POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria Informatica

Tesi Magistrale

Service Discovery Avanzato per Servizi a Bordo Veicolo

Relatore
prof. Fulvio Risso

Candidato
Davide Saggese

Tutor aziendale
Italdesign
dott. ing. Massimo Reineri

Anno Accademico 2018-2019

Abstract

Il numero di dispositivi elettronici a bordo degli autoveicoli è in continua crescita e possiedono una complessità sempre maggiore. Questo implica la necessità di avere dispositivi con prestazioni elevate e una connessione con banda maggiore; per far fronte a queste esigenze è stato standardizzato Adaptive AUTOSAR. Questo standard utilizza un'architettura service-oriented, formata da server che forniscono dei servizi e client che li consumano attraverso un protocollo di comunicazione sulla rete. Uno di questi protocolli è SOME/IP: un middleware che permette le chiamate a procedure da remoto e dispone di un sistema di eventing. In ambito automotive il nuovo trend è di utilizzare sistemi Android in quanto offrono un'interfaccia grafica user-friendly, utile per i sistemi di infotainment, e un framework noto sul quale sviluppare le applicazioni.

La tesi si è svolta su un'implementazione open source di SOME/IP chiamata `vsomeip`, sviluppata per funzionare su sistemi embedded Linux. Il desiderio di rendere la libreria compatibile con i framework emergenti fa si che il primo obiettivo della tesi sia stato la sua cross-compilazione per il sistema Android. Successivamente si è proposta un'estensione del protocollo di Service Discovery di SOME/IP definendo un modello per la descrizione dei suoi servizi. Questa nuova feature permette una maggiore flessibilità, in quanto i client quando scoprono un servizio ne ricevono una descrizione contenente le sue caratteristiche; ed è proprio grazie a questa che il client è in grado di conoscerne metodi e variabili e di riorganizzare l'interfaccia grafica. Il modello proposto specifica la struttura che deve possedere il file di descrizione, precisando i nomi dei campi e quali devono essere obbligatoriamente presenti.

Infine, per poter mostrare concretamente le nuove funzionalità, è stato realizzato un dimostratore composto da una dashboard e da due versioni di uno stesso servizio, le quali vengono alternativamente connesse alla rete. Il client, che è in ascolto sul servizio, è in grado di adattare la propria interfaccia in base alla versione disponibile.

Indice

Capitolo 1

Introduzione

Se dal punto di vista meccanico l'evoluzione dell'autoveicolo è stata per lo più graduale e mai drastica, lo stesso discorso non si può applicare per la parte elettronica, dato che, negli ultimi anni ha subito una notevole evoluzione. Numerosi sono gli esempi di sensori inseriti nell'ultimo decennio all'interno delle vetture per agevolare la guida, renderla più sicura nonché efficiente: alcuni esempi sono i sensori di parcheggio per agevolare le manovre più difficili o i sensori degli angoli ciechi per coprire quell'area che gli specchietti non scorgono. Anche i sistemi di infotainment sono in continua evoluzione, con molteplici funzionalità, dalla connessione bluetooth con lo smartphone a un sistema di diagnostica dell'automobile.

Tutto ciò ha come conseguenza l'aumento del numero di centraline a bordo veicolo: al giorno d'oggi un auto di un segmento medio raggiunge il centinaio di dispositivi elettronici. Questi necessitano di comunicare tra loro, o con degli apparati centrali, creando così una rete di comunicazione, che a livello fisico significa cablare l'intera automobile.

L'architettura di rete usata in questo ambito è orientata ai servizi, nella quale troviamo dispositivi che offrono dei servizi mentre altri che ne usufruiscono. Data la quantità di quest'ultimi e della loro varietà è utile usufruire di un protocollo che sia in grado di scoprire tali servizi e sfruttarli, questo tipo di protocollo è chiamato `Service Discovery`.

In ambito automotive la nuova tendenza è di utilizzare sistemi Android ove possibile, ovvero per sistemi non safety-critical, in quanto offrono un'interfaccia grafica user-friendly che risulta utile soprattutto per la realizzazione di sistemi di infotainment. Inoltre, Android è un framework molto noto agli sviluppatori in ambito mobile, che possono quindi sfruttare le conoscenze già acquisite per realizzare applicazioni, senza la necessità di apprenderne di nuove.

Questa tesi è nata da una collaborazione tra il gruppo di Computer Networks del Politecnico di Torino e Italdesign. L'azienda in questione è tra le leader mondiali nella fornitura di servizi di sviluppo per l'industria automobilistica. Si occupano dello sviluppo di nuovi veicoli, dal design all'ingegneria e dalla produzione di prototipi al

collaudo e alla successiva validazione. Fondata nel 1968 da Giorgetto Giugiaro, ha sede a Moncalieri (TO) ed è stata acquisita dal gruppo Volkswagen.

1.1 Obiettivo della tesi

La tesi si è svolta lavorando all'estensione di un protocollo di rete usato per la comunicazione a bordo veicolo chiamato SOME/IP. Questo protocollo ha delle limitazioni soprattutto per quando riguarda la definizione dei servizi, i quali vengono definiti all'interno del codice delle applicazioni, ottenendo così un sistema poco flessibile. L'obiettivo principale della tesi è stati, quindi, quello di proporre delle modifiche al protocollo in modo da colmare queste limitazioni. Per raggiungerlo si è studiato lo stato dell'arte del Service Discovery, in modo da trarre vantaggio da ciò che già è stato definito in altri protocolli.

Il lavoro di tesi si è svolto su `vsomeip`, un'implementazione open-source di SOME/IP sviluppata per funzionare su sistemi Linux. Per andare in contro al nuovo trend di utilizzare sistemi Android, nasce il secondo obiettivo della tesi: il porting della libreria sopra citata su Android.

1.2 Struttura del lavoro

La parte restante della discussione è strutturato come segue:

- **Capitolo 2**: fornisce una panoramica delle tecnologie di rete più utilizzate in ambito automotive;
- **Capitolo 3**: offre una descrizione approfondita del protocollo SOME/IP e della sua implementazione `vsomeip`;
- **Capitolo 4**: motiva e descrive il lavoro eseguito per effettuare il porting della libreria `vsomeip`;
- **Capitolo 5**: illustra lo stato dell'arte del Service Discovery e mette a confronto i vari protocolli studiati tra essi e con SOME/IP–SD;
- **Capitolo 6**: presenta la nuova proposta per un protocollo SOME/IP–SD avanzato fonendo degli esempi esplicativi;
- **Capitolo 7**: fornisce una descrizione della struttura del dimostratore realizzato per mostrare l'impatto delle migliorie proposte;
- **Capitolo 8**: presenta le conclusioni di questa tesi.

Capitolo 2

Background

In questo capitolo vengono presentate le varie tecnologie di rete che vengono utilizzate in ambito automotive, in particolare per quanto riguarda la comunicazione a bordo veicolo (in-vehicle), nello specifico si parla degli standard bus CAN e Automotive Ethernet. Viene inoltre presentato il concetto di architetture orientate a servizi (Service-Oriented) e le motivazioni dietro la loro adozione a bordo veicolo.

2.1 Da CAN a Ethernet

Lo standard di comunicazione più comune a bordo veicolo è il Controller Area Network, noto anche come CAN-bus: è uno standard seriale di tipo broadcast e basato su messaggi, introdotto negli anni ottanta. Esso è stato progettato per riuscire a lavorare in ambienti con forti campi elettromagnetici. Il suo bit rate può raggiungere 1 Mbit/s in reti lunghe meno di 40 metri e va via diminuendo all'aumentare della lunghezza. La sua caratteristica principale è l'uso di un metodo di arbitraggio di risoluzione della contesa bit a bit senza perdita, che evita la necessità di ritrasmissioni quando più nodi tentano di inviare un messaggio contemporaneamente. Questo comportamento lo rende adatto a comunicazioni fortemente real-time.

I nuovi trend emergenti, come ad esempio la guida autonoma e i sistemi di infotainment avanzati, necessitano sempre di più velocità di connessione, oltre a prestazioni computazionali elevate. Per riuscire a soddisfare il bisogno di queste nuove applicazioni, diversi standard proprietari ad alta velocità esistenti, tra i quali MOST Protocol[1] e il FlexRay bus[2], possono essere utilizzati, comportando però

[1]Media Oriented System Transport Protocol: una tecnologia di rete multimediale ad alta velocità ottimizzata dall'industria automobilistica, che è utilizzata da quasi tutti i marchi automobilistici in tutto il mondo.

[2]Un bus ad alta velocità (10 Mbps) adatto alle comunicazioni time-sensitive, come drive-by-wire, steer-by-wire, brake-by-wire e così via.

costi elevati dati dalla loro complessità e dalle licenze associate.

Uno degli standard più diffusi oggigiorno nelle reti ICT è Ethernet, il quale però non è mai stato utilizzato in ambito automotive: il motivo principale risiede nel fatto che lo standard non è conforme ai requisiti di tolleranza delle interferenze elettromagnetiche. Recentemente però, Open Alliance, un gruppo di interesse senza fini di lucro composto principalmente da industrie automobilistiche e fornitori di tecnologia, ha iniziato a sponsorizzare la 100 Mbps BroadReach, che permette una trasmissione full-duplex e lavora a una frequenza ridotta per rispettare i requisiti di tolleranza sulle interferenze elettromagnetiche. Successivamente questa tecnologia è stata standardizzata dal gruppo IEEE 802.3 come 100BASE-T1 [1] (100 Mbps) e 1000BASE-T1 [2] (1000 Mbps) ed è comunemente indicato con il nome di Automotive Ethernet.

Apportando Automotive Ethernet una modifica soltanto al livello fisico, esso è completamente compatibile con i protocolli usati ogni giorno su vanilla Ethernet, potendo così usufruire di tutti vantaggi di anni di esperienza. Nonostante ciò, dei nuovi protocolli sono in fase di sviluppo per andare a soddisfare le richieste del mondo automotive; tra questi c'è SOME/IP, un middleware che sarà oggetto di studio nel capitolo 3.

AUTomotive Open System ARchitecture [3] è una partnership di sviluppo mondiale di parti interessate nel settore automobilistico fondata nel 2003 che persegue l'obiettivo di creare un'architettura software aperta e standardizzata per le centraline elettroniche di controllo automobilistiche (ECU) [4]. Per far fronte alle nuove esigenze nel mondo dell'automotive, AUTOSAR ha definito un nuovo standard suddiviso in:

- `Classic AUTOSAR`, è dedicato a tutte quelle applicazioni più tradizionali che necessitano di comunicazioni real-time, alta sicurezza e vengono eseguite su ECU con risorse limitate. In particolare, viene utilizzato per le centraline a diretto contatto con sensori e attuatori. La connessione è principalmente implementata mediante l'utilizzo di CAN bus;

- `Adaptive AUTOSAR`, invece è stato definito per supportare le nuovi tecnologie che hanno bisogno di computer con alte prestazioni multi-core, una rete switched ad alte prestazioni e architettura eterogenea ma con requisiti real-time meno stringenti. La comunicazione è orientata ai servizi ed è basata principalmente su SOME/IP.

2.2 Architetture Service-Oriented

L'architettura orientate a servizi, in inglese Service-Oriented Architecture (SOA), è uno stile di progettazione del software basato sui *servizi*, i quali vengono offerti dalle varie applicazioni, attraverso un protocollo di comunicazione su una rete. Un

servizio è un'unità di funzionamento a cui è possibile accedere da remoto, che può essere utilizzata e aggiornata in modo indipendente. I principi base delle SOA sono indipendenti dai fornitori, dai prodotti e dalle tecnologie.

Un servizio ho quattro proprietà fondamentali:

- rappresenta logicamente un'attività con uno specifico compito;
- è autonomo;
- è una black-box per i suoi consumatori, ovvero, non sanno come sia realizzato il servizio ma solamente cosa offre;
- può contenere dei servizi sottostanti.

La chiave di funzionamento sta nella totale assenza di business logic sul client SOA, il quale è totalmente agnostico rispetto alla piattaforma di implementazione, riguardo ai protocolli, al tipo di dati, alle policy con cui il servizio produrrà l'informazione richiesta. Tutto a beneficio dell'indipendenza dei servizi, che possono essere chiamati per eseguire i propri compiti in un modo standard, senza che il servizio abbia conoscenza dell'applicazione chiamante e senza che l'applicazione abbia conoscenza del servizio che effettivamente eseguirà l'operazione.

Diversi servizi possono essere utilizzati insieme per fornire la funzionalità di una grande applicazione software, esattamente come nella programmazione modulare. In questo modo è possibile sviluppare separatamente le sottofunzioni in compiti più piccoli e semplici che poi collaboreranno tra loro per realizzare la funzionalità finale.

Ogni blocco di funzionamento delle service oriented architecture può avere uno dei seguenti ruoli:

- `Service provider`, colui che fornisce il servizio;
- `Service broker, service registry o service repository`, la sua funzionalità principale è quella di diffondere le informazioni riguardo ai service provider disponibili nella rete;
- `Service requester/consumer`, colui che consuma il servizio.

Questa architettura permette uno sviluppo più rapido e costi contenuti, in quanto permette il riutilizzo dei servizi in diversi contesti.

Capitolo 3

SOME/IP

Scalable service-Oriented MiddlewarE over IP (SOME/IP) è un middleware di comunicazione progettato per l'utilizzo nel mondo Automotive Ethernet, ed è stato standardizzato da AUTOSAR per far fronte alle specifiche richieste dai nuovi scenari emergenti, come la guida autonoma degli autoveicoli, che necessitano di molta banda per la comunicazione e computer ad alte prestazioni.

SOME/IP realizza un'architettura di tipo service-oriented, sfruttando i protocolli di trasporto UDP e TCP.

3.1 Paradigmi di funzionamento

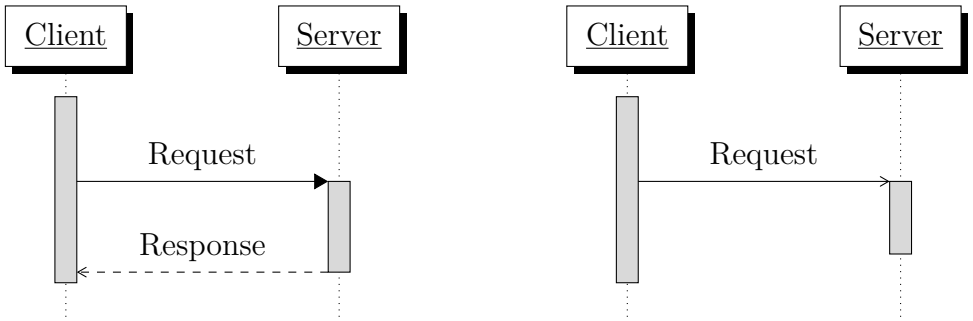
Questo middleware è progettato per fornire un'architettura service-oriented (SOA) e definisce un *servizio* come un'entità atomica che raggruppa `metodi` e `eventi`; istanze dello stesso servizio possono coesistere in device differenti ma anche nello stesso.

Due diversi pattern di comunicazione, request/response e publish/subscribe, sono offerti da SOME/IP, che saranno descritti nei due sottoparagrafi successivi.

3.1.1 Request/Response

Il paradigma di funzionamento request/response equivale a una Remote Procedure Call (RPC), ovvero alla chiamata di funzioni remote. Permette quindi di invocare funzioni che non si trovano sull'applicazione locale ma metodi su un service remoto.

Il funzionamento standard consiste, come illustrato nella figura 3.1a, in una prima fase in cui il client effettua la richiesta di invocare un metodo, quindi viene costruito un messaggio SOME/IP e inoltrato al server di destinazione contente il servizio. Una volta ricevuto il messaggio di richiesta, il server esegue il metodo specificato dal client e risponde con il risultato dell'operazione.



(a) request/response classico. (b) Fire & forget.

Figura 3.1: Diagramma di comunicazione request/response.

Una modalità di funzionamento differente è il fire & forget, rappresentata in figura 3.1b, che consiste in una comunicazione in una sola direzione, ovvero viene inviata la richiesta dal client al server ma non è presente alcuna risposta.

