage*. Il metodo aggiunge i messaggi in una coda che verrà consumata da un thread contenuto all'interno del modulo MH. Il processo di seguito farà le seguenti operazioni:

1. Decodifica il messaggio attraverso il bundle *sofia.sib.ssap.SsapService* (SSAP)
2. Processamento del messaggio da parte del *sofia.sib.store.StoreService*(SIB) nel quale è contenuto anche il grafo semantico.
3. Invia il risultato della precedente operazione, tramite il metodo *callback*, al bundle TCP, il quale provvedere a mandarla al cliente.

I comandi vengono codificati in un'unica tipologia di oggetto chiamata e_ssap. In esso sono contenute le informazioni riguardo alla connessione, tipologia di query, tipo di richiesta, id del nodo mandante, smart space e contenuto del comando vero e proprio. Questi vengono consegnati al bundle SIB, come descritto prima, dove avviene la loro vera e propria esecuzione. L'attuazione di alcuni di essi è delegata ad altri moduli, precisamente i comandi di JOIN e SUBSCRIBE sono svolti parzialmente dai rispettivi moduli *sofia.sib.joined.JoineService*(JOIN) e *sofia.sib.subscriptions.SubscritpionsService* (SUB). Tutti gli altri, che utilizza direttamente il grafo semantico, vengono processati dal modulo stesso e poi la risposta viene restituita tramite il meccanismo di chiamata a servizio al modulo MH.

Lo store ha il compito anche di generare le notifiche causate da modifiche dei dati ai quali certi KP si son sottoscritti. Infatti, durate i comandi di trasformazione grafo, chiede al bundle SUB se questi sollecitino qualche sottoscrizione attiva. In tal caso crea un oggetto contenente le informazioni ottenute da SUB, richiede la sua trasformazione in messaggio a SSAP e lo consegna a TCP. Questo, utilizzando l'informazione contenuta riguardo alla connessione che ha generato quella particolare sottoscrizione, invia al cliente l'evento. L'evento viene sempre fornito da SUB come informazione incrementale sullo stato inziale fornito all'atto della sottoscrizione. Tale comportamento permette un consumo minore di banda e risorse lato KPI.

La comprensione dell'architettura è stata fondamentale per la fase di progetto nella quale è stato necessario apportare delle modifiche al fine di raggiungere gli obbiettivi di questo lavoro.

Robustezza

La proprietà di robustezza in un sistema software è una definizione ampia. In generale si definisce che un componente software è robusto se produce le conseguenze per il quale è stato progettato nell'ambiente di competenza, senza effetti indesiderati [17]. Nel caso in esame quindi vorremo che la totalità delle parti in gioco (SIB e KP) si comportino come definito in fase di progetto. Poiché non stiamo parlando di una singola applicazione le conseguenze da rispettare potrebbero essere infinite. Per affrontare il problema definiremo prima quale aspetto specifico di robustezza vogliamo garantire poi in quale ambiente le considereremo soddisfatte. Come suggerito da [18] gli attributi che un sistema robusto deve rispettare sono:

- Availability – prontezza del servizio corretto
- Reliability – continuità del corretto servizio
- Safety – Assenza di conseguenze catastrofiche per l'utente e per l'ambiente
- Integrity – Assenza di alterazioni improprie del sistema
- Maintainability – abilità per un processo di resistere ad eventi di manutenzione.

Il nostro scopo è quello di fornire continuità del servizio corretto e safety per il servizio di pubblicazione e invio notifiche. Quindi vorremo garantire che sotto particolari condizioni la distribuzione degli eventi tra SIB e KP avvenga in maniera corretta. Nell'ambito applicativo d'interesse le applicazioni eseguono in un ambiente in cui la connessione tra le parti non può considerarsi permanente. In particolare sono frequenti transitori cali di banda o delle vere e proprie perdite di collegamento. Ad esempio in applicazioni mobili è frequente l'entrata in zone prive di segnale o nelle quali è talmente degradato che la ricezione di dati è impossibilitata. Il fine ultimo di questo trattato è proprio quello di conferire continuità di servizio e safety in caso di connettività limitata e intermittente. D'ora in poi ci riferiremo a queste proprietà con il termine generico di robustezza, intendendo che una funzionalità è robusta se garantisce la continuità di servizio sotto le suddette condizioni. Inoltre considereremo che un meccanismo conferisca tale attributo se dà la possibilità a politiche di alto livello di essere a loro volta robuste.

L'importanza di ottenere questa proprietà per un'implementazione del framework SMART M3 è avvalorata sia da studi precedenti sia dall'esperienza sul campo di sviluppatori che l'hanno utilizzato. Infatti in

[19] la **Reliability** è annoverata tra i 16 principi fondamentali che una piattaforma di interoperabilità deve rispettare. Inoltre indica che debba essere misurata sia a tempo di sviluppo che a runtime. Per tale motivo sono state fatte delle valutazioni empiriche sul comportamento di applicazioni sviluppate per SMART M3, utilizzando la SIB OSGi, nelle condizioni in esame. I transitori di connessione hanno compromesso la continuità di servizio in quasi la totalità dei casi. In particolare la funzionalità più compromessa è stata quella di sottoscrizione, la quale, dopo il ripristino della connessione con il broker, non consegnava più notifiche, impendendo al cliente di continuare il suo ruolo nell'applicazione distribuita. Perciò per ottenere robustezza nella SIB OSGi è necessario studiare dei meccanismi attraverso i quali garantire la continuità di servizio delle sottoscrizioni e delle notifiche.

Infine, studiando il problema proposto in [2] e descritto in *Scenario di riferimento*, è emerso che l'utilizzo del paradigma di lastwill è in grado di garantire il corretto comportamento dell'applicativo Agri-Eagle anche in condizioni critiche di scarsa connettività. Da qui nasce la necessità di studiare tale funzionalità inserita in un contesto semantico, al fine di fornire uno strumento robusto ai software basati sulla piattaforma di interoperabilità SMART M3.

Lastwill

Il lastwill è un contratto tra cliente e servitore nel quale il firmatario (servitore) garantisce l'esecuzione di certe volontà. Tali volontà dovranno essere attuate alla morte del nodo che le ha specificate. Questa eventualità, in un'applicazione distribuita, coincide con la scomparsa del nodo dalla rete di connessione in cui sussiste. Ragionando sulle proprietà che questo meccanismo conferisce ad applicativi che ne usufruiscano, possiamo considerare quella di continuità del servizio con

le condizioni a contorno precedentemente specificate. Infatti, nonostante l'evento avverso di caduta di connessione, l'espressività data da questa funzionalità permette a clienti non più connessi di fornire comunque informazioni critiche ad altri componenti software. Quindi essa, in generale, è robusta poiché continua a fornire dei servizi, seppur limitati, a fronte dell'evento negativo di disconnessione.

L'esempio è dato proprio dal problema specificato in Agri-Eagle, nel quale era necessario fornire il servizio di informazione dello stato di irrigazione del campo, nonostante fossero possibili rotture del drone e/o dei sensori. Nella soluzione con l'uso di lastwill (MQTT) è stato possibile stabilire in maniera affidabile se il drone fosse ancora operativo e se lo stato corrente del campo agricolo fosse correttamente aggiornato. Nella versione senza (SMART M3) è stato necessario progettare una serie di controlli a time out per ottenere lo stesso livello di servizio. Secondo il nostro studio la mancanza della possibilità di esprimere le ultime volontà, porta non solo minore facilità di sviluppo ma, più gravemente, soluzioni non affidabili.

Infatti, considerata l'implementazione SMART M3, si osserva che viene sfruttato, come informazione sulla funzionalità del drone, il tempo di inserimento dell'ultimo dato. Se questo è maggiore di un certo time out allora l'applicazione considera non attivo il drone. Il limite di tempo è fisso e quindi soffre di vari problemi tra cui la mancata reattività alla causa dell'errore. Se ad esempio per aver un numero minore di falsi positivi lo impostassimo a valori elevati non avremo tempestivamente l'avviso sulla rottura del drone e perciò la necessità di andare a sostituirlo. Al contrario avremmo spesso avvisi fuorvianti che innescherebbero risorse umane necessarie alla verifica dello stato del dispositivo automatico di volo. Potremmo allora adottare un time out

variabile. Il calcolo di tale valore sarebbe ottenibile attraverso una media mobile dei tempi tra l'invio di un dato e l'altro. Anche questa soluzione non garantisce risultati soddisfacenti. Infatti, se i tempi di percorrenza tra un sensore ed un altro sono molto incostanti, la media potrebbe comunque essere superata sebbene il drone si stia solo spostando al prossimo dispositivo di raccolta dati. In generale la sola informazione che un cliente non stia pubblicando alcun ché, non può essere utilizzata per stabilire in modo sicuro che esso non sia più attivo. A maggior ragione in casistiche in cui non può essere garantito che il produttore pubblichi dati in maniera periodica. Ad esempio quando debba fornire informazioni sporadiche a mo' di eventi (alarmi, sensori presenza etc.). Il lastwill è l'unico strumento che ci permette di fornire tale notizia al livello applicativo. Conseguentemente l'applicazione potrà utilizzarla al fine di attuare politiche robuste per raggiungere scopi ad alto livello.

Un altro esempio del perché la funzionalità trattata può essere utilizzata per garantire robustezza, è quello del caso in cui si voglia garantire fault tolerance (una declinazione di robustezza) attraverso risorse replicate. Supponiamo di avere due dispositivi attuatori, entrambi aprono la stessa porta e son duplicati. La duplicazione permette di controllare comunque la porta a seguito della rottura di uno dei due. Infatti si potrebbe pensare che il sistema indichi la sostituzione del dispositivo malfunzionante, in modo da averne sempre uno di scorta (Figura 5). Ciò significa voler offrire un servizio di apertura/chiusura passaggio che sia robusto a casi di rottura. Otteniamo tale livello grazie al lastwill:

- All'inizializzazione entrambi inseriscono nel sistema il loro stato. Sarà inserito stato attivo per entrambi.

- Essi comunicheranno come ultima volontà di cambiare tale stato in disattivo/rotto.
- L'applicazione sceglierà in maniera casuale uno dei due.
- Se sul dispositivo usato avverrà un guasto non previsto (crash, perdita di connessione, rottura etc.) scatterà l'esecuzione delle sue ultime volontà.
- In tal caso l'applicazione sottoscritta al cambiamento di stato, potrà ridirigere le prossime richieste di apertura all'attuatore rimasto attivo e notificare la necessità di manutenzione su quello rotto.

L'applicazione così definita è in grado di fornire continuamente il servizio di attuazione porta grazie alla duplicazione di risorse e lastwill (Per ulteriori dettagli su questo caso applicativo si rimanda a *Lastwill adattativo*).

Infine un ultimo caso d'uso fa parte della famiglia di problemi applicativi nei quali la disconnessione di un nodo è un evento naturale e previsto. Un'istanza del problema potrebbe essere, riprendendo l'idea dell'uso dei droni, un monitor nel quale venga visualizzato lo stato di una flotta di dispositivi per l'analisi aerea di un campo. Lo stato può essere in "stiva" oppure in "volo". Il passaggio tra uno stato e l'altro avviene quando un drone lascia la stiva e, per questo, si trova scollegato dal segnale Wi-Fi del deposito. Utilizzando il lastwill potremmo ottenere il comportamento desiderato. Infatti lasciata la zona coperta dalle onde radio il broker rileverebbe l'assenza del dispositivo che ha espresso le ultime volontà, allora le eseguirebbe, cambiando in sua vece lo stato da "stiva" a "volo". Una costruzione alternativa potrebbe essere quella di aggiornare manualmente lo stato dal drone una volta che è messo in

volo. L'informazione però sarebbe troppo proattiva e in certi casi potrebbe portare a risultati erronei. Con il lastwill, d'altra parte, la notifica arriva puntualmente nel momento di vero allontanamento del drone. Notiamo come in questo scenario le ultime volontà sono sfruttate per ottenere un requisito funzionale non per gestire qualche particolare eventualità che minacciasse la continuità del servizio. Possiamo però parlare comunque di robustezza, poiché il suo utilizzo conferisce in un certo senso la prontezza di servizio corretto (*Availability*), attributo tra quelli citati di un sistema robusto.

Concludiamo affermando che abbiamo dimostrato di come il meccanismo di dichiarazione ultime volontà sia robusto. Quindi per tale proprietà è stato scelto come funzionalità sperimentale da inserire nel framework SMART M3, utilizzando come banco di test il broker semantico OSGi.

Lastwill adattativo

In questa sezione vedremo le differenze teoriche tra l'espressività conferita da un lastwill basato su MQTT e quella data da uno implementato tramite SPARQL.

Per prima cosa vediamo quali azioni permette di dichiarare in un messaggio di ultima volontà, MQTT. Il messaggio di testamento ha il formato di una normale operazione di pubblicazione. Quindi è ragionevole considerare che esso avrà la stessa efficacia di una normale operazione di update. In questa tecnologia l'aggiornamento dei dati avviene tramite modifiche a variabili contrassegnate da determinate stringhe (topic) [20]. Ogni cambiamento può modificare una variabile alla volta e deve sempre inserire un nuovo valore. Perciò non è possibile esprimere se una certa proprietà diventi obsoleta, senza aggiungere eccezioni nel modello dei dati. Chiariamo con un esempio: consideriamo

la stringa "***persona1/haMacchina"***. In questo topic verrà pubblicato il modello di auto posseduto da ***persona1***. Dovremmo prevedere però che un utente possa vendere la sua macchina e decidere di utilizzare i mezzi pubblici perché si è trasferito in città. Per far ciò, in MQTT, è opportuno progettare un valore speciale che identifica questo particolare caso. Conseguentemente, accertarsi che tutti i clienti sottoscritti siano a conoscenza di tale informazione. Questa problematica limita l'interoperabilità e per di più è concettualmente simile all'utilizzo di valori NULL in programmazione ad oggetti, l'impiego dei quali generalmente è altamente sconsigliabile [21]. Ulteriormente la limitazione di modificare un unico campo per messaggio riduce le possibilità di intervento sul modello dei dati in caso di rilevazione "morte" del cliente. Normalmente non sarebbe molto limitante, trascurando casi critici di sincronizzazione, ma, lo è nel caso del testamento, in cui è ammesso al massimo un messaggio di aggiornamento.

In SMART M3 potremmo esprimere le azioni da eseguire come ultime volontà tramite l'utilizzo di un SPARQL UPDATE. Come sappiamo questa operazione ha una potenza espressiva ben maggiore rispetto al semplice aggiornamento di un valore. Infatti permette la verifica di condizioni e la modifica di certi valori che le rispettino. Per di più fornisce anche la possibilità di rimozioni di valori dal grafo semantico, supportando di default il concetto di proprietà obsolete.

Vediamo ora una situazione esemplificativa delle differenti possibilità offerte dai due costrutti. Riprendiamo il caso noto di porta con due attuatori e aggiungiamo un cliente che invia il comando di apertura alla porta, pubblicando una informazione nel broker (Figura 5).

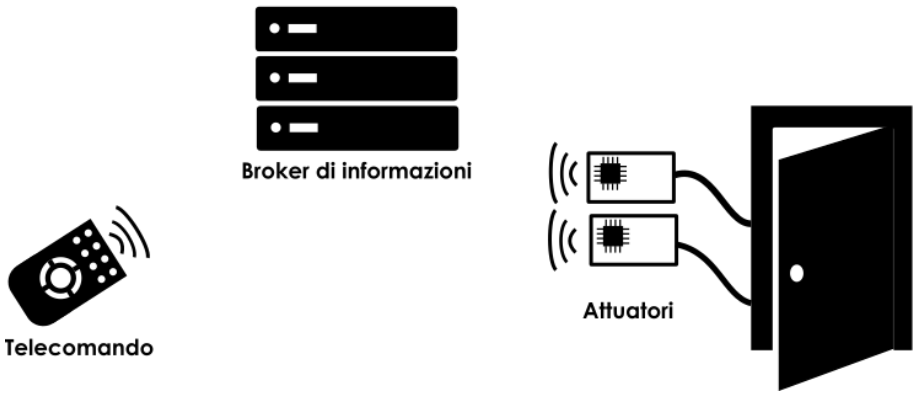

Figura 5 Architettura applicazione con doppio attuatore

Nel caso MQTT modelleremo il problema nel modo seguente:

- Il cliente che richiede l'apertura della porta pubblica "Aperta" nel topic ***/porta1/stato***
- I due attuatori sono sottoscritti in ***/porta1/stato*** e ad ***/porta1/gestore***. Il secondo campo identifica quale dei dispositivi può aprire/chiudere la porta1***.***
- Inoltre specificano come lastwill la pubblicazione al topic ***/gestoreN/stato*** di "Rotto". Con N che varia a seconda dell'identificativo del gestore.
- Il produttore si sottoscrive ai vari topic ***/gestoreN/stato*** e aggiorna conseguentemente il valore ***/porta1/gestore.*** Ricordiamo che questo compito potrebbe essere conseguito anche da un altro cliente esterno. Tuttavia ciò porterebbe ad altre problematiche che vedremo a breve nella discussione.

Studiamo ora il comportamento della soluzione nel caso critico: uno dei dispositivi di controllo si rompe. Vengono eseguite le seguenti operazioni in sequenza:

1. Le sue ultime volontà sono eseguite dal broker MQTT

2. Il produttore riceve la notifica e aggiorna ***/porta1/gestore***
3. L'altro attuatore riceve la notifica che ora lui può gestire la porta**.**
4. Il telecomando continua ad inviare comandi a ***/porta1/stato*** ottenendo l'esecuzione dal controllore rimasto.

La continuità del servizio è stata preservata e quindi in prima battuta possiamo affermare che la nostra applicazione è robusta. Cerchiamo però di valutare la soluzione nella sua interezza. Per garantire il corretto funzionamento della politica di fault tolerance dei due attuatori è stato necessario spostare la responsabilità di assegnazione dispositivo al cliente produttore di comandi. Questo non solo espone un dettaglio implementavo all'esterno ma rende la soluzione poco interoperabile. IoT è un ambiente dinamico, dove più dispositivi di diversi venditori possono collaborare per eseguire un obbiettivo comune. Nel caso in cui un nuovo telecomando volesse utilizzare la porta, dovrebbe coordinarsi con quello previsto dell'applicazione e se questo non fosse presente si deve far carico della gestione degli attuatori. Maggiormente, nel peggior scenario, il nuovo telecomando è stato sviluppato precedentemente all'introduzione della ridondanza e quindi è calibrato per pubblicare in solo ***/porta1/stato*** poiché, immaginiamo, è un topic standard/legacy usato da tutte le porte.

