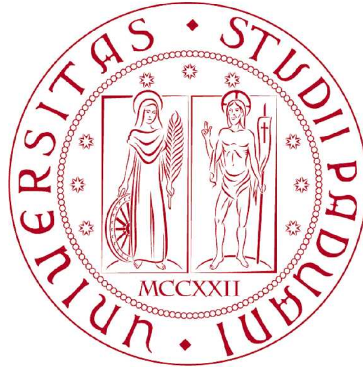
Università degli Studi di Padova

Scuola di Ingegneria
Dipartimento di Ingegneria dell'Informazione



Tesi di laurea magistrale in Ingegneria Informatica

Gestione lato client dei dati di esercizio di generatori elettrici per la propulsione navale

Laureando:
Fabio Marangon

Relatore:
Ch.mo Prof. **Carlo Ferrari**

Correlatore:
Ing. **Maurizio Rizzi**

Anno Accademico: 2014-2015

Indice

Indice i

Elenco delle figure iii

Elenco dei listati di codice v

Sommario vii

1 Introduzione 1
1.1 Il problema . . . 1
1.2 Contenuti . . . 2

2 Analisi dei requisiti 3
2.1 Il regolatore *MEC-100* . . . 3
2.2 Software client esistente . . . 4
2.3 Scheda *Hachiko* . . . 5
2.3.1 Il progetto *Yocto* . . . 5
2.4 Analisi dei requisiti . . . 6
2.5 Analisi del carico di lavoro . . . 8
2.5.1 Stato attuale . . . 9
2.5.2 Ottimizzazione . . . 9

3 Architettura del sistema 11
3.1 Organizzazione del software . . . 12
3.2 Interazioni client-server . . . 14
3.2.1 Dettagli sui pacchetti dati . . . 15

4 Database 19
4.1 Schema *ER* - *Entity-Relationship* . . . 19
4.1.1 Modellizzazione dei dispositivi hardware . . . 19
4.1.2 Modellizzazione dei dati rilevati e delle assistenze . . . 21
4.1.3 Modellizzazione degli allarmi . . . 22
4.2 Normalizzazione dello schema . . . 22
4.2.1 Definizioni . . . 23
4.2.2 *1NF* . . . 23
4.2.3 *2NF* . . . 23
4.2.4 *3NF* . . . 24
4.2.5 *BCNF* . . . 24
4.3 Costruzione del database . . . 24

5 Architettura client **25**
5.1 Struttura software . . . 25
5.1.1 Memoria condivisa . . . 26
5.1.2 Modulo *DataProcessor* . . . 27
5.1.3 Modulo *CommModule* . . . 30
5.1.4 Modulo *GUIModule* . . . 34
5.1.5 Modalità di utilizzo . . . 34
5.1.6 Dettagli sulla comunicazione client-server . . . 34

6 Sviluppi futuri e conclusioni **39**
6.1 Manutenzione . . . 39
6.2 G.U.I. . . . 39
6.3 Sicurezza . . . 40
6.4 Miglioramenti nella gestione dei file . . . 40
6.5 Altre estensioni . . . 40
6.6 Conclusioni . . . 41

A Codice SQL per la creazione del database **43**

Bibliografia **47**

Elenco delle figure

2.1 Un generatore *Marelli* . . . 3
2.2 Il regolatore *MEC-100* . . . 3
2.3 Pacchetto dati *MEC-100* . . . 4
2.4 Componenti hardware dal generatore alla rete . . . 5
2.5 Scheda *Hachiko* e display *Pengwyn* . . . 6

3.1 Architettura software sviluppata . . . 11
3.2 Architettura server . . . 13
3.3 Architettura client *Hachiko* . . . 13
3.4 Pacchetto di comunicazione . . . 14
3.5 *Payload* pacchetto *VALUES* . . . 16
3.6 *Payload* pacchetto *LOGIN* . . . 16
3.7 *Payload* pacchetto *MAINTENANCE* . . . 17

4.1 Schema *ER* . . . 20
4.2 Schema *ER* - Entità per le schede . . . 21
4.3 Schema *ER* - Entità per valori di esercizio, parametri e assistenze . . . 21
4.4 Schema *ER* - Entità per gli allarmi . . . 22

5.1 *Flow-chart* modulo dati . . . 28
5.2 *Flow-chart* modulo comunicazione . . . 31
5.3 Comunicazione client-server - *CODE 0* . . . 34
5.4 Comunicazione client-server - *CODE 1* . . . 35
5.5 Comunicazione client-server - *CODE 2* . . . 36
5.6 Comunicazione client-server - *CODE 3* . . . 37
5.7 Comunicazione client-server - *CODE 4* . . . 37
5.8 Comunicazione client-server - *CODE 5* . . . 38
5.9 Comunicazione client-server - *CODE 6* . . . 38

Elenco dei listati di codice

5.1 Avvio dei *thread* indipendenti . . . 25
5.2 Memoria condivisa . . . 26
5.3 Ingresso in modalità manutenzione . . . 29
5.4 Lettura dati di esercizio . . . 29
5.5 Invio dati al *CommModule* . . . 30
5.6 Controllo presenza dati da spedire . . . 32
5.7 Invio dei dati di allarme . . . 32
5.8 Costruzione pacchetti . . . 32
5.9 Costruzione *payload* pacchetto *BOOTSTRAP* . . . 33
A.1 Codice SQL per la creazione del database . . . 43

Sommario

L'evoluzione tecnologica spinge con sempre maggior forza ed intensità verso un mondo in cui *tutto* sia collegato in rete. Per quanto concerne la realtà industriale tale progresso apre la strada a nuove possibilità per le aziende nel soddisfare i propri clienti e offrire loro servizi sempre più efficienti o innovativi. L'azienda *Marelli Motori* di Arzignano (Vicenza), cogliendo questa opportunità, ha avviato il progetto che verrà presentato in questo elaborato.

L'obiettivo primario è quello di remotizzare il monitoraggio e il controllo dei generatori elettrici prodotti dall'azienda (la maggior parte dei quali viene utilizzata per la propulsioni di grandi navi), al fine di abbattere i costi di manutenzione e fornire nuovi servizi ai clienti. Il tutto integrando in un unico sistema software componenti classiche di tipo *client-server*, ma anche applicazioni *mobile* per una maggiore accessibilità ai dati.

Il lavoro, una volta raggiunto lo stadio di prototipo, verrà presentato al *Middle East Electricity* 2015, la più importante manifestazione riguardante la produzione di energia elettrica, che si terrà a *Dubai* agli inizi di Marzo 2015.

1. Introduzione

Il lavoro qui presentato illustra la progettazione e lo sviluppo di un'architettura software, basata sul *modello client-server*, allo scopo di soddisfare le esigenze di un'azienda produttrice di generatori elettrici: la *Marelli Motori* di Arzignano, Vicenza.

> *Marelli Motori S.p.A. ha una lunga tradizione che risale al 1891 anno in cui* ***Ercole Marelli*** *fondò l'azienda. Con oltre 100 anni di esperienza ed eccellenza produttiva, Marelli Motori è riconosciuta come un fornitore leader nei settori della Power Generation, Industriale, Petrolchimico e Marino offrendo un gamma completa di Motori e Generatori in Bassa, Media e Alta tensione. Questi prodotti di qualità sono ottenuti grazie ad un organizzazione di persone preparate che si occupano di vendita, servizio e supporto tecnico in grado di soddisfare gli alti standard richiesti dai nostri clienti. Marelli Motori progetta, produce e vende:*
>
> - *Generatori sincroni in Bassa, Media e Alta tensione*
> - *Generatori asincroni in Bassa, Media e Alta tensione*
> - *Generatori per applicazioni Idroelettriche, UPS, Cogenerazione e Industriali*
> - *Motori asincroni in Bassa, Media e Alta tensione*
> - *Motori per applicazioni in aree potenzialmente esplosive in Bassa, Media e Alta tensione (IP55, IP56, IP65)*
>
> *Questi prodotti sono disponibili in diversi range di potenza da 0,12 fino a 4000 kW per i Motori e da 15 a 9000 kVA per i Generatori. Offre una gamma completa di motori elettrici e generatori ad un grande numero di clienti in tutto il mondo attraverso il quartier generale in Italia e le sue filiali estere nel Regno Unito, Germania, Malesia, Sud Africa e Stati Uniti.*
>
> *Sito web Marelli Motori[1]*, 25/11/2014

In seguito verranno presentati tutti gli aspetti del progetto, ma l'elaborato si concentrerà maggiormente sulle problematiche e le scelte di progettazione della componente *client*.

Il progetto verrà presentato come *proof of concept* alla fiera *Middle East Electricity* 2015 (*MEE*2015) che si svolgerà a *Dubai* agli inizi di Marzo 2015. Si tratta della più importante fiera relativa al settore dell'energia elettrica a livello mondiale.

1.1 Il problema

L'esigenza concreta dell'azienda è quella di migliorare le attività di manutenzione dei propri componenti. Gran parte dei generatori prodotti vengono utilizzati per la propulsione delle navi e sono collocati in tutto il mondo. Ad ogni generatore viene collegata una scheda elettrica che opera da *regolatore* per il settaggio di vari parametri del generatore ed il monitoraggio continuo dei valori di esercizio (tensioni, correnti, frequenze, potenze, flag di stato, ecc.). Si tratta, cioè, di una scheda di interfaccia.

In fase di installazione il regolatore viene configurato per operare secondo determinate grandezze elettriche. Ogni generatore viene sottoposto a manutenzioni periodiche, le quali richiedono la

presenza fisica di un tecnico sul luogo. Questo presenta un doppio inconveniente: innanzitutto la necessità di compiere viaggi, poiché i generatori possono essere situati nei luoghi più disparati, determinando un considerevole spreco di tempo e risorse (ci sono circa 2000 generatori attivi). Il secondo inconveniente è rappresentato dalla difficoltà fisica dell'intervento: spesso i generatori si trovano in spazi angusti e il tecnico ha l'esigenza di collegare un portatile tramite cavo seriale al regolatore per poter effettuare la manutenzione. Le stesse problematiche si ripresentano anche per le manutenzioni straordinarie, ovvero quando si verificano guasti o malfunzionamenti.

L'azienda ha pertanto la necessità di migliorare il proprio servizio di assistenza ai clienti. In tal senso si è optato per la *remotizzazione delle operazioni* di intervento sui generatori, in modo da evitare la presenza fisica del tecnico sul luogo. Per raggiungere questo obiettivo è necessario dotare tutti i regolatori di accesso ad internet.

Nel seguito verranno illustrate tutte le strategie e le scelte effettuate per soddisfare in pieno tali esigenze.

1.2 Contenuti

Nel Capitolo 2 verrà illustrata una panoramica sul regolatore di riferimento usato per tutto il progetto, il *MEC-100*, e sulle funzionalità del software esistente. Verranno esposti tutti i requisiti e le specifiche emersi durante i vari colloqui con l'azienda e sarà presentata la scheda *Hachiko*. Si tratta della scheda scelta per affiancare il regolatore *MEC-100* in modo da fornire accesso ad internet ed ulteriori funzionalità avanzate che verranno discusse in dettaglio nel seguito. In questo capitolo verrà presentato, brevemente, anche il progetto *Yocto*. Infine sarà trattata l'analisi del carico di lavoro del sistema, ovvero un'idicazione della quantità di dati prodotti e immagazzinati nel corso di un anno nell'ipotesi che il sistema sia pienamente operativo.

Nel Capitolo 3 verrà presentata l'intera struttura *hardware* e *software* progettata per soddisfare le richieste e le specifiche analizzate. Si tratta di un'architettura di tipo *client-server* a tre livelli[1] in cui sono presenti diversi client sparsi nel mondo (i regolatori *MEC-100* con le relative schede *Hachiko*) che si collegano, tramite internet, ad un server. Il server interagisce con un *DBMS* (*Database Management System*) che gestisce un database (attualmente centralizzato) per la memorizzazione di vari tipi di dati relativi ai generatori e prelevati dal software client in esecuzione sulla scheda *Hachiko*. Nel sistema è presente un ulteriore componente, ovvero un'applicazione *Android* per *tablet/smartphone* in grado di mostrare all'utente alcuni dati presenti nel database. Verranno, infine, presentati i protocolli di comunicazione client-server e l'organizzazione dei dati inviati nei vari pacchetti.

L'intera architettura software utilizza un database di supporto dove vengono memorizzati diversi dati relativi ai regolatori *MEC-100*, alle schede *Hachiko* e ai clienti. Il Capitolo 4 fornirà una descrizione sintetica ma completa del database, i cui dettagli di progettazione esulano dallo scopo di questa tesi e verranno quindi tralasciati. Tuttavia sarà riportato lo schema *ER* (*Entity-Relationship -- Entità-Relazione*) completo.

Il capitolo 5 presenterà nel dettaglio l'architettura software del client. Verranno illustrate le principali funzionalità offerte e se ne analizzerà la robustezza in diversi scenari di utilizzo. Ulteriori dettagli implementativi e logici saranno inclusi nell'appendice.

Infine nel Captiolo 6 saranno trattati gli sviluppi futuri dell'applicazione client, come ad esempio un'interfaccia grafica per l'utilizzo del pannello *touchscreen* della scheda *Hachiko*, la cifratura dei pacchetti scambiati tra client e server o l'estensione per la gestione di nuovi tipi di regolatori che al momento sono ancora in fase di progettazione.

---

[1] In ingegneria del software, l'espressione "architettura a tre strati" indica una particolare architettura software di tipo multi-tier [...] che prevede la suddivisione dell'applicazione in tre diversi moduli o strati dedicati rispettivamente all'interfaccia utente, alla logica funzionale (*business logic*) e alla gestione dei dati persistenti.

Nel caso in esame l'interfaccia utente è rappresentata dai client *Android*, la logica funzionale è distribuita tra il server che i client *Hachiko* e la gestione dei dati persistenti è affidata al *DBMS* (residente sul server).

2. Analisi dei requisiti

L'analisi dei requisiti, componente fondamentale della progettazione, è durata un paio di mesi e si è svolta tramite diversi colloqui con il reparto tecnico della *Marelli Motori* e con possibili fornitori di schede elettriche da affiancare al regolatore *MEC-100*. Un'analisi iniziale approssimativa può portare a grandi ritardi o errori nella successiva fase di progettazione, quindi si è prestata particolare attenzione a questa fase.

Una volta delineati i punti fondamentali del progetto è stato possibile cominciare ad analizzare in dettaglio il regolatore *MEC-100* e le funzionalità del software esistente. L'obiettivo iniziale è stato quello di riprodurre le funzionalità principali del suddetto software (scritto in `Visual Basic 6.0` esclusivamente per ambiente *Microsoft Windows*) tramite un software scritto in `C++` progettato per l'almbiente *Linux*. La scelta del linguaggio è ricaduta sul `C++` per via delle maggiori possibilità di controllo sull'efficienza del codice rispetto ad un ambiente *Java* e per evitare di essere costretti a scegliere uno specifico ambiente di sviluppo (come era accaduto nel caso del `Visual Basic`). Successivamente si è integrato il software `C++` nel nuovo sistema aggiungendo le varie funzionalità necessarie (come la comunicazione remota al server o l'archiviazione, temporanea, dei dati letti).

Per fornire la connettività remota al regolatore *MEC-100*, tra le varie opzioni a disposizione, si è scelto di utilizzare la scheda *Hachiko* fornita da *Silica*. Infatti la scheda, come si vedrà in questo capitolo, offre tutto quello che serve attualmente all'azienda ed inoltre apre la strada ad eventuali espansioni future in quanto si tratta, di fatto, di un ''mini-computer''.

2.1 Il regolatore *MEC-100*

Marelli Motori produce generatori elettrici di vario tipo e li affianca a dei regolatori (prodotti internamente) per il monitoraggio continuo ed il controllo di funzionamento. Esistono vari modelli di generatori e regolatori: nelle immagini seguenti si vede un esempio di generatore e il regolatore *MEC-100* usato come riferimento per questo progetto.

Figura 2.1: Un generatore *Marelli*.

Figura 2.2: Il regolatore *MEC-100*.

Il regolatore *MEC-100* è in grado di rilevare sia i dati di esercizio del generatore, sia i parametri di configurazione dello stesso [2]. La scheda lavora con *words* a 16 bit e fornisce solo un'interfaccia seriale `RS-232` a 9 pin non alimentata tramite la quale è possibile leggere (e scrivere, nel caso dei parametri) questi dati. La comunicazione con il regolatore avviene tramite l'invio di richieste codificate da 7 *word*. La risposta contiene anch'essa un *header* da 6 *words*, n *words* per il *payload* e 1 *word* per il controllo di ridondanza ciclica *CRC* (*Cyclic Redundancy Check*).

Figura 2.3: Formato di un pacchetto dati di richiesta/risposta MEC-100.

I dati su cui il software può agire sono di tre tipi:

1. **parametri statici** (sola lettura). Ad esempio il codice identificativo della scheda e la versione del *firmware*;

2. **valori di esercizio** (sola lettura). Sono le grandezze elettriche che rappresentano i valori di esercizio della scheda oppure dei *flag booleani* che indicano il verificarsi di anomalie;

3. **parametri configurabili** (lettura/scrittura). Sono i vari settaggi della scheda, ovvero quei parametri di configurazione che il tecnico può regolare.

La scheda viene inizialmente configurata dall'azienda in base al generatore sul quale dev'essere installata. I parametri di configurazione riguardano grandezze elettriche che determinano il funzionamento del generatore, come frequenze, tensioni, correnti, ecc. Un tecnico, al momento dell'installazione della scheda, ne può modificare la configurazione di fabbrica, operazione che viene effettuata collegando un computer attraverso un cavo seriale ed utilizzando il software esistente.

Ogni coppia regolatore-generatore è sottoposta a controlli periodici di manutenzione. Il tecnico, in questo caso, si collega al regolatore per controllare la presenza di situazioni anomale verificatesi durante il periodo di normale funzionamento. Le anomalie sono rappresentate da flag chiamati *Alarm* del tipo `ON`/`OFF`. Se necessario il tecnico può modificare i valori di funzionamento settando dei nuovi parametri di configurazione.

