POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

RETARGET DI SOFTWARE IN AMBITO AVIONICO: IL CASO DI UN SIMULATORE DI VOLO PER UN ELICOTTERO

Relatore: Prof. Angelo Morzenti
Correlatore: Ing. Matteo Carlo Crippa

Tesi di Laurea Magistrale di:
Davide CASTELLONE, matricola 760304

Anno Accademico 2012-2013

Sommario

Il presente lavoro rappresenta il contributo alla realizzazione di un simulatore di volo a scopo di addestramento per un particolare modello di elicottero. Si tratta di un Full Flight Simulator, che sostituisce e integra legalmente le ore di volo per il conseguimento di una certificazione, licenza o brevetto.

Il nucleo di questo progetto è il cosiddetto "*retarget*" del software del computer di bordo (AMC), ossia il suo adattamento (*porting*) per l'uso all'interno del simulatore. L'apparato in questione incorpora principalmente le funzioni di FMS (Flight Management System) e di monitoraggio dello stato di salute dell'elicottero.

Il *retarget* ha comportato la riscrittura del software di base ("Middleware") e la ricostruzione di tutte le funzionalità di input/output per permettere la comunicazione con l'esterno, ossia col framework di simulazione o in alternativa con lo stesso sistema di test che viene utilizzato per la verifica e validazione dell'AMC vero e proprio. Tra le funzioni principali del Middleware che sono state riprodotte vi è la sincronizzazione e lo scambio di dati tra le due copie dell'AMC presenti sull'elicottero in configurazione ridondata.

Si sono dovute affrontare tutte le problematiche derivanti dal cambiamento di architettura, sia hardware sia software, da un PowerPC con un sistema operativo embedded a un PC con sistema operativo Windows, come fornito dall'assemblatore del simulatore. È stata realizzata una soluzione *ad hoc* per il passaggio ad un'architettura con una rappresentazione dei dati (*endianness*) diversa, eventualità non inizialmente prevista nello sviluppo del software dell'AMC. Il codice sorgente dell'AMC è interamente scritto in Ada, tuttavia le metodologie impiegate si possono applicare, con i dovuti adattamenti, ad altri casi di porting.

Indice

1 Introduzione **3**

1.1 Contesto . . . 3

1.1.1 Simulatori di volo . . . 3

1.1.2 Avionica e software di bordo . . . 7

1.1.3 Portabilità del software . . . 14

1.2 Obiettivi . . . 16

1.3 Verifica . . . 17

1.4 Struttura della tesi . . . 18

2 Stato dell'arte **20**

2.1 Il sistema preesistente: il *Virtual Test Environment* . . . 20

2.1.1 Architettura e caratteristiche . . . 21

2.1.2 Limiti . . . 23

2.2 Trasporto di dati avionici . . . 24

2.2.1 Bus e protocolli di comunicazione . . . 24

2.2.2 Rappresentazione dei dati avionici . . . 29

2.3 Il problema dell'*endianness* . . . 31

2.3.1 Complicazioni dovute all'endianness . . . 33

2.3.2 Il caso dei *bit field* . . . 33

2.3.3 L'endianness in Ada . . . 34

2.4 Tecnologie per simulatori: *High Level Architecture* . . . 36

2.4.1 Definizioni . . . 37

2.4.2 Servizi offerti dall'infrastruttura . . . 38

2.4.3 L'implementazione utilizzata . . . 39

3 Architettura del sistema **40**

3.1 Descrizione strutturale . . . 40

3.2 Descrizione funzionale . . . 43

3.2.1 Funzionalità offerte al Driver . . . 43

3.2.2 Funzionalità del sistema ridondato . . . 45

3.3 Obiettivi e scelte progettuali 48
3.3.1 Riutilizzo e adattamenti del codice del VTE 49
3.3.2 Linguaggi di programmazione 50
3.3.3 Scelta del compilatore 50
3.3.4 Approccio alla correzione dell'endianness 51
3.3.5 Sincronizzazione, comunicazione interprocesso e funzioni speciali 52

4 Dettaglio dei componenti 54
4.1 Middleware 54
4.1.1 Input/Output 55
4.1.2 Scheduler 57
4.1.3 Funzioni speciali 59
4.1.4 Ridondanza e allineamento dati (DBU) 60
4.1.5 Funzioni di servizio 69
4.2 Memoria condivisa 69
4.2.1 `GenericSharedMemory` 71
4.2.2 ARINC 429 71
4.2.3 Discreti e analogici 72
4.2.4 RS 422 72
4.2.5 AFDX 72
4.2.6 Funzioni speciali 74
4.2.7 Interfaccia esterna in C 74
4.3 *Endianizer* 75
4.3.1 Struttura del componente 75
4.3.2 Formato della configurazione 76
4.3.3 Compilazione dei messaggi ed endianizzazione 77
4.3.4 Generazione della configurazione 78
4.4 VTE Proxy 79
4.4.1 `VTE_Proxy` 79
4.4.2 `MessageHandler` 81
4.4.3 `VirtualBuffer` 81
4.4.4 `*SharedMemoryWrapper` 82

5 Implementazione e verifica 83
5.1 Problematiche e cambiamenti richiesti 83
5.1.1 Problemi di endianness non risolvibili dall'esterno . . 83
5.1.2 Collegamento con l'emulatore MCDU 84
5.1.3 Sincronizzazione delle *special functions* 84
5.2 Verifica e validazione 85

5.2.1 Test funzionali 86
5.2.2 Test di integrazione e di sistema 88

6 Conclusioni **89**
6.1 Valutazione a posteriori 89
6.2 Sviluppi futuri: la fase di integrazione 92
6.2.1 Il Gateway HLA 92
6.2.2 Mappatura dei dati 93
6.2.3 Testing e certificazione 93

Bibliografia **95**

Elenco dei simboli

AFDX Avionics Full-Duplex Switched Ethernet

AMC Aircraft Management Computer

AMMC Aircraft Management and Mission Computer

AMMS Aircraft Management and Mission System

AMS Aircraft Management System

BAG Bandwidth Allocation Gap

BCD Binary Coded Decimal

BNR Binario senza segno

CAN Common Automation Network

CVFDR Cockpit voice flight data recorder

DBU Database Update (Allineamento dati)

ENAC Ente Nazionale Aviazione Civile

FAA Federal Aviation Authority

FBW Fly-by-Wire

FCC Flight control computer

FFS Full Flight Simulator

FMC Flight Management Computer

FMS Flight Management System

FNPT Flight and Navigation Procedures Trainer

FOM Federation Object Model

FSTD Flight Simulator Training Device

FTD Flight Training Device

GW Gateway

HLA High Level Architecture

I/O Input/Output

ICD Interface Control Document

MCDU Multiple Control Display Unit

MOM Management Object Model

MQTG Master Qualification Test Guide

MSB Most Significant Byte (Byte più significativo)

MW Middleware

NTF Non-Time-Framed

NVRAM Non Volatile Random Access Memory

QTG Qualification Test Guide

RTI Run-Time Infrastructure

SDI Source/Destination Identifier

SHM Shared Memory (Memoria condivisa)

SOM Simulation Object Model

SSM Sign-Status Matrix

TF Time-Framed

VL Virtual Link

VTE Virtual Test Environment

Capitolo 1

Introduzione

La presente tesi riassume il lavoro svolto in circa nove mesi, a partire dalla seconda metà del 2012, presso la TXT e-solutions S.P.A. Il progetto si colloca all'interno del programma di sviluppo di un simulatore di volo per un elicottero (a scopo di addestramento) e coinvolge l'adattamento di software di bordo, dalla sua piattaforma nativa a Microsoft Windows, e la ricostruzione delle sue interfacce esterne, per il suo utilizzo all'interno del simulatore.

In questo capitolo si darà una breve illustrazione, per punti, del contesto e delle ragioni che hanno portato alla realizzazione di questo progetto, elencandone successivamente gli obiettivi e i limiti.

Alcune scelte tecniche sono state determinate da decisioni prese a monte dal costruttore, committente di questo progetto; di esse si tenterà di dare solo una spiegazione intuitiva.

1.1 Contesto

1.1.1 Simulatori di volo

Un simulatore di volo a scopo di addestramento (in sigla FSTD, dall'inglese *flight simulator training device*) è un dispositivo atto a ricreare artificialmente le condizioni, a terra o in volo, all'interno di un aeroplano (o elicottero, etc.) secondo uno degli standard esistenti. Dalla metà del secolo scorso, i simulatori sono impiegati in ambiti sia civili che militari per l'addestramento degli equipaggi, dati i loro indubbi vantaggi in termini di costo e di riduzione dei rischi. Permettono inoltre di ricreare situazioni improbabili o ad altissimo rischio, cui i piloti devono comunque essere preparati a reagire.

Sotto determinate condizioni, regolate da standard quali ICAO-9625 vol. II, l'addestramento tramite simulatore sostituisce e integra legalmente le ore di volo per il conseguimento di una certificazione, licenza o brevetto. A

questo scopo gli FSTD sono classificati in livelli, a seconda del grado di fedeltà offerto dalla simulazione, con valore legale diverso.

1.1.1.1 La classificazione internazionale

Esistono diversi standard internazionali che stabiliscono dei requisiti di qualità minimi per gli FSTD, suddivisi per livelli. Per gli elicotteri si fa principalmente riferimento a CS-FSTD(H) per l'Europa e FAA Part 60 per gli USA. Per quanto riguarda Italia, l'Ente nazionale per l'aviazione civile (ENAC) con la circolare ENAC OPV-17B ha recepito lo standard JAR-FSTD H, destinato ad essere soppiantato dal suddetto CS-FSTD(H). Pertanto nella descrizione si farà riferimento a quest'ultimo.

I livelli di fedeltà dipendono sia dalla quantità di caratteristiche riprodotte sia dalla loro qualità intesa come misura della verosimiglianza. Con "caratteristica" ci si riferisce ai diversi controlli e stimoli che possono essere riprodotti, ad esempio la resistenza meccanica dei comandi alle azioni del pilota (caratteristica che può essere presente o assente), la simulazione dei movimenti del velivolo, le condizioni atmosferiche, i suoni, la visuale esterna e così via.

Lo standard divide gli FSTD in tre categorie. In ordine decrescente:

FFS *Full Flight Simulator.* Una replica a dimensione reale della cabina di uno specifico modello, compresa *tutta* la strumentazione e il software necessario per rappresentare l'elicottero a terra o in volo, un *visual system* che proietta la visuale esterna (OTW) su un display collimato[1]e un *motion system* in grado di riprodurre in modo sufficientemente realistico i movimenti della cabina. Gli FFS sono classificati in quattro livelli, da A a D, dove D rappresenta il livello di verosimiglianza più alto.

FTD *Flight Training Device.* È una replica in scala 1:1 della cabina di uno specifico modello, ma a differenza di un FFS non sono richiesti un *motion system* e un *visual system* e la riproduzione della cabina può essere aperta. Un FTD deve essere in grado di riprodurre le condizioni a terra o in volo limitatamente ai dispositivi installati sul simulatore. Gli FTD sono divisi in tre livelli, da 1 a 3.

FNPT *Flight and Navigation Procedures Trainer.* Non è obbligato a rappresentare un modello specifico ma può riprodurre semplicemente una

[1]La luce uscente da un display collimato è formata da raggi il più possibile paralleli, così da ottenere la stessa messa a fuoco degli oggetti molto distanti dall'osservatore e non creare errori di parallasse tra postazioni diverse. [CS-FSTD(H), p. 109]

classe di aeromobili. Riproduce solo una parte della strumentazione. I FNPT si dividono in tre livelli, da I a III, più un quarto livello MCC per le esercitazioni di *Multiple Crew Co-operation*.

Il manuale ICAO-9625 vol. II, in modo ancora più dettagliato, assegna a ogni caratteristica un livello di fedeltà. Ne definisce quattro, qui elencati con una loro descrizione sommaria (l'interpretazione particolare di questi livelli è ridefinita per ogni caratteristica).

None (N) caratteristica non richiesta.

Generic (G) simula una caratteristica generica degli aeromobili di una determinata tipologia, ma non di una classe particolare.

Representative (R) simula una caratteristica di una particolare classe di aeromobili (ad esempio, aeroplano con quattro motori a turbina).

Specific (S) il livello di fedeltà più alto: rappresenta fedelmente uno specifico modello di aeromobile.

La certificazione di un simulatore di volo comporta una serie di test, con lo scopo di verificare che le caratteristiche dell'FSTD non si discostano dall'elicottero reale oltre un certo margine, in base al livello di qualificazione desiderato. I risultati sono raccolti in un documento chiamato QTG (*Qualification Test Guide*), che una volta approvato dall'autorità competente cambia nome in MQTG (*Master Qualification Test Guide*) e sarà utilizzato come riferimento nelle successive valutazioni periodiche cui è sottoposto l'FSTD per mantenere la qualifica.

1.1.1.2 Il caso analizzato

Nel caso di questo progetto sono stati commissionati simulatori di diverso livello, sempre per lo stesso modello di elicottero, fino ad arrivare a un FFS livello D, il livello massimo. Il committente ha deciso di reimpiegare lo stesso software di bordo, per quanto riguarda autopilota e l'AMC (il cosiddetto "computer di bordo"), probabilmente per motivi di realismo, di testabilità e anche per motivi pratici – appare più semplice reimpiegare il software originale, anche riducendolo, piuttosto che riscriverlo con requisiti modificati. Questa considerazione non è del tutto ovvia, sapendo che per altri FSTD sviluppati dalla stessa compagnia è stata fatta una scelta diversa; tuttavia non si trattava di dispositivi di complessità paragonabile a quella del software che sarà descritto più avanti. È stata subito esclusa, invece,

la possibilità di inserire il vero computer di bordo all'interno del simulatore dati i costi proibitivi dell'hardware.

Il simulatore in questione è un sistema distribuito, suddiviso in "modelli" che riproducono ciascuno un determinato componente dell'elicottero. La comunicazione e la sincronizzazione sono gestite attraverso un'implementazione propria dello standard IEEE 1516, un framework per la simulazione in tempo reale basato su un protocollo a oggetti con uno schema *publish-subscribe* e delle interfacce standard (da qui in poi semplicemente "Framework"). Il tempo è discreto e il passo (l'intervallo virtuale tra un istante e il successivo) è uguale per tutti i modelli ed è sincronizzato da un timer centrale.

1.1.1.3 L'uso di software avionico nei simulatori

L'utilizzo di software di bordo all'interno di un simulatore è una pratica comune, tanto che sono state scritte delle linee guida quali ARINC-610B che descrivono i requisiti comuni con indicazioni su come implementarli e i problemi più frequenti, con indicazioni su come evitarli.

In ARINC-610B sono descritte delle funzionalità aggiuntive che si applicano solo al software per il simulatore, tra cui:

- *failure injection*, la simulazione di guasti a comando,
- diversi tipi di *freeze*, ossia il blocco del valore di alcuni parametri (ad esempio la posizione o la velocità; il documento ne definisce un certo numero) lasciando che il tempo continui a scorrere normalmente; in questo caso non devono essere prodotti avvertimenti o valori errati
- il *riposizionamento* o comunque la variazione artificiale di qualche parametro (tipicamente si cambia la posizione, tra una sessione e l'altra), che come nel punto precedente non deve avere effetti collaterali, per esempio produrre errori nel calcolo dei consumi
- *snapshot* e *recall*, analoghi delle operazioni "salva" e "carica" di un videogame, utilizzati quando l'addestramento viene interrotto per essere ripreso in un momento successivo oppure per tornare a uno stato precedente dopo uno schianto
- la funzione di "avanti veloce", che può rappresentare un problema per la stabilità dei controlli (barre, pedali).

Nel simulatore è previsto un sottoinsieme di queste funzioni, che possono essere già previste nel design del software di bordo oppure – come nel nostro caso – essere aggiunte successivamente tramite modifiche *ad hoc*.

1.1.2 Avionica e software di bordo

"Avionica" – dalla contrazione delle parole "aviazione" ed "elettronica" – è il termine col quale si indicano le apparecchiature installate a bordo di un aeromobile che dipendono dall'elettronica per il loro funzionamento [Collinson, 2011]. Il termine cominciò ad essere utilizzato negli Stati Uniti dagli anni '50, fino a raggiungere una diffusione su vasta scala ed entrare nel linguaggio comune.

L'avionica è ormai fondamentale nell'aviazione moderna, sia civile sia militare. Generalmente compone circa il 30% del costo di un velivolo. Essa comprende una grande varietà di apparecchiature, a partire da giroscopi e accelerometri lineari per misurare l'orientamento e il movimento nello spazio, sensori dell'aria che misurano la velocità all'aria, l'altitudine, la velocità verticale, la pressione, la temperatura; servo-attuatori idraulici comandati dal pilota o dall'autopilota per regolare l'assetto e di conseguenza la direzione e la velocità, per citarne alcuni. I velivoli caratterizzati da sistemi *Fly-by-Wire* (FBW) dipendono totalmente dall'elettronica, in quanto i comandi del pilota sono comunque processati da un calcolatore – il *Flight control computer* (FCC) – e non esiste una via diretta per comandare gli attuatori nel caso si guasti.

Oltre ai dispositivi legati direttamente al controllo del volo vi sono i sistemi di interfaccia utente (display di vario tipo, spesso grafici e a colori), sistemi di comunicazione (radio, transponder) e di navigazione (vedi FMS). Questi ultimi comprendono sistemi per l'atterraggio strumentale che servono a calcolare la distanza e la direzione della pista, e altri che permettono di calcolare la posizione del velivolo nello spazio. Vi sono infine sistemi di monitoraggio delle prestazioni e dello stato di salute del velivolo e di tutti gli apparati che lo compongono (*health/performance monitoring*) e naturalmente i sistemi di comunicazione interna che permettono lo scambio di dati tra gli apparati elettronici.

1.1.2.1 Calcolatori di bordo

I calcolatori di bordo generalmente si occupano dell'automatizzazione di alcuni compiti che sarebbero altrimenti svolti dall'equipaggio. Ciò permette di ridurre le dimensioni dell'equipaggio o di svolgere attività più complesse con meno distrazioni. Le funzionalità offerte dai calcolatori a bordo di un velivolo civile sono molteplici e possono essere suddivise in macro-funzioni. A seconda di come sono raggruppate, tali funzioni possono essere assegnate a un numero diverso di calcolatori, pertanto le loro sigle variano tra velivoli di modelli diversi anche dello stesso costruttore.

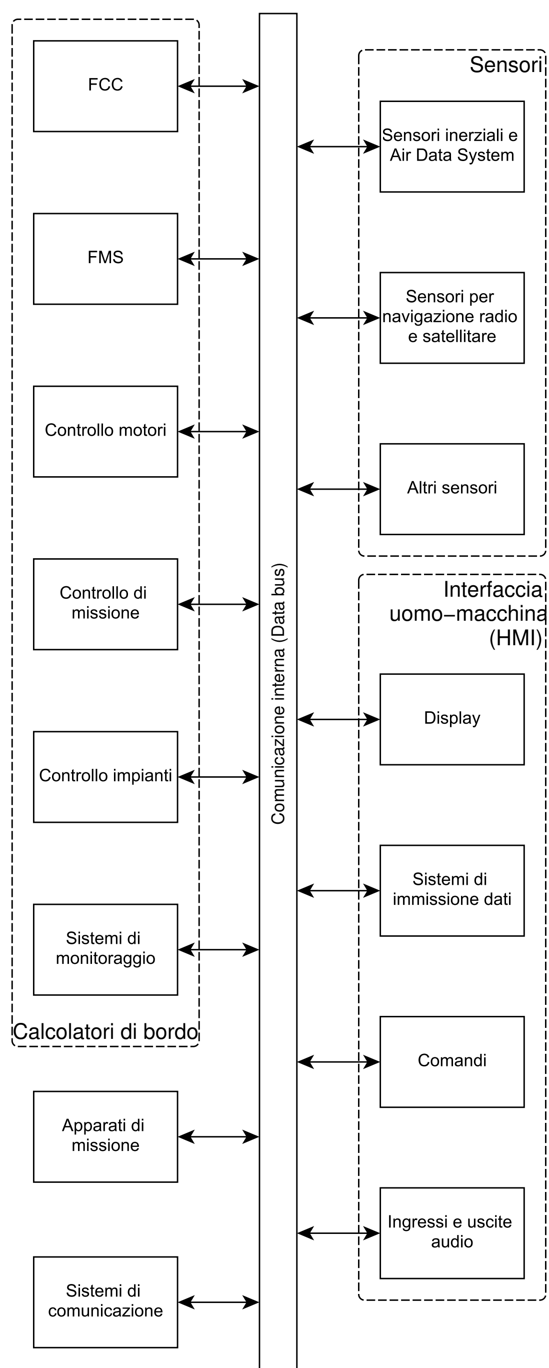


Figura 1.1: Schema semplificato dei componenti di un generico velivolo. Per semplicità i bus – che normalmente sono molteplici e di diversi tipi – sono rappresentati come un singolo elemento.

Controllo Come citato sopra, il sistema di controllo è affidato a un calcolatore detto FCC, con il compito di mantenere la stabilità e di agire come autopilota, su indicazione del sistema di navigazione. Esso riceve da diversi sensori i dati sull'assetto e sullo stato del motore e invia i comandi direttamente agli attuatori che in un aeroplano muovono le superfici di controllo, mentre in un elicottero regolano l'inclinazione delle pale.

Navigazione Con "navigazione" si intende l'insieme di conoscenze e tecniche atte a stabilire la rotta e a definire la propria posizione nello spazio. In un velivolo questo processo è impersonato dal *Flight Management System* (FMS), mentre il calcolatore preso singolarmente è detto *Flight Management Computer* (FMC).

La funzione principale dell'FMS è il calcolo continuo della direzione da seguire, comunicata al pilota (attraverso i display) o all'autopilota (attraverso il bus) sotto forma di pochi dati sintetici, il più importante dei quali è l'angolo di rollio (*bank angle*), che determina il raggio della virata. Il calcolo combina i dati provenienti da diversi sensori e considera la velocità, la massa, il baricentro e le caratteristiche del velivolo, insieme alle norme che regolano il traffico aereo e segue il piano di volo inserito precedentemente dal pilota con l'assistenza dell'FMS, che a questo scopo si serve di un database contenente informazioni su aeroporti, procedure, radioassistenze e punti di passaggio. Il piano di volo (*flight plan*) contiene la sequenza delle procedure da seguire, dal decollo all'atterraggio, e le azioni da intraprendere in caso di mancato avvicinamento; da esso sono calcolati i consumi e l'orario di arrivo previsti.

Oltre alla direzione il sistema dà indicazioni sulla velocità, allo scopo di raggiungere la destinazione all'orario previsto ottimizzando i consumi e consente di riorganizzare il piano di volo in caso di imprevisti. Grazie allo stesso database, l'FMS è inoltre in grado di sintonizzare automaticamente i sensori per la radionavigazione sugli emettitori previsti dall'itinerario. Detti "radioassistenze" (*navigation aids*) essi sono dei trasmettitori radio, collocati in corrispondenza degli aeroporti e sul territorio, che a seconda del tipo sono utilizzati per il calcolo della posizione, dell'orientamento o della direzione da seguire.

Una funzione critica è l'atterraggio strumentale, che permette di atterrare in sicurezza anche in condizioni di scarsa visibilità. Esso sfrutta dei trasmettitori radio collocati in prossimità della pista o in alternativa sistemi satellitari per guidare il velivolo lungo l'avvicinamento, dando le indicazioni necessarie al pilota o all'autopilota.

L'utilizzo di un FMS rende più semplice la gestione del traffico aereo, riducendo il margine di errore sulla traiettoria. Questo margine viene continuamente aggiornato dal calcolatore per valutare lo scostamento della traiettoria rispetto alla rotta. Il margine dipende dalla procedura in corso ed è più stretto nelle fasi di decollo e atterraggio.

Monitoraggio e acquisizione dati Il sistema di monitoraggio si occupa del controllo costante dello stato di tutti i componenti del velivolo, con l'emissione degli eventuali avvisi visivi o sonori nei casi più gravi, e della registrazione dei guasti o dei segnali di usura in un apposito registro, con lo scopo di semplificare le procedure di manutenzione. È inoltre in grado di avviare test automatici o su richiesta. Lo stato di salute del motore viene valutato attraverso la misurazione delle vibrazioni e delle dimensioni dei frammenti che si staccano per l'attrito dal sistema di trasmissione (*wear debris*).

Queste funzioni, eventualmente unite a quelle dell'FMS, appartengono ad un sottosistema comunemente denominato *Aircraft Management System* (AMS) – le sigle possono variare –, composto da uno o più calcolatori (*Aircraft Management Computer*, AMC) e dall'eventuale sistema che li collega e sincronizza. Comandano le loro interfacce utente (display e sistemi di immissione – tastiere, trackball, comandi vocali) possibilmente in comune con altri sottosistemi. Se non è affidata ad un sottosistema diverso, l'AMS si occupa dell'acquisizione dei dati dai sensori, l'integrazione dei dati che provengono da fonti diverse e la loro diffusione attraverso il sistema di comunicazione interna.

La parte di monitoraggio comprende anche il CVFDR o FDR (*Cockpit voice flight data recorder*, o semplicemente *Flight data recorder*), la cosiddetta "scatola nera".

Missione L'avionica di missione è quella che caratterizza la specializzazione del velivolo. A seconda della specializzazione (ricerca e soccorso, antisommergibile, eccetera) possono essere montati diversi apparati aggiuntivi, tra cui armi, sistemi di puntamento (preferibilmente integrati con i display), contromisure, dispositivi di comunicazione militare crittografata, sensori tattici (ad esempio radar o telecamere FLIR – *Forward Looking Infrared*).

Quando l'avionica di missione è integrata con l'AMS si parla di *Aircraft Management and Mission System* (AMMS) e il corrispondente calcolatore è denominato AMMC. Anche in questo caso le sigle possono variare a seconda del produttore.

Comunicazione interna Sebbene la parte di comunicazione interna (i *bus*) sia comune a tutte le funzionalità, a seconda della tecnologia utilizzata, in alcuni casi diventa un vero e proprio sottosistema con i suoi dispositivi di controllo. La comunicazione interna impiega tecnologie differenti a seconda della destinazione civile o militare del velivolo. In ogni caso, all'interno dello stesso aeromobile solitamente convivono tecnologie diverse, in modo da riutilizzare componenti già disponibili sul mercato. Le più comuni sono lo standard ARINC 429, per uso civile e il MIL-STD 1553 per uso militare. Il primo consiste in un collegamento punto-multipunto monodirezionale – richiede un cavo per ogni direzione della comunicazione – il secondo è un bus seriale half-duplex controllato da un *bus controller* che funge da arbitro tra gli altri *remote terminal*. In tempi più recenti per ovviare ai limiti dell'ARINC 429 è stato sviluppato dalla Airbus il protocollo AFDX (ARINC 664), *Avionics Full-Duplex Switched Ethernet*, basato sull'Ethernet con alcune variazioni che lo rendono adatto all'uso avionico, ottenendo un comportamento deterministico.

Oltre a bus strettamente avionici, in un velivolo si possono trovare anche collegamenti analogici (portano un solo dato, proporzionale alla tensione o alla corrente), discreti (interruttori), CAN bus (*Common Automation Network*) già diffusi nel mondo *automotive*, seriali EIA RS-422 e altri.

1.1.2.2 Affidabilità dei sistemi *safety-critical*

Requisiti di sicurezza Lo sviluppo di hardware e software avionico è molto più complesso e costoso dello sviluppo per altri settori, a causa dei requisiti di sicurezza molto stringenti. Il processo di sviluppo di software avionico è descritto dallo standard DO-178B o ED-12C, sigle con cui è pubblicato rispettivamente negli Stati Uniti dalla FAA e in Europa dalla EUROCAE. Esso divide il software, a seconda del rischio, in quattro livelli, da A a D, dove A è il massimo livello di rischio (*safety-critical*) tale per cui un guasto in un sistema di tale livello provoca un rischio immediato per la vita dell'equipaggio e la perdita del velivolo. Il livello B è relativo ai rischi che possono causare ferimenti o ridurre la capacità di lavoro dell'equipaggio o vittime ma solo tra i passeggeri. I livelli C e D si riferiscono a rischi minori. Lo sforzo per la certificazione del software dipende dal livello di rischio.