Un'altra opzione potrebbe essere quella di spostare la logica di gestione dispositivi ridondanti in un altro cliente (Figura 6). Anche in questa situazione individuiamo delle problematiche consistenti che inficiano sulla continuità del servizio. Infatti se il nuovo cliente perde la connettività oppure si spegne in modo inavvertito ci troveremo in un caso ricorsivo, cioè sarebbe necessario un altro cliente che gestisca

questa eventualità e poi un altro che controlli quello di prima e così via. Con questa scelta non possiamo in generale garantire la robustezza dell'applicazione. Infatti potremmo soddisfarla solo se è possibile stabilire con sicurezza che il gestore risorse sia sempre attivo (esempio: si trova sullo stesso nodo del broker).

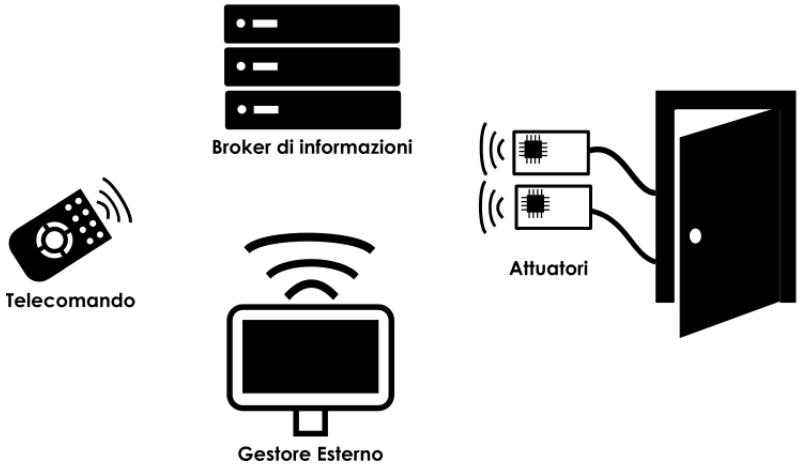

Figura 6 Architettura alternativa

L'ultima opzione è quella di dare l'onere della gestione dei controllori ai dispositivi stessi, cioè di auto gestirsi. Brevemente il nuovo comportamento sarebbe quello in cui i due attuatori si sottoscrivono reciprocamente a ***/gestoreN/stato*** prendendo il comando se l'altro non è più funzionante. Seppure molto interoperabile anche in questo caso non siamo esenti da situazioni di rischio. Per l'appunto, se sfortunatamente tutti i dispositivi di controllo dell'entrata vengono a non funzionare improvvisamente (esempio: caduta di tensione), lo stato dei dati non è correttamente aggiornato. Il telecomando potrebbe essere convinto che il gestore della porta sia attivo, leggendo la proprietà ***/porta1/gestore***, e inviare comandi ad esso, i quali però non sortirebbero alcun effetto.

In tutti gli esempi trattati abbiamo evidenziato situazioni in cui l'applicazione sviluppata non è conforme a proprietà importanti come

l'interoperabilità e la robustezza, vediamo ora come si comporterebbe lo stesso applicativo se fosse sviluppato con SMART M3 e lastwill.

Il problema verrebbe modellato come segue, utilizzeremo i concetti quali produttore per il telecomando e attuatore per i due controllori della porta:

- Il produttore inserisce i comandi attraverso aggiornamenti del grafo che ha la tripla ***<url/portaID> <url/haStato> <"Aperto">***
- I due attuatori si sottoscrivono con una query simile a Codice 1 grazie alla quale ricevono informazioni circa il comando da eseguire e chi è correntemente il gestore in carica. L'informazione sul gestore è **protetta** e solo i vari dispositivi di attuazione possono leggerla.
- Sempre gli attuatori specificano le loro ultime volontà come mostrato in Codice 2

```
PREFIX dp: <http://porte/>
SELECT ?gestore ?status
WHERE {
       dp:porta0 dp:haStato  ?status .
       dp:porta0 dp:haGestore ?gestore
}
```

Codice 1 Query di sottoscrizione di uno dei due attuatori

Come nel caso precedente esponiamo il comportamento della soluzione nel caso critico, uno dei dispositivi di controllo si rompe:

1. Vengono eseguite le sue ultime volontà. Grazie alla semantica di SPARQL UPDATE ora in ***<dp:porta0> <dp:haGestore> <atp:gestore1>*** c'è il valore esatto di chi deve attuare i comandi di apertura/chiusura.
2. L'altro attuatore rimasto riceve la notifica che ora è lui il gestore.

3. Il telecomando continua ad inviare comandi a ***<url/portaID> <url/haStato> <"Aperto">***

   ottenendo l'esecuzione dal controllore rimasto.

```
PREFIX dp: <http://porte/>
PREFIX atp: <http://porte/attuatori>
DELETE {
       dp:porta0 dp:haGestore ?vecchioGestore.
       atp:gestore0 atp:haStato "Online"
}
INSERT {
       atp:gestore0 atp:haStato "Offline".
       dp:porta0 dp:haGestore ?nuovoGestore
}
WHERE {
       SELECT ?nuovoGestore {
              atp:gestore0 atp:haStato "Online".
              OPTIONAL {
                     ?findGestore atp:haStato "Online".
              }.
              BIND(IF(!bound(?findGestore),"GESTORIROTTI",?findGestore) AS ?nuovoGestore))
       } LIMIT 1
}
```

Codice 2 Lastwill di uno dei gestori (gestoreo)

Per prima cosa notiamo che non è necessario in alcun modo aggiungere altri clienti gestori o incaricare il produttore di tener coerente lo stato. Ne risulta un'architettura più snella e con una chiara separazione delle responsabilità. Inoltre i dettagli implementativi sono complementarmente nascosti agli utenti del servizio "apertura/chiusura porta", il quale può avere o non avere meccanismi robusti in modo totalmente trasparente. Infine notiamo come è stato sufficiente specificare un lastwill per avere il comportamento logico di: un solo gestore può essere attivo e alla scomparsa di uno di essi; uno di quelli attivi deve prendere il comando. Questo è dovuta alla possibilità di utilizzare variabili nell'esecuzione ritardata dell'aggiornamento dei dati.

Non solo, il comportamento del lastwill SMART M3 è equivalente a quello di un update ritardato [22], differendo solo nella tempistica di quando questo venga effettivamente eseguito. Infatti il primo viene eseguito alla caduta di un cliente mentre il secondo viene attuato dopo

un certo periodo di tempo. In entrambi però si ha la possibilità di esprimere un comportamento in caso del mancato avvenimento di un evento. Potremmo appunto affermare la volontà di eseguire l'update sole se nel periodo di vita del cliente non si è verificata una modifica particolare dell'ontologia. Chiariamo questo livello di espressività con la seguente dimostrazione.

Supponiamo che in una determinata area coperta da un hotspot Wi-Fi ci debba essere sempre almeno un dispositivo mobile connesso. Nel caso in cui non ci fosse nessuno vogliamo che nel sistema venga mantenuta questa informazione sotto forma di una tripla di qualsiasi tipo. Immaginiamo due produttori Figura 7 entrambi producono una tripla del tipo ***<mp:areaHotSpot1> <mp:copertaDa> <"IDKP">***. La presenza dell'altro KP è deducibile dal cambiamento del valore IDKP nella tripla condivisa. Seguiamo le varie fasi con l'ipotesi che il sistema si trovi senza alcun KP all'interno e quindi in condizione "NOKP".

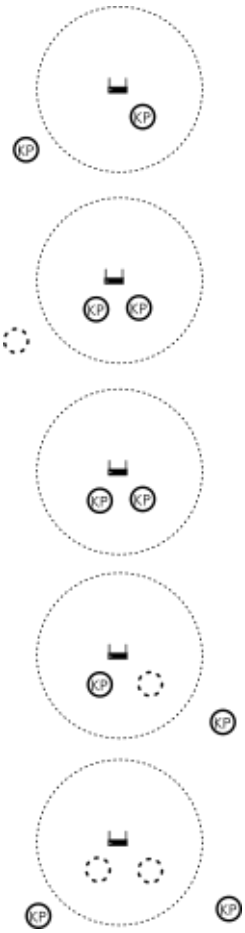

Figura 7 KP mobili

- Il primo KP entra nell'area, pubblica KP1 e esprime le sue ultime volontà.
- Il secondo entra nell'area, modifica il valore IDKP con KP2 e come KP1 esprime un lastwill.
- KP1 ora è consapevole che l'area è coperta da un altro KP e si sottoscrive a ***<sp:areaHotSpot1> <mp:haStato> ****
    - Se lui uscirà dal sistema il suo lastwill non verrà eseguito

    - Mentre se esce prima KP2 lui verrà notificato a causa dell'esecuzione postuma dell'ultime volontà di 2 e reinserirà la ***<mp:areaHotSpot1> <mp:copertaDa> KP1***
- Se Kp2 uscirà e non sarà rimasto nessuno il lastwill scatterà poiché la tripla è ancora ***<mp:areaHotSpot1> <mp:copertaDa> KP2***

Perciò per ottenere il comportamento richiesto è sufficiente che ognuno esprima come ultima volontà quella di produrre la tripla della condizione di assenza clienti se nessun altro ha modificato la tripla condivisa (Codice 3). Il meccanismo ottenuto è stato possibile poiché l'update ritardato di tipo ultime volontà può, appunto, esprimere la seguente condizione: "Se fin quando ero presente nel sistema nessuno è entrato allora scrivi che non c'è nessuno nell'area".

La differenza può sembrare sottile rispetto a MQTT ma ha grande potenzialità. Infatti il meccanismo di lastwill migliorato dall'utilizzo di SPARQL in SMART M3 è in grado di adattarsi alle condizioni postume alla morte del cliente, reagendo allo stato del sistema nonostante il cliente non è più connesso. Nell'altro framework è necessario, nella maggioranza dei casi complessi, della presenza di un ulteriore cliente che sia incaricato di aggregare le informazioni inviatogli e adoperare le giuste modifiche al modello dei dati.

Concludiamo quindi che secondo tale ragioni la funzionalità che andremo a sviluppare è semanticamente superiore a quella riservata ai clienti di un broker basato sull'utilizzo di topic e di fondamentale importanza se si vuole ottenere accurata robustezza a livello applicativo.

```
//Lastwill KP2
PREFIX mp: <http://mobile/>
PREFIX sp: <http://sistema/>
INSERT {
       sp:areaHotSpot1 sp:haStato “NOKP”
}
WHERE {
       mp:areaHotSpot1 mp:copertaDa “KP2”.
}
```

Codice 3 LastWill di uno dei due kp mobili

Sottoscrizioni robuste

Nel sistema SmartM3 un malfunzionamento delle sottoscrizioni provoca comportamenti erronei a livello applicativo, compromettendo l'affidabilità generale dell'applicazione [7]. Per garantire la robustezza della SIB OSGi è quindi di importanza critica studiare l'implementazione attuale del meccanismo in esame rispetto a questo requisito e proporre soluzioni, qualora necessario, che riescano a soddisfarlo. La valutazione teorica che segue è stata fatta in modo da essere indipendente dal framework utilizzato; studieremo infatti una generica funzionalità di pubblicazione e sottoscrizione di informazioni. In *Sottoscrizioni* invece passeremo all'applicazione di tali ragionamenti sul broker semantico.

Una sottoscrizione è, nel senso ampio, una dimostrazione di interesse riguardo un particolare fatto. La sua funzionalità è quella di ottenere uno stream di dati che riguardano modifiche di tale interesse. Il sottoscrittore, dunque, si attende che ogni aggiornamento gli sia prontamente consegnato in qualsiasi caso. Aggiungiamo anche questa funzionalità deve essere servita in maniera remota e che quindi il sottoscrittore sia in un nodo di rete diverso da dove risiede il distributore delle notifiche.

Perciò definiremo una sottoscrizione robusta, se il framework sottostante che la supporta, è in grado di fornire almeno parte della sua funzionalità, anche dopo aver superato un transitorio di caduta connessione.

Come si può notare abbiamo rilassato il vincolo di robustezza. In tal modo possiamo comprendere situazioni specifiche dove, per garantire continuità di servizio, non sia necessario ripristinare completamente la funzionalità di sottoscrizione. Si sono quindi definiti vari livelli di robustezza che un framework di pubblicazione e sottoscrizione può offrire:

- Livello 0: Sottoscrizioni semplici
- Livello 1: Sottoscrizioni con riconnessione.
- Livello 2: Sottoscrizioni con riconnessione e recupero di stato (auto-recovery)
- Livello 3: Sottoscrizioni con riconnessione e rispristino storico (full-recovery)
- Livello 4: Sottoscrizioni con riconnessione e ripristino storico temporizzato (full-time recovery)

Ognuno dei livelli offre un'espressività diversa e copre necessità implementative differenti, descriviamoli considerandoli uno ad uno.

Livello 0

Correntemente è il livello di servizio offerto dalle sottoscrizioni nella SIB OSGi. In questa modalità la proprietà di robustezza è assente. Non vi è infatti nessun livello di garanzia sulla funzionalità offerta. Molto probabilmente una caduta di connessione compromette il funzionamento delle notifiche e la perdita di funzioni applicative. Ciò nonostante può essere utilizzando quando si ha la certezza che la perdita di connessione sia un'eventualità molto rara, o che il suo avvenimento sia così catastrofico che l'applicazione non si comporterebbe comunque normalmente. In questi casi si può giovare della semplicità di questo

meccanismo, risparmiando risorse (si evitano controlli complessi per il ripristino), senza compromettere l'andamento della soluzione software progettata.

Livello 1

Viene garantito che in caso di caduta di connessione la sottoscrizione viene riattivata appena è possibile comunicare nuovamente con il broker. Non vi è però nessun supporto di sincronizzazione tra lo stato del dato conosciuto precedentemente dal cliente e quello corrente nel server. Infatti durante l'assenza altri clienti potrebbero aver modificato l'informazione a cui quello disconnesso era interessato. In questo livello però il nuovo valore non viene fornito ma viene ripristinata la normale funzionalità di consegna notifiche che, dal momento della nuova connessione, arriveranno come previsto. Il caso d'uso è quello di clienti state less, cioè che, all'arrivo dell'evento, compiono una determinata azione idempotente rispetto alla notifica ricevuta. Esempi notevoli sono logger non critici oppure attuatori senza stato come un piccolo altoparlante che suona ogni volta che un sensore rileva il passaggio di una persona (perso un evento non ha senso suonare comunque al ripristino della connettività). Un altro è quello di un aggregatore di dati in streaming, come un sottoscrittore che riceve dei valori in Fahrenheit e deve convertirli in celsius. In questo caso non avrebbe senso ripristinare il contesto poiché probabilmente il valore sarebbe già obsoleto. Questo caso risolve già un buon numero di casi applicativi reali con vantaggio di essere semplice e leggero poiché è necessario solo un rudimentale meccanismo di ripristino connessione.

Livello 2

Come il livello 1 si garantisce la riattivazione del servizio di sottoscrizione qual ora vi siano le condizioni per comunicare con il broker. In aggiunta il

cliente viene informato se il nuovo stato presente nel broker è differente da quello pregresso. Questa nuova informazione permette di fatto la sincronizzazione dei due stati alla riconnessione. Un caso d'uso sarebbe quindi proprio quello di un monitor che visualizza dei valori raramente variabili. Un'istanza potrebbe essere quello di uno schermo intelligente che visualizza il numero di studenti all'interno di un'aula studio. Per il suo corretto funzionamento è fondamentale che al ripristino del dialogo con il broker venga comunicato se il numero è aumentato o diminuito. Altrimenti fornirebbe un risultato incorretto fino alla prossima entrata/uscita di uno studente dall'aula. Al contrario del livello precedente i clienti sono state full e hanno bisogno di informazioni precise sullo stato. Il livello d'espressività è superiore rispetto ai precedenti e possiamo affermare che la maggioranza dei casi applicativi può considerare una sottoscrizione robusta se garantisce questo livello. Il consumo di risorse è maggiore poiché è necessario che lo stato prima dell'evento avverso venga mantenuto per poterlo poi confrontare con quello aggiornato al ristabilimento del contatto (vedi *Sottoscrizioni*).

Livello 3

Stessa qualità di servizio delle sottoscrizioni con riconnessione e recupero di stato, ma è assicurato l'invio di tutti gli eventi accorsi durante l'assenza. Infatti nel livello precedente le informazioni storiche sulla modifica dello stato vengono perse. Appunto, utilizzando l'esempio precedente, vengono persi i dati su quanti studenti sono usciti e entrati durante il transitorio di disconnessione. Ad esempio se il monitor è disconnesso quando sono presenti 5 persone in aula e, mentre rimane in quella condizione ne escono due e ne entrano due alla riconnessione, non è ricevuta nessuna notifica poiché non c'è differenza tra numero di alunni obsoleto e corrente. La funzione di questo tipo di applicazione è

comunque ripristinata poiché può essere garantita utilizzando il livello di servizio 2, aggiungendo un semplice requisito però la situazione cambia. Viene richiesto che il monitor debba anche tenere traccia di quali studenti siano entrati e usciti al fine di assicurare sicurezza e raccogliere informazioni, quali tipo di frequentatori dei corsi universitari che l'utilizzano. In questo caso è necessario garantire uno storico di entrate e uscite anche quando il monitor per qualche motivo è stato disconnesso dalla rete. In generale il livello di servizio corrente dovrà essere garantito per quelle applicazioni nelle quali non è permessa la perdita di dati riguardanti modifiche dello stato. Più precisamente in applicativi software nei quali si ha bisogno della sequenza degli eventi per ottenere un'informazione valida.

In questa versione la funzionalità della sottoscrizione è stata completamente ripristinata fornendo la vera continuità del servizio anche dopo l'evento avverso. Rispetto ai livelli precedenti questo caso richiede un impiego di risorse maggiore, poiché in principio deve tenere traccia di tutte le modifiche accadute durante l'assenza del cliente. Il soddisfacimento di tal proprietà è formalmente un problema insidioso per il consumo di memoria. Difatti la memoria impiegata cresce in maniera direttamente proporzionale al tempo in cui il sottoscrittore è assente dal sistema e nel caso egli non ne facesse più ritorno si occuperebbero risorse inutilmente.