3.1.2 Publish/Subscribe

In alternativa al pattern request/response, è possibile sfruttare quello publish/subscribe, il quale permette di realizzare un sistema di notificazione. Come illustrato in figura 3.2, il client si iscrive a un `evento` una sola volta con un messaggio di `subscribe`, dopo di ché ogni volta che il valore della variabile di cui si è fatta l'iscrizione varia, viene inviato un messaggio di notifica a tutti gli iscritti con un nuovo valore. In questo modo il programma non deve specificare a chi inviare il messaggio di notifica, ma il middleware gestisce tutto in trasparenza.

In base al caso d'uso, esistono tre strategie per l'invio delle notifiche:

- `cyclic updates`, il messaggio di notifica viene inviato a un intervallo di tempo costante;
- `updates on change`, la notifica è inviata non appena viene apportata una modifica alla variabile;
- `epsilon change`, la notifica viene trasmessa solamente quando il nuovo valore si discosta, rispetto al vecchio, più di una certa soglia.

Uno dei vantaggi di offerto dall'utilizzo del publish/subscribe è l'ottimizzazione dell'invio dei messaggi e quindi della banda utilizzata: in base a quanti client hanno fatto l'iscrizione all'evento, il middleware si occuperà di inviare i messaggi ai singoli destinatari, o di utilizzare il multicast; inoltre, nel caso in cui nessuno è iscritto all'evento, non verrà inviato nessun messaggio.

Il protocollo SOME/IP si occupa della fase di notificazione, mentre la fase di iscrizione è gestita da SOME/IP–SD, un suo modulo che offre funzionalità di Service Discovery, il cui funzionamento verrà approfondito del paragrafo §3.4.

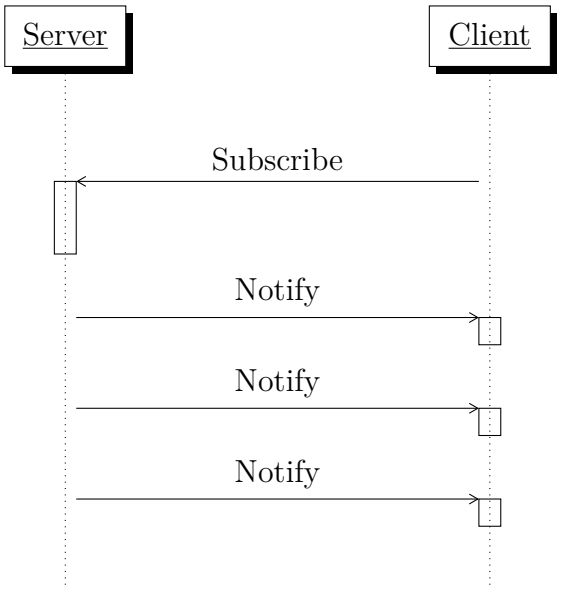


Figura 3.2: Diagramma di comunicazione publish/subscribe.

3.2 Protocollo di trasporto

SOME/IP opera al di sopra del protocollo di trasporto, il quale fornisce le funzionalità necessarie per la consegna dei messaggi dal mittente al destinatario. I due protocolli al momento supportati per la comunicazione sono: UDP e TCP. Nonostante ciò, possono essere usati anche altri protocolli di trasporto come Network File System (NFS) nel caso siamo più adatti.

3.2.1 UDP

UDP è un protocollo di tipo connectionless, ovvero senza connessione. Per inviare un messaggio SOME/IP, il middleware lo ingloba semplicemente all'interno di un pacchetto UDP e lo inoltra in rete. È inoltre possibile inviare più messaggi in un solo pacchetto per ottimizzare le trasmissioni; quando il destinatario riceve il pacchetto è in grado di separare i messaggi grazie al campo `length` all'interno dell'header SOME/IP.

La dimensione massima du un messaggio SOME/IP su UDP è fissata a 1400 byte, in modo da non avere messaggi IP frammentati visto che la Maximum Transmission Unit (MTU) di default su Ethernet è pari a 1500 byte. Per poter trasmettere messaggi più grandi, è stata standardizzata un'estensione del protocollo chiamata SOME/IP–TP. Quest'ultima si occupa di frammentare il messaggio direttamente nel framework, di associare ad ogni frammento un header SOME/IP con i campi necessari per la ricostruzione del messaggio originale e, infine, di inviare ognuno di essi in un pacchetto UDP distinto.

Questo protocollo di trasporto è da preferire, come suggerito nella documentazione, sia nel caso di messaggi piccoli che nel caso la frammentazione debba essere utilizzata, in quanto è un protocollo leggero che aggiunge un piccolo header e permette delle latenze molto basse. Molti messaggi nelle applicazioni automotive sono ciclici: di conseguenza, il miglior approccio a un errore di trasmissione è attendere il messaggio successivo, invece di rilevarlo ed effettuare la ritrasmissione. Infine UDP permette il multicast, che a sua volta permette di ottimizzare l'utilizzo della rete.

3.2.2 TCP

L'alternativa all'utilizzo di UDP è il protocollo TCP, molto più complesso, che offre delle funzionalità come la gestione della perdita di pacchetti e della presenza di pacchetti duplicati, il riordinamento e la ritrasmissione nel caso sia richiesta, tutto in maniera trasparente.

A differenza di UDP, TCP è un protocollo connection-oriented, quindi viene stabilita una connessione la prima volta che il client contatta il server, che viene usata per la comunicazione di diversi pacchetti e poi chiusa dal client quando non è più utile.

Offrendo funzionalità sul controllo del flusso e della congestione della rete, TCP è un protocollo pesante, ovvero, che introduce parecchio overhead. Il suo utilizzo è suggerito quando non è richiesto un basso tempo di latenza o quando è necessario inviare una grande quantità di dati.

3.3 Formato On-Wire

Il messaggio SOME/IP è composto da un `header`, che specifica i parametri necessari per la comunicazione, e il `payload` che contiene i dati da trasferire.

L'header è lungo 16 byte, come possiamo vedere in figura 3.3, e nello specifico è composto dai seguenti campi:

- `Service ID`, che contiene l'identificativo univoco del servizio, insieme al Method ID forma il Message ID;
- `Method ID`, specifica il metodo RPC del servizio che deve essere eseguito o l'evento a cui appartiene il messaggio di notifica;
- `Length`, contiene la lunghezza in byte a partire dal campo successivo (Client ID) fino alla fine del messaggio SOME/IP;
- `Client ID`, identifica il client che ha effettuato la richiesta; combinato con il Session ID forma il Request ID;
- `Session ID`, un identificativo che viene incrementato a ogni nuovo messaggio;

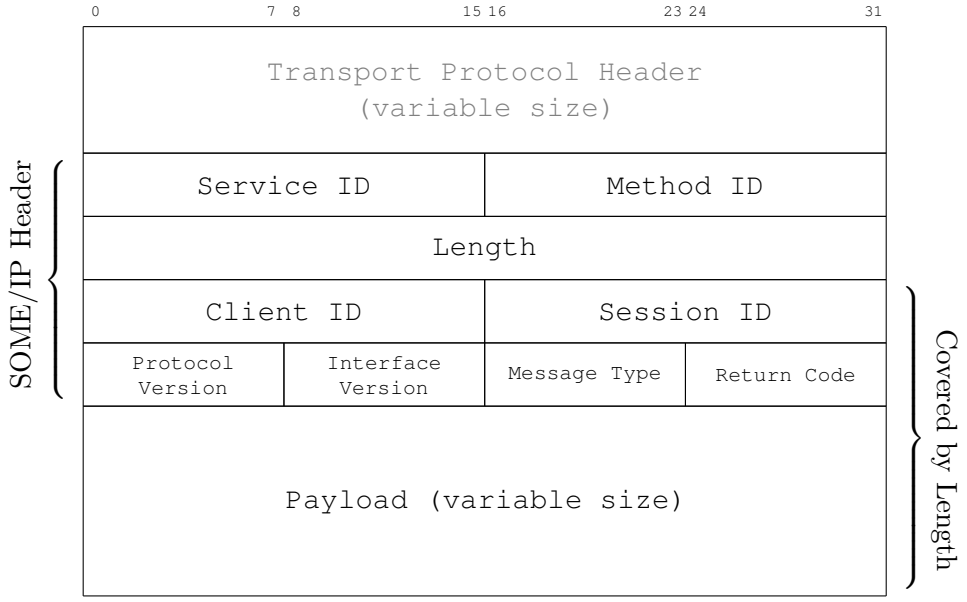


Figura 3.3: Messaggio SOME/IP.

- `Protocol Version`, specifica la versione di SOME/IP;
- `Interface Version`, indica la versione dell'interfaccia del servizio;
- `Message Type`, viene usato per identificare il tipo di messaggio: come ad esempio `REQUEST`, `RESPONSE` e `NOTIFICATION`; include inoltre l'informazione che il pacchetto è frammentato per permettere a SOME/IP–TP di gestire la frammentazione;
- `Return Code`, specifica, nella risposta, se l'operazione richiesta è andata a buon fine o se si è verificato un errore.

Le specifiche di SOME/IP forniscono anche delle regole per la serializzazione dei dati del payload in base al tipo (numeri, stringe, array, ...).

3.4 SOME/IP Service Discovery

Una delle principali funzionalità offerte dal protocollo SOME/IP è il Service Discovery, la quale è fornita dal modulo chiamato SOME/IP–SD. Le funzioni principali di questo modulo sono la gestione delle disponibilità dei servizi nell'automobile e il controllo dell'invio di messaggi di evento [5]. Esso permette all'applicazione di fornire le funzionalità dei servizi, mentre la parte di scoperta e iscrizione è gestita in maniera trasparente dal framework.

Grazie al Service Discovery, un'ECU è in grado di offrire un'istanza di un servizio e di scoprire le istanze disponibili nella rete; una `Service Instance` è un'implementazione di un determinato servizio. Inoltre, SOME/IP–SD implementa la gestione del Publish/Subscibe.

Le principali caratteristiche vengono fornite attraverso gli `offer message`, ovvero dei messaggi che ogni device invia in multicast contenente l'elenco di tutti i servizi che esso offre. Nel caso in cui un client voglia richiedere un determinato servizio, senza attendere un `offer message`, può inviare un `find message`, il quale viene inviato in multicast e se ricevuto dal servizio di interesse risponderà al client. Inoltre, SOME/IP–SD gestisce la pubblicazione e l'iscrizione agli *eventgroup*. Un eventgroup è un insieme di eventi, i quali condividono l'iscrizione. In altre parole, non ci si può iscrivere a un evento singolo ma a un eventgroup.

3.4.1 Formato dei pacchetti

SOME/IP–SD utilizza UDP come protocollo di trasporto e possono essere inviati sia con messaggi unicast ai singoli client sia multicast raggiungendo tutti i client in ascolto.

Ogni messaggio SOME/IP–SD è composto da un suo header, da una lista di `Entry` e da un lista di `Option`. Le entry servono a contenere le offerte e le richieste riguardanti i servizi e le offerte e le iscrizione agli eventgroup. Le option invece contengono delle descrizioni aggiuntive e sono associate a un'entry, la quale può possedere diverse option o nessuna. I messaggi di Service Discovery iniziano con un header SOME/IP con dei valori particolari, che servono ad identificare che la restante parte del messaggio contiene un pacchetto SOME/IP–SD.

Analizziamo ora i campi di SOME/IP–SD:

- `Flags`, è composto da 8 flag utili per il protocollo, come ad esempio il flag per indicare se il messaggio è unicast o meno;
- `Reserved`, sono 24 bit riservati al protocollo;
- `Length of Entries Array`, specifica la lunghezza in byte dell'array contenente le Entry;
- `Entries Array`, è il vettore di `Entry`, utili per sincronizzare lo stato delle istanze dei servizi;
- `Length of Options Array`, indica la lunghezza in byte del vettore delle `Options`;
- `Options Array`, è il vettore di Option.

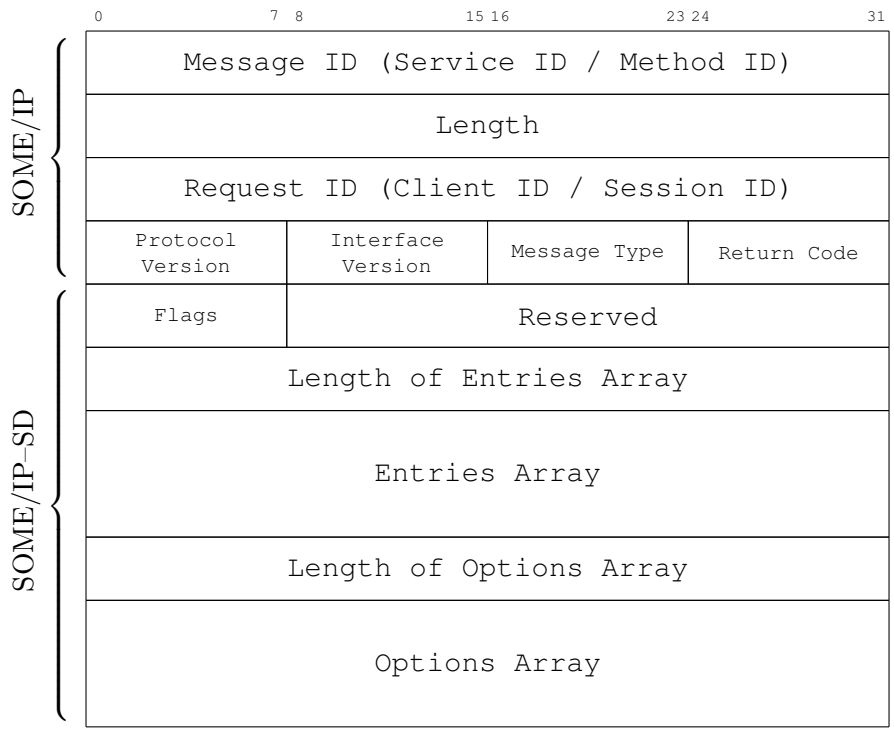


Figura 3.4: Formato dell'header SOME/IP–SD.

Ogni messaggio SOME/IP–SD contiene quindi diverse `Entry` e diverse `Option`. Le entry si suddividono in `Service Entry Type` e `Eventgroup Entry Type`; come suggeriscono i nomi, il primo serve per fare l'advertisement dei servizi disponibili, mentre il secondo, degli eventgroup.

Le Entry sono lunghe 16 byte e sono formate dai seguenti campi:

- `Type`, identifica il tipo di entry, i quatro tipi principali sono:
  - `OfferService (0x01)`, serve per fare l'advertisement di una data istanza di servizio, è periodicamente inviato dal protocollo e può essere utilizzato anche per comunicare che un dato servizio non è più disponibile;
  - `FindService (0x00)`, usato per velocizzare il processo di scoperta, richiedendo uno specifico servizio;
  - `Subscribe (0x06)`, viene usato dai client per iscriversi a un eventgroup;
  - `SubscribeAck (0x04)`, comunica se l'iscrizione all'eventgroup è andata a buon fine o meno.
- `Index first options`, è l'indice della prima `option` primaria associata;
- `Index second options`, è l'indice della prima `option` secondaria associata;
- `Number of opt 1`, specifica quante sono le opzioni primarie;
- `Number of opt 2`, specifica quante sono le opzioni secondarie;
- `Service ID`, identifica il servizio a cui è associata questa entry;
- `Instance ID`, identifica l'istanza a cui si riferisce la entry, nel caso in cui sia per qualsiasi istanza il suo valore è uguale a 0xFFFF;
- `Major Version`, è la versione major del servizio;
- `TTL`, è il tempo di vita della entry espresso in secondi;
- I restanti 32 bit sono differenti nel caso si tratti di un Service o Eventgroup Entry:
  - nel caso di Service Entry specificano la `Minor Version`;
  - negli Eventgroup Entry, invece, in parte sono riservati mentre 16 bit vengono usati per specificare l'`Eventgroup ID`.