Sottosistemi diversi all'interno dello stesso velivolo possono avere livelli diversi e anche all'interno dello stesso calcolatore, con l'utilizzo di tecnologie certificate che isolano completamente l'esecuzione dei diversi componenti (ARINC 653).

Ridondanza Per ottenere la tolleranza ai guasti necessaria, gli apparati critici sono sempre presenti in configurazione “ridondata”, cioè sono presenti in copie multiple che vengono adoperate contemporaneamente o alternativamente, in modo che al guastarsi di una copia le altre possano continuare a svolgere le stesse funzioni. Il numero di copie dipende dalla disponibilità necessaria, intesa come probabilità che in un determinato momento il sistema nel suo complesso funzioni (*availability*). Nel caso della “riserva a caldo”, le due o più copie operano sincronizzate e ricevendo gli stessi ingressi producono le stesse uscite. In caso di spegnimento di una delle copie le altre continuano a funzionare in modo trasparente per il resto degli apparati. Più realisticamente, per motivi vari (ad esempio il controllo del bus o di dispositivi di memoria di massa) esiste sempre una copia cui è affidato il ruolo di *master*, mentre le altre sono dette *standby* (“riserve”). In caso di caduta del master, il suo ruolo viene preso in carico da una delle riserve, secondo un ordine di priorità. In alcuni casi il master può essere scelto dal pilota. Per mantenere sincronizzati i dati in uscita anche in caso di discrepanza tra gli ingressi è necessario un sistema di comunicazione interna tra le copie (il cosiddetto “cross-talk”).

1.1.2.3 Il caso analizzato: l’AMC

Il software oggetto di questa tesi è l’AMC di un elicottero, che incorpora le funzionalità di FMS e monitoraggio (spiegate al paragrafo 1.1.2.1 a pagina 9). È presente in una configurazione duale ridondata, il cui comportamento dovrà essere riprodotto nel simulatore. Non è considerato un componente *flight-critical*, pertanto è classificato come software di livello B.

Funzionalità Nel dettaglio, l’AMC ha le seguenti funzionalità:

- il monitoraggio e la registrazione dello stato di salute e usura del velivolo
    - controllo delle vibrazioni
    - usura del sistema di trasmissione (wear debris), del motore e della cellula
- avviare test periodici o su richiesta
- la generazione di allarmi visivi e sonori
- la gestione dell’orologio di sistema

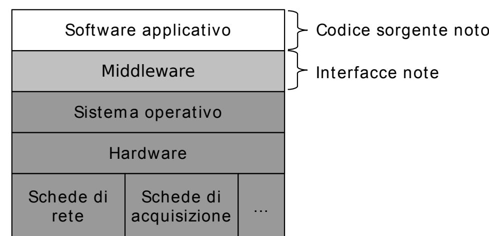


Figura 1.2: Architettura dell'AMC

- la gestione dei supporti di memorizzazione di massa
- l'integrazione dei dati provenienti dai vari sensori
- il monitoraggio della quantità di carburante
- la gestione del volo (FMS)
- il controllo delle radio
- l'interfaccia utente
  - il controllo di alcune parti dei display PFD (*primary flight display*) e MFD (*multifunction flight display*)
  - l'input di dati attraverso trackball, tastiere, touch-screen.

Architettura hardware/software L'AMC è un calcolatore assemblato appositamente per l'uso avionico. Monta un processore PowerPC e un sistema operativo embedded real-time per applicazioni *safety-critical*, fornito da terze parti. Sopra il sistema operativo vi è uno strato software che chiameremo "Middleware", fornito dal produttore dell'hardware, che lo specializza rispetto alle applicazioni in ambito avionico. Esso permette di accedere a tutta una serie di funzionalità per il controllo delle interfacce di input/output, lo scheduling – come sarà spiegato successivamente –, la sincronizzazione tra le due copie dell'AMC e utilità varie, astraendo le funzioni del sistema operativo.

Infine vi è lo strato applicativo, che specializza l'AMC per questo particolare modello di elicottero, scritto dallo stesso costruttore. Per lo svolgimento del progetto, ci è stato fornito il codice sorgente di questa parte, insieme alle specifiche delle interfacce del Middleware. Per dare un'idea della sua complessità, il volume di questo codice sta nell'ordine di grandezza del milione di righe. Il linguaggio di programmazione utilizzato dallo strato applicativo e dalle interfacce del Middleware è esclusivamente l'Ada 95. Questo dettaglio,

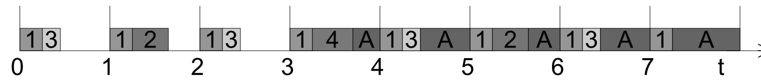


Figura 1.3: Esempio di scheduling. In figura è rappresentata l'occupazione della CPU in un periodo lungo 8 minor frame*. I task da 1 a 4, in ordine decrescente di priorità, hanno rispettivamente periodo 1, 4, 2 e 8. Dal frame 3 si aggiunge il task statistico A.*

insieme all'architettura del processore, saranno fattori determinanti per le successive decisioni tecniche.

Scheduling Tra le particolarità di questo ambiente vi è da segnalare lo scheduling. Dato che l'AMC riceve input che sono aggiornati periodicamente dai sensori, con una frequenza che dipende sia dalle tempistiche del bus sia dal particolare dato, la temporizzazione del software è organizzata di conseguenza ed è pertanto ciclica, con un periodo sincronizzato con quello degli input. Il software è suddiviso in task che sono eseguiti ciclicamente secondo una tabella fissata a priori. Per l'AMC il periodo minimo è fissato in 20 ms, detto *minor frame*, che corrisponde ad una frequenza di 50 Hz; 32 o 64 *minor frame* compongono un *major frame* da 640 o 1280 ms (a seconda della specializzazione dell'AMC), che costituisce il periodo massimo di un task.

I task possono essere di tre tipi: deterministici, deterministici su evento, statistici. La tabella dei task specifica quali deterministici vengono avviati in ogni minor frame. Ogni task deterministico deve terminare la sua esecuzione all'interno del minor frame in cui è stato avviato, o almeno prima della sua successiva esecuzione. All'interno del minor frame i task competono per l'utilizzo della CPU secondo la priorità che è assegnata loro nella tabella. I task statistici invece impegnano il rimanente tempo-macchina, avendo priorità più bassa ma nessun vincolo di tempo e si occupano delle operazioni lunghe come ad esempio la lettura/scrittura della memoria di massa.

1.1.3 Portabilità del software

Con "portabilità del software" (in Inglese *porting*; sono simili *retarget* e *rehost*[2]) si intende la ricompilazione – eventualmente con modifiche – dello stesso codice sorgente in ambienti diversi tra loro, quali possono essere siste-

[2] La differenza tra *retarget* e *rehost*, secondo la definizione data nel documento ARINC-610B, sta nel fatto che nel *rehost* è l'eseguibile, già compilato, ad essere rieseguito in un ambiente simulato, mentre il *retarget* comporta anche una ricompilazione del codice sorgente. Nel nostro caso dunque si tratta di un *retarget*.

mi operativi differenti oppure sistemi con CPU di architetture diverse. Nel nostro caso valgono entrambe le condizioni.

Le difficoltà possono essere date dalla differenza tra le interfacce software dei diversi sistemi operativi o dalla diversa rappresentazione dei dati tra architetture differenti, oltre alle piccole differenze tra le implementazioni dei diversi compilatori.

La portabilità è un problema ben noto in letteratura, tanto che i principali linguaggi di programmazione dispongono di funzioni per rendere il codice indipendente dall'architettura e dal sistema operativo, almeno entro un certo livello. L'Ada 95 è tra questi, con la notevole eccezione di alcuni costrutti che lo rendono dipendente dall'architettura hardware (sebbene rimanga ben disaccoppiato dal sistema operativo) causando incompatibilità.

Un codice sorgente (o, per estensione, un linguaggio di programmazione) è tanto più "portabile" quanto minori sono le modifiche necessarie per adattarlo al nuovo sistema. Ciò ovviamente dipende anche dall'entità della differenza tra il sistema di partenza e quello di destinazione.

1.1.3.1 Limitazioni

La ricompilazione del codice sorgente per un ambiente diverso, anche qualora abbia successo, non garantisce la riproduzione esatta del comportamento originale.

Cambiando l'hardware, cambiano le caratteristiche di carico e di conseguenza i tempi di esecuzione. Se la macchina di partenza è molto meno potente e la situazione è quella dell'AMC descritto sopra ad esempio, i tempi di esecuzione dei task statistici, che non sono vincolati ad un timer come quelli dei task deterministici, saranno molto ridotti rispetto all'originale. Questo per il simulatore non rappresenta una complicazione, ma se si volesse (come si è fatto) usare il software ricompilato come modo più semplice ed economico per testare il sistema vero, il problema appena descritto rischia di portare a risultati più ottimistici – tempi di esecuzione più bassi, dato che i task statistici terminano prima – e di conseguenza creare delle false aspettative. Lo stesso vale per l'occupazione di memoria.

Per quanto riguarda il sistema operativo, passando ad esempio da un SO real-time a uno come Windows, si perdono quelle garanzie di precisione nella temporizzazione, di rispetto delle priorità che un sistema operativo real-time dà, proprio perché si tratta di sistemi operativi con scopi (e costi) diversi. Occorre quindi definire dei criteri di verosimiglianza entro cui l'approssimazione del comportamento è accettabile, secondo gli obiettivi del progetto.

1.1.3.2 Problematiche correlate

Anche nei casi più favorevoli, ricompilare un software per l'esecuzione in un sistema operativo e un'architettura diversi non è mai un'operazione indolore. Nel nostro caso si è dovuto affrontare anche il problema di cambiare il produttore del compilatore, che pure per lo stesso linguaggio di programmazione presenta delle piccole differenze che hanno causato errori di compilazione all'inizio e di esecuzione successivamente. Inoltre alcune caratteristiche di un compilatore possono non essere implementate da un altro o essere realizzate in modo diverso, tali da rendere necessarie delle modifiche.

Vi è inoltre l'importante problema dell'*endianness*, che si manifesta tra architetture che hanno una differente rappresentazione in memoria dei dati. Nel nostro caso, passando da un'architettura *big-endian* ad un'architettura *little-endian*, i dati che occupano più di un byte sono rappresentati invertiti[3] e ciò ha causato incompatibilità nei casi in cui, durante lo sviluppo del software, si era data per scontata una particolare rappresentazione oppure che la rappresentazione interna dei dati fosse la stessa utilizzata nel trasporto dei dati sulla rete, non prevedendo così le necessarie operazioni di conversione. Questa problematica è meglio affrontata nella sezione 2.3.

1.2 Obiettivi

Dato lo scenario descritto nella sezione precedente, è possibile qui ricapitolare gli obiettivi del progetto trattato nella presente tesi.

Obiettivo principale è il *retarget* dell'AMC di un elicottero, per il suo utilizzo all'interno di un simulatore. L'applicativo è stato concepito per essere eseguito in un sistema operativo real-time proprietario con un processore della famiglia PowerPC (big-endian). Della parte applicativa abbiamo il codice sorgente scritto in Ada 95, insieme alle interfacce dei componenti del Middleware, che dovrà essere reimplementato per il sistema di destinazione, una versione a 32 bit di Microsoft Windows. Vi sono inoltre i seguenti vincoli.

- Per motivi di verosimiglianza, il codice applicativo deve essere modificato il meno possibile. È possibile invece agire sullo strato intermedio (Middleware).

[3]Per quanto riguarda gli array, i dati sono i singoli elementi. Pertanto le stringhe non hanno subito alcuna mutazione, in quanto array di byte.

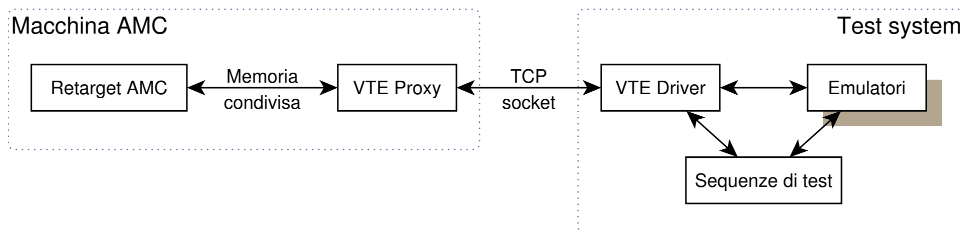


Figura 1.4: Struttura del test system. VTE sta per "Virtual Test Environment". Ogni blocco rappresenta un processo.

- Devono essere riprodotte (virtualmente) le infrastrutture di Input/Output e devono essere fornite delle interfacce (API) per rendere possibile la lettura/scrittura da parte del framework di simulazione.
- Devono essere aggiunte le funzionalità specifiche del simulatore come descritto al paragrafo 1.1.1.3.
- Deve essere garantita un'approssimazione accettabile – secondo criteri che saranno definiti successivamente – dei tempi di risposta e dello scheduling dell'AMC originale.
- La verosimiglianza del prodotto rispetto all'AMC originale deve essere testabile.

Il Framework di simulazione, l'implementazione dello standard IEEE 1516 (HLA) e l'integrazione dell'AMC all'interno del Framework sono un progetto a parte e non saranno trattati qui. Appartiene invece allo stesso progetto il componente di comunicazione e traduzione dei dati ingegneristici dal Framework in messaggi di basso livello per l'AMC; tuttavia tale componente è stato sviluppato come sottoprogetto separato e non sarà oggetto di questa tesi.

1.3 Verifica

Il prodotto *retarget* si considera verificato nella misura in cui è verificata la aderenza al sistema originale (il requisito principale del progetto); il prodotto deve quindi rispettare quasi tutti i requisiti del sistema originale. A tal fine si è ricorso alla stessa infrastruttura di test utilizzata per la verifica del software avionico, già predisposta per i test di apparati virtuali; il sistema è detto *Virtual Test Environment* (VTE). La verifica è compiuta attraverso un insieme di sequenze di test, che riproducono gli stimoli all'AMC e controllano che le risposte corrispondano a quelle previste secondo i requisiti, considerando i tempi di risposta.

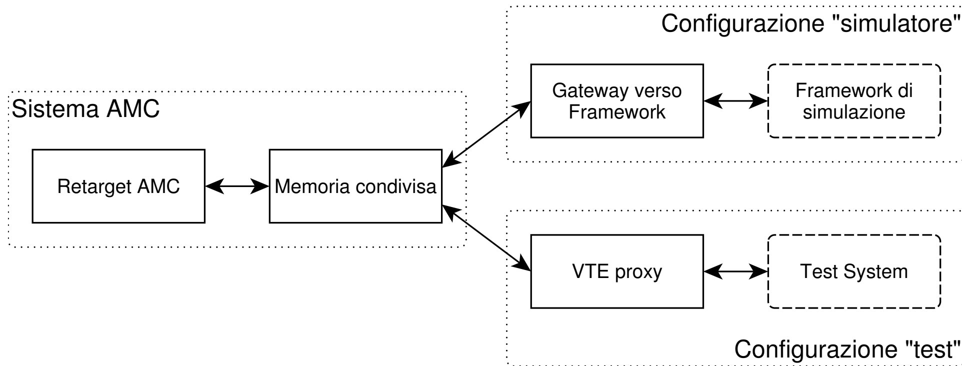


Figura 1.5: Rappresentazione delle due configurazioni possibili, "simulatore" e "test". Esse sono in mutua esclusione ma il sistema AMC, fino alla memoria condivisa, è lo stesso.

La simulazione del tempo all'interno del test system è totalmente virtuale, ossia il timer dello scheduler è guidato dall'esterno, in modo che sia sempre mantenuta la sincronizzazione tra l'apparato simulato e il sistema di test, soprattutto quando la velocità di quest'ultimo è inferiore.

Per permettere al *retarget* dell'AMC di comunicare col sistema di test – che utilizza un protocollo particolare basato sui messaggi di basso livello, diverso da quello utilizzato nel Framework che trasmette direttamente i dati ingegneristici già decodificati – è stato sviluppato un componente apposito, che si aggancia al Middleware attraverso le sue interfacce di input/output e sostituisce lo strato di comunicazione col Framework.

La verifica così effettuata è in grado di dimostrare che il comportamento "ai morsetti" del retarget è analogo a quello del sistema originale, ma non è in grado di testare l'integrazione tra il retarget e il simulatore, che dovrà essere effettuata in una fase successiva. Per quanto riguarda le funzioni specifiche del simulatore che non esistono nell'apparato originale devono essere prodotte delle sequenze di test apposite.

1.4 Struttura della tesi

Nel successivo capitolo 2 vengono descritte le soluzioni che l'industria e l'accademia offrono per i nostri scopi, insieme allo stato delle cose su cui il presente progetto si è andato a innestare.

Nel capitolo 3 si descrive il design di alto livello del prodotto e le scelte tecniche principali, motivandole.

Il capitolo 5 elenca le problematiche sorte nella fase di implementazione, le soluzioni adottate; sono inoltre descritti modalità e risultati dei test.

Infine, il capitolo 6 rivede a posteriori le scelte fatte all'inizio e le valuta da un punto di vista tecnico. Viene inoltre descritta in modo sommario la successiva fase di integrazione e i problemi che ha comportato.

Per tutti gli aspetti sui quali era presente un vincolo di riservatezza si è proceduto semplificando ma cercando di non togliere comprensibilità alla problematica generale.

Capitolo 2

Stato dell'arte

Questo capitolo tratterà della situazione preesistente al progetto, ossia della parte già costruita in precedenza, sebbene con scopi diversi, da cui il progetto si è sviluppato estendendola o discostandovisi. Seguirà una breve descrizione del problema dell'*endianness* – uno dei maggiori problemi affrontati – con le sue soluzioni, e dello standard cui si riferisce il Framework di simulazione, destinazione del progetto.

2.1 Il sistema preesistente: il *Virtual Test Environment*

Prima dell'inizio dello sviluppo del retarget dell'AMC per il simulatore, esisteva già un retarget – sebbene parziale – a scopo di test, su cui mi sono inizialmente basato. Il sistema è detto *Virtual Test Environment* (VTE) e consiste nella versione dell'AMC ricompilata per Sun Solaris (per un processore SPARC), scelto perché ha la stessa *endianness* del sistema originale – quindi la stessa rappresentazione dei dati – tra le destinazioni possibili dello stesso compilatore. Il VTE simula il comportamento dell'AMC, compresi ingressi e uscite; le sequenze di test simulano gli ingressi e controllano che le uscite corrispondano entro un margine d'errore a quelle previste secondo i requisiti.

La ragione principale della creazione del VTE è economica: rendere più efficiente la fase di sviluppo e svolgere dei test più frequenti e in modo più veloce e parallelizzabile. Permette di risparmiare preziose ore di utilizzo degli AMC (così come di altri apparati) veri, che hanno un costo dell'ordine delle centinaia di migliaia di Euro e sono pertanto disponibili in numero limitato. Al contrario il VTE gira su macchine che hanno il costo di normali server e possono gestire più utenti contemporaneamente.

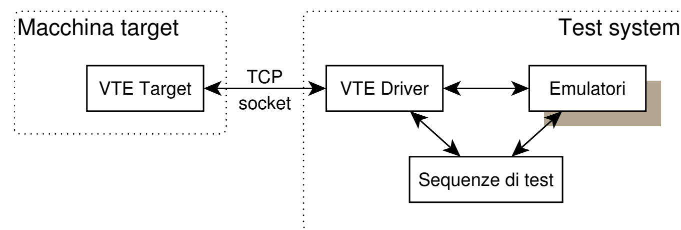


Figura 2.1: Componenti del test system e del VTE

2.1.1 Architettura e caratteristiche

Il sistema VTE si compone di due parti principali.

Target Il software dell'AMC ricompilato, insieme al Middleware (già definito nel § 1.1.2.3 a pagina 13).

Driver Un processo che viene eseguito sulla stessa macchina del test system e che gestisce gli ingressi e le uscite del target, comunicando dall'altra parte con le sequenze di test e gli emulatori.

Del test system fanno parte anche l'esecutore delle sequenze di test e un insieme di emulatori che riproducono il comportamento di alcuni apparati collegati all'AMC quali radio, sensori, display ecc. Gli emulatori e il software che esegue le sequenze sono gli stessi utilizzati per testare l'AMC vero, collegato al server tramite una scheda appositamente progettata, che alloggia tutte le necessarie interfacce di rete.

Il medesimo test system è stato impiegato per la verifica del retarget, per il quale abbiamo reimplementato anche il protocollo di comunicazione col Driver.

2.1.1.1 Implementazione del Middleware

La versione del Middleware inclusa nel target del VTE è una reimplementazione parziale delle interfacce Ada del Middleware originale ed è scritta in parte nello stesso linguaggio e in parte in C (per motivi storici). Questa parte di software è in comune tra diversi progetti che utilizzano hardware simile ed è il risultato della stratificazione di una ventina di anni di sviluppo, attraversando epoche diverse con linguaggi di programmazione diversi, da cui la divisione nelle due parti. La parte in C si preoccupa delle funzioni di più basso livello, tra cui la comunicazione col driver, ed è più stabile nel tempo rispetto alla parte in Ada.

Il codice Ada del target – Middleware e codice applicativo – è compilato con stesso compilatore utilizzato per l'AMC reale – ciò assicura un com-

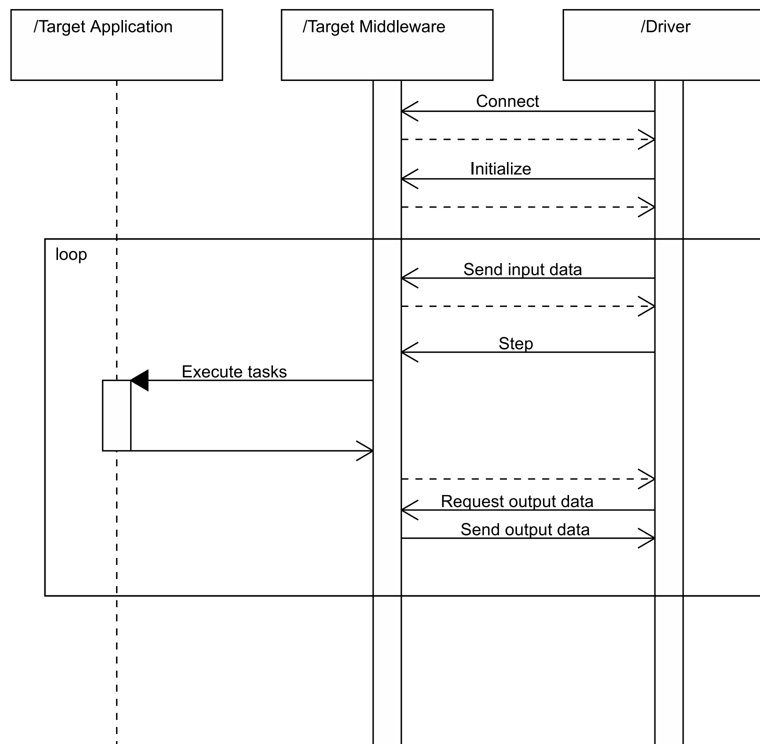


Figura 2.2: Diagramma di sequenza di una tipica sessione del VTE

portamento e un linguaggio più consistenti tra le due versioni ma non sarà possibile nella versione per Windows.

2.1.1.2 Protocollo di comunicazione

Il target e il driver sono collegati attraverso un socket TCP con un protocollo scritto ad hoc. Il target è l'attore passivo in quanto tutte le azioni sono iniziate dal driver, cui il target trasmette le eventuali risposte. Il driver si preoccupa anche della sincronizzazione dell'esecuzione, inviando periodicamente al target il comando di *step*, che causa l'esecuzione di un singolo frame (definito nel § 1.1.2.3 a pagina 14). La simulazione degli ingressi e delle uscite segue una logica "fisica", ossia i dati sono trasmessi nella forma dei messaggi grezzi che l'AMC si aspetta sulle sue interfacce di rete (nel caso dei segnali analogici, si intendono già digitalizzati), e sono identificati attraverso le loro coordinate fisiche, secondo il sistema di coordinate specifico di ogni interfaccia (descritto al § 2.2.1).

Anche il protocollo di comunicazione riflette la stratificazione delle tecnologie nell'arco dei decenni. La parte più antica (che comprende l'interfaccia ARINC 429) definisce l'invio e la ricezione sotto forma di comandi di lettu-

ra/scrittura grezza su indirizzi di memoria interna del target – una sorta di memoria condivisa remota – poiché nei primi tempi non esisteva la divisione tra target e driver e stava tutto in un unico eseguibile. Le interfacce aggiunte successivamente, come il recente AFDX, indirizzano i messaggi secondo le loro coordinate (nel caso dell'AFDX ad esempio sono Virtual link e porta UDP).

2.1.1.3 Implementazione dello scheduler

Lo scheduler dell'AMC (definito nel § 1.1.2.3 a pagina 14) è così approssimato: ogni task dell'AMC è un thread (un task Ada) ma l'esecuzione dei deterministici è serializzata artificialmente (ordinandoli per priorità), in modo che in ogni istante solo un task deterministico può essere attivo. I task statistici invece sono dei thread liberi di girare, a priorità più bassa. In questo modo non si riesce a simulare le situazioni di *overrun,* che si verificano quando un task supera il tempo consentito, ma come già detto sopra la differenza tra l'hardware e il sistema operativo originali e il VTE sono molto grandi e una tale simulazione sarebbe così approssimativa da risultare inutile.

La sovrapposizione tra input/output comandato dal driver ed esecuzione dei task è evitata sospendendo tutti i task tra la fine di un frame e l'inizio del successivo. Al contrario durante l'esecuzione dei task la comunicazione col driver è lasciata in attesa (la figura 2.2).

2.1.2 Limiti

Essendo un retarget, il VTE presenta gli stessi limiti (1.1.3.1 a pagina 15): date le differenze con hardware e sistema operativo originali, non è possibile riprodurre con precisione il carico della CPU, l'utilizzo di memoria e di conseguenza i tempi di risposta, se non con la granularità di un singolo *minor frame* per i soli task deterministici. Inoltre, la differenza tra i sistemi operativi e le implementazioni del Middleware può esporre a comportamenti anomali (bug) differenti.

Per quanto riguarda l'utilizzo all'interno del simulatore, il VTE non implementava alcune funzionalità che invece sono richieste dal simulatore, ad esempio la gestione di più di un AMC virtuale alla volta; pertanto sono assenti quelle funzioni di sincronizzazione e scambio dati tra le due copie, necessarie nella configurazione duale ridondata riprodotta nel simulatore.

Infine, l'implementazione del VTE ignora il problema dell'*endianness* assumendo che la rappresentazione interna dei dati sia la stessa della rete; infatti non prevede funzioni di conversione dei dati in ingresso e in uscita.

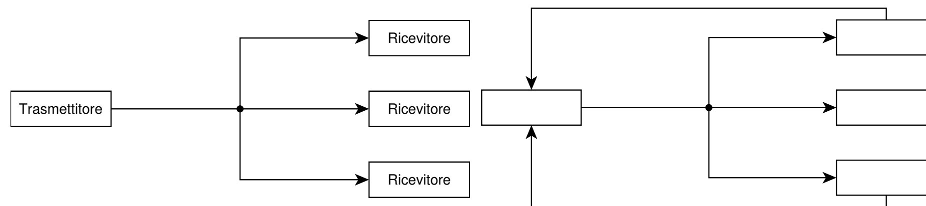


Figura 2.3: Esempi di collegamento ARINC 429. Ogni arco rappresenta una coppia di fili intrecciati. a) Esempio di collegamento monodirezionale; b) Esempio di cablaggio bidirezionale: è necessario un doppino per ogni trasmettitore.

8	7	6	5	4	3	2	1	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Label								SDI		Dati																			SSM		P

Figura 2.4: Struttura di un messaggio ARINC 429. I bit sono rappresentati in ordine di trasmissione.

Inoltre, in ragione della stratificazione delle modifiche avvenuta nei decenni, la struttura del Middleware accoppia eccessivamente la gestione dei dati e il componente di comunicazione col driver limitando di fatto le possibilità di riutilizzo del codice, cosicché la scelta di sostituire quest'ultimo componente ha richiesto una riscrittura di ampie parti del Middleware.