2.2 Software client esistente

Il software esistente è stato scritto in `Visual Basic 6.0` e funziona solo su piattaforme *Microsoft Windows*. Il programma, una volta stabilita una connessione con una scheda MEC-100 va a leggerne i parametri di configurazione e poi legge in *polling*[1] (a intervalli di 1 secondo) i valori di esercizio. Esso offre una visualizzazione grafica compatta dei dati, compresi gli eventuali allarmi. Inoltre offre la possibilità di visualizzare un grafico dell'andamento di alcune grandezze di esercizio rilevate.

Tale programma presenta un limite fondamentale, ovvero la necessità del tecnico di trovarsi fisicamente sul luogo per utilizzarlo. Infatti non è previsto alcun tipo di remotizzazione e questo è uno dei motivi che ha portato allo sviluppo del progetto.

[1] Per *polling* si intende, in informatica, la lettura periodica dello stato di un componente esterno utilizzando un programma client in maniera sincrona. Qui il termine è usato in maniera leggermente impropria, in quanto il regolatore e la scheda *Hachiko* non hanno bisogno di sincronizzarsi (in senso stretto) per lo scambio dei dati.

2.3 Scheda *Hachiko*

Per poter remotizzare il monitoraggio dei dati di esercizio il *MEC-100* deve essere dotato di connettività Internet. Essendo la porta seriale l'unica interfaccia disponibile in questa scheda è necessario utilizzare un terzo dispositivo in grado di comunicare tramite porta seriale e di collegarsi alla rete.

Nella Figura 2.4 è riportato lo schema di collegamento tra i dispositivi coinvolti. Il generatore è collegato al *MEC-100* per il normale funzionamento. Il regolatore viene ora collegato ad un dispositivo (rappresentato da una classica *black box*) che sia in grado di collegarsi ad internet.

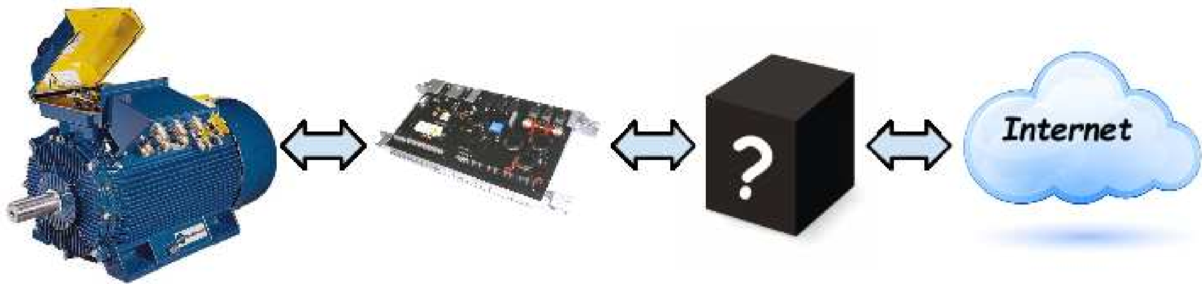


Figura 2.4: Connessione tra i dispositivi coinvolti, dal generatore alla rete internet.

La scheda scelta per fare da tramite tra il regolatore e la rete internet è stata la scheda *Hachiko* prodotta da *Silica* [3]. Questa scelta è giustificata dal supporto esteso che i partner di *Marelli* possono fornire e dal fatto che l'*Hachiko* è paragonabile ad un computer per funzionalità, tanto che utilizza un sistema operativo minimale *Linux* basato sul progetto *Yocto*. Ciò permette di sviluppare un software molto simile a quello che si userebbe per un normale computer, evitando gran parte degli inconvenienti tipici dei sistemi *embedded* e riducendo i tempi di sviluppo. Ad esempio, per il *driver* seriale, è bastato configurare correttamente quello presente nel sistema *Linux* tramite la libreria `termios.h` senza doverne implementare uno *ad-hoc*.

Altri punti a favore della scheda *Hachiko* sono la possibilità di installare RAM aggiuntiva fino a 256 MB e il suo processore *Cortex-A9* multicore.

Inoltre la scheda *Hachiko* viene fornita completa di tutti i componenti concordati, che sono saldati sul circuito stampato rendendola più robusta alle sollecitazioni esterne dovute alla difficile condizione ambientale in cui il regolatore opera.

La scheda può essere inoltre dotata di uno schermo *LCD* (*Liquid-Crystal Display*) di tipo *touchscreen* da 4.3 pollici (chiamato *Pengwyn*[2]) e di diversi altri canali di comunicazione che potrebbero essere utilizzati per sviluppi futuri. In Figura 2.5 sono riportati la scheda *Hachiko* e il display *Pengwyn*.

Altre schede prese in esame, pur essendo più economiche, non offrono le stesse funzionalità o lo stesso livello di supporto della scheda *Hachiko*.

2.3.1 Il progetto *Yocto*

Si tratta di un progetto *open source* [4] che mira a fornire *template*, metodi e strumenti per facilitare la creazione di sistemi basati su *Linux* per prodotti *embedded* indipendentemente dall'architettura hardware utilizzata: sono infatti presenti profili di *build* per varie piattaforme come *ARM*, *x86*, *PPC*, ed altre.

In particolare permette di compilare un'immagine *Linux* utilizzabile nella scheda *Hachiko* che occupa meno di 10 MB di RAM. Il tutto tramite un semplice comando `bitbake` che sfrutta file di configurazione liberamente accessibili sul web.

[2] Formato $16:9$, risoluzione massima $480x272$, $65k$ colori, *touchscreen* di tipo resistivo.

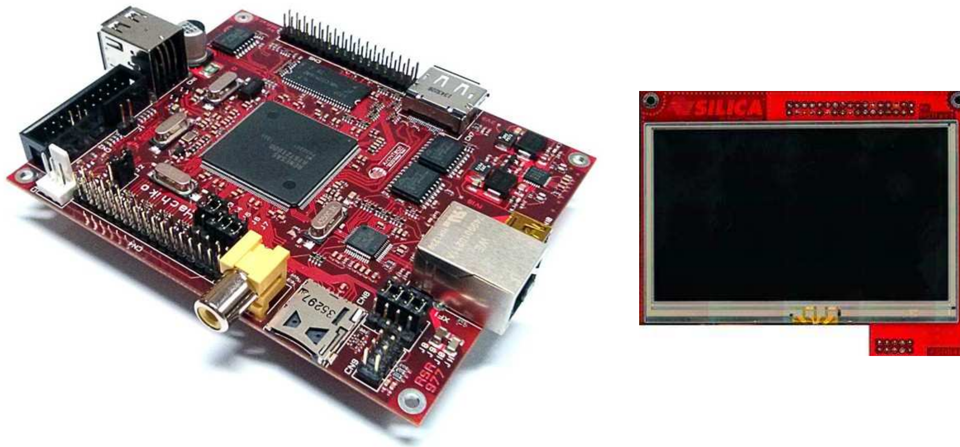


Figura 2.5: La scheda *Hachiko* e il display *Pengwyn*.

L'installazione è ulteriormente configurabile per includere i driver ed i pacchetti extra necessari, il tutto anche attraverso una comoda interfaccia grafica chiamata *Hob*.

Infine, il progetto *Yocto* mette a disposizione un *plug-in* per *Eclipse* per facilitare ulteriormente lo sviluppo di software per piattaforme *embedded* producendo automaticamente tutti i file di configurazione necessari per le compilazioni *CMake* o *Autotools*.

2.4 Analisi dei requisiti

Dai colloqui aziendali sono emerse 3 esigenze principali che il progetto dovrà soddisfare:

1. migliorare e remotizzare la manutenzione del generatore/regolatore e l'assistenza al cliente;
2. rilevare statistiche di funzionamento al fine di migliorare la fase di progettazione e/o produzione di nuovi generatori o regolatori;
3. fornire un nuovo servizio ai clienti con il quale possano consultare lo stato dei propri regolatori.

Per far fronte a tutte queste esigenze si è scelto di progettare un sistema software composto da 4 componenti:

1. un'applicazione client installata sulle schede *Hachiko* per il rilevamento dei dati;
2. un database di supporto per la memorizzazione di tutti i dati rilevati;
3. un'applicazione server che interagisca con il database per soddisfare tutte le richieste;
4. un'applicazione Android per fornire un servizio di consultazione dei dati ai clienti.

Per i dettagli sull'architettura software si rimanda al Capitolo 3.

Gli applicativi inizialmente sono stati sviluppati per uno specifico modello di regolatore, il *MEC-100*. Tuttavia in futuro potranno essere utilizzati altri regolatori. L'applicazione client dovrà essere in parte adattata per supportare il nuovo hardware, ma il database è stato progettato fin da ora per essere facilmente esteso ai nuovi regolatori.

Diverse versioni di *MEC-100* sono state commercializzate finora ed il protocollo di comunicazione tramite porta seriale è cambiato nell'arco del tempo. Il progetto prende come riferimento lo standard più recente, ma potrebbero essere necessari alcuni aggiustamenti software per rendere retro-compatibili le schede *Hachiko*.

Nuove funzionalità

Il sistema progettato introduce diverse *nuove* funzionalità utili a soddisfare le richieste di cui sopra.

Storico dei dati di esercizio: i valori letti dalla scheda *Hachiko* vengono opportunamente archiviati aggregando più dati, con la possibilità di esplorare le letture a grana sempre più fine, fino ad arrivare alla singola lettura. Un solo *MEC-100* genera infatti 86400 letture giornaliere ed è quindi impensabile (e anche relativamente inutile) renderle tutte sempre disponibili nel database in forma di *tuple* nelle tabelle. Tuttavia può essere necessario recuperare abbastanza velocemente un determinato dato.

Storico degli interventi di assistenza: per ogni regolatore è possibile risalire all'elenco delle operazioni di manutenzione che vi sono state svolte. Per ogni assistenza vengono registrate tutte le attività che il tecnico svolge durante la manutenzione. Interventi futuri possono essere migliorati controllando le operazioni svolte nelle assistenze precedenti.

Configurazione remota: alcune configurazioni devono poter essere fatte da remoto, per limitare l'intervento dei tecnici sul luogo.

Statistiche di funzionamento: è prevista la possibilità di fornire statistiche di vario tipo riguardanti i regolatori. Tramite i dati memorizzati all'interno del database si possono effettuare analisi circa il funzionamento, i guasti e gli interventi di manutenzione.

Manutenzione

L'assistenza sul posto non è eliminabile al 100% e nei casi in cui sia necessario l'intervento diretto la scheda *Hachiko* fornirà un'interfaccia grafica tramite il pannello *touchscreen* con la quale il tecnico avrà a disposizione le stesse funzionalità di manutenzione presenti nel software che si utilizzava da computer.

L'assistenza/manutenzione prevede tre tipologie di intervento:

- *commissioning*. Si tratta dell'installazione iniziale del regolatore sul generatore del cliente. I settaggi sono pre-impostati dalla sala prove *Marelli* secondo le esigenze del cliente. Queste operazioni sono effettuate tramite software e vengono generati dei file di documentazione e delle fotografie archiviate in un database *Access*. Tali informazioni andranno in futuro integrate nel nuovo sistema;
- *guasto*. Riguarda l'intervento sul posto con l'eventuale sostituzione del regolatore guasto. Se il regolatore potrà essere riparato, verrà poi riconsegnato al cliente che lo utilizzerà come scorta. In questo caso il cliente disporrà di due regolatori per un solo generatore;
- la *configurazione remota* che in futuro sarà resa disponibile dal nuovo sistema.

Altri vincoli emersi durante i colloqui

Durante la raccolta di informazioni sono emersi anche ulteriori vincoli, di vario tipo, qui brevemente riportati:

- il database deve prevedere la possibilità di inserire un numero variabile di foto e documenti a supporto delle *entry* relative agli interventi di assistenza;
- per ogni sessione di assistenza solamente un tecnico interviene sul regolatore. Questi effettua alcune configurazioni della scheda procedendo per tentativi fino ad ottenere la configurazione ottimale. Tutte le modifiche intermedie apportate devono essere memorizzate e si deve prevedere una modalità *maintenance* della scheda *Hachiko* durante la quale i dati relativi al normale funzionamento non vengono ''sporcati'' dalle modifiche relative alla manutenzione;

- i tecnici devono essere abilitati alla manutenzione locale del regolatore (attraverso l'interfaccia grafica della scheda *Hachiko*) mediante un controllo di accesso. Tale controllo viene fatto in remoto tramite il server di autenticazione. Nel caso in cui la connessione Internet non sia disponibile, l'autenticazione viene fatta localmente, controllando tra gli utenti *Linux* registrati nella scheda *Hachiko*. L'applicazione deve prevedere un controllo relativamente complesso sui valori da assegnare ai parametri per ridurre al minimo il rischio di incidenti;
- il data base dovrà memorizzare anche dati riguardanti i generatori, i quali hanno parametri e informazioni a sé stanti che per il momento non sono stati considerati;
- ogni regolatore è di proprietà di un determinato cliente, è installato su un determinato motore ed è identificato univocamente. Nella maggior parte dei casi un singolo *MEC-100* viene montato su di un unico generatore, ma sono comunque possibili sostituzioni, per cui è necessario mantenere uno storico delle associazioni tra generatore e *MEC-100*;
- un regolatore può essere venduto al cliente come componente singolo da associare successivamente al generatore, oppure può essere fornito direttamente assieme al generatore;
- le schede *Hachiko* possono venir smontate dal *MEC-100* su cui erano inizialmente installate. Un protocollo di tipo *heart-beat* è necessario per controllare la continua connessione tra scheda *Hachiko* e regolatore. Inoltre la connessione alla rete Internet non è sempre disponibile: molti generatori sono installati su navi e potrebbero rimanere senza collegamento per molto tempo. La scheda *Hachiko* rileva comunque i dati con continuità, ma dovrà memorizzarli temporaneamente nella memoria secondaria, per poi inviarli appena possibile;
- attualmente i vari clienti della *Marelli* possiedono complessivamente circa 2000 regolatori, ciò significa che il sistema registrerà circa 170 milioni di letture al giorno. Di seguito è presente un'analisi più dettagliata del carico di lavoro gestito dal sistema.

2.5 Analisi del carico di lavoro nel sistema

L'analisi del carico di lavoro si basa sulle seguenti ipotesi (emerse durante i colloqui):

- è necessario considerare che nel sistema opereranno circa 2000 client *Hachiko*;
- ciascun client legge i dati in modo continuativo ogni secondo;
- si vuole una stima sul lungo periodo (un anno).

Per come è stata progettata la comunicazione tra client e server c'è un discreto spreco di risorse: i file di *log* compressi che vengono inviati risultano eccessivamente ridondanti nelle informazioni. Questa scelta era stata operata per agevolare un eventuale operatore umano che dovesse andare a leggere uno di questi file di *log*. Inoltre l'implementazione attuale riduce e semplifica l'*overhead* del server in fase di *drill-down* dei dati. Pertanto le considerazioni seguenti si basano su questo tipo di file e sono eccessivamente pessimistiche. Tuttavia sono quello che il sistema deve gestire allo stato attuale. Verrà comunque presentata anche un'analisi nel caso di ottimizzazione dei file di *log*, la quale, però, è una miglioria che potrebbe essere implementata in futuro e allo stato attuale non è disponibile.

Inoltre dall'analisi vengono tralasciati tutti i "dati minori", come gli *header* dei pacchetti o le informazioni generate dalle assistenze, ma anche i valori aggregati spediti a fine giornata. Si tratta, infatti, al massimo di pochi KB, mentre la maggior parte dei dati sarà composta dai file di *log* generati dai client *Hachiko* durante il funzionamento normale. Anche i file di *log* delle assistenze possono essere trascurati, in quanto queste sono relativamente "rare", nel senso che generano un numero di file di *log* trascurabile rispetto a quelli relativi ai dati di esercizio rilevati.

2.5.1 Stato attuale

Allo stato attuale ciascuna scheda *Hachiko* genera, *giornalmente*, dei file di *log* testuali che occupano circa 38 MB. Questi, una volta compressi, arrivano ad occupare poco meno di 12 MB in media. Ciò significa che l'intero sistema produce quotidianamente circa 24 GB di file di *log*. Il che significa che in un anno di esercizio la mole di dati da memorizzare è compresa tra gli 8 e i 9 TB (10^{12} B).

In realtà l'azienda ha confermato che i dati più vecchi possono anche essere cancellati, quindi dal punto di vista del server non si tratta di una mole di dati impossibile da gestire. Tuttavia per i client potrebbe essere un inconveniente dover spedire ogni giorno 12 MB di dati, visto che potrebbero trovarsi in situazioni in cui la connessione alla rete è di scarsa qualità oppure assente.

2.5.2 Ottimizzazione

Modificando sia il software client che quello server è possibile ridurre notevolmente la quantità di dati trasferiti (e quindi memorizzati) al costo di accettare un maggior *overhead* da parte dell'applicazione server per la gestione dei file di *log*.

In questo scenario ottimizzato ciascun client *Hachiko* potrebbe produrre file di *log* delle dimensioni di circa 21 MB, compressi in 9 MB. Il totale giornaliero sul server diventerebbe di 18 GB e il totale annuale sarebbe compreso tra i 6 e i 7 TB.

In altre parole si avrebbe un risparmio di risorse del 25% circa.

Se il carico di lavoro risultasse in ogni caso troppo elevato quando il sistema sarà a regime, sarà comunque possibile distribuire il database (e anche il server) su più calcolatori collegati in rete in modo da alleggerire il carico su ciascuna macchina.

É possibile ottimizzare ulteriormente il consumo di risorse utilizzando *file binari* anziché testuali, ma con un ulteriore aumento dell'*overhead* nel server. Si veda la Sezione 6.5 per i dettagli.

3. Architettura del sistema

Il sistema è stato progettato sul modello *client-server* a più livelli. La base del sistema è rappresentata dal database di supporto, il quale interagisce direttamente con l'applicazione server. Le applicazioni client (*Hachiko* e *Android*) interagiscono con il server, e non hanno accesso diretto al database.

L'architettura software è schematicamente rappresentata dalla seguente Figura 3.1.