Livello 4

Anche se il livello 3 è sufficiente ha ripristinare completamente una sottoscrizione, in particolari scenari potrebbe essere necessario garantire un ulteriore qualità di servizio. In questo livello vengono consegnate tutte le notifiche perse in aggiunta al tempo in cui si son verificate. Normalmente questo è un requisito funzionale da aggiungere

al concetto stesso di sottoscrizione. Infatti la funzionalità da garantire è diversa rispetto a quello di sottoscrizione senza temporizzazione. Tale servizio si modifica in: ottenere uno stream di dati temporizzato riguardo alle modifiche di un'informazione di interesse. Il caso d'uso tipico di questa qualità del servizio è quello di una applicazione medica che tiene traccia di certi valori dei pazienti. Il medico ha sul cellulare l'applicazione che mostra i livelli quali battito cardiaco, glicemia, saturazione etc. Egli vuole ricevere un allarme se certi dati escano da un intervallo di sicurezza, non solo, vuole sapere anche in che periodo del giorno tale allarme è stato generato. Il livello 3 non basterebbe a garantire ciò poiché i tempi non sono presi in considerazione. Nonostante ciò il problema è simile poiché sarebbe sufficiente tenere traccia oltre degli eventi passati anche del tempo in cui si son verificati.

Si nota quindi che il livello corrente è il caso più generale di garanzie specificabili poiché se si garantisce questo allora tutti gli altri son soddisfatti. Il consumo di risorse di memoria è simile a quello del livello 3 ma è necessario un processing aggiuntivo degli eventi per aggiungervi l'istante di accadimento.

Ulteriori considerazioni

Il problema della robustezza del meccanismo di sottoscrizioni remote è influenzato da quello dell'affidabilità di connessione tra due nodi. Tanto più è possibile garantire la robustezza di una connessione tanto più il servizio di notifiche sarà robusto. Infatti se per assurdo la comunicazione non cadesse mai avremo risolto alla radice il problema e non sarebbe necessario studiare delle politiche per la sua risoluzione. Al contrario in ambito IoT non è possibile garantire tale condizione, data l'eventuale presenza di nodi mobili e/o network con bassa qualità di servizio.

Possiamo però valutare la probabilità che in una data scelta implementativa il collegamento tra servitore e cliente cada. Poi valuteremo tanto più robusta una soluzione che abbia minor probabilità di perdita comunicativa. Per tali ragioni è stata studiata l'implementazione attuale del meccanismo di connessione tra SIB e OSGi e una proposta di modifica. Specificatamente è stato valutato se l'utilizzo di una connessione per sottoscrizione sia più robusto rispetto a quello con una connessione per cliente attraverso quale veicolare tutte le notifiche.

Infine dagli studi eseguiti è emerso che una soluzione più robusta sarebbe quella di utilizzare la modifica proposta poiché la probabilità di perdita di funzionalità del sistema è minore rispetto a quella attuale. Rimandiamo il lettore all' *Appendice: Studio robustezza sottoscrizioni con singola socket* per la discussione dettaglia sul problema e per conoscere i pro e contro della modifica proposta.

Conclusioni

Abbiamo visto come la robustezza è una proprietà fondamentale di una piattaforma interoperabile e quali livelli possono essere garantiti riguardo alla funzionalità di sottoscrizione. Inoltre è stato studiato un meccanismo robusto, la dichiarazione di ultime volontà, ed è stata dimostrata la sua utilità in casi applicativi critici

Progetto

Dallo studio del problema è emersa la necessità di implementare due meccanismi importanti per garantire robustezza al broker semantico: Lastwill semantico e sottoscrizioni con riconnessione automatica. Perciò come requisito funzionale è necessario introdurre un nuovo comando interpretabile dalla SIB che contenga la volontà di un KP nel caso di

disconnessione o di rottura inaspettata. Poi queste volontà dovranno essere eseguite nell'eventualità in cui venga rilevata l'assenza dal sistema del KP che l'ha specificata. Inoltre un'altra specifica è quella di implementare un meccanismo che sia in grado di riattivare una sottoscrizione dopo un malfunzionamento di rete o di caduta della SIB. Questo dovrà almeno garantire un livello di servizio pari alla sottoscrizione con recupero del contesto, specificata nello studio del problema. Infine oltre ai requisiti funzionali sopracitati sono stati aggiunti due requisiti non funzionali. Il primo è quello di garantire la reto compatibilità sia del protocollo applicativo che della SIB OSGi con tutte le versioni precedenti di KPI. Dualmente è richiesto anche che se ci fossero modifiche da apportare alle KPI anch'esse siano compatibili con le vecchie versioni di SIB già sviluppate. Le motivazioni a sostegno di questo importante requisito non funzionale, che influenza marcatamente le modalità di implementazione, sono date dal grande numero di software accademico e di ricerca nato utilizzando versioni precedenti del broker semantico. Il secondo invece è quello di non degradare le performance di esecuzione dei comandi interpretati. In questo caso la motivazione è quella di poter garantire buoni tempi di risposta in un ambito così critico come l'IoT.

Esporremo, dunque, come le funzionalità studiate sono state implementate nell'ecosistema SIB OSGi. Avremo cura di evidenziare le varie fasi di progettazione e mostrare diverse implementazioni, spiegando i loro punti di forza e debolezza. Per prima cosa parleremo di come si è svolto lo sviluppo del lastwill per poi vedere la funzionalità di sottoscrizione con recupero del contesto.

Lastwill

Dallo studio dei requisiti e dal ambito del problema sono emersi tre punti aperti di implementazione:

1. In che modo il cliente e quindi le KPI possono specificare le loro ultime volontà? Qual è il formato del messaggio? e come questo si inserisce nel SSAP?
2. Come verifico l'assenza del KP dal sistema?
3. Una volta stipulato in contratto tra cliente e servitore le volontà sono modificabili o immutabili?

Partiamo dal punto più importante cioè come modificare il protocollo SSAP aggiungendo la funzionalità di lastwill, considerando i requisiti non funzionali. Un primo approccio potrebbe essere quello di modificare il messaggio di JOIN aggiungendo un parametro opzionale per la specifica di uno SPARQL o RDF update che verrà valutato come volontà del KP che si presenta alla SIB (Codice 4). Riutilizzare un comando già previsto e decorarlo con un nuovo parametro, permette un'ottima reto compatibilità poiché se il parametro non fosse previsto questo verrebbe ignorato e quindi permetterebbe un utilizzo invariato del nuovo protocollo con le SIB precedenti.

```
<SSAP_message>
    <message_type>REQUEST</message_type>
    <transaction_type>JOIN</transaction_type>
    <transaction_id>28</transaction_id>
    <node_id>05dc59c2-e789-4df7-9591-a7630684308d</node_id>
    <space_id>X</space_id>
    <parameter name="lastwill">
      PREFIX dp: <http://drones/prop/>
      DELETE { <http://drones/drone0> dp:status  "Online" }
      INSERT { <http://drones/drone0> dp:status  "Offline" } WHERE {}
    </parameter>
</SSAP_message>
```

Codice 4 Lastwill incorporato in un messaggio di JOIN

D'altra parte potrebbe essere necessario informare lo sviluppatore di applicazioni SMART M3 dell'incompatibilità del messaggio di lastwill con la versione di SIB usata. Esso potrebbe utilizzare questa informazione per attuare politiche volte a simulare la funzionalità in questione come mostrato in [2], oppure valutare un aggiornamento del broker semantico. Inoltre questa scelta limiterebbe le opzioni del punto 3, costringendo a definire una volontà unica per ogni sessione di KP. Infatti per ogni sessione è ammesso un unico messaggio di JOIN quindi per modificare le intenzioni del KP in caso di malfunzionamenti sarebbe necessario prima uscire dal sistema con una LEAVE e poi inviare un nuovo comando di JOIN contenente il nuovo messaggio di lastwill.

Un'altra scelta potrebbe essere quella di definire un nuovo comando nel protocollo applicativo SSAP. Definiremo quindi come esso possa essere serializzato in formato xml all'interno del SSAP e quale sia la sua semantica. La semantica è definita dal significato di lastwill. Quindi la valutazione di questa nuova richiesta corrisponde con la registrazione delle volontà specificate dal KP richiedente e l'esecuzione di esse se e solo se quest'ultimo non fosse più presente all'interno del sistema. Il formato invece non differisce da quello di un semplice messaggio di UPDATE, poiché le informazioni contenutevi sono le stesse (Codice 5).

```
<SSAP_message>
    <transaction_type>LASTWILL</transaction_type>
    <message_type>REQUEST</message_type>
    <transaction_id>30</transaction_id>
    <node_id>3b5f20ac-e2a0-4ec8-a81e-4a2f820abf34</node_id>
    <space_id>X</space_id>
    <parameter name="query" encoding="SPARQL-UPDATE">
      PREFIX dp: <http://drones/prop/>
      DELETE { <http://drones/drone0> dp:status  "Online" }
      INSERT { <http://drones/drone0> dp:status  "Offline" } WHERE {}
    </parameter>
    <parameter name = "confirm">TRUE</parameter>
  </SSAP_message>
```

Codice 5 Lastwill come nuovo comando

L'ultima soluzione proposta non sarebbe retro compatibile, poiché SIB precedenti non riconoscerebbero il formato del nuovo comando. D'altro canto in questo caso è possibile informare facilmente l'utente delle KPI grazie al messaggio di errore fornito dalle vecchie versioni nel caso il comando non fosse riconosciuto. Inoltre darebbe più flessibilità sulla scelta di mantenere le volontà immutabili oppure no durante la sessione del cliente.

L'implementazione scelta è quella di creare un nuovo tipo di comando, nonostante violasse in principio il requisito di reto compatibilità. In realtà questa lo viola solo apparentemente, poiché essendo il lastwill una nuova funzionalità, il suo utilizzo necessità comunque di aggiornare la SIB e di utilizzare le nuove KPI. Inoltre l'informazione di comando non riconosciuto, come abbiamo specificato pocanzi, è necessaria per il corretto funzionamento dell'applicazione distribuita. Quindi l'utilizzo di una politica silent-failure causerebbe a runtime comportamenti inaspettati in scenari critici come la caduta di connessione tra SIB e KP. Infine permette maggiore flessibilità sul formato del messaggio e dà la facoltà di modificare le volontà durante il ciclo di vita del KP senza dover utilizzare l'operazione macchinosa di LEAVE-REJOIN.

Il secondo punto fondamentale per il funzionamento di questo nuovo meccanismo è quello di rilevare in maniera accurata la presenza e/o l'assenza di un cliente nel sistema. Il broker OSGi supporta correntemente un solo tipo di protocollo cioè il SSAP veicolato su TCP. Nonostante ciò si è voluto dare alla soluzione creata un certo livello di generalità per permettere l'utilizzo di lastwill anche a futuri protocolli. Per ora ci concentreremo sulla soluzione nel protocollo TCP, rimandando il lettore al capitolo di progetto su SIB OSGi per approfondire le tecniche che hanno generalizzato questa soluzione.

La verifica della presenza di cliente collegato tramite TCP è un problema noto in letteratura e prende il nome di verifica di connessioni cadute o parzialmente-aperte [23]. Sapere se un cliente è collegato o no ci permette di inferire il suo stato. Infatti l'impossibilità di comunicare dati nel caso di applicazioni distribuite significa di fatto che il componente software non è più presente nel sistema e perciò può essere considerato "morto". Tra le varie soluzioni proposte c'è quella dell'utilizzo di un heartbeat tra le due parti. L'heartbeat è un messaggio di controllo inviato da un nodo ad un altro ad intervalli regolari. Il layer di trasposto più famoso al mondo, infatti, non conferisce alcuna garanzia di notifica nel caso in cui la connessione tra i due nodi venga interrotta mentre questi non comunichino. Al contrario, se tra i due nodi c'è uno scambio di dati, il protocollo è in grado di determinare l'impossibilità dell'invio tramite RTO, un time out calcolato sui tempi di percorrenza dei segmenti nella rete [24]. Per questo l'heartbeat permette a chi l'invia di rilevare la presenza di chi è in ascolto del messaggio, tramite l'eccezione data dal livello di trasporto. Altre soluzioni implicherebbero l'uso della funzionalità di keep alive. Esso però è un servizio controverso, infatti non supportato allo stesso modo da tutti gli SO fino ad arrivare a casi estremi in cui non è implementato affatto come nel caso di stack TCP integrati in hardware. Inoltre poiché implica l'invio di segmenti vuoti di tipo ACK secondo il [25] questi potrebbero essere eliminati dai router lungo il percorso di connessione tra i due nodi. Per tali motivazioni si è utilizzata la soluzione di rilevazione tramite heartbeat.

Una volta definito il meccanismo di rilevazione di connessione caduta è stato necessario specificare il formato del messaggio di controllo inviato dal servitore al cliente e come questo venga inviato. La direzione di comunicazione è obbligata dal fatto che è il broker che necessita

dell'informazione sulla presenza KP. Per tale motivazione si è previsto un componente software attivo che riutilizzi la connessione del comando di lastwill per mandare periodicamente al cliente i messaggi di controllo. In particolare è stato previsto un thread che iterasse sulla collezione di connessioni rimaste aperte. Esse vengono tenute aperte in due casi:

1. Sono connessioni instaurate da comandi di sottoscrizione. In questo caso vengo riutilizzate per inviare notifiche al KP. La rilevazione in questo caso è importante per la liberazione delle risorse occupate da una sottoscrizione.
2. Sono derivate da un comando di lastwill. Quindi oltre al processing del messaggio sarà necessario implementare anche la gestione di questo ulteriore tipo di connessione.

In entrambi i casi il thread potrà rilevare se la connessione è ancora attiva o no e giudicare quindi lo stato del KP. Il formato del messaggio di controllo invece è un aspetto critico di questa fase di progetto, soprattutto perché va a toccare la reto compatibilità del broker. Infatti, poiché esso viene inviato anche su connessioni derivate da sottoscrizioni, vecchie KPI potrebbero non accettare il formato e uscire in modo inaspettato dalla routine di ricezione notifiche. Le opzioni sono quelle di definire un messaggio di notifica vuoto oppure utilizzare un carattere che è trasparente ad un generico parser xml. La prima scelta è valida ma darebbe all'utente l'onere di dover gestire notifiche nulle e consumerebbe molto banda (656 byte per messaggio) a causa della verbosità xml. Per tale motivo si è optato per l'utilizzo del carattere '\n' risparmiando banda (1 byte asci) e poiché il parser xml lo ignora non generando problemi con implementazioni precedenti. Il [26] specifica come carattere nullo per messaggi testuali i caratteri CRLF ('\r' '\n'), nel

nostro caso però è stato sufficiente utilizzare '\n' poiché è sempre considerato dagli interpreti xml allo stesso modo di CRLF.

L'ultimo punto rimasto aperto è quello di scegliere tra volontà immutabili e mutabili. Secondo la nostra valutazione non ci sono ragioni per imporre il vincolo di immutabilità agli sviluppatori di applicazioni di terze parti. Infatti la mutabilità del testamento di un KP potrebbe portare vantaggi quali ad esempio definire direttive a seconda dell'ultimo stato in cui si trovasse prima della rilevazione del fallimento. Supponiamo un KP sensore critico, il suo scopo è rilevare l'intrusione di personale non autorizzato in determinate zone di un cantiere e solo dopo la fine dell'orario di lavoro. Il KP è collegato tramite ethernet con la SIB e supponiamo che inizi il suo ciclo di vita come non attivo, cioè durante le ore di lavoro. In questo stato vuole che il suo lastwill sia un messaggio di sostituzione o warning, poiché il personale potrà tempestivamente trovare le ragioni di malfunzionamento. Durante la fase attiva invece il suo lastwill sarà quello di attivare comunque l'allarme, poiché il suo mancato funzionamento potrebbe essere dovuto al taglio dei cavi ethernet da parte di qualche mal intenzionato. A causa di ciò è stato il vincolo di immutabilità del lastwill è stato rilassato perciò è possibile sovrascrivere una volontà durante la sessione di un KP.

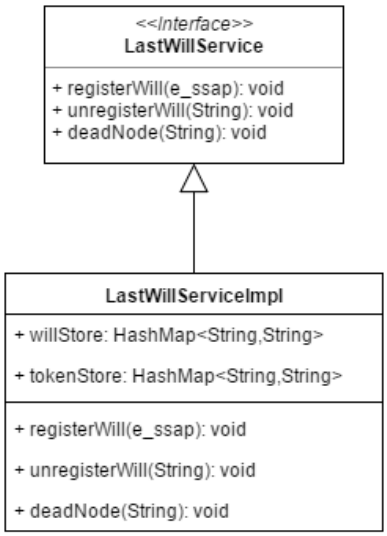

Figura 8 Diagramma UML del servizio di lastwill

Lastwill nella SIB OSGi

Vediamo ora come l'analisi progettuale è stata tradotta in modifiche puntuali e mirate all'architettura del broker semantico OSGi. Lo sviluppo

è stato diviso in due fasi: la prima è stata incentrata sull'aggiunta del nuovo comando di lastwill, la seconda invece si è dedicata all'implementazione del meccanismo di riconoscimento connessioni cadute.

L'azione più conservativa e naturale è stata quella di aggiungere un bundle che si occupasse di gestire i le volontà dei vari KP. In **Errore. L'origine riferimento non è stata trovata.** sono mostrati i metodi offerti dal nuovo servizio di lastwill. In particolare è possibile registrare una nuova volontà dato l'e_ssap. In quest'ultimo sono contenute due informazioni: la prima è l'identità dal KP quindi il token che identifica la connessione e l'id del nodo, la seconda è il messaggio di update cioè il lastwill vero e proprio. Il *LastWillServerceImpl* associa il token all'e_ssap utilizzando l'hashMap willStore. Inoltre accoppia il token con il l'id del nodo, con il fine di verificare se per un dato nodo è già presente una volontà salvata, in tal caso il servizio provvederà a sovrascriverla. Il metodo unregisterWill si occupa invece del compito duale, è utilizzato nel caso in cui il KP chieda tramite una primitiva di LEAVE l'uscita graceful dal sistema. In questo modo le volontà non vegano eseguite se il KP chiude di proposito i contatti con il broker. In ultimo il metodo deadNode avvia il meccanismo di esecuzione delle volontà del KP con un certo node id.