2.2 Trasporto di dati avionici

2.2.1 Bus e protocolli di comunicazione

La comunicazione tra gli apparati all'interno di un velivolo si serve di diversi bus e diversi protocolli. Di seguito sono illustrati i più importanti tra quelli impiegati in questo progetto, delle cui interfacce si è dovuto ricostruire il comportamento, a livello del Middleware.

2.2.1.1 ARINC 429

ARINC 429 è lo standard più diffuso sugli aeromobili civili e talvolta su quelli militari, data l'elevata disponibilità di componenti già disponibili sul mercato (*commercial-off-the-shelf*, COTS). Si tratta di un collegamento punto-multipunto monodirezionale realizzato con un cavo che collega un solo trasmettitore a un numero variabile di ricevitori, che ricevono lo stesso segnale. Il cavo è un doppino schermato da 78 Ω, che contiene una coppia di fili intrecciati [AIT]. Il segnale digitale è trasmesso come differenza di potenziale sui due fili con una modulazione con ritorno allo zero (*return-to-zero*, RZ); questa modulazione prevede tre stati, distinti per differenza di potenziale: alto (+10V), nullo (0V), basso (-10V). Il bit 1 viene trasmesso come "alto"

per la prima metà del ciclo e "nullo" per la seconda metà; il bit 0 al contrario è "basso" seguito da "nullo". In questo modo il segnale è sincronizzato automaticamente e non è necessario nessun segnale di clock. Lo standard ARINC 429 prevede solo due frequenze di trasmissione, 12,5 o 100 kbit/s. Ogni collegamento è configurato per trasmettere in una sola delle modalità.

I dati vengono trasmessi come parole da 32 bit; tra una parola e l'altra è prevista una pausa di almeno 4 bit, in cui il bus è nello stato "nullo". Il trasmettitore trasmette continuamente, o dati o lo stato nullo. All'interno di una parola dati, i primi 8 bit trasmessi contengono la *label*, che identifica il dato. Per convenzione, i numeri di label sono scritti in ottale. Sebbene il produttore sia libero di scegliere le label tra le 255 accettabili, alcune di queste, di uso comune, hanno un significato convenzionale (ad esempio 010 "Present position – Latitude"). L'ultimo bit della parola è il bit di parità, che è mantenuta sempre dispari, fungendo da controllo di errore (la figura 2.4). Questi sono gli unici due campi obbligatori; tutti gli altri possono essere usati per contenere i dati. Normalmente i due bit 30 e 31 sono dedicati al campo SSM (*sign-status matrix*) che contiene informazioni sulla validità del dato e in alcuni casi il segno; la sua codifica dipende dalla codifica del dato. Valori tipici sono "normal operation", "failure/warning", "no computed data" e "test". I 19 o 21 bit restanti contengono i dati, che possono essere codificati come numeri interi con o senza segno (BNR), *binary coded decimal* (BCD) oppure alfanumerici con l'alfabeto ISO n° 5. I due bit 9 e 10 contengono opzionalmente l'identificatore della sorgente o del destinatario (*source/destination identifier*, SDI).

Un trasmettitore può essere collegato a più ricevitori, che ricevono gli stessi dati. Se un ricevitore è interessato solo ad un sottoinsieme dei dati trasmessi può scartarli basandosi sul loro numero di label o sul valore del campo SDI, che può distinguere fino a tre o quattro destinatari – il valore 0 è solitamente utilizzato per i dati rivolti a tutti i destinatari. Le coordinate per rappresentare un messaggio ARINC 429 sono dunque l'identificativo del bus (ossia il canale), label e SDI ove previsto.

A livello applicativo, il bus può essere usato in due modalità, *time-framed* (TF) e *non-time-framed* (NTF), a seconda del comportamento del buffer di invio o di ricezione. Per i canali NTF, l'invio e la ricezione sono effettuati attraverso una coda: i messaggi sono inviati nell'ordine in cui l'applicazione li scrive sul buffer e sono letti nello stesso ordine in cui sono ricevuti. I canali TF sono invece utilizzati per i dati che sono trasmessi periodicamente e hanno quindi una validità temporale: lo smistamento dei dati secondo i loro label e SDI viene fatto a livello di sistema operativo o di scheda di rete; il nuovo messaggio ricevuto va a sostituire quello vecchio e un indicatore di

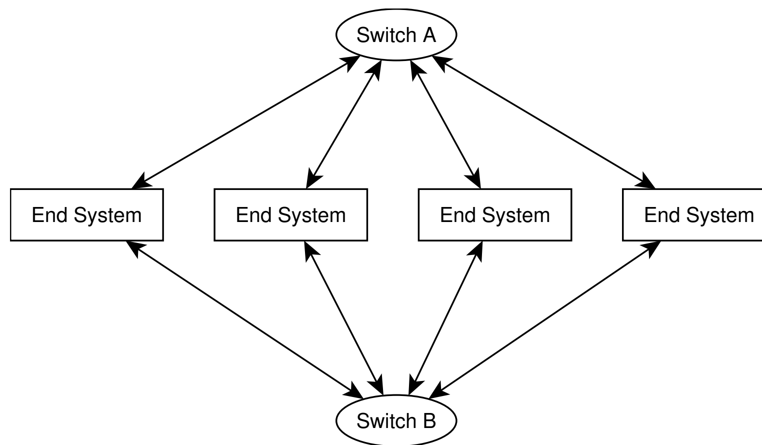


Figura 2.5: Esempio di rete AFDX

"freschezza" segnala all'applicazione se il dato è stato aggiornato dall'ultima lettura. Per la trasmissione la scheda di rete segue ciclicamente una tabella di temporizzazione impostata all'avvio. Questa tabella divide il periodo in *frame* – proprio come lo scheduler dell'AMC – e per ogni frame sono specificate per ogni canale le label dei messaggi da inviare. Quando l'applicazione chiede la scrittura di un dato, questo viene sostituito a quello vecchio nel buffer ma viene inviato solo nel *frame* stabilito dalla tabella. Sono possibili eccezioni per quei messaggi che su un canale TF non devono essere inviati periodicamente. In tal caso si parla di messaggi "aciclici" e sono configurati attraverso un'altra tabella.

2.2.1.2 Avionics Full-Duplex Switched Ethernet (AFDX)

Il protocollo AFDX nasce come iniziativa dell'Airbus durante lo sviluppo dell'A380, per superare i limiti dell'ARINC 429 sia aumentando la portata della rete sia riducendo il cablaggio complessivo. È pubblicato come standard ARINC 664. L'AFDX è basato sullo standard Ethernet con alcune differenze nel protocollo di trasmissione in modo da rendere il comportamento della rete adatto all'uso in sistemi *safety-critical* e *real-time*; lo strato fisico tuttavia è lo stesso dell'Ethernet. Ciò permette il riutilizzo di componenti hardware già disponibili sul mercato.

I terminali, detti *End system*, sono collegati ad uno switch attraverso due coppie di fili intrecciati (*twisted pairs*), una per ricevere e l'altra per trasmettere (*full-duplex*), in modo da evitare collisioni e causare ritardi imprevedibili nella trasmissione. L'utilizzo di una rete a stella ha lo scopo di minimizzare la quantità di cavo necessaria per collegare tutti gli apparati.

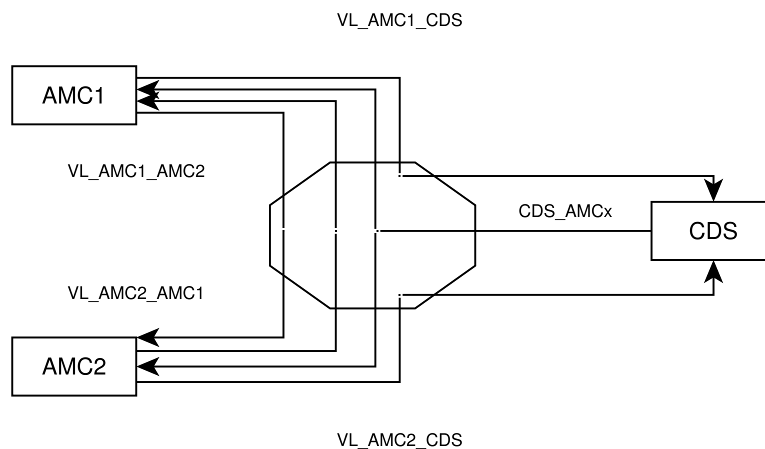


Figura 2.6: Rappresentazione grafica dei Virtual Link AFDX, sotto forma di collegamenti diretti che attraversano lo switch (al centro)

Per esigenze di sicurezza la rete AFDX è ridondata: vi sono due reti parallele e indipendenti, ognuna coi suoi cavi e il suo switch. Le schede di rete dei terminali o i loro driver si preoccuperanno di scartare i messaggi identici. [GE]

Le differenze tra AFDX e Ethernet stanno in parte nell'indirizzamento (livelli OSI 2 e 3) e in parte nella temporizzazione della trasmissione. Lo switch è configurato per collegare tra loro gli apparati attraverso connessioni multicast virtuali dette *virtual link* (VL). Ogni VL rappresenta una connessione monodirezionale con un solo trasmettitore e un certo numero di destinatari, imitando così i collegamenti fisici dell'ARINC 429. La destinazione è segnata nell'intestazione del pacchetto Ethernet occupando i 16 bit più bassi del MAC di destinazione col numero del VL (gli altri bit dell'indirizzo sono costanti e non utilizzati). I VL si spartiscono i 100 Mbit/s di banda disponibile secondo una configurazione impostata staticamente. A ogni VL è assegnata una banda massima attraverso due parametri BAG e L_{max}. Il BAG (*Bandwidth Allocation Gap*) è l'intervallo minimo tra le trasmissioni di due pacchetti dello stesso VL, mentre L_{max} è la lunghezza massima dei pacchetti di un VL. Il rispetto di questi parametri è un obbligo dell'End system mittente, che a questo scopo ha un componente detto *VL scheduler* che ritarda opportunamente l'invio dei pacchetti secondo uno degli algoritmi disponibili.

I dati sono generalmente trasmessi come pacchetti UDP e il numero della porta è utilizzato per distinguere le diverse applicazioni, cosicché per identificare un messaggio sono sufficienti il numero del VL, la porta di origine e la porta di destinazione. A livello applicativo si distinguono due tipi di

porte: *sampling* e *queuing*, a seconda del comportamento del buffer del ricevitore. Le porte *queuing* accodano i pacchetti ricevuti, che vengono letti dall'applicazione nell'ordine di ricezione, rimuovendoli dalla coda. Le porte *sampling* sono utilizzate per i dati che vengono aggiornati periodicamente: il nuovo pacchetto sostituisce nel buffer di ricezione quello vecchio e un indicatore di "freschezza" segnala all'applicazione se il pacchetto è stato aggiornato dall'ultima lettura, che non rimuove il dato vecchio. Le porte sampling e queuing sono analoghe alle modalità TF e NTF delle schede ARINC 429, con la differenza che nel nostro caso non è il driver AFDX a preoccuparsi dell'invio periodico dei dati ma questo compito è lasciato al livello applicativo.

La struttura del contenuto di un messaggio AFDX è lasciata libera all'applicazione. Tuttavia lo standard suggerisce l'utilizzo di uno dei formati consigliati. I messaggi possono avere una struttura esplicita (definita all'interno del messaggio stesso) o implicita (concordata tra i partecipanti all'applicazione). La struttura implicita definisce un numero limitato di tipi di dati, tra cui floating point IEEE 754 a 32 o 64 bit, interi con o senza segno da 32 o 64 bit, stringhe, booleani e dati opachi. I dati sono comunque allineati a 32 bit, mentre i primi 4 byte sono riservati per usi futuri. Il contenuto del messaggio è definito come una successione di *data set*; prima di ogni gruppo di quattro data set vi sono quattro byte di *Functional Status Set* (FSS), che specificano ognuno lo stato di un data set. Lo stato può essere "normal operation", "failure/warning", "no computed data" o "functional test" esattamente come per il campo SSM dei messaggi ARINC 429.

2.2.1.3 Altri protocolli

All'interno di un velivolo sono utilizzati anche protocolli non strettamente avionici. Segue una breve lista.

CAN *Controller Area Network.* È uno standard progettato appositamente per l'automazione in campo *automotive* ma ha trovato applicazione anche in campi diversi. Si tratta di un bus seriale broadcast (tutti i terminali ricevono gli stessi messaggi) senza arbitraggio centrale. Ogni nodo può trasmettere sul bus, ma non contemporaneamente; i nodi si organizzano da soli e in caso di collisioni vince il messaggio con priorità più alta, senza danneggiare il dato già in trasmissione, mentre gli altri nodi ritenteranno la trasmissione se necessario. È utilizzato ad esempio per collegare sensori e attuatori ai computer di bordo.

EIA RS-422 Un collegamento seriale che utilizza un protocollo di trasmissione a byte. È utilizzato per collegarsi ad alcuni tipi di radio o di sensori.

Discreti Trasmettono un solo dato binario (aperto/chiuso) e corrispondono a degli interruttori. In uscita, possono essere usati per comandare l'accensione di alcuni apparecchi o spie mentre in ingresso sono spesso collegati agli interruttori presenti in cabina e a segnali di stato di alcuni apparati (per esempio l'allarme incendio dell'Auxiliary Power Unit).

Analogici Trasmettono un solo dato analogico sotto forma di tensione o di corrente, che viene poi convertito in digitale con un ADC. Oltre a misurare tensioni e correnti dei cavi di alimentazione stessi, spesso rappresentano pressioni e temperature.

2.2.2 Rappresentazione dei dati avionici

Nel corso del progetto si è dovuto mettere il software a conoscenza della struttura interna dei dati e della loro codifica, per compiere le conversioni che si sono rese necessarie. Analogamente, nella fase di integrazione col simulatore, le medesime informazioni hanno reso possibile la comunicazione tra un mondo basato sui *bus* (il retarget) a uno basato sulle *variabili* (il Framework).

Di seguito sono descritte le codifiche impiegate in questo progetto e le tecnologie adottate per rappresentare – sia per l'uso elettronico sia per la consultazione – le informazioni sull'identificazione dei dati e sulla loro struttura.

2.2.2.1 Codifiche dati

Indipendentemente dal mezzo di trasmissione – purché sia digitale – le codifiche utilizzate per la trasmissione dei dati costituiscono un insieme molto limitato. Di seguito sono elencate quelle da noi utilizzate, con le loro sigle.

BCD *Binary Coded Decimal.* Con questo termine si indicano le codifiche che rappresentano ogni cifra decimale separatamente. Nel nostro caso ogni cifra è rappresentata con 4 bit eccetto quella più significativa, che può occupare un numero di bit inferiore se possibile. È usata ad esempio per le frequenze delle radio.

BIN/BNR Interi senza segno o in complemento a due. Si usa sia per rappresentare numeri interi che decimali a virgola fissa. In quest'ultimo

caso il trasmettitore e il ricevitore devono concordare sulla risoluzione a e lo scostamento b, in modo che il valore rappresentato sia $v = ax+b$, con x input intero.

SBIN Interi come segno e modulo.

DIS Dati discreti. Possono essere valori booleani oppure, se su più bit, numeri interi che rappresentano un'etichetta (enumerati).

FLT Virgola mobile IEEE 754, a 32 o 64 bit.

Dato che un messaggio, indipendentemente dal protocollo, può contenere più campi, oltre alle informazioni sulla codifica sono necessarie le informazioni per l'individuazione del campo all'interno del messaggio. Di solito si usano la lunghezza in bit, il numero del byte iniziale e la posizione del bit più significativo (MSB) all'interno di tale byte. La convenzione più usata assegna al primo byte il numero 1. Per l'ARINC 429 il numero del byte non è necessario poiché i bit sono già numerati da 1 a 32. Salvo rari casi espressamente indicati, la codifica è *big-endian*.

2.2.2.2 Interface Control Document (ICD) Database

Ogni componente dell'apparecchiatura avionica è accompagnato da un documento detto *Interface Control Document* (o *Description*) che descrive le sue interfacce, in termini di ingressi e uscite, di cui sono definiti la struttura dei dati e il loro significato. Il documento può descrivere l'interfaccia tra due specifici apparati o più generalmente tutti gli ingressi e le uscite di un apparato.

Originariamente tali documenti erano scritti e trascritti a mano; ora le informazioni sulle interfacce dei componenti, almeno per quanto riguarda la struttura dei dati, sono interamente codificate in un database, in modo che siano processabili elettronicamente (*machine-readable*). Dal database viene comunque generata la documentazione cartacea o elettronica, a scopo di archiviazione. Il database contiene le seguenti informazioni.

- Descrizione della topologia di rete. Ad ogni connessione è assegnato un nome univoco, una sorgente e una o più destinazioni. Di ogni interfaccia è inoltre descritto il tipo di bus e le informazioni che lo caratterizzano (ad esempio, di un bus ARINC 429 è segnata la frequenza).

- Descrizione dei messaggi scambiati, cui è assegnato un nome univoco. Ogni messaggio è legato ad un'interfaccia ed è identificato attra-

verso le coordinate tipiche di quell'interfaccia (per esempio, nel caso dell'AFDX sono indicati il VL e la porta). La descrizione contiene anche informazioni indipendenti dal protocollo quali il periodo di trasmissione.

- Descrizione dei campi dei messaggi, identificati attraverso lunghezza in bit, MSB e posizione del byte iniziale nel messaggio. Ad ogni campo è assegnato un nome e sono descritte tutte le informazioni necessarie per la codifica/decodifica: il tipo di codifica, la scala (risoluzione e scostamento), l'unità di misura e un insieme di coppie nome-valore se si tratta di un tipo enumerativo.

Il database ha un ruolo fondamentale nella costruzione del test system poiché permette alle sequenze di test e agli emulatori di riferirsi ai dati tramite il loro nome, ignorando ove possibile il protocollo con cui sono trasmessi e le loro coordinate. Ciò agevola il riutilizzo del software tra apparati e velivoli di modelli diversi.

Tuttavia l'utilità di un ICD database è solo interna all'azienda. Ad oggi non sembra esistere un formato standard per lo scambio di ICD, che tra fornitore e cliente devono spesso essere trascritti a mano oppure con la scrittura di codice ad hoc. Ciò accade anche tra società dello stesso gruppo.

Inoltre nel nostro caso l'ICD è gestito attraverso lo stesso software utilizzato per la tracciatura dei requisiti, ma i dati sono inseriti sempre a mano e non sono direttamente legati al codice, che è scritto da terze persone leggendo i requisiti. Può accadere di trovare degli errori (campi con indici sbagliati ad esempio), delle mancanze, o una nomenclatura non consistente tra ICD e codice. In altre situazioni il codice di bordo che si occupa della lettura, scrittura e conversione dei dati è generato automaticamente a partire dall'ICD, migliorando molto la manutenzione del codice, ma non in questo progetto.

Nel caso del simulatore, dall'ICD database sono stati generati gli scheletri dei file di configurazione che mappano i dati "fisici" alle variabili del Framework di simulazione. Ciò ha fornito i valori iniziali; le corrispondenze vere e proprie sono state inserite manualmente, poiché non esiste nessun documento *machine-readable* che specifica la semantica di ogni dato.

2.3 Il problema dell'*endianness*

La *endianness* di un calcolatore è legata al modo in cui sono rappresentati i dati che occupano più di un byte. [Intel, 2004] È la caratteristica che crea più complicazioni nell'adattamento del software ad un'architettura diversa (nel nostro caso da SPARC a Intel x86). I due tipi di endianness più importanti

Little-endian	44	33	22	11
Big-Endian	11	22	33	44

Little-endian	0010 0010	1100 1100	0100 0100	1000 1000
Big-endian	0001 0001	0010 0010	0011 0011	0100 0100

Figura 2.7: Esempio di rappresentazione dei dati in architetture di diversa endianness. *Il dato codificato è sempre* 44332211_{16}. a) *rappresentando i byte come entità compatte, come se i loro bit fossero trasmessi in parallelo,* b) *rappresentando tutti i bit nell'ordine naturale dell'architettura. Si noti come i bit appaiono invertiti ma i byte rappresentati siano esattamente gli stessi.*

sono *big-endian* e *little-endian*[1]. Le architetture big-endian rappresentano gli interi (così come gli altri tipi numerici) a partire dal byte più significativo; al contrario in quelle little-endian il primo byte è quello significativo. La conseguenza più evidente si nota osservando com'è rappresentato in memoria un numero (figura 2.7). Prendendo ad esempio il numero 11223344 (in esadecimale), in un'architettura big-endian esso sarà rappresentato con quattro byte nell'ordine 11 22 33 44 – così come siamo usi scrivere i numeri, da sinistra a destra – mentre un'architettura little-endian lo stesso numero sarà rappresentato come 44 33 22 11. Alla classe big-endian appartengono le piattaforme Motorola 68k, Sun SPARC, Intel Itanium, la Java Virtual Machine; alla famiglia little-endian appartengono tra gli altri gli Intel x86 e i DEC Alpha. Alcuni processori, come gli ARM o i MIPS possono comparire in entrambe le versioni e tra questi alcuni modelli permettono di cambiare a piacimento l'endianness attraverso uno speciale registro del processore. Esistono anche altri tipi minoritari di endianness, i "*middle-endian*", che nell'esempio precedente corrispondono agli ordini 11 33 22 44 oppure 44 22 33 11. La loro diffusione resta scarsa anche a causa delle loro difficoltà di utilizzo.

L'endianness di un'architettura si rispecchia anche nei formati nativi di rappresentazione dei dati. Ad esempio il formato immagine BMP, sviluppato dalla Microsoft per il sistema operativo Windows, mirato specificamente agli Intel x86, ha una rappresentazione dei dati little-endian, mentre il formato MacPaint sviluppato dalla Apple negli anni '80 per MacOS ha una rappresentazione big-endian in quanto era quella naturale dei processori Motorola impiegati dalla Apple all'epoca. Le stesse distinzioni valgono per i protocolli

[1]Questi due termini derivano dalle fazioni dei Lillipuziani de *I viaggi di Gulliver* di J. Swift, che disputavano se le uova sode dovessero essere rotte dal lato più grande o dal lato più piccolo. Allo stesso modo, così come il dibattito tra i Lillipuziani *big-endian* e *little-endian* è una questione di lana caprina, lo stesso dibattito in informatica prende spesso una piega più politica che tecnologica.

di rete, pertanto esistono protocolli big-endian (TCP/IP ad esempio) e protocolli little-endian (come l'USB). Quando un software scritto per una piattaforma riceve dei dati in un formato con un'endianness diversa dalla sua, deve attuare delle operazioni di spostamento dei byte (*byte-swapping*) per poterli decodificare. Questo è il motivo per cui i formati che sono sviluppati per una determinata piattaforma hanno la stessa endianness della piattaforma – risparmiando qualche istruzione le operazioni di lettura e scrittura possono essere più veloci.

2.3.1 Complicazioni dovute all'endianness

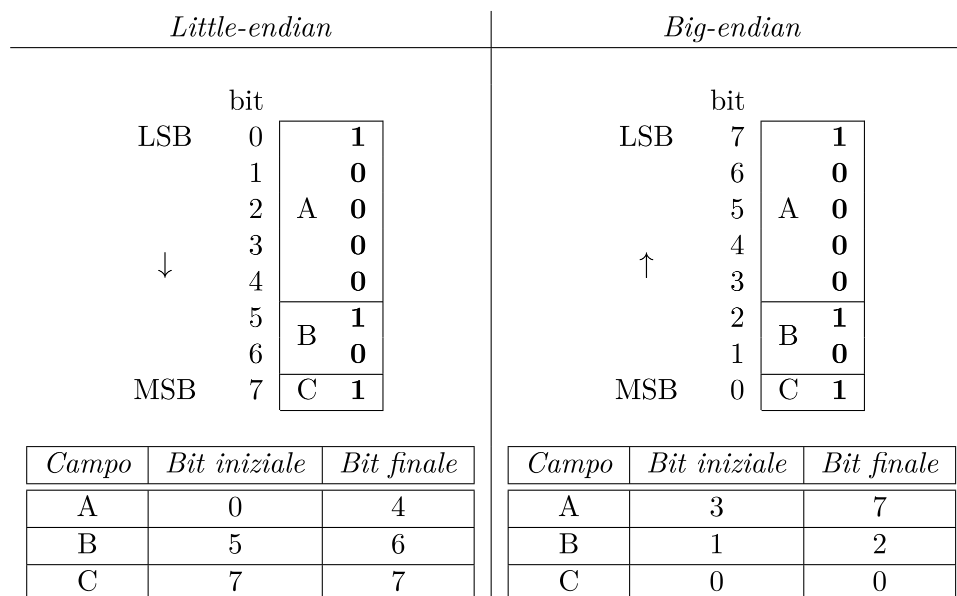
In generale il software è scritto considerando una sola endianness, o peggio una sola piattaforma. Così si rischia di produrre del codice non portabile, che ricompilato per un'altra piattaforma ha un'alta probabilità di manifestare comportamenti diversi da quelli previsti. La compatibilità in questo caso dipende solo dal modo in cui il codice è scritto, non dalle operazioni che deve svolgere.

Oltre al caso del *retarget*, i problemi sorgono quando sono messi in comunicazione apparecchi programmati per endianness diverse. Esistono delle linee guida e dei *best-known methods* per scrivere del codice multipiattaforma, indipendente dall'endianness (*endian-neutral*). [Intel, 2004] Il mancato rispetto di queste linee guida può causare problemi di compatibilità. Ad esempio, leggere il byte meno significativo di un intero da 4 byte attraverso il suo indirizzo è una pratica che porta risultati diversi con endianness diverse.

Per la rappresentazione esterna dei dati esistono tre soluzioni: fissare un'endianness e mantenerla, inserendo istruzioni di byte-swapping nella versione big- o little-endian del software; utilizzare un formato indipendente dall'endianness, per esempio un formato testuale come l'XML. Nel nostro caso, dato che uno degli obiettivi era la riproduzione il più possibile fedele del comportamento e degli output del sistema, si è mantenuta come rappresentazione esterna dei dati quella originale, big-endian.

2.3.2 Il caso dei *bit field*

Alcuni linguaggi di programmazione, tra cui il C e l'Ada, permettono di definire delle strutture dati con campi di lunghezza non multipla di un byte (*bit-field*). Queste definizioni risultano molto comode per rappresentare strutture dati di basso livello, di protocollo. Ad esempio, il primo byte dell'intestazione di un pacchetto IP è composto da due campi, "versione" e



Little-endian

Campo	*Bit iniziale*	*Bit finale*
A	0	4
B	5	6
C	7	7

Big-endian

Campo	*Bit iniziale*	*Bit finale*
A	3	7
B	1	2
C	0	0

Figura 2.8: Struttura con bit-field a cavallo dei byte e rappresentazione dell'intera struttura come sequenza di bit. I campi sono posizionati in modo da definire la stessa struttura in entrambi i casi; per evidenziare la posizione del bit meno significativo (LSB) di ogni campo, i valori dei campi sono fissati a 1.

"lunghezza dell'intestazione", di 4 bit ciascuno. Lo svantaggio di questa rappresentazione è la totale dipendenza dall'endianness. Nei sistemi big-endian la numerazione dei bit all'interno dei byte, così come dei byte all'interno di un intero, parte dal bit più significativo; l'opposto nelle architetture little-endian (la figura 2.8). Il riempimento dei byte con i bit-field segue lo stesso ordine. Questo accade perché se la numerazione dei bit fosse opposta a quella dei byte, i campi che si trovano a cavallo di due byte occuperebbero intervalli di bit discontinui qualora l'intera struttura venisse letta come un singolo numero intero (la figura 2.9). Se il linguaggio di programmazione non fornisce altri strumenti, l'unica soluzione per scrivere codice indipendente dall'endianness è l'uso di operazioni aritmetiche (*shift-and-mask*) per l'estrazione dei bitfield anziché le definizioni.