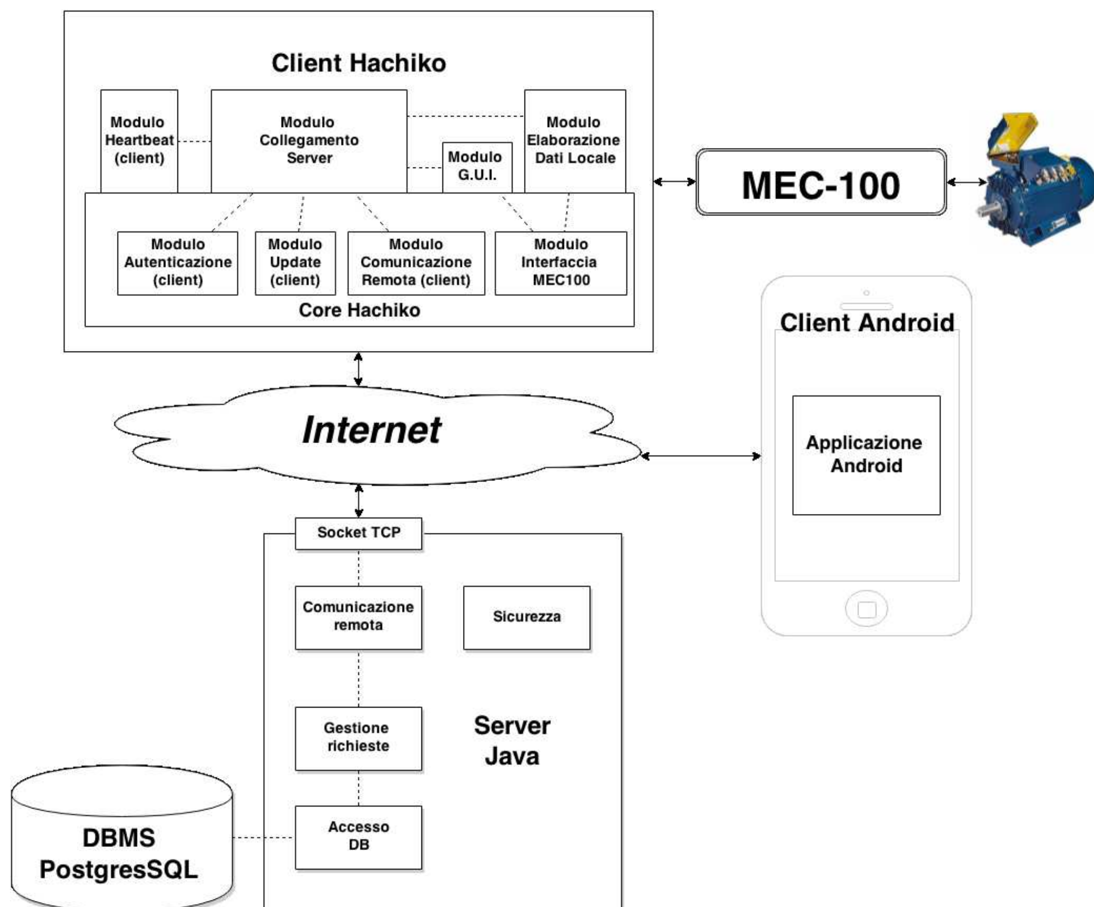


Figura 3.1: Schema dell'architettura software sviluppata. Si tratta di un sistema *client-server* con un database di supporto e un *front-end* costituito dal client *Android*.

Nell'architettura a tre livelli classica l'interfaccia verso l'utente è solitamente costituita da un web server, con le sue eventuali pagine statiche. In questo progetto, invece, il *front-end* è costituito dall'applicazione *Android*. Per il momento non è previsto l'uso di un web server tramite il quale consultare le informazioni contenute nel database via *browser*, ma si tratta di un'estensione che potrà essere realizzata facilmente in futuro.

Il client *Hachiko* invia al server varie informazioni, illustrate in dettaglio nella Sezione 3.2, ricavando i dati necessari dal regolatore *MEC-100*. La comunicazione avviene tramite *socket TCP* (*Transmission Control Protocol*), ciò garantisce l'ordinamento, l'integrità e l'effettiva ricezione di tutti i pacchetti. Ciascuna connessione viene utilizzata per l'invio di un solo messaggio e viene chiusa subito dopo. Il server, quindi, non deve tenere aperte delle connessioni indefinitamente. Molte delle richieste del client non prevedono una risposta da parte del server. In questi casi la chiusura della connessione va interpretata come l'effettiva ricezione corretta dei dati.

L'altro tipo di client presente nel sistema è il client *Android*. Esso invia al server delle richieste di tipo diverso, in quanto si tratta per la maggior parte di richieste di accesso ai dati presenti nel database. Anche il client *Android* utilizza connessioni su *socket* per la comunicazione, ma, a differenza del client *Hachiko*, le connessioni *Android* sono già crittografate utilizzando il protocollo *SSL*.

Quando il server riceve un messaggio dal client *Hachiko* lo analizza e, in base al tipo di richiesta, effettua le varie operazioni necessarie sul database: nella maggior parte dei casi si tratta di inserimenti nelle tabelle opportune. Quando, invece, le richieste arrivano da client *Android* le *query* sono di tipo `SELECT`, ma può essere necessario elaborare i dati letti dal database. Per esempio se vengono richieste letture dei valori di esercizio di un regolatore, le quali erano state archiviate in forma aggregata, è necessario eseguire le opportune operazioni di *drill-down* al fine di ricavare i dati richiesti. Nel caso delle manutenzioni remote, sarà il server ad iniziare una comunicazione con un client. A tale scopo viene usato un protocollo di tipo *heart-beat* tramite il quale il server riesce a tener traccia dell'indirizzo *IP* di ciascun client (i client infatti sono da considerarsi mobili e il loro indirizzo *IP* può variare nel tempo).

Il *Database Management System - DBMS* si occupa dell'intera gestione dei dati spediti dal client *Hachiko* e degli archivi aggregati.

Sia il server che il database sono centralizzati, ma non sarebbe difficile renderli distribuiti qualora il carico di lavoro del sistema diventasse eccessivo per un sistema centralizzato.

3.1 Organizzazione del software

L'applicazione server è stata sviluppata in `Java`, mentre il client *Hachiko* è stato scritto in `C++` per ottimizzare l'uso delle risorse disponibili (che sono certamente più limitate rispetto ad un server). Anche il client *Android* è stato sviluppato in `Java`.

L'intero software (sia client che server) è stato progettato e implementato in modo modulare, come rappresentato in Figura 3.1. Questo permetterà, in futuro, di estenderlo per supportare nuovi regolatori o di modificarne facilmente le funzionalità.

Nello specifico (Figura 3.2), il programma server è composto da quattro moduli principali: tre di questi implementano le funzionalità base e sono i moduli *Comunicazione Remota*, *Gestione delle richieste* e *Accesso al database*. Il quarto modulo si occupa di tutti gli aspetti relativi alla sicurezza come la gestione delle chiavi di cifratura.

Il componente di *Comunicazione remota* si occupa di gestire le connessioni da / verso i client *Hachiko* e *Android*. Le richieste che arrivano vengono quindi passate al modulo di *Gestione delle richieste* che si occuperà di smistarle in base alla tipologia e comunicherà con il modulo di *Accesso al database* per le interazioni con il *DBMS*.

Il client *Android*, essendo al momento piuttosto semplice, è stato realizzato come un blocco unico. Più interessante è invece il client *Hachiko* (Figura 3.3) i cui dettagli saranno esposti dettagliatamente nel Capitolo 5. Ma per offrirne una rapida ed essenziale panoramica, esso è composto da alcuni

moduli che vanno a comporre il *core* del client e si occupano dell'interazione con il regolatore, della comunicazione con il server e dell'aggiornamento del software, coadiuvati da altri moduli di livello "più alto" che utilizzano le funzionalità dei moduli *core* per implementare logiche di funzionamento più evolute.

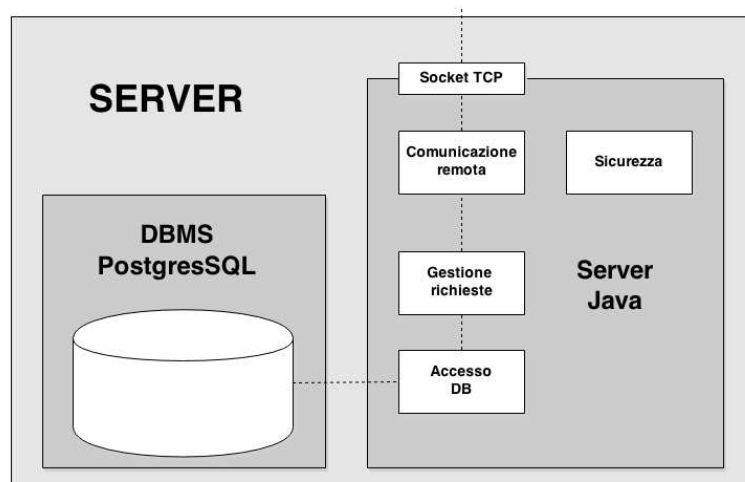


Figura 3.2: Il server, nel suo complesso, è formato da un'applicazione `Java` e da un *DBMS*. Quattro moduli compongono il programma e garantiscono la comunicazione con i client, lo smistamento delle varie richieste, l'accesso al database e alcune funzionalità di sicurezza di complemento.

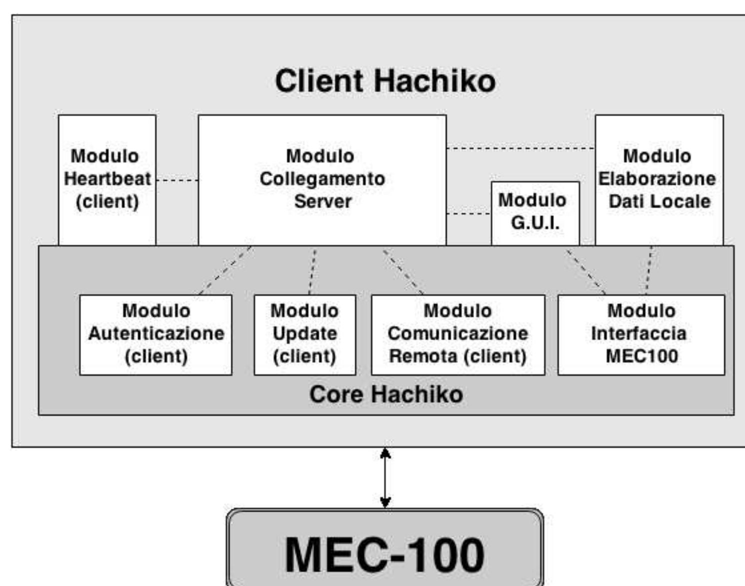


Figura 3.3: Il client *Hachiko* è composto da moduli di due "livelli": al livello *core* i moduli offrono funzionalità di basso livello e si possono immaginare come dei *driver*. Al livello superiore, invece, i componenti software realizzano tutte le funzionalità richieste al client, utilizzando i servizi messi a disposizione dal *core*.

Ad un livello superiore sono poi presenti altri moduli software che basano il proprio funzionamento sui servizi messi a disposizione dal *core*. Questi moduli di ''alto livello'' si occupano delle richieste al server (la creazione dei pacchetti), della raccolta dei dati dal regolatore *MEC-100* secondo opportune modalità non banali e dell'interfaccia utente per il display *touchscreen*.

3.2 Interazioni tra il client *Hachiko* e il server

É stato sviluppato un protocollo *ad hoc* per la comunicazione tra client (*Hachiko*) e server che si affida al protocollo *TCP* per lo scambio dei pacchetti. I pacchetti generati dall'applicazione hanno un formato ben definito, rappresentato nella figura seguente:

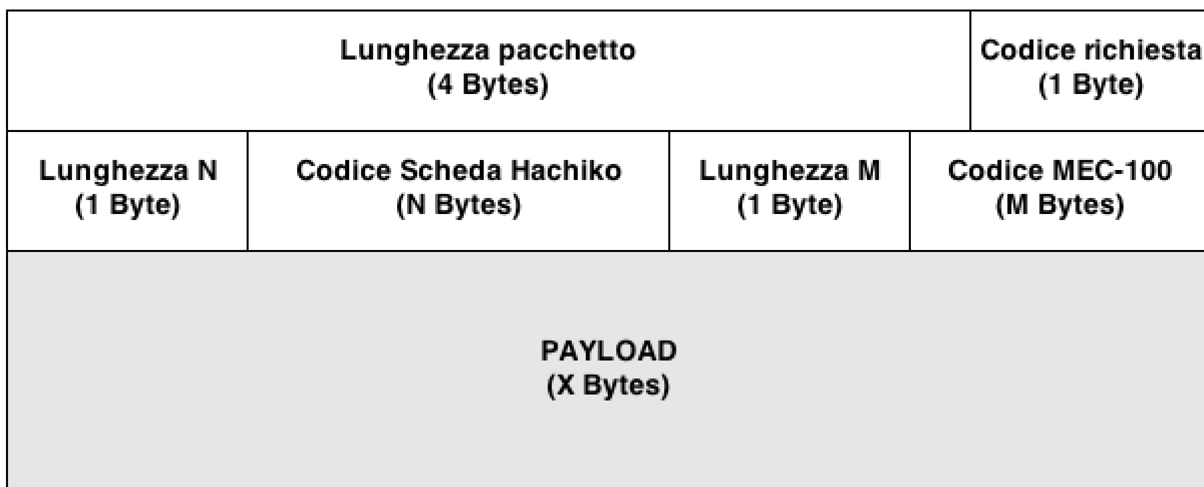


Figura 3.4: Formato dei pacchetti di comunicazione usati tra client e server.

Di seguito sono riportate le specifiche dei vari campi del pacchetto.

- **Lunghezza pacchetto [4 B]**: un *unsigned integer* contenente il numero *totale* di bytes. Ovvero i bytes di *header* e *payload*. Il limite massimo alla dimesione dei pacchetti è impostato a 2 GB (2^{31}) perché il primo bit del pacchetto viene sempre impostato a 0 in modo da permettere future estensioni del protocollo. La lunghezza *incude* i 4 bytes di questo campo.
- **Codice richiesta [1 B]**: identifica il tipo di richiesta contenuta nel pacchetto. I valori possibili sono compresi tra 0 e 127 per lasciare il primo bit impostato a 0 per lo stesso motivo del campo precedente. Attualmente sono codificati 7 tipi di richiesta:
  - **codice 0 - AUTHENTICATION**: autenticazione della coppia *Hachiko* / *MEC-100* nel sistema;
  - **codice 1 - BOOTSTRAP**: prima connessione *Hachiko* $\leftrightarrow$ *MEC-100* oppure avvio della scheda *Hachiko*;
  - **codice 2 - VALUES**: inserimento nel database dei valori di esercizio letti in *polling*;
  - **codice 3 - ALARM**: inserimento nel database di valori di esercizio contenenti flag di allarme attivi;
  - **codice 4 - HEART-BEAT**: aggiornamento *heart-beat*;
  - **codice 5 - LOGIN**: autenticazione di un tecnico (per le sessioni di manutenzione);
  - **codice 6 - MAINTENANCE**: inserimento nel database di dati relativi alle manutenzioni.

- **Lunghezza N [1 B]**: indica la dimensione, in bytes, del campo seguente (la lunghezza dell'ID *Hachiko*).
- **Codice scheda *Hachiko* [N B]**: codice identificativo della scheda *Hachiko*.
- **Lunghezza M [1 B]**: indica la dimensione, in bytes, del campo seguente (la lunghezza dell'ID *MEC-100*).
- **Codice scheda *MEC-100* [M B]**: codice identificativo della scheda *MEC-100*.
- **PAYLOAD [X B]**: il messaggio effettivamente scambiato tra client e server.

3.2.1 Dettagli dei vari tipi di pacchetti

Codice 0 - AUTHENTICATION: è il primo messaggio che la scheda *Hachiko* manda al server al momento dell'attivazione. Scopo di tale messaggio è quello di tener traccia delle associazioni tra schede *Hachiko* e regolatori *MEC-100* in modo da risalire a tutte le sostituzioni che possono aver avuto luogo nel tempo. Inoltre, in futuro, questo tipo di messaggio verrà utilizzato per lo scambio delle informazioni necessarie a cifrare tutti i successivi messaggi (si veda la Sezione 6.3).

Codice 1 - BOOTSTRAP: questo messaggio viene inviato una sola volta all'accensione della scheda, dopo il messaggio di autenticazione. Il messaggio contiene i parametri di configurazione del *MEC-100* appena rilevati. Lo scopo della comunicazione è di poter tener traccia nel database di eventuali modifiche "non autorizzate" alla configurazione del regolatore, ovvero modifiche effettuate senza utilizzare il software del client *Hachiko* (ad esempio utilizzando il vecchio software o altri metodi). poiché sono comunque possibili sessioni di manutenzione *offline*, cioè senza connessione ad internet, il client dovrà essere in grado di gestire questi scenari "anomali".

Codice 2 - VALUES: l'aggiornamento dei dati di esercizio avviene comunicando al server i valori aggregati di diverse letture ed un file contenente tutti i valori puntuali. L'operazione ha come scopo quello di ridurre il traffico dati nella rete: i valori puntuali rilevati ogni secondo vengono memorizzati localmente e vengono poi spediti in blocco ad intervalli molto più ampi.

Codice 3 - ALARM: messaggio speciale di aggiornamento dei dati di esercizio. In presenza di situazioni anomale (flag di allarme settati su ON) l'aggregazione dei valori ordinari viene interrotta e i dati vengono spediti al server. Prima, però, viene spedita la singola lettura dove sono stati rilevati gli allarmi attivi. In questo modo, se la scheda *Hachiko* è collegata ad internet, sul database sono "immediatamente"[1] disponibili le informazioni più critiche.