Il comportamento atteso dal servizio specificato è anche estrapolabile dal unit test associato alla classe di *LastWillServerceImpl* (Codice 6).

```
public class LastWillServiceImplTest {

	private static final String TOKEN3 = "TOKEN3";
	private static final String TOKEN1 = "Token1";
	private static final String TOKEN2 = "Token2";
	LastWillServiceImpl impl;
	e_ssap message ;
	FakeBundleContext context;
	//……

	/**
	 * Test method for {@link
sofia.sib.lastwill.impl.LastWillServiceImpl#registerWill(sofia.sib.tools.e_ssap)}.
	 */
	@Test
	public void testRegister() {
		message.setToken(TOKEN1);
		impl.registerWill(message);

		assertTrue(impl.getNodeStore().containsKey("Node1"));
		assertTrue(impl.getStore().containsKey(TOKEN1));
	}

	/**
	 * Test method for {@link
sofia.sib.lastwill.impl.LastWillServiceImpl#deadNode(java.lang.String)}.
	 */
	@Test
	public void testDeadNode() {
		message.setToken(TOKEN1);
		impl.registerWill(message);

		assertTrue(impl.getNodeStore().containsKey("Node1"));
		assertTrue(impl.getStore().containsKey(TOKEN1));

		impl.deadNode(TOKEN1);

		assertFalse(impl.getNodeStore().containsKey("Node1"));
		assertFalse(impl.getStore().containsKey(TOKEN1));
	}

	@Test
	public void testMultipleTokenOneNode() {
		message.setToken(TOKEN1);

		impl.registerWill(message);

		assertTrue(impl.getNodeStore().containsKey("Node1"));
		assertTrue(impl.getStore().containsKey(TOKEN1));

		message.setToken(TOKEN2);
		impl.registerWill(message);

		assertFalse(impl.getStore().containsKey(TOKEN1));
		assertTrue(impl.getStore().containsKey(TOKEN2));
		assertEquals(impl.getNodeStore().get("Node1"),TOKEN2);
	}

	@Test
	public void testRegisterForBrokenCon() {
		message.setToken(TOKEN3); //Not in the gate connections
		impl.registerWill(message);

		assertFalse(impl.getNodeStore().containsKey("Node1"));
		assertFalse(impl.getStore().containsKey(TOKEN1));
	}
```

Codice 6 Junit test della classe LastWillServiceImp

Verificato che il comportamento della classe implementata fosse quello atteso, prima di passare all'integrazione con il sistema si è passati allo sviluppo del meccanismo di verifica connessione. Questa funzionalità ha richiesto la modifica del bundle TCP in modo da tener traccia, oltre delle connessioni provenienti da sottoscrizioni, anche di quelle proventi da comandi di registrazione lastwill. Utilizzando il percorso naturale dei comandi è stato possibile riconoscerne la tipologia durante la chiamata di callback. Come si può verificare nel Codice 7 nella versione attuale le socket associate alla connessione vengono chiuse per tutti i comandi tranne quelli derivanti da una transazione di tipo SUBSCRIBE o LASTWILL.

```
if (os.transaction_type.equals("SUBSCRIBE")) {
                        //SUBSCRIBE
        if (!os.getSubscriptionID().equals("") && os.getMessageType().equals("CONFIRM")) {
                synchronized(pendingRequests) {
                        pendingRequests.get(token).setSubscription(true);
                        pendingRequests.get(token).setSubID(os.getSubscriptionID());
                        activeSubscriptions++;
                }

        }
}
else if(os.transaction_type.equals("LASTWILL")){
        Logger.log(VERBOSITY.INFO, tag, "Registering lastWill ");
}
else {
        //UPDATE/QUERY
        //Release token and free request
        synchronized(pendingRequests) {
                if (!pendingRequests.get(token).isSubscription()) {
                        tokener.releaseToken(token);
                        pendingRequests.remove(token);
                }
        }

        try {
                out.close();
                clientSocket.close();
                clientSocket = null;
        } catch (IOException e) {...}
}
```

Codice 7 Gestione callback del lastwill

Nella coda *pendingRequest* saranno quindi contenute tutte le richieste che ancora devono essere soddisfatte (e.s. comandi che attendono di essere processati dal motore della SIB), le connessioni delle sottoscrizioni e quelle del lastwill. A questo punto per attuare la

funzionalità di heartbeat è stato sufficiente aggiungere un thread che periodicamente venisse risvegliato e mandasse, per ognuna delle socket contenute nella coda, il carattere '\n'. L'impossibilità di inviare tale messaggio di controllo indica una caduta di connessione e/o crash del cliente, eventualità che viene indicata da un'eccezione nella primitiva di *write.* A questo punto è necessario comunicare l'informazione rilevata ai due bundle interessati cioè quello di lastwill e quello di subscription. L'operazione può essere eseguita o tramite chiamata diretta oppure tramite un approccio ad eventi.

La prima soluzione implicherebbe una dipendenza diretta tra il livello di comunicazione e funzionalità di alto livello come la gestione delle sottoscrizioni e delle volontà. Inoltre all'introduzione di nuovi requisiti potrebbe essere necessario inserire nuove chiamate dirette a nuovi tipi di bundle, rendendo necessaria un ulteriore modifica del bundle TCP. Un esempio potrebbe essere il requisito di velocizzare il processing dei comandi cancellando i messaggi in coda che sono derivati da connessioni cadute. Utilizzando questa soluzione dovremo aggiungere fisicamente la chiamata al bundle messageHandler il che richiederebbe quindi la modifica di due moduli OSGi. Infine questa soluzione limita i vantaggi offerti dal framework utilizzato, poiché in caso di update di un servizio di alto livello come il lastwill dovrebbe essere fermato anche il TCP negando una funzionalità di base nonostante la SIB potrebbe ancore processare tutti gli altri tipi di comandi.

Per queste ragioni e data la natura dell'informazione "caduta di un nodo" prettamente di tipo eccezionale, si è scelto l'utilizzo del paradigma di programmazione ad eventi. OSGi offre tra i suoi moduli base anche un broker per il dispaccio di eventi tra i vari componenti software registrati. Purtroppo tale servizio non è ugualmente supportato tra le diverse

versioni del framework. In più gli eventi e i sottoscrittori non sono fortemente tipati, provocando la necessità di check manuali su stringhe per verificare che l'evento ricevuto sia del tipo specificato. Date queste problematiche si è optato per lo sviluppo di un ulteriore servizio che ne facesse le veci, l'EventHub (riprenderemo in seguito la trattazione sull'implementazione di questo bundle). L'architettura del sistema è quindi quella specificata in Figura 9.

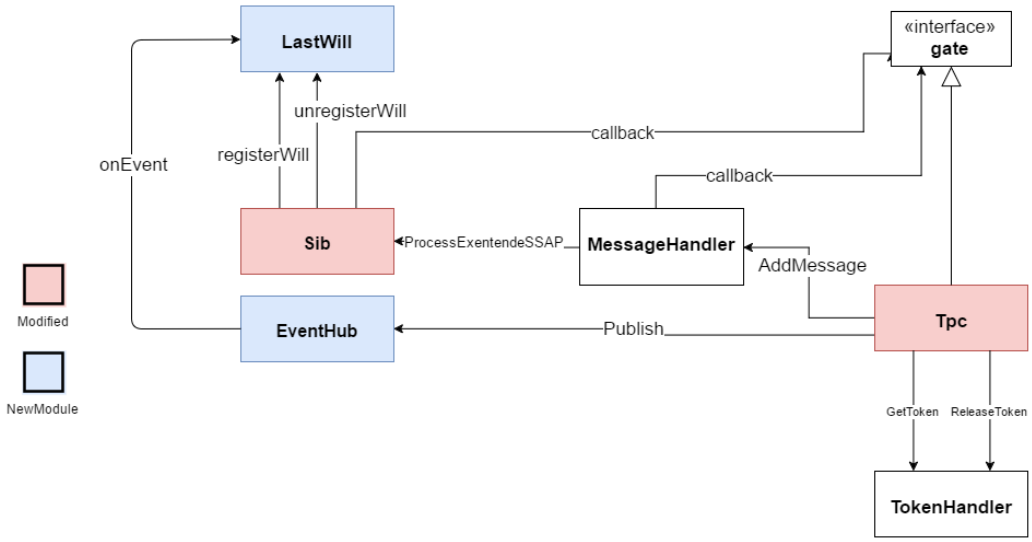

Figura 9 Architettura dopo la modifica, con lastwill e eventhub.

L'utilizzo del nuovo bundle per la gestione di eventi permette l'indipendenza tra il lato rilevazione caduta nodo e azioni da intraprendere se questa eventualità accada. La soluzione quindi è abbastanza generale da permettere l'introduzione di nuovi protocolli senza dover modificare nessuno dei bundle definiti. Maggiormente permette in maniera trasparente l'aggiunta del meccanismo di processing solo dei massaggi con connessione attiva, specificato nel precedente esempio.

Per concludere il meccanismo di lastwill manca proprio l'esecuzione delle volontà. Esse devono essere passate al bundle che si occupa dell'interpretazione dei comandi, *sofia.sib.store.Service*. Rimanendo più coerenti con l'architettura e per non creare problemi di sincronizzazione tra il thread di smistamento comandi e quello di smistamento eventi, le ultime volontà vanno consegnati al servizio *sofia.sib.messageHandler.messageHandlerService*. In entrambi i casi abbiamo comunque un problema di dipendenze circolari (Figura 10) risolvibile tramite un principio noto in letteratura chiamato inversione della dipendenza [27]. Esso è utilizzato principalmente tra dipendenze di classi ma può essere applicato efficacemente anche nel caso di relazioni tra moduli. Infatti è stato sufficiente astrarre i servizi in un'interfaccia e la problematica è stata risolta (Figura 11). Questo tipo di design pattern è incoraggiato anche da bndtools e viene annoverato tra quelli fondamentali per far uso corretto del framework [28].

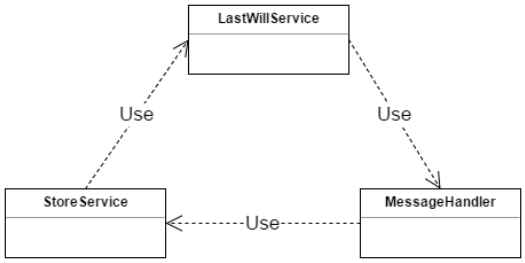

Figura 10 Schema dipendenze circolari

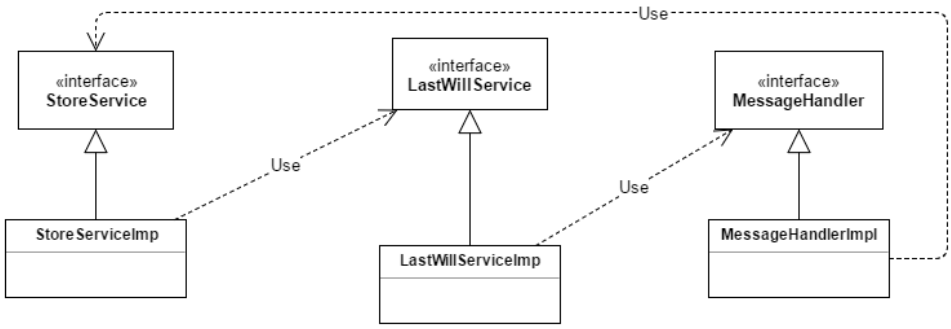

Figura 11 Dipendenze dopo aver applicato il principio di inversione

L'ultimo problema affrontato è stato quello della concorrenza. I due thread concorrenti attivi durate l'esecuzione del comando di lastwill

sono quello definito in *sofia.sib.messageHandler.messageHandlerService* e quello che si occupa della consegna degli eventi (sollecitato dal processo di verifica connessione). Un punto critico è quello della registrazione delle ultime volontà. L'operazione in questione deve essere atomica poiché se un evento di morte del nodo accadesse in contemporanea alla sua esecuzione il sistema potrebbe trovarsi in uno stato non valido. Ad esempio se l'informazione arrivasse appena prima dell'inserimento del messaggio nella mappa delle volontà esso verrebbe correttamente registrato ma non sarebbe mai più eseguito occupando inutilmente spazio di memoria. Il corretto funzionamento prevedrebbe che poiché le ultime volontà sono state richieste per una connessione già morta allora esse non devono essere registrate ma è necessario che siano processate al più presto per poterle eseguite. Per tale motivazioni si è deciso di proteggere l'operazione critica tramite un lock concorrente (Notare come in *Codice 8* ogni metodo d'interfaccia sia coperto da un *synchronized*). Inoltre è stata aggiunta una dipendenza ulteriore tra il servizio di lastwill e il livello di connessione estendendo l'interfaccia gate con un metodo aggiuntivo che è necessario per appurare lo stato della connessione all'inserimento del messaggio. Analizzando il codice è possibile verificare che le volontà vengono inserite se e solo se il metodo *verifyConnection* restituisce vero e che se anche arrivasse in maniera concorrente una chiamata al metodo *deadNode*, il lock garantisca che questa venga eseguita dopo l'avvenuta registrazione. Dualmente se la chiamata *deadNode* avviene prima il metodo *veryConnection* restituirà falso e perciò il lastwill verrà prontamente inserito tra i comandi da eseguire (Codice 8,Figura 12).

```
public class LastWillServiceImpl implements LastWillService,Subscriber<BrokenConnectionEvent> {
	private static final String tag = "LastWill";
	private HashMap<String, e_ssap> willStore = new HashMap<>();
	private HashMap<String,String> tokenStore = new HashMap<>();

	private BundleContext context;
	private MessageHandlerService handler;

	public LastWillServiceImpl(BundleContext context,MessageHandlerService handler) {
		this.context = context;
		this.handler = handler;
	}

	@Override
	public synchronized void registerWill(e_ssap lastwill_notice) {
		ServiceReference<gate> callbackRef = (ServiceReference<gate>) context
				.getServiceReference(lastwill_notice.getCallbackServiceName());
		gate callbackService = context.getService(callbackRef);

		if (!callbackService.verifyConnection(lastwill_notice.getToken())) {
			Logger.log(VERBOSITY.WARNING, tag,
					"last will for a broken connection with token: " +
lastwill_notice.getToken());
			transformToUpdate(lastwill_notice);
			handler.addMessage(lastwill_notice);
		} else {
			addNoticeToStores(lastwill_notice);
		}

		context.ungetService(callbackRef);
	}

	private void addNoticeToStores(e_ssap lastwill_notice) {
		//NOTICE: read put and remove javadoc for miss/hit behavior
		willStore.remove(tokenStore.get(lastwill_notice.getNodeID()));
		tokenStore.put(lastwill_notice.getNodeID(), lastwill_notice.getToken());
		willStore.put(lastwill_notice.getToken(), lastwill_notice);
	}

	@Override
	public synchronized void deadNode(String dead_node_token) {
		 if(willStore.containsKey(dead_node_token)) {
			e_ssap last_will = willStore.get(dead_node_token);
			transformToUpdate(last_will);
			unregisterWill(dead_node_token, last_will.getNodeID());
			handler.addMessage(last_will);
		}else{
			Logger.log(VERBOSITY.WARNING, tag, "No last will found for connection with
token: "+dead_node_token);
		}
	}

	private void unregisterWill(String dead_node_token,String node_id) {
		tokenStore.remove(node_id);
		willStore.remove(dead_node_token);
	}

	private void transformToUpdate(e_ssap update_request) {
		update_request.setTransactionType("UPDATE");
		update_request.setSSAP(update_request.getSSAP().replaceFirst("LASTWILL", "UPDATE"));
	}


	@Override
	public synchronized boolean unregisterWill(String node_id) {
		if(tokenStore.containsKey(node_id)){
			String token = tokenStore.get(node_id);
			unregisterWill(token, node_id);
			return true;
		}
		return false;
	}

	@Override
	public void onEvent(BrokenConnectionEvent e) {
		deadNode(e.getConnectionToken());
	}
```

Codice 8 Codice d'implementazione lastwillService

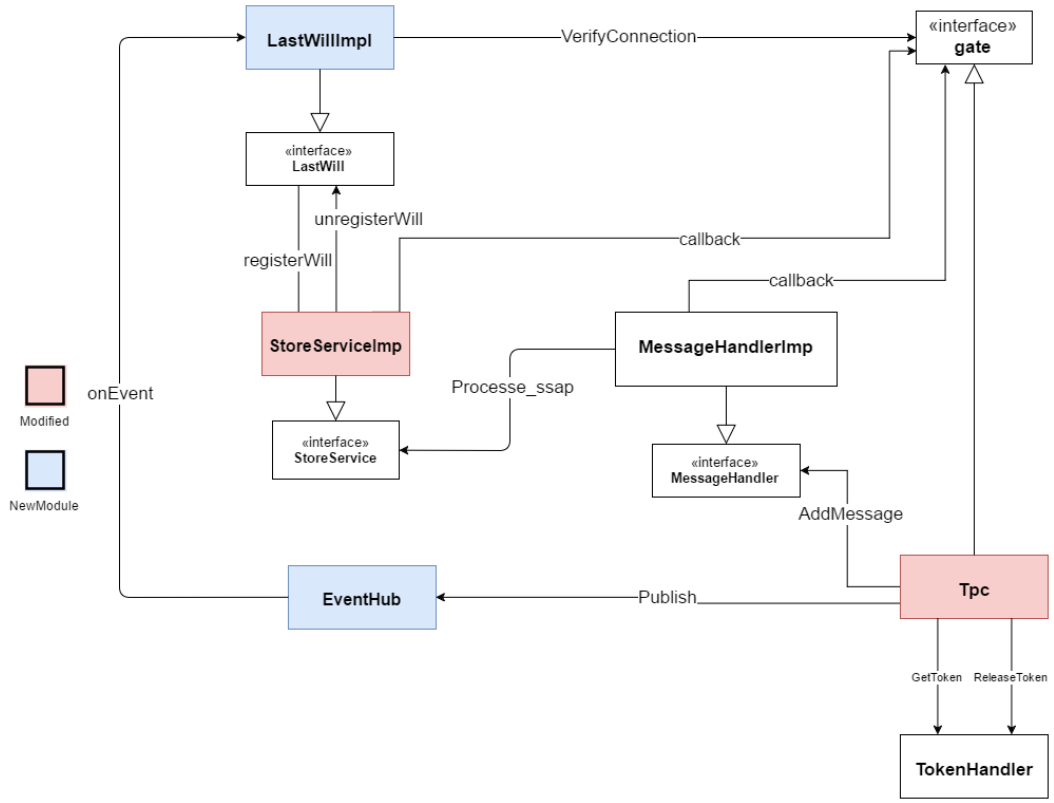

Figura 12 Architettura finale modificata

EventHub

L'EventHub è un nuovo modulo OSGi inserito nell'architettura della SIB per sfruttare il modello di programmazione ad eventi. La struttura di questo componente software si inspira ad altre già presenti correntemente nel marcato come l'EventBus [29] e EventAdmin [30], servizio di eventi presente nel framework. L'obbiettivo è stato quello di creare un servizio semplice, debolmente legato agli altri moduli e fortemente tipato, caratteristiche mancanti nel servizio offerto di base da OSGi. Infatti EventAdmin è stato sviluppato nel periodo antecedente all'introduzione in java dei generici e quindi ogni evento è stato rappresentato da una classe non estendibile identificata da una stringa e un insieme di proprietà. Il vantaggio è quello di poter aggiungere proprietà dinamicamente, a discapito però della sicurezza di tipo. Infatti

lato sottoscrittore (ricevente) se si è inscritto ad N messaggi con M proprietà è richiesto di individuare il tipo di messaggio ricevuto tramite una cascata di if/else con check sulla stringa identificativa per poi eseguire un cast per ogni proprietà interessante dell'evento. Per ovviare a queste problematiche nel EventHub (Figura 13) gli eventi sono sottoclassi della classe base *sofia.sib.eventhub.Event* e i riceventi ottengono direttamente l'oggetto evento della classe specificata alla sottoscrizione. Infatti essi possono sottoscriversi implementando l'interfaccia *sofia.sib.eventhub.Subscriber* specificando a che tipo di evento sono interessati nel generico. Questo meccanismo permette la sottoscrizione pulita a più tipi di eventi infatti se si è interessati a tipi diversi è sufficiente implementare varie interfacce Subscriber con diversi tipi generici avendo un metodo gestore per ogni evento interessante. Non solo è inoltre possibile sfruttare la gerarchia delle classi per inscriversi a "tutti gli eventi che sono sotto tipo di questo Evento", ottenendo così un alto grado di espressività.