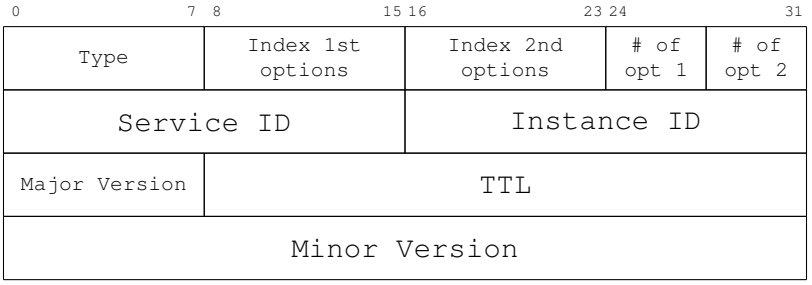


Figura 3.5: SOME/IP–SD Service Entry Type.

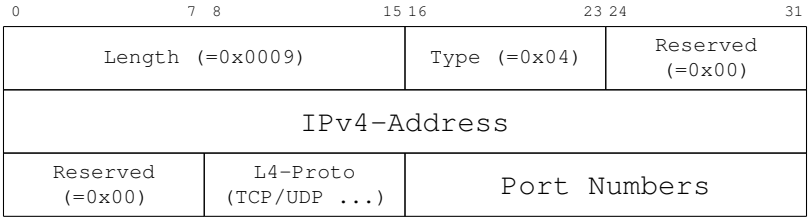


Figura 3.6: SOME/IP–SD IPv4 Endpoint Option.

Le `Options` sono usate per trasportare informazioni aggiuntive sulle entry, come ad esempio i parametri di rete tra cui l'indirizzo IP in cui si trova un determinato servizio.

Ogni `option` è espressa in formato Type-Length-Value, ovvero i primi due campi sono fissi:

- `Length`, che specifica la lunghezza in byte dell'option esclusi i campi length e type;
- `Type`, indica il tipo dell'option.

La restante parte del pacchetto è specifica per il tipo di option; vengono definiti dei tipi standard con un loro formato, in questo modo i campi saranno specifici alle esigenze del tipo, senza avere campi inutilizzati o mancanti. Ad esempio l'option per specificare i parametri di rete con IPv4, come possiamo vedere in figura 3.6, contiene un campo per l'indirizzo IP, uno per il protocollo di trasporto e uno per il numero di porta.

I principali tipi di option definiti nelle specifiche sono:

- `Configuration Option`, usato per trasportare delle stringhe di configurazione arbitrarie, sotto forma di `[lunghezza]id=valore`;
- `Load Balancing Option`, è formato da solo due campi oltre i primi 32 bit: `Priority` e `Weight`. La sua utilità è quella di assegnare diverse priorità a diverse istanze di un servizio;

- `IPv4 Endpoint Option`, è usata da SOME/IP–SD per segnalare gli endpoint;
- `IPv6 Endpoint Option`, uguale al precendente ma con indirizzamento IP versione 6;
- `IPv4 Multicast Option`, è usato dai server per specificare gli indirizzi multicast, il livello di trasporto e la porta associati ad eventi; esiste `IPv6 Multicast Option` per il medesimo scopo ma con IPv6.

Un messaggio SOME/IP–SD può contenere diverse entries e quindi al suo interno possiamo trovare diverse istanze di servizi che si annunciano, molteplici advertisement di eventgroup, oppure richieste di servizi o eventgroup da parte dei client. Inoltre, ogni entry può avere associata una o più option per la descrizione del servizio o dell'eventgroup contenuto in essa.

3.5 vsomeip

Lo stack `vsomeip` è un'implementazione open-source del protocollo SOME/IP, definito come parte del progetto GENIVI[1]. È scritto in linguaggio `C++` ed è diviso in tre moduli principali:

- `libvsomeip.so`, è la libreria principale, svolge le funzionalità base del protocollo SOME/IP. Essa si occupa di stabilire le connessioni tra i server e i client e della trasmissione dei pacchetti, inoltre, gestisce l'eventing. Un altro lavoro svolto da questa libreria è la creazione e serializzazione degli header dei messaggi, mentre la serializzazione dei payload è lasciata allo sviluppatore dei programmi `vsomeip`;
- `libvsomeipi-sd.so`, questa libreria di supporto implementa quanto descritto nelle specifiche di SOME/IP–SD. Nel caso in cui il programma necessiti di funzionalità come la scoperta dei servizi disponibili e l'iscrizione a eventi, la libreria principale si occupa del caricamento di questa libreria; essa è necessaria per il paradigma di funzionamento publish/subscribe;
- `libvsomeip-cfg.so`, la sua funzione principale consiste nella lettura e interpretazione dei file di configurazione JSON, che verranno descritti in seguito (§3.5.3), e rendere disponibili le informazioni ottenute a tutta la libreria; anche questo modulo viene caricato a runtime dalla libreria `libvsomeip.so`.

[1]GENIVI è un'alleanza del settore automobilistico senza fini di lucro impegnata a guidare l'ampia adozione del software open source, In-Vehicle Infotainment (IVI) e fornire tecnologia aperta per l'auto connessa. [6]

Sebbene l'implementazione di vsomeip abbia seguito le specifiche di SOME/IP, alcune funzionalità non sono ancora state implementate. Ad esempio la gestione delle priorità definita in SOME/IP–SD e la gestione della frammentazione definita in SOME/IP–TP.

3.5.1 Architettura

Una rappresentazione dell'architettura di vsomeip la possiamo trovare in figura 3.7 [7], nella quale due ECU sono connesse allo stesso link Ethernet. La prima ECU ospita due applicazioni vsomeip, mentre la seconda solamente una.

Partendo dall'alto, subito sotto le applicazioni troviamo il modulo che espone le API dalla libreria, che permettono l'utilizzo delle sue funzionalità di vsomeip da parte delle applicazioni. Tale modulo funge come interfaccia tra le applicazioni e il livello sottostante della libreria. Inoltre comprende le classi che permettono la creazione dei messaggi SOME/IP e quindi la serializzazione e deserializzazione.

La parte centrale della libreria, è composta dalla classe routing_manager, che si occupa dell'inoltro effettivo dei messaggi, sia per quando riguarda la comunicazione tra applicazioni che risiedono nella stessa ECU, detta InterProcess Comunication (IPC), sia per la comunicazione tra applicazioni remote. Per questa ragione esistono due differenti tipologie di Routing Manager:

- routing manager, è la versione completa del routing manager, esso viene istanziato quando l'applicazione viene eseguita e non esistono altre applicazioni vsomeip già avviate sullo stesso device. Si occupa della comunicazione sulla rete e quindi di sfruttare un protocollo di trasporto per l'invio dei pacchetti e della comunicazione con le altre applicazioni locali. Inoltre, fa il caricamento del modulo per il Service Discovery se necessario e, essendo l'unico del device a possedere questo modulo, a propagare le informazioni ottenute con il Service Discovery alle altre applicazioni locali;

- routing manager proxy, è una versione semplificata del routing manager ed è usato quando esiste già un routing manager attivo in locale. Si occupa delle comunicazioni IPC; per poter inviare i messaggi sulla rete e per sfruttare le funzionalità di SOME/IP–SD comunica con il manager principale.

3.5.2 Esempi di applicazioni

In questo sottocapitolo vengono presentati due esempi di applicazioni vsomeip, un client e un server che comunicano sfruttando il paradigma request/response.

Il listato 3.1 illustra un'applicazione che realizza un server, molto semplice e senza controlli di errore, sfruttando vsomeip. Come prima cosa nel main viene chiamato il costruttore di una variabile di tipo vsomeip::application, essa racchiude la

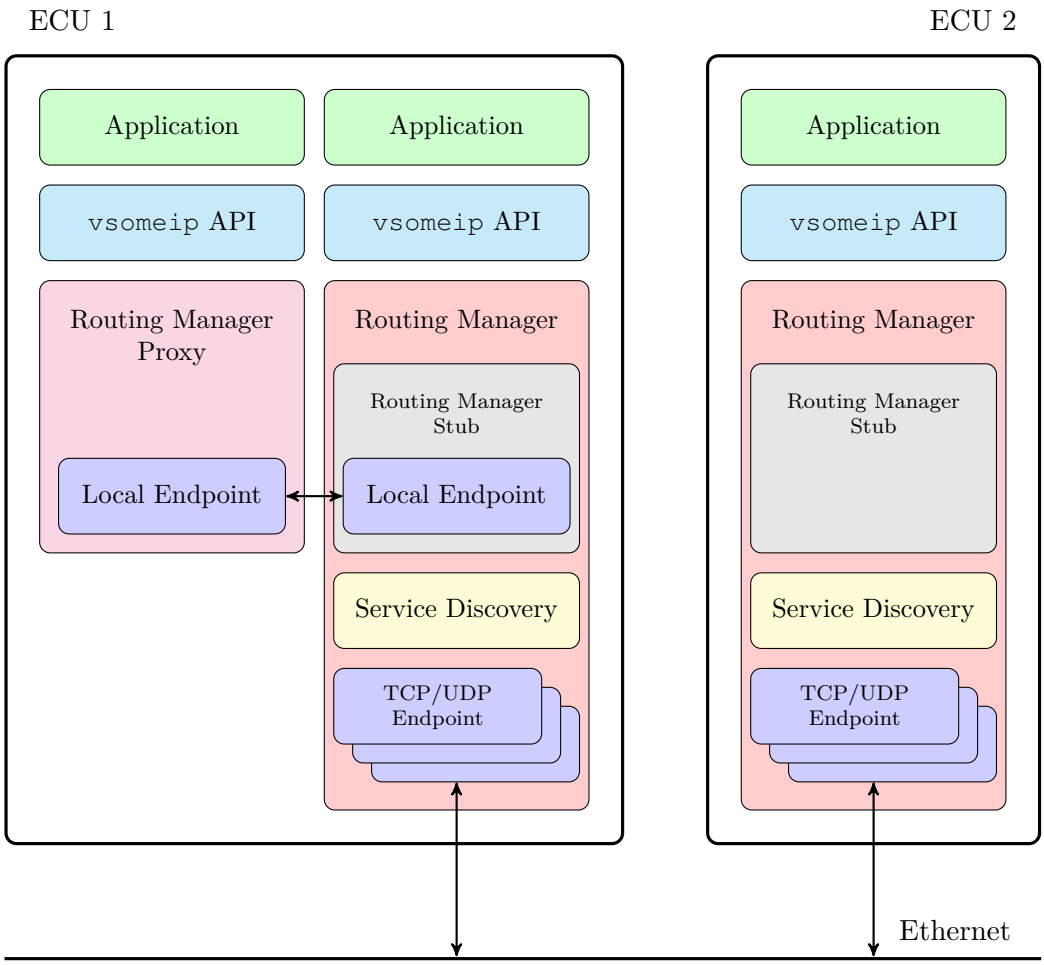


Figura 3.7: Architettura `vsomeip`.

Listato 3.1: Esempio di Server.

```
1  #include <...>
2  #include <vsomeip/vsomeip.hpp>
3
4  std::shared_ptr<vsomeip::application> app;
5
6  int main() {
7      app = vsomeip::runtime::get()->create_application("Server");
8      app->init();
9      app->register_message_handler(SERVICE_ID, INSTANCE_ID, METHOD_ID, on_message);
10     app->offer_service(SERVICE_ID, INSTANCE_ID);
11     app->start();
12 }
13
14 void on_message(const std::shared_ptr<vsomeip::message> &request) {
15     auto its_response = process_request(request);
16     app->send(its_response);
17 }
```

maggior parte delle funzionalità; subito dopo, a riga 8, viene chiamato un suo metodo per l'inizializzazione. Nella riga successiva viene registrato un `message_handler` ovvero, viene associata la chiamata del client di un determinato metodo (in questo caso `METHOD_ID`) a una funzione di callback. In questo programma la funzione si chiama `on_message`, la quale processa la richiesta e risponde al client. Penultimo passaggio, dice al framework che questa applicazione offre un servizio, identificato dal `SERVICE_ID` e dal `INSTANCE_ID`. L'ultima riga del main chiama il metodo `start` che fa partire l'applicazione `vsomeip`.

L'esempio di applicazione client è raccolto nel listato 3.2; come nell'esempio del server, il client crea l'`application` come prima cosa, passando però come nome "Client", e chiama la funzione di inizializzazione. Successivamente, attraverso la funzione `register_availability_handler`, viene registrata una funzione di callback che viene chiamata quando il servizio specificato diventa disponibile. Questa funzione di callback, chiamata `on_availability`, crea un messaggio attraverso la funzione `prepare_request` e lo invia al server. Poi, come per il server, viene registrata una funzione di callback anche per quando viene ricevuto un messaggio e, prima di eseguire lo start dell'applicazione, viene detto al framework che l'applicazione è interessata al servizio specificato con `SERVICE_ID` e `INSTANCE_ID`.

3.5.3 File di configurazione

La libreria `vsomeip` utilizza dei file di configurazione in formato JSON per gestire l'inizializzazione di alcuni parametri di rete e di funzionamento in generale. Due esempi di file di configurazione sono i listati 3.3 e 3.4.

Le principali proprietà dei file di configurazione sono:

Listato 3.2: Esemio di Client.

```
#include <...>
#include <vsomeip/vsomeip.hpp>

std::shared_ptr<vsomeip::application> app;

int main() {
    app = vsomeip::create_application("Client");
    app->init();
    app->register_availability_handler(SERVICE_ID, INSTANCE_ID, on_availability);
    app->register_message_handler(SERVICE_ID, INSTANCE_ID, METHOD_ID, on_message);
    app->request_service(SERVICE_ID, INSTANCE_ID);
    app->start();
}

void on_availability(...) {
    if (/* the requested service is available */) {
        vsomeip::message request = prepare_request();
        app->send(request);
    }
}

void on_message(vsomeip::message response) {
    process_response(response);
}
```

Listato 3.3: Configurazione del Server.

```
{
    "unicast" : "192.168.12.1",
    "applications" : [{
        "name" : "Server",
        "id" : "0x1343"
    }],
    "services" : [{
        "service" : "0x1234",
        "instance" : "0x5678",
        "reliable" : { "port" : "31000"},
        "unreliable" : "31000"
    }],
    "service-discovery" : {
        "enable" : "true",
        "multicast" : "224.244.224.245",
        "port" : "30490",
        "protocol" : "udp"
    }
}
```

Listato 3.4: Configurazione del Client.

```
{
    "unicast" : "192.168.12.2",
    "applications" : [{
        "name" : "Client",
        "id" : "0x1344"
    }],
    "service-discovery" : {
        "enable" : "true",
        "multicast" : "224.244.224.245",
        "port" : "30490",
        "protocol" : "udp"
    }
}
```

- `unicast`, indica l'indirizzo IP dell'interfaccia locale usato per la trasmissione dei messaggi;
- `application`, è un array di oggetti `application` i quali specificano i nomi e gli identificativi delle applicazioni, questi ID corrispondono ai Client ID all'interno dell'header SOME/IP;
- `services`, indicano quali sono le istanze di servizio e gli eventgroup offerti dalle applicazioni;
- `routing`, esiste un solo routing manager per device, il quale viene assegnato alla prima applicazione `vsomeip` eseguita oppure all'applicazione che viene specificata in questo campo;
- `service-discovery`, utile per la configurazione del Service Discovery nel caso sia abilitato. In questo caso i parametri obbligatori sono l'indirizzo multicast, usato per l'invio dei messaggi di service discovery, e la porta e il protocollo. Gli altri parametri determinano quanto spesso gli `offer messages` sono inviati.

Capitolo 4

Porting su Android

Oggigiorno i sistemi di infotainment sono sempre più presenti all'interno degli autoveicoli. Questi vengono chiamati in-car entertainment (ICE) o in-vehicle infotainment (IVI) e sono una raccolta di hardware e software che forniscono principalmente un intrattenimento audio o video ma includono anche sistemi di navigazione e connettività di vario genere (USB, Bluetooth, ecc.). I sistemi ICE sono sempre più comuni nei nuovi modelli di veicoli e sono sempre più complessi.