Codice 4 - HEART-BEAT: ad intervalli regolari di un minuto il client *Hachiko* invia un pacchetto *heart-beat* al server, il quale è in grado di memorizzare l'IP del client e il *timestamp* di ultima ricezione. Questo permette di avere un indirizzo per contattare il client che, se il *timestamp* è recente, sarà molto probabilmente valido.

Codice 5 - LOGIN: l'operazione di *login* è sempre necessaria prima di poter effettuare una manutenzione attraverso la scheda *Hachiko*. Il pacchetto di login viene inviato quando un tecnico tenta l'autenticazione tramite l'interfaccia grafica della scheda ed è disponibile una connessione ad internet. In caso di mancanza di connettività il *login* è comunque possibile, ma con modalità differenti.

Codice 6 - MAINTENANCE: al termine di ciascuna sessione di assistenza effettuata in loco il client invia al server la nuova configurazione dei parametri e un file contenente tutte le operazioni intermedie svolte sulla scheda. Vengono inoltre inviati i valori di esercizio rilevati durante la manutenzione che verranno tenuti separati, nel database, rispetto ai valori ordinari aggregati. Al momento non è ancora stato definito dettagliatamente come avverrà la manutenzione da remoto, quindi potrebbe essere necessario definire un nuovo tipo di messaggio da utilizzare appositamente per le assistenze remote.

[1] Non è possibile fornire nessuna garanzia sull'effettiva tempestività della comunicazione a causa della natura mobile dei client *Hachiko* e della latenza nella comunicazione.

Payload dei vari pacchetti

I pacchetti **AUTHENTICATION** e **HEART-BEAT** sono gli unici che prevedono un payload vuoto. Per i pacchetti **BOOTSTRAP** e **ALARM** il payload è composto solo da un file di testo (compresso) contenente: nel primo caso i parametri di configurazione del *MEC-100*, mentre nel secondo i dati di esercizio contenenti allarmi. Nel seguito sono illustrati nel dettaglio i payload dei rimanenti pacchetti, che sono più complessi.

Per il pacchetto **VALUES** il payload ha il seguente formato:

Numero di parametri (1 byte)	
Codice parametro #1 (6 bytes)	Valore aggregato #1 (4 bytes)
...	...
Codice parametro #n (6 bytes)	Valore aggreagato #n (4 bytes)
File di testo (compresso) con tutti i valori puntuali rilevati	

Figura 3.5: Formato del payload del pacchetto VALUES.

- **Numero di parametri [1 B]**: numero di parametri da aggiornare. Questo numero indica quante coppie *CODICE_PARAMETRO-VALORE_AGGREGATO* seguono.
- **Codice parametro #x [6 B]**: contiene la stringa che identifica il parametro nel database e nel file di *log*. Ogni carattere è rappresentato da un byte in codifica `ASCII`.
- **Valore aggregato #x [4 B]**: valore aggregato (media) per l'x-esimo parametro. I quattro bytes possono rappresentare un parametro *FLOAT*, *INTEGER* o *BOOLEAN*.
- **File di testo**: file di testo (compresso) contenente tutti i valori di esercizio letti dall'ultimo invio.

Il payload del pacchetto **LOGIN** è rappresentato nella figura seguente:

Lunghezza username - N (1 byte)	Username (N bytes)
Lunghezza password - M (1 byte)	Password (M bytes)

Figura 3.6: Formato del payload del pacchetto LOGIN.

- **Lunghezza username - N [1 B]**: numero di caratteri dello *username* da inviare.
- **Username [N B]**: contiene il nome utente da utilizzare per l'autenticazione.
- **Lunghezza password - M [1 B]**: numero di caratteri della *password* da inviare.

- **Password [M B]**: contiene la password da utilizzare per l'autenticazione.

Infine il pacchetto **MAINTENANCE** ha il formato:

Lunghezza username - N (1 byte)	Username (N bytes)
Lunghezza file configurazione finale (4 bytes)	File configurazione finale
Lunghezza file log di manutenzione (4 bytes)	File log di manutenzione
Lunghezza file valori esercizio MEC-100 (4 bytes)	File valori esercizio MEC-100

Figura 3.7: Formato del payload del pacchetto MAINTENANCE.

- **Lunghezza username - N [1 B]**: numero di caratteri del nome utente del tecnico che ha svolto la manutenzione.
- **Username [N B]**: nome utente del tecnico.
- **Lunghezza file configurazione finale [4 B]**: lunghezza (in bytes) del primo file.
- **File configurazione finale**: file di testo (compresso) contenente i parametri di configuraione finali assegnati al *MEC-100* alla fine della sessione di manutenzione.
- **Lunghezza file log di manutenzione [4 B]**: lunghezza (in bytes) del secondo file.
- **File log di manutenzione**: file di *log* con la registrazione di tutte le variazioni dei parametri di configurazione avvenute durante la sessione di manutenzione.
- **Lunghezza file valori esercizio *MEC-100* [4 B]**: lunghezza (in bytes) del terzo file.
- **File valori esercizio *MEC-100***: file di *log* con la registrazione di tutte le letture dei dati di esercizio del *MEC-100* acquisite durante la fase di manutenzione.

4. Il database di supporto

Il database a supporto dell'applicazione è stato progettato con cura, tenendo conto delle varie esigenze dell'azienda e delle funzionalità necessarie per il software. La progettazione ha seguito uno schema classico le cui fasi principali sono state: raccolta delle informazioni, progettazione concettuale volta ad ottenere uno schema relazionale *ER* (*Entity-Relationship*), la successiva traduzione in uno schema logico e infine l'implementazione vera e propria utilizzando un *DBMS*.

Tra i vantaggi nell'usare un modello *ER* vi è anche il fatto che si tratta di un formalismo semplice da capire anche per i ''non addetti'' ai lavori, il che ha permesso di ottenere un costante *feedback* dalle varie funzioni aziendali interessate direttamente durante la fase di progettazione.

I dettagli specifici di ciascuna fase esulano dallo scopo di questo elaborato, quindi di seguito verrà presentato principalmente lo schema *ER* finale con una spiegazione concisa dei componenti fondamentali e un'analisi relativa alla normalizzazione dello schema.

4.1 Schema *ER* - *Entity-Relationship*

Lo schema *ER* completo è riportato in Figura 4.1. Dallo schema sono stati omessi tutti gli attributi delle entità per migliorarne la leggibilità. Nelle sezioni seguenti verranno descritti in dettaglio i componenti fondamentali dello schema.

4.1.1 Modellizzazione dei dispositivi hardware

Il ''sotto-schema'' di Figura 4.2 contiene le entità relative ai dispositivi hardware del sistema. L'entità principale dello schema è l'entità ***Scheda*** che rappresenta i regolatori *MEC-100*. Come chiave primaria si è scelto di utilizzare il numero seriale del regolatore, che è univoco. L'entità ***Scheda*** non va confusa con ***Regolatore***: quest'ultimo rappresenta il *modello* di *MEC-100*, mentre ***Scheda*** rappresenta la singola unità. Questa distinzione offre espandibilità futura al database, senza doverne modificare la struttura: infatti se si dovesse introdurre un nuovo tipo di regolatore basterà aggiungerne i relativi dati nelle tabelle corrispondenti a ***Regolatore*** e ***Param Reg*** (che rappresenta i vari parametri di ciascun tipo di regolatore).

Generatore rappresenta lo specifico generatore elettrico a cui è collegato il regolatore e, come nel caso di questi ultimi, le entità per le singole unità e per le tipologie sono state separate.

L'entità ***Hachiko*** rappresenta una scheda *Hachiko*. Un codice seriale la identifica univocamente ed è usato come chiave primaria. In questa entità si possono inserire ulteriori attributi legati al software della scheda. La relazione tra ***Hachiko*** e ***Scheda*** rappresenta il collegamento effettivo tra le due schede e lo storico dei collegamenti passati. L'ordinamento tramite un attributo *timestamp* può fornire la cronologia.

L'entità ***Cliente*** contiene i dati relativi ai clienti dell'azienda. Questa entità ha attributi *username* e *password* che potranno essere utilizzati per accedere ai dati dei dispositivi associati attraverso l'applicazione *Android*. Qui andranno memorizzati anche gli eventuali dati anagrafici o di contatto. aziendali.

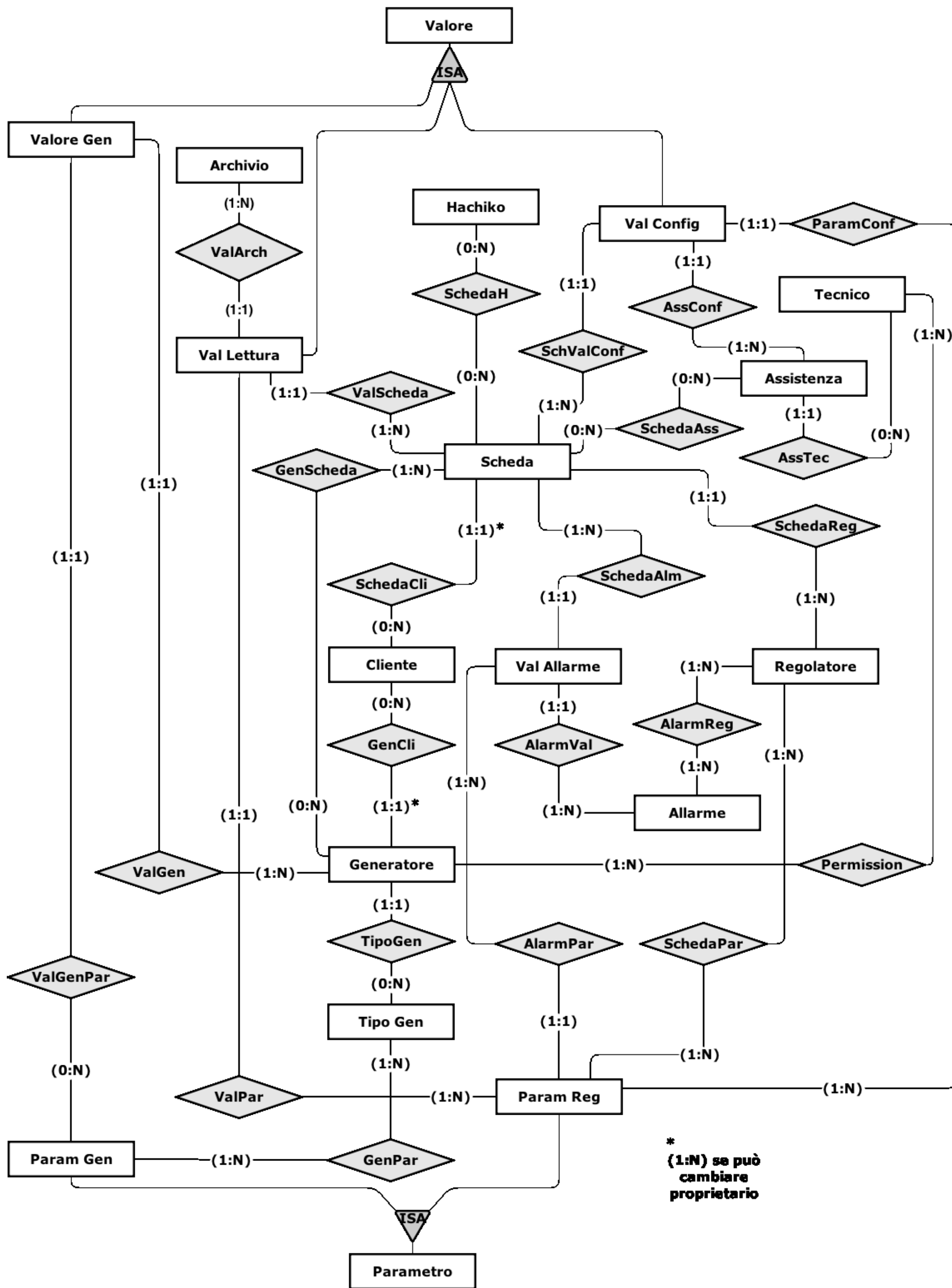


Figura 4.1: Lo schema *ER* completo del database.

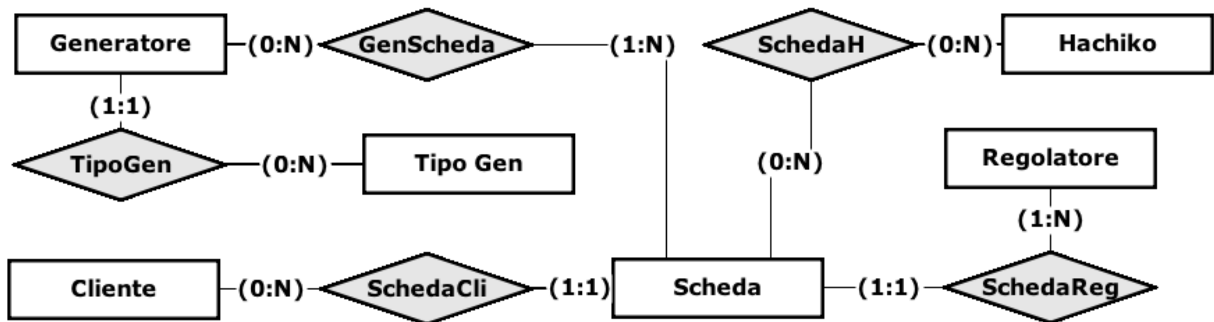


Figura 4.2: Sezione dello schema *ER* per la rappresentazione delle schede *MEC-100* e *Hachiko* nel database. Le singole istanze dei regolatori (e dei generatori) sono disaccoppiate dalla rispettiva tipologia, per permettere l'inserimento di nuovi regolatori in futuro.

4.1.2 Modellizzazione dei dati rilevati e delle assistenze

Nel "sotto-schema" di Figura 4.3 sono raffigurate le entità utili a modellare i dati di esercizio e i parametri di configurazione rilevati dai regolatori *MEC-100*. poiché il database deve poter supportare nuovi regolatori non ancora disponibili, esso utilizza l'entità ***Param Reg*** per rappresentare i vari parametri e i dati di esercizio che i regolatori possono gestire. In questo modo ogni regolatore viene associato ad un (sotto)insieme di parametri e lo schema risulta estremamente flessibile.

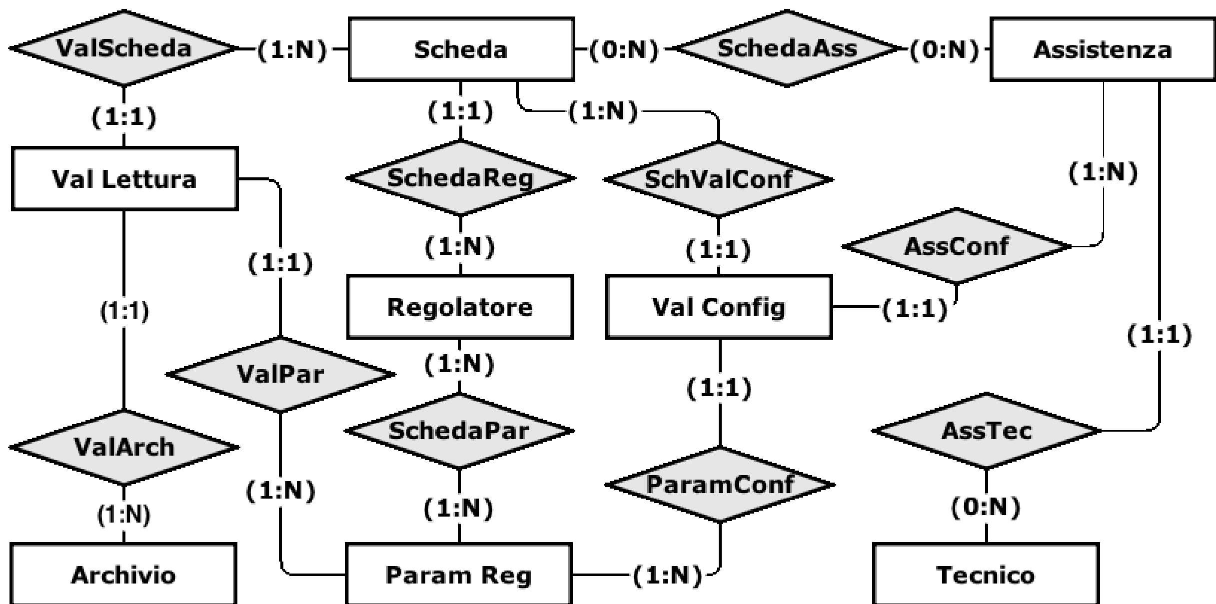


Figura 4.3: Sezione dello schema *ER* per la rappresentazione dei valori di esercizio, dei parametri di configurazione e delle informazioni di manutenzione.

I singoli valori numerici vengono memorizzati nelle due entità ***Val Lettura*** e ***Valore Manut*** a seconda che si tratti, rispettivamente, di valori di esercizio o di parametri di configurazione. Questi valori sono associati ai rispettivi regolatori di appartenenza grazie all'associazione con l'entità ***Scheda***. Le due entità sono separate perché rappresentano dati concettualmente diversi, che vengono letti

e/o scritti secondo modalità differenti. I valori di esercizio sono associati ad un'entità ***Archivio*** dove verranno memorizzati tutti i file di *log* compressi contenenti le letture puntuali. Infatti, come precedentemente esposto, i dati memorizzati nel database sono di tipo aggregato, nel caso di funzionamento normale, in modo da ridurre il carico di lavoro sia del client che del server. Per i dettagli relativi al carico di lavoro gestito dal sistema si veda la Sezione 2.5.

Nell'entità ***Valore Manut*** viene utilizzato un campo *timestamp* per ordinarne i valori, in modo da poter ottenere uno storico di tutte le configurazioni associate a ciascun *MEC-100*. Tale entità è associata anche alle assistenze per la memorizzazione dei parametri modificati durante le manutenzioni.

L'entità ***Assistenza*** contiene attributi binari (*BLOB - Binary Large Object*) per la memorizzazione di eventuali foto, file di *log* e documenti a corredo delle altre informazioni. Inoltre essa è associata, con partecipazione obbligatoria, all'entità ***Tecnico***, perché deve sempre essere possibile risalire a quale tecnico ha effettuato la sessione di manutenzione. Anche la configurazione iniziale di un regolatore verrà memorizzata nel database come una normale assistenza.