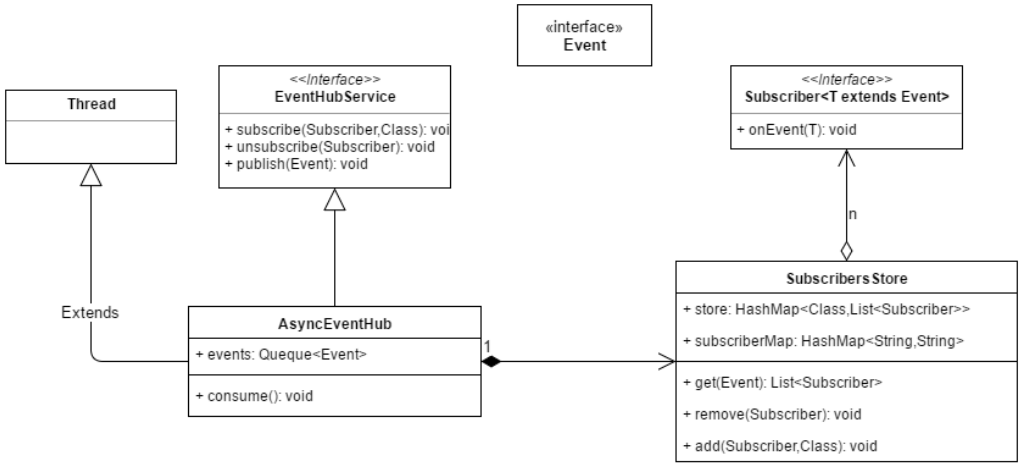

Figura 13 Diagramma uml delle classi costituenti di un EventHub

L'EventHub è un componete attivo di struttura simile a quella di un consumatore di messaggi. Perciò è stato implementato come un thread

java che continuamente ritira eventi da una coda e gli consegna ai servizi sottoscritti. Il riconoscimento del giusto ricevente avviene nella classe (Figura 13) con il seguente algoritmo:

- Alla sottoscrizione le istanze di *sofia.sib.eventhub.Subscriber* vengono aggiunte allo store associandole al tipo dell'evento a cui sono interessate in una mappa < TipoEvento, Lista sottoscrittori>
- All'arrivo dell'evento si richiedono gli oggetti interessati. Lo store in maniera iterativa accede alla mappa con il tipo dell'evento corrente poi lo sostituisce con quello dalla sua super classe. Il processo continua fino a raggiungere la classe *sofia.sib.eventhub.Event.*

Il meccanismo di sottoscrizione avviene in maniera automatica fruttando un altro pattern di utilizzo largamente usato in OSGi, modello a lavagna bianca [31]. Il principio propone di non chiamare direttamente i servizi al quale siamo interessati (e.s. chiamo direttamente la primitiva di sottoscrizione) ma di presentarci al framework e aspettare di venir interpellati. Tale funzionalità è semplice da ottenere grazi alle primitive di registrazione di servizi native, le quali sono state utilizzate nell'implementazione del EventHub. Come possiamo notare dal Codice 9 il servizio di smistamento eventi si registra per ricevere notifiche sul ciclo di vita di bundle di tipo *sofia.sib.eventhub.Subscriber.* All'evento di entrata nel sistema si incarica di sottoscriverle in modo automatico completando così il processo di sottoscrizione. Si nota dunque la differenza con il caso opposto nel quale il sottoscrittore debba manualmente chiamare il bundle EventHub. In questo approccio potrebbero verificarsi errori a causa della mancata presenza del servizio al momento di inizializzazione oppure anche a runtime nell'eventualità

di aggiornamenti del servizio. La strategia scelta invece evita tutto ciò poiché la registrazione è fatto dal servizio di eventi stesso e quindi ho la sicurezza della sua presenza nel sistema. Infine l'inversione della dipendenza permette la dichiarazione degli eventi in concomitanza con i bundle che li generano, creando un albero di relazioni chiaro tra chi produce e chi consuma le notifiche.

```
public class EventHubActivator implements BundleActivator, ServiceTrackerCustomizer<Subscriber,
ServiceRegistration<EventHubService>> {

	/*
	 * Dati privati
	 */
	public void start(BundleContext context) throws Exception
	{
		/*
		* Inizializzazione privata
		*/
		ServiceTracker<Subscriber, ServiceRegistration<EventHubService>> serviceTracker = new
ServiceTracker<Subscriber, ServiceRegistration<EventHubService>>(context,Subscriber.class,this);
		serviceTracker.open();
	}

	/*
	 * Registrazione nuovo sottoscrittore.
	 */
	private <T extends Event> void subscribeService(ServiceReference<Subscriber> arg0) {
		Subscriber<T> service = (Subscriber<T>) context.getService(arg0);
		Type[] types = service.getClass().getGenericInterfaces();
		for(Type t : types){
			if(t instanceof ParameterizedType){
				Class<T> subs = (Class<T>)
((ParameterizedType)t).getActualTypeArguments()[0];
				asyncEventHub.subscribe(service, subs);
			}
		}
	}

}
```

Codice 9 Registrazione nuovo sottoscrittore utilizzando il ciclo di vita dei bundle

Lastwill e KPI

Una volta aggiunto il comando di registrazione ultime volontà lato server, si è passati al lato cliente cioè KPI. La modifica è stata minima poiché è stato sufficiente inserire un metodo che creasse un messaggio di ultima volontà del formato specificato e lo inviasse attraverso le

primitive già esistenti su una socket creata ad hoc. Questa connessione sarà poi passata ad un thread creato dopo aver ricevuto la conferma dell'avvenuta registrazione. Il thread ha il compito di consumare i dati di controllo inviati sulla socket e quindi liberare il buffer TCP. Inoltre il meccanismo potrebbe essere esteso in modo da poter notificare l'utente in caso di mancata ricezione del carattere di ping dopo un certo time-out. La precedente funzionalità permetterebbe di verificare lo stato della connessione lato cliente e quindi, nel caso essa venga a mancare, attuare politiche di risparmio energetico, possibilità molto importante nell'ambito IoT.

Demo Lastwill

Per dimostrare le importanti ripercussioni dell'aggiunta di questo tipo di comando, è stata realizzata una semplice applicazione che simula un monitor che mostra all'utente lo stato di connessione di un drone. Per l'appunto il software è formato da due KP: uno è il monitor e l'altro è il drone. Quest'ultimo è un produttore e genera un'unica tripla che contiene il suo stato corrente (Codice 10). Inoltre specifica un lastwill contenente come messaggio l'update SPARQL mostrato nel Codice 10. Il KP monitor è invece un attuatore e si sottoscrive tramite la query mostrata nel Codice 11 .Utilizzerà tale informazione per visualizzare all'utente lo stato del drone: spunta verde se lo stato è "Online"; una croce rossa se è "Offline". Come si può notare dall'immagini il link tra SIB e drone è valido fin quando la finestra che fa le veci del drone viene chiusa. La chiusura è interpretata dal broker come una perdita di connessione. Questo attiva l'esecuzione delle ultime volontà del drone che imposteranno il suo stato a "Offline". Se una nuova istanza del KP drone verrà avviata questa cancellerà gli effetti del ultimo lastwill con

l'update e inserirà nuovamente le sue ultime volontà, ritornando così allo stato "Online". (Figure 1)

```
//Update
PREFIX dp: <http://drones/prop/>
DELETE { <http://drones/drone0> dp:status  "Offline" }
INSERT { <http://drones/drone0> dp:status  "Online" } WHERE {}

//Lastwill
PREFIX dp: <http://drones/prop/>
DELETE { <http://drones/drone0> dp:status  "Online" }
INSERT { <http://drones/drone0> dp:status  "Offline" } WHERE {}
```

Codice 10 SPARQL update e lastwill del KP drone

```
PREFIX dp: <http://drones/prop/>
SELECT ?status
WHERE { <http://drones/drone0> dp:status  ?status }
```

Codice 11 SPARQL query del KP monitor

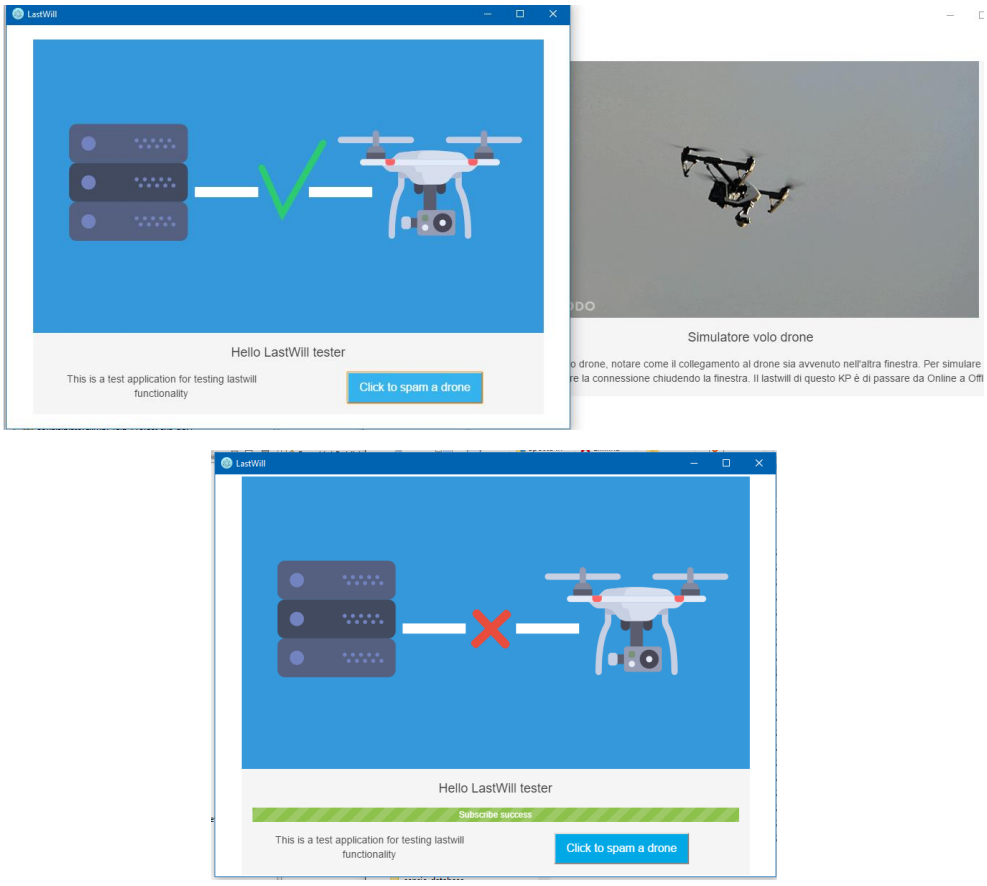

Figure 1 Immagini riprese dall'applicazione demo creata

Sottoscrizioni

Analizzando il codice della precedente versione del broker semantico, possiamo stabilire che il livello robustezza del servizio offerto dalle sottoscrizioni è di tipo 0, cioè sottoscrizioni semplici. Tale affermazione è supportata dalla constatazione dell'assenza di delle funzionalità che siano in grado di recuperare lo stato di connessione a fronte di un errore di rete. In tal eventualità la socket lato server si chiuderebbe (lanciando eccezione di IO) diventando così inutilizzabile per ulteriori invii di dati. Invece lato cliente la socket rimarrebbe per sempre in stato ricezione. Il risultato netto è quello che le notifiche non vengono più inviate e il cliente rimane ignaro della condizione di errore. Come abbiamo ampiamente discusso questa causalità è accettabile in certi ambiti applicativi, mentre, se consideriamo un ambiente più dinamico, con connessioni instabili e risorse limitate, sarebbe bene garantire almeno un livello di robustezza pari al 2. Per ottenere il suddetto requisito abbiamo prima ragionato su quale dei due parti in gioco dovesse essere modificata, conseguentemente abbiamo implementato il meccanismo di livello 1 e infine il 2, definendo un comportamento configurabile a seconda delle esigenze.

In un'applicazione cliente servitore vi è un'asimmetria nel dialogo fra le due parti. Infatti è il cliente che inizializza il dialogo mentre il servitore rimane passivo in ascolto di nuove richieste. A causa di ciò la maniera naturale di ripristinare la connessione tra le due parti è quella di incaricare il cliente di riattivare il contatto con la parte remota, poiché già conosce il suo indirizzo. Tale strategia inoltre permette di supportare la riconnessione automatica anche in versioni più datate di broker semantico, tuttavia ne restringe l'uso solo sulle KPI in cui è implementata. Riguardo alla memorizzazione dello stato non c'è una

scelta preferenziale se mantenerlo lato cliente o servitore, per questo è stato necessario attuare uno studio sei vantaggi e gli svantaggi dell'una o l'altra opzione. Ricordiamo che lo stato corrente del grafo associato ad una sottoscrizione è necessario al fine di garantire un livello di servizio pari al *Livello 2*; poiché al ripristino della sottoscrizione è necessario informare il cliente di cosa è cambiato rispetto all'informazione pregressa ottenuta dalle precedenti notifiche. La possibilità di mantenere lo stato lato cliente permette un'ottima scalabilità ma appesantisce il cliente, aggravandolo del compito di memorizzare l'evoluzione del grafo. L'altra scelta sarebbe meno scalabile poiché la tutti gli stati di tutte le sottoscrizioni di ogni KP di tipo aggregatore o attuatore dovranno essere mantenuti nella SIB. Ovviamente in questo caso i limiti di memoria sulla singola macchina potrebbero essere raggiunti molto presto. In particolare si può calcolare che la velocità con cui lo spazio di RAM viene occupato è data da N*M*K, con N numero di record medio per sottoscrizione, M numero di sottoscrizioni medio per KP, K numero di KP. Inoltre poiché lo stato deve essere mantenuto per tutto il tempo in cui il KP rimarrà scollegato, non è possibile stabile con certezza quando liberare risorse e quindi sarebbe necessario utilizzare meccanismi a time-out i quali generano problematiche (es. cosa succede se il KP si riconnette appena scaduto il time-out? Il servizio garantito sarebbe di tipo 1 poiché non è possibile ripristinare lo stato). Così è stata scelta la prima implementazione pagando però in risorse occupate su ogni KP. Tuttavia abbiamo proposto un design pattern per risolvere casi in cui si abbiano risorse limitate e si voglia comunque avere un livello di servizio fino al *Livello 3*.(*Sottoscrizioni full recovery: un design pattern*)

Il problema della riconnessione può essere visto con una macchina a stati, mostrata in (Figura 14). L'automa parte dallo stato connesso ne quale permane fin tanto che rileva un arrivo di notifiche. Quando non vengono più ricevuti caratteri di ping da un determinato periodo di tempo avviene l'evento di disconnessione che porta la macchina allo stato 1. In questa situazione si ritenta la riconnessione, cioè si cerca di inizializzare un dialogo tra il cliente e il servitore utilizzando la primitiva *connect*. Instaurata la comunicazione si passo allo stato 2, quello di sottoscrizione. In tal caso avviene una nuova richiesta di sottoscrizione per la query scelta. Se ha successo è possibili tornare allo stato connesso, altrimenti o il broker è caduto ed è stato riattivato perciò vado allo stato 3 oppure ho avuto di nuovo un errore di rete, quindi torno allo stato 1. Nel 3 l'automa si presenta nuovamente alla SIB con la primitiva di *JOIN*, poiché la sottoscrizione era fallita a causa della perdita di tale informazione dovuta al riavvio del server. A questo punto torna al 2 fin quando non avrà esito positivo e si porterà al 1. L'utilizzo dell'automa a stati finiti ha permesso di affrontare il problema in maniera formale e certificarne la funzionalità in uno scenario critico come la perdita di connessione.