2.3.3 L'endianness in Ada

L'Ada 95, il linguaggio utilizzato dal software applicativo dell'AMC oggetto di questa tesi, non fornisce particolari ausili per la scrittura di codice endian-neutral; di conseguenza ci si deve attenere alle stesse linee guida sopracitate. Per il software di basso livello, l'Ada 95 incoraggia l'utilizzo di strutture dati (che chiama *record*) con rappresentazioni esplicite del loro formato in termini

Campo	*Lunghezza*
LABEL	8
SDI	2
DATA	19
SSM	2
P	1

Little-endian

→

	bit		byte 0		byte 1		byte 2		byte 3
LSB	0		**1**	SDI	**1**		**0**		**0**
	1		**0**		**0**		**0**	...DATA	**0**
	2	LABEL	**0**	DATA...	**1**	...DATA...	**0**		**0**
	3		**0**		**0**		**0**		**0**
↓	4		**0**		**0**		**0**		**0**
	5		**0**		**0**		**0**	SSM	**1**
	6		**0**		**0**		**0**		**0**
MSB	7		**0**		**0**		**0**	P	**1**

Sequenza di bit risultante:
10000000 10100000 00000000 00000101
In decimale (singoli byte): 1, 5, 0, 160
In esadecimale (intero da 32 bit): 1050000A

Big-endian

→

	bit		byte 0		byte 1		byte 2		byte 3
LSB	7		**1**		**0**		**0**	P	**1**
	6		**0**		**0**		**0**	SSM	**1**
	5		**0**	DATA...	**0**		**0**		**0**
↑	4	LABEL	**0**		**0**	...DATA...	**0**		**1**
	3		**0**		**0**		**0**	...DATA	**0**
	2		**0**		**0**		**0**		**0**
	1		**0**	SDI	**1**		**0**		**0**
MSB	0		**0**		**0**		**0**		**0**

Sequenza di bit risultante:
00000001 01000000 00000000 00001011
In decimale (singoli byte): 1, 64, 0, 11
In esadecimale (intero da 32 bit): 0140000B

Figura 2.9: Rappresentazione di una struttura che adopera dei bit-field *nelle sue versioni big- e little-endian. L'esempio è quello di un pacchetto ARINC 429 (v. fig. 2.4), definito attraverso le lunghezze dei singoli campi (in alto). Per mostrare la posizione del bit meno significativo di ogni campo, tutti i campi hanno valore 1.*

```
type ARINC429_WORD is
record
        LABEL:  INTEGER range 0 .. 255;
        SDI:    INTEGER range 0 .. 3;
        DATA:   INTEGER range 0 .. 16#7FFFF#;
        SSM:    INTEGER range 0 .. 3;
        PARITY: BOOLEAN;
end record;

for ARINC429_WORD use record
        LABEL  at 0 range 0 .. 7;
        SDI    at 0 range 8 .. 9;
        DATA   at 0 range 10 .. 28;
        SSM    at 0 range 29 .. 30;
        PARITY at 0 range 31 .. 31;
end record;
```

Figura 2.10: Definizione di un record *in Ada 95 con una rappresentazione esplicita del formato.*

di bit iniziale e bit finale di ogni campo. Per i motivi descritti al punto precedente ciò non produce un risultato portabile. Se il software non è stato scritto dall'inizio considerando la compatibilità, sono necessarie modifiche alle strutture o la riscrittura di tutte le istruzioni che ne leggono o scrivono i campi. Ulteriori problemi sorgono per l'interpretazione dei "record varianti" – l'equivalente vago delle *union* in C – la cui struttura può variare in base al contenuto di uno dei campi.

In tempi recenti, per versioni dell'Ada successive a quella del 1995, sono state introdotte estensioni proprietarie [Quinot, 2013] che permettono di specificare l'endianness di una determinata struttura, ma hanno delle limitazioni [AdaCore, 2012]: non è applicabile per record più lunghi del massimo intero macchina, richiede modifiche del codice invasive (alterare tutte le dichiarazioni dei tipi coinvolti) e non vale per l'Ada 95.

2.4 Tecnologie per simulatori: *High Level Architecture*

Lo standard utilizzato dal simulatore in cui questo progetto si va a inserire è lo standard IEEE 1516, che definisce un'architettura di simulazione in tempo reale detta *High Level Architecture* (HLA). Lo scopo principale dello standard HLA è definire un'architettura comune che favorisse il riutilizzo di modelli e l'interoperabilità tra simulazioni diverse. La sua gestazione è iniziata nel 1995 da uno sforzo congiunto di governi, industria e università e la sua prima bozza è stata pubblicata nel 1996 [Dahmann et al., 1998], fino ad arrivare alla pubblicazione come standard IEEE nel 2000. Ha avuto fin

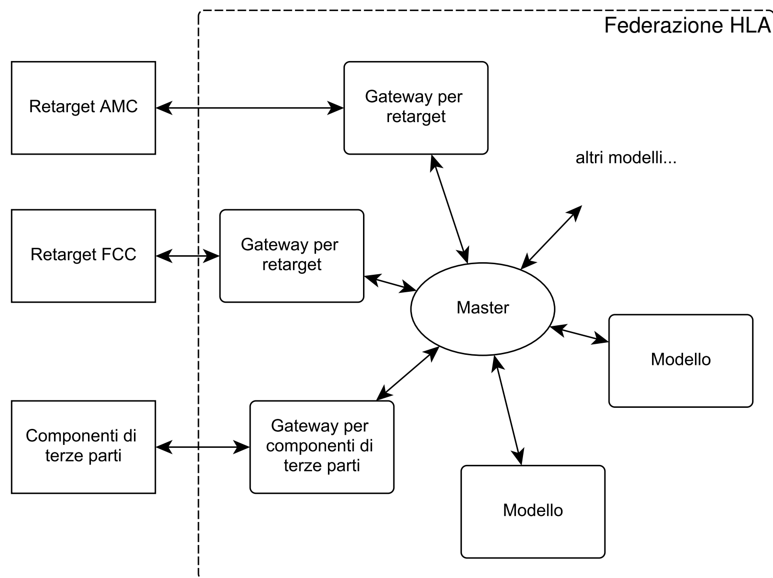


Figura 2.11: Rappresentazione molto schematica del framework di simulazione, nella versione impiegata dal nostro simulatore.

da subito sponsor importanti, tra cui il Department of Defense statunitense, che lo adotta tuttora come standard ufficiale per la simulazione. È ora in vigore la versione del 2010, che introduce alcuni cambiamenti – come l'uso dell'XML – per adattare l'architettura alla tecnologia degli ultimi anni.

2.4.1 Definizioni

Lo standard IEEE 1516 definisce la struttura, i princìpi, le linee guida e i requisiti della simulazione, arrivando a definire delle API in alcuni linguaggi di programmazione.

Un insieme di applicativi, avente un determinato nome, che contribuiscono alla simulazione in un dato momento è detto "federazione"; un applicativo che supporta l'interfaccia HLA ed è in grado di unirsi alla federazione è detto "federato". La simulazione è portata avanti dai federati stessi, che possono essere simulazioni di particolari parti (anche manuali), interfacce uomo-macchina o utilità (raccolta dati e visualizzazione, per esempio). L'infrastruttura software che permette l'operazione della federazione è detta *Run-Time Infrastructure* (RTI), utilizzata dai federati attraverso un'interfaccia standardizzata, e può essere vista come una sorta di sistema operativo distribuito. Non è definita un'implementazione standard per la RTI ma è definito in un sotto-standard (IEEE 1516.3) un processo di sviluppo delle federazioni (FEDEP). Un federato può essere collegato anche a elementi

esterni alla federazione, o a una RTI diversa. L'esecuzione di una federazione (*federation execution*) è l'operazione di una federazione come entità unica per il raggiungimento di un determinato obiettivo.

Vi sono tre modelli di oggetti all'interno di una federazione. Per primo vi è il *Simulation Object Model* (SOM), che specifica i tipi di informazioni che un singolo federato può fornire alla federazione o ricevere dagli altri federati. Le informazioni possono essere istanze di oggetti o interazioni (cioè eventi). Secondo è il FOM (*Federation Object Model*), che definisce le informazioni scambiate durante l'esecuzione nel raggiungimento di determinati obiettivi. Terzo è il *Management Object Model* (MOM), un insieme predefinito di costrutti che controlla l'esecuzione della federazione. Tipicamente il FOM è realizzato come sottoinsieme del SOM, selezionando le informazioni prodotte da ogni federato che sono richieste dagli altri. Anche se utilizza termini presi in prestito dalla programmazione ad oggetti, l'infrastruttura HLA non è necessariamente object-oriented, sebbene spesso l'implementazione lo sia. Ad esempio, gli oggetti del SOM sono descritti completamente dai loro attributi e non definiscono delle operazioni (metodi). Inoltre gli oggetti non interagiscono tra loro direttamente ma solo attraverso i federati (la cui implementazione è libera).

2.4.2 Servizi offerti dall'infrastruttura

Quello che segue è un elenco dei servizi più importanti forniti dalla RTI della HLA.

- Servizi che coordinano l'esecuzione (punti di sincronia, aggiunta/rimozione di un federato, salvataggio e ripristino).
- *Declaration management.* Permette ai federati di pubblicizzare i tipi di dati che inviano o ricevono, con un meccanismo *publish-subscribe.*
- *Object management.* Servizi che realizzano l'effettivo scambio di dati, sotto forma di oggetti e interazioni.
- *Time management.* Mantiene la sincronizzazione tra i federati, secondo un tempo distribuito virtuale, e assicura la causalità tra gli eventi.
- Altri servizi: gestione delle autorizzazioni/privilegi, gestione delle liste di distribuzione, servizi di supporto.

2.4.3 L'implementazione utilizzata

Nel nostro simulatore la sincronizzazione e la distribuzione dei dati sono affidati ad un nodo detto "master". Il tempo è gestito come tempo discreto, diviso in *frame* che sono sincronizzati dal master e hanno la stessa frequenza di aggiornamento della visuale esterna.

La comunicazione con software di terze parti o comunque software che non fa parte della Federazione – tra cui il retarget dell'AMC – è delegata ad appositi federati detti "gateway". Ogni federato rappresenta un componente della simulazione (l'impianto elettrico, il motore, i serbatoi, le radio, ...). Vi sono federati appositi per l'interfaccia umana (i display, i comandi, ...) e per i retarget sono utilizzati dei gateway scritti appositamente, che traducono i dati "fisici" nelle variabili della federazione e viceversa, e inviano il comando di step ai retarget secondo la loro frequenza, che è diversa da quella della simulazione.

Capitolo 3

Architettura del sistema

In questo capitolo è descritto il design di alto livello del "sistema retarget AMC" insieme alle scelte progettuali che sono state compiute e alle loro motivazioni.

3.1 Descrizione strutturale

Il prodotto "Retarget AMC" propriamente detto (figura 3.1, cfr. figura 1.5) è un unico processo, composto da codice applicativo, middleware (MW) e memoria condivisa (SHM), cui si collegano i processi "VTE Proxy" o "HLA Gateway" – a seconda della configurazione: rispettivamente "test" o "simulatore" – che insieme al Retarget compongono il "sistema retarget AMC". Quest'ultimo è visto dall'esterno come un'unica entità con una sola via d'accesso e nella versione simulatore ha il ruolo di nodo della federazione.

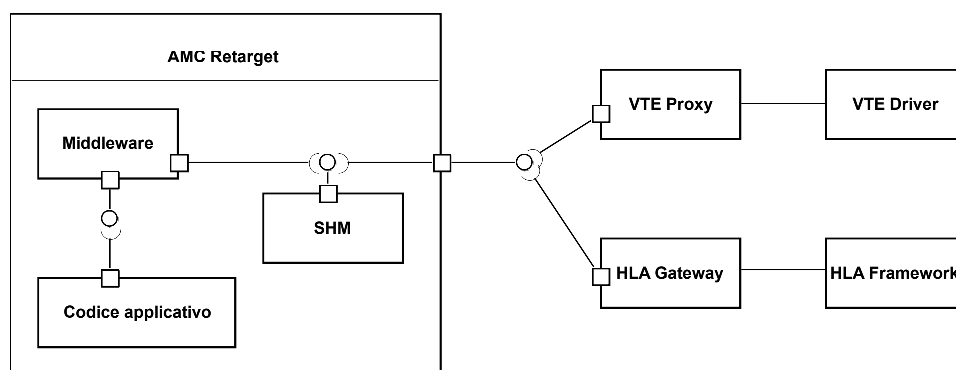


Figura 3.1: Diagramma di struttura composta del "sistema retarget" con una parte del contesto

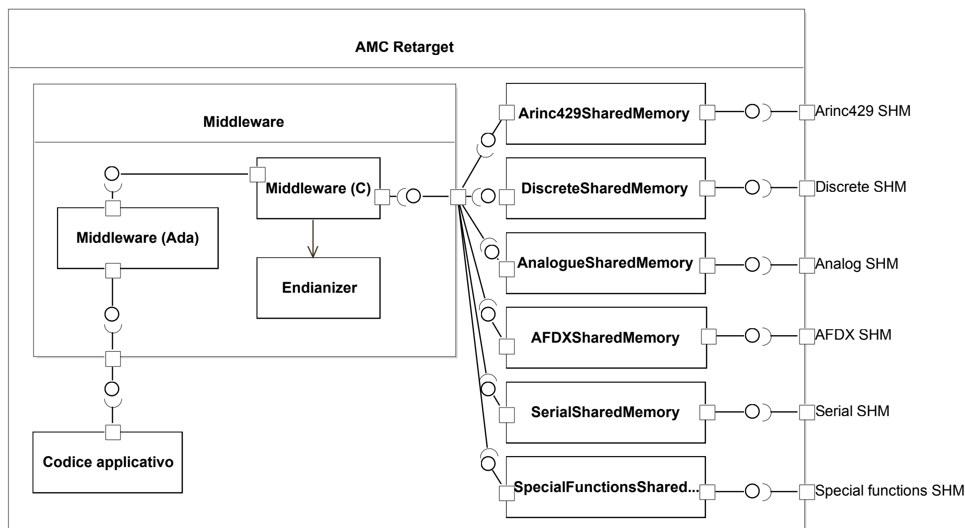


Figura 3.2: Diagramma di struttura composta del solo Retarget AMC più in dettaglio, con tutti i componenti

In questo capitolo si indicherà talvolta come "Target" il processo che ospita il Middleware e il codice applicativo, mentre con "Driver" si designerà il processo che ad esso si collega per pilotarlo, alternativamente il VTE Proxy o l'HLA Gateway.

La struttura è stata ricalcata da quella del VTE originale, scorporando la parte di protocollo di rete – affidata ora al VTE Proxy – da quella del Middleware. Le motivazioni di questa scelta sono spiegate al § 3.3.1.

Nella figura 3.2 è espansa la struttura del Middleware, suddiviso in una parte Ada e una parte C/C++, con l'aggiunta di un componente che abbiamo deciso di chiamare *Endianizer*, che si occupa della correzione dell'endianness dell'input/output (§ 4.3). La memoria condivisa, che non è un'entità atomica, è scomposta nelle sue parti indipendenti.

Di seguito la definizione di ogni singola parte.

3.1.0.1 Codice applicativo

Si tratta dello stesso codice di bordo dell'AMC reale, scritto in Ada 95, col minor numero di alterazioni possibile. Esso vede il mondo esterno solo attraverso il Middleware.

3.1.0.2 Middleware, parte Ada

È la reimplementazione dell'interfaccia del Middleware originale. Di fatto è solo un front-end per la parte C/C++.

3.1.0.3 Middleware, parte C/C++ (§ 4.1 a pagina 54)

Questo è il componente che svolge il vero ruolo del Middleware originale. Si occupa dell'inizializzazione del processo, della gestione dell'input/output, dello scheduling, del calendario, della ridondanza e sincronizzazione tra le due copie dell'AMC, della ricezione dei comandi speciali dal simulatore.

Poiché intende essere riutilizzato nei futuri progetti che si riferiscono alla stessa famiglia di Middleware, è realizzato come libreria condivisa.

3.1.0.4 *Endianizer* (§ 4.3 a pagina 75)

Componente che risolve la discordanza tra l'endianness del sistema originale e quello di destinazione inserendosi nell'input/output e alterando appropriatamente il contenuto dei messaggi. Questa operazione richiede la conoscenza della struttura dei dati.

3.1.0.5 Memorie condivise (§ 4.2 a pagina 69)

Implementate come librerie dinamiche, permettono la comunicazione tra processi diversi. Ognuna di esse simula una diversa interfaccia di rete e ne conserva lo stato.

In maniera analoga, le funzioni speciali del simulatore sono trasmesse attraverso un'interfaccia di rete simulata, sebbene l'implementazione differisca leggermente.

3.1.0.6 VTE Proxy (§ 4.4 a pagina 79)

Processo che agisce da tramite tra il Retarget vero e proprio e il VTE Driver, interpretando il ruolo di target. Comunica col VTE Driver nel suo protocollo originale attraverso un socket TCP.

3.1.0.7 HLA Gateway

Processo che funge da driver per il Retarget e allo stesso tempo da federato per il Framework di simulazione. Codifica i dati del Framework che arrivano sotto forma di dati ingegneristici in pacchetti che sono inviati al Retarget attraverso le memorie condivise (e viceversa); analogamente per i comandi speciali del simulatore. Non è oggetto di questa tesi.

3.2 Descrizione funzionale

Di seguito si darà una descrizione comportamentale del sistema visto dall'esterno, secondo le funzionalità offerte.

3.2.1 Funzionalità offerte al Driver

Nella figura 3.3 sono rappresentati i casi d'uso del Driver, con diagrammi di sequenza per i casi più significativi. Per le definizioni degli attori coinvolti "Driver" e "Target" si rimanda al paragrafo 3.1.

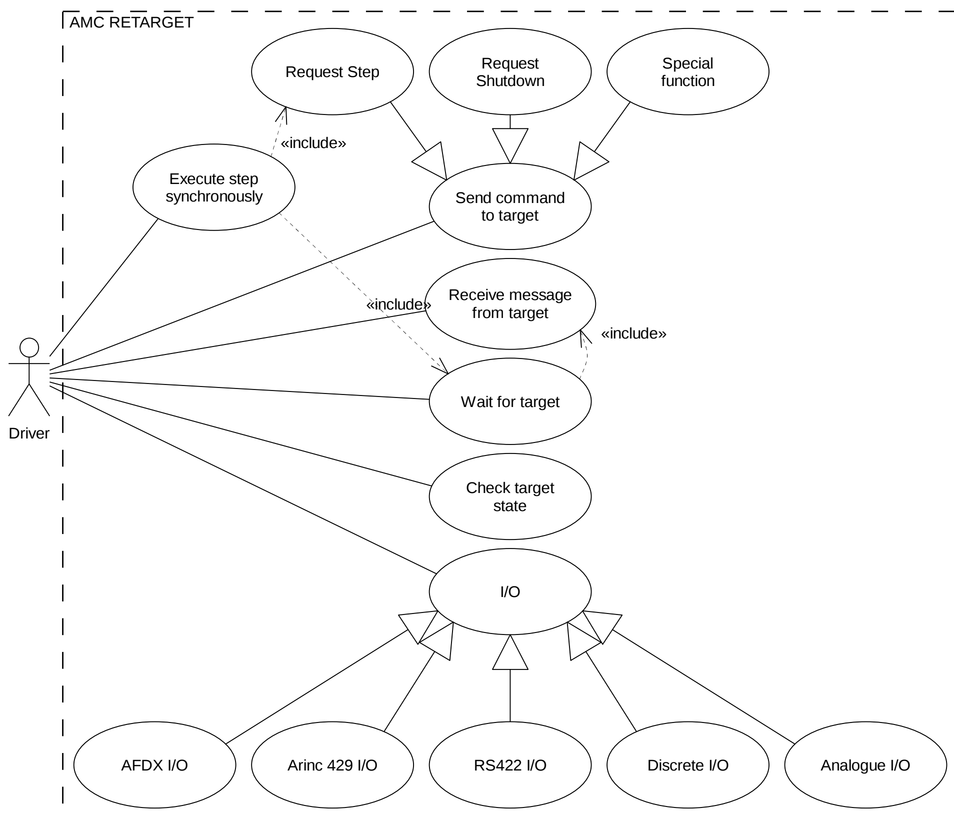


Figura 3.3: Diagramma dei casi d'uso del "sistema retarget" visto dal Driver

3.2.1.1 Invio comandi al Target

Il Driver può inviare comandi al Target, a scelta tra *step*, *reset*, *freeze* o *parameter change*, il cui significato è definito in ARINC-610B. I comandi hanno dei parametri che dipendono dal tipo di comando. Per esempio, *parameter change* ha come parametro l'identificatore della variabile e il nuovo

valore, mentre *step* non ha parametri. Altri comandi (*snapshot* e *recall* per citare alcuni di quelli suggeriti in ARINC-610B) possono essere aggiunti in seguito senza dover modificare il sistema.

Il Driver e il Target hanno un puntatore allo stesso segmento di memoria condivisa, che contiene due code di messaggi, una diretta al Target e l'altra diretta al Driver. Se la coda è piena, il thread del mittente viene messo in attesa finché il destinatario legge un messaggio; in caso contrario la procedura non è bloccante.

3.2.1.2 Ricezione messaggi dal Target

Analogamente, il Driver può ricevere comandi dal Target, finora di due tipi: *ready* (notifica della fine dell'esecuzione del frame) e *aural warning* (avvertimenti sonori; ogni avvertimento è identificato da un numero). Se un messaggio è già presente in coda, la procedura ritorna immediatamente, altrimenti si blocca finché il Target non invia un messaggio – lo scopo è non perdere messaggi. Per evitare riempimenti della coda e blocchi il Driver dovrebbe avere sempre un thread in ascolto.

3.2.1.3 Attesa del Target

Il Driver attende che il Target termini l'inizializzazione o l'esecuzione dei task di un frame. Ciò implica la ricezione di un messaggio di tipo *ready* da parte del Target.

3.2.1.4 Controllo stato del Target

Il Driver è in grado di controllare se il Target è disconnesso, connesso e libero o connesso ma occupato. Questa funzione è stata inserita per evitare di inviare messaggi senza che ci sia un Target in grado di riceverli, bloccando l'esecuzione.

3.2.1.5 Input/Output

Il Driver e il Target comunicano inviando e ricevendo dati sulle scheda di comunicazione simulate (ARINC 429, AFDX, discreti, analogici, porte seriali). Queste azioni sono realizzate come letture e scritture su segmenti di memoria condivisa. I parametri di queste procedure e il formato dei dati sono diversi per ogni interfaccia e dipendono dalle caratteristiche proprie del suo protocollo. Per questo motivo non esiste una via d'accesso unificata per tutte le schede. Le interfacce per il Driver e per il Target sono identiche;

è il chiamante a specificare di volta in volta la direzione della comunicazione. Ciò permette di realizzare dei monitor che osservano il contenuto della memoria condivisa in entrambe le direzioni.

3.2.1.6 Esecuzione sincrona di uno *step*

Nella modalità sincrona il Driver comanda l'esecuzione di un frame da parte del Target e attende il suo termine. Nonostante ciò comporti un sottoutilizzo delle risorse della CPU, si è preferito evitare che input/output del Driver ed esecuzione del Target si sovrappongano, con risultati indesiderati.

Nella figura 3.4 nella pagina successiva è rappresentata la tipica esecuzione sincrona di uno step: il Driver invia i dati sulle schede simulate, comanda l'esecuzione dello step, attende il messaggio di *ready* e legge i dati sulle interfacce simulate. Quanto è mostrato per l'AFDX vale per tutte le altre interfacce. Ciò è ripetuto per ogni minor frame, ogni 20 ms.

3.2.2 Funzionalità del sistema ridondato

Sono ora esaminati i casi d'uso relativi alla realizzazione del sistema duale ridondato, che comporta la compresenza di due processi Retarget AMC identici, chiamati AMC1 e AMC2. Per ognuna di esse l'altra istanza è detta "partner". Eccetto brevi intervalli transitori, delle due istanze una ha il ruolo di *master* e l'altra di *standby*. Il Master ha il controllo su un certo numero di caratteristiche (per esempio i display), mentre lo Standby si deve tenere sincronizzato in modo da poter prendere il suo posto in caso di guasto o di spegnimento. Se è attiva solo un'istanza, questa assume automaticamente il ruolo di master. Le due istanze gestiscono autonomamente la sincronizzazione, la rilevazione dello stato del partner e la negoziazione del ruolo (master/standby). Il cambio di ruolo può essere anche comandato dall'utente, così come su richiesta si effettua l'operazione di "allineamento dati" (DBU, *data base update*) con cui lo stato interno viene interamente copiato dal Master allo Standby, in caso di disallineamento o quando la seconda istanza viene avviata a una certa distanza dalla prima.

La linea seguita nella specifica di queste funzionalità è stata quella dell'imitazione il più possibile vicina dei collegamenti tra gli AMC reali.

3.2.2.1 Comunicazione dello stato al partner

Il retarget dell'AMC comunica periodicamente al partner i suoi parametri vitali, anche quando il partner non è in grado di ricevere. La comunicazione può essere realizzata in vari modi (memoria condivisa, socket UDP). Il

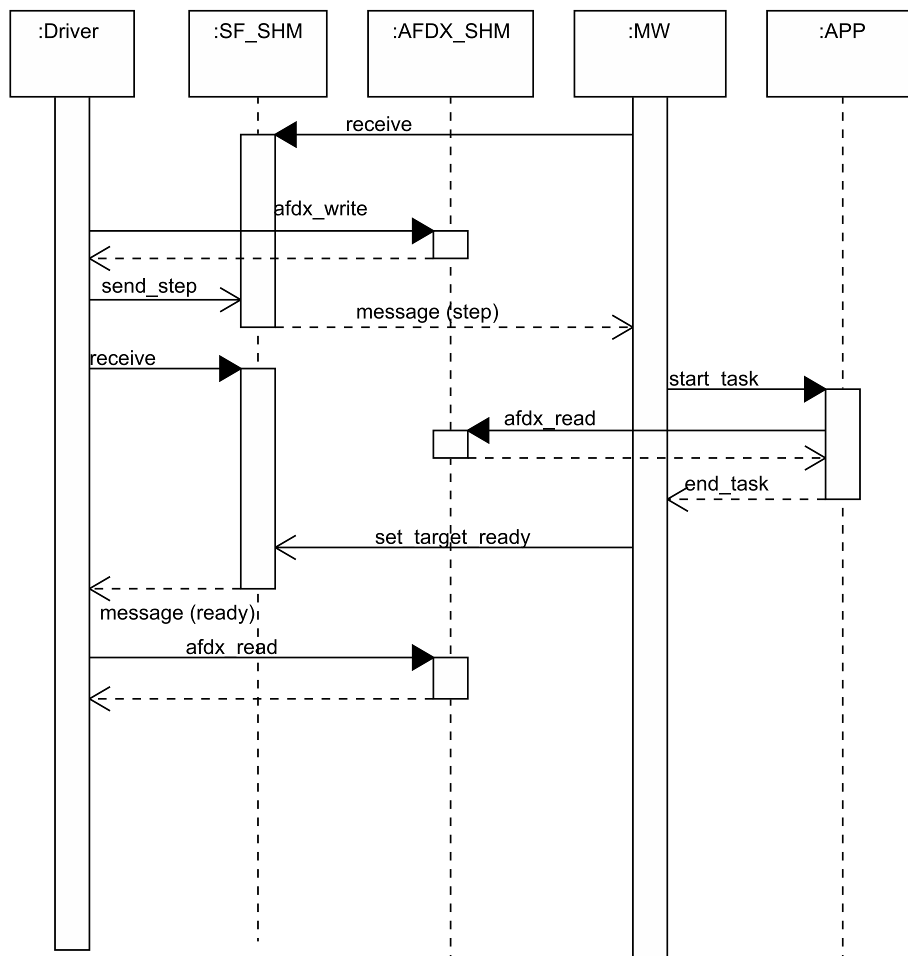


Figura 3.4: Diagramma di sequenza dell'esecuzione sincrona di un tipico step

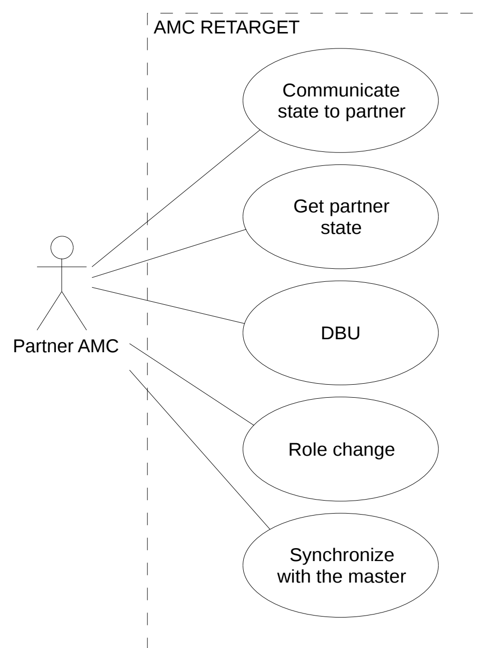


Figura 3.5: Diagramma dei casi d'uso relativo alla ridondanza

messaggio contiene tra le altre cose il ruolo, il numero di sequenza del minor frame, e lo stato della DBU. Questa operazione è svolta a livello di Middleware ed è trasparente per il codice applicativo, cui tuttavia sono offerte delle funzioni per leggere l'ultimo valore di tali dati.