Le centraline che si occupano di applicazioni hard real-time e con un alto impatto sulla sicurezza dei veicoli, come ad esempio la frenata automatica di emergenza, possiedono CPU molto veloci e le loro applicazioni sono sviluppate su sistemi Linux per ragione di performance. I sistemi di infotainment, invece, possiedono requisiti differenti: non necessitano di velocità così restrittive, ma pongono più attenzione all'usabilità, all'interfaccia grafica e all'estetica. Per questo motivo, ultimamente, la tendenza è quella di utilizzare al posto di Linux il sistema Android, il quale offre già queste peculiarità. Inoltre è un sistema ben noto agli sviluppatori mobile, fattore che permette di realizzare applicazioni velocemente.

Recentemente si è pensato di sfruttare i dispositivi personali come smartphone e tablet al fine di controllare i sistemi di infotainment, aumentando l'accessibilità del sistema e permettendo, ad ogni passeggero munito di tali dispositivi, di interagire con l'IVI.

Per realizzare la comunicazione tra infotainment è fondamentale avere una connessione con una banda elevata, in quanto spesso è necessario inoltrare contenuti multimediali come streaming di video. Per questo le connessioni vengono realizzate con standard come Automotive Ethernet. Inoltre l'architettura di rete è basata sui servizi per questo viene utilizzato il protocollo SOME/IP.

Per i motivi precedentemente affrontati il primo obiettivo della tesi è stato il porting su Android della libreria `vsomeip`, che come visto nel capitolo 3 è un'implementazione open source di SOME/IP, sviluppata per Linux.

Figura 4.1: Esempio di in-car infotainment. [8]

4.1 Problemi affrontati

La libreria `vsomeip` è scritta in C++ e si appoggia a Boost, una collezione di librerie open source che estendono le funzionalità del medesimo linguaggio. Essa fornisce un'astrazione di alto livello per semplificare l'uso dei socket C, per la gestione del multithreading, dell'algebra lineare e così via.

Boost è composto per la gran parte da header, che non hanno bisogno di essere compilati; `vsomeip` però si appoggia anche su tre librerie che necessitano la compilazione: system, thread e log. Il primo problema affrontato quindi è stato la cross-compilazione delle librerie Boost.

Come specificato nelle dipendenze di `vsomeip`, la versione deve essere maggiore o uguale alla 1.55. Al momento Boost è arrivato alla versione 1.71.0 e non tutte le versioni sono retrocompatibili; l'ultima versione compatibile con `vsomeip` è la 1.65.1. Le librerie sopracitate sono state scritte per essere multipiattaforma e sono disponibili nel formato sorgente, tuttavia, compilarle per una determinata piattaforma può risultare complesso.

Boost dispone di un programma chiamato Boost.Build, o più brevemente b2, che si preoccupa di compilare i file sorgenti con le giuste opzioni e di creare le librerie statiche (static) e dinamiche (shared). Questo programma prende diversi input, tra i quali: l'elenco delle librerie che deve compilare, la cartella di destinazione e la *toolchain*; quest'ultimo parametro è utile per la cross-compilazione. Di conseguenza si è scaricata la toolchain di Android [9] e si è eseguito il programma b2 con gli adeguati parametri. Questo processo però si è rivelato dare diversi errori, probabilmente

dovuti dall'incompatibilità della versione di Boost con la versione della toolchain Android.

A tal proposito la loro compilazione è stata realizzata grazie all'utilizzo di uno script caricato sul sito GitHub [10] che fa uso di docker e fornisce un Dockerfile per la compilazione. Grazie a questa tecnica non c'è bisogno di scaricare o installare nulla sul computer host, ma semplicemente seguendo la spiegazione passo passo che è presente nella parte iniziale del Dockerfile si ottengono le librerie compilate per Android.

Successivamente è stato eseguito il porting di `vsomeip` su Android; per il lavoro è stato adoperato come ambiente di sviluppo integrato (integrated development environment - IDE) Android Studio. `vsomeip` utilizza CMake (cross platform make), un software free e open-source per la definizione di processi di compilazione che funziona su più piattaforme. Esso è un tool molto modulare che è in grado di generare Makefile, inoltre dispone di una particolare sintassi comprensiva di moltissime macro ed il loro utilizzo è possibile mediante un apposito file chiamato CMakeLists.txt. Questo è il primo file che è stato modificato, in quanto va specificato che le librerie Boost non sono più da cercare tra le librerie della macchina host, funzionamento tipico di CMake, ma da caricare da uno specifico percorso, dentro al quale si trovano gli header e le librerie cross-compilate. Inoltre è stato aggiunto Android come nuova corrispondenza sul controllo del sistema operativo, al fine di poter settare i flag di compilazione adeguati o eventuali altre variazioni rispetto agli altri sistemi operativi.

Passando al codice C++, uno dei principali problemi affrontati è stato il cast dinamico tra alcuni oggetti, questo perché la classe di partenza e quella di destinazione appartengono a due moduli distinti di `vsomeip`: la libreria è divisa in più moduli e quello principale fa il caricamento dinamico degli altri moduli. Per ovviare a questo problema ho compilato `vsomeip` come un unico modulo che racchiude tutto quanto, modificando il file CMakeList.txt e escludendo la parte di codice che si occupa di fare il loading dinamico. Sebbene questo sia un workaround e modifichi la struttura originale, non inficia in alcun modo sul funzionamento della libreria.

Per rendere il codice compatibile con il nuovo sistema, si è dovuto incrementare la sua generalità e considerare tutta una serie di vincoli posti da Android, ad esempio per ragioni di sicurezza la visibilità della memoria di archiviazione è molto ristretta; ciò ha comportato molte modifiche al codice. Diverse modifiche riguardano le "define" che, ad esempio, contengono percorsi che funzionavano solamente su macchine Linux e che su Android non esistono oppure non sono accessibili dell'applicazione per restrizioni di sicurezza, o ancora, che contengono i nomi di file di librerie con estensioni non concesse dal compilatore di Android.

Le modifiche del codice sono state inserite all'interno di blocchi ifdef sulla define `__ANDROID__`, in tal modo le modifiche mantengono il codice compatibile con gli altri sistemi operativi.

4.2 Problemi aperti

`vsomeip` fa uso della Shared Memory per far comunicare processi che sono eseguiti sullo stesso calcolatore. Per renderlo possibile utilizza funzioni della libreria `shm` di Linux che però non esiste su Android. Anche se inizialmente si è provato a sostituire l'utilizzo di questa libreria con una per la gestione della memoria condivisa di Android, riscontrate diverse difficoltà, si è deciso di escludere dal porting questa funzionalità. Questa scelta è giustificabile vista la remota esigenza di far eseguire due applicazioni `vsomeip` distinte su un solo dispositivo, ricordando che si possono avere diversi servizi attivi in contemporanea in una sola applicazione.

Sempre nella porzione di codice che si occupa della comunicazione tra applicazioni diverse, viene utilizzata la libreria `pthread`, la quale possiede un'implementazione ridotta in Android rispetto a Linux.

4.3 Nuove funzionalità

Il funzionamento classico di `vsomeip` prevede l'acquisizione del proprio indirizzo di rete dal file di configurazione in formato JSON, questa modalità si presta bene a reti locali dove gli indirizzi restano statici, se però si pensa al mondo Android, in particolare a dispositivi mobili come smartphone e tablet, questo meccanismo risulta molto scomodo. Per ovviare a questo problema è stato aggiunta una nuova funzionalità al protocollo: l'acquisizione automatica dell'indirizzo IP dalla propria scheda di rete. Per poter comunque utilizzare la vecchia modalità è stato aggiunto un nuovo campo nel file di configurazione chiamato `ip_autoselected`: se il suo valore è `true` l'IP viene chiesto al sistema operativo, viceversa, se è `false` viene utilizzato l'indirizzo di rete presente nel file di configurazione.

Capitolo 5

Service Discovery

Il Service Discovery è un protocollo che rileva i dispositivi e i loro servizi in una rete di computer ed è uno strumento molto utilizzato nelle architetture service-oriented.

In questo capitolo verranno presentati i più comuni protocolli facenti parte dello stato dell'arte del Service Discovery, dopo di ché verranno confrontati fra loro e con il protocollo di Service Discovery di SOME/IP, ovvero SOME/IP–SD, che è già stato presentato nel sottocapitolo §3.4.

5.1 Definizione

Il Service Discovery è un processo di rilevamento automatico dei dispositivi e dei servizi che fanno parte di una rete. Tradizionalmente il suo scopo era ridurre gli sforzi di configurazione da parte degli utenti per l'utilizzo di risorse di rete come stampanti o server. Più recentemente, il suo utilizzo si è esteso, con lo scopo di realizzare reti basate sui servizi. Un service discovery protocol (SLP) è un protocollo di rete che consente di eseguire il rilevamento dei servizi.

Gli attori principali del Service Discovery sono:

- `Service Provider`, colui che fornisce uno o più servizi;
- `Service Consumer`, ottiene la "posizione" di offerente da un service registry e si connette al service provider;
- `Service Registry`, colui che contiene il database che contiene i servizi presenti nella rete e la loro posizione.

Ci sono due tipi di service discovery:

- `Server-side`, permettono alle applicazioni client di trovare i servizi attraverso i router o i load balancer;

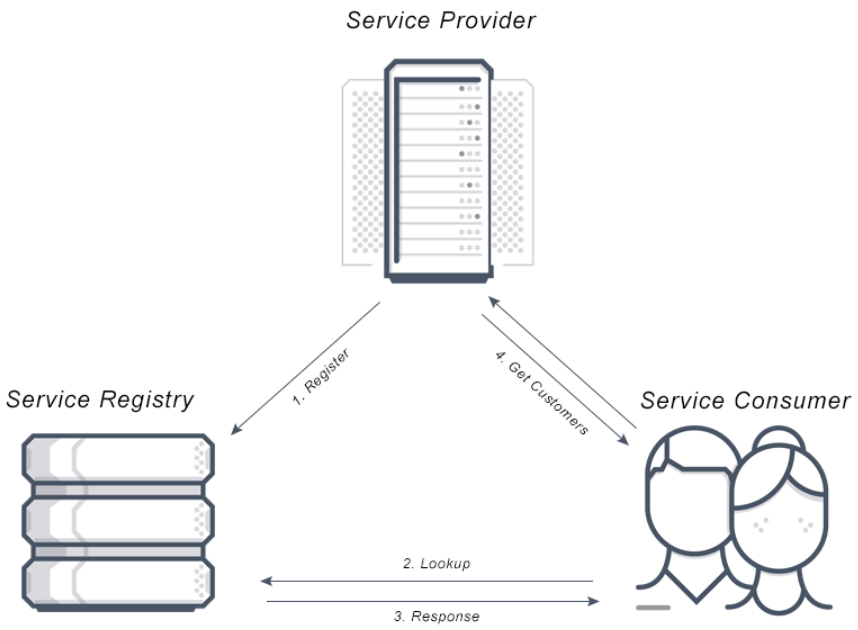


Figura 5.1: Struttura del Service Discovery. [11]

- `Client-side`, permettono alle applicazioni client di trovare i servizi cercando o interrogando un `service registry`, il quale possiede l'elenco delle istanze dei servizi;

Il `Service Registry` non è obbligatoriamente presente, soprattutto nelle reti di piccole dimensioni. In tal caso i consumatori dei servizi contattano direttamente i `Service Provider`.

5.2 Stato dell'arte

Lo studio dello stato dell'arte è utile per capire in cosa il protocollo SOME/IP–SD differisce, in cosa si distingue e quali sono le sue mancanze o comunque come può essere migliorato. In questa sezione analizzeremo tre dei più noti protocolli di Service Discovery: Zeroconf, Service Location Protocol e Universal Plug and Play.

5.3 Zeroconf

Zeroconf o Zero Configuration Networking è un protocollo standard dell'IETF per la configurazione dinamica dei nodi di una rete utilizzante il protocollo IP. È un protocollo che realizza una rete che non richiede nessuna configurazione da parte

degli utenti e nessuna amministrazione, che non dipende da infrastrutture come server DHCP, server DNS o simili.

Il gruppo di lavoro IEEE Zero Configuration Networking (Zeroconf) [12] fu formato nel settembre del 1999. Lo standard però non è mai stato definito: infatti, non esiste un RFC che descrive il protocollo, nonostante esistano già diverse implementazioni. Data la mancanza dello standard, è stato utile usufruire della documentazione di `Bonjour`, ovvero l'implementazione di Zeroconf di Apple [13].

5.3.1 Fondamenta

Zeroconf è basato su tre tecnologie:

- `IPv4 Link-Local Addressing`: vengono assegnati in automatico indirizzi link local, indirizzi che sono validi solamente all'interno dello stesso dominio di broadcast; il protocollo IPv4 riserva a questo scopo gli indirizzi 169.254.0.0/16. Grazie a questa tecnologia, non è necessario assegnare gli indirizzi ai device manualmente;
- `Multicast DNS`: è un protocollo che risolve i nomi host in indirizzi IP all'interno di piccole reti che non includono un server DNS;
- `DNS Service Discovery`: permette di scoprire la lista dei nomi dei servizi disponibili dato un tipo di servizio e risolve i nomi in indirizzi usando DNS query standard; è compatibile con il Multicast DNS, ma non dipende da esso.

A ogni servizio vengono associati tre record DNS: un service record (SRV), un pointer record (PTR) e un record di testo (TXT); quest'ultimo contiene dati che potrebbero essere necessari per risolvere o utilizzare il servizio, sebbene spesso sia vuoto.

Service Record

Il record SRV mappa il nome dell'istanza del servizio con le informazioni necessarie al client per l'utilizzo del servizio.

Il record contiene due campi: il nome dell'host e numero della porta. Il nome dell'host è il nome del dominio dove il servizio può essere trovato al momento. Il motivo per cui non viene inviato un indirizzo IP è che l'host che ospita il servizio potrebbe avere più indirizzi IP, oppure un indirizzo IPv4 e uno IPv6. Il numero di porta identifica la porta UDP o TCP associata al servizio.

I nomi assegnati ai record SRV hanno la seguente struttura:

```
<Nome dell'istanza>.<Tipo di servizio>.<Dominio>.
```

- `<Nome dell'istanza>` è il nome dell'istanza del servizio ed è una stringa Unicode con codifica UTF-8;

- `<Tipo di servizio>` è il nome di un protocollo IP standard preceduto da un underscore e seguito dal protocollo di trasporto anch'esso preceduto da un underscore;

- `<Dominio>` è un domino standard DNS.

La riga sottostante mostra un esempio di record SRV per una stampante che si chiama `PrintAlot` in esecuzione sulla porta 515 TCP:

```
PrintsAlot._printer._tcp.local.  120 IN SRV 0 0 515
                    blackhawk.local.
```

Il record è stato creato dalla stampante con nome `blackhawk.local`.

Pointer Record

I record PTR permettono di mappare il tipo di servizio con un elenco di nomi di istanze di servizio di quel determinato tipo.

Questo record è composto da una sola informazione che è il nome dell'istanza del servizio e questo è composto da `<Tipo di servizio>.<Dominio>`.

Text Record

Il record TXT contiene ulteriori informazioni riguardo l'istanza del servizio, tipicamente non più di 100 – 200 byte ed è opzionale.

Il formato dei dati è specifico per ogni tipo di servizio, quindi ognuno di questi deve definire il suo formato e pubblicarlo come parte della sua documentazione.

5.3.2 Operazioni

Le operazioni fondamentali sono tre: pubblicazione, scoperta e risoluzione.

Pubblicazione

La figura 5.3 mostra i passi fondamentali per la pubblicazione di un servizio di condivisione di musica basato su Zeroconf. Il primo passo è l'acquisizione dell'indirizzo di rete link-local, nello specifico il device sceglie un indirizzo e chiede alla rete se è disponibile, se non riceve risposta se lo assegna. Lo stesso procedimento viene ripetuto nel punto due per la selezione del nome del servizio e una volta trovato un nome univoco, esso viene avviato. In conclusione, avviene la fase di pubblicazione, dove il dispositivo annuncia alla rete il suo servizio.