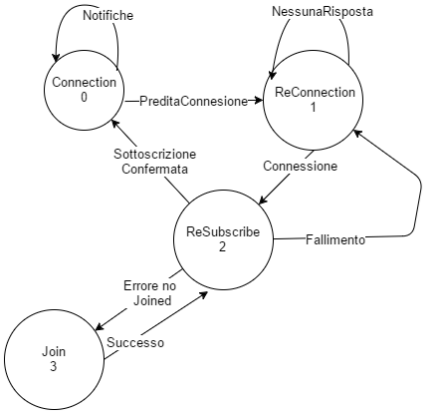

Figura 14 Automa a stati finiti di una riconnessione

L'implementazione di macchina a stati in ambito object oriented è una tecnica nota in letteratura e prende il nome di State design pattern [32] (Figura 15). È stata quindi definita la classe *sofia_kp.subscriptions.AutoSubcription* che estende thread e definisce il contesto dell'automa. Il thread chiamerà continuamente il metodo

handle dello stato corrente sostituendolo con quello ricevuto da esso, raggiunto lo stato speciale *EXIT* uscirà dolcemente.

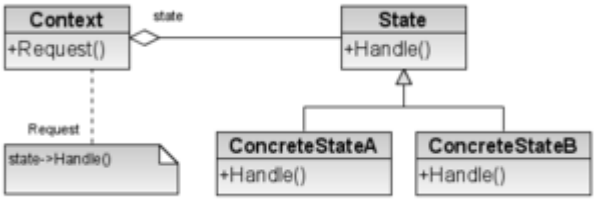

Figura 15 Tipica struttura delle classi usando il pattern State

La classe è mostrata nel Codice 12, oltre al comportamento descritto si può osservare che lo stato mantiene le seguenti informazioni:

- In_socket: la socket di connessione iniziale
- Hand: l'istanza del oggetto di ascolto notifiche definito dall'utente
- Address: l'indirizzo del server, utilizzato nella fase di riconnessione
- Query: la query associata a questa sottoscrizione, verrà utilizzata nello stato 2 per la riattivazione della sottoscrizione.
- Api: un'istanza delle KPI per poter utilizzare la funzionalità di JOIN.

Inoltre nel costruttore si imposta il valore di time-out livello sistema operativo per la socket in ascolto notifiche. Questo passaggio è necessario al fine di rilevare la caduta di connessione. Infatti in sua assenza il cliente rimarrebbe in ascolto perenne di notifiche o caratteri di ping, perciò grazie a questo, se dopo un determinato periodo di tempo non avessimo ricevuto né ping né notifiche potremmo rilevare che la connessione è caduta. L'utilizzo di questo meccanismo è rischioso poiché necessita di un tuning del tempo di attesa in modo tale da

verificare con precisione lo stato della connessione. Però nel caso di falsi positivi, cioè comunicazione rilevata assente ma in realtà attiva, l'unico danneggiamento è quello di passare inutilmente allo stato di riconnessione poi attivazione sottoscrizione, non inficiando assolutamente sulla correttezza del comportamento applicativo. Importante è ricordare che nello stato 1 la socket della sottoscrizione precedente viene chiusa comunicando al server (nel caso di connessione falsamente persa) di eliminarla. In tal modo si evitano sovrapposizioni di notifiche rimanendo trasparente al livello applicativo. Per questo il time-out è sembrato un meccanismo semplice da utilizzare ma comunque efficace ai fini dell'implementazione della funzionalità di rilevazione caduta connessione.

```
public class AutoSubscription extends Subscription {
	static SubscritpionState connection = new ConnectionState();
	protected SubscritpionState currentState;
	private final String query;
	private PrintWriter bufferedWriter;
	private final SocketAddress address;
	private iKPIC api;

	public AutoSubscription(Socket in_sock, iKPIC_subscribeHandler2
hand,SocketAddress address,String query,iKPIC api) throws IOException {
		super(in_sock, hand);
		this.address = address;
		this.query = query;
		this.setApi(api);
		in_sock.setSoTimeout(10000);
		setPrintWriter(new PrintWriter(new
OutputStreamWriter(in_sock.getOutputStream())));
		currentState = new ConnectionState();

	}

	@Override
	public void run() {
		while(!this.isInterrupted() ||
currentState.equals(SubscritpionState.EXIT)){
			currentState = currentState.handle(this);
		}
	}
	// Metodi di accesso stato
	// ......//

}
```

Codice 12 Classe base di una sottoscrizione con riconnessione

I vari stati dell'automa sono rappresentati da classi che implementano

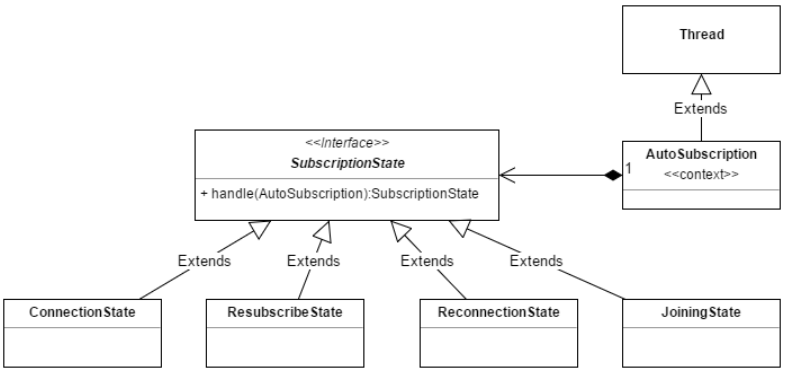

Figura 16 Uml della macchina a stati realizzata

l'interfaccia *sofia_kp.subscriptions.SubscriptionState* in ognuna delle quali il metodo *handle* è stato implementato seguendo il comportamento specificato (Figura 16). Osserviamo ad esempio lo stato di riconnessione (Codice 13). Come si può notare in via preliminare si chiude la socket corrente per le suddette motivazioni. Poi si tenta la connessione e in caso di errore si comunica al thread che si vuole rimanere sullo stato corrente. Solo se la socket è stata correttamente inizializzata e si è instaurato il dialogo TCP tra le due parti, si passa allo stato di riattivazione sottoscrizione. Infine citiamo anche che ogni stato è visibile solo all'interno del package di *sofia_kp.subscriptions* in questo modo si evita di esporre il dettaglio implementativo agli utenti della libraria KPI.

Essendo *sofia_kp.subscriptions.AutoSubcription* sottoclasse di *sofia_kp.Subscription* la funzionalità è completamente trasparente all'utente il quale dovrà semplicemente specificare il livello di servizio richiesto utilizzando la famiglia di metodi *subscribeX(query,handler,level)*. Inoltre specifichiamo che di default, per

mantenere la reto compatibilità, il metodo *subscribeX(query,handler)* garantisce il livello di servizio pari a 0.

```
class ReconnectionState implements SubscritpionState {

	@Override
	public SubscritpionState handle(AutoSubscription context) {
		SubscritpionState nextState = this;

		prepareNewConnection(context);
		Socket socket = context.getSocket();
		try {
			socket.connect(context.getAddress());
			BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
			PrintWriter printWriter = new PrintWriter(new
OutputStreamWriter(socket.getOutputStream()));
			context.setPrintWriter(printWriter);
			context.setReader(bufferedReader);
			nextState = new ReSubscribeState();
		} catch (IOException e) {
			nextState = this;
		}
		return nextState;
	}

	private void prepareNewConnection(AutoSubscription context) {
		Socket socket = context.getSocket();
		if(!socket.isClosed()){
			try {
				context.setSocket( new Socket() );
				socket.close();
			} catch (IOException e) {
				e.printStackTrace();
			}
		}
	}

}
```

Codice 13 Stato riconnessione

Per garantire il secondo livello di servizio è stato necessario estendere la macchina a stati definita pocanzi modificando lo stato numero 2 e aggiungendo un handler speciale che tenesse traccia del contesto della query SPARQL o RDF. L'estensione è stata fatta utilizzando il pattern

proposto da Brian Chin e Todd Millstein [33], il quale ha permesso un ottimo riutilizzo del codice sviluppato per la sottoscrizione con riconnessione a fronte di piccole modifiche. Per seguire tale metodo di sviluppo è stato modificato l'automa precedente in modo da prevedere dei metodi factory che creassero gli stati nella classe base *sofia_kp.subscriptions.AutoSubcription* poi in ogni stato son stati utilizzati per la creazione dei prossimi (`nextState = context.ReSubscribeState();`). Questa tecnica ha permesso l'implementazione di una nuova macchina a stati *sofia_kp.subscriptions.AutoStateFullSubscription* la quale ereditando dalla prima ridefinisse il metodo per la creazione dello stato di riattivazione sottoscrizione. Infatti è in quello che andrà cambiata la politica scelta, poiché dovrà ricevere lo stato corrente del grafo e confrontarlo con quello pregresso salvato. Il grafo viene memorizzato incrementalmente da uno speciale handler che contiene quello definito dall'utente, invia tutte le notifiche ad esso ma allo stesso tempo tiene traccia delle modifiche fatte. Infatti viene inizializzato con la risposta della query di sottoscrizione la quale può essere vista come un insieme con duplicati di righe di una tabella e per ogni evento inviato dalla SIB va a togliere o aggiungere righe ad essa. Solitamente i risultati di query SPARQL e RDF sono tratti in maniera diversa poi i primi sono una lista di valori associati alle variabili della query mentre i secondi sono triple. Al fine però di tener traccia di questi valori essi possono essere appunti visti come record in una tabella, la quale in RDF avrà sempre 3 colonne fisse identificanti soggetto predicato oggetto, mentre in SPARQL le colonne saranno di numero diverso a seconda delle incognite definite. Inoltre la query RDF non contempla duplicati poiché ogni tripla è univoca all'interno del grafo mentre le SPARQL possono contenerli, poiché una

variabile potrebbe essere associata ad uno stesso valore più volte. La gestione di tutte queste casistiche è affidata ad una classe *sofia_kp.subscriptions.context.SubscriptionContext* (Figura 17).

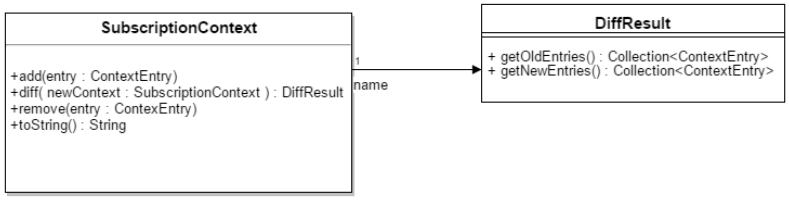

Figura 17 UML delle classi coinvolte nella gestione del contesto

Esso è agnostico rispetto alla query utilizzata poiché, appunto, tratta i dati come record di una tabella. Inoltre questa classe è grado di calcolare le differenze tra due tabelle comunicando quali sono le righe obsolete e quali sono state aggiunte. Come vedremo questa possibilità sarà fondamentale per poter comunicare in maniera trasparente al lato applicativo il nuovo stato attraverso il meccanismo delle notifiche.

Le ipotesi per il corretto funzionamento dell'algoritmo per il calcolo della differenza tra due *SubscriptionContext* sono che:

- Per ogni inserimento di un record venga tenuto traccia del numero di righe uguale a quella data.
- Per ogni rimozione del record venga decrementato di 1 il numero di righe uguale a quella eliminata
- Se il *SubscriptionContext* contiene dei duplicati allora dovrà avere un flag che identifica la loro presenza.
- Se il *SubscriptionContext* contiene dei duplicati allora deve essere possibile in maniera efficiente selezionare solo le righe duplicate.

Per ottenere queste condizioni inziali e per ragioni di efficienza, sono state utilizzati una mappa dati – numero righe uguali e un insieme di righe contenente solo quelle duplicate. Le due strutture dati sono

mantenute coerenti dai metodi di aggiunta e rimozione record e, poiché sono implementate come HashMap e HashSet, i tempi di accesso sono costanti nel tempo. Date le precedenti ipotesi l'algoritmo esegue i seguenti passi:

1. Ottiene l'insieme delle chiavi delle mappe dei due contesti, siano ***chiavi1***(vecchio contesto) e ***chiavi2*** (nuovo contesto)
2. Calcola intersezione insiemistica tra ***chiavi1*** e ***chiavi2***, sia ***intersezione***.
3. Sottrae da chiavi1 l'intersezione ottenendo una prima lista delle righe obsolete chiamiamole: ***obsolete***.
4. Sottrae da chiavi2 l'intersezione ottenendo una prima lista delle righe aggiunte rispetto all'informazione precedente, chiamiamole: ***nuove***
5. Se uno dei due contesti contiene duplicati (il flag ha valore vero)
    a. Siano ***vecchiD*** e ***nuoviD*** i due insiemi delle righe duplicate contenute nel vecchio e nel nuovo contesto
    b. Per ogni record presente in ***vecchiD*** e in ***obsolete*** si moltiplica l'istanza di esso del numero di occorrenze indicato dalla mappa dati – numero righe uguali contenuta nel vecchio contesto. Questa fase gestisce casi in cui sono state eleminati tutti record di quel tipo dal grafo.
    c. Per ogni record presente in ***nuoviD*** e in ***nuove*** si moltiplica l'istanza di esso del numero di occorrenze indicato dalla mappa dati – numero righe uguali contenuta nel nuovo contesto Questo per gestire casi in cui sono stati aggiunti più record uguali ne grafo.

d. Per ogni elemento in ***intersezione*** guardo il vecchio numero di occorrenze e il nuovo sfruttano le due mappe dei contesti. Se ci sono differenze aggiungo il giusto quantitativo di record in ***obsolete*** o ***nuove***. Esempio: se per il Record R avevo 3 occorrenze e ora ne ho 1 allora aggiungo due istanze di Record R a ***obsolete***. Quindi in questa fase vengo coperti i casi in cui il numero di record memorizzati prima della disconnessione è diverso da quello dopo. Un metodo alternativo sarebbe quello di fare l'intersezione tra ***nuoviD*** e ***vecchiD*** per poi verificare il numero dell'occorrenze solo sul risultato di tale operazione. Questo potrebbe dare un vantaggio computazione solo se i record duplicati nei due contesti sono minori del numero dei record condivisi. Poiché nel caso di record duplicati la probabilità che siano contenuti nell'intersezione è alta, non si ottiene il beneficio sperato.

6. Restituisco nuove e obsolete.

In Figura 18 si possono vedere le varie operazioni insiemistiche. Si noti come la gestione dei duplicati appesantisce l'algoritmo poiché necessita almeno di ulteriori N e M confronti, con N numero di record contenuti nell'insieme dei vecchi duplicati e M numero di record all'interno dell'intersezione. Per tale motivo l'algoritmo è stato studiato in modo da attivare questa funzionalità solo al bisogno tramite l'utilizzo del flag apposito. A questo punto in ***obsolete*** e ***nuove*** ho le informazioni per

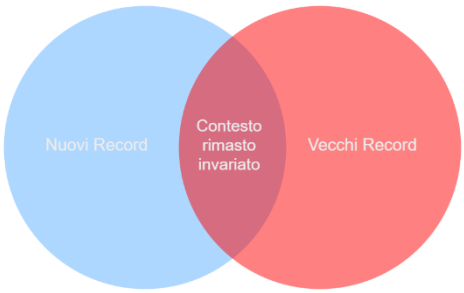

Figura 18 Visione insiemistica dei contesti della vecchia sottoscrizione a confronto con la nuova

creare una notifica incrementale dello stato del grafo. La quale viene creata nel nuovo stato di riattivazione sottoscrizione rappresentato dalla classe *sofia_kp.subscriptions.FullResuscribeState*. In essa, una volta confermata la sottoscrizione e ricevuto il grafo corrente, si costruisce il nuovo contesto andando ad inserire tutti i record aggiornati. Dopodiché si utilizza il metodo *diff* dall'istanza del vecchio contesto e si ottengono, appunto, le liste contenenti le nuove e le eliminate righe di tabella. Esse verranno poi trasformate in liste di triple nel caso la query fosse RDF o in lista di variabili con valori associati per le SPARQL. Ciò ci permette di informare il lato applicativo in maniera trasparente poiché all'atto di riconnessione viene notificato con solo le differenze rispetto a quello che aveva già registrato. Supponendo, ad esempio, l'applicazione di monitoraggio risorse suggerita nella trattazione teorica (Sottoscrizioni). Nel caso non ci sia alcun cambiamento nella condizione dei dispositivi d'interesse, durante la caduta di connessione, la GUI non verrebbe notificata ma se lo stato fosse cambiato la GUI riceve la notifica solo della risorsa con condizioni diverse da quelle precedentemente mostrate all'utente. Anche in questo caso l'attivazione di una sottoscrizione con recupero di contesto avviene tramite la funzione *subscribeX(query,handler,2).*

Concludendo abbiamo esposto come è stato possibile ottenere il livello di robustezza pari a 2 partendo incrementalmente dalla versione precedente del KPI. Durante lo sviluppo abbiamo avuto cura di mantenere la retro-compatibilità con le vecchie versioni SIB riducendo le modifiche di protocollo a zero. Inoltre le varie funzionalità aggiunte sono state progettate in modo da rendere il lato applicativo completamente agnostico riguardo al meccanismo utilizzato. Infine si è scelto di permette lo sviluppa di soluzioni software con diversi livelli di garanzia

configurando il livello di robustezza delle sottoscrizioni tramite l'utilizzo dei metodi offerti dalla classe *KPICore*.

Sottoscrizioni full recovery: un design pattern

Nonostante l'implementazione del livello 2 copra un buon numero di casi applicativi non ne copre la totalità. Per di più la soluzione proposta incarica il lato cliente di computazioni complesse e richiede che sia disponibile un spazio ragionevole di memoria. Nelle applicazioni in cui sia necessario un livello di robustezza maggiore di quello a ricostruzione di contesto oppure in cui non si abbiano a disposizione abbastanza risorse computazionali lato cliente, proponiamo l'utilizzo di un design pattern sviluppato per affrontare tale problematica. Esso sfrutta i meccanismi più semplici definiti precedentemente per un comportamento più complesso quale quello di sottoscrizione full recovery e full time recovery. Secondo la definizione:

> *In informatica, nell'ambito dell'ingegneria del software, un design pattern (traducibile in lingua italiana come schema progettuale, schema di progettazione, schema architetturale), è un concetto che può essere definito "una soluzione progettuale generale ad un problema ricorrente". Si tratta di una descrizione o modello logico da applicare per la risoluzione di un problema che può presentarsi in diverse situazioni durante le fasi di progettazione e sviluppo del software, ancor prima della definizione dell'algoritmo risolutivo della parte computazionale. [34]*

Chiariamo formalmente quale problema ricorrente vogliamo risolvere utilizzando questo pattern: è necessario che se un KP venga scollegato dalla rete alla sua riconnessione vogliamo che riceva tutti gli eventi accaduti mentre era assente. Inoltre specifichiamo che utilizzeremo unicamente concetti presenti nel modello applicativo SMART M3.