3.2.2.2 Ricezione stato del partner

Il Retarget riceve periodicamente informazioni sullo stato del partner (il converso del punto precedente). La mancata ricezione delle informazioni sullo stato è considerata segno di guasto, e il partner è dichiarato morto dopo un timeout fissato. A questo punto l'AMC rimanente assume incondizionatamente il ruolo di master.

3.2.2.3 Cambio di ruolo

Il Master può comunicare la propria intenzione di cedere il ruolo allo Standby, che deve assumerlo immediatamente (cioè entro il frame successivo). La richiesta arriva dall'utente al Middleware attraverso il codice applicativo.

3.2.2.4 Sincronizzazione col Master e negoziazione del ruolo

All'avvio, le due istanze dell'AMC nascono con un ruolo indefinito. Se le due istanze sono avviate contemporaneamente – entro un margine – una costante cablata nel Middleware determina quale delle due assumerà il ruolo di master. Se le due istanze sono avviate ad una certa distanza nel tempo, la prima avviata sarà il Master. Nell'AMS reale questa negoziazione è implementata in hardware con un collegamento dedicato.

Sempre all'avvio, lo Standby deve sincronizzare il suo *frame counter* con quello del Master. Nel caso del modello di AMC considerato, lo Standby ripete il frame 1 finché il Master non inizia il major frame successivo. Il Middleware dello Standby interviene in caso di disallineamento dei frame, con aggiustamenti minimi del timer dello scheduler. Nel caso del simulatore questo requisito non è stringente, dato che la sincronia è data da un segnale esterno, comune alle due istanze.

3.2.2.5 Database Update

La DBU è una funzione del Middleware con lo scopo di allineare i dati tra le due istanze. Il trasferimento è pilotato dallo strato superiore (il codice applicativo) come richiesta di invio di un buffer opaco, di cui al Middleware sono dati solo l'indirizzo e la dimensione. La dimensione del buffer inviato è limitata, pertanto l'applicativo è incoraggiato a suddividere i dati in segmenti appropriati. La direzione è sempre da Master a Standby, che devono chiamare rispettivamente la funzione di invio e quella di ricezione nello stesso momento – con un adeguato margine – con gli stessi parametri. Gli identificativi dei due task coinvolti devono essere uguali. In caso di discrepanza tra numero del task o dimensione del buffer il trasferimento è rifiutato. La comunicazione sfrutta un collegamento dedicato e gli errori di trasmissione vengono gestiti e risolti dal Middleware. La figura 3.6 mostra la sequenza tipica delle operazioni.

3.3 Obiettivi e scelte progettuali

In questo paragrafo sono raccolte le scelte che hanno determinato il design dell'intero progetto o di alcune sue parti importanti, con le loro motivazioni e conseguenze.

Gli obiettivi che sono stati tenuti maggiormente in considerazione sono i seguenti.

- Mantenere il più possibile inalterato il codice applicativo,

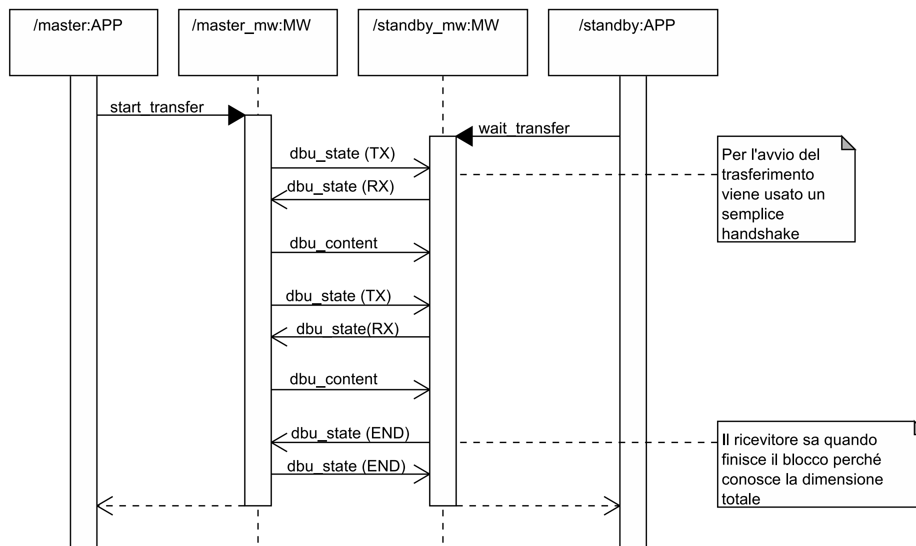


Figura 3.6: Diagramma di sequenza della procedura di Database Update.

- non influenzare significativamente le prestazioni del sistema,
- riutilizzare il codice esistente,
- dare una struttura modulare anche dove in precedenza non lo era,
- unificare il più possibile la configurazione per il test system e quella per il simulatore,
- fare in modo che il codice sia portabile, cioè utilizzabile per sistemi operativi diversi.

3.3.1 Riutilizzo e adattamenti del codice del VTE

Per la prima bozza del Retarget si è deciso di riciclare per intero il codice del VTE, in ragione del fatto che nonostante i suoi limiti era un prodotto già ampiamente testato, sebbene sotto una piattaforma diversa; abbiamo stimato che reimplementare da zero tutti i componenti del Middleware avrebbe richiesto uno sforzo maggiore piuttosto che partire dal codice del VTE e adattarlo al nuovo ambiente. La previsione si è dimostrata sostanzialmente corretta, benché lo scorporo della parte di protocollo e lo spostamento delle interfacce di rete simulate nella memoria condivisa abbia richiesto la riscrittura di un'ampia parte del Middleware. Si è scelto infatti di non utilizzare per il simulatore lo stesso protocollo di rete del VTE, ritenendolo non

adeguato in termini di prestazioni (è un protocollo domanda/risposta totalmente sincrono) e intrinsecamente non sicuro: l'input output è realizzato come letture/scritture su indirizzi di memoria grezzi. Inoltre si è preferito separare la parte di protocollo in vista di un suo cambiamento non retrocompatibile o abbandono in favore di un protocollo diverso. In questo modo i cambiamenti sarebbero assorbiti dal VTE Proxy, che rimane comune tra i diversi modelli di Middleware. All'opposto, la separazione si rende utile per gli eventuali retarget di apparati con un Middleware completamente diverso, per i quali non si dovrà reimplementare il protocollo di rete.

3.3.2 Linguaggi di programmazione

Posto che il software da cui partivamo era scritto parte in Ada 95 e parte in C, mentre il linguaggio del Framework di simulazione è il C++, per le parti che dovevano essere sviluppate da zero ho deciso di utilizzare lo stesso C++, poiché non richiedeva l'installazione di strumenti aggiuntivi, è compatibile col C, con prestazioni paragonabili ma molto più flessibile e con la comodità delle librerie standard. Inoltre mi ha permesso di utilizzare le librerie Boost (di pubblico dominio) per quanto riguarda la sincronizzazione e la comunicazione interprocesso. Per le API delle librerie condivise ho preferito utilizzare il C, sia per interfacciarle – almeno inizialmente – alla parte C del Middleware, sia per mantenere l'interoperabilità tra oggetti creati con compilatori diversi (in quanto non esiste una ABI standard per il C++ sotto Windows).

I moduli del Middleware, nel corso del loro adattamento alla nuova architettura, sono stati progressivamente e col minimo sforzo portati dal C al C++, per comodità di scrittura – una valutazione soggettiva.

3.3.3 Scelta del compilatore

La scelta del compilatore è stata una scelta quasi obbligata: non avendo il cliente a disposizione una versione per Windows dello stesso compilatore Ada che si usa per l'architettura AMC reale – nonostante questa versione esista, si tratta sempre di problemi di costo – non avendo necessità di certificare per il volo il software compilato, abbiamo ripiegato sul compilatore open-source GNAT, che fa parte della ben nota suite GCC della Free Software Foundation ed era già stato utilizzato nell'azienda per altri progetti, con successo.

Ciò ha richiesto delle correzioni puntuali – poche, fortunatamente – ad alcuni costrutti che non sono supportati da entrambi i compilatori e che causavano errori di compilazione, e una riformattazione del codice, con uno

strumento automatico, in modo da avere una sola unità di compilazione per file e nomi di file che rispettano una determinata convenzione (restrizioni che solo GNAT impone). Alcune delle correzioni puntuali, ad esempio, erano dovute alla differente implementazione dei tipi standard (gli intervalli di tempo ad esempio) o al diverso allineamento standard dei dati. In generale GNAT si è dimostrato più restrittivo rispetto ad alcuni vincoli che nello standard Ada sono opzionali.

Per quanto riguarda la parte in C/C++, abbiamo fatto in modo che fosse compilabile sia con GCC che con Microsoft Visual C++, il compilatore utilizzato nel Framework e che le librerie dinamiche compilate nei due modi fossero perfettamente sostituibili.

3.3.4 Approccio alla correzione dell'endianness

Il codice applicativo dell'AMC *non* è stato scritto con l'obiettivo della portabilità. Di conseguenza nel retarget ci si aspetta tutta una serie di incompatibilità (si veda § 2.3 a pagina 31), soprattutto per ingressi e uscite, e in rari casi per i dati interni. Abbiamo deciso di considerare questi ultimi come "errori di programmazione", sebbene sull'AMC vero non diano problemi, in quanto frutto di cattive pratiche e spesso di violazioni del *coding standard* – ad esempio ottenere la parte alta di un intero attraverso il suo indirizzo e non con le funzioni fornite dal linguaggio. In questi casi si è proceduto alla correzione diretta del codice applicativo e alla segnalazione al team di sviluppo *on-board* per l'integrazione della correzione nel codice originale.

Per quanto riguarda le strutture in ingresso e uscita, dato il loro numero e la loro distribuzione imprevedibile all'interno del codice, la correzione manuale non è stata possibile, per diverse ragioni. Non è facile capire, all'interno di un codice di circa un milione di righe, quali sono le strutture collegate all'I/O – sulle quali applicare una trasformazione come suggerito in [Andress, 2005] – poiché oltre a quelle direttamente utilizzate nell'I/O ve ne sono alcune che vengono in modo "non pulito" convertite in dati opachi che vanno poi a riempire dei campi dei pacchetti in uscita. Non esiste un metodo automatizzato per ricostruire questo tipo di flusso di dati.

Si è ovviato a questo problema intervenendo direttamente sui messaggi in ingresso e in uscita, evitando di modificare il codice dell'applicativo AMC originale. Ciò è stato facilitato dal fatto che in avionica i messaggi sono quasi sempre a struttura fissa, formalizzata attraverso l'ICD (§ 2.2.2.2 a pagina 30). Tuttavia, poiché le conseguenze del cambio di endianness dipendono dall'implementazione della lettura e della scrittura dei dati, si sono fatte delle assunzioni basate su un veloce esame del codice applicativo.

Si è lasciato spazio ad una configurazione manuale per i casi in cui queste assunzioni non sono confermate.

3.3.5 Sincronizzazione, comunicazione interprocesso e funzioni speciali

Per la comunicazione tra i processi che fanno parte del "sistema retarget" si è optato per un'interfaccia a memoria condivisa, che ha il vantaggio di essere molto veloce, di poter collegare un numero indefinito di processi, di contenere le informazioni sul suo stato ed essere di conseguenza ispezionabile da processi terzi. L'unico svantaggio importante è che costringe i processi che la utilizzano a risiedere nella stessa macchina. Nel nostro caso non si sono posti problemi, dato che il sistema ha già un'interfaccia remota che è alternativamente il VTE Proxy o il Gateway HLA, i quali, occupandosi solo del trasporto dei dati, sono adatti a condividere le risorse della macchina col processo dell'AMC.

Per quanto riguarda le funzioni speciali, che includono anche la sincronizzazione dei task, tra i vari modelli possibili ho preferito quello della "coda di messaggi" – bidirezionale in questo caso – in modo che i comandi siano eseguiti nell'ordine in cui sono ricevuti e il Target sia in grado di sospendersi fino all'arrivo del successivo comando. Una *condition variable* segnala al Target o al Driver l'arrivo o la lettura di un messaggio.

Un'alternativa più semplice alla coda di messaggi sarebbe una struttura in memoria condivisa con un flag per ogni possibile comando, che indica se il comando è "in corso" e un campo per ogni possibile parametro; un'altra porzione della struttura indicherebbe lo stato di esecuzione del comando da parte del target come *acknowledgement*. La *condition variable* svolgerebbe lo stesso ruolo che ha nella coda di messaggi. Una tale implementazione avrebbe lo svantaggio di richiedere una modifica della struttura ogni volta che si intendesse aggiungere una funzione speciale (l'insieme delle funzioni speciali, determinato dalle richieste del cliente, si è dimostrato molto instabile nel corso dello sviluppo) e non permetterebbe di dare due volte lo stesso comando nello stesso turno – eventualità ad ogni modo difficile, data la natura di questi comandi, che indicano variazioni di parametri per cui il secondo comando sovrascriverebbe il primo, con l'eccezione di *snapshot* e *recall*, che possono comunque essere gestiti sia come coda che come variabili di stato.

Si noti che nonostante il Driver e il Target vivano in due processi differenti essi eseguono le loro operazioni in modo serializzato, ossia il Driver si sospende finché il Target non ha terminato l'esecuzione dei task (figura

3.4). Ciò è stato deciso per evitare che l'esecuzione dei task si sovrapponga con l'I/O da parte del Driver e dia dei risultati spurii, sebbene l'esecuzione parallela dei due processi sia tecnicamente possibile.

Capitolo 4

Dettaglio dei componenti

In questo capitolo sono descritti i singoli componenti nel loro dettaglio.

4.1 Middleware

Questo componente del Retarget è la reimplementazione del Middleware originale, basata sulla versione del VTE, che espone al codice applicativo le stesse interfacce in Ada 95 della versione originale, e dall'altra parte comunica con le interfacce di comunicazione simulate attraverso la memoria condivisa.

Come già descritto nelle sezioni precedenti, il Middleware si compone di due parti, una in Ada e una in C (successivamente portata al C++), realizzate come librerie dinamiche separate, per motivi più che altro storici. La figura 4.1 rappresenta la struttura interna, che è sostanzialmente la stessa del VTE per la parte Ada ed è stata invece ricostruita completamente per la parte C, che nel VTE era un componente quasi monolitico.

Inizialmente, negli anni 90 nel VTE esisteva solo la parte in C del Middleware, chiamata direttamente dal software applicativo che all'epoca era scritto in Pascal. Col passaggio della piattaforma all'Ada 95, per non alterare il codice C già funzionante, era stato creato lo strato in Ada come proxy: i parametri delle procedure venivano convertiti dal nuovo formato a quello vecchio (l'opposto per i parametri in uscita), sfruttando il fatto che differivano solo minimamente. Per il Retarget si è deciso di mantenere la distinzione tra le due parti solo per non dover riscrivere ulteriore codice, nonostante questa divisione avesse ormai perso il suo significato. Per i le funzionalità mancanti, implementate per il Retarget, si è scelto di volta in volta il linguaggio più appropriato, preferendo l'Ada per le funzioni più spe-

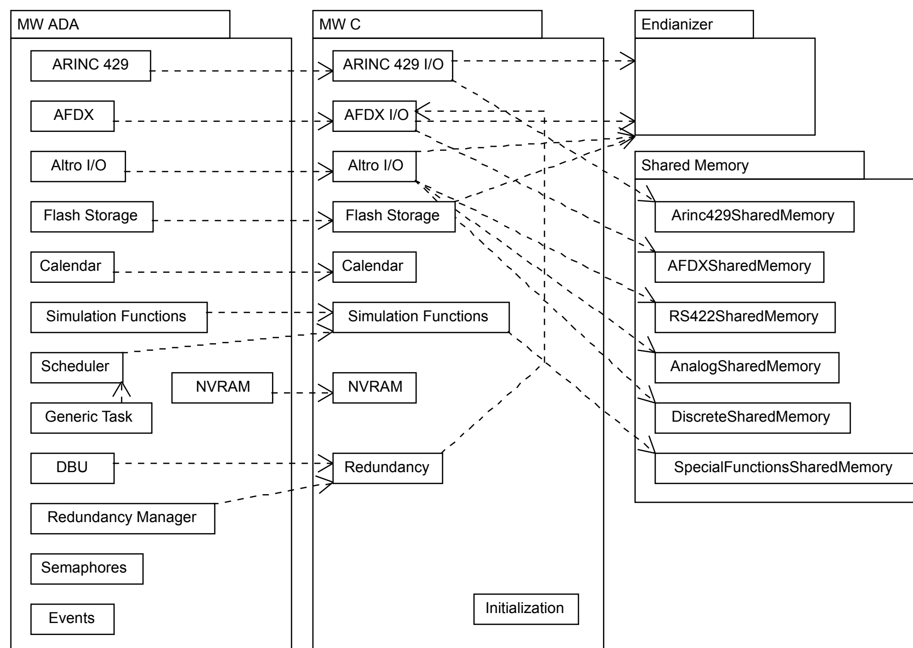


Figura 4.1: Diagramma dei componenti principali del Middleware

cifiche di questo modello di AMC e il C++ per quelle che paiono essere più stabili nel tempo e tra i modelli.

Come si può notare dalla figura 4.1, per i motivi suddetti, molti moduli del Middleware della parte Ada sono solo dei proxy verso i loro analoghi nella parte C++; ciò vale per tutti i moduli di I/O. Nella riscrittura si è cercato di ridurre i passaggi intermedi inutili. Allo stesso tempo si è cercato di minimizzare l'utilizzo delle strutture dati nell'interfaccia tra i due linguaggi, allo scopo di evitare problemi di compatibilità (allineamento) e di manutenzione. Alcune funzionalità erano e sono rimaste implementate direttamente in Ada, soprattutto lo scheduling e le funzioni relative alla sincronizzazione (semafori, eventi, code).

4.1.1 Input/Output

I moduli di I/O forniscono al codice applicativo le funzioni per la gestione delle interfacce di comunicazione: configurazione, invio, ricezione, controllo dello stato. Ogni interfaccia ha il suo protocollo, coi suoi parametri e le sue particolarità, ragion per cui non è possibile a questo basso livello una gestione unificata. Come è consuetudine nell'ambito avionico, la configurazione è statica e caricata all'avvio da tabelle in qualche formato – nel

nostro caso in formato binario e cablate nel codice. Le tabelle contengono le informazioni necessarie a indirizzare e codificare/decodificare i messaggi, insieme a informazioni sullo strato fisico che per il retarget non hanno nessun significato.

Dall'altro lato i moduli di I/O comunicano con le memorie condivise, che oltre allo stato del bus contengono la maggior parte della logica del protocollo; il Middleware si occupa solo della gestione della configurazione e del trasporto dei dati, con le opportune conversioni, come da configurazione.

Per realizzare i collegamenti di *cross-talk* tra un'istanza dell'AMC e l'altra si è aggiunta un'opzione al file di configurazione del retarget, che viene caricato dal Middleware all'avvio, per specificare quali VL debbano essere rediretti all'altra istanza. In questo caso i messaggi in uscita, oltre ad essere scritti sulla propria memoria condivisa vengono scritti anche in quella del partner (in ingresso). È inoltre possibile redirigere alcune connessioni (identificate dal loro VL e dalla porta UDP, nel caso dell'AFDX) via UDP anziché attraverso le memorie condivise. Questa funzionalità è stata aggiunta per poter collegare direttamente alcuni emulatori dei display, che già sfruttavano questo protocollo, oppure per realizzare il *cross-talk* qualora si intendesse spostare le due istanze dell'AMC su due macchine separate.

In generale, in tutto il Middleware, si sono implementati solo i metodi utilizzati dal codice applicativo, ignorando gli altri. È opportuno sottolineare che ai fini del simulatore non è necessario riprodurre nel dettaglio le funzionalità di controllo dello stato di salute delle interfacce, in quanto simulare tutte le centinaia di tipologie di guasti possibili non è fra i nostri scopi. Pertanto ci si è limitati a restituire sempre lo stato "buono"; al contrario, ove dei malfunzionamenti erano richiesti, si sono inseriti dei *flag* attivabili dall'esterno – tramite l'interfaccia delle funzioni speciali – per simularli.

Tra i moduli di I/O ve ne sono ulteriori due che rappresentano la memoria di massa dell'apparato, ossia la NVRAM () e una memoria flash. La NVRAM è implementata nel retarget come un *memory-mapped file*, per ottenere una forma di persistenza che viene gestita automaticamente dal sistema operativo. Una volta allocata, il Middleware deve solo dare al codice applicativo l'indirizzo di quest'area di memoria.

La memoria flash ha una struttura molto semplice, suddivisa in segmenti la cui lunghezza è configurata dal codice applicativo e non può essere modificata. Al suo interno non è previsto nessun file system. È utilizzata per il database del FMS e per l'elenco dei componenti opzionali installati sull'elicottero e per altri dati costanti nel tempo. Nel retarget viene salvato un file per ogni segmento, per compatibilità con il VTE. In questo modo l'operatore, tra un'esecuzione e l'altra, può sostituire i segmenti per riprodurre

configurazioni differenti.

La correzione dell'endianness si inserisce subito prima della trasmissione e subito dopo la ricezione sulle schede di rete simulate. I segmenti della flash sono "endianizzati" allo stesso modo; in questo caso il messaggio da correggere corrisponde con l'intero file.

Rispetto al VTE, sono stati reimplementati quasi completamente tutti i moduli di I/O ad eccezione della memoria flash, dopo aver valutato che per l'adattamento al nuovo backend con le memorie condivise avrebbe richiesto più sforzo l'adattamento del vecchio codice. In tal modo il volume di questa parte è diminuito drasticamente, spostando la logica dei protocolli nei moduli delle memorie condivise e rimuovendo tutte le conversioni di formato inutili e le funzionalità non più supportate che si erano accumulate con gli anni.

4.1.2 Scheduler

Lo scheduler ha il compito di organizzare l'esecuzione dei task, basandosi su un segnale di sincronia esterno che indica l'inizio di un frame. Secondo le specifiche del Middleware, ogni task è l'istanza di un package generico[1] con due procedure come parametri: `TASK_INIT` per l'inizializzazione e `TASK_JOB` che è la procedura che viene eseguita periodicamente. Per motivi di riutilizzo del codice, nonostante fossero possibili diverse ottimizzazioni, è stata mantenuta quasi per intero l'implementazione del VTE, secondo la quale ogni task deterministico è definito come un task Ada (cioè un thread) ed è l'istanza di un task generico. È pertanto utilizzato il meccanismo di sincronizzazione dei *rendez-vous* Ada, sebbene l'esecuzione dei task non sia parallela (figura 4.2):

- il thread principale (Main) crea i task e successivamente avvia lo scheduler
- lo scheduler inizializza tutti i task e si mette in attesa del comando di step
- all'arrivo del comando di step lo scheduler aggiorna il contatore del minor frame e avvia il primo task deterministico della tabella e si mette in attesa del suo termine

[1] In Ada 95 un package generico equivale ad una *template class* in C++, i cui parametri possono essere dei tipi o delle procedure. Le specializzazioni di questi generici sono dette "istanze".

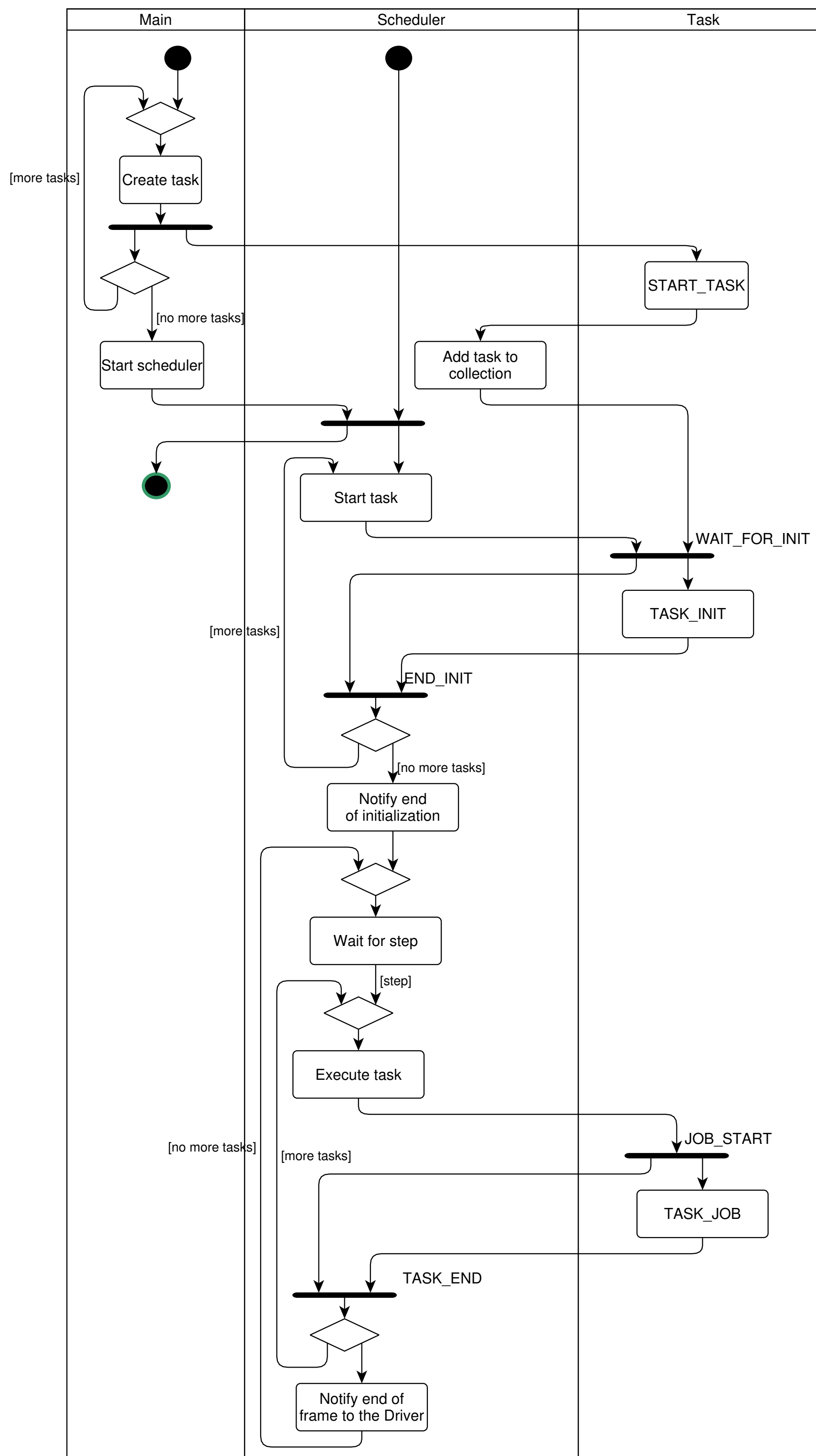


Figura 4.2: Diagramma di attività dello scheduler, con le fasi di inizializzazione e di esecuzione. Ogni corsia rappresenta un thread. I task sono multipli ma sono mostrati come un unico thread; la loro esecuzione non si sovrappone. I punti di sincronia sono le entry *dei* rendez-vous.