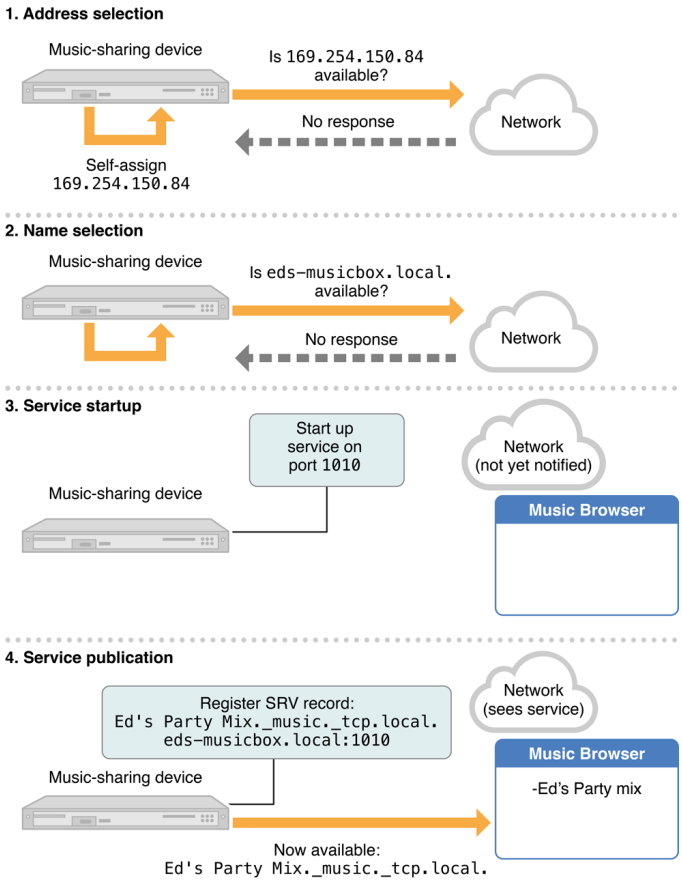


Figura 5.2: Pubblicazione di un servizio di condivisione di musica.

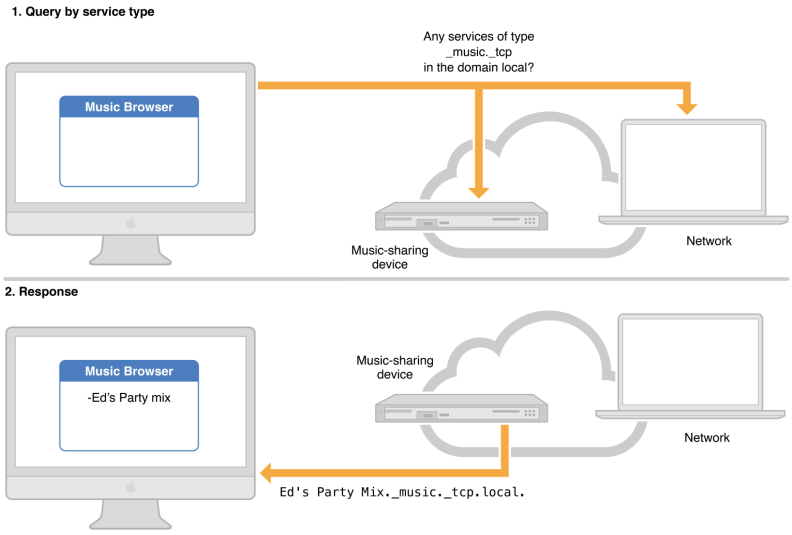


Figura 5.3: Scoperta di un servizio di condivisione di musica.

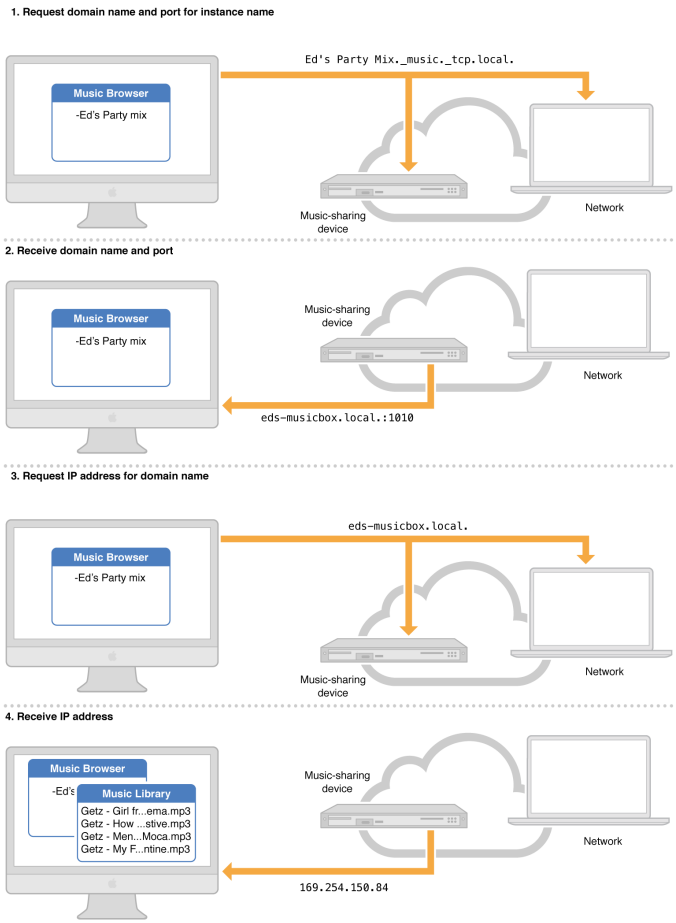


Figura 5.4: Risoluzione di un servizio di condivisione di musica.

Scoperta

Quando un'applicazione è interessata a un servizio, esegue una richiesta specificando il tipo di servizio a cui è interessato per ottenere i record PTR corrispondenti. Se nella rete è presente un'istanza del tipo richiesto riceverà la risposta e proseguirà con la risoluzione.

Risoluzione

Dopo aver ottenuto il nome dell'istanza nell'operazione precedente, il client ora necessita dei parametri di rete per poter contattare il servizio.

Per prima cosa l'applicazione esegue un DNS lookup per un record SRV, in questo modo ottiene il nome dell'host e il numero della porta. Nel terzo step, come illustrato in figura 5.4, il client fa una richiesta multicast per ottenere l'indirizzo IP e questo processo si svolge ogni volta che il servizio viene utilizzato. Questo sistema

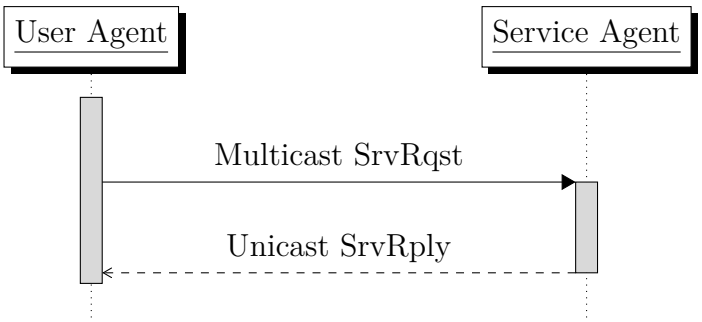


Figura 5.5: Service Request/Service Response.

di doppia richiesta fornisce due importanti funzionalità: in primo luogo, il servizio è identificato da un nome human-readable (leggibile dall'uomo) invece che da una coppia dominio-porta e in secondo luogo, il client può accedere al servizio anche se i suoi parametri di rete variano, fino a quando il suo nome rimane lo stesso.

5.4 Service Location Protocol

Il Service Location Protocol (SLP) [14] è un protocollo di service discovery che permette ai dispositivi connessi a una LAN (Local Area Network) di trovare i servizi presenti senza alcuna configurazione. Questo protocollo è stato progettato per scalare da piccole reti a grandi reti aziendali.

5.4.1 Protocollo

SLP crea un framework in base al quale le applicazioni client sono chiamate "User Agent" e i servizi sono pubblicati da "Service Agent". Una terza entità, denominata "Directory Agent" fornisce scalabilità al protocollo.

L'User Agent inoltra un "Service Request" (SrvRqst) specificando le caratteristiche del servizio richiesto dal client, successivamente riceverà una risposta (SrvRply) contenente le specifiche di tutti i servizi che soddisfano la richiesta. La Service Request può essere inviata direttamente a un Service Agent, oppure, inviata in multicast.

Nelle reti più estese vengono introdotti uno o più Directory Agent che funzionano come cache. I Service Agent comunicano i propri servizi ai Directory Agent tramite un messaggio di registrazione (SrvReg) contenente tutti i servizi che offrono; dopo averlo ricevuto, il Discovery Agent risponde con un messaggio di conferma (SrvAck). Questo procedimento deve essere ripetuto periodicamente in quanto le informazioni hanno una scadenza temporale; dopo di ché gli User Agent possono comunicare con il Discovery Agent con messaggi unicast.

La scoperta dei Directory Agent da parte sia degli User che dai Service Agent avviene in due modalità: la prima, con dei messaggi multicast (SrvRqst) al loro avvio, la seconda, grazie a dei messaggi di advertisement che vengono inviati raramente dai Discovery Agent. In entrambi i casi vengono inviati e ricevuti dei DA Advertisement (DAAdvert).

I servizi sono raggruppati in "scopes" che vengono utilizzati per indicare una location, un gruppo o delle categorie. Uno User Agent, solitamente, viene associato a uno o più "scope"; se non viene associato a nessuno in particolare risulta associato a tutti. Essere associati a uno scope significa scoprire, e quindi poter utilizzare, solo servizi di quel determinato scope.

Esiste un sistema di firme digitali grazie al quale i diversi Agent possono verificare l'identità degli altri Agent.

Ogni servizio deve possedere un URL che viene usato per localizzarlo e questo Service URL ha una struttura specifica:

`"service:"<srvtype>"://"<addrspec>.`

`<srvtype>` indica il tipo di servizio, mentre `<addrspec>` è l'hostname oppure la forma decimale dell'indirizzo IP seguito opzionalmente da ':' e numero di porta.

SLP prevede la descrizione dei servizi tramite l'uso di attributi, ovvero una coppia nome-valore. La loro presenza è opzionale.

5.5 Universal Plug and Play

Universal Plug and Play (UPnP) è un protocollo di rete che permette a diversi terminali di connettersi l'uno all'altro e di semplificare drasticamente l'utilizzo di reti domestiche e aziendali. Il termine Plug and Play in inglese significa letteralmente *Inserisci e Utilizza* e indica la possibilità di utilizzare un dispositivo non appena viene connesso al computer o alla rete.

Le tecnologie sfruttate nell'architettura UPnP sono molteplici: IP, UDP, TCP, SSDP, SOAP, GENA e XML. L'utilizzo di standard di rete permette di sfruttare le loro caratteristiche, come il supporto a diversi mezzi fisici e la compatibilità con la rete internet, in modo da ottenere una maggiore usufruibilità e quindi un'espansione di utilizzo maggiore.

UPnP Forum è un'iniziativa ideata per consentire una connettività semplice e robusta tra dispositivi e PC di molti fornitori diversi. Questa iniziativa cerca di sviluppare standard per la descrizione di protocolli e di dispositivi basati su schema XML con lo scopo di consentire l'interoperabilità da dispositivo a dispositivo in un ambiente scalabile e collegato in rete.

Questa architettura definisce due classi di device: `controlled devices` o semplicemente `devices` e `control points`. I controlled devices funzionano come ruolo di service rispondendo alle richieste provenienti dai control devices. Più

<table>
<tr><td colspan="4">UPnP Vendor</td></tr>
<tr><td colspan="4">UPnP Forum</td></tr>
<tr><td colspan="4">UPnP Device Architecture</td></tr>
<tr><td rowspan="2">SSDP</td><td rowspan="2">Multicast events</td><td>SOAP</td><td>GENA</td></tr>
<tr><td colspan="2">HTTP</td></tr>
<tr><td colspan="2">UDP</td><td colspan="2">TCP</td></tr>
<tr><td colspan="4">IP</td></tr>
</table>

Tabella 5.1: Stack del protocollo UPnP

devices, control points o entrambi possono essere eseguiti sullo stesso endpoint contemporaneamente.

Il protocollo si divide in sei step fondamentali: Indirizzamento, Scoperta, Descrizione, Controllo, Eventing e Presentazione.

5.5.1 Indirizzamento

L'indirizzamento è lo step alla base, esso consiste nel assegnamento degli indirizzi IP alle interfaccie dei devices. Nel caso in cui sia presente un server DHCP, gli indirizzi verranno assegnati da esso. Altrimenti devices e control points utilizzano Automatic IP (Auto-IP, definito nel RFC 3927 [15]) per ottenere gli indirizzi, il quale sceglierà in maniera pseudo-casuale un indirizzo di rete e verificherà che nessun altro device lo stia utilizzando facendo una richiesta ARP (Address Resolution Protocol). Il range degli indirizzi utilizzati è 169.254.0.0/16 ad esclusione del primo e dell'ultimo che sono riservati.

5.5.2 Scoperta

Quando un device è aggiunto alla rete il protocollo di discovery di UPnP permette ad esso di pubblicare il suo servizio ai control points della rete. Nello specifico invierà un messaggio di discovery in multicast per se stesso, i suoi device embedded e i suoi servizi. Se un device dispone di più interfacce dovrà inviare i messaggi di discovery su ognuna di esse.

Allo stesso modo un control point quando viene aggiunto alla rete può cercare i device e/o servizi di cui è interessato. Questa procedura viene effettuata inviando un messaggio di discovery in multicast. Tutti i devices devono essere in costante ascolto sull'indirizzo multicast e rispondere nel caso in cui se stesso o uno dei suoi

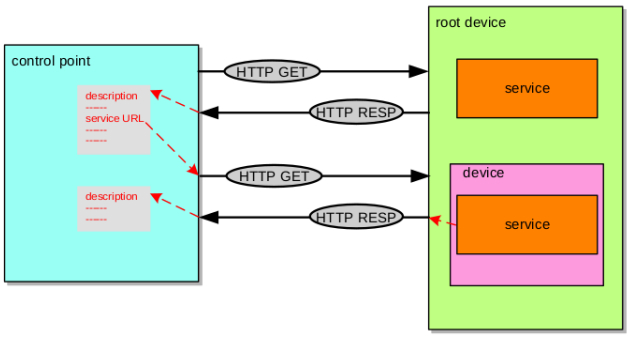


Figura 5.6: Architettura descrizione.

servizi combacino con la richiesta ricevuta. Inoltre, un control point ha la possibilità di inviare un messaggio di discovery unicast, specificando il suo indirizzo IP.

Dopo la scoperta di un servizio, se il control point è interessato a conoscere meglio il service può richiedere una sua descrizione.

Nel caso in cui un device venga rimosso dalla rete, se possibile, invierà un messaggio multicast specificando che non è più disponibile. Questa procedura viene effettuata anche quando il device cambia indirizzo IP, seguita però da un nuovo messaggio di discovery dove il service si annuncia con il nuovo indirizzo.

Questa fase di scoperta/discovery utilizza il protocollo di rete Simple Service Discovery Protocol (SSDP). Esso è un protocollo text-based basato su HTTPU, che a sua volta è basato su HTTP ma che utilizza UDP come protocollo di trasporto invece di TCP.

5.5.3 Descrizione

Un control point dopo aver scoperto un device, non ha molte informazioni su esso, se non quello che è contenuto nel messaggio di discovery: il tipo, un numero identificativo e un URL per accedere alla sua descrizione.