Come spesso accade creiamo un esempio pratico per spiegare in cosa consiste il pattern proposto. Pensiamo ad un'applicazione formata da due KP un produttore e un attuatore. Immaginiamo che il produttore invii dei comandi ad un robot. I comandi sono formati dalla direzione e dal tempo di durata. Quindi un comando potrebbe essere: *avanti,10s* e il risultato atteso è quello che il robot si muova avanti per 10 secondi. Non vogliamo che nessuno dei comandi venga perso dal robot ma sappiamo che ci potrebbero essere delle perdite di connessione con la SIB dovute all'utilizzo di un network Wi-Fi non affidabile. Imponiamo anche che il KP inserisca i comandi in questo modo: Inserisce il comando, aspetta 1 secondo, lo cancella. Tale metodo di inserimento è curioso e sicuramente il KP potrebbe anche inserire tutti i comandi assieme nel grafo, modellandoli ognuno come una tripla diversa. Immaginiamo, però, che stiamo utilizzando un telecomando robot "legacy" e non possiamo modificare questo comportamento. Infatti serve solo al fine di modellare una casistica specifica nel quale gli eventi interessanti sono costituiti da modifiche volatili del grafo.

Ora il livello 2 non ci garantisce il recovery di tutti gli eventi accaduti mentre il KP attuatore era offline. Alla riconnessione, appunto, verremo informati solo dell'ultimo comando inserito (se esso è ancora presente) o non riceveremo nessun comando (nel caso fosse già stato eliminato). La strategia che possiamo utilizzare è quella di utilizzare lastwill più sottoscrizione con riconnessione senza recupero di contesto.

L'idea è quella di modificare il sistema come in Figura 19. Quello che è stato fatto è aggiungere un KP (fantasma) aggregatore sostituto dell'attuatore sul nodo della SIB. La presenza del fantasma sullo stesso nodo della SIB permette di escludere eventi come la disconnessione eliminando la ricorsione del problema (dovremmo progettare il sostituto

del sostituto e così via). Il fantasma quando viene attivato si sottoscrive allo stesso grafo in cui vengono scritti i comandi dal KP produttore, poi aggiorna un particolare sotto grafo protetto in scrittura, in modo da essere il solo a potervi scrivere. Qui inserisce la lista dei comandi ricevuti fino a che non venga disattivato. L'attivazione e la disattivazione avviene grazie al meccanismo di lastwill del KP attuatore. Esso, all'ingresso nel sistema, disattiva il fantasma inserendo una determinata tripla nel grafo e si sottoscrive tramite riconnessione semplice ai comandi inviati dal KP produttore. Come ultime volontà, invece, inserisce la tripla duale comunicando l'attivazione del fantasma. Alla riconnessione la sottoscrizione viene automaticamente riattivata, dopodiché il KP robot disattiva il fantasma con l'inserimento dalla tripla di prima e fa una query nel sotto grafo ne quale il sostituto pubblicava i comandi. Dopo una fase di eliminazione dei duplicati, nel caso in cui siano arrivate notifiche sulla sottoscrizione riattivata mentre si faceva la query, il KP è di nuovo connesso e ha ricevuto tutti gli eventi accaduti mentre non era online. (Codice 14)

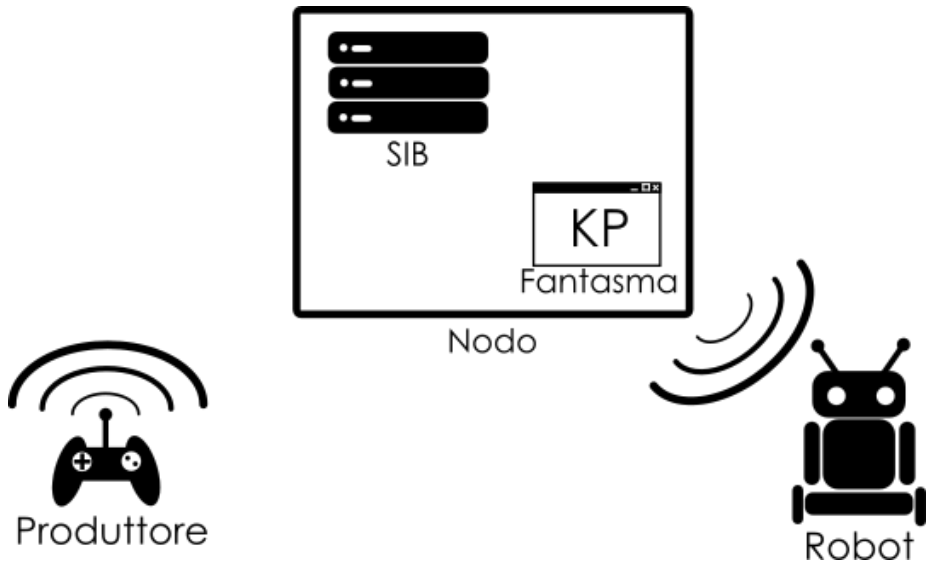

Figura 19 Struttura del sistema

Formalmente il design pattern consiste nel:

1. inserire un KP aggregatore fantasma sullo stesso nodo della SIB
2. Definire il formato di tripla per l'attivazione e la disattivazione del fantasma
3. Specificare un formato per il salvataggio degli eventi
4. Proteggere il grafo che contiene le notifiche da scritture esterne
5. Sottoscrivere il fantasma alla stessa query della quale si vuole un livello di robustezza superiore al 2.
6. L'attuatore deve produrre la tripla attivazione/disattivazione ogni qualvolta si connette o si disconnette. Per far ciò deve definire un lastwill
7. L'attuatore si sottoscrive con una semplice sottoscrizione con riconnessione
8. In caso di caduta e ritorno di connessione l'attuatore deve:
    a. Disattivare il fantasma
    b. Fare una query al sotto grafo nel quale sono pubblicati gli eventi persi
    c. Rimuovere i duplicati nel caso siano arrivate notifiche sulla sottoscrizione riattivata

Vediamo ora come questo pattern può essere utilizzato anche quando si affronta la seconda problematica: un KP necessità di un livello di robustezza sottoscrizioni superiore o uguale al due ma non ha le risorse computazionali tali da rendere inutilizzabile le sottoscrizioni con recupero del contesto. L'utilizzo del fantasma permette di decentralizzare il carico computazionale e di memoria dal KP remoto. Inoltre il pattern riesce a supportare il formalismo delle sottoscrizioni di tipo 3 quindi anche superiore a quello di recupero di contesto. Per dipiù impiegando il KP fantasma si ottiene una buona scalabilità, dipendente

dal numero di processi gestibili dalla macchina in cui è installata la SIB (il quale potrebbe essere molto elevato se installata su cluster o in server farms).

Come avevamo introdotto la metodologia proposta usa in modo intelligente le funzionalità di base aggiunte alla SIB. Attraverso esse risolve problematiche di alto livello grazie a delle politiche di buona progettazione. Infatti senza la presenza di una delle due questo comportamento non sarebbe raggiungibile, aggiungendo quindi un ulteriore rilevanza al lavoro svolto.

```
// Update d'inziallizazzione
// ogni voltà che il kp torna online inserisce
// queste triple
PREFIX fp: <http://fantasma/>
DELETE {
       fp:Fantasma0 fp:haStato "ATTIVO"
}
INSERT {
       fp:Fantasma0 fp:haStato "DISATTIVO"
}
WHERE { }

// LastWill che avvia il fantasma
PREFIX fp: <http://fantasma/>
DELETE {
       fp:Fantasma0 fp:haStato "DISATTIVO"
}
INSERT {
       fp:Fantasma0 fp:haStato "ATTIVO"
}
WHERE { }

// Query di sottoscrizione fatta dal fantasma.
// A seconda del valore di command attiva o disattiva
// la prossima query
PREFIX fp: <http://fantasma/>
SELECT ?command
WHERE {
       fp:Fantasma0 fp:haStato ?command
}

// Query di sottoscrizione fatta dal KP robot.
// Questa query viene attivata/disattava dal
// fantasma ogni volta che è richiesto.
PREFIX fp: <http://fantasma/>
PREFIX tp: <http://telecomando/>
SELECT ?azione ?durata
WHERE {
       fp:telecomando0 tp:haInviatoComando ?command.
       ?comando tp:haTipo ?azione.
       fp:telecomando0 tp:haDurata ?durata.

}

// Insert di una notifica della SIB da parte del
// KP fantasma. Ogni nuovo comando è indentificato
// da un valore incrementale sostituito a X. TIPO
// e DURATA sono valori parametrici presi dalla query di
// sottoscrizione
PREFIX fp: <http://fantasma/>
INSERT {
       fp:comandoX fp:haTipo TIPO.
       Fp:comandoX fp:haDurata DURATA
}
WHERE { }

// Query fatta dal robot per ricevere tutte le notifche salvate
PREFIX fp: <http://fantasma/>
SELECT ?command ?durata ?tipo
WHERE {
       ?command  fp:haTipo ?tipo.
       ?command fp:haDurata ?durata
}
```

Codice 14 Varie primitive SPARQL inerenti al design pattern KP fantasma

Impatto sviluppo applicativi

L'introduzione del lastwill ha permesso di rivalutare alcune delle considerazioni fatte nella precedente trattazione [2]. Infatti uno dei punti in cui il framework SMART M3 è stato carente rispetto a quello di MQTT è stato la semplicità di sviluppo. Nonostante alcune affermazioni siano ancora del tutto valide, come ad esempio la difficoltà nel comprendere la presentazione dei dati fornita dalla KPI, possiamo affermare che il meccanismo introdotto riduce in modo significativo il numero delle righe di codice necessarie per ottenere le stesse funzionalità. Riprendendo lo studio sulle LoC presentato in Tabella 1, si può vedere che l'impatto più significativo è dato proprio dalla verifica dello stato del cliente. Questo controllo era necessario per ottenere informazioni riguardo al grado di funzionamento del dispositivo, ad esempio se il drone fosse ancora attivo o se gli irrigatori erano funzionanti. Come mostrato anche dall'implementazione MQTT questo requisito è stato ampiamente soddisfatto utilizzando il concetto di LastWill. Se proponiamo l'utilizzo di esso anche in SMART M3 le linee di codice scendono a 4 per un totale di 39(Tabella 2).

CONNECT **Client a Broker**	**DISCONNECT** **Cliente dal broker**	**SUBSCRIPTION**	**PUBLISH**	**RILEVARE LO STATO DEL CLIENTE**	**TOTAL LoC**
SET-UP Client: 4 CONNECT:6	SET-UP Client: 4 CONNECT: 6	SUBSCRIBE: 4 Linee medie per manipolare i risultati: 1	PUBLISH: 1 Linee medie per preparare i dati: 9	Reagire al problema: 25	60
10 loc	10 loc	5 loc	10 loc	25 loc	

Tabella 1 Linee di codice necessarie senza lastwill

CONNECT Client a Broker	DISCONNECT Cliente dal broker	SUBSCRIPTION	PUBLISH	RILEVARE LO STATO DEL CLIENTE	TOTAL LoC
SET-UP Client: 4 CONNECT:6	SET-UP Client: 4 CONNECT: 6	SUBSCRIBE: 4 Linee medie per manipolare i risultati: 1	PUBLISH: 1 Linee medie per preparare i dati: 9	Set-up: 1 Reagire al problema: 3	39
10 loc	10 loc	5 loc	10 loc	4 loc	

Tabella 2 Linee di codice con lastwill

Come ci aspettavamo data la maggior espressività di un sistema con dichiarazione ultime volontà il numero di linee di codice necessario a risolvere lo stesso problema è minore. Notiamo, infine, che ora la differenza tra quelle utilizzate per sviluppare un'applicazione MQTT e quelle con l'uso di SMART M3 è trascurabile, mentre per quanto riguarda le performance non ci sono state modificazioni rispetto al lavoro precedente (Tabella 2 Tabella 3).

CONNECT Client a Broker	DISCONNECT Cliente dal broker	SUBSCRIPTION	PUBLISH	RILEVARE LO STATO DEL CLIENTE	TOTAL LoC
SET-UP Client: 5 CONNECT:5	SET-UP Client: 5 CONNECT: 5	SUBSCRIBE: 1 Linee medie per manipolare i risultati: 10	PUBLISH: 1 Linee medie per preparare i dati: 5	Set-up: 1 Reagire al problema: 4	42
10 loc	10 loc	11 loc	6 loc	5 loc	

Tabella 3 Linee di codice in MQTT con lastwill

Conclusione

La tesi ha raggiunto i quattro obbietti principali preposti: l'aggiunta della funzionalità di lastwill; dimostrato come essa garantisca robustezza a soluzioni software che la utilizzino; definito nuove politiche di sottoscrizioni robuste; migliorato la modularità del broker semantico OSGi e definito un nuovo tipo di interazione tra i moduli basato su eventi.

In particolare è stato descritto come il lastwill possa garantire, in scenari critici, fault tollerance e degradazione dolce delle funzionalità applicative. Inoltre è stato messo in evidenza come la versione implementata in SMART M3 abbia un'espressività maggiore rispetto a quella in MQTT, garantendo la consistenza dello stato anche in scenari proibitivi.

Riguardo la funzionalità sottoscrizione, poiché le disconnessioni improvvise causavano a livello applicativo una perdita di sincronia, le aggiunte introdotte hanno permesso la ricostruzione del contesto dopo il transitorio di connessione. Questo ha conferito una maggiore solidità del sistema e degli applicativi che ne facessero uso.

La ricostruzione architetturale del broker semantico attraverso l'impiego del principio di inversione della dipendenza ha consentito un notevole speed up dello sviluppo e ha reso più coerente le relazioni tra i vari moduli. Inoltre combinato al nuovo paradigma di programmazioni ad eventi facilità future modifiche o aggiunte alla SIB OSGi tramite un approccio modulare.

Infine i cambiamenti proposti, come descritto nel capitolo *Impatto sviluppo applicativi*, hanno colmato il divario con MQTT dal punto di vista di righe di codice necessarie alla programmazione. Infatti

l'implementazione di Agri-Eagle con SMART M3 richiedere, ora, solamente 39 linee contro le 42 richieste dal framework concorrente.

Un proseguimento del presente lavoro di tesi potrebbe essere volto all'implementazione all'interno nel broker semantico di sottoscrizioni con riconnessione e ripristino storico come pure il livello più generale di robustezza individuato, cioè quello di riconnessione e ripristino storico temporale dello stato. Sarebbe inoltre necessario valutare la possibilità di configurare un timer per l'heartbeat del meccanismo di registrazione ultime volontà in modo da poter seguire diverse esigenze. Infine, seguendo le indicazioni descritte in *Appendice: Studio robustezza sottoscrizioni con singola socket*, è consigliabile la modifica del protocollo applicativo SSAP in modo che per ciascun KP sia utilizzata un'unica connessione in caso di sottoscrizioni multiple.

Appendice: Studio robustezza sottoscrizioni con singola socket

Correntemente l'unico protocollo di trasporto supportato dalla SIB OSGi è il protocollo TCP. In particolare viene sfruttato per la definizione del seguente protocollo applicativo:

- Per ogni richiesta (Query,Update,LastWill etc..) viene aperta una connessione tra KP e SIB.
- Al soddisfacimento della richiesta viene inoltrato il messaggio di risposta sulla stessa connessione.
- Se la connessione era stata aperta da una richiesta di tipo Subscribe o Lastwill essa rimane aperta per l'invio di notifiche da parte della SIB.
- Altrimenti la connessione viene chiusa.

Quindi in un dato momento nel sistema SIB-KPs sono presenti $M_0 + \cdots + M_{N-1}$ connessioni TCP aperte, con N numero dei Kp di tipo attuatore o aggregatore e M_i numero di sottoscrizioni per l'i-esimo Kp. Questa scelta implementativa richiede però un utilizzo di risorse elevato, quindi possibilmente non adatta in un contesto iot, soprattutto considerando l'overhead richiesto lato cliente nella gestione di M_i connessioni TCP.

Per tali motivazioni viene proposta una soluzione alternativa che modifica il protocollo nel seguente modo:

- Per ogni richiesta (Query,Update,LastWill etc..) viene aperta una connessione tra KP e SIB.
- Al soddisfacimento della richiesta viene inoltrato il messaggio di risposta sulla stessa connessione.
- Se la connessione era stata aperta da una richiesta di tipo Subscribe o Lastwill:
    - Se è la prima connessione di quel nodo essa viene mantenuta e registrata come canale comunicativo tra KP e Sib
    - Nel caso contrario viene chiusa.
- Altrimenti la connessione viene chiusa.

Valutiamo le due soluzioni secondo i seguenti aspetti: risorse impiegate, costo computazionale, affidabilità e troughtput.

Risorse

Come introdotto prima, la soluzione corrente prevede $M_0 + \cdots + M_{N-1}$ connessioni TCP aperte contro le *N* di quella proposta. Ogni connessione TCP è definita univocamente da una coppia di socket e ad ognuna di esse vengono allocate risorse di sistema per il loro corretto funzionamento. In

particolare vengono allocati: dei buffer per la ricezione e per l'invio dei dati, file descriptor per l'indirizzamento di tali buffer, strutture dati per il controllo di flusso e di congestione, stato della connessione e timer per l'affidabilità come il RoundTripTime e RetrasmissionTimeOut che sono fondamentali per garantire il rinvio dei pacchetti e accertarsi che il pacchetto sia stato ricevuto [35]. A queste si aggiungo le strutture dati necessarie a livello applicativo per la gestione di più connessioni. Esse consistono in una lista lato servitore nella quale son contenute tutte le socket e di M_i thread lato cliente.

Concludendo le risorse totali occupate dalla soluzione attuale lato servitore sono:

$$R_0^S = (M_0 + \cdots + M_{N-1}) * K + L$$

lato cliente invece sono:

$$R_0^C = M_i * K + M_i * T$$

Con

- *K* risorse richieste da una singola connessione TCP
- *L* risorse occupate dalla lista
- T risorse utilizzate per thread

Mentre le risorse utilizzate dalla soluzione proposta sono lato servitore:

$$R_1^S = N * K + L$$

lato cliente:

$$R_0^C = 1 * K + T\ .$$

La nuova modifica del protocollo quindi diminuisce in generale il consumo di risorse utilizzato nel sistema SIB-KPs poiché $\sum_{i=0}^{N-1} M_i \geq N$ quindi $R_0^S \geq R_1^S$ e vale anche $R_0^C \geq R_1^C$.