4.1.3 Modellizzazione degli allarmi

Gli allarmi sono stati trattati in modo simile a quanto fatto per i dati di esercizio. Essi sono identificati da codici memorizzati nell'entità ***Allarme***. Come per i normali parametri, anche il numero e il tipo di allarmi può variare da un regolatore all'altro. Quando la scheda *Hachiko* esegue una lettura dei dati d'esercizio e rileva dei *flag* di allarme attivati, i valori letti vengono inseriti in ***Val Allarme***: l'entità rappresentante i dati d'esercizio affetti da anomalie.

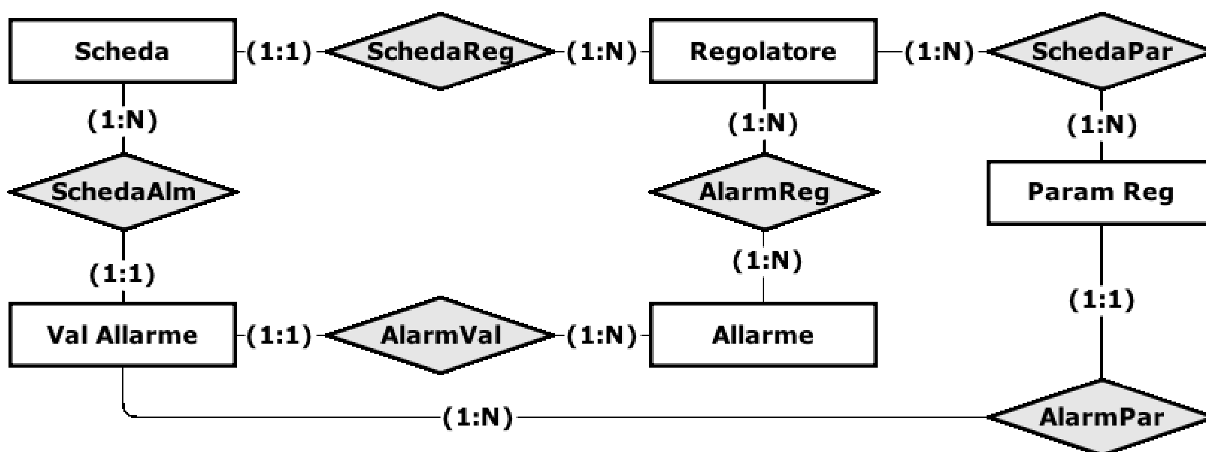


Figura 4.4: Sezione dello schema *ER* per la rappresentazione delle situazioni di funzionamento anomale, quando cioè vengono rilevati dei flag di allarme impostati a `ON`.

4.2 Analisi di normalizzazione del database

La normalizzazione di uno schema *ER* è il processo tramite il quale si eliminano eventuali ridondanze informative al fine di ridurre il più possibile il rischio di incorenze nei dati del database. Esistono vari livelli di normalizzazione, corrispondenti a varie *forme normali*. Nel seguito si vedrà come il database progettato rispetti le prime tre forme normali e la forma normale di *Boyce-Codd*.

poiché dallo schema di Figura 4.1 sono stati omessi tutti gli attributi, e in particolare le chiavi primarie e candidate, l'analisi seguente non potrà scendere troppo nei dettagli ed è riportata solo per completezza.

4.2.1 Alcune definizioni utili nel seguito

Le definizioni seguenti sono state prese dal libro "Sistemi di basi di dati - Fondamenti, 5° edizione" [5].

- ***Superchiave***: è un insieme di attributi che possono identificare univocamente le tuple di una relazione.
- ***Chiave candidata***: è un insieme di attributi *minimale* di una relazione che consente di identificarne univocamente le tuple. In altre parole si tratta di una *superchiave minimale*. In una relazione possono esserci più chiavi candidate, una di esse viene scelta come *chiave primaria*.
- ***Attributo primo***: in uno schema di relazione, si dice *primo* un attributo che è parte di una qualche chiave candidata dello schema, mentre si dice non primo un attributo che non fa parte di alcuna chiave candidata dello schema.
- ***1NF - Prima forma normale***: uno schema di relazione **R** è in prima forma normale (*1NF*) se il dominio di ciascun attributo comprende solo valori atomici ed è un valore singolo del dominio dell'attributo stesso. Inoltre deve sempre esistere una chiave primaria per identificare univocamente le tuple di ogni relazione.
- ***2NF - Seconda forma normale***: uno schema di relazione **R** è in seconda forma normale (*2NF*) se rispetta la prima forma normale e se ogni attributo non-primo **A** di **R** *non* è parzialmente dipendente da alcuna chiave di **R**. Ovvero dipende solo dalla chiave primaria.
- ***3NF - Terza forma normale***: uno schema di relazione **R** è in terza forma normale (*3NF*) se rispetta la seconda forma normale e se per ogni dipendenza funzionale (non banale) **X** → **Y** definita su di esso, o X è una superchiave di **R** o **Y** è un attributo primo di **R**.
- ***BCNF - Forma normale di Boyce e Codd***: uno schema di relazione **R** è in forma normale di Boyce e Codd (*BCNF*) se rispetta la terza forma normale e se per ogni dipendenza funzionale (non banale) **X** → **Y** definita su di essa, **X** è una superchiave di **R**.

Un buono schema di relazione dovrebbe rispettare almeno le prime tre forme normali (la forma normale di Boyce e Codd impone un vincolo un po' più stringente rispetto alla 3NF).

4.2.2 1*NF* - Prima forma normale

Tutte le relazioni dello schema vengono identificate da attributi che, nella maggior parte dei casi, sono dei semplici ID che assumono valori atomici univoci. In alcuni casi la chiave primaria è composta da due o tre attributi, ma sempre con valori atomici. In tutti i casi la chiave contiene un singolo valore del proprio dominio e quindi lo schema rispetta la *1NF*.

Alcuni esempi sono:

- la chiave primaria della relazione ***Tecnico*** è costituita dall'attributo *username*;
- la chiave primaria della relazione ***SchedaPar*** è formata dai riferimenti alle chiavi primarie di ***Regolatore*** e ***Param Reg***, che sono entrambe degli ID a valore atomico.

4.2.3 2*NF* - Seconda forma normale

Come precedentemente esposto, la maggior parte delle relazioni usa un semplice ID come chiave primaria. In tutti questi casi è possibile notare che la seconda forma normale è rispettata. Infatti gli attributi *non-primi* non possono dipendere parzialmente dalla chiave, essendo questa costituita di un solo attributo. La verifica è immediata anche per le relazioni in cui tutti gli attributi presenti costituiscono la chiave primaria: è il caso, ad esempio, delle relazioni "di collegamento" come ***AlarmVal***.

Tutte le relazioni presenti nel database progettato rientrano in una di queste due categorie. Di conseguenza si può affermare che lo schema rispetta la *2NF*.

4.2.4 3*NF* - Terza forma normale

Il database è stato progettato in modo che tutti gli attributi (non-primi) delle varie relazioni dipendano *solo ed esclusivamente* dagli attributi della chiave primaria, escludendo quindi ogni possibile dipendenza transitiva. In altre parole, nessun attributo dipende da altri attributi non-chiave, perciò lo schema rispetta anche la *3NF*.

4.2.5 *BCNF* - Forma normale di Boyce e Codd

Nessuna delle relazioni presenti nel database possiede più di una chiave candidata, quindi non possono esserci due o più chiavi sovrapposte. Ciò è sufficiente a garantire che, se le relazioni sono in *3NF*, allora rispettano anche la *BCNF*.

poiché tutte le relazioni rispettano queste condizioni si può affermare che il database è in forma normale di Boyce e Codd (*BCNF*).

4.3 Costruzione del database

Dallo schema *ER* illustrato in questo capitolo si è poi ricavato uno schema logico relazionale per la successiva implementazione utilizzando il *Data Base Managment System* ***PostgreSQL*** [6]. É stato creato un file *.sql* con tutte le tabelle e le definizioni di dominio necessarie, in modo da poter ricostruire l'intero database con un singolo comando. Il file in questione è riportato in appendice.

5. Architettura client

In questo capitolo verrà analizzato in dettaglio il funzionamento del programma client in esecuzione sulla scheda *Hachiko*. Si tratta di un software *multi-thread*, in cui ogni *thread* svolge una funzione logica distinta. I *thread* comunicano tra loro mediante un meccanismo di memoria condivisa basato su meccaniche di mutua esclusione.

5.1 Struttura software

Il software è stato scritto in `C++` e fa uso delle librerie `boost`, in particolare sono utilizzate le librerie: `foreach`, `thread`, `tokenizer` e `iostreams` [7]. All'avvio del programma principale vengono letti alcuni parametri di configurazione e viene inizializzata la memoria condivisa, dopodiché vengono avviati i principali *thread* indipendenti. Si tratta di tre componenti, di cui due sono attualmente operativi, mentre il terzo verrà implementato in futuro (si veda la Sezione 6.2):

- modulo di gestione dei dati - *DataProcessor*;
- modulo di comunicazione - *CommModule*;
- modulo di interfaccia grafica - *GUIModule* (attualmente non implementato).

```
57 // Create separate threads for DataProcessor, CommModule and GUIModule.
58 // Threads will communicate using shared memory.
59 mutex mtx;
60 boost::thread dataProcTh, commModTh;//, guiTh;
61
62 // Create and start DataProcessor thread.
63 DataProcessor dataProc(mtx, sMem);
64 dataProcTh = boost::thread(bind(&DataProcessor::activate, &dataProc, cfg.serialDev));
65
66 // Create and start CommModule thread.
67 CommModule commMod(mtx, sMem, cfg);
68 commModTh = boost::thread(bind(&CommModule::activate, &commMod));
69
70 // TODO Create and start G.U.I. thread.
71 //GUIModule gui(mtx, sMem, cfg);
72 //guiTh = boost::thread(bind(&GUIModule::activate, &gui));
73
74 // Wait for threads to end (threads should run "forever").
75 dataProcTh.join();
76 commModTh.join();
77 //guiTh.join();
```

Codice 5.1: Avvio dei *thread* indipendenti.

Il software client è stato suddiviso in varie classi in base alla logica di funzionamento dei vari componenti. In particolare alcuni di essi formano l'ossatura base del programma e possono essere

utilizzati, con modifiche minime, anche per altri regolatori futuri, mentre altre classi sono specifiche per il regolatore *MEC-100*. L'elenco seguente illustra brevemente le varie classi implementate.

- `SerialDriver` fornisce funzioni di basso livello per la comunciazione seriale, come ad esempio `int config(chCfg &cfg)` per la configurazione del canale o `int sendByteArray(vByte bArr)` e `int receiveAll(vByte &b)` per l'invio e la ricezione di dati.
- `MEC100Comm` è una classe che usa le funzionalità del driver seriale per comunicare con il regolatore *MEC-100* in modo "intelligente". Infatti il driver seriale "vede" solo flussi di bytes, mentre il `MEC100Comm` è in grado di interpretarli tramite funzioni come `MEC100Params processParams()` o `MEC100Values processValues()`.
- `MEC100Values`: si tratta di una classe utilizzata per rappresentare i valori di esercizio rilevati dal regolatore *MEC-100*. Se in futuro si introdurrà un nuovo regolatore questa classe dovrà essere (parzialmente) riscritta in base al numero e al tipo di valori gestiti dal nuovo regolatore.
- `MEC100Params` ha una funzione molto simile alla precedente, ma serve a modellare i parametri di configurazione anziché i dati di esercizio. Valgono le stesse considerazioni fatte per `MEC100Values` per quanto riguarda eventuali regolatori futuri.
- `DataProcessor` è la classe che gestisce i dati, per i dettagli si veda la Sezione 5.1.2.
- `CommModule` è la classe che si occupa della comunicazione remota, per i dettagli si veda la Sezione 5.1.3.
- `GUIModule` è la classe per la gestione dell'interfaccia grafica. Attualmente non è stata implementata in quanto l'azienda non ne ha ancora definito le specifiche di progetto (si veda la Sezione 6.2).

Oltre alle classi principali sopra elencate, sono stati definiti vari altri componenti: tipi di dato personalizzati, macro (indispensabili per una gestione ordinata delle decine di parametri e dati del regolatore *MEC-100*) e alcune librerie di funzioni e classi di supporto come la classe `Zipper`, utilizzata per la compressione dei file di *log* testuali.

5.1.1 La memoria condivisa per la comunicazione *inter-thread*

Tutti i *thread* in esecuzione nel client *Hachiko* hanno accesso ad una **struct** di memoria condivisa tramite la quale vengono scambiati dati e messaggi. Tale struttura è rappresentata dalla variabile `sMem` in tutti i listati di codice di questo elaborato.

Il seguente blocco di codice riporta la struttura completa della memoria condivisa utilizzata:

```
25  // Thread "messages" are stored in FIFO queues.
26  typedef deque<Str> FIFOq;
27  typedef deque<pair<Str, vStr> > MAINTq;
28  typedef deque<pair<Str, MEC100Params> > PARAMq;
29
30  // Define a struct for sharing messages between the Hachiko client's local threads.
31  // Data are sent to the server according to a FIFO queue, with the alarm data
32  // taking priority over aggregated ones. Therefore data must be inserted in the back
33  // of the queue and extracted from the front (or vice-versa).
34  typedef struct sharedData {
35
36      // Log files path.
37      Str logPath;
38
39      // MEC100 initial configuration data.
40      Str startCfg;
41      Str hachikoSN;
```

```
42     Str MEC100SN;
43
44     // Heart-beat data.
45     uint heartBeat;
46
47     // Queue for (daily) log files.
48     FIFOq logFiles;
49
50     // Queue for aggregated data log files.
51     FIFOq aggrFile;
52
53     // Queue for alarm data.
54     FIFOq almData;
55
56     // Maintenance shared data.
57     bool maintMode;
58     MAINTq maintenance;
59     PARAMq params;
60
61     // Pointer to MEC100 values data for the G.U.I. thread.
62     MEC100Values *vals;
63 } shared;
```

Codice 5.2: Struttura per la memoria condivisa per la comunicazione inter-thread.

Come si può vedere, i dati da inviare al server vengono memorizzati in code di tipo *FIFO - First In First Out*. Questo avviene perché non c'è garanzia che la scheda abbia accesso alla rete nel momento in cui i dati sono pronti per l'invio. Utilizzando code di tipo *FIFO* è possibile spedire i dati nell'ordine in cui sono stati generati, indipendentemente da *quando* sono stati disponibili. Inoltre utilizzando code distinte è possibile dare priorità ai dati di allarme rispetto agli altri. É anche previsto un puntatore agli ultimi valori di esercizio rilevati che verrà utilizzato dal modulo di interfaccia grafica per visualizzare in tempo reale i dati letti.

5.1.2 Il modulo di gestione dati *DataProcessor*

Nella Figura 5.1 è riportato lo schema di funzionamento *flow-chart* dettagliato del modulo di gestione dei dati.

Come si vede dalla figura il *DataProcessor* esegue alcune operazioni preliminari per permettere l'invio dei pacchetti *AUTHENTICATION* e *BOOTSTRAP*, dopodiché avvia un ciclo infinito nel quale va a leggere periodicamente i dati di esercizio dal regolatore *MEC-100*.

Dal grafico è chiaramente visibile che questo modulo si occupa anche di generare gli *heart-beat*, i quali non sempre vengono generati a intervalli precisi di un minuto. Tuttavia eventuali piccoli ritardi non costituiscono un problema. Infatti il ritardo può essere al massimo di pochi secondi, a meno che non sia attiva una sessione di manutenzione. In tal caso possono verificarsi due differenti scenari: manutenzione effettuata sul luogo da un tecnico presente fisicamente, oppure assistenza da remoto. In entrambi i casi la scheda sospende temporaneamente la lettura in *polling* dei dati ordinari, come anche l'invio di eventuali *heart-beat*. Nel caso della manutenzione sul luogo si suppone che, in quel momento, nessuno vorrà effettuare un'assistenza da remoto quindi se anche il server perde, temporaneamente, la capacità di contattare il client questo non rappresenta un problema. Nel caso, invece, di una manutenzione remota, client e server dovranno scambiarsi vari pacchetti dati (che allo stato attuale non sono ancora stati definiti), quindi si suppone che la connessione sia stabile.