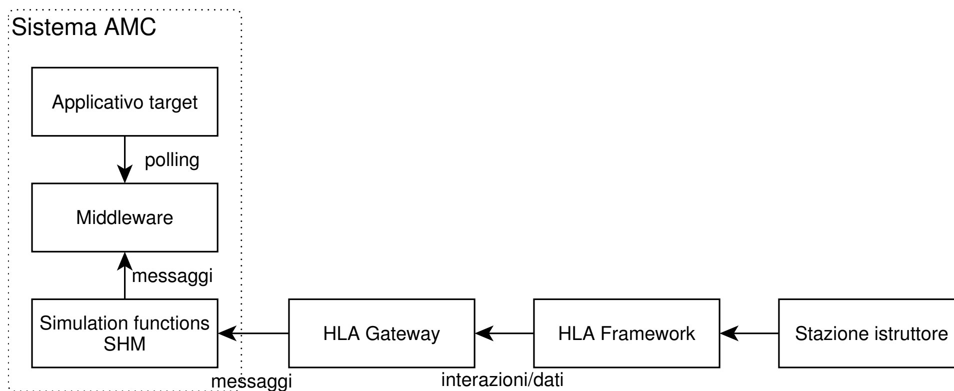


Figura 4.3: Realizzazione delle simulation functions*. La comunicazione fluisce dal Framework fino al codice applicativo, in parte sotto forma di messaggi e in parte sotto forma di variabili di stato.*

- al termine del task lo scheduler avvia il task deterministico successivo, se ci sono altri task da avviare, altrimenti notifica la fine del frame al Driver e si mette in attesa del successivo comando di step.

Per quanto riguarda i task statistici, essi sono avviati all'avvio dello scheduler e da quel momento si autogestiscono, ripetendo la propria `TASK_JOB` all'infinito.

L'unico cambiamento significativo rispetto al VTE è l'aggiunta di un punto di sincronia alla fine del frame. Infatti nel VTE l'esecuzione dei task e la ricezione dei comandi dal Driver erano compiute da due thread diversi, sincronizzati solo all'inizio del frame ma non alla fine, col rischio di restituire il controllo al Driver prima che tutti i task avessero terminato l'esecuzione e conseguente sovrapposizione di input/output.

4.1.3 Funzioni speciali

Le funzioni speciali dell'ARINC 610B inizialmente previste sono la *failure injection*, diversi tipi di *freeze* e variazione parametri in due forme, *slew* (variazioni progressive) e *set* (variazioni istantanee). L'effettiva esecuzione di queste richieste è delegata al codice applicativo, che sarà appositamente alterato, non prevedendo queste funzionalità all'inizio del progetto. Compito del Middleware e dell'interfaccia a memoria condivisa è consentire il trasporto delle informazioni dal Framework di simulazione fino al codice applicativo. Dal lato del codice applicativo abbiamo previsto un'interfaccia che conserva le informazioni sulle funzioni speciali come variabili di stato, che possa essere interrogata periodicamente dall'applicativo. È stata scelta questa modalità di comunicazione anziché una a eventi (com'è invece per le

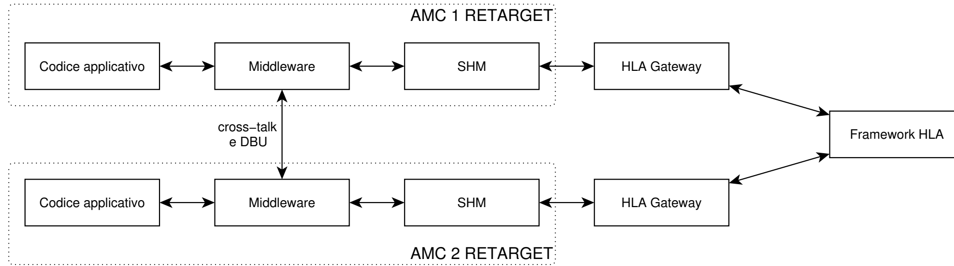


Figura 4.4: Le due istanze dell'AMC sono guidate ognuna dal suo Driver (il Gateway HLA). La comunicazione tra le due istanze avviene a livello di Middleware.

funzioni analoghe di altri simulatori) poiché si adatta meglio alla periodicità dell'applicativo, all'interno del quale il *polling* è prassi.

I parametri che sono soggetti a variazione o *freeze* (quali la velocità all'aria, la pressione, l'altitudine) sono indirizzati attraverso un numero identificativo; in questo modo è possibile aggiungere nuovi parametri senza alterare il protocollo di comunicazione (la memoria condivisa). L'interfaccia verso l'applicativo fornisce le seguenti informazioni, per ognuno dei parametri che sono soggetti a *freeze* o variazione:

Comando	Campi
freeze	id. parametro, stato attivo/inattivo (booleano)
slew/set	id. parametro, stato attivo/inattivo (booleano), nuovo valore (virgola mobile)

Dal lato delle memorie condivise la comunicazione è realizzata attraverso code di messaggi, in quanto permette lo scambio di informazioni senza che la memoria condivisa ne conosca il formato, mentre la sincronizzazione della coda evita la perdita dei messaggi. Ogni messaggio corrisponde alla variazione dello stato di un parametro, e contiene pertanto gli stessi argomenti della tabella sopra.

Lo stesso canale di comunicazione è utilizzato per altri comandi quali *step* e *reset*, che comandano rispettivamente l'esecuzione di un frame e lo spegnimento.

4.1.4 Ridondanza e allineamento dati (DBU)

La ridondanza e la sincronizzazione sono gestite da un apposito modulo del Middleware (parte C++), che implementa le funzionalità descritte al paragrafo 3.2.2, progettato e scritto da zero per questo lavoro. Per la comunicazione ci siamo ispirati all'apparato reale: le due istanze sono direttamente collegate da un collegamento Ethernet dedicato e inviano al partner le informazioni sullo stato, ad ogni frame. Nel retarget abbiamo riutilizzato

l'infrastruttura AFDX del Middleware, che già prevede la redirezione dei VL di *cross-talk* direttamente sulla memoria condivisa del partner o via UDP. Abbiamo quindi aggiunto un VL fittizio per queste informazioni di basso livello e per i dati della DBU (*database update*). È definito come VL bidirezionale[2], identificato dal numero 0 – valore non utilizzabile dall'applicativo – con due porte:

REDUNDANCY_DATA (*sampling*) per i dati sullo stato, trasmessi periodicamente;

DBU_DATA (*queuing*) per i dati della DBU e relative comunicazioni di servizio.

I dati sono scambiati dal Middleware all'inizio di ogni frame, nello stesso thread dello scheduler, prima dell'esecuzione dei task. Tutte le comunicazioni sono non bloccanti e *best-effort*; il sistema deve tollerare la mancata ricezione per un numero configurabile di turni consecutivi.

4.1.4.1 Arbitraggio del ruolo e sincronizzazione

Alla fine dell'inizializzazione (il cosiddetto frame 0) e all'inizio di ogni frame, prima dell'esecuzione dei task, le due istanze si scambiano attraverso la porta `REDUNDANCY_DATA` una struttura che tra le altre cose contiene i seguenti campi (il loro contenuto è riferito allo stato del mittente).

[2] Lo standard non permette VL bidirezionali ma la nostra implementazione permette di configurare porte con lo stesso VL e numero ma direzioni opposte e le tratta come due porte distinte.

Nome	Tipo	Descrizione
`role`	enumerato (`MASTER`, `STANDBY`)	ruolo corrente
`current_minor`	intero (0..64)	numero del *minor frame* attuale (oppure 0 se lo scheduler non è stato ancora avviato o non è sincronizzato)
`current_tst`	intero	identificatore della tabella dei task corrente
`dbu_state`	`enum DBUState`	stato dell'automa della DBU (v. sotto)
`dbu_task`	intero	identificatore del task che sta richiedendo la DBU, se una richiesta è in corso
`dbu_size`	intero	dimensione del segmento da trasferire, se una richiesta di DBU è in corso

La ricezione di questi messaggi è sempre non bloccante per il Master e ha un timeout di circa un decimo della durata del frame (valore scelto empiricamente) per lo Standby. Questa asimmetria è stata inserita per ridurre la perdita di messaggi da parte dello Standby, il cui scheduler (quindi il minor frame) deve essere sincronzizato con quello del Master. Se un messaggio è perso, lo Standby continua l'esecuzione del frame sotto l'ipotesi che il Master non abbia cambiato stato. La ricezione dei dati del partner precede sempre l'invio dei propri.

L'automa della ridondanza obbedisce alle seguenti regole:

- Se non si ricevono messaggi dal partner dopo un numero di frame fissato, il partner è dichiarato morto.
- Se il partner è morto, l'istanza rimanente assume immediatamente il ruolo di master.
- Una delle due istanze è contrassegnata come "master predefinito", tipicamente quella sul lato del pilota.
- All'avvio il ruolo è sempre impostato a `MASTER`.
- Se il proprio ruolo e quello del partner sono entrambi `MASTER` ed entrambi sono stati appena avviati (`current_minor` = 0), l'istanza

non contrassegnata come “master predefinito” passa immediatamente a standby.

- Se il proprio ruolo e quello del partner sono entrambi `MASTER` e il partner è stato avviato precedentemente (`current_minor` $\neq 0$), l'istanza corrente passa immediatamente a standby.
- Se il proprio ruolo e quello del partner sono entrambi `STANDBY`, significa che il partner ha ceduto il ruolo di master e l'istanza corrente deve immediatamente assumerne il ruolo.
- Se il proprio ruolo è STANDBY e lo scheduler non è ancora sincronizzato (`current_minor` $= 0$), il contatore dei frame non viene incrementato finché quello del partner non vale 1. Un valore di 0 per current_minor viene interpretato dallo scheduler come “minor frame 1”. Ciò realizza il requisito di sincronia “lo Standby ripete il primo frame finché il numero di frame non è sincronizzato con quello del Master”.
- Se il proprio ruolo è `STANDBY`, la porta `REDUNDANCY_DATA` è aggiornata, lo scheduler ha raggiunto la sincronia (`current_minor` $\neq 0$) e dopo aver incrementato `current_minor` il proprio valore differisce da quello del partner, la sincronia è persa. Per il retarget si è deciso che tale condizione non abbia nessuna conseguenza, in quanto uno scarto di qualche minor frame è stato considerato un'approssimazione accettabile della sincronia e inoltre nel simulatore il segnale di sincronia proviene dal Framework ed è comune a entrambe le istanze dell'AMC.

In figura la figura 4.5 è rappresentato l'automa a stati finiti. I nomi degli stati uniscono il ruolo (`MASTER`, `STANDBY`) e lo stato di sincronizzazione: `INIT` corrisponde a `current minor` $= 0$, altrimenti `SYNC`. Di seguito la descrizione delle transizioni.

`START_SCHEDULER` Corrisponde alla chiamata di funzione da parte del codice applicativo che avvia lo scheduler, alla fine dell'inizializzazione.

`PARTNER_LOST` Scatta quando per un numero fissato di turni non viene ricevuto nessun messaggio dal partner.

`PARTNER_*` Ricezione dello stato dal partner, indicato nel nome dell'evento.

È opportuno sottolineare che funzionalità relative alla ridondanza sono implementate anche nel codice applicativo, parzialmente replicando quelle del

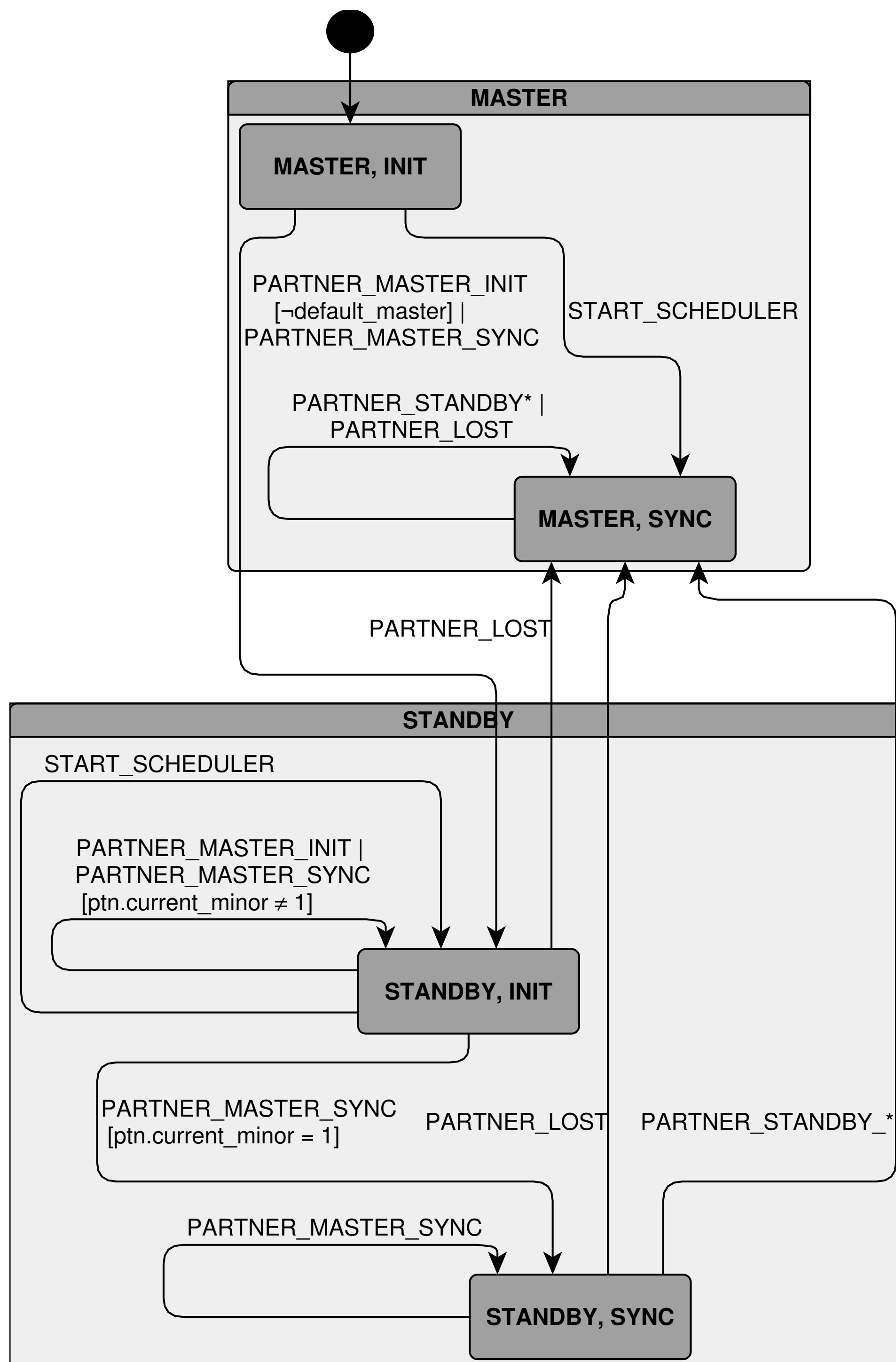


Figura 4.5: Diagramma degli stati dell'automa della ridondanza (rappresentazione UML statechart). La barra verticale indica l'alternativa tra due condizioni; l'asterisco sostituisce una sequenza di caratteri qualsiasi; tra

Middleware. Per esempio, le due istanze possono conoscere lo stato dell'alimentazione e il frame attuale del partner attraverso il Middleware, ma possono ottenere la stessa informazione attraverso il virtual link del *cross-talk* gestito dall'applicativo, con il quale si scambiano informazioni sullo stato – oltre ai principali dati provenienti da sensori e calcolati quali la posizione.

4.1.4.2 Allineamento dati (DBU)

La funzione di DBU appartiene anch'essa al thread dello scheduler. L'invio dei dati è razionato in modo da simulare approssimativamente la velocità dell'operazione di DBU sull'apparato reale (circa 10 secondi per 3 MB). L'operazione di DBU a livello del Middleware Ada è sincrona e bloccante – è concepita per essere chiamata all'interno di un task statistico, come tutte le operazioni lunghe di lettura/scrittura – pertanto è stato inserito un meccanismo di sincronizzazione col livello più basso, che utilizza un callback dal lato C++ e un semplice semaforo sul lato Ada (figura 4.6).

La gestione del protocollo della DBU è implementata come un automa a stati finiti (figura 4.7), che corrisponde alla funzione `dbu_state_machine()` della figura 4.6, chiamata all'inizio di ogni minor frame, dopo la ricezione del messaggio `REDUNDANCY_DATA` dal partner e prima della spedizione del proprio.

Il protocollo implementato è una semplice variante del Go-Back-N (finestra di trasmissione infinita) con un meccanismo di *handshake* che sfrutta il messaggio di stato `REDUNDANCY_DATA` scambiato ad ogni frame, che rende visibile lo stato al partner – il codice applicativo già sfrutta un meccanismo simile di comunicazione del proprio stato attraverso il VL di *cross-talk*.

L'enumerato `DBUState` può assumere i seguenti valori: `IDLE`, `SENDING`, `RECEIVING`, `CANCELLING`, `COMPLETED`. Lo stato attuale è visibile al partner attraverso il messaggio `REDUNDANCY_DATA`. Gli stati `SENDING` e `RECEIVING` hanno inoltre tre sottostati, che nella figura sono chiamati `WAIT`, `TRANSFER` e `CLOSING` e non sono visibili dal partner.

`IDLE` Nessuna operazione in corso. È lo stato iniziale.

`SENDING` L'istanza ha aperto una richiesta di invio o sta inviando dati.

`RECEIVING` L'istanza ha aperto una richiesta di ricezione o sta ricevendo dati.

`COMPLETED` L'istanza ha finito di *ricevere* dati e attende che il Master chiuda l'operazione.

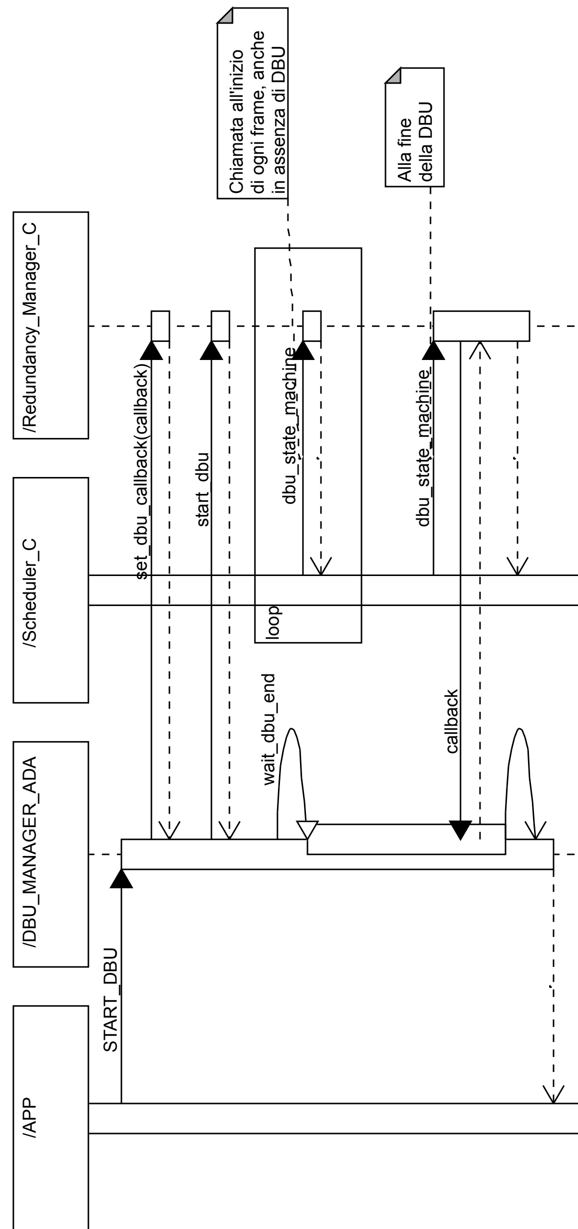


Figura 4.6: Sincronizzazione dell'operazione di DBU tra livello Ada e livello C++. Da sinistra a destra: codice applicativo, classe DBU_MANAGER del lato Ada del MW, moduli Scheduler e Redundancy della parte C++.

Figura 4.7: Diagramma degli stati dell'automa della DBU (rappresentazione UML Statechart). L'etichetta dopo la barra (/) indica l'esito del trasferimento. L'asterisco indica tutti gli eventi che non sono già nominati in uscita da uno stato.

CANCELLING L'istanza ha ricevuto dall'applicativo una richiesta di annullamento e attende che anche il partner annulli il trasferimento.

WAIT L'istanza sta attendendo che anche il partner apra la corrispondente richiesta di trasferimento.

TRANSFER Il trasferimento è in corso. Ad ogni turno viene trasferita una porzione dei dati e sono rispettate le richieste di ripetizione.

CLOSING Il Master ha finito di trasmettere dati ma è ancora aperto alle richieste di ritrasmissione e attende la conferma dallo Standby.

Gli eventi possibili, che causano le transizioni dell'automa, sono le seguenti.

START_DBU Richiesta proveniente dallo strato applicativo di avviare un trasferimento. Il Master la interpreta come una richiesta di invio, di ricezione lo Standby.

PARTNER_* Messaggio che comunica lo stato del partner, come espresso dal nome dell'evento.

TRANSFER_COMPLETE Evento generato internamente quando sono stati ricevuti o inviati tutti i dati.

CANCEL Richiesta di annullamento proveniente dal codice applicativo. Può essere ricevuta in qualsiasi momento.

REWIND_REQUEST Richiesta di ritrasmissione da parte dello Standby (v. sotto).

TIMEOUT Evento generato internamente quando il partner non reagisce per un numero fissato di frame.

MISMATCH Evento generato internamente quando i dati della richiesta (task richiedente e dimensione del segmento) non corrispondono tra le due istanze. Causa l'annullamento dell'operazione.

Tutte le transizioni che portano allo stato `IDLE` causano la fine della trasmissione e l'invio del segnale `callback` alla parte Ada, che può quindi sbloccare la chiamata al servizio di DBU e ritornare. Il segnale callback ha come argomento il risultato del trasferimento, che può essere "successo" o "fallimento" e viene restituito al codice applicativo Ada; il valore "successo" viene restituito solo a seguito di un evento `PARTNER_IDLE` ad eccezione di quello proveniente dallo stato `CANCELLING`. Ogni evento non elencato nel

diagramma causa un annullamento del trasferimento e il ritorno allo stato `IDLE`.

Le ritrasmissioni sono gestite in questo modo: il protocollo è equivalente a un Go-Back-N con una finestra di trasmissione infinita; le ritrasmissioni avvengono solo su richiesta esplicita (un NAK). Ogni pacchetto ha un'intestazione che indica il suo numero progressivo; se lo Standby rileva una discontinuità tra questi numeri invia una richiesta di ritrasmissione (a partire dal pacchetto successivo all'ultimo ricevuto correttamente) al Master, attraverso la porta `DBU_DATA`. La conferma finale della ricezione è effettuata passando allo stato `COMPLETED`, mantenuto finché il Master non riconosce la fine della trasmissione e torna allo stato `IDLE`.

4.1.5 Funzioni di servizio

Tra queste si contano la gestione della data e ora (che possono essere alterate con un'apposita funzione di simulazione e sono salvate su file e caricate all'avvio), l'estrazione delle informazioni dalla tabella dei task, la lettura del file di configurazione caricato all'avvio, la registrazione degli avvisi (logging) a scopo di diagnostica.

4.2 Memoria condivisa

Le interfacce a memoria condivisa simulano ognuna il comportamento di un tipo di bus e ne conservano lo stato. Sono distribuite come librerie condivise separate, la cui implementazione è in C++ ma l'interfaccia esterna in C per garantire compatibilità tra compilatori diversi.

Nella figura 4.8 sono elencate le classi, coi loro attributi e metodi principali. Tutte le interfacce a memoria condivisa derivano dalla stessa interfaccia generica, che si preoccupa della creazione, inizializzazione e distruzione del segmento di memoria e fornisce metodi per sincronizzare l'accesso all'intero segmento, con un mutex.

Per quanto riguarda l'implementazione, quasi tutte le memorie condivise hanno una struttura analoga, cioè delle strutture fisse, inizialmente copiate da quelle del VTE, che già allocano lo spazio per tutti i dati possibili. Fanno eccezione l'eccezione l'AFDX e le funzioni speciali, rispettivamente per l'uso dell'allocazione dinamica e per la sincronizzazione.

Tutti i metodi di lettura hanno un'opzione per non segnare il dato come "letto" (ove previsto), per permettere l'ispezione dei bus da processi terzi che agiscano da *sniffer*.

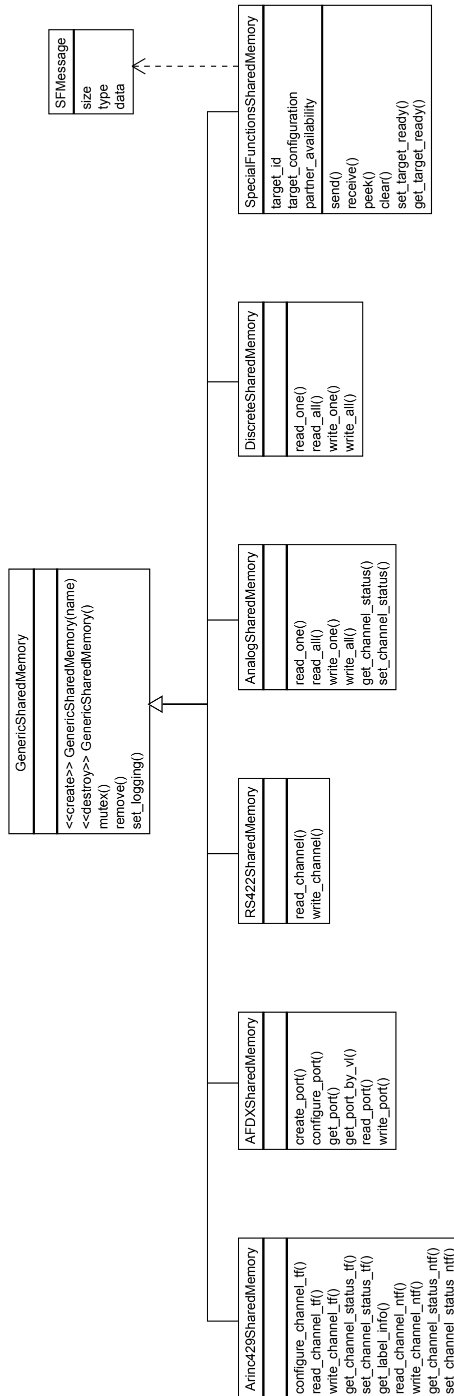


Figura 4.8: Diagramma delle classi di memoria condivisa. Sono mostrati solo operazioni e attributi principali.

4.2.1 `GenericSharedMemory`

Questa classe gestisce la creazione e la distruzione della memoria condivisa. È in verità una classe generica (*template*) avente tra i parametri il tipo `T` della struttura dati da ospitare; più precisamente, le classi figlie derivano da una specializzazione di `GenericSharedMemory`.

Ogni segmento è identificato da un nome formato dall'apparato da simulare (nel nostro caso "AMC1" e "AMC2") e del bus. La dimensione del segmento è nota al momento della compilazione e dipende dalla dimensione di `T`. All'inizio del segmento, `GenericSharedMemory` aggiunge un'intestazione che ha lo scopo di evitare conflitti di versione (quando versioni diverse della libreria cercano di accedere allo stesso segmento) o di dimensione e che contiene un mutex implementato con uno *spinlock* [Gaztañaga, 2012] per sincronizzare l'accesso all'intero segmento. L'inizializzazione è effettuata, nel costruttore, dal primo processo che accede al segmento creandolo e chiama a sua volta il costruttore di `T`; è così garantito che l'inizializzazione avviene una volta sola.

Per la distruzione ci si affida al sistema operativo: il segmento viene distrutto quando viene chiuso da tutti i processi. È comunque disponibile un metodo per eliminarlo esplicitamente.

4.2.2 ARINC 429

Questo modulo simula per intero una scheda ARINC 429 e può operare sia in modalità time-framed (TF) sia in modalità non-time-framed (NTF). I numeri massimi di canali TF e NTF in invio e in ricezione sono fissi e corrispondono al numero di ingressi e di uscite della scheda vera.

Ogni canale ha un attributo che ne indica lo stato: attivo, inattivo o malfunzionante. L'ultimo caso è utilizzato per simulare esplicitamente un problema di trasmissione sulla linea (*link fail*).