Costo computazionale

Le due soluzioni differiscono sensibilmente anche per il carico computazionale richiesto ai due lati della connessione. Infatti all'atto della connessione e alla trasmissione dei dati il livello tcp richiede dei calcoli per il funzionamento corretto del protocollo. Inoltre anche lato applicativo è richiesto del tempo di CPU per il processamento e la gestione delle connessioni nel sistema. Possiamo però trascurare il calcolo dovuto all'invio di pacchetti sulla rete poiché solitamente è piccolo e comunque è comune ad entrambe le soluzioni. L'operazione di connessione, invece, è considerata in generale un'operazione dispendiosa per il protocollo TCP poiché, durante la connessione è richiesto il calcolo del RTT [36], avviene il three-way-handshake, l'allocazione delle risorse e la loro inizializzazione [35]. Definendo questo costo come C_{tcp} le due soluzioni lato cliente richiedono:

$$C_0^c = M_i * C_{tcp}$$

$$C_1^c = C_{tcp}$$

Poiché il servitore deve periodicamente testare la presenza del sottoscrittore, al fine di non lasciare occupate risorse nella SIB e per supportare il meccanismo di lastwill, il numero di cicli richiesto dalla gestione lato applicativo è dato da:

$$C_0^s = \sum_{i=0}^{N-1} M_i$$

$$C_1^s = N$$

Similmente al caso delle risorse occupate, anche in questo aspetto il nuovo metodo implementativo permette di migliorare il sistema poiché $C_0^c \geq C_1^c$ e $C_0^s \geq C_1^s$.

In questo caso considereremo una scelta implementativa tanto più affidabile tanto più garantisca con maggiore probabilità il funzionamento di una particolare funzione applicativa, in presenza di malfunzionamenti di rete. La funzionalità che studieremo per le sottoscrizioni è quella di invio di una o più notifiche da SIB ad un singolo KP. Entrambe le soluzioni sono egualmente inaffidabili, in mancanza di ulteriori meccanismi, nel caso in cui il link tra SIB e KP venga interrotto (i.e caduta di un router a livello ip). Infatti alla caduta del nodo il layer tcp non riuscirà a inviare il segmento dati e chiuderà la connessione con errore. Nel caso attuale l'errore avverrà in sequenza per tutte le connessioni registrate tra SIB e Kp mentre in quella proposta verrà chiusa solo una connessione. Il risultato è quindi la de-allocazione della sottoscrizione e l'impossibilità di mandare ulteriori notifiche al KP causando mancanze nelle funzioni dell'applicazione.

Invece le due scelte implementative si differenziano nei casi in cui la connessione TCP sussiste su un network con alta variabilità del RTT e/o disconnessioni spurie, ciò è dovuto a temporanee perdite di pacchetti causate da alta congestione o inaffidabilità del link layer. Consideriamo ancora come funzionalità l'invio di notifiche da SIB a KP ma specifichiamo la differenza tra KP che necessitano di tutte le sottoscrizioni attive per il loro corretto funzionamento da quelli che hanno connessioni ridondanti o opzionali.

Prima di analizzare entrambe le casistiche specifichiamo che in TCP ogni connessione è un'entità indipendente dalle altre anche se si riferisce agli stessi nodi [37] [38] [24]. Ciò implica che il calcolo del RTT e di conseguenza del RTO viene eseguito per ogni connessione. Inoltre ricordiamo che in questo protocollo di trasferimento una connessione è

considerata persa se viene superato il RTO per tre volte consecutive, cioè l'host ricevente non ha confermato con un ACK i pacchetti inviatogli dopo che RTO è scaduto tre volte di seguito [35] [36].

In ogni caso andremo quindi a valutare l'affidabilità della scelta progettuale calcolando la probabilità che il sistema formato dalle varie connessioni rispetti la funzionalità richiesta, cioè che ci sia possibilità di comunicazione tra SIB e KP. Definiamo quindi questa probabilità come $P_{aff} = P(Y = 0)$ con Y variabile binomiale che se è 0 identifica che la connessione è attiva e se 1 invece indica una connessione persa.

Caso KP che necessitano di tutte le sottoscrizioni attive. In questa eventualità possiamo calcolare la probabilità che la soluzione attuale sia affidabile come quella di un sistema in serie, poiché la compromissione di una singola connessione significa la perdita della funzionalità sotto studio. Quindi possiamo definire la probabilità che il sistema non funzioni correttamente come [39]

$$P0_e = 1 - P[(Y_0 = 0) \cap \dots \cap (Y_{N-1} = 0)]$$

Le variabili Y_i possono essere considerate come indipendenti poiché il RTO è calcolato su ogni connessione indipendentemente. Questa approssimazione potrebbe non essere vera nel caso il traffico generato da esse crei intasamento e quindi influisca sul calcolo in ognuna. In generale però TCP gestisce automaticamente questi casi con i suoi meccanismi di controllo di flusso e congestione quindi possiamo scrivere:

$$P0_e = 1 - \prod_{i=0}^{N-1} P(Y_i = 0)$$

Mettiamola a confronto con la probabilità che la soluzione proposta non funzioni correttamente:

$$P1_e = 1 - P_{aff}$$

In prima battuta effettuiamo un'approssimazione cioè che le probabilità di affidabilità delle connessioni considerate siano uguali a P_{aff}. Allora la probabilità che il sistema implementato utilizzando la gestione delle connessioni attuale non funzioni è maggiore di quella del sistema dopo la modifica proposta, cioè

$$P0_e \geq P1_e$$

Dimostrazione:

$$P_{aff}{}^n \leq P_{aff}$$

$$-P_{aff}{}^n \geq -P_{aff}$$

$$1 - P_{aff}{}^n \geq 1 - P_{aff}$$

$$P0_e \geq P1_e$$

c.v.d.

Questo è valido solo nella semplificazione considerata cioè che $\prod_{i=0}^{N-1} P(Y_i = 0) = P_{aff}{}^n$. Vediamo ora nel caso generale. Per prima cosa consideriamo come probabilità di affidabilità della soluzione proposta una a caso tra quelle delle varie connessioni

$$P_{aff} = P(Y_\alpha = 0)$$

$$0 \leq \alpha < N$$

$$\prod_{i=0}^{N-1} P(Y_i = 0) \leq P(Y_\alpha = 0) = P_{aff}$$

Se vale la relazione proposta attraverso lo stesso approccio dimostrativo allora vale anche:

$$P0_e \geq P1_e$$

Dimostrazione:

Definiamo queste grandezze e relazioni per ipotesi

$$P(Y_i = 0) = \frac{1}{x_i} \; ; \; P_{aff} = \frac{1}{x_\alpha}$$

$$x_i \geq 1$$

$$0 \leq \frac{1}{x_i} \leq 1$$

$$0 < \prod_{i=0}^{N-1} \frac{1}{x_i} \leq 1$$

Ragioniamo per assurdo e consideriamo che valga

$$\prod_{i=0}^{N-1} \frac{1}{x_i} > \frac{1}{x_\alpha}$$

$$x_\alpha * \prod_{i=0}^{N-1} \frac{1}{x_i} > \frac{1}{1}$$

$$\prod_{i \neq \alpha}^{N-1} \frac{1}{x_i} > 1$$

Poiché $\prod_{i \neq \alpha}^{N-1} \frac{1}{x_i}$ è un prodotto tra coefficienti compresi tra 0 e 1 la relazione è falsa quindi è valido:

$$\prod_{i=0}^{N-1} \boldsymbol{P(Y_i = 0)} = \prod_{i=0}^{N-1} \frac{\mathbf{1}}{\boldsymbol{x_i}} \leq \frac{\mathbf{1}}{\boldsymbol{x_\alpha}} = \boldsymbol{P_{aff}}$$

Equazione 1

c.v.d.

Quindi possiamo affermare che nella configurazione in esame la soluzione proposta è più affidabile di quella attuale.

Caso KP con connessioni ridondanti. La SIB OSGi non definisce restrizioni su quante volte si possa sottoscrivere per una data query. Perciò un KP potrebbe sottoscriversi più volte creando connessioni multiple per la stessa notifica. Questo implica che esso dovrà essere in grado di gestire notifiche duplicate ma intuitivamente possiamo pensare che l'impiego di un numero maggiore di socket connesse tra cliente e servitore, aumenti l'affidabilità del "dialogo" tra le due parti. Infatti facendo uno studio simile a quello precedente possiamo dimostrare, per questo caso particolare, che la probabilità che il sistema corrente soddisfi la funzionalità in esame è maggiore di quella della soluzione proposta.

Il sistema a più collegamenti può essere visto come uno parallelo e quindi la probabilità che ci sia un fallimento è data da:

$$P0_e = P[(Y_0 = 1) \cap \dots \cap (Y_{N-1} = 1)]$$

Nell'ipotesi di indipendenza delle variabili possiamo scriverla come:

$$P0_e = \prod_{i=0}^{N-1} P(Y_i = 1)$$

Quindi definita $P1_e = P(Y_\alpha = 1)$ con $0 \leq \alpha < N$ casuale possiamo sfruttare la relazione (Equazione 1) dimostrata nella sezione precedente e perciò vale:

$$P0_e \leq P1_e$$

Cioè la probabilità che il sistema con l'implementazione o fallisca è minore di quella del nuovo sistema, ossia l'affidabilità di So è maggiore di S1.

Troughtput

Per garantire il controllo della congestione TCP, oltre ad altri meccanismi, applica il principio di *fairness* tra le connessioni. Cioè ad una connessione viene garantito un insieme di risorse proporzionale a quelle richieste e a quelle disponibili, evitando così che una singola comunicazione prenda tutte le risorse della rete facendo soffrire le altre di *starvation*. Ad esempio se due host comunicano attraverso un collo di bottiglia di banda limitata R ad ognuna delle K connessioni TCP tra i due nodi dovrebbe essere allocata una banda pari a $\frac{R}{K}$ [40].

Questo implica che nel sistema in esame l'implementazione attuale permette di riservare maggior banda per l'invio di notifiche da SIB a KP dipendente da R di $\frac{R}{N}$. Lo stesso meccanismo è sfruttato da http per aver un caricamento più veloce delle pagine web [24]. Il risultato netto è l'aumento delle notifiche al secondo inviabili e quindi un aumento del troughtput rispetto alla nuova soluzione. Tuttavia il principio di *fairness* non è rispettato in tutte le condizioni e su tutte le tipologie di network supportate di TCP come è stato dimostrato nello studio su Wireless 802.11 MAC, nel quale è emersa l'esistenza di un'asimmetria tra l'impiego di risorse nella *base station* [41]. Questo lascia aperta la questione sull'effettivo vantaggio della soluzione a connessioni multiple. Infatti considerando la natura del campo applicativo in cui stiamo operando andrebbe investigato maggiormente se l'utilizzo di un'unica connessione penalizzi in modo significativo il *troughtput* delle notifiche inviate dalla SIB.

Le due implementazioni (connessione multipla o singola) sono state valutate secondo questi aspetti: risorse utilizzate, costo computazionale, affidabilità e troughtput. Dallo studio eseguito possiamo affermare che l'implementazione a singola connessione riduce il consumo di risorse utilizzate lato cliente e servitore inoltre diminuisce anche il costo computazionale. Riguardo l'affidabilità è stato dimostrato che la soluzione proposta la aumenta nella maggior parte di casi applicativi, mentre perde nettamente nel caso studiato di "sottoscrizioni ridondanti". Inoltre cambiando l'approccio a multi-connessione si potrebbe perdere in troughtput cioè in numero di notifiche al secondo inviato da SIB a KP.

Riferimenti

[1] H. Kopetz, «Internet of Things,» in *Real-Time Systems*, Springer US, 2011, pp. 307-323.

[2] C. Gruppioni, «Comparing semantic based and non semantic based information flow processing in a case of drone driven agricoltural irrigation,» Bologna, 2016.

[3] J. Honkola, H. Laine, R. Brown e O. Tyrkkö, «Smart-M3 information sharing platform,» in *Computers and Communications (ISCC), 2010 IEEE Symposium on*, 2010.

[4] N. Bartlett, OSGi in practise, 2009.

[5] T. Berners-Lee, J. Hendler and O. Lassila, "The Semantic Web," *Scientific American Megazine,* 2001.

[6] World Wide Web Consortium (W3C), «W3C Semantic Web Activity,» 2011.

[7] I. V. Galov e D. G. Korzun, «Fault Tolerance Support for a Smart-M3 Application on the Software Infrastructure Level,» in *16th Conf. Open Innovations Association FRUCT*, Oulu, 2014.

[8] E. Prud'hommeaux e A. Seaborne, «SPARQL Query Language for RDF,» 2008.

[9] N. Bartlett, «Introduction,» in *OSGi In Practice*, 2009.

[10] A. R. Jack, «Jar hell,» in *Krysalis Community Project*, 2004.

[11] OpenJDK, «Project Jigsaw,» 2016. [Online]. Available: http://openjdk.java.net/projects/jigsaw/. [Consultato il giorno 1 03 2017].

[12] OpenJDK, «Penrose,» [Online]. Available: http://openjdk.java.net/projects/penrose/. [Consultato il giorno 11 02 2017].

[13] «Maven,» [Online]. Available: http://maven.apache.org/. [Consultato il giorno 15 1 2017].

[14] «Ivy,» [Online]. Available: http://ant.apache.org/ivy/. [Consultato il giorno 20 02 2017].

[15] «Gradle,» [Online]. Available: https://gradle.org/. [Consultato il giorno 2 03 2017].

[16] N. Bartlett, «Dynamic Services,» in *OSGi In Practice*, 2009, pp. 77-79.

[17] National Research Council, Software for dependable systems: Sufficient evidence?, National Academies Press, 2007.

[18] J.-C. Laprie, A. Avizienis, B. Randell e C. Landwehr, «Basic concepts and taxonomy of dependable and secure computing,» *IEEE Transactions on Dependable and Secure Computing,* vol. 1, n. 1, pp. 11-33, 2004.

[19] X. Liu e Y. Li, Advanced Design Approaches to Emerging Software Systems: Principles, Methodologies and Tools, 2011.

[20] HIVEMQ enterprise MQTT broker, "MQTT Essentials Part 4: MQTT Publish, Subscribe & Unsubscribe Subscribe & Unsubscribe," [Online]. Available: http://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe. [Accessed 20 12 2016].

[21] B. Meyer, A. Kogtenkov e E. Stapf, «Avoid a Void: The Eradication of Null Dereferencing,» in *Reflections on the Work of C.A.R. Hoare*, Springer London, 2010, pp. 189-211.

[22] F. Morandi, F. Vergari, A. D'Elia, L. Roffia e T. S. Cinotti, «Delayed SPARQL update,» in *SMART-M3 v.0.9: A semantic event processing engine supporting information level interoperability in ambient intelligence*, Bologna, 2013, pp. 10-12.

[23] S. Cleary, «Detection of Half-Open (Dropped) Connections,» 16 05 2009. [Online]. Available:

http://blog.stephencleary.com/2009/05/detection-of-half-open-dropped.html. [Consultato il giorno 01 03 2017].

[24] S. Groš, «Calculating TCP RTO,» 5 2 2012. [Online]. Available: http://sgros.blogspot.it/2012/02/calculating-tcp-rto.html. [Consultato il giorno 5 2 2017].

[25] Internet Engineering Task Force, "RFC 1122," R. Braden, Ottobre 1989. [Online]. Available: https://tools.ietf.org/html/rfc1122. [Accessed 20 01 2017].

[26] D. H. Crocker, «RFC 822,» 13 08 1982. [Online]. Available: https://tools.ietf.org/html/rfc822. [Consultato il giorno 13 02 2017].

[27] J. Ocampo, J. Meridth e e. al, Pablo's SOLID Software Development, Los Techies, 2008.

[28] Bndtools, «Bndtools Tutorial,» [Online]. Available: http://bndtools.org/tutorial.html. [Consultato il giorno 2 03 2017].

[29] C. Decker, «EventBusExplained,» 24 06 2015. [Online]. Available: https://github.com/google/guava/wiki/EventBusExplained.

[30] cziegeler, «Apache Felix Event Admin,» 4 9 2014. [Online]. Available: http://felix.apache.org/documentation/subprojects/apache-felix-event-admin.html.

[31] N. Bartlett, «The Whiteboard Pattern and,» in *OSGi In Practice*, 145-164, 2009.

[32] E. Gamma, R. Helm, R. Johnson e J. Vlissides, «STATE,» in *Design Patterns Elements of Resuable Object-Oriented Software*, 1995, pp. 305-313.

[33] B. Chin e T. Millstein, «An Extensible State Machine Pattern for Interactive Applications».

[34] «Design pattern,» [Online]. Available: https://it.wikipedia.org/wiki/Design_pattern.

[35] Information Sciences Institute University of Southern California, "tools.ietf.org," September 1981. [Online]. Available: https://tools.ietf.org/html/rfc793#section-2.4.

[36] Internet Engineering Task Force, «Computing TCP's Retransmission Timer,» June 2011. [Online]. Available: https://tools.ietf.org/html/rfc6298.

[37] B. DeLap, «Understanding RTT Impact on TCP Retransmissions,» 29 April 2014. [Online]. Available: http://blog.catchpoint.com/2014/04/29/understanding-rtt-impact-on-tcp-retransmissions/. [Consultato il giorno 5 02 2016].

[38] «Linux kernerl TCP RTO,» [Online]. Available: http://lxr.linux.no/linux+v3.2.4/include/net/tcp.h#L545.

[39] MitOpenCourseWare, «RELIABILITY OF SYSTEMS WITH VARIOUS ELEMENT CONFIGURATIONS,» [Online]. Available: https://ocw.mit.edu/courses/civil-and-environmental-engineering/1-151-probability-and-statistics-in-engineering-spring-2005/lecture-notes/app1_reli_final.pdf.

[40] C. Easwaran, «TcpFairness,» 28 3 2006. [Online]. Available: http://www.cs.newpaltz.edu/~easwaran/CCN/Week9/tcpFairness.pdf. [Consultato il giorno 5 2 2017].

[41] P. Saar, R. Ramachandran, R. Danny, S. Yuval e P. Sinha, «Understanding TCP fairness over Wireless LAN,» *Proceedings of IEEE Infocom*, 2003.