Durante il funzionamento normale del *DataProcessor* i dati di esercizio rilevati dal regolatore vengono letti a intervalli di un secondo e salvati in un file di *log*, oltre che aggregati in alcune variabili temporanee. Alla mezzanotte di ogni giorno, o quando viene rilevato un allarme, il file di *log* viene chiuso e spedito al server (segnalando al modulo di comuncazione che i dati sono pronti), insieme con i valori aggregati raccolti fino a quel momento. Si noti che l'indicazione temporale utilizzata

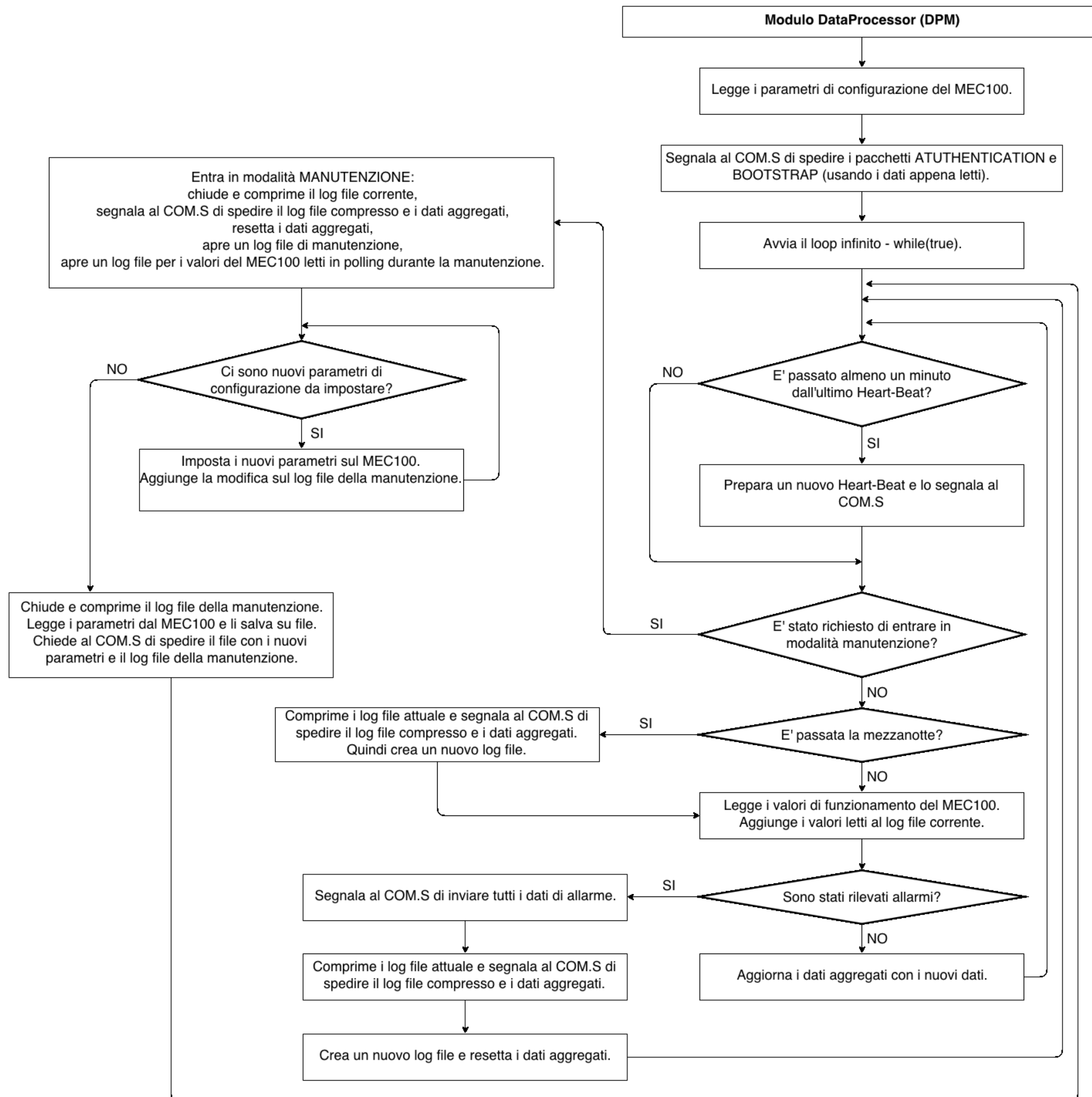


Figura 5.1: Diagramma di flusso (*flow-chart*) per il modulo di gestione dei dati *DataProcessor*.

è quella dell'orologio locale presente nella scheda *Hachiko* la quale, per ora, non effettua nessuna sincronizzazione con orologi presenti in rete. Anche questo aspetto sarà implementato in futuro (Sezione 6.5).

Nel caso in cui venga rilevato almeno un allarme in una delle letture viene generato e spedito anche un pacchetto di tipo *ALARM*, il quale ha priorità assoluta rispetto a tutti gli altri dati da spedire (si veda il *CommModule* 5.1.3 per i dettagli). La presenza di un allarme impone la chiusura del file di *log* attuale e la generazione di un nuovo file.

Infine, quando arriva una richiesta per l'ingresso in ''modalità manutenzione'' quello che succede

è analogo a quando viene rilevato un allarme oppure è passata la mezzanotte: si chiude il file di *log* attuale e si segnala al modulo di comunicazione che i dati sono pronti per l'invio. Contestualmente vengono generati tre nuovi file che andranno a memorizzare:

- i valori di esercizio letti durante la manutenzione;
- tutte le modifiche ai parametri di configurazione effettuate (infatti molto spesso i tecnici devono procedere per tentativi);
- la configurazione finale del regolatore, rilevata al termine della manutenzione.

Al termine dell'assistenza il modulo dati riprende il suo normale funzionamento come descritto sopra.

Alcuni listati di codice per il modulo *DataProcessor*

Nei blocchi di codice seguenti sono riportati alcune parti significative del modulo *DataProcessor*.

```
101 // Check if other threads requested to enter maintenance mode (i.e. server maintenance request).
102 {
103     mutex::scoped_lock mtxLock(*mtx);
104     if(sMem->maintMode) {
105 
106         // Send current archive (if any) to the communication module.
107         if(currArch.compare("") != 0)
108             sendValues(currArch, aggr, count);
109         currArch = "";
110 
111         // Do maintenance operations.
112         maintenanceMode(com, maintCnt++);
113 
114         // Create new log file.
115         if(getDay() == day) {
116             currArch = newArchName(++arcCnt);
117             title = newArchTitle(currArch, sn);
118             appendToFile(sMem->logPath + currArch, title);
119         }
120     }
121 }
```

Codice 5.3: Controllo delle condizioni per l'ingresso in modalità manutenzione.

```
143 // Read MEC100 data.
144 MEC100Values vals;
145 while(vals.serialN.compare("") == 0)
146     vals = com.retrieveValues();
147 
148 // Update shared memory data pointer (for G.U.I. thread).
149 {
150     mutex::scoped_lock mtxLock(*mtx);
151     sMem->vals = &vals;
152 }
153 
154 // Print time information on log file (current reading time).
155 dataHeader = newDataHeader();
156 appendToFile(sMem->logPath + currArch, dataHeader);
157 
158 // Write current data to file.
159 vals.appendToFile(sMem->logPath + currArch);
160 
161 // No alarms: store the average (aggregation).
162 if(!vals.isAlarmsSet()) {
163     aggr += vals;
```

```
164     count++;
165 }
```

Codice 5.4: Lettura e memorizzazione dei valori di esercizio.

```
193 // Alarm data has top priority.
194 if(alm) {
195     mutex::scoped_lock mtxLock(*mtx);
196     Str almFile("ALARM_" + file);
197     almData.toFile(sMem->logPath + almFile);
198     sMem->almData.push_back(sMem->logPath + almFile);
199 }
200 
201 // Create aggregated data log file.
202 aggr /= n;
203 Str aggrFile("aggr_" + file);
204 aggr.toFile(sMem->logPath + aggrFile);
205 
206 // Compress log file and remove original.
207 Zipper z;
208 Str zipped = z.zipFile(sMem->logPath + file, true);
209 
210 // Store data in shared memory (to be used by the communication module thread).
211 {
212     mutex::scoped_lock mtxLock(*mtx);
213     sMem->logFiles.push_back(zipped);
214     sMem->aggrFile.push_back(sMem->logPath + aggrFile);
215 }
```

Codice 5.5: Invio dei dati letti (anche in presenza di allarmi) al modulo di comunicazione.

5.1.3 Il modulo di comunicazione *CommModule*

Nella Figura 5.2 è riportato lo schema di funzionamento dettagliato del modulo di comunicazione.

Il *CommModule* genera due *thread* separati: uno per l'invio e uno per la ricezione dei dati. Il *thread* di ricezione (*COM.R*) rimane in ascolto di eventuali richieste da parte del server. Al momento è prevista solo una richiesta per manutenzioni remote, ma potrebbero esserne aggiunte altre in futuro (come la possibiltà per i client *Android* di richiedere l'aggiornamento immediato dei valori di esercizio sul server, si veda la Sezione 6.5). La richiesta di assistenza, attualmente, non viene comunque gestita in quanto non sono ancora state definite le specifiche di funzionamento. Il modulo *DataProcessor* è in grado di modificare i parametri di configurazione del *MEC-100* e gestire i file di *log*, tuttavia la modifica di alcuni valori potrebbe causare danni al regolatore o anche al generatore elettrico, arrivando anche a causare rischi di sicurezza. Pertanto finchè non saranno state definite dettagliatamente le specifiche relative alla manutenzione il modulo *COM.R* non potrà essere completato.

Per quanto riguarda il secondo componente (*COM.S*), esso si occupa di inviare i dati al server, siano essi di manutenzione, di allarme o normali letture di esercizio. I primi dati che vengono inviati sono i pacchetti *AUTHENTICATION* e *BOOTSTRAP*, spediti appena la scheda *Hachiko* viene avviata e stabilisce una connessione con il regolatore *MEC-100*. Quando verranno implementate funzionalità di cifratura dei dati, in questa prima fase verranno anche negoziate le chiavi di cifratura fra client e server (tra l'invio dei due pacchetti).

Dopo la fase iniziale anche questo modulo, come il *DataProcessor*, avvia un ciclo infinito di controllo periodico della memoria condivisa alla ricerca di eventuali dati da spedire. Difatti il modulo di comunicazione assembla i pacchetti, ma non produce i dati da inviare, i quali vengono controllati secondo un preciso ordine di priorità:

- eventuali *log* di manutenzione (in modo da avere sempre la configurazione più aggiornata disponibile sul server);

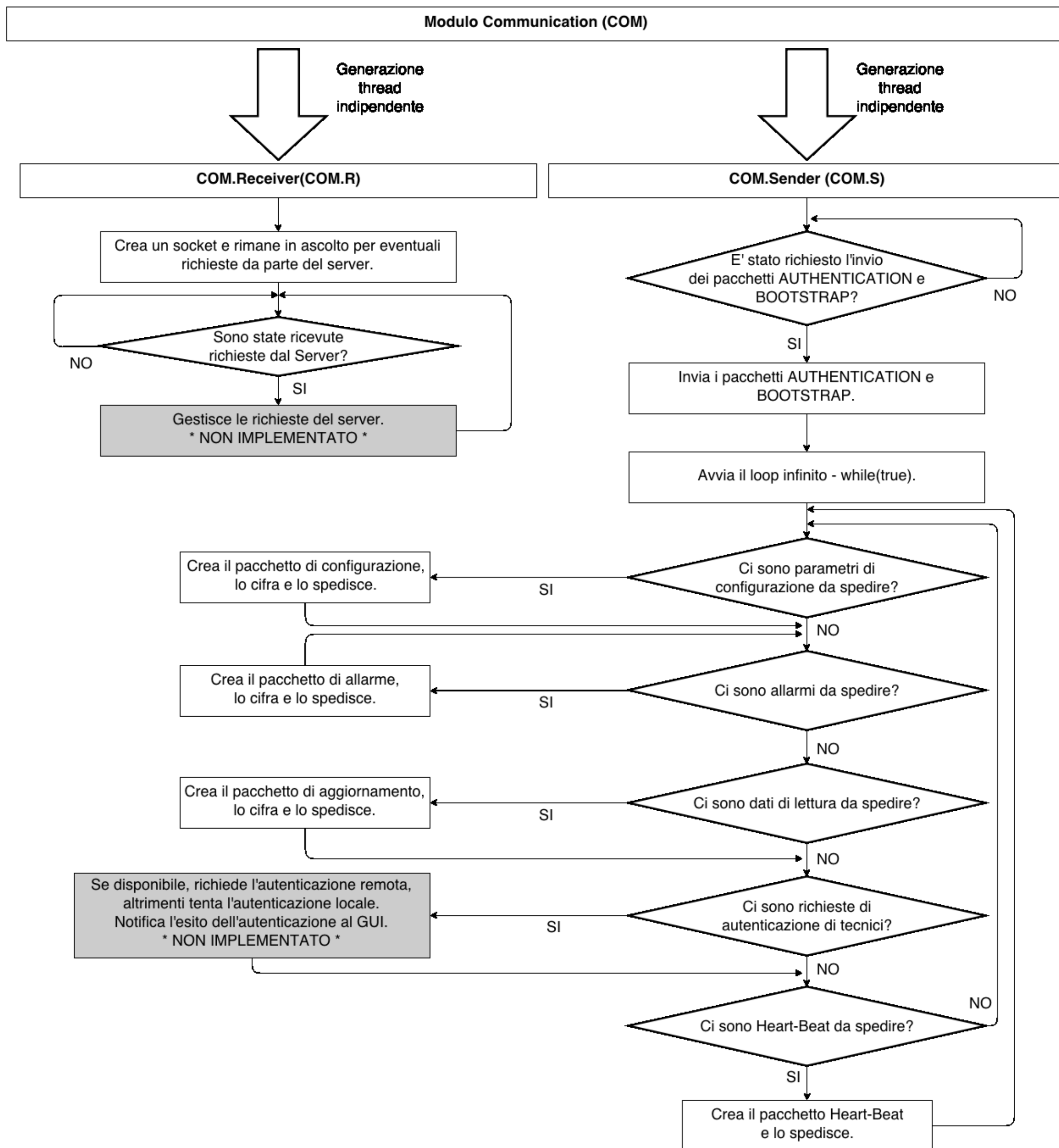


Figura 5.2: Diagramma di flusso (*flow-chart*) per il modulo di comuniczioane *CommModule*.

- dati di esercizio contenenti allarmi per poter agire tempestivamente sulle situazioni potenzialmente problematiche;
- dati di esercizio ordinari;
- dati relativi a richieste di autenticazione remota di tecnici (generate dal modulo *GUIModule*);
- *heart-beat*.

In tutti i casi, nel momento in cui vengono rilevati dati da spedire, viene generato il corrispondente pacchetto e viene inviato al server. Nei blocchi di codice seguenti sono riportati alcuni frammenti significativi del modulo di comunicazione.

Alcuni listati di codice per il modulo *CommModule*

```
230 // Sender thread should run forever.
231 while(true) {
232     bool cfgToSend  = false;
233     bool almDataToSend = false;
234     bool dataToSend  = false;
235     bool hbToSend  = false;
236     {
237         mutex::scoped_lock mtxLock(*mtx);
238         if(sMem->maintenance.size() > 0)
239             cfgToSend = true;
240 
241         if(sMem->almData.size() > 0)
242             almDataToSend = true;
243 
244         if(sMem->logFiles.size() > 0)
245             dataToSend = true;
246 
247         if(sMem->heartBeat > 0)
248             hbToSend = true;
249     }
```

Codice 5.6: Controllo per la presenza di dati da inviare.

```
270     // Alarm data has priority over aggregated data: send ALL alarm data before sending any aggregated data.
271     while(almDataToSend) {
272         Str alm(sMem->almData.front());
273 
274         // Acquire lock to delete data that has been sent.
275         if(sendAlarmData(alm)) {
276             mutex::scoped_lock mtxLock(*mtx);
277             sMem->almData.pop_front();
278             remove(alm.c_str());
279         }
280 
281         // Check if there are other alarm data to send.
282         {
283             mutex::scoped_lock mtxLock(*mtx);
284             if(sMem->almData.size() > 0)
285                 almDataToSend = true;
286             else
287                 almDataToSend = false;
288         }
289     }
```

Codice 5.7: Invio di dati di esercizio contenenti allarmi.

```
374 // Function used to build any type of packet.
375 vByte CommModule::buildPacket(Byte reqType, Str dataFile, Str logFile, Str valsLog, Str username) {
376     vByte serialHachiko = strToBytes(hachikoSN);
377     vByte serialMEC100 = strToBytes(MEC100SN);
378 
379     // Build packet header.
380     vByte header = buildHeader(reqType, serialHachiko, serialMEC100);
381 
```

```
382     // Build packet payload.
383     vByte payload;
384     payload.clear();
385     switch(reqType){
386         case REQ_TYPE_BOARD_AUTH:
387             break;
388         case REQ_TYPE_BOOTSTRAP:
389             payload = buildBootstrapPayload(dataFile);
390             break;
391         case REQ_TYPE_VALUES:
392             payload = buildValuesPayload(dataFile, logFile);
393             break;
394         case REQ_TYPE_ALARM:
395             payload = buildAlarmPayload(dataFile);
396             break;
397         case REQ_TYPE_HEART_BEAT:
398             break;
399         case REQ_TYPE_TECH_LOGIN:
400             //TODO build tech login payload
401             break;
402         case REQ_TYPE_MAINTENANCE:
403             payload = buildMaintenancePayload(username, dataFile, logFile, valsLog);
404             break;
405     }
406
407     // Build packet: packet length, header, payload.
408     int len = 4 + header.size() + payload.size();
409     vByte pLen = intToBytes(len);
410     vByte packet;
411     packet.clear();
412     packet.insert(packet.end(), pLen.begin(), pLen.end());
413     packet.insert(packet.end(), header.begin(), header.end());
414     packet.insert(packet.end(), payload.begin(), payload.end());
415     return packet;
416 }
```

Codice 5.8: Funzione di costruzione dei pacchetti dati.

```
431 vByte CommModule::buildBootstrapPayload(Str cfg) {
432     vByte payload;
433
434     // Build payload - Log file.
435     // Read file length (position to the last char, read position (int), reset position to first char).
436     ifstream in(cfg);
437     in.seekg(0, in.end);
438     uint len = in.tellg();
439     in.seekg(0, in.beg);
440
441     // Read file into a byte array.
442     vByte bytes;
443     char *buffer = new char[len];
444     in.read(buffer, len);
445     for(uint i = 0; i < len; i++)
446         bytes.push_back((Byte)buffer[i]);
447
448     // Add configuration file text to the payload.
449     payload.insert(payload.end(), bytes.begin(), bytes.end());
450
451     // Close file and clear memory.
452     in.close();
453     delete[] buffer;
454     return cipher(payload);
455 }
```

Codice 5.9: Costruzione del *payload* per il pacchetto *BOOTSTRAP*.

5.1.4 Il modulo di interfaccia grafica *GUIModule*

Allo stato attuale il modulo di interfaccia grafica non è stato implementato, né si è stabilito con esattezza quali saranno le sue funzioni o come dovrà apparire l'interfaccia. Come minimo dovrà permettere di consultare i dati di esercizio dalla scheda *Hachiko* in tempo reale e dovrà consentire di modificare i parametri di configurazione del regolatore. Ma eventuali altre funzionalità, come i grafici delle grandezze elettriche in tempo reale, non sono ancora state discusse in azienda. Per ulteriori dettagli si veda la Sezione 6.2 nel capitolo sugli sviluppi futuri.