La descrizione viene partizionata in due:

- `descrizione del device` che include informazioni specifiche del prodotto come il nome e il numero del modello, il numero serale, il nome del produttore, URL del sito web, ecc. Inoltre contiene la lista dei servizi che contiene e alcune loro informazioni: tipo, nome del servizio, un URL per la sua descrione;

- `descrizione dei servizi` che include la lista delle azioni con i relativi argomenti e la lista delle variabili con una loro descrizione;

```
<actionList>
   <action>
      <name>actionName</name>
      <argumentList>
         <argument>
            <name>argumentNameIn1</name>
            <direction>in</direction>
            <relatedStateVariable>stateVariableName</relatedStateVariable>
         </argument>
         <!-- Declarations for other IN arguments defined by UPnP Forum working
              Committee (if any) go here -->
         <argument>
            <name>argumentNameOut1</name>
            <direction>out</direction>
            <retval/>
            <relatedStateVariable>stateVariableName</relatedStateVariable>
         </argument>
         <argument>
            <name>argumentNameOut2</name>
            <direction>out</direction>
            <relatedStateVariable>stateVariableName</relatedStateVariable>
         </argument>
         <!-- Declarations for other OUT arguments defined by UPnP Forum working
              committee (if any) go here -->
      </argumentList>
   </action>
   <!-- Declarations for other actions defined by UPnP Forum working committee
        (if any)go here -->
   <!-- Declarations for other actions added by UPnP vendor (if any) go here -->
</actionList>
```

Figura 5.7: Struttura del file XML che descrive le azioni di un servizio.

Nell'architettura UPnP un singolo device fisico può contenere più device logici, che possono essere modellati come un singolo root device con gli altri device al suo interno o come device distinti (tutti root).

La descrizione utilizza la sintassi XML e solitamente è basata su un template standard (UPnP Device Template e UPnP Service Template) che viene prodotto da UPnP Forum.

La richiesta della descrizione di un device viene effettuata tramite una richiesta HTTP GET sull'URL che si trova nel messaggio di discovery. Analogamente, viene eseguita la stessa procedura per ottenere la descrizione di un servizio, in questo caso l'URL si trova all'interno della descrizione del device che fornisce il servizio.

5.5.4 Controllo

Una volta che il control point ha ricevuto la descrizione del device e dei suoi servizi è in grado di usufruire degli stessi.

È possibile invocare un'azione, che funziona come una chiamata a funzione da remoto, e ricevere come risposta il risultato dell'azione o nel caso di un errore il messaggio d'errore. Per far ciò il control point deve inviare un control message all'URL dell'azione corrispondente che ha trovato nella descrizione ricevuta nella fase precedente.

Le azioni possono essere usate per settare i valori delle variabili di stato contenute nei servizi.

I messaggi di controllo utilizzano il protocollo SOAP.

5.5.5 Eventing

Dopo aver scoperto e ricevuto la descrizione del servizio, il control point è in grado di utilizzare l'eventing. All'interno del file di descrizione oltre alle azioni sono presenti le variabili di stato, alcune delle quali sono `evented`.

Il server pubblicherà ogni modifica delle variabili evented, e se un control point è interessato a ricevere tali informazioni può iscriversi a tali eventi.

Esistono due tipologie di eventi:

- `unicast eventing`: il control point si inscrive per rivere gli aggiornamenti;
- `multicast eventing`: le variabili sono definite come eventi multicast e i messaggi vengono inviati su UDP, in questo modo chiunque ascoltando sul indirizzo multcast riceve gli aggiornamenti.

I messaggi di eventing contengono il nome di uno o più variabili di stato e i corrispondenti valori espressi in XML.

5.5.6 Presentazione

La pagina è una pagina HTML che permette all'utente di visualizzare lo stato di un device e eventualmente controllarlo. L'indirizzo di questa pagina è presente nel file di descrizione del device.

5.6 SD a confronto

In questo sottocapitolo vengono confrontati i diversi protocolli di Service Discovery visti in precedenza nello studio dello stato dell'arte e del protocollo di discovery di SOME/IP.

In questo modo possiamo analizzare le loro differenze e capire quali funzionalità aggiuntive potrebbero essere inglobate in SOME/IP, permettendo di migliore il protocollo.

Le principali qualità che caratterizzano i service discovery sono:

- tipo di advertisement: è la modalità con cui viene scoperto un servizio. Può essere pull, quando è il client a inviare la richiesta per un servizio, o push se sono i server a pubblicare i propri servizi periodicamente;

	Zeroconf	SLP	UPnP	SOME/IP–SD
Push/Pull	Pull	Pull	Both	Both
Eventing	✗	✗	✓	✓
App-layer dependency	✗	✗	✓	✓
Service description	✗	✓ (attribute-value)	✓ (XML)	✗
Classification by type	✓	✗	✓	✗
Hierarchical categories	✗	✗	✗	✗

Tabella 5.2: Tabella confronto protocolli di Service Discovery.

- eventing: consiste nella possibilità dal client di iscriversi a un evento e di essere informato riguardo gli aggiornamenti delle variabili di stato. Questo meccanismo è trasparente per l'applicazione;
- dipendenza dal livello applicativo: quando l'applicazione per funzionare deve utilizzare delle apposite API;
- descrizione dei servizi: se le caratteristiche, come ad esempio i metodi e le variabili di stato, sono descritte dal server;
- classificazione per tipo: se i servizi sono catalogati per tipo, per esempio tutti i dispositivi che offrono la possibilità di stampare faranno parte del tipo `stampante`;
- categorie gerarchiche: quando i servizi sono classificati per tipo e questi si dispongono con una struttura gerarchica.

Come possiamo vedere dalla tabella 5.2 il protocollo più spoglio è Zeroconf seguito poi da SLP, mentre il più completo è UPnP, questo è dovuto al fatto che i primi due sono indipendenti dal livello applicativo e quindi non possono offrire servizi di alto livello. Da questo punto di vista SOME/IP–SD e UPnP si assomigliano, anche se possiamo notare dalla tabella che SOME/IP–SD è un protocollo più semplice in quanto non offre alcuna descrizione dei servizi e tanto meno una loro divisione in tipologie.

Da queste analisi si evince che il protocollo SOME/IP–SD non è provvisto di alcuna descrizione dei servizi e tanto meno di una classificazione per tipo, di conseguenza non possiede una struttura gerarchica dei servizi. Un protocollo invece provvisto di un modello per la descrizione dei servizi, come Universal Plug and

Play, permette di ridefinire le caratteristiche di quest'ultimi senza dover riscrivere il codice degli applicativi, quindi, di avere una flessibilità maggiore. Per queste ragioni si è deciso di proporre un'estensione per il protocollo SOME/IP–SD che definisce un modello di descrizione dei servizi, come sarà illustrato nel capitolo 6.

Capitolo 6

Descrizione

Come visto nel capitolo precedente, SOME/IP–SD è un protocollo che permette un tipo di advertisement sia push che pull ed è in grado di fare eventing, ma a differenza del Service Location Protocol e di Universal Plug and Play, non prevede nessuna descrizione dei propri servizi. Questa mancanza comporta una minore flessibilità dell'architettura in quanto una volta associato a un servizio una serie di metodi e variabili essi non potranno essere rimossi o non ne potranno essere aggiunti di nuovi senza andare a modificare il codice delle applicazioni.

Un possibile palliativo è andare a definire più servizi con identificativi distinti per avere più versioni dello stesso servizio ma con caratteristiche diverse. Questo però non permette di definire con flessibilità un servizio, anche solo per andare a modificare un singolo parametro bisogna definire un nuovo servizio con un nuovo ID associato, che inoltre i client e i server devono riconoscere come una nuova versione e quindi vanno modificate anche le loro applicazioni.

Questo capitolo presenta la proposta di estensione del protocollo SOME/IP per la descrizione dei servizi a bordo veicolo. Nella prima parte viene descritto il modello che è stato definito nella tesi, in particolare modo, quali campi sono necessari per descrivere un servizio. Nella seconda parte viene proposto come integrare la descrizione all'interno dell'implementazione `vsomeip`.

6.1 Modello di descrizione

Per poter descrivere le caratteristiche di un servizio, ma in generale un qualunque oggetto nell'informatica, è necessario definire un modello. In tal modo, le due o più parti che si scambiano le informazioni sulle caratteristiche, riescono a dare un significato a tali informazioni. Questo concetto lo possiamo ritrovare anche in Universal Plug and Play, come visto nel capitolo 5, il quale definisce degli `schema` XML chiamati UPnP Device Template e UPnP Service Template.

Listato 6.1: Schema JSON per la descrizione.

```
{
    "$schema": "http://json-schema.org/schema#",
    "type": "object",
    "properties":
    {
        "id": { "type": "string", "pattern": "^(0x)[a-fA-F0-9]+$" },
        "name": { "type": "string" },
        "description": { "type": "string" },
        "version": { "type": "string" },
        "priority": { "type": "string" },
        "variables":
        {
            "type": "array",
            "items": { "ref": "schema_variable.json#/definitions/variable" }
        },
        "methods":
        {
            "type": "array",
            "items": { "ref": "schema_method.json#/definitions/method" }
        }
    },
    "required": ["variables", "methods"]
}
```

Per prima cosa bisogna capire quali caratteristiche vanno descritte e realizzare un modello che le contenga; compito del seguente sottoparagrafo è appunto descrivere il modello scelto.

6.1.1 Definizione del modello

In SOME/IP esistono due modalità di funzionamento principali: la chiamata di procedure da remoto (RPC: Remote Procedure Call) e l'eventing. Nel primo caso il client effettua una chiamata a una funzione di un servizio passando gli adeguati parametri e ricevendo, ove previsto, il valore di ritorno come risposta. Per quanto riguarda l'eventing, il client fa una sottoscrizione a una variabile di stato di un servizio e da quel momento in poi riceverà tutti gli aggiornamenti riguardanti quella determinata variabile. Perciò la descrizione sarà suddivisa in due sezioni: una per descrivere i metodi e una per le variabili.

Visto l'utilizzo dei file JSON da parte di `vsomeip` nei file di configurazione, anche la descrizione dei servizi è stata strutturata su file JSON. Il modello è descritto grazie a degli schema nel medesimo formato, prendendo come standard di riferimento JSON Schema[1].

Come mostrato nel Listato 6.1, il modello contiene due campi:

- `id`, è un numero esadecimale che identifica in modo univoco il servizio;

[1]JSON Schema è un vocabolario che consente di annotare e convalidare i documenti JSON. [16]

- `name`, è una stringa che contiene il nome leggibile del servizio;
- `description`, contiene una breve descrizione human-readable del servizio;
- `version`, specifica la versione del servizio, in questo modo possono esistere i servizi possono essere aggiornati o essere presenti in più varianti;
- `priority`, specifica la priorità che è attribuita a una versione, più il valore è basso e più la priorità è alta;
- `variables`, è un array che comprende l'elenco delle variabili presenti nel servizio. La definizione del modello dei suoi `items`, ovvero le variabili, si trova in un file separato chiamato "schema_variables.json";
- `methods`, che contenente l'elenco dei metodi, anch'essi definiti in un file esterno nominato "schema_method.json".

Come possiamo notare dal campo `required`, entrambi i vettori devono essere obbligatoriamente presenti, però possono anche non contenere elementi al loro interno nel caso non siano presenti metodi o variabili nel servizio.

Passiamo ora alla definizione del modello per quanto riguarda le variabili.

Le caratteristiche che si è deciso di descrivere sono:

- `id`, è un codice identificativo univoco della variabile, non ha una funzione in SOME/IP;
- `name`, è la stringa che contiene il nome della variabile;
- `unit`, contiene una stringa che indica l'unità di misura in cui è espressa la variabile considerata;
- `type`, specifica il tipo della variabile e può essere:
  - `integer`, cioè un valore intero;
  - `float`, ovvero un numero reale;
  - `boolean`, ovvero una variabile che può essere vero o falso.
  - `string`, cioè una stringa;
  - `integer_array`, è un vettore di interi;
  - `float_array`, ovvero un vettore che contiene dei numeri reali;
  - `boolean_array`, un array di booleani;
  - `string_array`, ovvero un vettore di stringhe.
- `min_value`, è il minimo valore assegnabile alla variabile;

Listato 6.2: Schema JSON per la descrizione delle variabili.

```
{
    "$schema": "http://json-schema.org/schema#",
    "definitions": {
        "variable": {
            "type": "object",
            "properties":
            {
                "id": { "type": "string", "pattern": "^(0x)[a-fA-F0-9]+$" },
                "name": { "type": "string" },
                "unit": { "type": "string" },
                "type": {
                    "type": "string",
                    "enum": [ "integer", "float", "boolean", "string", "integer_array",
                        "float_array", "boolean_array", "string_array"]
                },
                "min_value": { "type": "number" },
                "max_value": { "type": "number" },
                "resolution": { "type": "number" },
                "invalid_value": { "type": "number" },
                "description": { "type": "string" },
                "eventgroup_id": { "type": "string", "pattern": "^(0x)[a-fA-F0-9]+$" },
                "event_id": { "type": "string", "pattern": "^(0x)[a-fA-F0-9]+$" },
                "getter_id": { "type": "string", "pattern": "^(0x)?a-fA-F0-9]+$" },
                "setter_id": { "type": "string", "pattern": "^(0x)[a-fA-F0-9]+$" }
            },
            "required": ["id", "type", "" ]
        }
    }
}
```

- `max_value`, è il valore massimo. Il suo valore e quello di `min_value` devono essere compatibili con il `type`, per esempio se type è uguale a integer, min_value non può valere 0.5;
- `resolution`, è la variazione minima che può avere la variabile. Anche questa grandezza deve essere compatibile con il tipo della variabile (per esempio, non ha senso una variazione dello 0.1 se il valore di `type` è `integer`). Nel caso non sia presente la resolution nel file di descrizione, viene intesa la più bassa possibile: per gli interi è 1, mentre nel caso di valori `float` si intende la variabile come continua;
- `invalid_value`, è il valore che viene assegnato alla variabile quando non si trova in uno stato corretto;
- `description`, contiene una breve descrizione testuale della variabile;
- `eventgroup_id`, indica l'eventgroup a cui appartiene la variabile;
- `event_id`, è l'identificativo dell'evento.
- `getter_id`, è l'ID del metodo che può essere usato per ottenere il valore corrente della variabile;
- `setter_id`, il suo valore corrisponde al codice identificativo del metodo utile per settare il valore della variabile.

Sempre dal Listato 6.2, possiamo osservare quali proprietà siano richieste obbligatoriamente, i campi obbligatori sono solamente l'`id` e il `type`, ciò significa che alcuni campi devono possedere dei valori di default. I valori massimi e minimi se non presenti nella descrizione vengono intesi +infinito e -infinito, mentre la mancanza della risoluzione equivale al valore minimo possibile. L'`invalid_value` se non specificato non esiste, lo stesso vale per il getter, setter, eventgroup e event.

Ci sono dei campi all'interno del modello che non sono compatibili in determinate situazioni: se la variabile è booleana non ha senso che siano presenti `min_value`, `max_value` e tanto meno `resolution`, anche se non esiste alcun vincolo nel modello.

Nel modello possiamo fare una distinzione tra le proprietà in base alla loro utilità: proprietà utili per il funzionamento del protocollo e proprietà utili per l'interazione con l'utente. I campi come `type`, `invalid_value`, `eventgroup_id`, `event_id`, `setter_id` e `getter_id` sono utili per l'interazione tra client e server nel protocollo SOME/IP e per la logica di funzionamento delle applicazioni. Altri campi, come `description`, `unit` e `name`, non sono necessari per il funzionamento ma sono un valore aggiunto molto utile per l'usabilità da parte dell'utente e quindi per l'interfaccia grafica. Grazie alla descrizione dei servizi, i client sono in grado di costruire interfacce grafiche custom per il servizio stesso, pur non riuscendo

a interpretare l'utilità delle variabili, in quanto questo compito viene demandato all'utente finale.

Sebbene il protocollo SOME/IP distingua le modalità di funzionamento in Remote Procedure Call e Eventing, la variabile descritta fin'ora non corrisponde alla sola variabile di stato associata ad un evento, ma anche a variabili non "evented". Di fatti tra le proprietà troviamo `eventgroup_id` e `event_id` ma non fanno parte dei campi obbligatori. Una variabile non associata ad un evento, infatti, può avere associato un `setter_id` e/o un `getter_id`. Una modalità di funzionamento però non esclude l'altra, una variabile può essere sia associata ad un evento che avere un metodo associato per essere settata e uno per essere ottenuta. Allo stesso modo, a livello teorico, può non avere né un setter né un getter e né un evento associato, questa configurazione però rende del tutto inutilizzabile la variabile, in quanto il client sa della sua esistenza ma non ha una modalità per ottenerla o settarla.