4.2.2.1 Time-framed

Le letture e le scritture sono principalmente effettuate canale per canale. I dati (parole) sono indirizzati per direzione (lettura/scrittura), canale, label e SDI e devono essere configurati prima dell'inizio della trasmissione, in modo che la scheda sappia quali label sono da considerarsi "extended", cioè usano i bit del campo SDI per i dati e non per identificare la sorgente. Ciò influisce sul modo in cui una parola sostituisce il suo valore precedente. La configurazione di una parola contiene i seguenti dati: label (da 0 a 377

ottale), SDI (da 0 a 3, oppure “extended”) e numero di bit di dati, da 1 a 19 o 21 se extended.

La lettura di un canale restituisce tutte le parole configurate per quel canale, con un bit di “freschezza” che indica se il dato è stato aggiornato dall’ultima lettura, e segna tutte le parole come lette azzerando il bit suddetto. Il metodo “multi read” è stato aggiunto per permettere con una sola chiamata la lettura selettiva di alcune parole, le cui coordinate (label e SDI) sono fornite come argomento; le altre parole sono lasciate inalterate. Il metodo “delete label” permette al sistema di test di segnare una parola come “mancante” per simulare la sua mancata trasmissione da quel momento in poi.

4.2.2.2 Non-time-framed

I canali NTF si comportano come delle code, con una lunghezza massima fissata. Se una scrittura arriva a riempire completamente la coda, le successive parole sono ignorate. La coda è infatti considerata contemporaneamente coda di lettura da un capo e di scrittura dall’altro.

4.2.3 Discreti e analogici

Nel caso dei discreti e degli analogici ogni canale porta un solo dato, rispettivamente booleano o decimale. Ogni dato è accompagnato dalla sua validità, che indica se il suo cavo è collegato (1) o scollegato/fuori scala (0). Non sono previsti indicatori di freschezza. Per comodità sono disponibili metodi per leggere o scrivere tutti i canali contemporaneamente, che per ragioni di efficienza nel nostro codice applicativo sono preferiti rispetto ai metodi che leggono/scrivono un canale alla volta.

4.2.4 RS 422

RS 422 è un protocollo seriale monodirezionale a byte. La semantica è la stessa dei canali ARINC 429 NTF, con la differenza che in questo caso il trasferimento minimo è di un byte anziché di una parola. Quando la coda è piena i dati in eccesso sono ignorati. Il numero di canali è fissato. Ogni canale ha un indicatore di stato (attivo, inattivo, malfunzionante).

4.2.5 AFDX

La memoria condivisa dell’AFDX è implementata diversamente dalle precedenti in quanto impiega l’allocazione dinamica della memoria all’interno del segmento. La scelta dell’allocazione dinamica è quasi obbligata poiché

le dimensioni dei buffer sono configurabili dall'applicativo e allocare la massima dimensione possibile avrebbe portato ad un segmento troppo grande. Inoltre non era possibile importare direttamente la struttura utilizzata nel VTE in quanto questa già faceva uso dell'allocazione dinamica. A questo scopo si sono utilizzate la libreria Boost.Interprocess e Circular buffer.

Gli oggetti gestiti dall'AFDX sono le *porte*, che possono essere di due tipi, *sampling* o *queuing* a seconda del comportamento del buffer di ricezione. Le porte *sampling* contengono un solo pacchetto che è l'ultimo ricevuto e un indicatore di freschezza; le porte *queuing* si comportano invece come coda. La creazione di una porta avviene in due passaggi: configurazione e creazione. Questa distinzione è stata mantenuta per coerenza con le API originali del Middleware, che sono compatibili con le API dello standard ARINC 653[3].

Al momento della configurazione, per ogni porta sono fornite le seguenti informazioni:

- nome (stringa)
- direzione: TX o RX
- virtual link (intero: 0–65536)
- porta UDP (intero: 0–65536)
- tipo: *sampling* o *queuing.*

`create_port()` restituisce un codice identificativo (*handle*) per la porta, che sarà utilizzato nelle chiamate successive. Al momento dell'apertura della porta sono specificate le informazioni mancanti:

- lunghezza massima dei messaggi
- numero massimo di messaggi in coda.

I metodi per la lettura e la scrittura hanno una sintassi analoga a quella delle funzioni `send()` e `recv()` della libreria standard C, tuttavia `read_port()` aggiunge i parametri `validity` (in uscita) e `flush`, entrambi booleani, che indicano rispettivamente se il messaggio è stato aggiornato dall'ultima lettura e se si intende segnare il messaggio come letto. Dal punto di vista della

[3] ARINC 653 è uno standard per la suddivisione del software avionico in *partizioni* indipendenti, ognuna col suo spazio di memoria virtuale e vincoli temporali. Lo stesso standard definisce delle API per la comunicazione interprocesso compatibili con l'AFDX: la comunicazione è basata sulle porte *sampling* e *queuing*, che sono tuttavia identificate per nome. La fase di *configurazione* del nostro Middleware fornisce le informazioni che non sono date al momento della *creazione* della porta.

sintassi non è fatta alcuna distinzione tra porte *sampling* e *queuing*, a differenza delle API ARINC 653, che distinguono tra `READ_SAMPLING_PORT` e `READ_QUEUING_PORT` (analogamente per la scrittura).

4.2.6 Funzioni speciali

La memoria condivisa delle funzioni speciali contiene due code di messaggi, per la comunicazione dal Driver al Target e viceversa. A differenza dell'AF-DX, l'invio e la ricezione sono operazioni bloccanti: l'operazione di ricezione non termina finché la coda è vuota, l'operazione di invio non termina finché la coda è piena. Per implementare questo comportamento si è ricorso a una *condition variable*, usata per segnalare ai processi in attesa il cambiamento dello stato delle code. Per semplicità, vi è un'unica *condition variable* per entrambe le code. Inizialmente erano state utilizzate le *interprocess condition variable* di Boost.Interprocess, sostituite a causa della loro inefficienza da una nostra reimplementazione basata su [Birrell, 2004] che utilizza i semafori di sistema.

La memoria condivisa contiene anche alcune variabili di stato:

`target_ready` (booleano) Indica se il target è inizializzato e pronto a ricevere comandi

`target_id` (intero) Identificatore numerico del target (1 per AMC1, 2 per AMC2)

`target_conf` (intero) Identificatore del ruolo del target (master o standby)

`partner_available` (booleano) Indica la presenza dell'alimentazione elettrica del partner. Utilizzato solo nel test system.

4.2.7 Interfaccia esterna in C

Per l'interfaccia esterna delle memorie condivise è stato scelto il linguaggio C sia per permettere interoperabilità tra compilatori diversi sia perché alcuni componenti di altri progetti che le utilizzeranno sono scritti in C. Il principio seguito è quello del "C a oggetti", operando di fatto il *name mangling* manualmente. Per ogni metodo, costruttore o distruttore è implementata una funzione adattatore secondo la seguente trasformazione.

- I metodi statici sono esportati come mere funzioni globali.
- Gli oggetti sono visibili dall'esterno solo come puntatori a strutture opache.

- Le funzioni esportate non possono generare eccezioni.
- I costruttori sono esportati come funzioni che restituiscono un puntatore all'oggetto creato. Esempio:

  `Arinc429SharedMemory(const char*)`

  diventa

  `Arinc429SharedMemory* arinc429_init(const char*).`
- I distruttori sono esportati come funzioni che hanno come unico parametro un puntatore all'oggetto, ad esempio

  `arinc429_fini(Arinc429SharedMemory*).`
- I metodi di istanza sono esportati come funzioni che hanno gli stessi parametri del metodo più un puntatore all'oggetto come primo parametro. Esempio:

  `AnalogSharedMemory::set_channel_status(int channel, int status)`

  diventa

  `analog_set_channel_status(AnalogSharedMemory* self, int channel, int status).`

La fatica di scrivere a mano una funzione adattatore per ogni metodo è stata ripagata dal risultato di totale interoperabilità raggiunto. Dall'altra parte, la comodità di utilizzo delle librerie non è venuta meno, eccetto l'impossibilità di utilizzare gli *smart pointers*. Sarebbe stato possibile scrivere delle classi *wrapper* in C++ che permettessero di usare gli oggetti con la normale sintassi del C++ ma non lo si è ritenuto opportuno in quanto avrebbe peggiorato la manutenibilità del codice aggiungendo un'ulteriore duplicazione di dichiarazioni.

4.3 *Endianizer*

4.3.1 Struttura del componente

Abbiamo chiamato *Endianizer* il componente che si occupa della correzione dell'endianness dei dati, inserendosi subito prima dell'invio e subito dopo

la ricezione. Le interfacce coinvolte sono AFDX e memoria flash; inizialmente era predisposta anche per l'ARINC 429, per il quale si è rivelata non necessaria, in quanto l'estrazione dei campi dalle parole ARINC 429 è già compito del Middleware. La configurazione è caricata all'avvio e contiene una forma riassunta del formato dei messaggi, insieme ai loro indicatori, più indicazioni di operazioni di byte-swapping, inserite manualmente nei casi in cui il comportamento predefinito non è soddisfacente. Il componente è suddiviso in tre parti:

- un *parser* che legge i file di configurazione (un formato testuale molto semplice ispirato a quello dell'ICD) e li converte in strutture dati;
- un *compilatore* che trasforma le indicazioni sul formato dei messaggi in liste di operazioni di spostamento o di byte-swapping, possibilmente ottimizzando le operazioni consecutive in alcuni casi;
- un *interprete* che esegue queste operazioni.

4.3.2 Formato della configurazione

I messaggi sono identificati da una struttura che contiene il tipo del bus e le coordinate del messaggio, che une volta codificata è interamente contenuta in un intero da 64 bit, per velocizzare le operazioni di ricerca. Ad esempio, nel caso dell'AFDX la struttura contiene:

- il tipo, che indica AFDX (8 bit)
- la direzione (TX/RX, 1 bit)
- il VL (16 bit)
- la porta UDP (16 bit).

Nel caso dei segmenti di memoria flash l'unica coordinata è il numero del segmento.

Per ogni messaggio, la configurazione definisce l'elenco dei campi, basata sull'ICD, ognuno con:

- nome (usato solo per diagnostica)
- posizione del byte iniziale (che per convenzione parte da 1)
- posizione del MSB
- lunghezza in bit

- flag per indicare se un campo è un *bit-field*.

È inoltre possibile specificare esplicitamente dei blocchi sottoposti a bit- o byte-swapping, che hanno come parametri la posizione del byte iniziale e la lunghezza in bit. Questa possibilità è stata inserita per i casi in cui il comportamento predefinito non portava al risultato desiderato, o in presenza di errori nella stesura dell'ICD, ossia discrepanze tra il formato del messaggio così come specificato nel codice applicativo e quello specificato nell'ICD.

4.3.3 Compilazione dei messaggi ed endianizzazione

Il formato di ogni messaggio viene "compilato" per ottenere una lista di operazioni di spostamento sui bit dei dati che convertono il messaggio dal formato di partenza (big-endian) al formato che il retarget si aspetta (little-endian). Ogni operazione di spostamento è una tupla `(start_bit, length, dest_bit, swap_bytes, swap_bits)` che indica la copia di una porzione di messaggio lunga `length`, a partire da `start_bit` nel messaggio originale e `dest_bit` nel messaggio modificato – il messaggio originale e quello alterato occupano porzioni di memoria diverse, cosicché le operazioni successive non possano interferire tra loro. I flag `swap_bytes` e `swap_bits` indicano che l'intervallo indicato debba essere rovesciato rispettivamente bit per bit o byte per byte, nel messaggio originale, prima della copia. L'endianizzazione funziona allo stesso modo per i messaggi in ingresso (direzione "avanti") e per quelli in uscita (direzione "indietro"), nel qual caso le operazioni sono svolte in ordine inverso e gli swap, ove previsti, sono eseguiti alla fine dell'operazione anziché all'inizio.

La compilazione sfrutta un insieme di assunzioni sul formato del messaggio e procede in modo diverso a seconda del bus e dell'implementazione delle funzioni di lettura nel codice applicativo dell'AMC. Nel caso dell'AFDX le assunzioni sono le seguenti.

- I campi dei messaggi si possono tutti raggruppare in blocchi da 4 byte oppure 8 byte nel solo caso dei numeri a virgola mobile in precisione doppia (questa assunzione discende direttamente dal protocollo AFDX).

- Per i campi da 2, 4 o 8 byte che non sono bit-field è sufficiente emettere un'istruzione di byte-swapping.

- Se un blocco da 4 byte è formato interamente da campi da 8 bit, anche se sono contrassegnati come bit-field, essi sono letti separatamente dall'applicativo e non necessitano di conversioni.

- Le sequenze di bitfield sono sempre lette a blocchi di 4 byte. Se una di queste sequenze è composta solo da campi lunghi un bit, il blocco viene letto come array di booleani e deve pertanto essere rovesciato bit per bit; altrimenti il blocco viene letto come un intero da 32 bit e i campi estratti dall'applicativo attraverso l'aritmetica. In quest'ultimo caso il blocco deve essere rovesciato byte per byte. Esistono eccezioni – blocchi composti da campi booleani che sono letti come interi opachi – e questi devono necessariamente essere corretti a mano.

Il processo di compilazione cerca di gestire i casi di ambiguità più semplici, ad esempio campi da 8 bit che occupano un intero byte, che possono essere interpretati sia come parte di una "parola mista" da 32 bit, letta come un singolo intero (caso più probabile), sia come singoli byte. Inoltre controlla che i campi non si sovrappongano – possibile solo per errori di copiatura o errori nell'ICD DB – nel qual caso emette un messaggio di errore.

4.3.3.1 Limiti

L'implementazione attuale non gestisce i messaggi a dimensione variabile, per gestire i quali sarebbe necessaria un'estensione dell'interprete in modo da poter specificare operazioni condizionali e posizioni variabili o lette da appositi registri, che sarebbero riempiti a partire dal contenuto di alcuni campi con istruzioni apposite. Dovrebbe inoltre essere estesa la sintassi del file di configurazione. È stata valutata la possibilità di aggiungere queste funzionalità ma non è stata ritenuta opportuna.

4.3.3.2 Eccezioni

Nei casi particolari è previsto l'inserimento di funzioni di endianizzazione scritte a mano. Questa possibilità è stata sfruttata per tutti i segmenti della memoria flash, specialmente per il database del FMS, a causa della sua struttura molto variabile e della non uniformità delle funzioni che nel codice applicativo decodificano i singoli record delle tabelle.

4.3.4 Generazione della configurazione

Come detto sopra, i file di configurazione dell'Endianizer sono ricavati dall'ICD database. Nella figura 4.9 è mostrato più precisamente il processo che porta alla loro generazione. Abbiamo fatto riferimento al formato di ICD utilizzato all'interno del Test system. Da questo sono ricavati sia le versioni iniziali dei file di configurazione dell'Endianizer sia quelli del Gateway verso il Framework. Questa duplicazione di informazioni si è resa necessaria

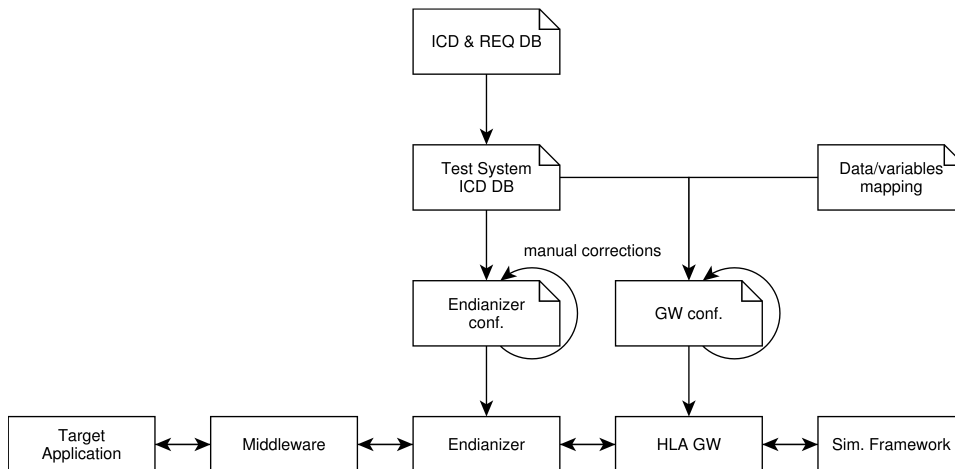


Figura 4.9: Flusso di informazioni dall'ICD database all'Endianizer (verticale); flusso di dati attraverso l'Endianizer (orizzontale) dal Driver al Target e viceversa

in quanto le due configurazioni hanno informazioni con scopi diversi. Nel caso dell'Endianizer sono inoltre aggiunte correzioni manuali che non troverebbero posto in una configurazione in comune col Gateway, e giustificano la nostra scelta di non utilizzare direttamente il formato del Test system, sebbene il nostro sia molto simile. Analogamente per la configurazione del Gateway, che oltre alle informazioni sulle coordinate e sul formato delle variabili dell'ICD deve specificare anche la corrispondenza con le variabili del Framework, informazione che deve essere inserita a mano, dato che collega dati organizzati in modo completamente differente.

4.4 VTE Proxy

Questo componente si preoccupa della comunicazione tra il Test System (rappresentato dal processo VTE Driver) e il Target (attraverso le memorie condivise). È stato scritto da zero e sostituisce la vecchia parte di comunicazione del VTE, che prima era incorporata nel Middleware. Il protocollo di comunicazione col driver VTE è un semplice protocollo domanda/risposta, totalmente sincrono, sopra il TCP/IP. Di seguito la descrizione delle classi più importanti.

4.4.1 `VTE_Proxy`

Questa è la classe principale, che viene avviata dalla funzione `main()`. È un ciclo infinito che apre la porta TCP, si mette in ascolto, riceve i messaggi e chiama i MessageHandler appropriati per elaborarli. L'inizializzazione del

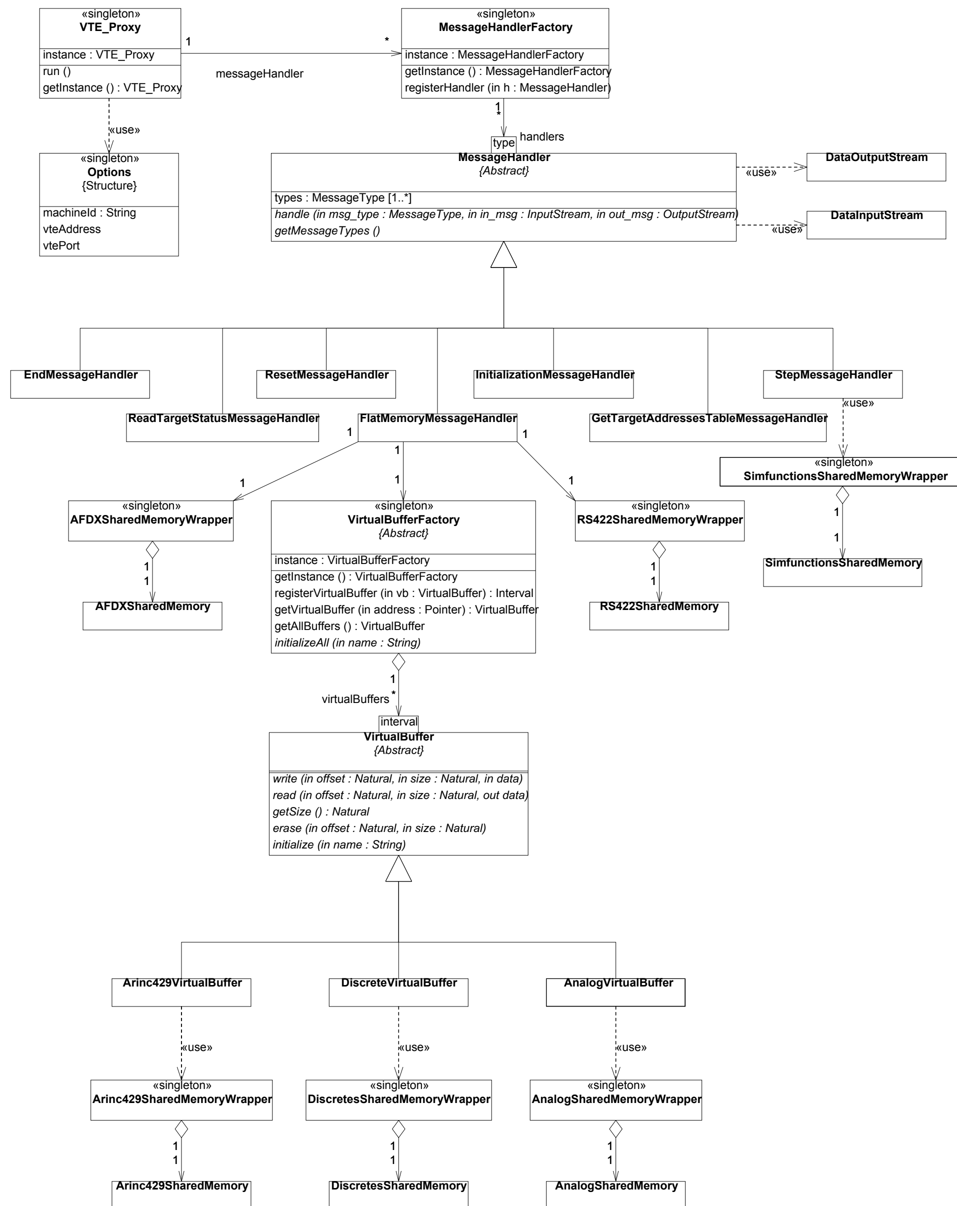


Figura 4.10: Diagramma di alcune classi del VTE proxy

processo e la lettura delle opzioni dal file di configurazione stanno nel modulo della funzione `main()`.

4.4.2 `MessageHandler`

I messaggi che possono essere ricevuti dal VTE Proxy sono diversi e sono identificati dal primo carattere che li compone. Poiché il protocollo non prevede nell'intestazione del messaggio un modo per segnalare la sua lunghezza, ai diversi gestori dei messaggi sono passati direttamente i puntatori al flusso in entrata e al flusso in uscita. È utilizzato il *pattern* della *factory* per fare in modo che i `MessageHandler` si registrino automaticamente all'avvio, segnalando quali tipi di messaggi (cioè i loro codici identificativi) sono in grado di gestire.

Nella figura 4.10 sono mostrate alcune delle classi figlie di `MessageHandler`. I messaggi più importanti sono senza dubbio quelli di `Step`, che comanda l'esecuzione di un minor frame del Target e `FlatMemory` che si occupa della lettura/scrittura. I messaggi sono raggruppati in catene, sempre terminate da un messaggio di `End`, che causa lo svuotamento del buffer in uscita del VTE Proxy, col conseguente invio di tutte le risposte alle richieste della catena, o un semplice "ack" se non sono previste risposte.

L'input/output è realizzato tramite comandi di lettura/scrittura verso indirizzi di memoria grezzi, che dovrebbero contenere le strutture dati del VTE. Il messaggio `GetTargetAddressTable` fornisce al Driver VTE gli indirizzi di base delle strutture di ogni bus, mentre il messaggio `FlatMemory` ha i seguenti parametri: direzione, tipo (per estenderlo con altri protocolli), indirizzo, lunghezza della richiesta e dati (solo nel caso della direzione in scrittura). Nel nostro caso non esponiamo nessun indirizzo reale ma simuliamo il contenuto della memoria interpretando le richieste sulla base dell'indirizzo attraverso il meccanismo dei `VirtualBuffer`. Gli unici due protocolli che fanno eccezione sono AFDX e RS 422, aggiunti in tempi più recenti, che indicano nell'intestazione del messaggio il protocollo (nel parametro "tipo") e le coordinate (il numero della porta nel caso della seriale RS 422, virtual link e porta UDP nel caso dell'AFDX). Per questi protocolli la richiesta viene direttamente inoltrata alla memoria condivisa.

4.4.3 `VirtualBuffer`

I `VirtualBuffer` si preoccupano di simulare il contenuto in memoria delle strutture dei bus del VTE, senza realmente contenerle. Sono identificati ognuno da un intervallo di memoria; all'avvio si registrano presso la classe

VirtualBufferFactory indicando la dimensione del loro spazio di indirizzamento e viene loro assegnato un indirizzo di base (non fissato a priori per evitare collisioni nel caso si aggiungessero altri VirtualBuffer). All'arrivo di una richiesta di lettura o scrittura, FlatMemoryMessageHandler cerca il VirtualBuffer che contiene l'indirizzo della richiesta e chiama su di esso il metodo write() o read() con l'offset dell'indirizzo della richiesta rispetto al loro indirizzo di base. Per la scrittura, il VirtualBuffer decodifica la richiesta e la inoltra alla corrispondente memoria condivisa; nel caso di una lettura il VirtualBuffer legge la memoria condivisa corrispondente e ne restituisce una versione codificata nel formato della memoria del VTE opportuna.

Per identificare la richiesta in base all'indirizzo si tiene conto del fatto che le strutture del VTE sono rigide e sono composte da array di strutture.

4.4.4 *SharedMemoryWrapper

La lettura e la scrittura verso le memorie condivise sono intermediate da degli adattatori (wrapper) che sono stati utili quando si è voluto collegare lo stesso VTE Proxy a più Target ridondati. In questo caso le memorie condivise per ogni protocollo sono una per ogni target e le operazioni di scrittura sono replicate per ognuna di esse. Le classi SharedMemoryWrapper si preoccupano inoltre dell'inizializzazione delle memorie condivise al loro primo utilizzo.

Capitolo 5

Implementazione e verifica

Questo capitolo discute le problematiche manifestatesi nel corso della realizzazione del progetto col loro impatto sull'architettura del sistema e descrive la fase di verifica e validazione.

5.1 Problematiche e cambiamenti richiesti

In questa sezione sono discussi i principali problemi verirficatisi nel corso del progetto, che hanno causato cambiamenti ai requisiti o hanno richiesto soluzioni *ad hoc*.

5.1.1 Problemi di endianness non risolvibili dall'esterno

Il principio di "endianizzazione" secondo il quale è sufficiente alterare l'input/output lasciando intatto il codice applicativo (§ 3.3.4) non è stato sempre efficace. In un caso notevole si è preferito ricorrere ad una correzione puntuale del codice dell'applicativo AMC originale.

Il caso in questione è quello del protocollo ARINC 739, utilizzato per il collegamento con la MCDU (*Multiple Control Display Unit*), un tipo di pannello con display testuale e tastiera alfanumerica in grado di connettersi a più sorgenti diverse selezionabili dall'utente, l'interfaccia principale con cui si accede alle funzioni del FMS e altre funzioni dell'AMC. Il protocollo ARINC 739 è un protocollo applicativo che utilizza l'ARINC 429 come trasporto. La sua caratteristica principale è che la struttura dei messaggi (le parole ARINC 429) non è fissa e può assumere diverse forme a seconda non solo di label e SDI ma anche del contenuto di una parte del campo dati

L'"endianizzazione" a struttura fissa (il caso normale) non avrebbe funzionato: la struttura variabile dei messaggi impiega i bit-field, pertanto il modo in cui la loro struttura viene alterata cambiando l'endianness non è

uniforme tra le diverse strutture possibili. Si sarebbe dovuto espandere l'endianizzatore per prevedere dati a struttura variabile, o inserire una funzione di endianizzazione apposita per i dati della MCDU – soluzione quest'ultima per nulla "pulita".

Si è ritenuto più semplice agire sulle dichiarazioni delle strutture (solo una ventina) riscrivendole in versione little-endian (cioè invertendo la numerazione dei bit) e aggiungendo delle direttive di compilazione condizionale che selezionano una versione o l'altra della dichiarazione a seconda dell'architettura.

5.1.2 Collegamento con l'emulatore MCDU

Il problema del protocollo non è stato l'unico inconveniente con la MCDU. Mentre l'implementazione del Retarget era in corso, insieme al cliente si è stabilita la modalità di integrazione della MCDU all'interno del simulatore. La direttiva specificava di riciclare un emulatore di MCDU già in uso in azienda a scopo di test, con i dovuti adattamenti sia dal punto di vista della grafica sia del comportamento. Questo emulatore utilizzava un protocollo di comunicazione con l'AMC semplificato, che richiedeva un'apposita *patch* nel codice applicativo dell'AMC, che andava a sostituire il modulo che implementava il protocollo ARINC 739 con il suo.

È stato ritenuto più conveniente, in termini di tempo, utilizzare questa *patch* anziché adattare l'emulatore al protocollo ARINC 739. Il sistema di test, tuttavia, utilizza il protocollo originale. Di conseguenza si è dovuto creare eseguibili separati, rilasciati separatamente, per le configurazoni "test" e "simulatore", venendo meno a uno dei propositi di generalità che ci eravamo posti all'inizio del progetto.