5.1.5 Modalità di utilizzo del software client

Il software è stato progettato anche per l'utilizzo in condizioni di funzionamento "anomale". In particolare è possibile:

- avviare il client *Hachiko* indipendentemente dal collegamento con il regolatore *MEC-100*. Se i due dispositivi non dovessero essere connessi, il client tenterà di stabilire una connessione a intervalli regolari di pochi secondi. Appena il collegamento viene stabilito il software riprende il normale funzionamento;
- è possibile scollegare il *MEC-100* e ricollegarlo senza dover spegnere la scheda *Hachiko* (*hot-swap*, anche sostituendo lo specifico regolatore);
- nel caso non sia disponibile una connessione a internet, o questa venga persa, il client è in grado di riprendere l'invio dei dati da dove era stato interrotto;
- in futuro (si veda la Sezione 6.4) il client sarà robusto anche rispetto ad eventuali black-out. Infatti è prevista l'implementazione di un meccanismo per la ripresa dell'attività da dove è stata interrotta, senza perdita di dati.

5.1.6 Dettagli sulla comunicazione tra client e server

In questa sezione sarà illustrata nel dettaglio la comunicazione tra client e server, con particolare riferimento a quali dati vengono scambiati nelle varie fasi.

CODE 0 - AUTHENTICATION (autenticazione *Hachiko*/*MEC-100* nel sistema)

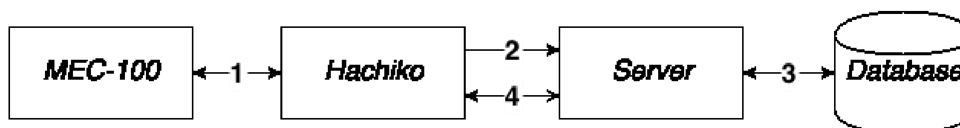


Figura 5.3: Comunicazione client-server relativa al pacchetto *0 - AUTHENTICATION*.

Il pacchetto *AUTHENTICATION* viene inviato una sola volta, prima del pacchetto *BOOTSTRAP*, quando la scheda *Hachiko* viene accesa e collegata al relativo *MEC-100*. Quindi è il primo pacchetto che la scheda manda al server dopo l'accensione.

1. La scheda *Hachiko* interroga il *MEC-100* cui è collegata per ottenere il suo ID.

2. La scheda *Hachiko* manda un pacchetto di autenticazione al server contenente solo l'*HEADER [CODE 0]*. L'header conterrà gli ID delle schede *Hachiko* e *MEC-100*.

3. Il server interroga il database e, se necessario, aggiorna l'associazione *Hachiko* $\leftrightarrow$ *MEC-100* presente, o ne crea una nuova (tabella ***HACHIKO_BOARD***, relazione ***SchedaH*** dell'*ER*).

4. Il server conferma l'avvenuta autenticazione / registrazione del client nel sistema. In futuro client e server si scambieranno i messaggi necessari a creare una connessione sicura per le successive trasmissioni: l'header del pacchetto viaggerà sempre in chiaro, altrimenti il server non potrebbe selezionare la chiave di decodifica corretta, ma il payload verrà cifrato.

CODE 1 - BOOTSTRAP (prima connessione *Hachiko* $\leftrightarrow$ *MEC-100* / avvio della scheda *Hachiko*)

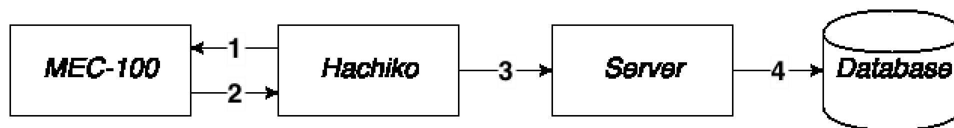


Figura 5.4: Comunicazione client-server relativa al pacchetto *1 - BOOTSTRAP*.

Il pacchetto *BOOTSTRAP* viene inviato solo dopo che è stato spedito il pacchetto *AUTHENTICATION*. Quando sarà implementato il meccanismo di cifratura il *payload* di questo pacchetto dovrà essere anche cifrato.

1. La scheda *Hachiko* tenta di comunicare con il *MEC-100* chiedendo i valori dei vari parametri di configurazione.

2. Il *MEC-100*, se collegato, risponderà con i valori dei parametri. A questo punto la scheda *Hachiko* può inviare un pacchetto *AUTHENTICATION* al server auntenticandosi quindi all'interno del sistema. La scheda *Hachiko* memorizza su file (compresso) tutti i parametri ricevuti e provvede ad inviarli al server (si veda punto 3). Contemporaneamente (*multi-threading*) la scheda inizia la lettura in polling dei dati di esercizio del *MEC-100* aggregandoli e memorizzandoli su disco. La lettura dei valori può essere interrotta (il file di *log* corrente viene chiuso, compresso e inviato al server insieme ai valori aggregati raccolti fino a quel momento) se la scheda deve entrare in modalità manutenzione (richiesta di modifica della configurazione, locale o remota) o se viene rilevato un allarme. Nel primo caso la scheda riprende la lettura in polling appena si è conclusa la sessione di manutenzione, mentre nel secondo caso la lettura riprende immediatamente dopo aver inviato i dati di allarme al server.

3. Una volta avvenuta l'autenticazione all'interno del sistema, la scheda invia il pacchetto con i dati di configurazione nel seguente formato:

   - HEADER [CODE 1] (contenente gli ID delle schede *Hachiko* e *MEC-100*);
   - PAYLOAD (un file di testo compresso contenente tutti i parametri di configurazione letti).

   Sarà poi compito del server elaborare il file per estrarne tutti i parametri e memorizzarli. Tutti i parametri sono codificati nel formato "XXXXXX=val" con il nome sempre composto da 6 caratteri e il valore che può essere interpretato come `INTEGER`, `FLOAT` o `BOOLEAN`.

4. Il server, una volta estratti i dati, interagisce con il DB:

- inserisce nella tabella ***HACHIKO_BOARD*** (relazione ***SchedaH*** dell'*ER*) la nuova associazione nel caso in cui la più recente associazione *Hachiko* ↔ *MEC-100* non corrisponda ai codici appena ricevuti. Questo può avvenire nel caso in cui il *MEC-100* sia stato sostituito senza spegnere la scheda *Hachiko* (se fosse stata spenta o sostituita la scheda *Hachiko* l'aggiornamento sarebbe stato fatto tramite il pacchetto *AUTHENTICATION)*;
- aggiorna l'IP presente in ***HACHIKO_BOARD*** e nella tabella che il software tiene in memoria (in tal modo il server può provare a raggiungere la scheda in caso di manutenzione usando l'ultimo IP noto). L'IP viene comunque aggiornato costantemente tramite i pacchetti *HEART-BEAT*;
- controlla l'ultima configurazione dei parametri di quel determinato *MEC-100*. Sono possibili i seguenti due casi.
  - Se i parametri corrispondono a quelli nel database (inseriti nell'ultima manutenzione, che può essere la configurazione di fabbrica) non vi è stata nessuna manutenzione e non sono necessarie ulteriori azioni.
  - Se, invece, i parametri non corrispondono significa che c'è stata una manutenzione non registrata, ovvero i parametri del *MEC-100* sono stati modificati senza utilizzare il software client o server. Si tratta di una manutenzione non autorizzata. In quest'ultimo caso i dati vanno comunque aggiornati, ma è necessario farlo tramite un'ulteriore assistenza di default che assumerà quindi il significato di ''assistenza non autorizzata''.

CODE 2 - VALUES (agiornamento dei valori letti in *polling*)

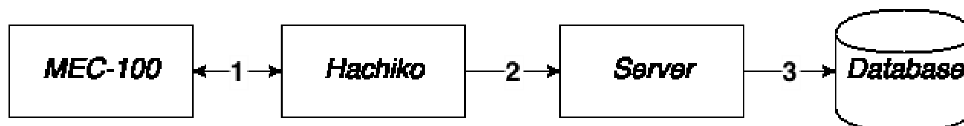


Figura 5.5: Comunicazione client-server relativa al pacchetto *2 - VALUES*.

Il pacchetto *VALUES* viene inviato una volta al giorno (durante il funzionamento ordinario) e contiene l'aggregazione di tutti i valori letti nella giornata, più un file di *log* compresso con tutti i valori puntuali rilevati. In caso di lettura con allarmi l'aggregazione viene interrotta e il pacchetto viene inviato con i dati rilevati fino al momento dell'allarme, dopodiché il software client provvede ad avviare una nuova aggregazione fino alla fine della giornata o fino al prossimo allarme.

1. La scheda *Hachiko* interroga il *MEC-100* in polling per ottenere i valori di esercizio a intervalli di un secondo. I dati vengono scritti su un file di *log* e vengono aggregati (media di tutti i valori letti).
2. La scheda *Hachiko* manda un pacchetto al server contenente l' *HEADER [CODE 2]* e un *PAYLOAD* nel formato descritto nella Sezione 3.2. In futuro, il payload verrà cifrato prima dell'invio.
3. Il server legge i parametri aggregati e li inserisce nel database, inserisce il file binario nella tabella ***LOG*** (entità ***Archivio*** dell'*ER*).

CODE 3 - ALARM (inserimento dati di allarme)

Il pacchetto *ALARM* viene inviato ogni volta che si presenta almeno un allarme nei valori che vengono letti in *polling*. In questo caso la lettura viene interrotta (il corrispondente file di *log* viene chiuso, gli ultimi valori inseriti sono quelli che contengono l'allarme) e viene generato un file di *log* per la sola lettura di allarme. Tale file viene inviato separatamente rispetto ai normali *log* e in modo prioritario dall'applicazione client.

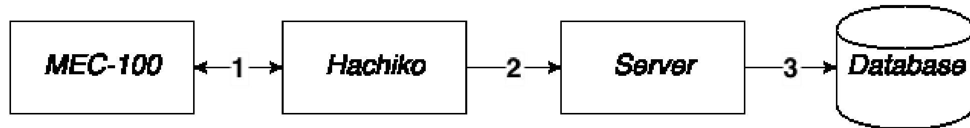


Figura 5.6: Comunicazione client-server relativa al pacchetto *3 - ALARM*.

1. La scheda *Hachiko* riceve i dati di allarme dal *MEC-100* durante la normale lettura in polling. I dati vengono scritti su un file di *log* separato e inviati "immediatamente"[1] in modo prioritario rispetto ai dati normali.
2. La scheda *Hachiko* manda un pacchetto al server contenente l'*HEADER [CODE 3]* e con un *PAYLOAD* contenente solo il file (compresso) con la lettura che ha generato l'allarme.
3. Il server estrae dal file tutti i valori e gli allarmi attivi e li inserisce nell'apposita tabella del database. Memorizzando tutti i dati esplicitamente in una tabella non è necessario mantenere anche il file, inoltre i valori puntuali della lettura che ha generato l'allarme sono comunque memorizzati nel file di *log* che è stato chiuso alla rilevazione dell'allarme (ma che sono stati esclusi dall'aggregazione).

CODE 4 - HEART-BEAT (aggiornamento *heart-beat*)

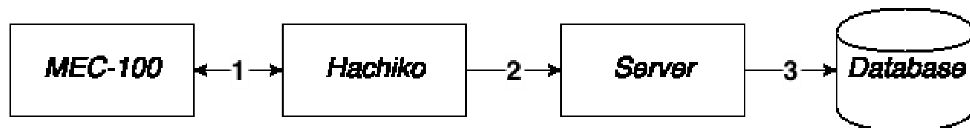


Figura 5.7: Comunicazione client-server relativa al pacchetto *4 - HEART-BEAT*.

Il pacchetto *HEART-BEAT* viene inviato ogni minuto dal client verso il server, il pacchetto non viene cifrato (e non sarà cifrato neanche in futuro), in quanto non contiene informazioni sensibili (il *PAYLOAD* è vuoto).

1. La scheda *Hachiko* riceve periodicamente (ogni secondo) i valori di funzionamento dal *MEC-100* e può quindi controllare che la connessione al regolatore non venga interrotta.
2. Periodicamente (ogni minuto) la scheda *Hachiko* genera un pacchetto *HEART-BEAT* che ha lo scopo di tenere aggiornato il server sullo stato del client, in particolare tenere aggiornato l'IP del client e l'associazione *Hachiko* $\leftrightarrow$ *MEC-100*.
3. Il server aggiorna quindi l'IP del client.

CODE 5 - LOGIN (autenticazione tecnico)

Il pacchetto viene inviato quando un tecnico tenta di effettuare il login direttamente dalla scheda *Hachiko*. Se la scheda non è collegata alla rete, il login può essere effettuato localmente se le credenziali inserite corrispondono a quelle memorizzate nella scheda.

1. La scheda *Hachiko* invia una richiesta di autenticazione contenente i dati del tecnico. Il pacchetto sarà formato dall'*HEADER [CODE 5]* e da un *PAYLOAD* contenente username e password.

[1] Non è possibile fornire nessuna garanzia sull'effettiva tempestività della comunicazione a causa della natura mobile dei client *Hachiko* e della latenza nella comunicazione.

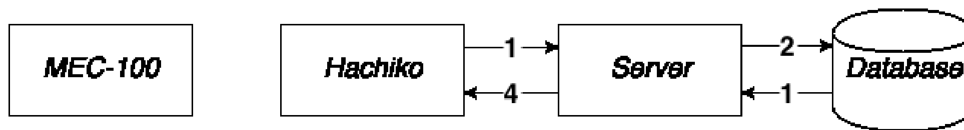


Figura 5.8: Comunicazione client-server relativa al pacchetto *5 - LOGIN*.

2. Il server interroga il database per vedere se esiste un tecnico nella tabella ***TECHNICIAN*** (entità ***Tecnico*** dell'*ER*) con username e password corrispondenti. Il server deve anche verificare la lista dei permessi che ogni tecnico ha nei confronti delle schede (ad esempio un tecnico *Marelli* ha accesso completo a tutte le schede, mentre un tecnico di un cliente ha accesso solo alle proprie).

3. Il database restituisce una tupla se il tecnico è autorizzato, altrimenti restitusce un valore `NULL`.

4. Il server invia una risposta alla scheda *Hachiko*, la quale può quindi chiudere il socket. La scheda è adesso abilitata ad entrare in modalità manutenzione, sospendendo l'attività di lettura e logging dei dati dal *MEC-100*. Eventuali file di *log* precedenti ancora in coda per l'invio vengono inviati in *background* durante l'attività di manutenzione.

CODE 6 - MAINTENANCE (aggiornamento valori di manutenzione)

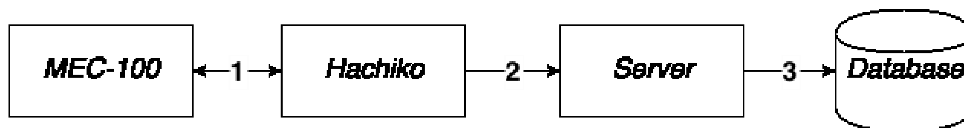


Figura 5.9: Comunicazione client-server relativa al pacchetto *6 - MAINTENANCE*.

Per prima cosa il tecnico deve autenticarsi nel sistema (pacchetto *CODE 5*).

1. La scheda *Hachiko* entra in "modalità manutenzione" e interrompe l'aggregazione dei dati. Tutte le modifiche effettuate da quel momento in poi andranno a formare un apposito file di *log*. Quando il tecnico conferma le modifiche uscendo dalla "modalità manutenzione" il messaggio verrà inviato.

2. Il messaggio comunica username del tecnico, i parametri di configurazione finali e altri file di *log* al server in un *payload* codificato secondo le specifiche definite nella Sezione 3.2.

3. Il server creerà nel database le varie tuple necessarie, in particolare nella tabella ***ASSISTANCE*** (entità ***Assistenza*** dell'*ER*) saranno inseriti lo username del tecnico, l'ID del *MEC-100* e i file di *log*.

Il server dovrà inoltre prevedere un'interfaccia per l'inserimento (usando un computer, non l'*Hachiko*) di eventuali altri file relativi all'assistenza effettuata (principalemente documenti e foto relativi all'intervento).

6. Sviluppi futuri e conclusioni

Il software è stato scritto con l'intento primario di dimostrare la fattibilità del progetto (*proof of concept*) per la presentazione al *Middle East Electricity* di cui si è parlato nell'introduzione (Capitolo 1). Pertanto si è data priorità agli aspetti fondamentali di funzionamento tralasciando alcune funzionalità secondarie in modo da poter rispettare le tempistiche imposte. Di seguito viene presentato un elenco di queste estensioni, le quali verranno implementate in futuro.

6.1 Manutenzione locale e remota

Allo stato attuale la funzionalità più importate da aggiungere al software è la gestione delle manutenzioni, siano esse remote o locali. Infatti il programma mette a disposizione alcune funzioni di basso livello che sono in grado di andare a scrivere un nuovo *set* di parametri di configurazione all'interno del regolatore, tuttavia è necessario implementare una logica di controllo e gestione di più alto livello per gestire le varie problematiche intrinseche nell'aggiornamento.

Per esempio esistono alcuni parametri che sono tra loro correlati, e la modifica di un sottoinsieme di essi deve tenere in considerazione lo stato generale del gruppo. Inoltre, molti dei parametri devono soddisfare un requisito di *range*: i valori devono essere compresi in un certo intervallo. Se questi vincoli non sono rispettati è possibile causare danni al generatore collegato al *MEC-100* con conseguenti potenziali rischi per la sicurezza e per il personale.

Le funzionalità di manutenzione non sono ancora state implementate principalmente per tre motivi:

- la manutenzione *locale* è strettamente legata al modulo di interfaccia grafica, a cui si è scelto di dare una priorità relativamente bassa;
- le modalità per la manutenzione *remota* non sono ancora state definite in modo dettagliato dall'azienda;
- non è ancora stato prodotto un documento tecnico con tutti i vincoli sulle modifiche lecite ai parametri.

6.2 G.U.I. - Interfaccia grafica

Il modulo *G.U.I.* è strettamente legato alla funzionalità di manutenzione locale, ma dovrà anche fornire altre funzionalità, quali la consultazione in tempo reale dei dati di esercizio rilevati dal regolatore o dei grafici sull'andamento di alcune grandezze elettriche.