I metodi getter e setter non fanno parte dei metodi descritti, quindi non li troveremo all'interno dell'array `methods`. Questo perché hanno una struttura standard:

- il getter, ovvero la funzione che ritorna il valore della variabile, non ha nessun argomento e ritorna sempre un valore compatibile con la variabile associata, per esempio, prendendo come riferimento sempre il Listato 6.2, il getter della variabile temperatura è: `int getTemperature();`
- il setter invece, è il metodo che serve a impostare un valore alla variabile e prende sempre un solo parametro che è il valore che vogliamo settare e non ritorna mai nulla. Per esempio il setter della temperature è: `void setTemperature(int value);`.

Per quanto riguarda le proprietà che rappresentano un identificativo come, ad esempio, `eventgroup_id` e `getter_id`, oltre al campo `type`, è presente il `pattern`. Come suggerito dal nome, indica il pattern che il valore deve possedere, e il suo valore è un'espressione regolare (regex). La prima parte, `(0x)` indica che il valore deve iniziare con la stringa "0x", mentre la parte successiva `[a-fA-F0-9]` significa che deve continuare con un valore alfanumerico le cui lettere vanno dalla A alla F; questo è il pattern comune per scrivere numeri esadecimali. La regex è utile per poi validare la descrizione attraverso lo schema. È stato scelto il formato esadecimale in quanto SOME/IP utilizza questo formato anche per gli altri identificativi.

Passiamo ora alla descrizione del modello dei metodi; il relativo schema JSON è contenuto in un file chiamato "schema_method.json" ed è illustrato nel Listato 6.3.

Analizziamo ora le caratteristiche del modello per la descrizione dei metodi, il quale è costituito dai seguenti campi:

- `id`, è il numero univoco che identifica un metodo;
- `name`, è il nome del metodo;

Listato 6.3: Schema JSON per la descrizione dei metodi.

```
{
    "$schema": "http://json-schema.org/schema#",
    "definitions": {
        "method": {
            "type": "object",
            "properties":
            {
                "id": { "type": "string", "pattern": "^(0x)[a-fA-F0-9]+$" },
                "name": { "type": "string" },
                "arguments":
                {
                    "type": "array",
                    "items":
                    {
                        "type": "object",
                        "properties":
                        {
                            "name": { "type": "string" },
                            "type": {
                                "type": "string",
                                "enum": ["integer", "float", "boolean", ""]
                            },
                            "min_value": { "type": "number" },
                            "max_value": { "type": "number" },
                            "description": { "type": "string" }
                        },
                        "required": ["type"]
                    }
                },
                "retval":
                {
                    "type": "object",
                    "properties":
                    {
                        "name": { "type": "string" },
                        "type": {
                            "type": "string",
                            "enum": ["integer", "float", ""]
                        },
                        "invalid_value": { "type": "number" },
                        "error_value": { "type": "number" },
                        "description": { "type": "string" }
                    },
                    "required": ["type"]
                },
                "description": { "type": "string" }
            },
            "required": ["id", "arguments"]
        }
    }
}
```

- `arguments`, è la lista degli argomenti necessari per l'utilizzo del metodo, ogni argomento viene descritto grazie alle seguenti proprieta:
  - `name`, è il nome leggibile dell'argomento;
  - `type`, la stringa che indica di che tipo è la variabile, i valori possibili sono: integer e float.
  - `min_value`, è il valore minimo assegnabile all'argomento;
  - `max_value`, è il limite massimo valido per l'argomento;
  - `description`, consiste in una breve descrizione testuale dell'argomento.

  L'unica proprietà necessaria per il funzionamento è il `type`;
- `retval`, è il valore di ritorno della funzione ed è caratterizzato da:
  - `name`, che è il nome human-readable del valore;
  - `type`, specifica il tipo a cui appartiene (integer, float o boolean);
  - `invalid_value`, è il valore che il metodo ritorna quando uno o più argomenti sono invalidi;
  - `error_value`, è il valore che viene ritornato dal metodo nel caso in cui si sia verificato un errore di natura differente agli argomenti invalidi;
  - `description`, è una breve descrizione testuale del valore di ritorno del metodo.

  Anche per il valore di ritorno l'unico campo obbligatorio è il tipo;
- `description`, è un breve messaggio di testo che descrive il metodo.

Nella descrizione dei metodi, come in quella delle variabili, gli ID hanno un valore esadecimale nella forma 0x.

6.1.2 Esempi

Al fine di chiarire quanto presentato fin'ora, si presentano di seguito alcuni esempi che rispettano i modelli definiti nel sottocapitolo precedente.

Il primo esempio che andiamo a vedere è il file JSON nel Listato 6.4, nel quale abbiamo due variabili: la prima è identificata dall'ID 0x2222, è una variabile di tipo intero ed è compresa tra 0 e 100 con una risoluzione uguale a 1. Dal nome e dalla descrizione possiamo capire che indica la temperatura a bordo veicolo e, dal valore di `unit`, che la sua unità di misura sono i gradi centigradi. Inoltre, dalla presenza delle proprietà `eventgroup_id` e `event_id` riconosciamo che la variabile è di tipo "evented" ed è quindi possibile iscriversi per ottenere i suoi aggiornamenti. La variabile in questione ha anche un metodo setter associato ma non un getter, il che

Listato 6.4: Esempio di descrizione delle variabili.

```
{
    "variables":
    [
        {
            "id": "0x2222",
            "name": "Temperature",
            "unit": "°C",
            "type": "integer",
            "min_value": 0,
            "max_value": 100,
            "resolution": 1,
            "description": "Indicates the temperature on board the vehicle.",
            "eventgroup_id": "0x8010",
            "event_id": "0x0010",
            "setter_id": "0x0001"
        },
        {
            "id": "0x2223",
            "type": "float",
            "eventgroup_id": "0x8020",
            "event_id": "0x0020"
        }
    ]
}
```

significa che l'unico modo per conoscere il suo valore è attraverso gli eventi.
La seconda variabile con ID uguale a 0x2223 è di tipo float ed è associata a un evento. Oltre a questo però non è presente nessuna descrizione testuale come per la temperatura, rendendo così incomprensibile il suo scopo con il solo file JSON a disposizione.

Descrizioni accurate come quelle della temperatura permettono al client di creare un'interfaccia grafica ad-hoc, che si adatta alla descrizione ricevuta, visualizzando il valore della variabile, la sua unità di misura e una descrizione. In questo modo un utente è in grado di interpretare i valori visualizzati e quindi dargli un significato. Altri campi come `min_value`, `max_value` e `resolution` permettono la creazione di strumenti grafici utili per la modifica del valore delle variabili, come ad esempio slider, checkbox, ecc. In questo modo il client può essere completamente ignaro della funzione delle variabili ma il suo compito è solamente la creazione dell'interfaccia grafica e la comunicazione con il server.

Passiamo ora all'analisi delle descrizione riportata nel Listato 6.5, la quale contiene l'array `methods` con un solo elemento. Quest'ultimo descrive un metodo il cui ID è 0x1111 e il nome è "Addition" ovvero "Addizione". Il campo `arguments` contiene due elementi molto simili tra loro, dalla descrizione e dal nome possiamo capire che si tratta dei due addendi di tipo float e che devono avere un valore compreso tra 0 e 1000. Il campo `retval`, invece, contiene la descrizione del valore di ritorno, il cui nome è "Sum" ed è di tipo float. Viene inoltre definito un valore per `invalid_value`: nel caso in cui manchi uno o più argomenti oppure siano

Listato 6.5: Esempio di descrizione di un metodo.

```
{
    "methods":
    [
        {
            "id": "0x1111",
            "name": "Addition",
            "arguments": [
                {
                    "name": "Addend 1",
                    "type": "float",
                    "min_value": 0,
                    "max_value": 1000,
                    "description": "First addend of the addition."
                },
                {
                    "name": "Addend 2",
                    "type": "float",
                    "min_value": 0,
                    "max_value": 1000,
                    "description": "Second addend of the addition."
                }
            ],
            "retval":
            {
                "name": "Sum",
                "type": "float",
                "invalid_value": -1,
                "error_value": -1000,
                "description": "The sum of the addition."
            },
            "description": "This method do the addition between two float addend."
        }
    ]
}
```

errati, per esempio abbiano un valore negativo, il metodo ritornerà -1. Nel caso in cui si verifichi un errore di altro genere ritornerà -1000 (`error_value`).

6.2 Integrazione in SOME/IP

Definito il modello, il posso successivo è l'integrazione del file di descrizione all'interno del protocollo SOME/IP.

Come visto nel capitolo 3, il protocollo SOME/IP permette la scoperta dei servizi tramite i messaggi di Service Discovery, i quali possono contenere diversi `options` per ogni servizio. La descrizione potrebbe essere inserita in un `option` dedicato a tale funzione.

Osservando la figura 6.1 possiamo notare che al servizio S2 vi è associata una sola `option` ovvero la numero 1, la quale specifica l'indirizzo IP e la porta in cui si trova. Il servizio S9 invece, ha due `option` associate: la prima (option 2) che contiene i parametri di rete e la seconda (option 3) che contiene la descrizione del

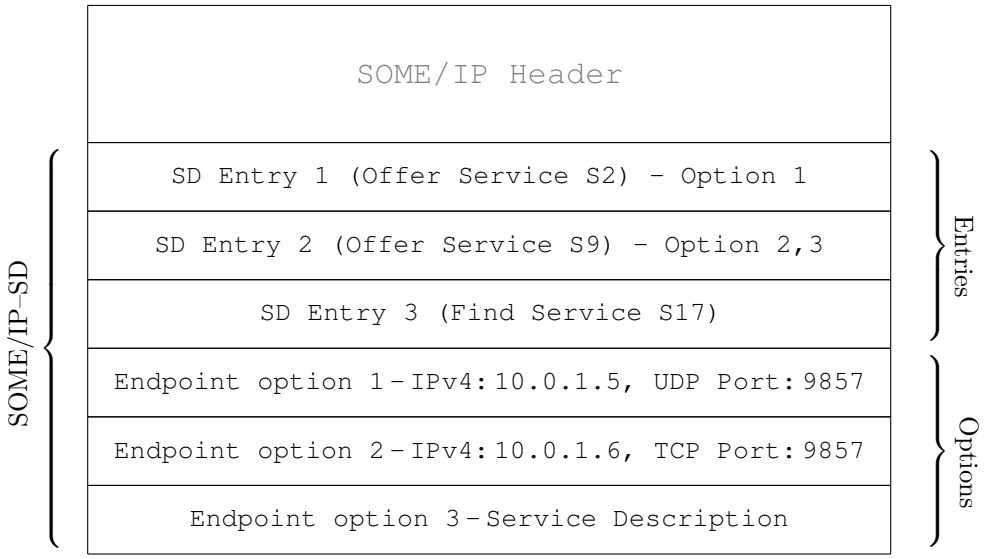


Figura 6.1: Struttura di un messaggio SOME/IP–SD.

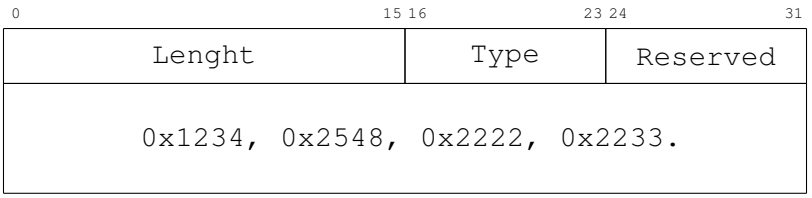


Figura 6.2: Esempio di `option` con elenco di variabili e metodi.

servizio stesso. In questo modo appena viene scoperto un servizio, se presente, il client riceve anche la sua descrizione.

Definito quando e dove inviare le informazioni, bisogna valutare quali informazioni vanno inviate. Nella parte seguente dello scritto saranno valutate due alternative.

La prima idea è quella di salvare i file JSON, contenenti le descrizioni, nella memoria dei client e in quelle dei server in modo da evitare la loro completa trasmissione nel pacchetto SOME/IP–SD. In questa soluzione i client possiedono una descrizione di tutti i metodi e di tutte le variabili che possono essere presenti nei servizi; i server invece conterranno le descrizioni di metodi e variabili presenti nei suoi servizi. Ciò consente di mantenere ridotta la dimensione del pacchetto in quando non è necessario inviare il file di descrizione completo; in figura 6.2 possiamo vedere un esempio di `option` con la lista dei codici esadecimali identificativi (più brevemente, ID).

Quando il client riceve il messaggio di Service Discovery cerca se è presente un servizio di suo interesse e se lo trova controlla i vari `options` a esso associati. Per ogni `option` controlla il suo `type` e si comporta di conseguenza. Nel caso in cui esso sia uguale a `list_var_meth_type` il client sa che il payload dell'`option`

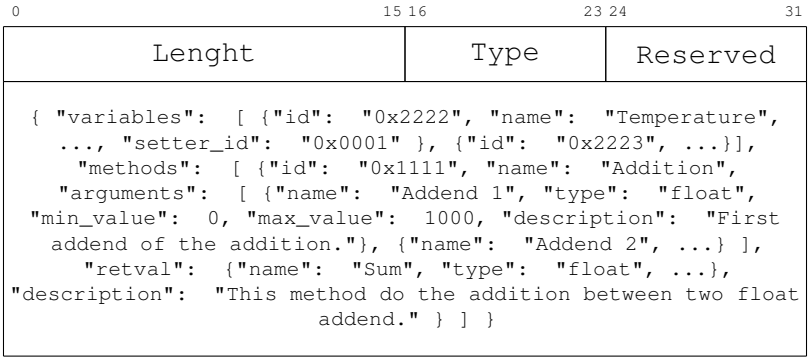


Figura 6.3: Esempio di `option` con file di descrizione.

contiene la lista di metodi e variabili. Dopo aver letto la lista, cerca le descrizioni associate nei file che possiede in locale e si comporta di conseguenza. Nel caso non trovi il match per un determinato ID, quella variabile/metodo sarà per lui sconosciuta e inutilizzabile.

Il vantaggio di questo metodo è che permette di avere messaggi più piccoli e di conseguenza di avere un tempo di trasmissione inferiore. D'altro canto però non è possibile definire nuove variabili o metodi senza andare a modificare i file contenuti su tutti i client della rete.

L'alternativa al metodo precedentemente illustrato consiste nell'inviare il file di descrizione completo all'interno del pacchetto di Service Discovery, nello specifico, dentro l'`option` dedicato allo scopo. In figura 6.3 troviamo un esempio.

Il vantaggio principale di questa modalità di funzionamento è la possibilità di creare un nuovo metodo o una nuova variabile o di modificarne una esistente e di non dover andare a modificare nulla negli altri device della rete. Inoltre, se pensiamo all'eventualità di possedere un grande numero di metodi e variabili si riduce lo spazio di memoria di massa occupata sui client che non devono più avere la descrizione di ognuno di essi salvata in locale.

Le due soluzioni non sono però incompatibili, riservando due codici differenti per il `type` degli `options` possiamo decidere di utilizzare uno piuttosto che l'altro in base alle esigenze del caso d'uso.

Una terza soluzione è un ibrido delle due precedenti: la descrizione non viene fornita ogni volta all'interno del pacchetto SOME/IP–SD, ma nel caso in cui il client non possieda la descrizione, può richiederla tramite un messaggio specifico. Una volta ricevuta la descrizione la aggiunge al suo database locale.

Capitolo 7

Dimostratore

L'ultimo obiettivo della tesi è stato la realizzazione di un dimostratore, con lo scopo di illustrare le potenzialità del lavoro svolto. Questa implementazione vuole risaltare le funzionalità aggiuntive svolte grazie alla descrizione dei servizi e la possibilità di sfruttare il protocollo `vsomeip` anche su dispositivi Android.

Il dimostratore simula il funzionamento di un sistema di climatizzazione dell'auto ed è composto da due board contenenti i servizi, da un Rasperry che simula la dashboard dell'autoveicolo e da un tablet Android per il controllo del sistema. La loro comunicazione è realizzata in parte su cavo Ethernet e in parte in Wifi, grazie a un router/switch.