5.1.3 Sincronizzazione delle *special functions*

Come descritto nel paragrafo 4.2.6, la sincronizzazione delle code nella memoria condivisa delle funzioni speciali inizialmente utilizzava le *condition variable* interprocesso fornite dalla libreria Boost.Interprocess, che sono state successivamente cassate in ragione delle loro prestazioni inadeguate, essendo sostanzialmente implementate con un *busy loop* che nei momenti di attesa saturava il consumo di CPU. Nonostante le prestazioni in termini di tempi di risposta fossero soddisfacenti e il consumo di energia del simulatore non fosse un problema, ciò che ci ha spinto a cambiare implementazione è stato l'impatto sulle statistiche di consumo dei processi del simulatore, la cui interpretazione veniva alterata significativamente dalla saturazione della CPU nei momenti di attesa.

Inizialmente ho scelto di lasciare inalterata l'implementazione della memoria condivisa reimplementando la *condition variabile* in modo che utilizzasse le primitive di sincronizzazione del sistema operativo. Non esistendo sotto Windows un'implementazione nativa delle *condition variable* [Schmidt and Pyarali, 1998], l'implementazione scelta è stata di emularla con tre semafori, come indicato in [Birrell, 2004]. Tale implementazione ha dimostrato dei grossi limiti, in quanto non prevede il caso in cui uno dei processi partecipanti viene interrotto mentre sta impegnando la risorsa e lascia i semafori in uno stato inconsistente, generando situazioni di *deadlock*.

Si è così abbandonata l'idea della *condition variable* e della coda in memoria condivisa, passando così alle *named pipe*, code interprocesso gestite direttamente dal sistema operativo, presenti sia in Windows che nei sistemi Unix (dove sono chiamate *FIFO*). L'implementazione è stata differenziata per Windows e per Solaris, senza tuttavia rinunciare alla possibilità di utilizzare lo stesso software nei due sistemi operativi. Si è utilizzato un sottoinsieme comune delle funzionalità offerte dalle *named pipe* nei due sistemi operativi; per esempio, vi è una coda per ognuna delle direzioni (le FIFO sono monodirezionali sotto Unix) e l'estremità in ricezione deve avere il ruolo di server. Ognuno dei due processi si trova quindi ad avere il ruolo sia di server che di client.

Il limite principale di questa ristrutturazione è stata la perdita del collegamento molti-a-molti che la mera memoria condivisa permetteva. Nella nuova versione non è possibile che un processo terzo possa vedere il contenuto dei messaggi senza partecipare alla "conversazione", e inoltre il numero di partecipanti è fissato a due dal design, un target e un driver. Se un secondo driver o un secondo target cercano di connettersi alla coda, la connessione fallisce. Inoltre non è possibile inviare un messaggio se l'altro capo della coda non è ancora connesso. La complessità della gestione delle code è cresciuta di molto. Ora, quando uno dei due processi si scollega, l'altro processo è responsabile della pulizia delle risorse in modo da poter subito riutilizzare il nome della coda.

5.2 Verifica e validazione

Questa sezione descrive il processo di verifica e validazione per il Retarget dell'AMC e i suoi risultati.

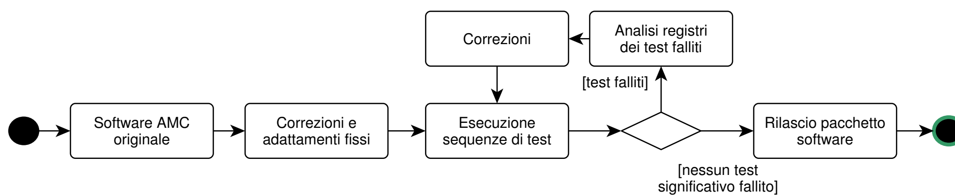


Figura 5.1: Processo iterativo di test e correzione del software. Il processo è stato ripetuto ad ogni nuova versione del codice applicativo dell'AMC.

5.2.1 Test funzionali

Poiché il requisito principale del progetto è simulare con un'ottima approssimazione il comportamento dell'AMC vero, si è adottato lo stesso sistema, la stessa metodologia e le stesse sequenze di test dell'apparato originale. La verosimiglianza è quindi misurata dal numero e dall'importanza delle procedure di test (sequenze di test) che danno lo stesso risultato tra il sistema originale e il Retarget, *fallimenti compresi*.

Nel caso del Retarget, nel corso dello sviluppo i test funzionali sono stati utilizzati anche come test di non regressione e hanno permesso di individuare errori causati da alcune modifiche al software. A tale scopo abbiamo adottato uno schema di versionamento per i rilasci parziali del Retarget.

5.2.1.1 Ambiente di test

L'architettura del sistema di test è già stata anticipata nelle sezioni 1.3 e 2.1. Come già detto, si è riutilizzato completamente il sistema di test del VTE, che è lo stesso dell'apparato reale, eccetto il componente di comunicazione: quando viene testato l'AMC vero, esso è collegato al server del Test System direttamente e l'I/O è gestito da una libreria software apposita. Quando è il VTE ad essere testato, la comunicazione tra l'ambiente di test – formato da emulatori e sequenze di test – è gestita dal Driver VTE. Per rendere possibile la comunicazione tra il Driver VTE e il Retarget abbiamo realizzato un componente apposito detto "Proxy VTE" (§ 4.4 a pagina 79), che comunica col Driver via TCP/IP nel suo protocollo originale. Il Proxy VTE va a sostituire una parte del Middleware che è stata espunta in quanto troppo accoppiata con la vecchia rappresentazione dei dati interni, sostituita nel Retarget dalle interfacce a memoria condivisa.

5.2.1.2 Sequenze di test

Le sequenze di test (procedure di test automatici) sono state scritte a partire dagli stessi requisiti del software applicativo, parallelamente allo sviluppo

dello stesso, e archiviate utilizzando lo stesso software di tracciatura dei requisiti del cliente. Ogni sequenza testa uno specifico insieme di requisiti, agendo sugli ingressi dell'AMC e verificando che le uscite concordino coi requisiti. Comandano anche gli emulatori che sostituiscono alcuni strumenti collegati all'AMC quali le radio e i ricevitori GPS.

Le sequenze di test adottano sia la metodologia *black-box* per cui vengono stimolati gli ingressi, con tutti i casi limite secondo i requisiti, e si controlla che le uscite rispettino gli stessi requisiti, sia quella *white-box*, ossia conoscendo il codice, utilizzata per ottenere gli altissimi livelli di copertura (in termini di istruzioni e di rami) richiesti per le certificazioni di sicurezza dei sistemi avionici *safety critical*.

Le sequenze di test utilizzate sono state selezionate in base alla loro applicabilità al Retarget e al driver VTE. Di conseguenza, non si è potuta utilizzare l'infrastruttura di test automatici per le sequenze che richiedono la presenza di entrambi gli AMC (limitazione del Driver).

5.2.1.3 Esecuzione dei test

Le sequenze sono suddivise per componente e sottocomponente in base ai requisiti che vanno a verificare. Si è scelto di somministrarle in ordine di complessità, iniziando con le sequenze del sistema di monitoraggio, seguito dalle radio, e tenendo le sequenze del FMS per ultime, essendo quelle che riguardano più componenti rispetto alle altre e che hanno una più lunga fase di set-up – la fase in cui vengono create le condizioni iniziali.

I test sono stati eseguiti periodicamente (v. fig. 5.1), per ogni nuova versione del codice applicativo dell'AMC, e l'analisi dei registri dell'esecuzione delle sequenze che fallivano è stata di fondamentale importanza per trovare nel codice gli errori che causavano il discostamento del comportamento da quello previsto. Spesso erano errori o imprecisioni nell'implementazione del Middleware, ma sono stati individuati anche errori del codice applicativo originale che si potevano manifestare solo in un'architettura little-endian, dovuti principalmente ad operazioni non consentite dal *coding standard*, cioè che facevano assunzioni sulla rappresentazione in memoria delle strutture dati.

Per quanto riguarda le funzionalità non testabili attraverso il test system, si è scelto di rimandare la loro verifica all'inizio della fase di integrazione, quando si è resa disponibile un'infrastruttura che consentisse di eseguire le due istanze dell'AMC in configurazione ridondata. Per quanto riguarda queste le funzionalità di ridondanza e allineamento dati, sono state redatte e operate delle sequenze di test manuali in totale isolamento, ossia senza

l'utilizzo del Driver e pertanto con un I/O limitato ad alcune variabili, impostate manualmente con l'ausilio dell'interfaccia grafica del Gateway HLA, eseguito senza collegamento al Framework. Non si è inteso in questo modo coprire tutti i casi necessari, ma solo quelli più importanti al fine di individuare eventuali errori gravi che causassero interruzioni (*crash*) o blocchi (*hang*).

5.2.1.4 Risultati

Sebbene i risultati iniziali non fossero incoraggianti, progressivamente siamo arrivati a ottenere il 100% di test passati per la parte di monitoraggio e complessivamente, esclusi i test che falliscono anche sull'apparato vero, la porzione di test passati superava il 95%. Possiamo quindi affermare che, escluse le questioni di ridondanza e di funzioni di simulazione, il comportamento del Retarget è conforme a quello dell'AMC vero.

5.2.2 Test di integrazione e di sistema

I test funzionali hanno dimostrato la qualità del Retarget preso isolatamente, ma non sono predittivi della bontà del prodotto quando utilizzato all'interno del simulatore, data la complessa rete di relazioni che si vanno ad aggiungere.

I test di sistema per il Retarget sono possibili solo nella fase di integrazione, che tuttavia esula dallo scopo di questa tesi. Per quanto riguarda le funzioni speciali, data la difficoltà di ricreare senza il simulatore le condizioni affinché esse abbiano effetto, la loro verifica è rimandata al momento dell'integrazione, quando anche nel simulatore saranno implementate.

Per quanto riguarda le prestazioni, nelle macchine utilizzate per i test – che non sono tuttavia uguali a quelle del simulatore – il tempo medio di esecuzione si attestava intorno ai 4-6 ms, a fronte di uno slot di 20 ms. Con l'aggiunta dei tempi del Gateway non si superano comunque i 15 ms. Si tratta di valutazioni approssimative, ma sufficienti per dare una valutazione positiva ai tempi di risposta del Retarget.

Capitolo 6

Conclusioni

In questo capitolo si discutono le scelte progettuali alla luce dei risultati ottenuti si dà una breve illustrazione della fase successiva.

6.1 Valutazione a posteriori

Riutilizzabilità Il Middleware che ho sviluppato per il Retarget, insieme alle memorie condivise e al VTE Proxy, è oggi utilizzato anche su Solaris, grazie ad un cross-compilatore. Inoltre lo stesso Middleware è stato impiegato, con risultati iniziali incoraggianti, per l'AMC di un altro elicottero, basato sulla medesima famiglia di computer. È già stato pianificato di riutilizzare le interfacce a memoria condivisa, prese isolatamente o insieme al VTE Proxy, per progetti diversi dall'elicottero in oggetto, in cui è necessario simulare il comportamento di un bus avionico. Ciò dà una conferma concreta che gli obiettivi di portabilità e riutilizzabilità del prodotto sono stati rispettati.

Interfaccia a memoria condivisa e sincronizzazione Per quanto riguarda l'interfaccia a memoria condivisa invece, con le conoscenze sui protocolli di comunicazione avionici che non avevo all'inizio del progetto, avrei organizzato i dati diversamente, in modo più generico, con un solo segmento di memoria condivisa in cui tutti gli oggetti sono trattati come "porte", *sampling* o *queuing*, indipendentemente dal bus (per esempio i discreti sarebbero stati delle porte sampling) indirizzati con le coordinate del loro protocollo compattate come ho successivamente previsto nell'Endianizer. In questo modo si sarebbe ridotta la mole di codice e soprattutto sarebbe stato possibile accedere al contenuto di tutti i bus con gli stessi metodi, cosa che avrebbe semplificato di molto la scrittura di un analizzatore dei bus ma

anche del Gateway. Non è da escludere in futuro una reimplementazione in quel senso, lasciando tuttavia la compatibilità col codice esistente.

Non mancano tuttavia le note negative, prima fra tutte l'implementazione del trasporto delle funzioni speciali (attraverso la memoria condivisa o le *named pipe*). Nessuna delle tre versioni realizzate è esente da problemi che si possono risolvere solo stravolgendo l'implementazione. La ragione sta nel fatto che è molto difficile trovare delle funzioni o librerie di sincronizzazione interprocesso che siano sufficientemente portabili e soprattutto che resistano all'interruzione dei processi. Anche un'implementazione basata sui socket avrebbe i suoi limiti (la necessità del ruolo di server, la configurazione delle porte) sebbene sia una possibile soluzione futura.

Ristrutturazione del codice esistente, prestazioni e manutenibilità La ristrutturazione del codice del VTE è stata resa possibile da una pianificazione che ha previsto tempo e risorse per perfezionare il sistema esistente. Come già spiegato nella sezione 4.1, ho scelto di reimplementare una buona parte del Middleware, eliminando dove possibile le stratificazioni non più necessarie, spostando la logica dei bus nelle memorie condivise e migrando le strutture dati dal C al C++ usando le librerie standard. Ho dato la priorità alle parti meno documentate (che coincidono con quelle più stratificate), che producevano un numero eccessivo di thread e a quelle che consumavano più CPU delle altre, individuate con un veloce *profiling.*

Si sono avuti di conseguenza dei notevoli miglioramenti dal punto di vista delle prestazioni, in termini di tempi di risposta ma anche di numero di thread, sceso drasticamente grazie ad alcune modifiche allo scheduler e alla reimplementazione dei semafori e degli eventi del Middleware. La rimozione di un'ampia parte di codice morto e la riscrittura di alcune funzioni implementate alla bisogna ha sicuramente migliorato la manutenibilità. L'implementazione della ridondanza e della DBU, nonostante sia ottimizzabile e non consideri tutti i casi limite, si è rivelata robusta alle perdite di sincronia che possono capitare occasionalmente al software del simulatore quando la macchina è sovraccarica, soprattutto durante lo sviluppo e i test di integrazione, che sono operati su hardware ridotto.

Endianness Nonostante il nostro approccio alla correzione dell'endianness abbia richiesto lo sviluppo di un componente in più e aggiunto uno strato intermedio nell'I/O – con un piccolo aumento del consumo di CPU, intorno al 2% – esso ha risolto completamente il problema e ha richiesto molto meno sforzo rispetto alla soluzione che prevedeva di alterare direttamente l'applicativo. Spiace tuttavia che non sia riutilizzabile se non per

progetti molto simili, ossia applicativi della stessa famiglia sviluppati dallo stesso produttore (l'esempio sopra menzionato del retarget AMC di un altro elicottero che utilizza lo stesso hardware e ricicla parte del software), avendo incluso numerose assunzioni sull'implementazione delle funzioni di lettura/scrittura nel codice applicativo. Si spera tuttavia che non sia più necessario nei retarget a venire, cioè che il codice applicativo sia già scritto con la portabilità in mente. Sorprende infatti che del codice che ha meno di 15 anni soffra ancora di problemi di endianness.

Collegamento con la MCDU Un'altra scelta che non avrei ripetuto, nonostante non riguardi direttamente il Retarget ma la fase di integrazione, è stata non aver adottato il protocollo ARINC 739 per il collegamento tra AMC e MCDU, emulando quindi il collegamento vero, che ci avrebbe risparmiato molte ore passate a correggere i problemi che si sono creati nell'adattare il protocollo semplice del vecchio emulatore MCDU alle nuove funzionalità, per renderlo utilizzabile nel simulatore. Questo tuttavia discendeva da una decisione presa a monte senza aver fatto una valutazione precisa del costo delle implementazioni alternative.

Rispetto delle scadenze Il progetto aveva due scadenze, una intermedia alla fine del 2012 e quella finale alla fine di maggio 2013. Nella prima rientrava la reimplementazione dell'I/O, tra Middleware e memorie condivise, e il collegamento col Driver VTE; nell'ultima si aggiungevano le correzioni di endianness (Endianizer compreso), le funzioni relative alla ridondanza e i test funzionali. Le scadenze sono state perfettamente rispettate, sia per una pianificazione a monte molto cautelativa, che prevedeva risorse anche per i margini di miglioramento, sia per una scelta attenta di quali parti sarebbe stato conveniente reimplementare e per quali parti accontentarci di una soluzione funzionante ma con un ampio margine di miglioramento, con un *trade-off* tra il tempo di sviluppo e la qualità del risultato. Ad esempio, stanti i vincoli temporali che ci sono stati dati, abbiamo dovuto trascurare il perfezionamento di alcuni aspetti: la riorganizzazione non ha riguardato l'intero Middleware e la sincronizzazione tra il Target e il Driver ha dei problemi noti, per cui si possono creare delle condizioni, per quanto rare, in cui non possiamo garantire il corretto funzionamento.

Insieme al codice sorgente e all'eseguibile compilato ho fornito la documentazione, come documento di design (SDD), manuale utente e note di rilascio, cui si è aggiunto in una fase successiva il documento dei test (STD), ad opera del team di testing. Al momento della consegna era quasi giunto al termine anche lo sviluppo del Gateway HLA (cui ho partecipato), com-

ponente necessario all'integrazione nel simulatore, che già era iniziata per procedere gradualmente.

Varie Infine lo sviluppo non ci ha risparmiato delle sorprese. È accaduto che una funzione implementata come "uovo di Pasqua" (cioè una funzione nascosta), come la riproduzione degli avvisi sonori direttamente dal Middleware, se i loro file audio si trovano in una determinata cartella, si è rivelata utile ed è diventata ufficialmente una caratteristica del Retarget nel simulatore – non più nascosta.

6.2 Sviluppi futuri: la fase di integrazione

Prima della consegna del progetto "Retarget AMC" è iniziata la fase di integrazione all'interno del simulatore. Tale fase è consistita prevalentemente, per quanto riguarda il Retarget, nello sviluppo del componente che si preoccupa del trasporto dei dati – il Gateway – e nel determinare una corrispondenza tra le variabili dei modelli del Framework (ognuno simula una parte diversa dell'elicottero o del mondo esterno) e quelle dell'ICD, ossia i dati che sono effettivamente trasmessi e ricevuti dall'AMC.

6.2.1 Il Gateway HLA

Come già illustrato nel paragrafo 3.1, il Gateway è quel componente che si occupa di fare da ponte tra il Retarget e il Framework di simulazione, agendo da federato. Il Gateway mette in comunicazione un mondo basato sui bus, qual è il Retarget, e un mondo basato sulle variabili, quali il Framework. Per ottenere questo sono necessari due tipi di informazioni (figura 4.9): da una parte l'ICD, che assegna delle coordinate fisiche ai nomi dei dati, e dall'altra l'abbinamento (*mapping*) tra i nomi dei dati dell'ICD e le variabili del simulatore, che sono campi delle strutture che i federati si scambiano, identificati dal nome della struttura e dal nome del campo.

Svolgendo una funzione simile, il Gateway è strutturato in maniera analoga al Proxy VTE (sempre al paragrafo 3.1), tuttavia con un design diverso, essendo stato sviluppato separatamente. Il Gateway è composto da diverse classi, che agiscono da codificatori/decodificatori (*codec*) ognuno per un tipo di bus diverso. I codec si preoccupano di convertire effettivamente i dati grezzi in dati ingegneristici ognuno secondo la sua configurazione, che contiene il suddetto mapping, e li passano a un altra classe che li posiziona all'interno della struttura corrispondente. Uno di questi codec si occupa delle funzioni speciali del simulatore, nonché di avviare e abbattere il processo

dell'AMC a comando, ossia basandosi sullo stato dell'alimentazione elettrica dell'AMC, che è una delle variabili del Framework.

6.2.2 Mappatura dei dati

Si noti che il problema della mappatura non è solo un problema di formato (che è risolto efficacemente dal Gateway) ma di semantica. Essendo stati sviluppati indipendentemente modelli e AMC, non esistendo spesso una corrispondenza diretta, si sono rese necessarie delle aggiunte che si preoccupassero di tradurre la logica dei dati dei modelli nella logica di quelli dell'AMC. Per esempio, la modalità operativa di un apparato collegato all'AMC può essere rappresentata da un'unica variabile all'interno del suo modello, mentre in ingresso all'AMC è data dalla combinazione di più ingressi separati.

Molte delle variabili indicano degli stati di salute o delle validità che per il simulatore non hanno senso (in quel caso il valore deve essere fissato a "valido") ma che devono avere un valore coerente per permettere all'AMC di funzionare. Considerando che le variabili dell'ICD sono in tutto qualche decina di migliaia, non è possibile trovare una corrispondenza con tutte. Ciò accade perché nel simulatore ogni apparato è considerato sempre funzionante, a meno di indicazioni esplicite di simulazione di guasti; al contrario, nel mondo reale dell'AMC, un apparato non è considerato funzionante se non comunica il suo stato correttamente e con tutti i messaggi previsti.

Complessivamente, la fase di integrazione del solo AMC ha richiesto più lavoro rispetto al design e all'implementazione sommati. L'aspetto più difficile non è stata la mappatura dei dati in sé (dal Retarget al Framework), nonostante sia stata fatta a mano una variabile alla volta, ma la selezione di un sottoinsieme di dati e loro validità in modo da riprodurre il comportamento corretto dei componenti utilizzati nel simulatore. Spesso i problemi che si sono verificati erano dovuti, più che a questioni tecniche, a problemi di comunicazione tra i vari team e soprattutto alla scarsa disponibilità e interpretabilità dei documenti dei requisiti, vanificando tutto il lavoro di tracciatura, dai requisiti fino al codice ma anche del flusso di dati, che viene normalmente fatto nel corso dello sviluppo del software avionico, che per motivi a me ignoti non è stato possibile sfruttare durante l'integrazione.

6.2.3 Testing e certificazione

Durante la fase di integrazione si sono manifestati degli errori nel sistema Retarget che non si erano manifestati nella fase di test in isolamento. Alcuni erano errori del Middleware che i test funzionali non avevano individuato poiché ancora in corso, oppure riguardavano funzionalità che il VTE non

prevede, come la ridondanza e la DBU o la riproduzione degli avvertimenti sonori.

Si sono manifestati anche dei bug del Gateway, che non è stato possibile testare prima della fase di integrazione; è stato possibile effettuare dei test parziali, quando il Framework e i modelli non erano ancora disponibili, grazie a un'interfaccia grafica appositamente scritta per alterare a mano i valori delle variabili.

Per quanto riguarda i test di sistema, sono stati effettuati sia test automatici che manuali, come previsto dalla QTG (v. § 1.1.1.1).

Dopo le difficoltà iniziali di comunicazione, sia tra team diversi della stessa azienda sia tra le diverse aziende fornitrici che concorrevano allo sviluppo, si è riusciti a raggiungere un certo regime che ci ha permesso, dopo quasi un anno, di ottenere la qualificazione del simulatore da parte dell'autorità competente.

Bibliografia

AdaCore. *GNAT Reference Manual, version 4.8.1*, 2012. URL `http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gnat_rm/`.

AIT. *ARINC 429 protocol tutorial*. Avionics Interface Technologies, Omaha, NE, USA.

Randal P. Andress. Wholesale byte reversal of the outermost Ada record object to achieve endian independence for communicated data types. *Ada Lett.*, XXV(3):19–27, settembre 2005. ISSN 1094-3641. doi: 10.1145/1102251.1102252. URL `http://doi.acm.org/10.1145/1102251.1102252`.

ARINC-610B. Guidance for use of avionics equipment and software in simulators. ARINC report 610B, ARINC Inc., dicembre 2001.

Andrew D. Birrell. Implementing condition variables with semaphores. In *Computer systems theory, technology, and applications*, pages 29–37. Springer-Verlag, 2004.

R.P.G. Collinson. *Introduction to avonics systems*. Springer, third edition, 2011. ISBN 978-9400707078. doi: 10.1007/978-94-007-0708-5.

CS-FSTD(H). Certification specifications for helicopter flight simulation training devices. Annex to ED Decision 2012/011/R, EASA, giugno 2012.

J.S. Dahmann, R.M. Fujimoto, and R.M. Weatherly. The DoD High Level Architecture: an update. In *Simulation Conference Proceedings, 1998. Winter*, volume 1, pages 797–804, Washington, DC, USA, 1998. doi: 10.1109/WSC.1998.745066.

ENAC OPV-17B. Dispositivi per addestramento e controllo del personale navigante – procedure per la qualificazione e l'utilizzazione. Circolare OPV-17B, ENAC, luglio 2010.

Ion Gaztañaga. Boost.Interprocess documentation. In *Boost Library Documentation 1.51*. 2012. URL http://www.boost.org/doc/libs/1_51_0/doc/html/interprocess.html.

GE. *AFDX/ARINC 664 protocol tutorial*. GE Fanuc.

ICAO-9625 issue 3. Manual of criteria for the qualification of flight simulation training devices. ICAO Doc. 9625, ICAO, 2009.

ICAO-9625 vol. II. Manual of criteria for the qualification of flight simulation training devices. ICAO Doc. 9625, ICAO, 2012.

IEEE 1516. IEEE standard for modeling and simulation (M&S) High Level Architecture (HLA) – framework and rules. IEEE Std. 1516, IEEE, 2010.

Intel. Endianness white paper. novembre 2004. URL http://www.intel.com/design/intarch/papers/endian.htm.

JAR-FSTD H. Helicopter flight simulation training devices. Joint Aviation Requirements JAR-FSTD H, JAA, maggio 2008.

Thomas Quinot. Bridging the endianness gap. *Ada answers*, (Ada gem #140), gennaio 2013. URL http://www.adacore.com/adaanswers/gems/gem-140-bridging-the-endianness-gap/. scaricato 11/07/2013.

Douglas C. Schmidt and Irfan Pyarali. Strategies for implementing POSIX condition variables on Win32. *C++ Report, SIGS*, 10(5), giugno 1998. URL http://www.cs.wustl.edu/~schmidt/win32-cv-1.html.

Michael A. Smith. *Object-oriented software in Ada 95*. McGraw-Hill, second edition, maggio 2011.

Elenco delle figure

1.1 Schema semplificato dei componenti di un generico velivolo . 8
1.2 Architettura dell'AMC 13
1.3 Esempio di scheduling 14
1.4 Struttura del test system 17
1.5 Rappresentazione delle configurazioni "simulatore" e "test" . 18

2.1 Componenti del test system e del VTE 21
2.2 Diagramma di sequenza di una tipica sessione del VTE 22
2.3 Esempi di collegamento ARINC 429 24
2.4 Struttura di un messaggio ARINC 429 24
2.5 Esempio di rete AFDX 26
2.6 Rappresentazione grafica dei Virtual Link AFDX 27
2.7 Esempio di *endianness* 32
2.8 Struttura con bit-field a cavallo dei byte 34
2.9 Struttura che adopera dei *bit-field* nelle sue versioni big- e little-endian 35
2.10 Definizione di un *record* in Ada 95 con una rappresentazione esplicita del formato. 36
2.11 Schema del framework di simulazione 37

3.1 Diagramma di struttura composta del "sistema retarget" con una parte del contesto 40
3.2 Diagramma di struttura composta del solo Retarget AMC più in dettaglio, con tutti i componenti 41
3.3 Diagramma dei casi d'uso del "sistema retarget" visto dal Driver 43
3.4 Diagramma di sequenza dell'esecuzione sincrona di un tipico step 46
3.5 Diagramma dei casi d'uso relativo alla ridondanza 47
3.6 Diagramma di sequenza della procedura di Database Update. 49

4.1 Diagramma dei componenti principali del Middleware 55

4.2 Diagramma di attività dello scheduler 58
4.3 Realizzazione delle *simulation functions* 59
4.4 Comunicazione tra le due istanze dell'AMC 60
4.5 Diagramma degli stati dell'automa della ridondanza 64
4.6 Sincronizzazione dell'operazione di DBU tra livello Ada e livello C++ 66
4.7 Diagramma degli stati dell'automa della DBU 67
4.8 Diagramma delle classi di memoria condivisa 70
4.9 Flusso di dati e configurazioni attraverso l'Endianizer 79
4.10 Diagramma di alcune classi del VTE proxy 80

5.1 Processo iterativo di correzione del software 86