Non è ancora stato definito quale dovrà essere l'aspetto dell'interfaccia, ma verosimilmente sarà simile a quanto sviluppato per il client *Android*, visto che il pannello *touchscreen* è relativamente piccolo.

Per lo sviluppo di questo modulo sarà inoltre necessario utilizzare una scheda *Hachiko* estesa, ovvero dotata di un modulo di memoria RAM aggiuntivo per far fronte all'inevitabile maggiore consumo di risorse.

Molto probabilmente verranno utilizzate le librerie *QT* [8] per lo sviluppo dell'interfaccia grafica in quanto l'azienda produttrice delle schede *Hachiko* è in grado di fornire supporto diretto per questa soluzione.

6.3 Sicurezza e cifratura messaggi

Un altro aspetto che per ora è stato tralasciato è relativo alla sicurezza delle comunicazioni client $\leftrightarrow$ server. Prima della fase di *deployment* del progetto sarà comunque necessario implementare alcuni meccanismi di cifratura dei pacchetti allo scopo di garantire la *privacy* e, soprattutto, il segreto industriale. Il software è stato pensato fin da subito per la gestione di un sistema di sicurezza, quindi si tratterà solamente di definirne nel dettaglio il meccanismo. La bozza di progetto prevede l'utilizzo di chiavi simmetriche da essere concordate contestualmente all'invio del pacchetto *AUTHENTICATION*.

6.4 Miglioramenti nella gestione dei file su memoria secondaria

L'ultimo aspetto prioritario è lo sviluppo di un meccanismo che renda il software client molto robusto rispetto ad eventuali *black-out* e che migliori la gestione dei file sulla memoria secondaria della scheda. Infatti allo stato attuale, in caso di mancanza di alimentazione, il programma non è in grado di inviare i dati raccolti fino al momento precedente il crash. I dati sono comunque presenti nella memoria locale della scheda, ma l'applicazione non ha alcun riferimento ad essi, infatti al momento tali riferimenti sono contenuti esclusivamente in memoria RAM durante l'esecuzione del programma.

In futuro sarà quindi necessario che anche questi riferimenti vengano salvati su file temporanei in modo che sia possibile recuperarli all'avvio del client. Così facendo non si perderanno dati, se non quelli letti quando si è verificato il *black-out* e non ancora salvati sul file di *log*, ma si tratta di una perdita accettabile.

Inoltre sul lungo periodo l'applicazione tende ad accumulare alcuni file localmente, anche dopo che sono stati inviati al server. Per tale motivo sarà necessario migliorare la gestione globale dei file eliminando quelli non più necessari.

6.5 Altre estensioni meno prioritarie

Oltre agli aspetti sopra menzionati, ritenuti prioritari, ci sono altre funzionalità che sarà necessario implementare in futuro, ma che risultano relativamente meno importanti, almeno allo stato attuale del progetto.

1. **Estensioni ad altri regolatori**: si tratta di estendere il software (sia client che server), per aggiungere il supporto a nuovi regolatori. Tali regolatori sono tuttavia ancora in fase di progettazione, quindi questo aspetto è uno dei meno prioritari in assoluto.

2. **Web server e *browser***: sarà necessario fornire una funzionalità per permenttere la consultazione dei dati dei *MEC-100* anche attraverso un normale *browser web*, oltre che utilizzando l'applicativo *Android*. Anche questo aspetto ha una priorità molto bassa.

3. **Sincronizzazione orologio *Hachiko***: decisamente più importante è la scincronizzazione dell'orologio interno della scheda. Non è necessario che esso sia estremamente preciso, ma l'*Hachiko* resetta il proprio orologio al 1970 (*Unix Epoch*) ad ogni avvio. Molto probabilmente verrà

utilizzato un protocollo come l'*NTP* (*Network Time Protocol*) in modo da sincronizzare l'orlogio tramite la rete, ma si dovrà prevedere anche la possibilità di un'impostazione manuale perché la connessione di rete potrebbe non essere disponibile.

4. **Richiesta *Android* per l'aggiornamento immediato dei dati**: un cliente potrebbe voler accedere agli ultimi dati di esercizio di un determinato *MEC-100*. In questo caso è necessario implementare un nuovo tipo di richiesta, dal server verso il client, che imponga a quest'ultimo di inviare immediatamente i dati raccolti in modo che possano essere presentati tempestivamente al cliente. Ovviamente, per ragioni legate alla latenza delle comunicazioni, non sarà mai possibile garantire al cliente un accesso remoto di tipo *real-time* ai dati.

5. **Ottimizzazione file inviati**: come si è già illustrato nella Sezione 2.5.2 è possibile ottimizzare la dimensioni dei file testuali inviati aumentando l'overhead del server. É possibile ridurre ulteriormente le dimensioni in due modi:

   - utilizzando file *binari* anziché testuali. Questo riduce di molto la dimensione perché per ogni valore da memorizzare sarebbero sufficienti 4 bytes, tuttavia sarebbe necessario aggiungere *overhead* sia lato server che lato client per gestire i file, che risulterebbero più complessi;
   - si può aumentare l'intervallo di tempo tra una lettura e l'altra dei dati di esercizio. Infatti l'azienda non ha l'effettiva necessità di ottenere dati con una grana così fine e sarebbe sufficiente una lettura ogni 5 o 10 secondi, con una conseguente riduzione proporzionale delle dimensioni dei file.

6.6 Conclusioni

L'architettura proposta soddisfa tutte le richieste e le esigenze aziendali emerse durante i vari colloqui. Permette il monitoraggio continuo dei dati di esercizio da remoto, è facilmente estendibile a nuovi regolatori ed offre la possibilità di aggiungere nuove funzionalità in futuro. Eventuali anomalie di funzionamento possono essere rilevate e analizzate senza l'intervento fisico del tecnico sul posto, inoltre ora è possibile avere una lettura tempestiva degli allarmi rilevati, senza dover aspettare la manutenzione programmata. É possibile migliorare ulteriormente questo aspetto tramite una notifica, generata dal server, da inviare all'applicazione *Android* del proprietario del *MEC-100* dove è stato rilevato il problema, ottenendo quindi informazioni quasi in tempo reale.

La possibilità di impostare i parametri di configurazione da remoto ridurrà ulteriormente i costi di assistenza dell'azienda e migliorerà il servizio ai clienti grazie alla possibilità di effettuare interventi quasi contestualmente al verificarsi del problema. Inoltre le manutenzioni saranno migliorate anche dai dati storici accessibili ai tecnici che potranno avere sotto mano uno storico completo dei problemi rilevati per uno specifico *MEC-100*. I dati storici potranno essere utili anche per eventuali analisi sulla difettosità dei componenti o per la progettazione di nuovi regolatori.

A. Codice SQL per la creazione del database

```sql
-- DB creation by using postgreSQL DBMS
DROP SCHEMA IF EXISTS CLOUD CASCADE;
CREATE SCHEMA CLOUD;

-- Domains
CREATE DOMAIN CLOUD.FLOATVALUE AS REAL;

CREATE DOMAIN CLOUD.IP AS VARCHAR
CHECK (VALUE LIKE '%.%.%.%')
DEFAULT '...';

CREATE DOMAIN CLOUD.PORT AS INTEGER
CHECK (VALUE < 65536);

CREATE DOMAIN CLOUD.CODE AS CHAR(6)
CHECK ((VALUE NOT LIKE '%#%') AND (length(value) = 6) AND (VALUE NOT LIKE '% %'));

CREATE DOMAIN CLOUD.KIND AS CHAR(1)
CHECK ((length(value)=1) AND (VALUE LIKE 's' OR VALUE LIKE 'b' OR VALUE LIKE 'f'));

-- Tables
CREATE TABLE CLOUD.REGULATOR(
    id VARCHAR(32) NOT NULL,
    PRIMARY KEY(id)
);

CREATE TABLE CLOUD.CUSTOMER(
    id VARCHAR(32),
    username VARCHAR(32),
    password VARCHAR(32),
    PRIMARY KEY(id),
    UNIQUE (username)
);

CREATE TABLE CLOUD.BOARD(
    id VARCHAR(32),
    idCustomer VARCHAR(32),
    idRegulator VARCHAR(32),
    ip CLOUD.IP,
    port CLOUD.PORT,
    lastConnection TIMESTAMP default CURRENT_TIMESTAMP,
    PRIMARY KEY (id),
    FOREIGN KEY (idCustomer) REFERENCES CLOUD.CUSTOMER(id),
    FOREIGN KEY (idRegulator) REFERENCES CLOUD.REGULATOR(id)
);

CREATE TABLE CLOUD.HACHIKO(
    id VARCHAR(32),
    --other attributes..
    PRIMARY KEY (id)
```

```sql
51  );
52
53  CREATE TABLE CLOUD.HACHIKO_BOARD(
54      id SERIAL,
55      idHachiko VARCHAR(32),
56      idBoard VARCHAR(32),
57      tmstp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
58      PRIMARY KEY (id),
59      FOREIGN KEY (idHachiko) REFERENCES CLOUD.HACHIKO(id),
60      FOREIGN KEY (idBoard) REFERENCES CLOUD.BOARD(id)
61  );
62
63  --TABLE for regulator's parameters
64  CREATE TABLE CLOUD.PARAMETERREGULATOR(
65      id CLOUD.CODE,
66      name VARCHAR(72),
67      UM VARCHAR(10) DEFAULT '',
68      min CLOUD.FLOATVALUE DEFAULT 0,
69      max CLOUD.FLOATVALUE DEFAULT 0,
70      PRIMARY KEY(id)
71  );
72
73  --TABLE used to associate a single parameter to a regulator type:
74  --this creates the list of parameters of a specific MEC-100.
75  CREATE TABLE CLOUD.REGULATOR_PARAMETER(
76      idRegulator VARCHAR(32),
77      idParameterRegulator CLOUD.CODE,
78      PRIMARY KEY(idRegulator, idParameterRegulator),
79      FOREIGN KEY(idRegulator) REFERENCES CLOUD.REGULATOR(id),
80      FOREIGN KEY(idParameterRegulator) REFERENCES CLOUD.PARAMETERREGULATOR(id) ON UPDATE CASCADE
81  );
82
83  CREATE TABLE CLOUD.LOG(
84      id SERIAL,
85      data BYTEA,
86      PRIMARY KEY(id)
87  );
88
89  CREATE TABLE CLOUD.AGGREGATEVALUE(
90      id SERIAL,
91      val CLOUD.FLOATVALUE NOT NULL,
92      idBoard VARCHAR(32),
93      idParameterRegulator CLOUD.CODE,
94      idLog INTEGER,
95      tmstp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
96      PRIMARY KEY (id),
97      FOREIGN KEY(idBoard) REFERENCES CLOUD.BOARD(id),
98      FOREIGN KEY(idParameterRegulator) REFERENCES CLOUD.PARAMETERREGULATOR(id) ON UPDATE CASCADE,
99      FOREIGN KEY(idLog) REFERENCES CLOUD.LOG(id)
100 );
101
102 CREATE TABLE CLOUD.TYPEGENERATOR(
103     id VARCHAR(32),
104     code VARCHAR(32),
105     type VARCHAR(32),
106     PRIMARY KEY(id)
107 );
108
109 CREATE TABLE CLOUD.GENERATOR(
110     id VARCHAR(32),
111     idCustomer VARCHAR(32),
112     idTypeGenerator VARCHAR(32),
113     PRIMARY KEY(id),
114     FOREIGN KEY (idCustomer) REFERENCES CLOUD.CUSTOMER(id),
```

```
115     FOREIGN KEY(idTypeGenerator) REFERENCES CLOUD.TYPEGENERATOR(id)
116 );
117
118 CREATE TABLE CLOUD.PARAMETERGENERATOR(
119     id CLOUD.CODE,
120     name VARCHAR(50),
121     UM VARCHAR(10) DEFAULT '',
122     min CLOUD.FLOATVALUE DEFAULT 0,
123     max CLOUD.FLOATVALUE DEFAULT 0,
124     PRIMARY KEY(id)
125 );
126
127 CREATE TABLE CLOUD.PARAMETERGENERATOR_TYPEGENERATOR(
128     idParameterGenerator CLOUD.CODE,
129     idTypeGenerator VARCHAR(32),
130     PRIMARY KEY(idParameterGenerator,idTypeGenerator),
131     FOREIGN KEY(idParameterGenerator) REFERENCES CLOUD.PARAMETERGENERATOR(id) ON UPDATE CASCADE,
132     FOREIGN KEY(idTypeGenerator) REFERENCES CLOUD.TYPEGENERATOR(id) ON UPDATE CASCADE
133 );
134
135 CREATE TABLE CLOUD.BOARD_GENERATOR(
136     idBoard VARCHAR(32),
137     idGenerator VARCHAR(32),
138     PRIMARY KEY(idBoard, idGenerator),
139     FOREIGN KEY(idBoard) REFERENCES CLOUD.BOARD(id),
140     FOREIGN KEY(idGenerator) REFERENCES CLOUD.GENERATOR(id)
141 );
142
143 CREATE TABLE CLOUD.VALUEGENERATOR(
144     id SERIAL,
145     valueFloat CLOUD.FLOATVALUE,
146     valueString VARCHAR(32),
147     idParameterGenerator CLOUD.CODE,
148     idGenerator VARCHAR(32),
149     PRIMARY KEY(id),
150     FOREIGN KEY(idParameterGenerator) REFERENCES CLOUD.PARAMETERGENERATOR(id) ON UPDATE CASCADE,
151     FOREIGN KEY(idGenerator) REFERENCES CLOUD.GENERATOR(id)
152 );
153
154 CREATE TABLE CLOUD.TECHNICIAN(
155     username VARCHAR(32),
156     password VARCHAR(32),
157     --other attributes...
158     PRIMARY KEY(username)
159 );
160
161 CREATE TABLE CLOUD.PERMISSION(
162     usernameTechnician VARCHAR(32),
163     idGenerator VARCHAR(32),
164     PRIMARY KEY(usernameTechnician, idGenerator),
165     FOREIGN KEY(usernameTechnician) REFERENCES CLOUD.TECHNICIAN(username) ON UPDATE CASCADE,
166     FOREIGN KEY(idGenerator) REFERENCES CLOUD.GENERATOR(id)
167 );
168
169 CREATE TABLE CLOUD.ASSISTANCE(
170     id SERIAL,
171     description TEXT,
172     document BYTEA,
173     logFile BYTEA,
174     valueLogFile BYTEA,
175     tmstp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
176     usernameTechnician VARCHAR(32),
177     idBoard VARCHAR(32),
178     PRIMARY KEY(id),
```

```
179     FOREIGN KEY(usernameTechnician) REFERENCES CLOUD.TECHNICIAN(username) ON UPDATE CASCADE,
180     FOREIGN KEY(idBoard) REFERENCES CLOUD.BOARD(id)
181 );
182 
183 CREATE TABLE CLOUD.CONFIGURATIONVALUE(
184     id SERIAL,
185     valueKind CLOUD.KIND,
186     valueFloat CLOUD.FLOATVALUE,
187     valueString VARCHAR(32),
188     valueBoolean BOOLEAN,
189     idBoard VARCHAR(32),
190     idAssistance INTEGER,
191     idParameterRegulator CLOUD.CODE,
192     tmstp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
193     PRIMARY KEY (id),
194     FOREIGN KEY(idBoard) REFERENCES CLOUD.BOARD(id),
195     FOREIGN KEY(idAssistance) REFERENCES CLOUD.ASSISTANCE(id),
196     FOREIGN KEY(idParameterRegulator) REFERENCES CLOUD.PARAMETERREGULATOR(id) ON UPDATE CASCADE
197 );
198 
199 CREATE TABLE CLOUD.ALARM(
200     id CLOUD.CODE,
201     name VARCHAR(50),
202     description VARCHAR(50),
203     PRIMARY KEY (id)
204 );
205 
206 CREATE TABLE CLOUD.ALARMVALUE(
207     id SERIAL,
208     val CLOUD.FLOATVALUE,
209     tmstp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
210     idBoard VARCHAR(32),
211     idParameterRegulator CLOUD.CODE,
212     PRIMARY KEY(id),
213     FOREIGN KEY(idBoard) REFERENCES CLOUD.BOARD(id),
214     FOREIGN KEY(idParameterRegulator) REFERENCES CLOUD.PARAMETERREGULATOR(id)
215 );
216 
217 CREATE TABLE CLOUD.ALARMVALUE_ALARM(
218     idAlarmValue INTEGER,
219     idAlarm CLOUD.CODE,
220     PRIMARY KEY (idAlarmValue, idAlarm),
221     FOREIGN KEY(idAlarmValue) REFERENCES CLOUD.ALARMVALUE(id),
222     FOREIGN KEY(idAlarm) REFERENCES CLOUD.ALARM(id)
223 );
224 
225 CREATE TABLE CLOUD.REGULATORALARM(
226     idRegulator VARCHAR(32),
227     idAlarm CLOUD.CODE,
228     FOREIGN KEY(idRegulator) REFERENCES CLOUD.REGULATOR(id),
229     FOREIGN KEY(idAlarm) REFERENCES CLOUD.ALARM(id)
230 );
```

Codice A.1: Codice SQL per la creazione del database

Bibliografia

[1] Marelli Motori. *Marelli Motori Company Profile*. URL: http://www.marellimotori.com/page.asp?*p=93 (visitato il 25/11/2014).

[2] Stefano Massignani. *Marelli Energy Controller MEC-100 - Manuale dell’utente*. 2009.

[3] ArchiTech. *Hachiko Board*. URL: http://www.architechboards.org/product/hachiko-board (visitato il 25/11/2014).

[4] *Yocto Project*. URL: http://www.yoctoproject.org (visitato il 25/11/2014).

[5] Shamkant B. Navathe Ramez Elmasri. *Sistemi di basi di dati - Fondamenti*. Trad. da Stefano Montanelli. 5ª ed. Feb. 2007, pp. 382–398. ISBN: 978-88-7192-310-9.

[6] URL: http://www.postgresql.org (visitato il 25/11/2014).

[7] URL: http://www.boost.org (visitato il 25/11/2014).

[8] URL: http://www.qt-project.org (visitato il 25/11/2014).