È stato scelto il sistema dell'aria condizionata in quanto è un componente presente su quasi tutti gli autoveicoli odierni ed è quindi molto noto, inoltre è composto da diversi parametri ed è facile immaginare che una casa automobilistica possa offrire diverse versioni ai clienti.

Nella parte restante del capitolo verranno descritti in dettaglio i singoli componenti della demo.

7.1 Servizi

Due componenti fondamentali del dimostratore sono i servizi, che risiedono su due board differenti. Ognuno di essi è un'applicazione realizzata in C++, che utilizza la libreria `vsomeip`. Entrambi implementano il servizio di aria condizionata ma in due versioni differenti.

Le due versioni di aria condizionata sono la *basic* e la *premium*; come intuibile dai loro nomi, una realizza una versione più semplice, composta quindi da meno parametri, mentre l'altra è più completa. Ognuno dei due servizi possiede una descrizione conforme al formato definito nel lavoro di tesi.

Come possiamo notare dai Listati 7.1 e 7.2, i due servizi possiedono lo stesso id in quanto offrono lo stesso servizio ma in due versioni diverse. Inoltre possiamo notare

.

Listato 7.1: Servizio basic.

```
1      "id": "0x0010",
2      "name": "Air Conditioner Basic",
3      "description": "This service offers
           a basic air conditioner.",
4      "version": "0x0002",
5      "priority": "0xfff2",
```

Listato 7.2: Servizio premium.

```
1      "id": "0x0010",
2      "name": "Air Conditioner Premium",
3      "description": "This service offers
           a premium air conditioner.",
4      "version": "0x0006",
5      "priority": "0xff45",
```

che il servizio premium ha una priorità più alta, ovvero un valore di `priority` inferiore; in questo modo, nel caso entrambi i servizi siano disponibili, il client sceglierà il servizio premium.

Continuando con l'analisi dei file di configurazione, nei Listati 7.3 e 7.4, possiamo notare che il servizio premium ha la suddivisione della temperatura in due zone: lato guidatore e lato passeggero. Inoltre le temperature sono settabili da un minimo di 15 °C e 30 °C, mentre nel servizio basic il range va da 18 °C a 25 °C. Un'altra differenza riguarda il parametro `power` che nel servizio basic ha una `resolution` uguale a 25, consentendo così solo 5 stati (0, 25, 50, 75 e 100), mentre il servizio premium ha una `resolution` pari a 1, permettendo così un settaggio molto più fine della potenza.

Entrambe le versioni possiedono delle variabili booleane per impostare diverse funzionalità, che sono:

- `AC`, per accendere o spegnere l'aria condizionata;
- `Low`, per abilitare/disabilitare la fuoriuscita dell'aria dalle bocchette inferiori;
- `High`, per abilitare/disabilitare la fuoriuscita dell'aria dalle bocchette superiori;
- `Defog`, per abilitare/disabilitare lo sbrinamento del parabrezza;
- `Recirculate`, per abilitare/disabilitare il ricircolo dell'aria.

Tutte le variabili sono associate ad un `eventgroup` e ad un `event` per permettere il paradigma di funzionamento publish/subscribe. Inoltre possiedono un setter, grazie al quale è possibile impostare il valore delle variabili dal client.

7.2 Dashboard

Un altro componente fondamentale del dimostratore è la dashboard, ovvero colui che simula il cruscotto dell'automobile. Esso è composto da due blocchi: da una parte, da un'interfaccia grafica che riproduce un pannello di controllo del climatizzatore, e dell'altra, da un modulo di comunicazione che riceve e invia dati sulla rete.

Listato 7.3: Servizio basic.

```
"variables":
[
    {
        "id": "0x0003",
        "name": "AC",
        "type": "boolean",
        "eventgroup_id": "0x0030",
        "event_id": "0x8030",
        "setter_id": "0x0003"
    },
    {
        "id": "0x0001",
        "name": "Temperature",
        "unit": "°C",
        "type": "integer",
        "min_value": 18,
        "max_value": 25,
        "resolution": 1,
        "eventgroup_id": "0x0010",
        "event_id": "0x8010",
        "setter_id": "0x0001"
    },
    {
        "id": "0x0002",
        "name": "Power",
        "unit": "%",
        "type": "integer",
        "min_value": 0,
        "max_value": 100,
        "resolution": 25,
        "eventgroup_id": "0x0020",
        "event_id": "0x8020",
        "setter_id": "0x0002"
    },
```

Listato 7.4: Servizio premium.

```
"variables":
[
    {
        "id": "0x0003",
        "name": "AC",
        "type": "boolean",
        "eventgroup_id": "0x0030",
        "event_id": "0x8030",
        "setter_id": "0x0003"
    },
    {
        "id": "0x0001",
        "name": "Temp. Driver",
        "unit": "°C",
        "type": "integer",
        "min_value": 15,
        "max_value": 30,
        "resolution": 1,
        "eventgroup_id": "0x0010",
        "event_id": "0x8010",
        "setter_id": "0x0001"
    },
    {
        "id": "0x0001",
        "name": "Temp. Passenger",
        "unit": "°C",
        "type": "integer",
        "min_value": 15,
        "max_value": 30,
        "resolution": 1,
        "eventgroup_id": "0x0010",
        "event_id": "0x8010",
        "setter_id": "0x0001"
    },
    {
        "id": "0x0002",
        "name": "Power",
        "unit": "%",
        "type": "integer",
        "min_value": 0,
        "max_value": 100,
        "resolution": 1,
        "eventgroup_id": "0x0020",
        "event_id": "0x8020",
        "setter_id": "0x0002"
    },
```

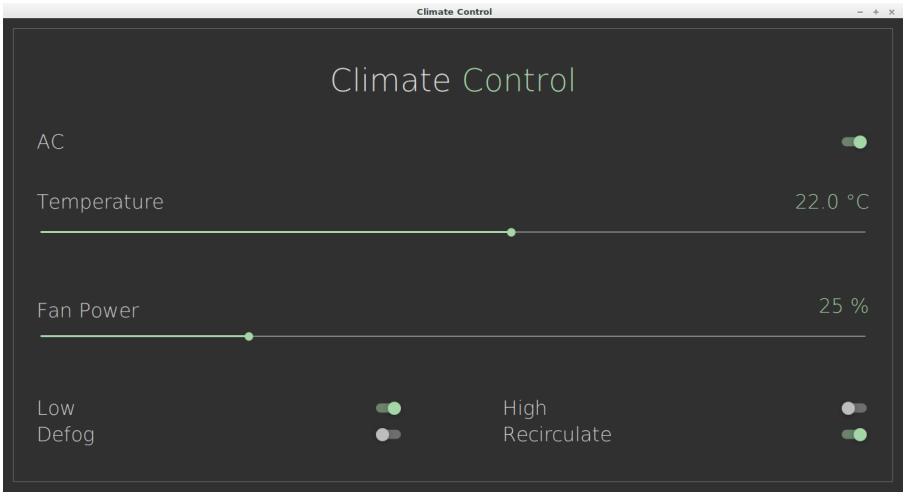


Figura 7.1: Dashboard del servizio basic.

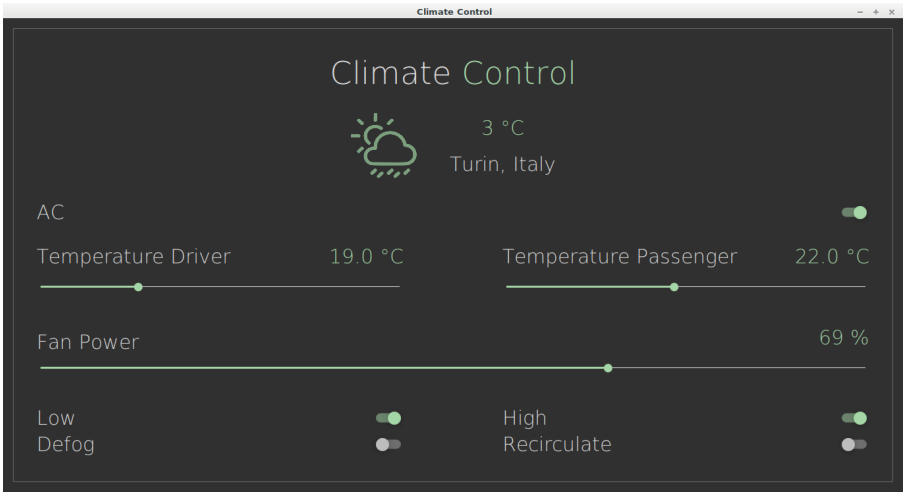
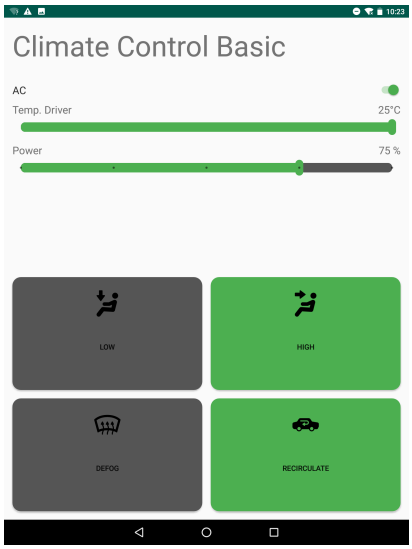


Figura 7.2: Dashboard del servizio premium.

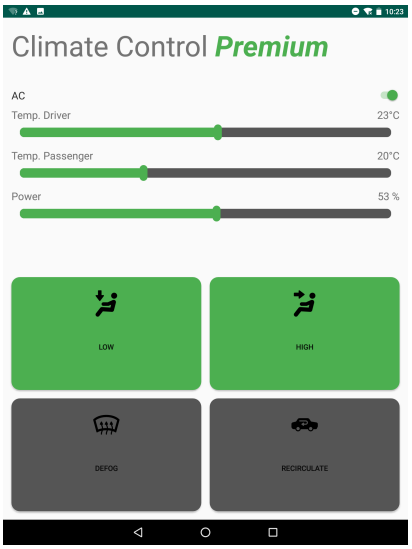
L'interfaccia grafica è stata realizzata con l'utilizzo del framework Qt, in particolare con il linguaggio QML, che permette di creare interfacce belle esteticamente.

Il modulo di comunicazione, invece, è scritto in C++ e utilizza la libreria `vsomeip`. Esso prende il ruolo di client ed è in ascolto per i due servizi, basic e premium. Quando nessun servizio è disponibile, viene visualizzata un'interfaccia di attesa, mentre viene visualizzata l'interfaccia di controllo nel caso un servizio sia disponibile. Le interfacce, come si può vedere dalle Figure 7.1 e 7.2, differiscono in base alla versione disponibile.

Nel caso in cui entrambi i servizi siano disponibili, il client sceglie quello con la priorità maggiore, in questo caso il servizio premium.



(a) Interfaccia Android con servizio basic.



(b) Interfaccia Android con servizio premium.

7.3 Android

L'ultimo componente del dimostratore è un tablet Android, sul quale è presente un'applicazione che si comporta da client. Esso simula il dispositivo di un passeggero, che grazie all'app riesce a visualizzare e controllare i parametri del climatizzatore.

L'applicazione è composta da una logica sviluppata in C++, che sfrutta la libreria `vsomeip`, per realizzare un client SOME/IP; quest'ultimo si comporta esattamente come la dashboard. La parte grafica invece è realizzata in XML e Java, come di consueto per Android. La comunicazione tra queste due parti è realizzata grazie a JNI (Java Native Interface), un framework del linguaggio Java che consente di richiamare, o essere richiamato da, codice "nativo".

7.4 Conclusioni

Collegando i servizi alla rete, la dashboard presente sul Raspberry e l'applicazione Android, visualizzano l'interfaccia corretta grazie ai file di descrizione ricevuti. Non appena vengono scollegati i servizi, non ricevendo più i messaggi di advertisement, l'applicazione client se ne accorge e visualizza un'interfaccia di attesa. Nel caso in cui entrambi i servizi siano collegati alla rete, le due applicazioni client visualizzano l'interfaccia e utilizzano il servizio premium, grazie al fatto che la priorità più alta è assegnata a questa versione.

Capitolo 8

Conclusioni

In questa tesi è stato definito un modello per la descrizione dei servizi SOME/IP, che consente di definire i servizi in maniera più flessibile e meno cablati all'interno del codice dell'applicazione. Inoltre permette di avere diverse versioni di uno stesso servizio con parametri differenti e con delle priorità associate. La descrizione, in aggiunta, permette di realizzare un'interfaccia custom in base ai parametri contenuti nei file. Tutto ciò consente di fare l'upgrade di alcuni servizi a bordo veicolo, come il sistema di climatizzazione o quello multimediale, senza dover aggiornare ogni componente e quindi di essere economici e meno invasi.

Il lavoro di tesi ha incluso anche la cross-compilazione della libreria `vsomeip` per Android, consentendo l'utilizzo di questo sistema per la realizzazione di infotainment, sfruttando così tutti i suoi vantaggi. Un altro aspetto importate del porting è la possibilità di realizzare applicazioni SOME/IP per i dispositivi personali come smartphone e tablet, permettendo così a questi device di interagire con l'infotainment.

Bibliografia

[1] «IEEE Standard for Ethernet Amendment 1: Physical Layer Specifications and Management Parameters for 100 Mb/s Operation over a Single Balanced Twisted Pair Cable (100BASE-T1)». In: *IEEE Std 802.3bw-2015 (Amendment to IEEE Std 802.3-2015* (mar. 2016), pp. 1–88. DOI: 10.1109/IEEESTD.2016.7433918.

[2] «IEEE Standard for Ethernet Amendment 4: Physical Layer Specifications and Management Parameters for 1 Gb/s Operation over a Single Twisted-Pair Copper Cable». In: *IEEE Std 802.3bp-2016 (Amendment to IEEE Std 802.3-2015 as amended by IEEE Std 802.3bw-2015, IEEE Std 802.3by-2016, and IEEE Std 802.3bq-2016)* (set. 2016), pp. 1–211. DOI: 10.1109/IEEESTD.2016.7564011.

[3] *AUTOSAR*. URL: https://www.autosar.org/.

[4] Wikipedia contributors. *AUTOSAR — Wikipedia, The Free Encyclopedia*. 2018. URL: https://en.wikipedia.org/w/index.php?title=AUTOSAR&oldid=856923595.

[5] AUTOSAR. *SOME/IP Service Discovery Protocol Specification*. 2017. URL: https://www.autosar.org/fileadmin/user_upload/standards/foundation/1-3/AUTOSAR_PRS_SOMEIPServiceDiscoveryProtocol.pdf.

[6] *GENIVI Alliance*. URL: http://genivi.org/.

[7] Genivi. *vsomeip in 10 minutes*. URL: https://github.com/GENIVI/vsomeip/wiki/vsomeip-in-10-minutes.

[8] *The Autonomous Car Is The Next Entertainment Frontier*. URL: https://www.forbes.com/sites/solrogers/2018/12/12/the-autonomous-car-is-the-next-entertainment-frontier/#4ab2ce1a16a9.

[9] *Standalone Toolchains*. URL: https://developer.android.com/ndk/guides/standalone_toolchain.html.

[10] dec1. *Boost-for-Android*. URL: https://github.com/dec1/Boost-for-Android.

[11] *Service Discovery Definition*. URL: https://avinetworks.com/glossary/service-discovery/.

[12] *Zero Configuration Networking (Zeroconf)*. URL: http://www.zeroconf.org/.

[13] Apple. *Bonjour Overview*. URL: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/NetServices/Introduction.html.

[14] E. Guttman et al. *Service Location Protocol, Version 2*. RFC 2608. RFC Editor, giu. 1999. URL: https://tools.ietf.org/html/rfc2608.

[15] S. Cheshire et al. *Dynamic Configuration of IPv4 Link-Local Addresses*. RFC 3927. RFC Editor, mag. 2005. URL: http://tools.ietf.org/html/rfc3927.

[16] *JSON Schema*. URL: https://json-schema.org/.